Abstraction and Specification in Program Development

Barbara Liskov
John Guttag

Contents

1 Introduction **9**
- 1.1 Decomposition and Abstraction . . . 10
- 1.2 Abstraction . . . 11
  - 1.2.1 Abstraction by Parameterization . . . 14
  - 1.2.2 Abstraction by Specification . . . 15
  - 1.2.3 Kinds of Abstractions . . . 17

2 Using C++ **21**
- 2.1 Program Structure . . . 21
  - 2.1.1 Header Files . . . 21
  - 2.1.2 Source Files . . . 23
- 2.2 Code Structure . . . 25
  - 2.2.1 Expressions . . . 28
  - 2.2.2 Functions . . . 29
- 2.3 The 6.170 Environment . . . 30
  - 2.3.1 Checks . . . 30
- 2.4 Strings and Arrays . . . 30
  - 2.4.1 Strings . . . 30
  - 2.4.2 Arrays . . . 34
- 2.5 Objects and Variables . . . 36
  - 2.5.1 Mutability . . . 39
- 2.6 Types and Type Checking . . . 40
  - 2.6.1 Type Hierarchy . . . 40
  - 2.6.2 Conversions . . . 41
  - 2.6.3 Defining New Types . . . 42
- 2.7 Function Calls . . . 42
  - 2.7.1 Semantics of Function Call . . . 42
  - 2.7.2 Call-by-value . . . 43
  - 2.7.3 Call-by-reference . . . 44
- 2.8 I/O . . . 46
  - 2.8.1 Examples . . . 46
  - 2.8.2 Output . . . 47

2.8.3 Errors and End Of File 49
2.8.4 Input 49
2.8.5 More Examples 51
2.9 Files 52
2.10 Examples 54

3 Procedural Abstraction **59**
3.1 The Benefits of Abstraction 59
3.2 Specifications 61
3.3 Specifications of Procedural Abstractions 62
3.3.1 Call-by-reference 64
3.3.2 Using const 66
3.4 Implementing Procedures 67
3.5 Designing Procedural Abstractions 69
3.6 Summary 73

4 Data Abstraction **75**
4.1 Specifications for Data Abstractions 77
4.1.1 Specification of intset 78
4.1.2 Excluded Operations 81
4.1.3 The poly abstraction 82
4.2 Using Data Abstractions 84
4.3 Implementing Data Abstractions 85
4.3.1 Implementation of intset 86
4.3.2 Implementation of poly 89
4.4 Related Routines 90
4.5 Aids to Understanding Implementations 94
4.5.1 The Abstraction Function 94
4.5.2 The Representation Invariant 96
4.5.3 Checking the Rep Invariant 100
4.5.4 Preserving the Rep Invariant 101
4.5.5 Exposing the rep 102
4.6 Lists 103
4.7 Discussion 107
4.7.1 Mutability 107
4.7.2 Benevolent Side Effects 107
4.7.3 Operation Categories 111
4.7.4 Adequacy 112
4.7.5 Reasoning about Data Types 112
4.8 Summary 114

5 Exceptions **117**
5.1 Specifications . . . 119
5.2 The C++ Exception Mechanism . . . 121
5.2.1 Signaling Exceptions . . . 122
5.2.2 Handling Exceptions . . . 122
5.2.3 Creating Exception Classes . . . 124
5.2.4 Catching Local Exceptions . . . 126
5.2.5 Unhandled Exceptions . . . 126
5.3 Using Exceptions in Programs . . . 127
5.4 Design Issues . . . 130
5.5 Very Robust Programs . . . 132
5.6 Summary . . . 133

6 More General Abstractions **135**
6.1 The Template Mechanism . . . 136
6.2 Polymorphic Procedures . . . 138
6.2.1 Implementing Polymorphic Procedures . . . 139
6.2.2 Using Polymorphic Procedures . . . 140
6.3 The Meaning of Equality . . . 142
6.4 Polymorphic Data Abstractions . . . 143
6.4.1 Using Polymorphic Data Abstractions . . . 143
6.4.2 Implementing Polymorphic Data Abstractions . . . 145
6.5 More on Equality . . . 146
6.6 Ordered Lists . . . 149
6.7 Summary . . . 151

7 Iteration Abstraction **155**
7.1 Iteration in C++ . . . 158
7.2 Specifying Iterators . . . 159
7.3 Using Iterators . . . 160
7.4 Predefined Iterators . . . 161
7.5 Implementing Iterators . . . 161
7.6 Rep Invariants and Abstraction Functions for Generators . . 163
7.7 Another Example . . . 165
7.8 Design Issues . . . 166
7.9 Summary . . . 168

8 Testing and Debugging **171**
8.1 Testing . . . 172
8.1.1 Black-Box Testing . . . 173
8.1.2 Testing Based on the Program Text . . . 175
8.1.3 An Example . . . 178
8.1.4 Testing Iterators . . . 179
8.1.5 Testing Data Types . . . 180

8.2 Unit and Integration Testing 182
8.3 Tools for Testing 185
8.4 Debugging 187
8.5 Defensive Programming 193
8.6 Summary 195

9 More on Specifications 197
9.1 Specifications and Specificand Sets 197
9.2 Some Criteria for Specifications 198
9.2.1 Restrictiveness 198
9.2.2 Generality 200
9.2.3 Clarity 201
9.3 Why Specifications? 204
9.4 Summary 206

10 Hierarchy and Inheritance 209
10.1 Type Hierarchy and Assignment 210
10.2 Defining a Type Hierarchy 212
10.2.1 Defining Subtypes 214
10.2.2 Implementing a Subtype 215
10.2.3 Casting Down 219
10.2.4 Polymorphic Hierarchy 220
10.3 Exception Types 221
10.4 Abstract Classes 223
10.5 The Meaning of Subtypes 226
10.6 Multiple Implementations 230
10.6.1 Constructing New Objects 233
10.7 Generators 235
10.8 Subtype vs. Parametric Polymorphism 235
10.9 Private and Protected Subclasses 236
10.10Multiple Inheritance 238
10.11Summary 239

11 A Preamble to Program Design 241
11.1 The Software Life Cycle 241
11.2 Requirements Analysis Overview 243
11.3 A Sample Problem 246
11.4 Summary 252

12 Design 255
12.1 An Overview of the Design Process 255
12.2 The Design Notebook 257
12.2.1 The Introductory Section 257
12.2.2 The Abstraction Sections 259

12.3 Problem Description 261
12.4 Getting Started on a Design 263
12.5 Discussion of the Method 266
12.6 Continuing the Design 268
12.7 The Doc Abstraction 272
12.8 The Line Abstraction 276
12.9 Review and Discussion 277
12.10 Summary 281

13 Between Design and Implementation 283
13.1 Evaluating a Design 283
13.1.1 Correctness and Efficiency 284
13.1.2 Structure 287
13.2 Ordering the Program Development Process 292
13.3 The Relationship of Abstraction to Efficiency 296
13.4 Summary 298

14 A Quick Look at Program Verification 301
14.1 Reasoning about Straight-Line Code 302
14.2 Reasoning about Programs with Multiple Paths 304
14.3 Reasoning about Procedures 310
14.3.1 Reasoning about Procedures with No Side Effects . 310
14.3.2 Reasoning about Code that Uses Arrays 316
14.4 A Few Remarks about Formal Reasoning 321
14.5 Summary 322

15 Extending our Approach 325
15.1 Stack vs. Heap 325
15.1.1 Defining Data Abstractions 328
15.1.2 Stack-or-Heap Abstractions 329
15.2 Explicit Memory Management 334
15.2.1 Explicit Deallocation 336
15.2.2 Implementing Copying 337
15.3 Summary 338

16 Using Other Languages 341
16.1 Procedural Abstractions in C 341
16.2 Data Abstraction 343
16.2.1 Issues 343
16.2.2 Stack vs. Heap 344
16.2.3 Heap Abstractions 345
16.2.4 The Stack Approach 348
16.3 Parameterized Abstractions 350
16.4 Summary 351

Chapter 1

Introduction

This book will develop a methodology for program construction. Our goal is to help programmers construct programs of high quality—programs that are reliable and reasonably easy to understand, modify, and maintain.

A very small program, consisting of no more than a few hundred lines, can be implemented as a single monolithic unit. As the size of the program increases, however, such a monolithic structure is no longer reasonable because the code becomes difficult to understand. Instead the program must be decomposed into a number of independent small programs, called *modules*, that together provide the desired function. We shall focus on this decomposition process: how to decompose large programming problems into small ones, what kinds of modules are most useful in this process, and what techniques increase the likelihood that modules can be combined to solve the original problem.

Doing decomposition properly becomes more and more important as the size of the program increases, for a number of reasons. First, many people must be involved in the construction of a large program. If just a few people are working on a program, they will naturally interact regularly. Such contact reduces the possibility of misunderstandings about who is doing what, and lessens the seriousness of the consequences should such misunderstandings occur. If many people work on a project, regular communication becomes impossible because it would consume too much time. Instead the program must be decomposed into pieces that the individuals can work on independently with a minimum of contact.

The useful life of a program (its *production* phase) begins when it is delivered to the customer. Work on the program is not over at this point, however. There are likely to be residual errors in the code that will need attention, and program modifications will often be required to upgrade the program's serviceability or to provide services better matched to the user's needs. This activity of program *modification* and *maintenance* is likely to consume more than half of the total effort put into the project.

For modification and maintenance, it is rarely practical to start from scratch and reimplement the entire program. Instead, one must retrofit modifications within the existing structure, and it is therefore important that the structure accommodate change. In particular, the pieces of the program must be independent, so a change to one piece can be made without requiring changes to all pieces.

Finally, most programs have a long lifetime. People will often have to deal with programs long after they have first worked on them. Moreover, there is likely to be substantial turnover of personnel over the life of any project, and program modification and maintenance are typically done by people other than the original implementors. All of these factors suggest a requirement that programs be structured in such a way that they can be understood easily.

In the methodology we shall describe in this book, programs will be developed by means of problem decomposition based on a recognition of useful abstractions. *Decomposition* and *abstraction*, the two key concepts in this book, form our next subject.

1.1 Decomposition and Abstraction

The basic paradigm for tackling any large problem is clear—we must "divide and rule." Unfortunately, merely deciding to follow Machiavelli's dictum still leaves us a long way from solving the problem at hand. Exactly how we choose to divide the problem is of overriding importance.

Our goal in decomposing a program is to create modules that are themselves small programs that interact with one another in simple, well-defined ways. If we achieve this goal, different people will be able to to work on different modules independently, without needing much communication among themselves, and yet the modules will work together. In addition, during program modification and maintenance it will be possible to modify some of the modules without affecting all of the others.

When we decompose a problem, we factor it into separable subproblems in such a way that

1. each subproblem is at the same level of detail;

2. each subproblem can be solved independently; and

3. the solutions to the subproblems can be combined to solve the original problem.

Sorting using merge sort is an elegant example of problem solving by decomposition. It breaks the problem of sorting a list of arbitrary size into the two simpler problems of sorting a list of size two and merging two sorted lists of arbitrary size.

Decomposition is a time-honored and useful technique in many disciplines. From Babbage's day onward people have recognized the utility of such things as macros and subroutines as decomposition devices for programmers. It is important to recognize, however, that decomposition is not a panacea, and that when used improperly it can have a harmful effect. Furthermore, large or poorly understood problems are difficult to decompose properly. The most common problem is creating individual components that solve the stated subproblems but do not combine to solve the original problem. This is one of the reasons why system integration is often difficult.

For example, imagine creating a play by assembling a group of writers, giving each a list of characters and a general plot outline, and asking each of them to write a single character's lines. The authors might accomplish their individual tasks admirably, but it is highly unlikely that their combined efforts will be an admirable play. It might be artistic, but it would lack any sort of coherence or sense. Individually acceptable solutions simply cannot be expected to combine properly if the original task has been divided in a counterproductive way.

Abstraction is a way to do decomposition productively by changing the level of detail to be considered. When we abstract from a problem we agree to ignore certain details in an effort to convert the original problem to a simpler one. We might, for example, abstract from the problem of writing a play to the problem of deciding how many acts it should have, or what its plot will be, or even the sense (but not the wording) of individual pieces of dialogue. After this has been done, the original problem (of writing all of the dialogue) remains, but it has been considerably simplified—perhaps even to the point where it could be turned over to another or even several others. (Dumas, père, churned out novels in this way.)

The paradigm of abstracting and then decomposing is typical of the program design process: Decomposition is used to break software into components that can be combined to solve the original problem; abstractions assist in making a good choice of components. We alternate between the two processes until we have reduced the original problem to a set of problems we already know how to solve.

1.2 Abstraction

The process of abstraction can be seen as an application of a many-to-one mapping. It allows us to forget information and consequently to treat things that are different as if they were the same. We do this in the hope

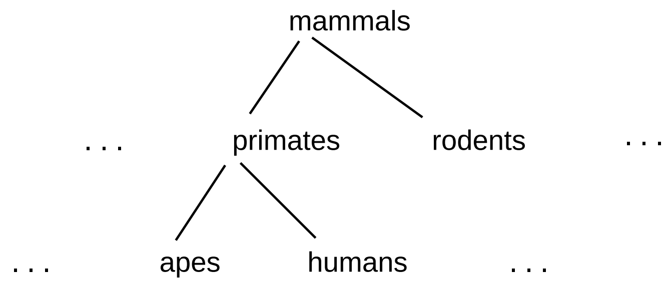

Figure 1.1 An abstraction hierarchy.

of simplifying our analysis by separating attributes that are relevant from those that are not. It is crucial to remember, however, that relevance often depends upon context. In the context of an elementary school classroom we learn to abstract both $(8/3) * 3$ and $5 + 3$ to the concept we represent by the numeral 8. Much later we learn, often under unpleasant circumstances, that on many computing machines this abstraction can get us into a world of trouble.

For example, consider the structure shown in figure 1.1. Here the concept is "mammal." All mammals share certain characteristics, such as the fact that females produce milk. At this level of abstraction, we focus on these common characteristics and ignore the differences between the various types of mammals.

At a lower level of abstraction, we might be interested in particular instances of mammals. However, even here we can abstract by considering not individuals, or even species, but groups of related species. At this level we would have groupings such as primates or rodents. Here again we are interested in common characteristics, such as the fact that all primates bear live young, rather than the differences between, say, humans and chimpanzees. Such differences will be relevant at a still lower level of abstraction.

The abstraction hierarchy of figure 1.1 comes from the field of zoology, but it might well appear in a program that implemented some zoological application. A more specifically computer-oriented example that is useful in many programs is the concept of a "file." Files abstract from raw storage and provide long-term, online storage of named entities. Operating systems differ in their realizations of files; for example, the structure of the file names differs from system to system, as does the way in which the files are stored on secondary storage devices.

In this book we are interested in abstraction as it is used in programs in general. The most significant development to date in this area is the development of high-level languages. By dealing directly with the constructs of a high-level language rather than with the many possible sequences of

```
// search upwards
      found = false;
      for (int i = 0; i < a→length(); i++)
        if (a→fetch(i) == e) {
            z = i;
            found = true; }

// search downwards
      found = false;
      for (int i = a→length()−1; i ≥ 0; i−−)
        if (a→fetch(i) == e) {
            z = i;
            found = true; }
```

Figure 1.2 Two Program Fragments.

machine instructions into which they can be translated, the programmer achieves a significant simplification.

In recent years, however, programmers have become dissatisfied with the level of abstraction generally achieved even in high-level language programs. Consider, for example, the program fragments in figure 1.2. At the level of abstraction defined by the programming language these fragments are clearly different: If there is an occurrence of e in a, one fragment finds the index of the first occurrence, and the other the indexof the last. If e does not occur in a, one sets z to *a->length()* and the other to *-1*. It is not improbable, however, that both were written to accomplish the same goal: to set *found* to false if there is no occurrence of e in a, and otherwise to set *found* to true and z to the index of some occurrence of e in a. If this is what we want, these program fragments by themselves clearly do not achieve the desired level of abstraction.

One approach to dealing with this problem lies in the invention of "very-high-level languages," built around some fixed set of relatively general data structures and a powerful set of primitives that can be used to manipulate them. For example, suppose a language provided *is_in* and *index_of* as primitive operations on arrays. Then we could accomplish the task outlined above simply by writing

```
found = is_in(a,e);
if (found)
   z = index_of(a,e);
```

The flaw in this approach is that it presumes that the designer of the programming language will build into the language most of the abstractions that users of the language will want. Such foresight is not given to many;

and even if it were, a language containing so many built-in abstractions might well be so unwieldy as to be unusable.

A preferable alternative is to design into the language mechanisms that allow programmers to construct their own abstractions as they need them. The most common such mechanism is the use of *procedures*. By separating procedure definition and invocation, a programming language makes two important methods of abstraction possible: *abstraction by parameterization* and *abstraction by specification*.

1.2.1 Abstraction by Parameterization

Abstraction by parameterization allows us, through the introduction of parameters, to represent a potentially infinite set of different computations with a single program text that is an abstraction of all of them. Consider the program text

$$\mathrm{x} * \mathrm{x} + \mathrm{y} * \mathrm{y}$$

This describes a computation that adds the square of the value stored in a particular variable x to the square of the value stored in another particular variable y. The *lambda expression*

$$\lambda \mathrm{x}, \mathrm{y} : \mathrm{int}.(\mathrm{x} * \mathrm{x} + \mathrm{y} * \mathrm{y})$$

on the other hand, describes the set of computations that square the value stored in some integer variable, which we shall temporarily refer to as x, and add to it the square of the value stored in another integer variable, which we shall temporarily call y. In such a lambda expression we refer to x and y as the *formal parameters* and $x * x + y * y$ as the *body* of the expression. We invoke a computation by binding the formal parameters to arguments and then evaluating the body. For example,

$$\lambda \mathrm{x}, \mathrm{y} : \mathrm{int}.(\mathrm{x} * \mathrm{x} + \mathrm{y} * \mathrm{y})(\mathrm{w}, \mathrm{z})$$

is identical in meaning to

$$\mathrm{w} * \mathrm{w} + \mathrm{z} * \mathrm{z}$$

In more familiar notation, we might denote the above lambda expression by

```
int squares (int x, y) {
   return x * x + y * y }
```

and the binding of actual to formal parameters and evaluation of the body by the procedure call squares(w, z);

Programmers often use abstraction by parameterization without even noticing that they are doing so. For example, suppose we need a procedure that sorts an array of integers *a*. At some time in the future we shall probably have to sort some other array, perhaps even somewhere else in this same program. It is highly unlikely, however, that every array we need to sort will be named *a*; we therefore invoke abstraction by parameterization to generalize the procedure and thus make it more useful.

Abstraction by parameterization is an important means of achieving generality in programs. A *sort* routine that works on any array of integers is much more generally useful than one that works only on a particular array of integers. By further abstraction we can achieve even more generality. For example, we might define a *sort* abstraction that works on arrays of reals as well as arrays of integers, or even one that works on arraylike structures in general.

Abstraction by parameterization is an extremely powerful mechanism. Not only does it allow us to describe a large (even infinite) number of computations relatively simply, but it is easily and efficiently realizable in programming languages. Nonetheless, it is not a sufficiently powerful mechanism to describe conveniently and fully the abstraction that the careful use of procedures can provide.

1.2.2 Abstraction by Specification

Abstraction by specification allows us to abstract from the computation (or computations) described by the body of a procedure to the end that procedure was designed to accomplish. We do this by associating with each procedure a *specification* of its intended effect and then considering the meaning of a procedure call to be based on this specification rather than on the procedure's body.

We are making use of abstraction by specification whenever we associate with a procedure a comment that is sufficiently informative as to allow others to use that procedure without looking at its body. A good way to write such comments is to use pairs of *assertions*. The *requires assertion* (or *pre-condition*) of a procedure specifies something that is assumed to be true on entry to the procedure. In practice what is most often asserted is a set of conditions sufficient to ensure the proper operation of the procedure. (This is often simply the vacuous assertion "true.") The *effects assertion* (or *post-condition*) specifies something that is supposed to be true at the completion of any invocation of the procedure for which the pre-condition was satisfied.

Consider, for example, the *sqrt* procedure in figure 1.3. Because a specification is provided, we can ignore the body of the procedure and take the

```
float sqrt (float coef) {
  % requires: coef > 0
  % effects: returns an approximation to the square root off _coef_
    float ans = coef/2.0;
    int i = 1;
    while (i < 7) {
      ans = ans − ((ans ∗ ans − coef)/(2.0∗ans));
      i = i + 1; }
    return ans;
  } // end sqrt
```

Figure 1.3 The *sqrt* procedure.

meaning of the procedure call $y = sqrt(x)$ to be "If x is greater than 0 when the procedure is invoked, then after the execution of the procedure, y is an approximation to the square root of x." Notice that the requires and effects assertions permit us to say nothing about the value of y if x is not greater than 0.

In using a specification to reason about the meaning of a procedure call, we follow two distinct rules:

1. After the execution of the procedure we can assume that the post-condition holds.

2. We can assume *only* those properties that can be inferred from the post-condition.

The two rules mirror the two benefits of abstraction by specification. The first asserts that users of the procedure need not bother looking at the body of the procedure in order to use it. They are thus spared the effort of first understanding the details of the computations described by the body and then abstracting from these details to discover that the procedure really does compute an approximation to the square root of its argument. For complicated procedures, or even simple ones using unfamiliar algorithms, this is a nontrivial benefit.

The second rule makes it clear that we are indeed abstracting from the procedure body, that is, forgetting some supposedly irrelevant information. This insistence on forgetting information is what distinguishes abstraction from decomposition. By examining the body of *sqrt*, users of the procedure could gain a considerable amount of information that cannot be gleaned from the post-condition, for example, that $sqrt(4)$ will return $+2$. In the specification, however, we are saying that this information about the returned result is to be ignored. We are thus saying that the procedure *sqrt* is an abstraction representing the set of all computations that return "an approximation to the square root of x."

In this book, abstraction by specification will be the major method used in program construction. Abstraction by parameterization will be taken almost for granted; abstractions will have parameters as a matter of course.

1.2.3 Kinds of Abstractions

Abstraction by parameterization and by specification are powerful methods for program construction. They enable us to define three different kinds of abstractions: procedural abstraction, data abstraction, and iteration abstraction. In general, each procedural, data, and iteration abstraction will incorporate both methods within it.

For example, *sqrt* is like an operation: It abstracts a single event or task. We shall refer to abstractions that are operationlike as *procedural abstractions*. Note that *sqrt* incorporates both abstraction by parameterization and abstraction by specification.

Procedural abstraction is a powerful tool. It allows us to extend the virtual machine defined by a programming language by adding a new operation. This kind of extension is most useful when we are dealing with problems that are conveniently decomposable into independent functional units. Often, however, it is more fruitful to think of adding new kinds of data objects to the virtual machine.

The behavior of the data objects is expressed most naturally in terms of a set of operations that are meaningful for those objects. This set includes operations to create objects, to obtain information from them, and possibly to modify them. For example, *push* and *pop* are among the meaningful operations for stacks, while integers need the usual arithmetic operations. Thus a *data abstraction* (or *data type*) consists of a set of objects and a set of operations characterizing the behavior of the objects.

As an example, consider *multi_sets*. Multi_sets are like ordinary sets except that elements can occur more than once in a multi_set. Multi_set operations include *empty*, *insert*, *delete*, *number_of*, and *size*. These operations create an empty multi_set, add and delete elements from a multi_set, tell how many times a particular element occurs in a multi_set, and tell how many elements are in a multi_set, respectively. The operations might be implemented within the runtime environment of the programming language by calls to various procedures. Programmers using multi_sets, however, need not worry about how these procedures are implemented. To them *empty*, *insert*, *delete*, *number_of*, and *size* are abstractions defined by such statements as

The *size* of the multi_set $insert(s,\ e)$ is equal to $size(s) + 1$.

For all e, the *number_of* times e occurs in the multi_set *empty*( ) is 0.

The key thing to notice is that each of these statements deals with more than one operation. We do not present independent definitions of each operation, but rather define them by showing how they relate to one another. The emphasis on the relationships among operations is what makes a data abstraction something more than just a set of procedures. The importance of this distinction is discussed throughout this book.

In addition to procedural and data abstraction, we shall deal in this book with *iteration abstraction*. Iteration abstraction is used to avoid having to say more than is relevant about the flow of control in a loop. A typical iteration abstraction might allow us to iterate over all the elements of a multi_set without constraining the order in which the elements are to be processed.

The remainder of this book is concerned with how to do program decomposition based on abstraction. Our emphasis will be on data abstraction. We believe that while procedural and iteration abstraction have valuable roles to play, it is data abstraction that most often provides the primary organizational tool in the programming process.

The next few chapters are concerned with the three kinds of abstractions—what they are, how to specify their behavior, and how to implement them in C++. We shall begin by providing a brief discussion of C++; the discussion defines some conventions on the use of C++ that we will adhere to for most of the book. The next several chapters will explain the various kinds of abstractions, using C++ as the implementation language.

The latter portion of the book focuses on the use of abstractions in program construction. We discuss the phases of program construction, how to do program design, and how to carry on into implementation. The book concludes with a discussion of a number of practical issues that arise in using the methodology in constructing real programs.

Further Reading

Boehm, Barry W., 1976. Software engineering. *IEEE Transactions on Computers* C-25(12): 1226–1241.

Brooks, Frederick P., Jr., 1975. *The Mythical Man-Month*. Reading, Mass.: Addison-Wesley Publishing Co.

Buxton, John M., Peter Naur, and Brian Randell (eds.), 1976. Software engineering concepts and techniques. In *Proceedings of the NATO Conference* (New York: Petrocelli/Charter).

Gries, David (ed.), 1978. Part 1: viewpoints on programming. In *Programming Methodology, A Collection of Articles by members of IFIP WG2.3* (New York: Springer-Verlag), pp. 7–74.

Exercises

1.1 Describe an abstraction hierarchy with which you are familiar.

1.2 Select a procedure that you have written or used and discuss how it supports abstraction by specification and by parameterization.

Chapter 2

Using C++

This chapter provides some basic information about C++ and how we will use it. The chapter is not intended to teach you C++, but it will help you get started. When you have questions about C++, you should make use of the C++ text.

In 6.170 we will be using only a subset of C++. **You are expected to stay within this subset.**

2.1 Program Structure

A C++ program is made up of modules, each of which defines a group of related abstractions. For example, a module might define a group of routines for a mathematics package. Or a module might define a data abstraction and some related routines. In this section we use as an example a module that contains a `stack` data abstraction and a related routine `pop_many`.

Modules are defined by header files and implemented in source files.

2.1.1 Header Files

A module is defined by a *header* or *include* file, which describes the interfaces of the abstractions in the module. The file name of a header file ends in the suffix `.h` and is usually selected to express some idea of the purpose of its module, e.g., the header file for a stack abstraction might be named `stack.h`.

The information for each abstraction in a header file corresponds roughly to a specification for the abstraction; specifications are discussed in later chapters. For example, here is part of `stack.h`, the header file for the stack abstraction module:

```
#ifndef _STACK_H
#define _STACK_H

//define the stack abstraction
class stack {
  //details of how to define classes will be given later
};

//define related routines
void pop_many(stack* s, int n);
// requires:  s has at least n elements
// modifies:  s
// effects:   Pops the top n elements of s

#endif // _STACK_H
```

The first two lines and the last line are pre-processor directives that allow `stack.h` to be safely included multiple times. In general we want to allow header files to be included multiple times, since they are included in every program that uses their module as well as in any file that implements the module. All header files should be bracketed with similar pre-processor directives.

A header file defines a data abstraction by means of a `class`; details of how to define classes will be given in Chapter 4.

2.1.2 Source Files

A module is implemented by providing a *source* or *code* file. The name of such a file ends in the suffix `.cpp` (or by a file whose name ends in `.t` in the case of a parameterized abstraction).

A source file always *includes* the header file for its module (except in the case of a `.t` file; these will be discussed later). For example, here is a sketch of a source file for the stack module:

```
#include "stack.h"

// Implement stack operations
stack::stack() {
    ...
}
...
// Implement related routines
void pop_many(stack* s, int n) {
    ...
```

```
}
```

Including the header file allows the C++ compiler to ensure that the right interfaces are being implemented. For example, if the implementation of `pop_many` had the header:

```
int pop_many(stack* s, int n)
```

there would be a compile-time error since the result type given here differs from that in the `.h` file.

Header files are also included in any files that use their abstractions. There are two kinds of usage: a module might use routines defined by the header file, or it might just refer to a type defined by the header file but not use any procedures defined by the header file. Any module that uses a routine must include the header file that describes the routine being called. For example, the following `.cpp` file calls the `stack` routine `pop_many` and therefore it must include `stack.h`.

```
#include "stack.h"
//include header files for any other modules used by this .cpp file here
...
int evaluate(stack* s) {
    ...
    pop_many(s,3);
}
```

A module that just refers to a type but does not use any routines defined by the type's module can either include the header file for that module, or it can use a *forward declaration*. For example, the following `.h` file for the module that defines the `evaluate` procedure uses such a forward declaration to let the compiler know that `stack` is a type:

```
#ifndef _EVAL_H
#define _EVAL_H

class stack; // Forward declaration.

int evaluate(stack* s);
// ...

#endif // _EVAL_H
```

Forward declarations are used any time there is a need to use a type before it is defined. This could happen within a `.h` file when defining mutually recursive types.

2.2 Code Structure

This section provides a brief introduction to the syntax and semantics of C++ code. Enough statement forms are described that you will be able to write all your programs using them. You are free to use some statement forms that aren't covered, for example the `switch` statement. On the other hand, **you must not use the goto statement**.

The basic unit for building a C++ program is a *block*. A block has the form:

```
{ sequence of statements and declarations }
```

For example, a block is used to define the *body* of a C++ function, the part that defines what the function does when it is called. A block may also be used as a statement, in which case the statements inside the block are executed in order.

Declarations introduce new local variables. A declaration states the type of the new variable, and also defines a name for it; in addition it can initialize the variable. For example,

```
int x = 3;
```

introduces a new variable named `x`, whose type is `int` (an integer), and initializes it to 3. The *scope* of a declaration, i.e., the part of the program in which the variable introduced by that declaration can be used, is the remainder of the block in which the declaration occurs.

Blocks can be nested. Variable names can be re-used in inner blocks, e.g., the following example is legal.

```
{int x = 3; //decl 1
...
  {int x = 7; // decl 2
  ...
  x = 13; // the x introduced in decl 2
  ...
  }
// the x introduced by decl 2 is no longer accessible
if (x == 3) // the x introduced in decl 1
...
}
```

However, reusing variable names can be confusing and should be avoided.

The basic statements are assignments and calls. These can be combined into larger units by sequencing, conditionals, and loops. Comments are introduced by the symbol // and continue from there to the end of the line. (Comments can also be bracketed by the symbols /* and */.)

Sequencing is achieved simply by writing the statements one after another. For example:

```
x = 7; // an assignment
int z = gcd(x,2); // a function call and an assignment
sort(a); // a call; assume a is an array of ints
```

The = symbol is used for assignment. Note that it is legal for a declaration to follow a statement; C++ allows declarations and statements to be interspersed.

`If` statements are used for conditionals; there are two forms, depending on whether there is an else clause:

```
if (expr) stmt1
if (expr) stmt1 else stmt2
```

stmt1 is the *then-clause*; *stmt2* (if present) is the *else-clause*. Note that the condition *expr* must be placed within parentheses. Note furthermore that the then-clause (and else-clause if present) must consist of a single statement. If more than one statement is needed, a *block* can be used.

Here is an example of an `if` statement:

```
if (x == 0) { // example 1
  y = 7;
  z = 13;
}
```

(The symbol == tests for equality.) Here is another example:

```
if (x == 3) // example 2
  if (y != 7) w = 0;
  else z = 27;
```

(The symbol != tests for inequality.) This example raises a question: when is `z` assigned the value 27? It depends on which `if` we associate the `else` clause with. If the `else` is associated with the second (inner) `if`, the assignment occurs when `x == 3` *and* `y == 7`. If the `else` is associated with the first (outer) `if`, the assignment occurs when `x != 3`, regardless of the value of `y`. C++ resolves this question by pairing the `else` clause with the closest `if` that it could associated with; thus, the assignment to `z` happens when `x == 3` and `y == 7`. If you want the other meaning, you can use explicit grouping, e.g.,

```
if (x == 3) { // example 3
  if (y != 7) z = 13; }
  else z = 27;
```

The `if` statement examples illustrate how the semicolon is used with statements. A semicolon is required after a statement unless the statement is a block or ends with a block. In general it doesn't hurt to place a semicolon where it isn't needed. The exception is an unneeded semicolon before an `else`; if the then-clause is a block (as in example 3), it must *not* be followed by a semicolon.

There are two forms of loops, `while` loops and `for` loops. Here is the form of the `while` loop:

```
while (expr) stmt
```

For example the following code sums the integers from 1 to 100:

```
int sum = 0;
int i = 1;
while (i ≤ 100) {
  sum = sum+i;
  i++ };
```

The form `i++` causes `i` to be incremented by 1.

The semantics of the `while` statement is as follows: The expression *expr* is evaluated. If it evaluates to false (or 0) the loop terminates. Otherwise *stmt* is evaluated, and then control continues at the start of the loop. (There is also a variant on the `while` statement, a do-while statement that has the test at the end of each execution of the loop.)

The `for` statement is convenient for looping over a range of integers; it can be used in more general ways, but a `while` statement is usually better in these cases. It has the following form:

```
for (expr_or_decl; expr1; expr2) stmt
```

The *expr_or_decl* is either an assignment or a declaration with initialization; it defines the loop variable and its initial value. (An assignment, e.g., `x = 0`, is an expression.) If it is a declaration, the variable introduced can be used *only* within the remainder of this `for` statement. (This restriction has only recently been added to C++; using a *decl* here can lead to problems with some compilers.) The *expr1* defines the condition under which execution of the loop will continue; typically *expr1* has the form `x < term`, where `term` is some termination value. Note that *expr1* is evaluated each time the loop executes, which can lead to performance problems if it is an expensive expression. The *expr2* defines an expression to evaluate each time the loop body completes. In most `for` loops, *expr2* simply increments the loop variable. The *stmt* is the loop body.

Here is a typical use of a `for` statement:

```
for (int x = 0; x < n; x++) a[x] = x;
```

Here `a` is assumed to be an array of integers, defined for indexes 0 up to `n` − 1; the loop initializes `a` one element at a time.

The `for` statement is just a more convenient way of expressing a loop that could have been written using a `while` statement. Its meaning is defined by the following translation.

```
for (expr_or_decl; expr1; expr2) stmt
```

translates to:

```
{ expr_or_decl;
  while (expr1) {stmt expr2;}
}
```

2.2.1 Expressions

C++ allows the full range of arithmetic and boolean expressions. Some examples were given above; here are some statements that use more expression forms:

```
x = x * y + z;
if (x < y && (y < z || y > w)) ...;
a[i] = a[i]/2;
```

Here `&&` is logical *and* and `||` is logical *or*; evaluation of both logical operators is *short-circuited*: if evaluation of the first clause is sufficient to determine the result, the second clause is not executed. Order of evaluation is as indicated in the text, e.g., in the `if` statement, `x < y` will be evaluated first. Short-circuit tests are particularly helpful when the second clause cannot be executed if the first is determined. For example, consider the following `if` statement:

```
if (i < a.length() && a[i] == j) { ... }
```

```
int gcd (int n, int d) {
  // the gcd is computed by repeated subtraction
  check (n > 0 && d > 0, "gcd call with a non-positive argument");
  while (n != d)
    if (n > d) n = n - d; else d = d - n;
  return n;
} // end gcd
```

Figure 2.1 The gcd Function.

If the first clause is false, then the array index `i` will be out of bounds, and the second clause will cause a runtime error. However, short-circuit evaluation will prevent the second clause from being evaluated in this case.

Precedence of operators and associativity are fairly standard. For example, `*` has higher precedence than `+`. For the exact details, consult the C++ text. The order can be controlled exactly by using parentheses, e.g., `(x + y) * z`. A sensible approach is to learn the precedence rules for a few common cases, and use parentheses to spell things out explicitly otherwise. Some of the less common operators, such as `<<`, `>>`, and `?:`, have possibly surprising precedence rules.

The part of the C++ semantics that is *not* standard is that all expressions are actually syntactic forms that control *method calls*. (Methods are called member functions in C++.) Thus `x+y` is actually a call on the "`operator +`" method of `x`, passing `y` as an argument. We will discuss this point further in Section 2.5.

2.2.2 Functions

In C++, functions are used to implement procedural abstractions and also operations of types. Figure 2.1 gives an example of a C++ function, `gcd`, which computes the greatest common divisor of its two inputs using repeated subtraction and returns the result.

The header of a function definition defines the *signature* of the function, which consists of the arguments and results with their types. Thus the signature of `gcd` indicates that it takes two ints and returns an int.

The `gcd` implementation uses `check` to check that its arguments are reasonable; `check` is defined in Section 2.3.1. It uses the `return` statement to terminate its execution and return control to the caller. The `return` statement can also be used to terminate execution in a function that has no result; the form is simply:

```
return;
```

2.3 The 6.170 Environment

The header file `6170.h` defines the interfaces for a number of abstractions that will be used in 6.170. (Actually, these interfaces are defined in files included in `6170.h`.) For example, it contains definitions of two data types `array` and `string`; these are defined in Section 2.4

You should include `6170.h` in all your `.h` files by putting

```
#include "6170.h"
```

at the beginning of the file. You won't also need to include it in your `.cpp` files since these include their associated `.h` files, and that is sufficient to cause `6170.h` to be included as well.

2.3.1 Checks

Sometimes you will want to have your code check to be sure some error condition has not occurred; if the error has occurred, you want to halt program execution with an error message. You can accomplish this by using the `check` function, which is defined in 6170.h. It takes two arguments, a boolean and a string:

```
void check(bool b, string s);
```

If the boolean is `true`, control continues at the statement after the call to `check`. Otherwise, the program halts with the given error message `s` plus information about where you are in the code. An example of using `check` is given in Section 2.2.2.

2.4 Strings and Arrays

C++ arrays and strings are unsafe: The runtime system does not keep track of the sizes of arrays and strings, and therefore it is easy to access them out of bounds. Therefore **You should not use C++ strings or arrays, except that you are permitted to use C++ string literals.** Instead, special array and string types are provided for your use in `6170.h`.

2.4.1 Strings

A `string` is a sequence of characters. A string of length `N` has `N` characters numbered `0..N-1`.

Strings should typically be allocated on the stack and passed by-value just as the built-in types `int`, `char`, `bool`, etc. are usually stack-allocated and passed by value; this issue is discussed further in Section 2.5.

Here are some short examples of functions using strings.

```
int find_char(string s, char c) {
    // effects: Returns the index in s where c first occurs, or
    //          the length of the string if it never occurs.
    int n = s.length();
    for (int i = 0; i < n; i++) {
        if (s[i] == c) return i;
    }
    return n;
}

string next_token(string s, int i) {
    // effects: Returns a string containing the next space−separated token
    //          that occurs on or after index i in the string s.
    int n = s.length();
    check(i ≤ n, "index out of range");
    while (i < n && s[i] == ' ') i++;
    string result;
    while (i < n && s[i] != ' ') {
        result = result.append(s[i]);
        i++;
    }
    return result;
}
```

Here are some examples of declaring and initializing `string` variables:

```
string s1;           //s1 is the empty string
string s2 = "abc"; //s1 contains a three−character string
string s3("c");      //s3 contains a one−character string
string s4 = s3;      //s4 contains the same one−character string as s3
```

The declarations of `s2` and `s3` illustrate the two ways of initializing a string variable to contains a particular sequence of characters.

Here are some examples of using `strings`, assuming the declarations of `s1`. . . `s4` are still in effect:

```
int x = s2.length( );   // x contains 3 (the length of s2)
char c = s2[0];         // c contains 'a' (the zeroth character of s2)
s2 = s3.append('d');    // s2 now contains "cd"
s2 = s2 + s3;           // s2 now contains "cdc"
s1 = "cdc";             // s1 also contains "cdc"
if (s1 == s2) ...       // true
s3 = s1[3];             // this is an error
```

In statement 3, we are producing a string by appending character 'd' to the characters contained by `s3`. In statement 4 we produce a string that contains the characters of `s2` followed by the characters of `s3`. The sixth statement tests the the equality of two `strings`. The last statement is in error since 3 is not a legal index for `s1` (its legal indexes are 0, 1, and 2); this error will be detected by the `string` implementation, which will signal an exception.

The specifications for the string operations, some of which were used in the above examples, are given below. There are two kinds of operations: *constructors* are used to initialize newly-created objects, and *methods* are used to manipulate existing objects. (Methods are called member functions in the C++ book.) Both constructors and methods belong to an object; the specifications explain what they do with their object. The specification refers to the object as `this`, which is the name used in C++ for this purpose. Constructors and methods will be discussed further in Chapter 4.

Here are specifications of the `string` constructors:

```
// constructors

  // the default constructor
  string();
  // effects:    Initializes this to be the empty string.

  string(char const* str);
  // effects:    Initializes this to contain the C++-style string str.

  // the copy constructor
  string(string const &s);
  // effects:    Initializes this to contain the same sequence of
  //             characters as s.
```

(The copy constructor takes its argument `s` by reference, as discussed in Section 2.7.3.)

Here are specifications for string methods. The use of the reserved word `const` in a method header (after the arguments) indicates that the method will not modify `this`.

```
//methods

  int length( ) const;
  // effects:    Returns the number of characters in this.

  char operator [ ] (int i) const throw (CE_Bounds);
  // effects:    If i is not a legal index (i.e., i < 0 or i ≥ length()),
  //   throws CE_Bounds else returns the ith character of this.
```

```
string operator + (string s) const;
// effects:    Returns a new string that contains the characters of this
//             followed by the characters of s.

string append (char c) const;
// effects:    Returns a new string that contains the characters of this
//             followed by c.

string substring(int start, int len) const
    throw (CE_Negative_Size; CE_Bounds);
// effects:    If len < 0 throws CE_Negative_Size, else if start < 0 or
//   start + len > length(), throws CE_Bounds, else returns a new string
//   containing the characters this[start]..this[start+len-1].

char* get_chars( ) const;
// effects:    Returns a C++ string containing the characters of this
//             followed by the termination character for C++ strings.

void operator = (string s);
// modifies:   this
// effects:    Replaces the contents of this with the contents of s.

//The following methods can be used to compare strings for equality
//and lexicographic ordering. All comparisons are performed in a
//case-sensitive manner. The specifications of these methods should
//be obvious from the names of the methods.

bool operator == (string) const;
bool operator != (string) const;
bool operator <  (string) const;
bool operator ≤  (string) const;
bool operator >  (string) const;
bool operator ≥  (string) const;

//Some additional methods will be defined later
```

Some of these methods are defined using the special `operator` syntax; this allows them to be called using special forms as shown below. For example,

```
c = s[1];
```

illustrates a call on the method `operator [ ]` of `s` with argument 1 and

```
if (s1 == s2) ...
```

illustrates a call on `operator ==` of `s1` with argument `s2`.

Note that the `operator [ ]` and `substring` methods will detect an attempt to access a `string` out-of-bounds; if such an attempt occurs, they throw an exception. If the calling code does not catch the exception, your program will halt with an error message. This is how you can deal with exceptions for now; we will discuss a better way of dealing with them (by *handling* them) in a later chapter.

Here are some short examples of functions using strings. Can you see how to rewrite `next_token` more efficiently using `substring` and `find_char`?

```
int find_char(string s, char c) {
    // effects: Returns the index in s where c first occurs, or
    //          the length of the string if it never occurs.
    int n = s.length();
    for (int i = 0; i < n; i++) {
        if (s[i] == c) return i;
    }
    return n;
}

string next_token(string s, int i) {
    // effects: Returns a string containing the next space-separated token
    //          that occurs on or after index i in the string s.
    int n = s.length();
    while (i < n && s[i] == ' ') i++;
    string result;
    while (i < n && s[i] != ' ') {
        result = result.append(s[i]);
        i++;
    }
    return result;
}
```

In addition to the `string` methods, there are also some routines that allow you to do I/O with strings; these are explained in Section 2.8.

2.4.2 Arrays

An `array<T>` is a contiguous extensible array of elements of type `T`. An array of length `N` has `N` elements numbered `0..N-1`. Arrays can grow and shrink as elements are appended to or removed from the end of the array.

Array is parameterized by the type of the array elements. Whenever an array is used, we must state what type of element is wanted. For example, `array<int>` indicates that we want elements of type `int`.

Arrays should be allocated on the heap and referred to using *pointers*. Here are some examples:

```
array<int> *y = new array<int>( ); // y refers to an empty array
y→append(3); //add 3 to the top of y
int z = y→fetch(0); // z contains 3
y→store(1, 7);   // an error since 1 isn't a legal index in y
```

The first statement declares variable `x` to be a pointer to an `array<int>` object. The declaration also creates the new object by calling the `new` primitive, which creates space for the new array on the heap, initializes it (by calling an array constructor) to be the empty array, and stores a pointer to the new array in `x`. Objects on the heap must always be created explicitly, by calling `new`. The rest of the statements illustrate the syntax that must be used to make a method call with a variable that is a pointer to the object: the pointer is *dereferenced* (using the → operator, which is actually typed in C++ programs as "`->`") to obtain the object, and then the method is called on that object.

Here are the definitions of (some of) the array constructors and methods; the definitions use `T` to refer to the type of the array elements:

```
//constructors

  // the default constructor
  array();
  // effects:   Initializes this to be the empty array.

//methods

  int length( ) const;
  // effects:    Returns the number of elements in this.

  T fetch(int i) const throw (CE_Bound);
  // effects:    If i < 0 or i ≥ length( ), throws CE_Bounds
  //   else returns the ith element of this.

  void store(int i, T value) throw (CE_Bound);
  // modifies:   this
  // effects:      If i < 0 or i ≥ length( ), throws CE_Bounds
  //   else replaces the ith element of this by value.

  T top( ) const throw (CE_Empty);
  // effects:    If this is empty, throws CE_Empty else
  //   returns the last element of this.

  void append(T value);
  // modifies:   this
  // effects:      Appends value to the end of this. This method
```

```
//            increases the length of this by one.

T remove( ) throw (CE_Empty);
// modifies:    this
// effects:     If this is empty, throws CE_Empty else
//   removes the last element of this and returns it.
//   This method decreases the length of this by one.

void remove_many(int n) throw (CE_Bounds)
// modifies:    this
// effects:     If n > length( ), throws CE_Bounds else
//   removes the last n elements of this.  This
//   method decreases the length of this by n.

void clear( );
// modifies:    this
// effects:     Removes all elements of this, causing the length
//   of this to be zero.

array<T>* copy( ) const;
// requires:    T has operator =.
// effects:     Returns a new array with the same number of elements as this.
//            Each element of the new array is obtained by using operator =
//            on the corresponding element of this.

bool equal (array<T> const * x) const;
// requires:    T has operator ==.
// effects:     Returns true if this and x have the same number of elements,
//            and each element of this is == the corresponding element of x.

//Some additional array methods will be defined later.
```

Note that bounds checking is done by all the methods where bounds errors are possible.

2.5 Objects and Variables

All data are accessed by means of variables, which reside on the stack. C++ allows the use of both *local* variables and *global* variables. **Global variables will not be used in 6.170.**

C++ allows objects to be stored either on the stack or in the heap. Objects on the stack are stored in their variables. Objects in the heap are not stored inside variables; instead they are referred to by *pointer* variables. In general, objects of the built-in C++ types plus I/O streams and our type

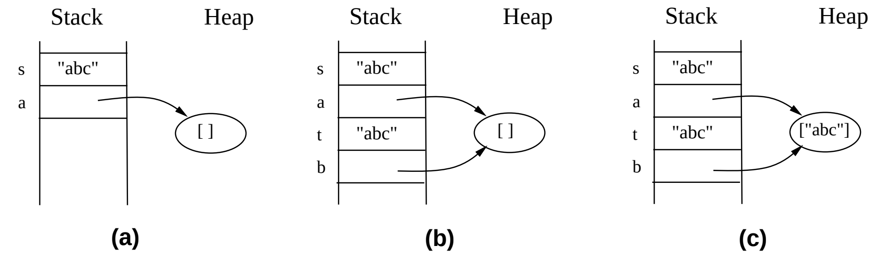

Figure 2.2 Objects and Variables.

`string` should be placed on the stack, and objects of other types, including arrays, should be placed in the heap.

For example, Figure 2.2a shows the effect of the following code

```
string s("abc");
array<string> *a = new array<string>( );
```

which declares two variables, `s` and `a`. The variable `s` is declared to be a string variable and initialized to hold the C++ string "abc". The variable `a` is declared to be a pointer to an `array<string>`; the primitive, `new`, creates a new array object in the heap; the object is then initialized by the array constructor to be the empty array, and a pointer to the new array is assigned to `a`.

All variables in C++ are initialized when they are declared. Declarations that do not include initialization cause the variable to contain an object that is initialized by the *default constructor*. The default constructor takes no argument and initializes its object to some value that makes sense for its type. However, for the built-in types the default constructor does nothing. If such a variable is used before being initialized, the program may fail mysteriously at some later point, and the source of the error is difficult to determine. A good way to avoid this problem is to always initialize variables explicitly. A pointer variable can be initialized to 0. When a pointer variable containing 0 is used, the program will fail immediately; this makes it easier to determine the cause of the problem.

Thus a variable always contains an object. Each object has a *state* or *value*. For example, the state of a string is the sequence of characters it contains; the state of an array of ints is the integers that are its elements; the state of a pointer is the location it points to. The state of an object can be observed and manipulated *only* by calling its methods. Examples of method calls were given in Section 2.4.

Assignment is actually accomplished via a method call, to the `operator` = method. For example, consider the following code

```
string t;
array<string> *b;
t = s;
b = a;
```

The result of running this code is illustrated in Figure 2.2b. As one might expect, the assignment to `t` causes it to contain the same string as `s`, while the assignment to `b` causes it to contain the same pointer as `a`.

This functionality is achieved as follows. Consider a general assignment with the form:

```
expr1 = expr2
```

The method `operator =` is called on the object obtained by evaluating *expr1*, passing it the object obtained by evaluating *expr2* as an argument. The code of *expr1*'s object's `operator =` actually does the work, and therefore the effect of the call depends on what its `operator =` does.

Thus, the assignment `t = s` calls `operator =` of `t`, passing `s` as its argument. As explained in Section 2.4.1, the operator replaces the contents of `t` with those of `s`, achieving the effect shown in Figure 2.2b. In the assignment `b = a`, the call goes to `operator =` of `b`, which is a pointer. The `operator =` method of pointers copies the value of `a`, which is a pointer to an array in the heap, into `b`. As a result `b` now contains a pointer to the same array as `a` does; these two variables *share* the object.

In general, only objects in the heap will be shared. Objects on the stack will not be shared, except when they are passed by-reference, which is discussed in Section 2.7.3. (In C++, it is also possible to create pointers to locations on the stack by using the `&` operator. **Creating pointers to locations on the stack is not allowed in 6.170** except with *call-by-reference*. This means that the `&` operator cannot be used in 6.170.)

Storage for local variables is created on the stack when the program scope that declares the variables is entered and the storage is deallocated automatically when the program exits that scope. This means that storage for objects allocated on the stack is managed automatically.

By contrast, new objects in the heap must be created explicitly using the `new` primitive. Furthermore, C++ programs are supposed to deallocate the storage explicitly when it is no longer needed. However, explicit deallocation is the source of very difficult-to-find program errors. There are two such errors: (1) If an object is deallocated when there is still a pointer to it, the result is a *dangling reference* — a pointer that points to nothing. (2) If an object is *not* deallocated when no more pointers refer to it, the result is a *storage leak* — storage in the heap is lost and cannot be used again in the program.

Both problems can be avoided if the heap is managed by a garbage collector that automatically reclaims storage for objects when they can no longer be reached from program variables. We are providing such a garbage collector for 6.170. **Explicit deallocation of objects should not be done in 6.170.** This means that you should not use the `delete` operator until we discuss explicit deallocation later in the term.

2.5.1 Mutability

All objects are either *immutable* or *mutable*: the state of an immutable object never changes, while the state of a mutable object can change. Changes of state are possible only if some method of the object modifies it. We call such methods *mutators*. A mutable object has at least one mutator, while an immutable object has no mutators. A mutator has a `modifies` clause in its specification stating that it modifies `this`. (Recall that `this` names the method's object.) Methods that are not mutators are called *observers*; observers do *not* include a `modifies` clause, and furthermore their headers contain the keyword `const`. We refer to a type as immutable if all of methods are observers; a type is mutable if some of the methods are mutators.

For example, arrays are mutable, since several methods, e.g., `store`, `append`, are mutators. Also `T& operator [ ](int i)` is a mutator: even though it doesn't modify the array, it gives its caller the ability to do a modification. Strings are "almost" immutable; the only method that modifies a string is `operator =`. In general, types whose objects can be on the stack will be mutable because they will provide an `operator =` method (and possibly other mutators). We will see examples of immutable types in Chapter 4.

If a mutable object is shared by two or more variables, modifications made through one of the variables will be visible when the object is used through the other variable. For example, suppose the shared array in Figure 2.2c is modified by the call

```
b→append(s);
```

(Recall that the `append` method for arrays appends the argument to the end of the array, thus increasing the size of the array by 1.) As discussed in the previous section, the → operator causes the code to follow the pointer from `b` to the array object it refers to. Then the `append` method is called on this object; this method is a mutator that increases its object's size by one and stores the string "abc" in its element with index 0, as shown in Figure 2.2c. This change is visible when the array is accessed using either variable `a` or variable `b`.

2.6 Types and Type Checking

Your C++ programs will be written using built-in and user-defined types. You will normally use the following built-in types: `int`, `char`, `float`, `bool`, `struct`, `enum`, pointers, and also I/O streams (see Section 2.8). In addition you will use the predefined types `string` and `array`. These latter types are actually user-defined types that we have created for you. In some of the problem sets there will be additional user-defined types that we will make available to you. All other types you will define yourselves; we will discuss how to do that in Chapter 4.

C++ is a strongly-typed language. This means that each variable is created by a declaration that defines its type, and also each routine definition declares the types of the routine's arguments and results. Furthermore, the compiler ensures that variables are used in accordance with their type declarations. For example, consider the following code:

```
struct { int a; } y = {3};
int x = y.a; // legal
y = x;  // illegal
```

Here `y` is defined to be a `struct`, which means that it is a collection of variables or *fields*; in the example it contains one field, with name `a` and type `int`. The variable `x` is an `int` variable; the assignment in the second line is legal because the `a` component of `y` is an `int`, and the result is that `x` contains 3. However, the assignment on the third line is not legal since a struct is not the same type as an int.

2.6.1 Type Hierarchy

C++ types are organized into a *hierarchy* in which a type can have a number of *supertypes*; we say the type is a *subtype* of each of its *supertypes*. We will also use the notation `S < T` to mean `S` is a subtype of `T`. In the C++ text, the term *superclass* (*subclass*) will sometimes be used to mean supertype (subtype); in addition the C++ text sometimes says a type is *derived* from another type to mean it is a subtype of the other type.

If `S` is a subtype of `T`, its objects are intended to be usable in any context that expects to use objects belonging to `T`. For `S` objects to be usable, they must have all the methods that `T` objects have; this requirement is enforced by the C++ compiler. In addition, all the method calls must behave the same way on `S` and `T` objects; this requirement is *not* enforced by C++. We will discuss this requirement in much greater detail later in the term when we discuss how to define sub- and supertypes. For now you will only make use of predefined type hierarchies, and you can assume that subtypes are defined properly.

One hierarchy that you will use in your programs is the I/O family of types. For example, input is done by using `istreams`, which are generic input streams; `ifstream` is a subtype of `istream` that allows you to do input from a file. Since `ifstream` is a subtype, you can be sure that a program defined to use an `istream` object will continue to work properly even if it is really using an `ifstream` object. More information about I/O is provided in Section 2.8.

Since objects of a subtype behave like those of a supertype, it makes sense to allow them to be referred to by a pointer to the supertype, and to be passed by-reference where the supertype is expected. These usages are permitted by C++. Thus the header of the ≪ operator for printing ints is:

```
ostream& operator << (ostream& out, int i);
```

A legal call is:

```
ofstream f("myoutput"); // create a new ofstream object
f << 3;                 // here f is the first argument of the call
```

The ≪ operator expects a reference to an `ostream` as its first argument; the call is legal because the actual argument is an `ofstream` object, and `ofstream` is a subtype of `ostream` that does output to a file.

2.6.2 Conversions

Assignment in C++ does not require that the type of the variable being assigned match the type of the right hand side expression exactly. Instead an assignment is legal if the expression on the right hand side has a type that is *convertible* to that of the variable on the left hand side. The exact rules about type conversions are complex and use of them can result in programs that are difficult to understand; therefore we will avoid conversions involving user-defined types in 6.170 as much as we can. The basis of conversion is the presence of constructors that take one argument of some other type, such as the 6.170 `string` constructor that takes a C++ string as an argument; C++ will insert calls to such constructors if doing so will result in the expected type, e.g., in an assignment, the expected type might be that of the variable. For now, you will only need to understand that the following conversions are possible:

`int` can be converted to `float`, `char`, `bool`
`float`, `char`, `bool` can be converted to `int`
C++ string (`char *`) can be converted to 6.170 `string`

In addition, extra conversions are permitted for the literal 0: it can be converted to a pointer or to a `string` (where it produces an empty string).

2.6.3 Defining New Types

A major focus of 6.170 will be the use of data abstraction in program design and implementation. We will make use of C++ `classes` to define new abstract data types. However, C++ has another facility for defining types, using `typedef`s. For example,

```
typedef array<int> int_array;
```

defines `int_array` to be a type. The `typedef` facility introduces *aliases* for types, i.e., additional names for them. Thus `int_array` is just another way of naming the type `array<int>`, and objects declared using either form can be used interchangeably. Such aliases are often useful to provide *short forms* for complicated types. More discussion about this topic can be found in Section 2.3.9 of the C++ text.

2.7 Function Calls

In this section we provide some information about calling functions.

An example of a function was given in Figure 2.1; another example appears in Figure 2.3. The `remove_dupls` function removes any duplicate elements of its argument array, so that, for example, if it is passed an array containing the elements [1, 3, 1, 4] when it returns the array will contain the elements [1, 3, 4]. It affects its caller by modifying the array argument; it does not return any results (which in C++ is indicated by stating that it returns type `void`). Note that the method used in removing duplicates from an array of size `n` requires roughly $n^2/2$ comparisons in the worst case (when there are no duplicates). Thus the algorithm is Order(n^2).

2.7.1 Semantics of Function Call

The header of a function defines a variable for each of its arguments; these variables are called the *formal arguments* or *formals* of the function. When a function is called, the caller must provide an *actual argument* or *actual* for each formal. The compiler will check that enough actuals are provided; in addition each actual must be convertible to the type of the corresponding formal (see the discussion in Section 2.6). The compiler will also check that the caller uses the result in accordance with the function header.

When a call happens, the runtime system creates a new *frame* on the program stack with room for a variable for each of the formals. Then it initializes the formals with information about the actuals of the call as discussed below and transfers control to the start of the called function. The code of the function makes use of the formals to access its arguments.

```
void remove_dupls (array<int> *a) {
  for (int i = 0; i < a→length( ); i++) {
    int x = a→fetch(i);
    int j = i + 1;
    // remove all dupls of x from the rest of a
    while (j < a→length ( ))
      if (x != a→fetch(j)) j++;
      else {
         a→store (j, a→top( ));
         a→remove( ); }
  }
} // end remove_dupls
```

Figure 2.3 Two Functions.

It terminates by *returning* to its caller; at this point it returns its result if any. (A function may also terminate by throwing an exception; exceptions are discussed in a later Chapter.) Returning causes control to transfer to the code in the caller immediately after the call; at this point the stack frame for the called function is deallocated by the runtime system.

In C++ there are two modes of argument passing: *by-value* and *by-reference*. The argument passing mode controls how the actuals are used to initialize the formals. The header indicates the mode; an argument is passed by-value unless the symbol `&` precedes the name of the formal. Thus, all arguments of the functions in Figure 2.3 use *call-by-value*.

2.7.2 Call-by-value

In call-by-value, the value of the actual argument is assigned to the formal. For example, consider

```
int x = gcd (y, z);
```

where `y` and `z` are integers. Here the value of variable `y` is assigned to the formal `n`, and the value of variable `z` is assigned to the formal `d`. The code then accesses the values by using the formals.

By-value formals are distinct from variables of the caller: an assignment to a formal within the function only affects the contents of that variable and has no effect on variables of the caller. The implementation of `gcd` depends on this property: it assigns to its formals `n` and `d` but these assignments are not visible to its caller. Instead, it affects its caller only by means of the value it returns. Thus in the call `gcd(y,z)` the values of `y` and `z` do not change as a result of the call.

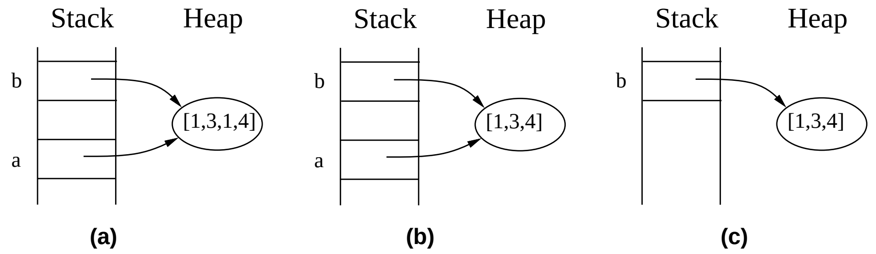

Figure 2.4 Procedure Call.

When by-value formals are pointers, it is possible to affect the caller by modifying the objects they point to (in 6.170 these must be objects in the heap). When a call happens, a pointer to an object in the heap is assigned to the formal, and thus the called function *shares* the object with the caller. It can follow the pointer to the object, and if the object is mutable, it can modify it and the change will be visible to the caller.

This situation is illustrated by the `remove_dupls` function, since its formal `a` is a pointer to an array. For example, consider a call to `remove_dupls`

```
array<int> *b = new array<int> ( );
// code in here causes b's array to have the state [1, 3, 1, 4]
remove_dupls(b);
```

The call causes the the value of variable `b` to be assigned to formal `a`. Since `b` is a pointer, `a` is assigned that pointer and thus it refers to the array referred to by `b`; the effect is shown in Figure 2.4a. Note that the formal variable `a` shares the array object with `b`. The formal is used by `remove_dupls` to modify the shared array. Figure 2.4b shows the state of the computation right before `remove_dupls` returns to its caller. At this point the shared array has been modified, and the stack frame for `remove_dupls` still exists. Figure 2.4c shows the state of the computation after `remove_dupls` returns; at this point its stack frame has been eliminated and its formal no longer exists.

The function `remove_dupls` and its caller both contain pointers to an array object in the heap. C++ allows a call of a function like `remove_dupls` to explicitly pass a pointer to an object on the stack; however **In 6.170, we will not use explicit pointers to objects on the stack.**

2.7.3 Call-by-reference

The other calling mode, *call-by-reference*, allows the caller and called function to share a variable. An example of a function that uses call-by-reference is given in Figure 2.5. The `search` function actually has two results: if the

```
bool search (array<int> const *a, int el, int &index) {
   for (int i = 0; i < a->length(); i++)
     if (a->fetch(i) == el) {
        index = i; // *** assignment to by-reference argument
        return true; }
   return false;
} // end search
```

Figure 2.5 Implementation of search function.

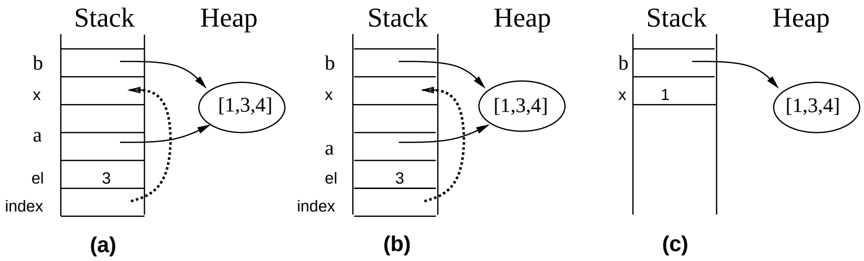

Figure 2.6 Call to search.

given value is in the array `a`, it returns `true` and also stores the index where the value appears in its formal, `index`; otherwise it returns `false` and does not modify `index`.

The header of `search` indicates that the formal, `index`, is being passed by-reference; this is the meaning of the symbol `&`, e.g., in `int &index`. The other two formals of search, `a` and `x`, are passed by-value. The use of `const` in the case of formal `a` indicates that the array object `a` refers to will not be modified by `search`. I.e., `search` is not supposed to call a mutator of this object; the C++ compiler will report an error if the code of `search` contains such a call. We will discuss `const` further in Chapter 3.

When an argument is passed by-reference, the actual should be a location; the formal becomes an *alias* for the actual, i.e., it is another name for that location. An assignment to the formal acts as an assignment to the actual. For example, consider the call

```
int x;
bool res = search(b, 3, x);
```

This call to `search` causes the formal `index` to denote the variable `x`; the effect of the call is illustrated in Figure 2.6a. If the execution of `search` assigns a value to `index`, the assignment will change the value of `x` too. Notice that such an assignment is contained in `search` — on the line marked

***. The assignment will take place, e.g., if array b has the state [1, 3, 4]; in this case the search is successful, causing the value 1 to be assigned to index and therefore to x as illustrated in Figure 2.6b. Thus when search returns, x has the value 1 as shown in Figure 2.6c.

2.8 I/O

This section describes useful C++ input and output abstractions. Section 2.8.1 gives a few simple examples that use the I/O facility. The remaining sections describe some parts of the I/O facility in more detail. Even more detail can be found in Chapter 10 of the C++ text.

2.8.1 Examples

Input occurs on objects of type istream and output occurs on objects of type ostream. (These types can be used by including the header file <iostream.h> in your code; however, this file is included in 6170.h, so that if you include that, you won't have to also include <iostream.h>.) Objects of these types should be allocated on the stack instead of the heap because all of the input and output operators are much more convenient to use on stack-allocated stream objects.

There are three predefined instances of these types:

```
istream  cin;     // Standard input to the program.
ostream  cout;    // Standard output from the program.
ostream  cerr;    // Error output from the program.
```

Streams have special routines that make it convenient to read and output values on them. The binary ≪ operator is used to print values to an ostream. The binary ≫ operator is used to read values from an istream.

The first example is a trivial complete program that prints the string Hello world followed by a newline to standard output.

```
#include <iostream.h> // special form used to refer to standard files
int main(int argc, char **argv) {
   cout << "Hello world\n";
}
```

The second example is a code fragment that prints the factorial of 10. It demonstrates how multiple values of different types (string and integers in this example) can be written to an output stream. The example assumes the presence of a function named factorial that computes the factorial of a non-negative number.

```
cout << "10! = " << factorial(10) << "\n";
```

The next example reads an integer from the standard input and prints its factorial.

```
int n;
cout << "Enter a non-negative integer: ";
cin >> n;
if (cin.good() && n > 0) {
   cout << n << "! = " << factorial(n) << "\n";
} else {
   cout << "bad input in call of factorial\n";
}
```

Note that the code explicitly checks for badly formatted input, by checking that the input stream is `good`, as discussed in Section 2.8.3, and then checking for a positive integer.

The last example reads two integers and prints their product to standard output. This example uses the fact that when an input stream is treated as a boolean, it implicitly calls the `good()` method.

```
int a, b;
cout << "Enter two integers to multiply: ";
if (cin >> a >> b) {
   cout << a << " * " << b << " = " << a * b << "\n";
} else {
   cout << "bad input in multiplication routine\n";
}
```

2.8.2 Output

The C++ types `int`, `char`, `float`, `bool`, and C++ strings can all be printed to an `ostream` with the $\ll$ operator (see Section 2.8.1 for examples). In addition, $\ll$ is often defined for abstract types. For example, `string` provides a definition for the operator.

The output operators for the C++ types `int`, `char`, `float`, and `char*` are given below. These routines all return the passed-in `ostream` so that multiple values can be printed to an `ostream` with a single statement, e.g.,

```
cout << "10! = " << factorial(10) << "\n";
```

They expect the `ostream` parameter to be passed by-reference as discussed in Section 2.7.3. A call on one of these routines might not succeed, e.g., because the `ostream` argument has been closed; in this case the call just returns without printing anything.

```
ostream& operator << (ostream& out, int i);
// modifies:    out
// effects:     Prints a string representation of the integer i to out.
//              Returns out.

ostream& operator << (ostream& out, char c);
// modifies:    out
// effects:     Prints the single character c to out.
//              Returns out.

ostream& operator << (ostream& out, float f);
// modifies:    out
// effects:     Prints a string representation of the real number f to out.
//              Returns out.

ostream& operator << (ostream& out, char* s);
// modifies:    out
// effects:     Prints s to out.
//              Returns out.
```

In addition, our 6.170 strings provide a routine that allows a string to be output:

```
ostream& operator << (ostream& out, string s);
// modifies:    out
// effects:     Prints s to out and then returns out.
```

Thus the binary operation $\ll$ can be used for writing strings to output streams.

You can change the width of integers and other predefined types printed to an `ostream` by printing a special expression of the form `setw(width)` to the `ostream`. The item printed to the `ostream` right after the `setw` will be right justified in the indicated space. The following example prints out the integers in array `a`, right aligned at fifteen spaces. It is important to remember that `setw` only sets the width of the *next* item printed to the `ostream`. `setw` is defined in `<iomanip.h>` and therefore the following code includes that file:

```
#include <iomanip.h>
...
int len = a->length( );
int ind = 0;
```

```
while (ind < len) {
  cout << setw(15) << a->fetch(ind) << "\n";
  ind++; }
```

2.8.3 Errors and End Of File

Some methods can be used to determine error conditions on streams. Streams can be in several different states, as discussed in the C++ text, section 10.3.2. The the following `istream` methods can be used to check for end-of-file and error conditions.

```
bool good( ) const;
// effects:   Returns true iff no error has occurred on the istream yet.
//            Errors may be caused by either system problems, or by
//            attempting to read past the end of the file, or by
//            the presence of invalidly formatted data on the input
//            stream. After a call to >>, the stream will be
//            non-good() if there was an error reading from
//            the stream.

bool bad( ) const;
// effects:   Returns true iff the last input operation on the istream
//            encountered an input error, possibly losing data. No
//            guarantees are made about the success of further
//            operations on the istream.

bool fail( ) const;
// effects:   Returns true iff the last input operation on the istream
//            failed. If the failure was due to badly formatted input,
//            the clear() method may allow further reading
//            from the stream.

void clear() const;
// effects:   Set the state of the stream back to good. This is
//            primarily useful if an input operation failed because
//            of badly formatted input.
```

2.8.4 Input

The C++ built-in types `int`, `char`, and `float` can all be read from an `istream` with the ≫ operator (see Section 2.8.1 for examples). In certain cases, it makes sense to provide a ≫ operator to allow the reading of a value of a user-defined type from an `istream`. For example, ≫ is provided for 6.170 strings.

All of the following $\gg$ operators first skip any white space present in the input stream and then read an item from the stream. These operators may not succeed all of the time. If the stream does not have any more data, or if the next item present in the stream does not have the expected format, then all of these operators behave in a similar manner. The variable that the item is being read into is *not modified*, and the stream is marked to indicate that an error has occurred while reading from it. The instream method `good` can then be used to find out whether an error has occurred.

```
istream& operator >> (istream& in, int& i);
// modifies:   in, i
// effects:    Skips any leading white space on in, reads an integer,
//             and stores it in i.

istream& operator >> (istream& in, char& c);
// modifies:   in, c
// effects:    Skips leading white space on in, reads a character,
//             and stores it in c. In many cases, the get method
//             described below is more useful for reading characters.

istream& operator >> (istream& in, float& f);
// modifies:   in, f
// effects:    Skips any leading white space on in, reads a real
//             number, and stores it in f.

istream& operator >> (istream& in, string& s);
// modifies:   in, s
// effects:    Skips any leading white space on in, reads the next word
//             present in the input stream, and stores it in s.
```

Thus the binary operator $\gg$ can be used for reading 6.170 strings from input streams, just as for reading ints and chars.

The $\gg$ operator for reading 6.170 strings skips leading white space like the other $\gg$ operators, and then reads the next word from the input stream. A word is defined to be the longest consecutive sequence of non-white-space characters present on the stream. Therefore if the standard input contains the sequence of characters "A big mouse!", the following code will set `str1` to "A", `str2` to "big", and `str3` to "mouse!".

```
string str1, str2, str2;
cin >> str1 >> str2 >> str3;
```

Similarly, the $\gg$ operator for reading characters skips leading white space. Often the `get` method of `istream` is useful instead:

```
int get();
// effects:   Returns the value of the next character on the input stream,
//            or the special value EOF if the input stream is positioned
//            at the end of the input.
```

6.170 strings also have a `read_line` routine that can be used to read a line at a time instead of a word at a time:

```
bool read_line(istream& in, string& s);
// modifies: in, s.
// effects: Reads a line from in, stores it in s,
//          and returns true. The terminating newline character
//          is read from the input stream, but is not stored in
//          s. When this routine returns, the stream is
//          positioned right after the terminating newline.
//          If in is already at the end
//          of the file when this routine is called, s is
//          not modified and the routine returns false.
```

2.8.5 More Examples

Here is one way to count the number of non-white-space characters present on standard input:

```
char c;
int count = 0;
while (true) {
    cin >> c;                    // Attempt to read a character
    if (! cin.good()) break;     // If there was an error, stop looping
                                 // (note that c then contains garbage.)
    count++;                     // Count this character
}
```

The loop given above is a little convoluted. A call to the ≫ operator can be tested directly as a boolean to determine whether it failed. Therefore the preceding code can be rewritten as follows:

```
char c;
int count = 0;
while (cin >> c)
    count++;
```

Note that if we wanted to count *all* characters, not just non-white-space characters, we would use the `get` method:

```
int count = 0;
while (EOF != cin.get())
   count++;
```

2.9 Files

The types `ifstream` and `ofstream` are used for streams bound to files. An `ifstream` is bound to a file for reading and is a subtype of `istream`. (See Section 2.6.1 for a discussion of subtypes.) Therefore an `ifstream` can be used any place it is legal to use an `istream`. An `ofstream` is bound to a file for writing and is a subtype of `ostream`. Therefore an `ofstream` can be used any place it is legal to use an `ostream`. The `ifstream` and `ofstream` types are defined in the header file `<fstream.h>`.

Below we define how to open and close file streams. When a file stream is created, it may be invalid. For example, an input file may not exist. The `good()` method should be used to test whether the stream is valid.

```
// ifstream constructor

   ifstream(char const* filename);
   // effects:    Initializes this so that it can be used to read the
   //             contents of the named file.

// ifstream method

   void close( );
   // effects:    Closes the connection between the stream and its
   //             associated file. No more input operations should
   //             be performed on the stream from now on.

// ofstream constructor

   ofstream(char const* filename);
   // effects:    Initializes this so that it can be used to write to
   //             the named file.

// ofstream method

   void close( );
   // effects:    Closes the connection between the stream and its
   //             associated file. No more output operations should
   //             be performed on the stream from now on.
```

Objects of type `ifstream` and `ofstream` should typically be allocated on the stack instead of the heap because all of the input and output operators are much more convenient to use on stack-allocated stream objects.

For convenient creation of file streams, the following additional functions have been provided. They are defined as follows in `file_io.h`:

Input

```
ifstream input_file(string filename);
  // Creates an input stream that is positioned at the start of the
  // existing file named filename. If the file does not exist,
  // the stream created is invalid.
```

Output

```
ofstream create_file(string filename);
  // Creates a new file whose name is filename and returns an output
  // stream for that file. If the file already exists, the output stream
  // will be invalid.

ofstream append_to_file(string filename);
  // Creates an output stream that appends to the file named by filename. If
  // that file does not already exist, the output stream will be invalid.

ofstream overwrite_file(string filename);
  // Creates an output stream that overwrites the file named by filename.
  // The file is immediately truncated to zero length. If
  // the file does not already exist, the output stream will be invalid.
```

Miscellaneous

```
bool access_file(string filename, int mode);
  // Report whether the file named by filename can be accessed in the
  // manner described by mode. This argument is a logical OR of some
  // of the following constants: R_OK, W_OK, X_OK, F_OK. The meaning
  // of mode is exactly as defined for the access system call.
  // Do man access for details.

bool remove_file(string filename);
  // Remove the file named by filename. Return true if successful.
```

The following simple example copies the contents of a file named `infile` to a file named `outfile`.

```
ifstream input = input_file("infile");
ofstream output = create_file("outfile");

// Loop until end of file, reading a character at a time from input
// and printing it out to output.
char c;
for (;;) {          // known as a "forever" loop!
   c = input.get();
   if (c == EOF) break;
   output << c;
}

// Close the files
input.close( );
output.close( );
```

Here is another example that reads a number of integers from a file named `numbers` and prints the factorials of these integers out to a file named `numbers.out`.

```
string fname = "numbers";
ifstream input = input_file(fname);
ofstream output = create_file(fname + string(".out"));

int n;
while (input >> n)
   output << factorial(n) << "\n";

input.close( );
output.close( );
```

2.10 Examples

This section contains the actual files for some of the examples in this chapter. Here is the `.h` file that contains specifications for the non-parameterized procedures; the name of this file is `procs.h`.

```
#include "6170.h"

  int gcd (int n, int d);
    // requires: n and d are positive.
    // effects: Returns the greatest common divisor of n and d.

  void remove_dupls (array<int> * a);
```

```
  // modifies: a
  // effects: Removes all duplicate elements from a. The order of
  //     the remaining elements may change. E.g., if
  //     if a has the state [1, 3, 1, 4] before the call, after the
  //     call it will contain the three elements 1, 3, 4 in some
  //     order.

bool search (array<int> const *a, int el, int &index);
  // modifies: index
  // effects: If el is in a, sets index to some value i such that
  //    el appears at the ith element of a and returns true.
  //    Otherwise returns false.
```

Here is the `.cpp` file for these procedures; the name of the file is `procs.cpp`. Note that the implementation of `gcd` checks to be sure the inputs are positive. The implementation is not constrained to do this, but checking requirements is a good idea provided the check is inexpensive.

```
#include "procs.h"

 int gcd (int n, int d) {
   // the gcd is computed by repeated subtraction
   check (n > 0 && d > 0, "gcd call with a non-positive argument");
   while (n != d)
     if (n > d) n = n - d;
     else d = d - n;
    return n;
 } // end gcd

 void remove_dupls (array<int> *a) {
   for (int i = 0; i < a->length( ); i++) {
     int x = a->fetch(i);
     int j = i + 1;
     // remove all dupls of x from the rest of a
     while (j < a->length ( ))
       if (x != a->fetch(j)) j++;
       else {
           a->store (j, a->top( ));
           a->remove( ); }
    } // end for
 } // end remove_dupls

 bool search (array<int> const *a, int el, int &index) {
   for (int i = 0; i < a->length(); i++)
     if (a->fetch(i) == el) { index = i; return true; }
   return false;
 } // end search
```

Finally, here is a file `procs-main.cpp` that tests the above code.

```
#include "procs.h"

int main(int argc, char **argv)
{
   /* test gcd */
   int x = 100;
   int y = 48;
   cout << "the gcd of " << x << " and " << y << " is "
        << gcd(x,y) << endl;

   /* test remove_dupls */
   {
      cout << "begin testing remove_dupls" << endl;

      array<int> *a = new array<int>();
      /* do empty array */
      remove_dupls(a);

      /* do 1-elt array */
      a->append(10);
      remove_dupls(a);
      if (a->length() != 1 || a->fetch(0) != 10)
         cout << "remove_dupls broken 1" << endl;

      /* do 2-elt array */
      a->append(10);
      remove_dupls(a);
      if (a->length() != 1 || a->fetch(0) != 10)
         cout << "remove_dupls broken 2" << endl;

      /* do 4-elt array */
      a->append(10);
      a->append(10);
      a->append(10);
      remove_dupls(a);
      if (a->length() != 1 || a->fetch(0) != 10)
         cout << "remove_dupls broken 4" << endl;

      /* do mixed elt array */
      a->append(20);
      a->append(10);
      a->append(20);
      a->append(10);
      remove_dupls(a);
```

```
        if (a→length() != 2 || a→fetch(0) != 10 || a→fetch(1) != 20)
            cout ≪ "remove_dupls broken mix" ≪ endl;
            // What is wrong about this test?

        cout ≪ "done testing remove_dupls" ≪ endl;
    }

    /* test search */
    {
        cout ≪ "begin testing search" ≪ endl;
        array<int> *a = new array<int>();
        int index;

        /* look for an element in an empty array */
        bool found = search(a, 1, index);
        if (found) cout ≪ " search broken 1" ≪ endl;

        /* add some elements, don't make them the same as the indexes */
        a→append(10);
        a→append(20);
        a→append(30);
        a→append(40);

        /* look for nonexistent element */
        found = search(a, 1, index);
        if (found) cout ≪ " search broken 2" ≪ endl;

        /* look for each element */
        found = search(a, 10, index);
        if (!found || index != 0) cout ≪ " search broken 3 " ≪endl;
        found = search(a, 20, index);
        if (!found || index != 1) cout ≪ " search broken 4 " ≪endl;
        found = search(a, 30, index);
        if (!found || index != 2) cout ≪ " search broken 5 " ≪endl;
        found = search(a, 40, index);
        if (!found || index != 3) cout ≪ " search broken 6 " ≪endl;

        cout ≪ "done testing search" ≪ endl;
    }
}
```

Exercises

2.1 Computing the greatest common divisor by repeated subtraction (figure 2.3) is not very efficient. Reimplement `gcd` to use division instead.

2.2 Implement a procedure with the header

```
int max (array<int> * a)
```

`max` should return the maximum value in `a`. You may assume that `a` is nonempty.

2.3 Illustrate calling the `gcd` procedure of figure 2.3 with a diagram like that in figure 2.4.

2.4 The array passed as an argument to `remove_dupls` in figure 2.4 contains duplicate elements 1 and 3. Why are these elements not shown in figure 2.4c?

2.5 Implement a procedure that determines whether or not a string is a palindrome. (A palindrome reads the same backward and forward; an example is "deed.")

2.6 Implement a procedure to read in a file containing words and white space and produce a compressed version of the file in an output file. The compressed version should contain all of the words in the input file and none of the white space, except that it should preserve lines.

Chapter 3

Procedural Abstraction

In this chapter we discuss the most familiar kind of abstraction used in programming, the *procedural abstraction*, or *procedure* for short. Anyone who has introduced a subroutine to provide a function that can be used in other programs has used procedural abstraction. Procedures combine the methods of abstraction by parameterization and specification in a way that allows us to abstract a single operation or event, such as computing the gcd of two integers or sorting an array.

A procedure provides a transformation from input arguments to output arguments. More precisely, it is a mapping from a set of input arguments to a set of output results, with possible modifications of the inputs. The set of inputs or outputs or both might be empty. For example, `gcd` has two inputs and one output, but it does not modify its inputs. By contrast, `remove_dupls` has one input and no output, and it does modify its input.

We begin with the benefits of abstraction and, in particular, of abstraction by specification. Next we discuss specifications and why they are needed. Then we discuss how to specify and implement procedures, and we conclude with some general remarks about their design.

3.1 The Benefits of Abstraction

An abstraction is a many-to-one map. It "abstracts" from "irrelevant" details, describing only those details that are relevant to the problem at hand. Its *realizations* or *instances* must all agree in the relevant details, but can differ in the irrelevant ones. Of course, distinguishing what is relevant from what is irrelevant is relative to the task at hand. A major portion of this book will be concerned with how this is done.

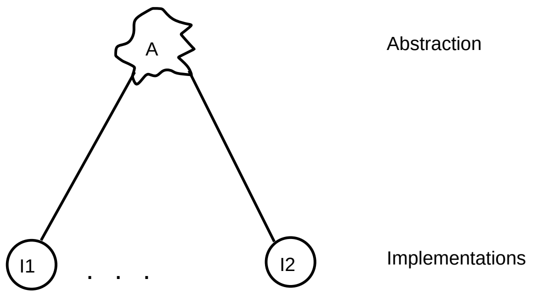

Figure 3.1 The general structure of abstraction by specification.

In abstraction by parameterization, we abstract from the identity of the data being used. The abstraction is defined in terms of formal parameters; the actual data are bound to these formals when the abstraction is used. Thus the identity of the actual data is irrelevant, but the presence, number, and types of the actuals are relevant. Parameterization generalizes modules, making them useful in more situations. A virtue of such generalizations is that they decrease the amount of code that needs to be written and, thus, modified and maintained.

In abstraction by specification, we focus on the behavior that the user can depend on and abstract from the details of implementing that behavior. Thus the behavior—"what" is done—is relevant, while the method of realizing that behavior—"how" it is done—is irrelevant. For example, in a `sort` procedure, the fact that an array is sorted is relevant, while the sorting algorithm—for example, merge sort—is irrelevant.

A key advantage of abstraction by specification is that the implementation is irrelevant, and we can therefore change to another implementation without affecting any program that uses the abstraction (figure 3.1). For example, we could reimplement `sort` using heap sort, and programs using `sort` would continue to run correctly with this new implementation (although some improvement in performance might be noticed). The implementations could even be written in different programming languages, provided that the data types of the arguments are treated the same in these languages. For example, in many systems implemented in higher-level languages, it is common to implement some abstractions in assembly language to improve performance.

Abstraction by specification provides a method for achieving a program structure with two advantageous properties. The first property is *locality*, which means that the implementation of one abstraction can be read or written without our needing to examine the implementation of any other abstraction. To write a program that uses an abstraction, a programmer need understand only its behavior, not the details of its implementation.

Locality is beneficial both when a program is being written and later when someone wants to understand it or reason about its behavior. Because of locality, different abstractions that make up a program can be implemented by people working independently. One person can implement an abstraction that uses another abstraction being implemented by someone else. As long as both people agree on what the used abstraction is, they can work independently and still produce programs that work together properly. Also, understanding a program can be accomplished one abstraction at a time. To understand the code that implements one abstraction, it is necessary to understand what the used abstractions are, but not the code that implements them. In a large program, the amount of information that is not needed can be enormous; we can ignore not only the code of the used abstractions but also the code of any abstractions they use, and so on.

The second property is *modifiability*. Abstraction by specification helps to bound the effects of program modification and maintenance. If the implementation of an abstraction changes but its specification does not, the rest of the program will not be affected by the change. Of course, if the number of abstractions that must be reimplemented is large, making a modification will still be a lot of work. As will be discussed later, the workload can be reduced by identifying potential modifications while designing the program and then trying to limit their effects to a small number of abstractions. For example, if the effects of machine dependencies can be limited to just a few modules, the result will be software that can be transported readily to another machine.

Modifiability leads to a sensible method of tuning performance. Programmers are notoriously bad at predicting where time will actually be spent in a complex system, probably because it is difficult to anticipate where bottlenecks will arise. Since it is unwise to invest effort in inventing structures that avoid nonexistent bottlenecks, a better method is to start with a simple set of abstractions, run the system to discover where the bottlenecks are, and then reimplement the abstractions that are bottlenecks.

3.2 Specifications

It is essential that abstractions be given precise definitions; otherwise, the advantages discussed in section 3.1 cannot be achieved. For example, we can replace one implementation of an abstraction by another only if everything that was depended on in the old implementation is supported by the new one. The entity depended on and supported is the abstraction. Therefore we must know what the abstraction is.

We shall define abstractions by means of *specifications*, which are written in a *specification language* that can be either formal or informal. The advantage of formal specifications is that they have a precise meaning. Informal specifications are easier to read and write than formal ones, but giving them a precise meaning is difficult because the informal specification language is not precise. Despite this, informal specifications can be very informative and can be written in such a way that readers will have little trouble understanding their intended meaning. We shall use both formal and informal specifications in this book, but shall concentrate on informal specifications. Formal specifications will be described in a later chapter.

A specification is distinct from any implementation of the abstraction it defines. The implementations are all similar because they implement the same abstraction; they differ because they implement it in different ways. The specification defines their commonality.

3.3 Specifications of Procedural Abstractions

The specification of a procedure consists of a header and a description of effects. The *header* gives the name of the procedure, the number, order, and types of its inputs, and the type of its output. In addition, names must be given for the inputs. For example, the header for `remove_dupls` is

```
void remove_dupls (array<int> *a);
```

while the header of `sqrt` is

```
float sqrt (float x);
```

The information in the header is just syntactic; it describes the "form" of the procedure. It is similar to a description of the "form" of a mathematical function, as in

$\mathrm{f} : \mathrm{integer} \rightarrow \mathrm{integer}$

In neither case is the meaning—what the procedure or the function does—described. The meaning is captured in the semantic part of the specification, in which the behavior of the procedure is described in English, possibly extended with convenient mathematical notation. This description makes use of the names of the inputs.

Figure 3.2 shows a template of a procedure specification. The semantic part of a specification consists of three parts: the requires, modifies, and effects clauses. These clauses should appear in the order shown, although the requires and modifies clauses are optional.

The *requires clause* states the constraints under which the abstraction is defined. The requires clause is needed if the procedure is partial, that is, if its behavior is not defined for some inputs. If the procedure is total, that is, if its behavior is defined for all inputs, the requires clause can

```
return_type pname (...);
  requires: this clause states any constraints on use
  modifies: this clause identifies all modified inputs
  effects:   this clause defines the behavior
```

Figure 3.2 Specification template for procedural abstractions.

```
string concat (string a, b);
  effects: returns a new string containing the
    characters of a (in the order they occur in a) followed by
    the characters of b (in the order they occur in b).

void remove_dupls (array<int> * a);
  modifies: a
  effects: Removes all duplicate elements from a. The order
    of the remaining elements may change. E.g., if a = [3, 13, 3, 6]
    before the call, then on return a contains
    three elements, 3, 13 and 6, in some order.

int int_sqrt (int x);
  requires: x > 0
  effects: Returns the largest integer y s.t. y*y ≤ x.
```

Figure 3.3 Procedural abstraction specifications.

be omitted. In this case, the only restrictions on a legal call are those implied by the header, that is, the number and types of the arguments. The *modifies clause* lists the names of any inputs that are modified by the procedure; it can be omitted when no inputs are modified. The absence of the modifies clause means that none of the inputs are modified. Finally, the *effects clause* describes the behavior of the procedure for all inputs not ruled out by the requires clause. It must define what outputs are produced and also what modifications are made to the inputs listed in the modifies clause. The effects clause is written under the assumption that the requires clause is satisfied, and it says nothing about the procedure's behavior when the requires clause is not satisfied.

Several specifications are given in figure 3.3. `Concat` does not modify its inputs, but `remove_dupls` modifies its input array, as indicated in the modifies clause. Note the use of an example in the `remove_dupls` specification. Examples can clarify a specification and should be used whenever convenient.

`concat` and `remove_dupls` are total, since neither specification contains a requires clause. `int_sqrt`, however, is partial; it only does its job if its argument is positive. Note that the effects clause does not state what `int_sqrt` does if the argument does not meet this constraint. In this case the implementer can do whatever is convenient; for example, the implementation could return a negative number or even run forever.

When a procedural abstraction modifies the state of some of its argument objects, the specification needs to relate the state of the object at return with its state at the time of call. This is what happens in the specification of `remove_dupls`. Writing such specifications can be simplified by having notation to identify these different states explicitly. We will make use of the following notation: the name of a formal argument, e.g., `x` denotes its state at the time of call, and $\mathtt{x}_{post}$ denotes its state at return. Thus an alternative way of writing the specification for `remove_dupls` is

> void remove_dupls (array<int> * a);
>   modifies: a
>   effects: Removes all duplicate elements from a. The order
>   of the remaining elements may change. E.g., **if** a = [3, 13, 3, 6]
>   then a_{post} contains three elements, 3, 13 and 6, in some order.

Specifications are written in a specification language, not a programming language. Thus, the specifications shown previously are not written in C++. Furthermore, specifications are usually quite different from programs in that they focus on describing what the abstraction is rather than how it is implemented. We shall discuss specifications further in a later chapter.

In C++ the headers of abstractions are given in a .h file; we will include the specifications for the abstractions as comments in that file.

3.3.1 Call-by-reference

As mentioned in Chapter 2, arguments of C++ functions can be passed either by-value or by-reference. This information must be reflected in the specification, in the header.

For example, consider a procedural abstraction `remove_dups_info` that removes duplicates like `remove_dupls`, but in addition returns a count of the number of distinct duplicates removed, and also the sum of all the removed elements. This procedural abstraction effectively returns two results (the count and the sum). However C++ functions can only return a single result. Therefore the procedural abstraction is defined to return the count and to store the sum in a by-reference argument. The specification is given in Figure 3.4, as it would appear in a `.h` file, with the semantics given as comments.

There are three reasons for using call-by-reference:

```
int remove_dups_info (array<int> * a, int &s);
  // modifies: a
  // effects: Removes all duplicate elements from a.  Returns the count
  //   of distinct duplicates removed and stores the sum of removed elements
  //   in s. The order of the remaining elements may change. E.g., if
  //   a = [3, 13, 3, 6, 3, 6], then returns 2 and on return a_post contains
  //   three elements, 3, 13 and 6, in some order, and s contains 12.
```

Figure 3.4 A specification using call-by-reference.

1. The function has several results. C++ allows a function to have only a single result. If there is a need for multiple results, this can be satisfied by having the function return one result, and using by-reference formals for the others, as was done in the definition of `remove_dups_info`. Or by-reference formals can be used for all the results.

2. Avoiding copying of large values. Call-by-value causes the value of the actual argument to be copied into the formal. This copy can be expensive if the value is large. We will mostly avoid this reason for using call-by-reference because of our policy that most objects are kept in the heap and variables are pointers to them; copying of pointers is cheap. It is also not necessary to pass strings by-reference even though they are on the stack because they are implemented as a small header plus a pointer to the actual characters of the string, which are stored on the heap. When the string is passed by-value only the header is copied.

3. Sharing of stack objects. If a mutable object on the stack is passed by-value, the called procedure has no access to the original object and therefore is unable to modify it. Call-by-reference can be used to give the called procedure access to the object. This is the reason the input and output operators $\gg$ and $\ll$ take the stream argument by-reference; in addition, taking the stream argument by reference avoids an expensive copy of its value.

In 6.170, call-by-reference will be used primarily to define procedural abstractions that have multiple results and for procedures with stream arguments.

Sometimes a procedure returns a reference rather than the value. This is the case for the stream operators $\gg$ and $\ll$, and also for the `operator [ ]` methods of arrays. For example, here is a simple function that simply returns the reference passed to it:

```
int& do_nothing (int& x) {
  return x;
}
```

Care must be taken in implementing such functions: the reference returned must not be to a variable defined in the called function, or you will end up with dangling reference. For example, the following code is in error:

```
int& bad_code (int& x) {
  int y;
  ...
  return y;  // ERROR
}
```

This code returns a reference to `y` (since its result is a reference). However, `y` is a local variable of `bad_code` and therefore its storage will be deallocated automatically when `bad_code` returns. The caller will receive a pointer to a location that doesn't exist.

The only functions that return references that you will define in 6.170 will be the stream operators.

3.3.2 Using const

The keyword `const` is used to indicate arguments that will not be modified by a function *in a way detectable by users*. This section discusses when to use this keyword.

For by-value arguments, assignments to the actual are not visible to the caller, and therefore by-value formals should not be declared `const`. There is no point in saying `int const` or `string const` or even `array<int> * const`, since all this does is rule out a modification (an assignment to the formal) that can't be seen by the caller. Instead `const` should be used only to rule out modifications that are visible to the caller.

If a by-value formal is a pointer to some type `T`, and `T` is a mutable type, and the function doesn't modify that argument, the formal should be declared to be `T const *`. For example, it does make sense to say `array<int> const *` if the function is not supposed to modify the array. This use of `const` occurs for parameter `a` of `search`. The specification for `search` (as it would appear in a `.h` file) is given in Figure 3.5.

On the other hand, if the formal is a pointer to an immutable object, again there is no point in using `const`; in this case, `const` has no meaning since it is not possible to modify an immutable object. We will see examples of formals like this in Chapter 4.

When a by-reference formal is used to avoid expensive copying, and the procedural abstraction is not going to modify that argument, that formal

```
bool search (array<int> const *a, int el, int &index);
  // requires: a is sorted in ascending order
  // modifies: index
  // effects: If el is in a, sets index to some value i such that
  //   el appears at the ith element of a and returns true.
  //   Otherwise returns false.
```

Figure 3.5 Specification of search.

should be marked `const`. This use of `const` can occur with streams. For example, if a function just checks the status of an `istream`, but does not modify it, its header should declare that that formal is an `istream const &`.

3.4 Implementing Procedures

The implementation of a procedure should produce the behavior defined by its specification. In particular, it should modify only those inputs that appear in the modifies clause, and for all inputs that satisfy the requires clause, it should produce the outputs in accordance with the effects clause.

Every programming language includes some mechanism for implementing procedural abstractions. In C++, these abstractions are implemented by means of functions. Such an implementation is given in a `.cpp` file; the file `include`s the .h file contains the function's specification.

Figure 3.6 shows two C++ functions that implement `search` (specified in figure 3.5); one uses linear search, while the other uses binary search. (Yet another implementation of `search` was given in Chapter 2.) These two implementations differ in many details. For example, for all but very small arrays, binary search is faster than linear search. Moreover, if `el` appears in `a` more than once, the two functions may return different indexes (in `index`). Finally, if `el` is contained in `a` but `a` is not sorted, the function using binary search may return false when the other function finds the index `x` or vice versa (as an example, consider `a` = [1, 7, 6, 4, 9] and `el` = 7). Nevertheless, both functions are correct realizations of the `search` abstraction since both provide behavior that is consistent with the specification.

In the example functions we have followed some conventions to enhance program readability. First, we have used the same names for the formals as were used in the specification. This convention makes it easier to relate an implementation of an abstraction to the specification. Following the header, we have included a comment explaining the algorithm in use. Finally, we have adopted formatting conventions to make the code easy to read, and we identify the end of the function with a comment saying `// end pname`.

```
bool search (array<int> const *a, int el, int &index) {
    for (int i = 0; i < a->length(); i++) {
      if (a->fetch(i) == el) {
          index = i;
          return true; };
      if (a->fetch(i) > el) return false;
    };
  return false;
} // end search

bool search (array<int> const *a, int el, int &index) {
  // uses binary search
  int low = 0;
  int high = a->length( ) - 1;
  while (low <= high) {
    int mid = (low + high) / 2; // computes the floor
    int val = a->fetch(mid);
    if (el == val) { index = mid; return true; }
    if (el < val) high = mid - 1; else low = mid + 1;
  }
  return false;
} // end search
```

Figure 3.6 Two implementations of `search`.

As a second example, consider sorting an array. One possible method is merge sort, which reduces the problem to that of merging two arrays that have already been sorted. We begin by dividing the array in half, then sort each half and merge the results:

% sort the first half of the array
% sort the second half of the array
% merge the two sorted halves

To carry out these steps we use two subsidiary procedures: `merge`, to do the merging, and `merge_sort`, to carry out the sorting of a subpart of the array. `merge_sort` itself will carry out the same three steps on the subpart of the array, giving a recursive algorithm.

Figure 3.7 shows the specifications of `sort` and the two subsidiary abstractions, as they might appear in a .h file, `sort.h`. Note that `sort` does not modify its input array and that both `merge` and `merge_sort` are partial. Figure 3.8 shows the `.cpp` file containing the C++ functions that implement the abstractions.

```
#include "6170.h"

array<int>* sort (array<int> const * a);
// effects: Returns a new array containing the elements of
//     a arranged in ascending order.

array<int>* merge_sort (array<int> const *a, int lo, hi);
// requires: 0 ≤ low ≤ hi ≤ length(a)
// effects: returns a new array containing elements a[lo],...,a[hi]
//     arranged in ascending order.

array<int>* merge (array<int> const *a, array<int> const *b);
// requires: a and b are sorted in ascending order
// effects: returns a new array containing the elements of a and b,
//     arranged in ascending order
```

Figure 3.7 Specification for merge sort.

3.5 Designing Procedural Abstractions

In this section we discuss a number of issues that arise in designing procedures. Procedures, as well as the other kinds of abstractions that we shall discuss later, should be designed to be *minimal*; care should be taken to constrain details of the procedure's behavior only to the extent necessary. In this way we leave more freedom to the implementer, who may be able to provide a more efficient implementation as a result. However, details that matter to users must be constrained, or the procedure will not be what is needed.

One kind of detail that is almost certainly left undefined is the method to be used in the implementation. Generally, users do not depend on such details. (There are exceptions, however; for example, a numerical procedure may be constrained to use a well-known numerical method so that its behavior with respect to rounding errors will be well-defined.) Some details of what the procedure does may also be left undefined, leading to a procedure that is *underdetermined*. This means that for certain inputs, instead of a single correct output, there is a set of acceptable outputs. An implementation is constrained to produce some member of that set, but any member will do.

The `search` procedure is underdetermined because we did not state exactly what index should be returned if `x` occurs in the array more than once. As mentioned earlier, the two implementations of `search` in figure 3.6 differ in their behavior in this regard. Yet each was a correct implementation.

`Remove_dupls` (see figure 3.3) is also underdetermined, since it does not

```
#include "sort.h"

array<int>* sort(array<int> *a) {
  // sort using merge sort
  if (a→length( ) == 0) return new array<int>( );
  return merge_sort(a, 0, a→length( )−1);
 } // end sort

array<int>* merge_sort (array<int> *a, int lo, int hi) {
  if (lo < hi) {// sort two halves of a and then merge
    int m = (lo + hi) / 2;
    return merge(merge_sort(a,lo,m),merge_sort(a,m+1,hi)); }
  // a is already sorted by we must return a new array
  array<int> *new_array = new array<int>();
  new_array→append(a→fetch(lo));
  return new_array;
  } // end merge_sort

array<int>* merge(array<int> *a, array<int> *b) {
  int a_lo = 0;
  int a_hi = a→length();
  int b_lo = 0;
  int b_hi = b→length();
  array<int> *c = new array<int>();
  // merge a and b into c
  while (a_lo < a_hi || b_lo < b_hi)
    if (a_lo ≥ a_hi || (b_lo < b_hi && b→fetch(b_lo) < a→fetch(a_lo))) {
      // move el from b, either because all els of a have been moved
      // or because the next el of b is less than the next el of a
      c→ append(b→fetch(b_lo));
      b_lo++; }
    else {// move element from a
      c → append(a→fetch(a_lo));
      a_lo++; }
  return c;
  } // end merge
```

Figure 3.8 An implementation of merge sort.

necessarily preserve the order of elements in its input array. This lack of constraint may be a mistake, because users may care about the order; if the input array is sorted, for example, it might be desirable to preserve the order. The important point is that what matters depends on what users need. Every detail that matters to users must be specified, while all others should be left undefined.

An underdetermined abstraction can have a *deterministic* implementation, that is, one that, if called twice with identical inputs, behaves identically on the two calls. Both implementations of `search` in figure 3.6 are deterministic. (Nondeterministic implementations require the use of either global data or class member variables, neither of which is used used in this book.)

In addition to minimality, another important property of procedures is *generality*, which is achieved by using parameters instead of specific variables or assumptions. For example, a procedure that works on arrays of any size is more general than one that works only on arrays of some fixed size. Similarly, a procedure that works on arrays of elements of arbitrary type is more general than one that requires the elements to be integers. Generalizing a procedure is only worthwhile, however, if doing so increases its usefulness. This is almost always true when size assumptions are eliminated, since by doing so we ensure that a minor change in the context of use (for example, doubling the size of an array) requires little, if any, program modification.

Another important property of procedures is simplicity. A procedure should have a well-defined and easily explained purpose that is independent of its context of use. A good check for simplicity is to give the procedure a name that describes its purpose. If it is difficult to think of a name, there may be a problem with the procedure.

Finally, the implementation of a procedure should really do something. Procedures are introduced during program design to shorten the caller and make its structure clearer. In this way the caller becomes easier to understand and to reason about. However, it is possible to introduce too many procedures. For example, the `merge` procedure in figure 3.8 is worth introducing because it has a well-defined purpose and because it allows us to separate the details of sorting from those of merging, thus making `merge_sort` easier to understand. Further decomposition is probably counterproductive, however. For example, the loop body in `merge` could be made into a procedure, but its purpose would be difficult to state, and neither `merge` itself nor the new procedure would do much.

Some of the procedures discussed earlier are partial, while others are total. This dichotomy leads to the question of when it is appropriate to define a partial abstraction. Partial procedures are not as safe as total ones, since they leave it to the user to satisfy the constraints in the requires clause.

When the requires clause is not satisfied, the behavior of a partial procedure is completely unconstrained, and this can cause the using program to fail in mysterious ways. For example, `search` might return the wrong index when its input array is not sorted. This error may not be noticed until long after `search` returns. By then, the reason for the error may be obscure, and data objects may have been damaged.

On the other hand, partial procedures can be more efficient to implement than total ones. For example, if `search` had to work even when the input array was not sorted, then neither implementation in figure 3.6 would be correct; only a less efficient implementation that examined all elements of the array (like the one given in Chapter 2) could be used.

In choosing between a partial and a total procedure, we have to make a trade-off. On the one hand is efficiency; on the other is safe behavior, with fewer potential surprises at runtime. How is such a choice to be made? An important consideration is the expected context of use. If the procedure is intended for general use (for example, if it is to be made available as part of a program library), safety considerations should be given great weight. In such a situation, it is impossible to perform a static analysis of all calls to ensure that they satisfy the constraints. Therefore it is wise not to rely on such an analysis.

Alternatively, some procedures are intended to be used only in a limited context. This was the situation with `merge` and `merge_sort`, which are supposed to be used only within the implementation of `sort` using merge sort. In a limited context, it is easy to establish that constraints are satisfied. For example, the two arrays passed to `merge` as arguments have just been sorted. Therefore we have safe behavior without sacrificing efficiency.

Another point is that the implementation of an abstraction is not forbidden to check the constraint given in a `requires` clause. Of course it doesn't make sense to check a constraint when the checking is very expensive, e.g., as it would be in the `sort` example. But sometimes a constraint is not expensive to check; this would be the case for the `int_sqrt` routine specified in Figure 3.3. In such a case it is a good idea to do the check in the abstraction's implementation (the `check` function provided in 6170.h and described in Chapter 2 can be used for this purpose). Since such checks aren't required by the specification, they can be disabled later, when the program is in production use, if this becomes necessary to achieve good performance.

Finally, it is worth noting that a specification is the only record of its abstraction. Therefore, it is crucial that the specification be clear and precise. How to write good specifications is the subject of a later chapter.

3.6 Summary

This chapter has been concerned primarily with procedures: what they are, how to describe their behavior, and how to implement them. We also discussed two important benefits of abstraction and the need for specifications.

Abstraction provides the two key benefits of locality and modifiability. Both are based on the distinction between an abstraction and its implementations. Locality means that each implementation can be understood in isolation. An abstraction can be used without our having to understand how it is implemented, and it can be implemented without our having to understand how it is used. Modifiability means that one implementation can be substituted for another without disturbing the using programs.

To obtain these benefits, we must have a description of the abstraction that is distinct from any implementation. To this end we introduced the specification, which describes the behavior of an abstraction using a special specification language. This language can be formal or informal. Users can assume the behavior described by the specification, and implementers must provide this behavior. Thus the specification serves as a contract between users and implementers.

A procedure is a mapping from inputs to outputs, with possible modifications of some of the inputs. Its behavior, like that of any other kind of abstraction, is described by a specification, and we presented a form for informal specifications of procedures. A procedure is implemented in C++ by a function; in other languages it would be implemented by a similar mechanism.

Since we are interested in design and how to invent good abstractions, we concluded the chapter with a discussion of what procedures should be like. Desirable properties include minimality, simplicity, and generality. Minimality often gives rise to underdetermined abstractions. We also discussed the pros and cons of partial and total procedures. We shall continue to discuss desirable properties in the following chapters as we introduce additional kinds of abstractions.

Further Reading

Abelson, Harold, and Gerald J Sussman, with Julie Sussman, 1985. *Structure and Interpretation of Computer Programs*, chapter 1. Cambridge, Mass.: MIT Press and New York: McGraw-Hill.

Dijkstra, Edgser W., 1972. Notes on structured programming. In *Structured Programming* (New York: Academic Press), pp. 1–82.

Wirth, Niklaus, 1971. Program development by stepwise refinement. *Communications of the ACM* 14(4): 221–227. Reprinted in *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*, edited by David Gries (New York: Springer-Verlag), 1978.

Exercises

3.1 Specify and implement a procedure `is_prime` that determines whether an integer is prime.

3.2 You are to choose between two procedures both of which compute the minimum value in an array of integers. One procedure returns the smallest integer if its array argument is empty. The other requires a nonempty array. Which procedure should you choose and why?

3.3 Suppose that the implementation of sorting by merge sort shown in figure 3.8 were changed as follows: Procedure `merge` is retained, but `merge_sort` is eliminated, so that its work is done directly in `sort`. Is this change a good idea? What purpose does `merge_sort` have? Discuss.

Chapter 4

Data Abstraction

In chapter 3 we discussed one of the major kinds of abstractions used in programming: the procedural abstraction. Procedures allow us to extend the base language with new operations. In addition to operations, however, the base level provides various types of data—for example, integers, reals, booleans. As discussed in chapter 1, we need to extend the base level with new types of data just as much as with new operations. This need is satisfied by *data abstraction*.

What new types are needed depends on the application domain of the program. For example, in implementing a compiler or interpreter, stacks and symbol tables are useful, while accounts are a natural abstraction in a banking system. Polynomials arise in a symbolic manipulation system, and matrices are useful in defining a package of numeric functions. In each case, the data abstraction consists of a set of objects—for example, stacks or polynomials—plus a set of operations. For example, matrix operations include addition, multiplication, and so on, and deposit and withdraw are operations on accounts.

The new data types should incorporate abstraction both by parameterization and by specification. Abstraction by parameterization can be achieved in the same way as for procedures—by using parameters wherever it is sensible to do so. We achieve abstraction by specification by making the operations part of the type. To understand why the operations are needed, consider what happens if we view a type as just a set of objects. Then all that is needed to implement the type is to select a storage representation for the objects; all the using programs can be implemented in terms of this representation. However, if the representation changes, or even if its interpretation changes, all programs that use the type must be changed.

On the other hand, suppose we include operations in the type, obtaining

$$\text{data abstraction} = \langle \text{objects}, \text{operations} \rangle$$

and we require users to call the operations instead of accessing the representation directly. Then to implement the type, we implement the operations in terms of the chosen representation, and we must reimplement the operations if we change the representation. However, we need not reimplement any using programs, because they depend only on the operations and not on the representation. Therefore we have achieved abstraction by specification.

If enough operations are provided, lack of access to the representation will not cause users any difficulty—anything they need to do to the objects can be done, and done efficiently, by calls on the operations. In general, there will be operations to create and modify objects and to obtain information about their values. Of course, users can augment the set of operations by defining procedures, but such procedures would not have access to the representation.

Data abstraction is the most important method in program design, as will be discussed further later on. Choosing the right data structures is crucial to achieving an efficient program. In the absence of data abstraction, data structures must be defined too early—they must be specified before the implementations of using modules can be designed. At this point, however, the uses of the data are typically not well understood. Therefore the chosen structure may lack needed information or be organized in an inefficient way.

Data abstraction allows us to defer decisions about data structures until the uses of the data are fully understood. Instead of defining the structure immediately, we introduce the abstract type with its objects and operations. Implementations of using modules can then be designed in terms of the abstract type. Decisions about how to implement the type are made later, when all its uses are understood.

Data abstraction is also valuable during program modification and maintenance. In this phase, data structures are particularly likely to change, either to improve performance or to accommodate changing requirements. Data abstraction limits the changes to just the implementation of the type; none of the using modules need be changed.

In this chapter we describe how to specify and implement data abstractions in C++. We also discuss ways to reason about the correctness of programs that use and implement types, and some issues that arise in designing new types.

```
class dname {

public:

// overview: A brief description of the behavior of the type's objects goes here.

// constructors:

  // specs for constructors go here

// methods:

  // specs for methods go here

}; // end dname
```

Figure 4.1 The form of a data abstraction specification.

4.1 Specifications for Data Abstractions

Just as was the case for procedures, the meaning of a type should not be given by any of its implementations. Instead there should be a specification that defines its behavior. Since objects of the type are used only by calling the operations, most of the specification consists of explaining what the operations do.

In C++, new types are defined by `classes`. Each class defines a type by defining a name for the type, a set of *constructors*, and a set of *methods*. Constructors are used to initialize new objects of the type. Once an object has been created (and initialized by a constructor), users can access it by calling its methods. (Methods are sometimes referred to as *member functions*.)

The specification will be included in the *class declaration*. A class declaration has a *public* section the defines information about the class that is visible to its users — the constructors and methods and their signatures. Since a specification only contains publically available information, we will include it (as comments) in the public section.

The form of a data abstraction specification is shown in Figure 4.1. The header `class dname` indicates that a new data type called `dname` is being defined. By convention, we include a comment `//end dname` at the end of the class to indicate what class declaration is being terminated.

The keyword `public` indicates the public section of the class; the entire specification is contained in this section. It is necessary to state that the section is public since the default is private; we will discuss the private section of a class declaration in Sections 4.1.2 and 4.3.

The specification has three parts: The *overview* gives a brief description of the data abstraction, including a way of viewing the abstract objects in terms of "well-understood" concepts. Sometimes it presents a model for the objects; that is, it describes the objects in terms of other objects that the reader of the specification can be expected to understand. For example, stacks might be defined in terms of mathematical sequences. The overview section also states whether the type is mutable or immutable.

The overview should also say where the type's objects are expected to reside (stack or heap). For now types will all have objects on the heap; we will discuss types whose objects are intended to be on the stack later. As we shall see then, "stack abstractions" behave differently than "heap abstractions"; this is why the specification must include this information.

The *constructors part* defines the constructors that initialize new objects, while the *methods part* defines the methods that allow access to the objects once they have been created. In these specifications, the concepts introduced in the overview section can be used. Methods can be either procedures or iterators. In this chapter we discuss only procedure methods; iterators are discussed in a later chapter.

Constructors and (procedure) methods are specified using the notation for specifying procedural abstractions with three differences: (1) methods and constructors both belong to objects. The object a method or constructor belongs to is available to it as an implicit argument. The object can be referred to in the method or constructor specification (and also in its implementation) as `this`. (2) constructors initialize their object and therefore their signatures do not indicate any return type. (3) methods are sometimes labeled `const`, meaning they do not modify their object.

A class is declared in a `.h` file. The file may in addition declare some related routines (see Section 4.4).

4.1.1 Specification of intset

Figure 4.2 gives a specification for the `intset` data abstraction. Intsets are unbounded sets of integers with operations to create a new, empty intset, test whether a given integer is an element of an intset, and add or remove elements. The overview indicates that intsets are intended to reside on the heap and that they are mutable. Also it indicates that we will model intsets in terms of mathematical sets. In the constructors and methods sections we specify each operation using this model. Note that the specifications of `insert` and `remove` use the notation $\mathtt{s}_{post}$ to indicate the value of `s` when the operation returns. An input argument name without the *post* qualifier always means the value when the operation is called.

The `intset` type has a single constructor that initializes the new set to be empty. Since a constructor always modifies `this` (to initialize it),

```
class intset {
public:

// Overview: intsets are mutable, unbounded sets of integers.
//     A typical intset is {x_1, ..., x_n}. intset is a heap abstraction.

// Constructors:

  intset ( );
    // effects: Initializes this to be empty

// Methods:

  void insert (int x);
    // modifies: this
    // effects: Adds x to the elements of this, i.e., s_post = s ⋃ x

  void remove (int x);
    // modifies: this
    // effects: Removes x from this, i.e., s_post = s − x

  bool is_in (int x) const;
    // effects: If x is in this returns true else returns false.

  int size ( ) const;
    // effects: Returns the cardinality of this.

  int choose ( ) const;
    // requires: this is not empty
    // effects: returns an arbitrary element of this

private:

  intset(intset const&); // no copy constructor
  void operator = (intset const&); // no operator equal

}; // end intset
```

Figure 4.2 Specification of the intset data abstraction.

we do not bother to indicate the modification in the `modifies` clause. In fact this modification is invisible to users: they do not have access to the constructor's object until after the constructor runs, and therefore they cannot observe the state change.

Once an `intset` object exists, elements can be added to it by calling its `insert` method, and also elements can be removed by calling `remove`. These two methods are *mutators* since they modify the state of their object; their specifications make it clear that they are mutators because they contain a `modifies` clause stating that `this` is modified.

The remaining methods are *observers*: they return information about the state of their object but do not change the state. The `choose` method returns an arbitrary element of the `intset`; note that it has a non-deterministic specification. Also note that it is partial: it is only defined when the `intset` is not empty.

Observers do not have a `modifies` clause. (More accurately, an observer does not have a `modifies` clause stating that `this`, or some argument object of its type, is modified; however typically observers don't modify anything.) Furthermore, observers for a mutable type such as `intset` are labeled as `const` in their header. The C++ compiler uses this information in two ways: (1) In such a method modifications to `this` are not allowed; this point is discussed further in Section 4.7.2. (2) If an abstract object is used in a `const` context, the compiler will allow only methods declared to be `const` to be called on it. For example, consider the following procedure specification:

```
intset * filter (intset const *s, bool (f) (int x));
  // effects: returns a new intset containing all the elements of s
  //      that pass the filter, i.e., all x in s such that f(x).
```

(The second argument of `filter` is a procedure that takes an `int` and returns a `bool`; the formal name of this argument is `f`.) Since `filter` declares that the `intset` object pointed at by `s` is `const`, it will only be able to call the `const` methods `size`, `is_in`, and `choose` on that `intset` object.

In this specification, we are relying on the reader knowing what mathematical sets are; otherwise, the specification would not be understandable. In general, this reliance on informal description is a weakness of informal specifications. It is reasonable to expect the reader to understand a number of mathematical concepts, such as sets, sequences, and integers. However, not all types can be described nicely in terms of such concepts. If the concepts are inadequate, we must describe the type as best we can, even by using pictures, but of course there is always the danger that the reader will not understand the description or will interpret it differently than we intended. As we shall see in a later chapter, this weakness is avoided if we use formal specifications.

4.1.2 Excluded Operations

The information given in the `public` section of a class provides a complete specification for the new type. In particular, it identifies all the operations that can be used on the objects of that type.

However, there are several operations that C++ will define for a class automatically if the class doesn't provide its own declaration. These are: the *default* constructor (the constructor that takes no arguments), the *copy* constructor (the constructor that takes an argument of the type being defined), the `operator =` method, and the *destructor* (the operation that is called whenever an object is deallocated explicitly). Therefore, unless we are careful, the new type will have these additional operations.

There are several reasons why we don't want the automatically provided operations. First, they may not make sense for the new type. In general `operator =` is not a desirable operation for any type whose objects reside on the heap. Furthermore, since it is a mutator, it must be disallowed for immutable types. A second point is that automatically generated implementations are almost certainly *not* correct: the automatically generated constructors are unlikely to provide properly initialized objects, and the automatically generated `operator =` method is unlikely to do the right kind of copying. Furthermore, at the time a new type is specified, its implementation has not yet been defined, and therefore you can't know whether the automatic implementation will be correct.

Fortunately, automatic implementations are easy to avoid by simply making sure to either declare these methods in the public part of the class *or explicitly disallow them* and **we will require you to do this in 6.170**. Operations are explicitly disallowed by including their headers in the `private` section of the class. For example, since the public section of the `intset` class does not provide specifications for the copy constructor or `operator =`, we must provide their headers in the private section, as illustrated in Figure 4.2.

Including headers of forbidden operations in the private section informs the compiler of the unavailability of these operations. This causes the compiler to not generate automatic implementations for them, and furthermore to prohibit uses of these operations in using code. Thus an attempt to use the copy constructor or `operator =` of `intset` will lead to a compile time error.

The specification of `intset` does not include any header for the destructor. Destructors do not matter for now since we are managing storage in the heap with a garbage collector. Destructors will matter if the heap is managed using explicit deallocation; then it will be important for each type with objects on the heap to have a public destructor. We will discuss this issue further in a later chapter.

4.1.3 The poly abstraction

A second example of a data abstraction specification is given in Figure 4.3. Unlike `intsets`, `polys` are immutable: once a `poly` has been created (and initialized by a constructor) it cannot be modified. Operations are provided to create a one-term poly, to add, subtract, and multiply polys, and to test two polys for equality.

The `poly` type has two constructors, one (the default constructor) to create the zero polynomial, and one to create an arbitrary polynomial. (It could also have had a copy constructor if we wanted; however, this constructor has been disallowed.) In general a type can have a number of constructors. All constructors have the same name, the type name, and they are all called using that name, e.g.,

```
poly *p = new poly ( );  // p points to the zero poly
poly *q = new poly (2, 3);  // q points to the poly 2x^3.
```

The compiler determines which constructor to call based on the number and types of the arguments. Therefore the constructors must differ in the number and types of arguments; otherwise there will be a compile time error.

Providing multiple definitions of the same name is called *overloading*. C++ allows function and method names to be overloaded as well. In general overloading of function and method names should be avoided since it can lead to confusion. However sometimes we will use it; an example will be given in Section 4.4 and also when we discuss subclassing in a later chapter.

There are two more points about the specification of `poly`. First note that all methods are observers: none has a `modifies` clause. This is what we expect to see for an immutable data abstraction. However, we have not marked any of the methods `const`. The reason is that it is not possible for users to modify `poly` objects; thus, indicating that a `poly` object is `const` in some using code will not detect any errors. Therefore there is no benefit to marking the methods `const` (and there is a disadvantage as discussed in Section 4.7.2).

We have disallowed `operator =`. As mentioned, we expect this operation to be disallowed for heap abstractions. It *must* be disallowed for immutable abstractions since it is a mutator.

Note that the specification does not use the *post* notation that was used in the `intset` specification. This notation is not needed for immutable abstractions: since object state doesn't change, the pre and post states of objects are identical.

```
class poly {
public:

// Overview: polys are immutable polynomials with integer coefficients.
//     A typical poly is c0 + c1x + .... poly is a heap abstraction.

// constructors:

  poly ( );
    // effects: initializes this to be the zero polynomial.

  poly (int c, int n);
    // requires: n ≥ 0
    // effects: initializes this to be the poly cx^n.

// methods:

  int degree ( );
    // effects: returns the degree of this, i.e., the largest exponent
    //     with a non−zero coefficient. Returns 0 if this is the zero poly.

  int coeff (int d);
    // effects: returns the coefficient of the term of this whose exponent is d.

  poly* add (poly* q);
    // effects: returns the poly this + q.

  poly* mul (poly * q);
    // effects: returns the poly this * q.

  poly *sub (poly *q);
    // effects: returns the poly this − q.

  poly *minus ( );
    // effects: returns the poly − this.

  bool *equal (poly *q);
    // effects: returns true if this and q have the same coefficients for
    //     the same terms; otherwise returns false.

private:

  void operator = (poly const &); // disallowed
  poly(poly const &); // disallowed

}; // end poly
```

Figure 4.3 Specification of the poly data abstraction.

```
poly *diff (poly *p) {
   poly *q = new poly ( );
   for (int i = 1; i ≤ p → degree( ); i++)
      q = q → add(new poly(p → coeff(i)*i, i−1));
   return q;
   } // end diff

void remove_dupls (array<int> *a) {
   intset *s = new intset( );
   int curr = 0;
   int ln = a → length( );
   for (int i = 0; i < ln; i++) {
      int el = a → fetch(i);
      if (s → is_in(el)) continue;
      s → insert(el);
      a → store(curr, el);
      curr++;
      } // end for
   a → remove_many(ln − curr);
   } // end remove_dupls
```

Figure 4.4 Using abstract data types.

The second point is that `poly` has a number of *binary* methods: methods that operate on both `this` and some other object of the type, which is an explicit argument of the method. Note that the explicit argument of such a method is a *pointer* to a `poly`; it must be a pointer since `poly` is a heap abstraction. In addition, `poly` has methods that return `polys`; again, what is returned is a pointer to a `poly` object. (If `poly` had been defined to be a stack abstraction, the signatures of these methods would be different; this is one of the reasons why we are considering only heap abstractions at this point.)

4.2 Using Data Abstractions

Figure 4.4 gives examples of functions that use data abstractions. The `diff` function returns a new `poly` that is the result of differentiating its argument `poly`; note that it doesn't declare its argument `p` to be `poly const *` because, as discussed above, there is no point in doing this for immutable abstractions. The `remove_dupls` function removes duplicates from an array, using an `intset` to keep track of elements it has already seen.

These functions are not able to access the implementation details of the abstract objects since this access is limited to implementations of the objects' constructors and methods. They can use methods to access object state and to modify that state if the object is mutable, and they can use constructors to initialize new objects; they cannot use any of the disallowed operations, however. Since all the data abstractions we are defining are heap abstractions, their objects are accessed by pointer variables, and new objects are created by calling `new`.

4.3 Implementing Data Abstractions

To implement a data abstraction we select a *representation*, or *rep*, for its objects and then implement the constructors to initialize the representation properly and the methods to use/modify the representation properly. The chosen representation must permit all operations to be implemented in a reasonably simple and efficient manner. In addition, if some of the operations must run quickly, the representation must make this possible. Often a representation that is fast for some operations will be slower for others. We might therefore require multiple implementations of the same type; we will discuss how to achieve this in a later chapter.

For example, a plausible representation for an `intset` object is an array of integers, where each integer in the `intset` occurs as an element of the array. We could choose to have each element of the set occur exactly once in the array or allow it to occur many times. The latter choice makes the implementation of `insert` run faster, but slows down `remove` and `is_in`. If `is_in` is likely to be called frequently, we would make the former choice.

A representation typically has a number of components; in C++ each of these is an *instance variable* of objects of the type being implemented. (Sometimes instance variables are referred to as *member variables*.) The implementations of the type's operations access and manipulate the instance variables. Thus when considered from an implementation point of view, objects have both methods and instance variable. To support abstraction, however, it is important to restrict access to the instance variables to the implementation of the methods and constructors; this allows one, for example, to reimplement an abstract type without affecting any code that uses the type. Therefore the instance variables should not be visible to users; code that uses the objects can refer only to their methods.

The instance variables are defined as part of the class declaration, in its *private section*. Defining them in the private section hides them from users. Therefore a class declaration contains both information of interest to users and information that is visible only to the code that implements the class. (C++ actually allows instance variables to be declared in the public section, in which case users could see them, but this is a bad idea as discussed above and **we will not do this in 6.170**.)

Sometimes it is useful to define some *private methods* that can be used only from within the class implementation. The interfaces of these methods are also defined in the private section of the class declaration.

The implementations of the constructors and methods of a data abstraction are contained in a separate file, a `.cpp` file that includes the `.h` file containing the class declaration. The `.cpp` file can also contain implementation of any helping functions that are useful, and also implementations of related routines, but none of these will have access to the representation details (unless they are declared to be *friends*; friends are discussed in Section 4.4).

C++ allows the methods and constructors to be implemented in the `.h` file but it is a bad idea to do this for two reasons: (1) if your rep is ok but there is an error in your implementation, you will have to recompile all the using code when you correct the error, and (2) whenever you include the `.h` file, all the code will be copied, leading to a code blowup. Neither of these problems happens if the code is in a separate file, and therefore we will keep code in a separate file.

4.3.1 Implementation of intset

This section gives a first example of an implementation, for the `intset` data abstraction. It shows both parts of the implementation: the private section of the class declaration where the instance variables and private methods are declared, and the actual implementations of the methods and constructors.

A full class declaration for the `intset` data abstraction is given in Figure 4.5. Here we have chosen the simple implementation mentioned earlier: an array containing the set elements. The class contains the declarations of the instance variables of the rep in its private section. Note that the `els` component of the rep is a pointer to the array; the pointer is necessary since array is a heap abstraction. The class also contains a comment about the interpretation of the rep: it notes that there will be no duplicate elements in the array. We will say more about these comments in Section 4.5 when we discuss rep invariants and abstraction functions.

The class also contains a declaration of a private method, `get_index`, that will be useful in implementing several of the public methods. Note that here we have chosen to use a non-legal index value to represent the case where `x` isn't in the `intset`. Although defining `get_index` this way isn't very safe, it's reasonable in this case because `get_index` will be used only in a limited context — in the implementation of `intset`.

Now that we have a representation for `intset`, we can see that automatically generated implementations for the copy constructor and `operator =` would be wrong. Both operations ought to cause `this` to contain a disjoint copy of its argument. However, the automatic implementations would

```
class intset {
public:

// as in Figure 4.2

private:

  array<int> *els; // the array contains the set elements without duplicates

  int get_index (int x) const;
    // effects: if x is in this returns an index where it occurs
    //     else returns −1.

  void operator = (intset const &); // disallowed
  intset (intset const &); // disallowed
}; // end intset
```

Figure 4.5 Full class declaration for intset.

cause `this` to share the `els` array with the argument, which is wrong. Note that the sharing would be visible to users: a later modification to one of the `intset` objects would affect both objects.

Figure 15.3 shows the implementation of the `intset` methods and constructors. Note that in the `.cpp` file, the names of the methods and constructors must be prefixed with `intset`, e.g., `intset::remove`. The prefix is needed because the `.cpp` file might implement more than just one data abstraction and therefore the code needs to indicate exactly what is being implemented.

Prefixing a name with a type name means that that method or constructor is part of the implementation of that type. Code marked in this way is passed its object as an additional, implicit argument when it is called. The code can use the reserved word `this` to denote its object; `this` is actually a pointer to the object and therefore the code could use it, e.g., to refer to a method `m` as `this` $\rightarrow$ `m`. (The code cannot assign to `this`.) However, the prefix is not needed: the code can refer to methods of its own object by just using their names. Furthermore, the code has a special, important privilege that no code without the prefix has (except for friends): it can access the instance variables of objects of its class. It can refer to the instance variables of its own object as `this` $\rightarrow$ `var_name`, but again the prefix is not needed. (The names of instance variables and methods must be distinct.) Thus in the methods and constructors in the figure, `els` refers to the `els` instance variable of `this`.

The implementation of `intset` is straightforward. The constructor ini-

```
intset::intset ( ) {
  els = new array<int>( );
  } // end

void intset::insert (int x) {
  if (get_index(x) < 0) els → append(x);
  } // end insert

void intset::remove (int x) {
  int i = get_index(x);
  if (i < 0) return;
  els → store(i, els → top( ));
  els → remove( );
  } // end remove

bool intset::is_in (int x) const {
  return (get_index(x) ≥ 0);
  } // end is_in

int intset::get_index (int x) const {
  for (int i = 0; i < els → length( ); i++)
    if (x == els → fetch(i))  return i;
  return −1;
  } // end get_index

int intset::size ( ) const {
  return els → length( );
  } // end size

int intset::choose ( ) const {
  check (els → length( ) > 0, "attempt to select element of empty intset");
  return els → fetch(0);
  } // end choose
```

Figure 4.6 Implementation of intset.

tializes its object by creating the array that will hold the elements and storing a pointer to it in `els`; since the array is empty, no more work need be done. The `insert`, `remove`, and `is_in` methods all make use of the private method, `get_index`, to determine whether the element of interest is already in the set. Doing this check allows `insert` to preserve the no-duplicates condition. This condition is relied on in `size` (since otherwise the length of the array would not be the same as the size of the set) and in `remove` (since otherwise there might be other occurrences of the element that would need to be removed).

Note that no implementation is provided for `operator =`. If an implementation had been provided, it could have been used only within the `intset` implementation. Since none is provided, even this code cannot use the method.

Note that C++ would have provided the right implementation for the default constructor: it would call the default constructor for array. Therefore it might seem that we should take advantage of this to avoid implementing the constructor ourselves. However, as mentioned earlier, we do not know a type's implementation when it is specified, and furthermore we want the specification to allow any legal implementation. Although the implicit implementation of the default constructor works for this particular implementation of `intset`, there is no guarantee that it will work for others (in fact, an example of an implementation where it would not work is given in Section 4.5.2). Therefore the specification should explicitly declare all operations of a type. (Furthermore, if an operation is not declared in the public section of a class, an implementation will *not* be able to implement it.)

4.3.2 Implementation of poly

Now we consider the implementation of the `poly` data abstraction. A simple way to represent `poly`s is to use an array whose ith element is the coefficient of the ith exponent; this representation makes sense if the `poly` is dense. The zero `poly` can be represented either as an empty array, or as a one-element array containing zero; we will use the latter approach. In addition we will have an instance variable that keeps track of the degree of the `poly` since this is convenient. Figure 4.7 gives the full class declaration for `poly`.

Figures 4.8 and 4.9 show the implementation of `poly`. Note again that no implementation is provided for `operator =`. Note also the use of `this` in the implementation of the `add` method to refer to the method's object. Furthermore, note how the implementation of `add` accesses the instance variables of other `poly` objects in addition to its own object.

Note how `sub` and `mul` are implemented in terms of other `poly` methods. Another point is the use of the default `poly` constructor in the implementa-

```
class poly {

public:

  // as in Figure 4.3

private:

   array<int> *trms;  // the coefficients up to the degree,
   int deg;  // the degree

   void operator = (poly &); // disallowed
   poly (poly const&); // disallowed

}
```

Figure 4.7 Complete class for the poly data abstraction.

tions of `add`, `mul`, and `minus` to initialize the new `poly`. In the case of `mul`, the zero `poly` is what you want. In the case of `add` and `minus`, however, you just want a new poly object, which will then be initialized by the method. Finally, note the care taken in the implementation to ensure that the last element in the array holds the coefficient of the degree term of the `poly`; this property is relied on in the implementation of some of the methods, e.g., the `equal` method.

4.4 Related Routines

Sometimes it is useful to define related routines at the same time that you define a data abstraction. For example, printing objects of the new type on a stream can be accommodated by providing the `operator` ≪ as a related routine. This operator is useful since it allows you to display `polys` in a meaningful way.

Figure 4.10 gives a specification of `operator` ≪ for `poly`. Note that the operator takes the `ostream` by reference, and also returns a reference to that `ostream`; this allows it to be used in combination with other calls to the operator, e.g.,

```
cout << "\n the poly is: " << p << "\n";
```

where `p` is a `poly *`. The figure also contains an implementation of the operator.

```
poly::poly ( ) {
  trms = new array<int> ( );
  trms → append(0);
  deg = 0;
  }

poly::poly (int c, int n) {
  check (n ≥ 0, "attempt to construct poly with negative exponent");
  trms = new array<int> ( );
  if (c == 0) {trms → append(0); deg = 0; return;}
  for (int i = 0; i < n; i++) trms → append(0);
  trms → append(c);
  deg = n;
  }

int poly::degree ( ) {
  return deg;
  } // end degree

int poly::coeff (int d) {
  if (d < 0 || d > deg) return 0;
  return trms → fetch(d);
  } // end coeff

poly* poly::add (poly* q) {
  poly *r = new poly( ); // get a new poly
  r → trms → remove( ); // remove the appended "0"
  poly *la, *sm;
  if (deg > q → deg) {la = this; sm = q;} else {la = q; sm = this;}
  for (int i = 0; i ≤ sm → deg; i++)
    r → trms → append((sm → trms → fetch(i)) + (la → trms → fetch(i)));
  for (int j = i; j ≤ la → deg; j++)
    r → trms → append(la → trms → fetch(j));
  for (int k = j−1; k > 0; k−−) {
    if (r → trms → fetch(k) != 0) break;
    r → trms → remove( );
    } // end for
  r → deg = r → trms → length( ) − 1;
  return r;
  } // end add
```

Figure 4.8 First part of poly implementation.

```
poly* poly::mul (poly * q) {
  poly *r = new poly( );
  // if either this or q is zero return the zero poly
  if ( (deg == 0 && trms → fetch(0) == 0) ||
       (q → deg == 0 && q → trms → fetch(0) == 0)) return r;
  for (int i = 0; i ≤ deg; i++)
    for (int j = 0; j ≤ q → deg; j++)
      r = r → add(new poly(trms →fetch(i) * q → trms → fetch(j), i + j));
  return r;
  } // end mul

poly *poly::sub (poly *q) {
  return add (q → minus( ));
  } // end sub

poly *poly::minus ( ) {
  poly *r = new poly ( );
  r → trms = new array<int>( );
  for (int i = 0; i ≤ deg; i++)
    r → trms → append(−(trms → fetch(i)));
  r → deg = deg;
  return r;
  } // end minus

bool poly::equal (poly *q) {
  return (trms → equal(q → trms));
  } // end equal
```

Figure 4.9 Rest of the implementation of the poly data abstraction.

There are many implementations provided for the $\ll$ operator, e.g., to output an int, a string, etc. Therefore this function name is overloaded. Each definition is constrained to take two arguments, of which the first is an `ostream&`. All definitions must differ in their second argument, since this is what determines the code that is actually called. Thus a call

```
o ≪ p;
```

where `p` is a `poly` will go to the operator defined in Figure 4.10.

The specifications of related routines are typically put in the same `.h` file that contains the class defining the data abstraction, and the routines are implemented in the `.cpp` file that implements the abstraction. When implementing these routines, you do not have access to the representation details, even if they are implemented in the same `.cpp` file as the class. If

```
// Specification (in file poly.h):

  ostream & operator << (ostream &o, poly *p);
    // modifies: o
    // effects: prints an abstract representation of p on o. The representation
    //     has the form c0 + c1x + ... + cnxn, where n is the degree of p.
    //     Only terms with non-zero coefficients are represented except that
    //     the zero poly is represented as 0. Returns a reference to o.

// Implementation (in file poly.cpp)

  ostream & operator << (ostream &o, poly *p) {
    o << "\n the poly is: ";
    bool pre = true;
    if (p -> coeff(0) != 0) o << p -> coeff(0); else pre = FALSE;
    for (int n = 1; n <= p -> degree( ); n++)
      if (p -> coeff(n) != 0) {
        if (pre) o << " + "; else pre = TRUE;
        o << p -> coeff(n) << 'x' << n;
        } // end if and for
    if (!pre) o << " 0";
    return o;
    } // end operator <<
```

Figure 4.10 Output operator for poly.

such access is needed, you must put the specification of the related routine inside the public part of the class and declare it to be a *friend*. Then when you implement it, you will be able to access the instance variables directly. A reason why the access might be needed is that you want to display the rep rather than the abstract value; this can be useful for debugging.

For example, when debugging `poly` you might want to see the actual value of `deg` and `trms` rather than the abstract representation shown in Figure 4.10. In this case you would declare `operator` $\ll$ to be a friend of `poly`. Figure 4.11 shows the specification and implementation in this case. The specification of $\ll$ is contained within the class declaration for `poly` (in the public section). Since it is declared to be a friend, it is not a method of `poly`, but rather is an independent function whose implementation can access the instance variables of `poly` objects. The implementation of the operator takes advantage of this ability.

```
// Specification:

class poly {

public:

  // as in Figure 4.3

  friend ostream & operator << (ostream &o, poly *p);
    // modifies: o
    // effects: prints deg and trms of this.
    //     Returns a reference to o.

}; // end poly

// Implementation:

  ostream & operator << (ostream &o, poly *p) {
    o << "\n the degree is: " << p -> deg << "\n the terms are: [ ";
    for (int i = 0; i < p -> trms -> length( ); i++)
      o << p -> trms -> fetch(i) << " ";
    o << "]";
    return o;
    } // end operator <<
```

Figure 4.11 Output operator for poly rep.

4.5 Aids to Understanding Implementations

In this section, we discuss two pieces of information, the abstraction function and the representation invariant, that are particularly useful in understanding an implementation of a data abstraction. This information should be included as comments in the implementation. In addition, we discuss checking the rep invariant, reasoning about the correctness of implementations, and exposing the rep.

4.5.1 The Abstraction Function

Any implementation of a data abstraction must define how objects belonging to the type are represented. In choosing the representation, the implementor has in mind a relationship between the reps and the abstract objects. Particular reps are expected to represent particular abstract objects. For example, in figure 15.3, `intsets` were represented by arrays, where the elements of the array were the elements of the set.

This relationship can be defined by a function, called the *abstraction function*, that maps from the instance variables that make up the rep of an object to the abstract object being represented:

A: $\mathbf{C} \rightarrow \mathcal{A}$

Here `C` is for the class implementing the new type and $\mathcal{A}$ stands for the set of abstract objects. For each object c belonging to `C`, $A(c)$ is the abstract object $a \in \mathcal{A}$ that c represents.

For example, the abstraction function for the `intset` implementation maps objects of the `intset` class to abstract `intset` objects. Figure 4.12 illustrates this function at some points; it shows how class objects containing various `els` components map to `intset` objects. Note that A is many-to-one: many objects of the class map to the same abstract element of $\mathcal{A}$. For example, the `intset` $\{1, 2\}$ is represented by a class object whose `els` component is [1, 2] and also by a class object whose `els` component is [2, 1]. Since the process of abstraction involves forgetting irrelevant information, it is not surprising that abstraction functions are often many-to-one. In this example, the order in which the elements appear in the `els` array is irrelevant.

The abstraction function is a crucial piece of information about an implementation. It defines the meaning of the representation, the way in which the objects of the class are supposed to implement the abstract objects. It should always be described in a comment in the implementation, in the private section of the class declaration.

In writing such a description, however, we are hampered by the fact that if the specification of the type is informal, the range of the abstraction function (the set $\mathcal{A}$) is not really defined. We shall overcome this problem in what follows by giving an informal description of a "typical" abstract object. When we discuss formal specifications in a later chapter, we shall see that they enable us to have well-defined ranges for abstraction functions.

To define the abstraction function, we first define a typical element of the abstract type. This provides us with a way to talk about the abstract objects. Then we can define the abstraction function in terms of this typical object. For example, for `intset`s we might say

// A typical intset is $\{\mathrm{x1}, \ldots, \mathrm{xn}\}$

Here we are using mathematical sets to denote `intset`s, just as we did in the specification of `intset`. Then we can say

// The abstraction function is
// $\quad \mathrm{A(c)} = \{\mathrm{c.els[i]} \mid 0 \leq \mathrm{i} < \mathrm{length(c.els)}\}$

where $\{x|p(x)\}$ is the set of all x such that $p(x)$ is TRUE. Note that in defining the abstraction function we do not bother with the full C++ syntax: we do not use the `->` notation to follow the pointer in `els`, and we use

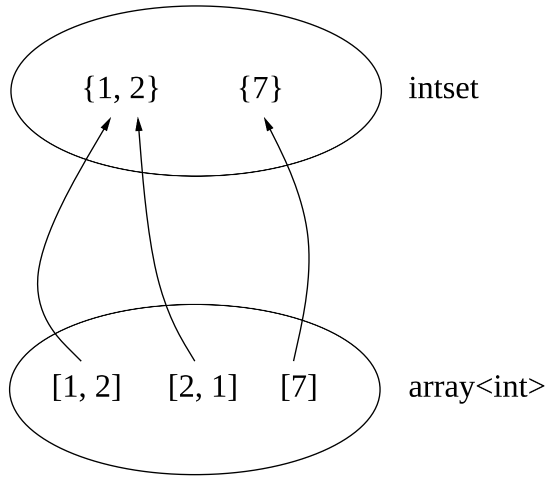

Figure 4.12 An example of an abstraction function.

the notation `c.els[i]` to stand for a call on the `fetch` method of the `els` array.

As a second example, consider the poly implementation. We chose to represent a poly as an array in which the ith element held the ith coefficient unless this coefficient was a trailing zero. We can describe this representation as follows:

```
// A typical poly is c0 + c1x + c2x^2 + ...
// The abstraction function produces the coefficients ci:
// ci = c.trms[i]    if 0 <= i < length(c.trms)
//    = 0            otherwise
```

4.5.2 The Representation Invariant

In C++, type checking guarantees that whenever a method or constructor is called, its object `this` belongs to its class. (Actually, this is true only if the program using the type avoids certain unsafe features such as unions and casts.) Frequently, however, not all objects of the class are legitimate representations of abstract objects. For example, the implementation of `intset` given in figure 15.3 could potentially generate objects with various `els` arrays, e.g., [1, 7, 6], [1, 6, 6], and [6, 1, 7]. However, we decided that each element of the set would be entered in the array exactly once. Therefore legitimate representations of `intset`s do not contain duplicate entries; for example, an object whose `els` array had state [1, 6, 6] is not a legitimate representation.

A statement of a property that all legitimate class objects satisfy is called a *representation invariant*, or *rep invariant.* A rep invariant I is a predicate

$$I : \mathbf{C} \rightarrow \text{bool}$$

that is true of legitimate objects. For example, for `intset`, we might state the following rep invariant:

```
// The rep invariant is
//     For all integers i, j such that 0 ≤ i < j < length(c.els)
//         c.els[i] ≠ c.els[j]
```

Note that I is false if `els` is [1, 6, 6] but true if `els` is [1, 7, 6] or [6, 1, 7].

As a second example, consider an alternative representation of `intsets` that consists of an array of 100 booleans plus an array of integers:

```
array<bool> *els;
array<int> * other_els;
int sz;
```

The idea here is that for an integer i in the range 0...99, we record membership in the set by storing true in `els[i]`. Integers outside this range will be stored in `other_els` in the same manner as in our previous implementation of `intset`. Since it would be expensive to compute the size of the `intset` if we had to examine every part of the `els` array, we also store the size explicitly in the rep. This representation is a good one if almost all members of the set are in the range 0...99 and if we expect the set to have quite a few members in this range. Otherwise the space required for the `els` array will be wasted. (Note that the automatically provided implementation of the default constructor would not be right for this representation.)

Thus we have

```
// The abstraction function is
//     A(c) = {c.other_els[i] | 0 ≤ i < length(c.other_els)}
//                        ∪
//            {j | 0 ≤ j < 100 & c.els[j]}
```

Here ∪ stands for the mathematical union of the two sets. Also,

```
// The rep invariant is
//     length(c.els) = 100
//     & all elements in c.other_els are not in the range 0 to 99
//     & there are no duplicates in c.other_els
//     & c.sz = length(c.other_els) + (count of "true" entries in c.els)
```

Note that the `sz` field of this rep is redundant: It holds information that can be computed directly from the other fields. Whenever there is redundant information in the rep, the relationship of this information to the rest of the rep should be explained in the rep invariant (for example, in the last line of this rep invariant).

Sometimes it is convenient to use a helping function in the rep invariant or abstraction function. For example, the last line of the rep invariant above could be rewritten

// r.sz = length(c.other_els) + cnt(c.els, 0)
//   where $\mathrm{cnt}(a, i) = \mathrm{if}\ i \geq \mathrm{length}(a)$ then 0
//   else if a[i] then $1 + \mathrm{cnt}(a, i+1)$
//   else $\mathrm{cnt}(a, i+1)$

The helping function *cnt* is defined by a recurrence relation.

The implementation of `poly` in figure 4.8 has an interesting rep invariant. Recall that we chose to store coefficients only up to the degree, without any trailing zeros except in the case of the zero polynomial. Therefore we do not expect to find a zero in the high element of the array unless the array has just one element. In addition, these arrays always have at least one element. Furthermore, `deg` must be one less than the length of `els`. Thus we have

// The rep invariant is
// $\mathrm{length(c.trms)} \geq 1$ & $\mathrm{deg} = \mathrm{length(c.trms)} - 1$
//   & $\mathrm{length(c.trms)} > 1 \Rightarrow \mathrm{top(c.trms)} \neq 0$

Recall that the implementation of the `coeff` operation depended on the length of the array being one greater than the degree of the poly; now we see this requirement spelled out in the rep invariant.

Sometimes all concrete objects are legal representations. Then we have simply

// The rep invariant is
//   true

The rep invariant should be stated explicitly even in this case. It may prevent the implementor from depending on a stronger, unsatisfied invariant.

A rep invariant is "invariant" because it always holds for the reps of abstract objects; that is, it holds whenever an object is used outside its implementation. The rep invariant need not hold all the time, since it can be violated while executing one of the type's operations. For example, the `poly::add` method may produce an array with zero in the high element, but such elements are removed from the array before `add` returns. The rep invariant must hold whenever operations return to their callers.

There is a relationship between the abstraction function and the rep invariant. The abstraction function is only of interest for legal representations, since only these represent the abstract objects. Therefore it need not be defined for illegal representations. We will see an example where this is important in Section 4.6.

There is an issue concerning how much to say in a rep invariant. A rep invariant should express all constraints on which the operations depend. A good way to think of this is to imagine that the operations are to be implemented by different people who cannot talk to one another; the rep invariant must contain all the constraints that these various implementors depend on. However, it need not state additional constraints. We will see an example of unstated constraints in Section 4.6.

Whenever a data abstraction is implemented, the rep invariant is one of the first things the programmer thinks about. It must be chosen before any operations are implemented, or the implementations will not work together harmoniously. To ensure that it is understood, the rep invariant should be written down and included as a comment in the code, in the private section of the class declaration (in addition to the abstraction function). Writing down the rep invariant forces the implementor to articulate what is known and increases the chances that the operations will be implemented correctly.

All operations must be implemented in a way that preserves the rep invariant. For example, suppose we implemented `insert` by

```
void intset::insert (int x) {
  els → append(x);
  } // end insert
```

This implementation can produce an object with duplicate elements. If we know that the rep invariant prohibits such objects, then this implementation is clearly incorrect.

The rep invariant is also useful for the reader of an implementation. For example, in an alternative implementation of `intset` we might have decided to keep the rep array sorted. In this case we would have

```
// The rep invariant is
// For all i, j such that 0 ≤ i < j < length(c.els)
//       c.els[i] < c.els[j]
```

and the operations would be implemented differently than in figure 15.3. The rep invariant tells the reader why the operations are implemented as they are.

```
bool poly::rep_ok( ) {
  if (deg != trms → length( ) −1) return FALSE;
  if (deg = 0) return TRUE;
  return trms → top( ) != 0;
  } // end rep_ok

bool intset::rep_ok( ) const {
 for (int i = 0; i < els → length( ); i++) {
   int x = els → fetch(i); // check for dups of x
   int j = i + 1;
   while (j < els → length ( ))
      if (x == els → fetch(j)) return FALSE;
   } // end for
 return TRUE;
 } // end rep_ok
```

Figure 4.13 Implementations of the rep_ok method.

4.5.3 Checking the Rep Invariant

This section discusses how to implement a method that checks the rep invariant. **You must implement such a method for every class.** There are two reasons why we require you to provide this method: it is useful for testing your implementation; and it will help you learn how to define rep invariants.

The method that checks the rep invariant has the following specification:

```
bool rep_ok( );
  // effects: returns true if the rep invariant holds for this;
  //     otherwise returns false
```

In the case of a mutable abstraction, the method will be labeled `const` in its header. The header is included in the public section of the class declaration; a specification need not be given, since the specification is identical for every type.

Figure 4.13 gives implementations of the `rep_ok` methods for the classes we have seen so far. It assumes that the header of the method has been added to the public section of the class declaration. Note that `rep_ok` is marked as `const` for `intset` (since it is mutable) but not for `poly` (since it is immutable).

The `rep_ok` method can be used in two ways: Test programs can call it to check up on whether an implementation is preserving the rep invariant. Also, if it is doesn't cost much to call `rep_ok`, you can use it inside the method and constructor implementations, probably as an argument to `check`, e.g.,

```
check(rep_ok( ), "bad rep in class name");
```

Constructors would call it before they return to ensure the rep invariant holds on the newly-initialized object. Also any methods that modify the rep of either old or newly created objects would call it on these objects before they return. For example, in `poly`, the `add` and `minus` routines would call it, but the `sub` and `mul` methods need not since they don't access the reps of objects directly. In `intset`, the mutators `insert` and `remove` would call it; however, `rep_ok` is costly for `intset` and therefore methods would probably not call it.

4.5.4 Preserving the Rep Invariant

As part of showing that a type is implemented correctly, we must show that the rep invariant holds for all objects of the class. We do this as follows. First, we show that the invariant holds for objects returned by constructors, such as the `poly` constructors. For methods, we can assume when they are called that the invariant holds for `this` and also for all argument objects of the type; we must show that it holds at return for `this` and any arguments of the type and also for returned objects of the type.

For example, the `intset` implementation of figure 15.3 has invariant

// For all integers i, j such that $0 \leq i < j <$ length(c.els)
// c.els[i] $\neq$ c.els[j]

The `intset` constructor establishes this invariant because the newly created array is empty and empty arrays have length 0. The `is_in` method preserves it because we can assume that the invariant holds for `this` when `is_in` is called and `is_in` does not modify `this`; the same is true for `size` and `choose` and private method `get_ind`. Method `insert` preserves the invariant because

1. the invariant holds for `this` at the time of the call;
2. the call to `get_ind` by `insert` preserves the invariant; and
3. `insert` adds `x` to `this` only if `x` is not already in `this` (i.e., `get_ind(x)` returns -1); therefore, since `this` satisfies the invariant at the time of the call, it still satisfies the invariant after the call.

As a second example, consider the poly implementation in figure 4.8 and recall that the invariant is

// $\quad$ $\text{length}(c.\text{trms}) \geq 1$ & $\text{deg} = \text{length}(c.\text{trms}) - 1$
// $\quad$ & $\text{length}(c.\text{trms}) > 1 \Rightarrow \text{top}(c.\text{trms}) \neq 0$

The `poly` default constructor preserves the invariant because it creates the one-element array that represents the zero polynomial; the other `poly` constructor preserves the invariant because it explicitly tests for the zero polynomial. The `mul` operation preserves the invariant because

1. the invariant holds for `this` and `q` at the time of the call;
2. if either `this` or `q` is the zero poly, this is recognized and the proper rep constructed; and
3. otherwise, neither `this` nor `q` contains a zero in its high term—therefore the high term of the `trms` array in the returned `poly`, which contains `top(trms) * top(q.trms)`, cannot be zero.

4.5.5 Exposing the rep

The reasoning described in the previous section is valid *provided* abstract objects cannot be modified outside their implementation. If they could be modified, we could not assume that the rep invariant holds when operations are called, because using modules could cause it to be violated.

In C++ it is difficult (although not impossible) for using code to access the instance variables through the use of unsafe features such as casts. Therefore we need to assume that using code does not do such things. Given this assumptions, reps of objects are hidden *unless* the implementation of the object's class *exposes the rep.* This means that the implementation makes the instance variables or objects they refer to accessible to using code.

One way to expose the rep is to put the instance variable declarations in the public section of the class. This is why we insist that the declarations be placed in the private section. However, even in that case it is still possible to expose the rep.

For example, suppose `intset` had a `members` method with the following specification:

```
array<int> * members ( ) const;
  // effects: Returns an array containing the elements of this, each
  //     exactly once, in arbitrary order.
```

and suppose this method were implemented (as part of the implementation given in Figure 15.3) as follows:

```
array<int>* intset::members ( ) const {
```

```
return els;
} // end members
```

This implementation would allow users of `intset` to access the `els` component directly. To avoid this problem the `members` implementation must return a copy of the `els` array.

Exposing the rep is an implementation error. It can happen either because a method returns an object in the rep as discussed above, or because a constructor or method makes an argument object part of the rep. For example, suppose `intset` had the following constructor:

```
intset(array<int> *elms);
  // effects: initializes this to contain as elements all the ints in elms.
```

and suppose the implementation were:

```
intset::intset (array<int> *elms) {
  els = elms;
  };
```

Again we have an implementation error that results in the rep being exposed.

4.6 Lists

When you design a data abstraction, the first thing you do is define its specification. At this point, you should write and compile your `.h` file. This will allow you to compile code that uses the abstraction even though it hasn't been implemented yet. You won't be able to run the using code, but at least you'll be able to get rid of errors that the compiler catches, such as type errors.

When you implement a data abstraction, you develop the rep and the code for the constructors and methods together. At this point you add information to the private section of the class declaration and you implement the class (in the `.cpp` file). Then you'll have to recompile the programs that use the abstraction, and you'll actually be able to run the programs.

This section illustrates this process by giving another example of a data abstraction, lists of integers. The example also illustrates a partial abstraction function and a recursive implementation.

Integer lists have operations to create an empty list, to form a new list consisting of a given element and the elements of an already existing list, and to decompose a list into its first element and the remaining elements.

A specification is given in figure 4.14. Note that lists are immutable; operations like `cons` do not modify their argument list, but instead return a new list. Note also that the specification includes the `rep_ok` method but does not state its effect since this is standard.

A convenient way to implement a list is as a pair, where the first element of a pair contains an element of the list and the second points to an object representing the rest of the list. This second object can simply be a list itself. In addition, we need a way to indicate that a list is empty. An easy way to do this is to record the size of this list; this information would also be useful if we had a `size` method. Thus we have the rep:

```
int sz; // the size
int el; // the first element
list * r; // the remaining elements
```

Note that the rep is recursive, i.e., the representation of a list is defined in terms of list.

Whenever you invent a rep you have an abstraction function in mind: the whole point of a rep is to represent the abstract objects and the abstraction function simply spells out the relationship. Therefore you should write down the abstraction function as soon as you have invented the rep. In this case, we have the abstraction function:

```
// a typical list is a sequence [e1,...,en]
// the abstraction function is:
//   if c.sz = 0 then [ ]
//       elseif c.sz = 1 then [c.el]
//       else [c.el] || A(c.r)
```

Here the abstraction function is defined recursively; when the size of the list is greater than 1, the result is obtained by concatenating the one-element sequence containing the element of `this` with the sequence we get by applying the abstraction function to the rest of the list. Recursion is frequently used to define an abstraction function, and especially in the case of a recursive rep.

Note that this abstraction function is *partial*. It is only defined when `sz` is non-negative. The requirement that `sz` be non-negative will be part of the rep invariant.

The next step is to identify any constraints on the rep and write them down in the rep invariant. This will typically involve thinking about the implementations of the constructors and methods. In this case the main issue concerns what to do with a one-element list: should the `r` field refer to an empty list, or should it simply be ignored? Using an empty list simplifies

```
class list {

public:

// overview: lists are immutable unbounded sequences of ints.
//   list objects are on the heap.

//constructors

  list ( );
  // effects: makes this be the empty list

//methods

  int first ( );
  // requires: this is not empty
  // effects: returns the first element of this

  list* rest ( );
  // requires: this is not empty
  // effects: returns the list containing all but the first element of this

  bool empty ( );
  // effects: returns true if this is empty else returns false.

  list* cons (int x);
  // effects: returns a list containing x as the first element
  //   followed by the elements of this in order.

  bool equal (list* z);
  // effects: returns true if this and z contain the same elements in
  //   the same order else returns false

  bool rep_ok ( );

private:

  void operator = (list const &); //disallowed
  list(list const&); //disallowed

}; // end list
```

Figure 4.14 Specification of the list data abstraction.

```
list class {

// as before

private:

   int sz; // the size
   int el; // the first element
   list * r; // the remaining elements

   // a typical list is a sequence [e1,...,en]
   // the abstraction function is:
   //   if c.sz = 0 then [ ]
   //      elseif c.sz = 1 then [c.el]
   //      else [c.el] || A(c.r)
   // the rep invariant is
   //    c.sz ≥ 0 & (c.sz > 0 ⇒ c.sz = c.r.sz + 1)

   void operator = (list const &); //disallowed
   list(list*); //disallowed

}; // end list
```

Figure 4.15 Private section of list class.

the implementation of `rest` without affecting the other operations, so let's make this choice.

Therefore we have the following rep invariant:

```
   // the rep invariant is
   //    c.sz ≥ 0 & (c.sz > 0 ⇒ c.sz = c.r.sz + 1)
```

As mentioned, the abstraction function and rep invariant should be included as comments in the private section of the class. Figure 4.15 shows the full private section of the `list` class.

The implementation of list is given in Figure 4.16. One point to notice is that in the case of the empty list, the other fields of the rep are not initialized. Of course we really can't initialize the `el` field in a meaningful way, since no value makes sense for that field. We could initialize the `r` field to contain 0. However, it is acceptable to not do this since an instance variable is accessible only in a limited context (in the implementations of the operations of the class) and none of the list operations access this field when `sz` equals 0. Note that the rep invariant does not constrain the value for the `r` field when `sz` equals 0; this implies that the operation implementations are not obliged to initialize the `r` field in this case.

4.7 Discussion

In this section, we discuss aspects of data abstractions that are of interest both to the definer of the abstraction and to those who use or implement that abstraction. The issues to be discussed are mutability, benevolent side effects, adequacy, and reasoning about properties of abstract objects.

4.7.1 Mutability

Data abstractions are either mutable, with objects whose values can change, or immutable. Care should be taken in deciding on this aspect of a type. In general, a type should be immutable if its objects would naturally have unchanging values. This might be the case, for example, for such mathematical objects as integers, polys, and complexes. A type should usually be mutable if it is modeling something from the real world, where it is natural for the values of objects to change over time. For example, an automobile in a simulation system might be running or stopped, and contain passengers or not. Similarly a type modeling storage, such as an array or a set, is likely to be mutable. However, we might still prefer to use an immutable type in such a case because of the greater safety immutability provides. This is why lists are immutable.

In deciding about mutability, it is sometimes necessary to make a tradeoff between efficiency and safety. Immutable abstractions are safer than mutable ones because no problems arise if their objects are shared. However, new objects may be created and discarded frequently for immutable abstractions, which means that storage management work (e.g., garbage collection) is done more frequently. For example, representing `intsets` as `lists` is probably not a good choice if `insert` and `remove` are used frequently.

Note in any case that mutability or immutability is a property of the type and not of its implementation. An implementation must simply support this aspect of its abstraction's behavior.

4.7.2 Benevolent Side Effects

A mutable abstraction must have a mutable rep, or it will not be possible to provide the required mutability. However, an immutable abstraction need not have an immutable rep. In fact, in C++ immutable reps are unlikely (although they are possible — by declaring all instance variables as const).

```
list::list ( ) {
  sz = 0;
  } // end

int list::first ( ) {
  check (sz > 0, "attempt to get first element of empty list");
  return el;
  } // end first

list* list::rest ( ) {
  check (sz > 0, "attempt to get rest of empty list");
  return r;
  } // end rest

list* list::cons (int x) {
  list *n = new list( );
  n->sz = sz+1;
  n->el = x;
  n->r = this;
  return n;
  } // end cons

bool list::empty ( ) {
  return sz == 0;
  } // end empty

bool list::equal (list *z) {
  if (sz != z->sz) return FALSE;
  if (sz == 0) return TRUE;
  if (el != z->el) return FALSE;
  return equal(z->r);
  } // end equal

bool list::rep_ok ( ) {
  if (sz == 0) return TRUE;
  if (sz < 0) return FALSE;
  return (sz == r->sz + 1);
  } // end rep_ok
```

Figure 4.16 Implementation of list.

For example, polys are immutable, but they have a mutable rep. A mutable rep is not a problem so long as modifications made to the rep cannot be observed by the abstraction's users. For example, sometimes it is useful to create an object by incrementally mutating its rep, though once the object is created, its rep is never modified again. This is the way `polys` are created in some of the `poly` methods.

Mutability is also useful for *benevolent side effects*, which are modifications that are not visible outside the implementation. As an example, suppose rational numbers are represented as a pair of integers:

```
int num, denom;
```

The abstraction function is

```
// A typical rational is n/d
// The abstraction function is
//      A(c) = c.num/c.denom
```

Given this rep, there are several choices to make: what to do with a zero denominator, how to store negative rationals, and whether or not to keep the rational in reduced form (that is, with the numerator and denominator reduced so that there are no common terms). We choose to rule out zero denominators, to represent negative rationals by means of negative numerators, and *not* to keep the rep in reduced form (to speed up operations like multiplication). Thus we have

```
// The rep invariant is
//      c.denom > 0
```

However, to test whether two rationals are equal, it is useful to compute reduced forms; they can be computed using the `gcd` procedure:

```
int gcd (int n, int d);
  // requires: n and d are positive
  // effects: returns the greatest common divisor of n and d.
```

The implementation of the `equal` method is shown in figure 4.17. Once computed, the reduced forms are stored in the rep because this will speed up the next equality test.

The modification of the rep performed by the *equal* operation is a benevolent side effect. Such side effects are often performed for reasons of efficiency. They are possible whenever the abstraction function is many-to-one, since there are then many rep objects that represent a particular abstract object. Sometimes it is useful within an implementation to switch from

```
bool rat::equal(rat * r) {
  if (num == 0) return r.num = 0;
  if (r.num == 0) return FALSE;
  reduce( );
  r.reduce( );
  return (num == r.num && denom == r.denom);
  } // end equal

// reduce is a private method that changes its object to its reduced form
void rat::reduce ( ) {
  int temp = num;
  if (num < 0) temp = −num;
  int g = gcd(temp, denom);
  num = num/g;
  denom = denom/g;
  } // end reduce
```

Figure 4.17 A benevolent side effect.

one of these rep objects to another. Such a switch is safe since the rep still maps to the same abstract object.

A problem arises with benevolent side effects in C++ when an observer marked as `const` does the side effect. This problem will not arise in immutable types, since we do not mark any of their methods as `const`. But it can occur with mutable types. For these types the observers are marked as `const`, and the C++ compiler does not allow the code to modify `this`. In such a method, the type of `this` is `T const *`, where `T` is the type being implemented; i.e., `this` is a pointer to a T object that cannot be modified.

This problem can be solved by explicitly *casting* `this` to `T *`:

```
T * x = (T*) this;
```

Then `x` can be used to access and modify the instance variables of `this`, and to call `T` methods, including mutators, of `this`. The cast, `(T*)`, states explicitly the type that is being produced.

For example, suppose that `intset` were implemented so that `insert` just added the new element to the end of `els`, but when `size` ran it removed the duplicates from `els`. Then we would have:

```
int intset::size ( ) const {
  intset * x = (intset*) this; // the cast
  // code to remove duplicates
  return els → length( ) −1;
  } // end size
```

Casting is unsafe in general, but it is safe when done within the implementation of the type being cast. `this` really is a `T *` — the `const` allows compile-time checking of using code but has nothing to do with the actual type of the object. Furthermore, the code using the cast is responsible for implementing the type correctly, and avoiding incorrect mutations is just a small part of what it must accomplish.

4.7.3 Operation Categories

The operations of a data abstraction fall into four categories:

1. *Creators.* These operations create objects of their type "from scratch" without taking any objects of their type as inputs. Most C++ constructors are creators, e.g., all the ones in the examples in this chapter. But sometimes constructors take arguments of their type (e.g., the copy constructor), and these are *not* creators.

2. *Producers.* These operations take objects of their type as inputs and create other objects of their type. They may be either constructors (e.g., a copy constructor) or methods. For example, `add` and `mul` are producers for `poly`, and `cons` and `rest` are producers for lists.

3. *Mutators.* These are methods that modify objects of their type. For example, `insert` and `remove` are mutators for `intsets`. Clearly, only mutable types can have mutators.

4. *Observers.* These are methods that take objects of their type as inputs and return results of other types. They are used to obtain information about objects. Examples are `size`, `is_in`, and `choose` for `intsets`, and `coeff` and `degree` for `polys`.

Usually, the creators produce some but not all objects; for example, the `poly` creators (the two constructors) only produce single-term polynomials, while the `intset` constructor only produces the empty set. The other objects are produced by producers or mutators. Thus `poly::add` can be used to obtain `polys` with more than one term, while sets containing many elements are produced by `intset::insert`.

Mutators play the same role in mutable types that producers play in immutable ones. A mutable type can have producers as well as mutators; for example, intsets might have a `copy` method that produces a new `intset` containing the same elements as its argument. Sometimes observers are combined with producers or mutators; for example, the array `remove` method modifies the array and returns the removed element.

4.7.4 Adequacy

A data type is *adequate* if it provides enough operations so that everything users need to do with its objects can be done with reasonable efficiency. It is not possible to give a precise definition of adequacy, although there are limits on how few operations a type can have and still be useful. For example, if we provide only the default constructor, and the `insert` and `remove` methods for `intset`, programs cannot find out anything about the elements in the set (because there are no observers). On the other hand, if we add to these three operations just the `size` method, we can learn about elements in the set (for example, we could test for membership by deleting the integer and the seeing if the size changed), but the type would be costly and inconvenient to use.

In general, a data abstraction should have operations from at least three of the four categories discussed in the preceding section. It must have creators, observers, and producers (if it is immutable) or mutators (if it is mutable).

Whether a type is adequate or not depends on the context of use; that is, it must have a rich enough set of operations for its intended uses. If the type is to be used in a limited context, such as a single program, then just enough operations for that context need be provided. If the type is intended for general use, a rich set of operations is desirable.

To decide whether a data abstraction has enough operations, identify everything users might reasonably expect to do. Next, think about how these things can be done with the given set of operations. If something seems too expensive or too cumbersome to do (or both), investigate whether the addition of an operation would help. Sometimes a substantial improvement in performance can be obtained simply by having access to the representation. For example, we could eliminate the `is_in` operation for `intsets` because this operation can be implemented outside the type by using the other operations. However, testing for membership in a set is a common use, and will be faster if done inside the implementation. Therefore, `intset` should provide this operation.

There can also be too many operations in a type. In this case the abstraction may be less comprehensible, and implementation and maintenance are more difficult. The desirability of extra operations must be balanced against the cost of implementing those operations. If the type is adequate, its operations can be augmented by procedures that are outside the type's implementation.

4.7.5 Reasoning about Data Types

To show that a program that uses a data type is correct, we must reason about properties of objects of the type. This reasoning is done at the abstract level; that is, we use the type's specification and ignore its implementation. Such reasoning will be discussed in detail in a later chapter.

As an example, sometimes it is useful to establish a property that holds for all objects of a type. Such a property is called an *abstract invariant* — it is the abstract analog of the rep invariant. Such a property is similar to a theorem about integers and is proved by a similar technique, namely, induction. To establish an abstract invariant, we must show that it holds for all objects of the class. Since only the creators, producers, and mutators produce or modify objects, we have the following scheme:

1. Show that the property holds for all objects returned by creators.

2. Show that the property holds for objects returned by producers, assuming that the property holds for objects passed to the producer as arguments.

3. Show that on exit from mutators the property holds for modified objects, assuming that the property held at the time the mutator was called.

For example, the size of an `intset` is always positive. This property can be established as follows:

1. It clearly holds for the only creator, since the default constructor returns a new, empty `intset`.

2. It holds for `insert` since this only increases the size of the `intset`.

3. It holds for `remove` since this removes an element from the set only if the element was in the set at the time of the call.

This kind of reasoning is called *data type induction*. The induction is on the number of procedure invocations used to produce the current value of the object. The first step of the induction is to establish the property for the creators(s); the induction step establishes the property for the producers and mutators. We do not need to be concerned with the observers since they do not modify their objects (in a way that users can notice) and therefore cannot affect the property.

Note that we are reasoning at an abstract level, not an implementation level. We are not concerned with how `intsets` are implemented. Instead, we work directly with the `intset` specification. Working at the abstract level greatly simplifies the reasoning.

Induction is also used in showing that operation implementations preserve the rep invariant (see section 4.5.4). However, this reasoning is done at the level of the implementation. As before, the first step is to establish the property for the creators. In the induction step, we establish the property for the mutators and producers. If the rep is mutable (which is usually the case), we also need to consider the observers in the induction step. Reasoning about an observer is trivial if it does not modify the rep but is non-trivial for an observer that performs a benevolent side effect.

4.8 Summary

This chapter has discussed data abstractions: what they are, how to specify their behavior, and how to implement them, both in general and in C++. Three examples were discussed in detail. `intsets` are mutable, while `lists` and `polys` are immutable.

In addition, we discussed some important aspects of data type implementations. In general, we want all objects of the class to be legal representations of the abstract objects; the rep invariant defines the legal representations. The abstraction function defines the meaning of the rep by stating the way in which the legal class objects represent the abstract objects. Both the rep invariant and the abstraction function should be included as comments in the implementation (in the private section of the class declaration). They are helpful in developing the implementation, since they force the implementor to be explicit about assumptions. They are also helpful to anyone who examines the implementation later, since they explain what must be understood about the rep.

Finally, we explored some issues that must be considered in designing and implementing data types. Care must be taken in deciding whether or not a type is mutable; an immutable abstraction can have a mutable rep, however, and observers can even modify the rep, provided these modification are "benevolent", i.e., not visible to users. Also care is needed in choosing a type's operations so that it serves the needs of its users adequately. We also discussed data type induction and how it is used to prove properties of objects.

The benefits of locality and modifiability apply to data abstractions as well as to procedures. However, these benefits can be achieved only if we have abstraction by specification. Locality requires that a representation be modifiable only within its type's implementation. If modifications can occur elsewhere, then we cannot establish the correctness of the implementation just by examining its code; for example, we cannot guarantee locally that the rep invariant holds, and we cannot use data type induction with any confidence. Modifiability requires even more—all access to a representation must occur within its type's implementation. If access occurs in some other module, we cannot replace the implementation without affecting that other module.

Thus it is crucial that access to the representation be restricted to the type's implementation. It is desirable to have the programming language help here, so that restricted access is guaranteed provided the implementor does not expose the rep. Otherwise restricted access is another property that must be proved about programs.

Further Reading

Morris, James H., Jr., 1973. Types are not sets. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pp. 120–124.

Hoare, C. A. R., 1972. Proof of correctness of data representations. *Acta Informatica* 1(1): 271–281. Reprinted in *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*, edited by David Gries (New York: Springer-Verlag), 1978.

Exercises

4.1 Implement the procedure

```
poly* diff(poly *p);
// effects: returns the poly that is the result of differentiating p, e.g.,
//    diff (x^3 + 7x + 6) = 3x^2 + 7
```

(See figure 4.3 for a specification of `poly`.)

4.2 Specify and implement the procedure

```
list* reverse (list* z);
```

which reverses the order of the elements in a list. (`lists` are defined in figure 4.14.)

4.3 Suppose `intsets` were implemented using an array of integers as in figure 15.3, but the `els` array was kept sorted in increasing size. Give the rep invariant and abstraction function for this implementation.

4.4 Suppose `polys` (figure 4.3) were implemented with the zero poly represented by the empty array. Give the rep invariant and abstraction function for this implementation.

4.5 Implement `intsets` (figure 4.2) using `lists` (figure 4.14). Be sure to include the rep invariant and abstraction function.

4.6 Suppose we wanted `poly` (figure 4.3) to provide a way of creating a `poly` by means of a $\gg$ operation. Specify and implement such an operation as part of the implementation given in figure 4.8. Does the operation need to be a friend of `poly`?

4.7 Bounded queues have an upper bound, established when a queue is created, on the number of elements that can be stored in the queue. Queues are mutable and provide access to their elements in first-in/first-out order. Queue operations include

```
queue(int);
void enq(int x);
int deq ( );
```

The constructor creates a new queue with maximum size `n`, `enq` adds an element to the front of the queue, and `deq` removes the element from the end of the queue. Provide a specification of queue, including extra operations as needed for adequacy. Implement your specification. Give the rep invariant and abstraction function.

4.8 Specify and implement a rational number type. Give the rep invariant and abstraction function.

4.9 Give an informal argument that the implementation of `poly` in figure 4.8 preserves the rep invariant.

4.10 Give an informal argument that the implementation of `list` in figure 4.16 preserves the rep invariant.

4.11 An abstract invariant of `list` (figure 4.14) is that the size of a list is never negative. Give an informal argument to establish this invariant.

4.12 Suppose we wanted to evaluate a `poly` (figure 4.3) at a given point:

```
int eval(poly *p, int x);
```
// **effects**: *returns the value of p at x, e.g., eval(*$x^2 + 3x, 2) = 10$*.*

Should `eval` be an operation of `poly`? Discuss.

4.13 Categorize the operations of `lists` (figure 4.14) into the four categories of section 4.7.3. Is this type adequate? (Hint: Consider implementing a procedure to concatenate two lists, such as `concat((1 2), (3 4)) = (1 2 3 4)`.)

4.14 A student proposes a type `matrix` with operations to add and multiply matrices and to invert a matrix. These matrices are mutable; for example, the `invert` operation modifies its argument to contain the inverse of the original matrix. A second student claims that a matrix abstraction ought not to be mutable. Discuss.

4.15 A student says that as long as programs outside a type's implementation cannot modify the rep, we have achieved as much as is possible from data abstraction. Discuss.

Chapter 5

Exceptions

A procedural abstraction is a mapping from arguments to results, with possible modification of some of the arguments. The arguments are members of the *domain* of the procedure, and the results are members of its *range*. By "procedure" here we include both C++ functions, and also methods and constructors; we will use the term "procedure" in this inclusive way for the remainder of the book.

Often a procedure makes sense only for arguments in a subset of its domain. For example, the `choose` operation of `intset`s makes sense only if its object is not empty. So far we have dealt with this situation primarily by using procedures whose behavior is defined over only part of their domain, such as

```
int choose ( );
// requires: this is not empty
// effects: returns an arbitrary member of this.
```

The caller of a partial procedure must ensure that the arguments are in the permitted subset of the domain, and the implementor can ignore arguments outside this subset. Thus in implementing `choose`, we could ignore the case of the empty `intset`. (In fact we chose to check for this case; as mentioned earlier, implementations should check the requires clause whenever this is practical.)

Partial procedures are generally a bad idea since there is no guarantee that their arguments are in the permitted subset, and the procedure may therefore be called with arguments outside the subset. When this happens, the result may be an obscure error that is difficult to track down. For example, the procedure may return, allowing the calling code to continue and possibly damage important data bases.

Partial procedures lead to programs that are not robust. A *robust* program is one that continues to behave reasonably even in the presence of errors. If an error occurs, the program may not be able to provide exactly the same behavior as if there were no error, but it should behave in a well-defined way. Ideally, it should continue after the error by providing some approximation to its behavior in the absence of an error; a program like this is said to provide *graceful degradation*. At worst, it should halt with a meaningful error message and without causing damage to permanent data.

A method that enhances robustness is to use *total* procedures: procedures whose behavior is defined for all inputs in the domain. If the procedure is unable to perform its "intended" function for some of these inputs, at least it can inform its caller of the problem. In this way, the situation is brought to the attention of the caller, which may be able to do something about it, or at least avoid harmful consequences of the error.

How should the caller be notified if a problem arises? One possibility is to define a special error result. For example, a factorial procedure might return zero if its argument is not positive:

```
int fact (int n);
// effects: if n > 0 returns n! else returns 0.
```

This solution is not very satisfactory. Since the call with illegal arguments is probably an error, it is more constructive to treat this case in a special way, so that a programmer who uses the procedure is less likely to ignore the error by mistake. Also, if every value of the return type is a possible result of the procedure, as is the case with `choose`, this solution is impossible, since there is no value available to indicate the error. We can handles such cases by introducing an additional result, e.g, a bool that indicates whether the `intset` is empty or not. Even so, it is easy for users to ignore the error by mistake. A method that makes errors more visible, and harder to ignore, is preferable. It would also be nice if the method allowed the handling of errors to be separated from the normal program control flow.

Such a method can be achieved by generalizing procedures. Instead of having a single input domain, we divide the domain D into a number of subsets whose union is D:

$$\mathrm{D} = \mathrm{D}_0 \cup \mathrm{D}_1 \cup \ldots \cup \mathrm{D}_\mathrm{n}$$

Each subset is "interesting" in the sense that the procedure behaves differently in each case. For example, the input of `choose` would be divided into two subsets:

D_0 = {all nonempty intsets}
D_1 = {the empty intset}

For an input in D_0, `choose` returns the chosen integer as before. For an input in D_1, it still returns, but in a way that brings the problem to the attention of its caller. As noted, it is not possible to do this notification by returning an integer, since any integer is a potential outcome for inputs in D_0. Therefore, some other method of notification is needed.

We provide notification by having different output ranges for each domain subset and by naming the different cases. We arbitrarily name the case of input from D_0 the "normal" case; the names of the other cases can be chosen by the definer of the procedure. Thus we have

$$\begin{aligned} \mathrm{p{:}} = \ & \mathrm{D_0} \rightarrow \mathrm{normal} \\ & \mathrm{D_1} \rightarrow \mathrm{ename_1} \\ & \ldots \\ & \mathrm{D_n} \rightarrow \mathrm{ename_n} \end{aligned}$$

where $ename_1, \ldots, ename_n$ are the names of the "exceptional" situations corresponding to input in $D_1, \ldots, D_n$, respectively. $Ename_1, \ldots,$ $ename_n$ are called *exceptions*. For example, we have

$$\begin{aligned} \mathrm{choose{:}} \ & \mathrm{D_0} \rightarrow \mathrm{normal} \\ & \mathrm{D_1} \rightarrow \mathrm{empty} \end{aligned}$$

The use of exceptions supports the practice of *defensive programming*, that is, writing each procedure to defend itself against errors. Errors can be introduced by other procedures, by the hardware, or by the user entering data; these latter errors will continue to exist even if the software is error-free. Of course, checking for these errors takes time, and it is tempting not to bother with the checks, or to use them only while debugging, and suppress them during production. This is generally an unwise practice. Defensive programming is particularly valuable during production because it can prevent a small error from causing a large problem, such as a damaged data base. Removing checks during production is analogous to disconnecting warning lights in an airplane; a pilot would never do this because the results could be catastrophic. Checks should be removed only if we have proved that the errors can never occur or if the checks are prohibitively expensive.

In this chapter we discuss how to specify, implement, and use procedures with exceptions. We also discuss a number of related design issues.

5.1 Specifications

A procedure that can signal exceptions is indicated in C++ by having a *throw clause* in its header. The throw clause actually lists types that can be thrown:

```
throw (list_of_types)
```

Although procedures can throw "ordinary" types, such as ints and chars, we will make use of special exception types for this purpose. The names of these types will serve to name the exceptions. For now we will assume that these types exist somehow; in fact a number of them are defined in `6170_except.h`. We discuss how to define additional exception types in Section 5.2.3.

For example,

```
int choose ( ) const throw (CE_Empty);
```

states that `choose` can terminate by signaling an exception, and in this case it throws an object of type `CE_Empty`. Note that the throw clause must be placed after `const` in the case of a const method.

A procedure can throw more than one exception, e.g.,

```
int search (array<int> const * a, int x, int &cnt)
    throw (CE_Notin, CE_Dup);
```

states that `search` can throw two exceptions: `CE_Notin` (if `x` is not in `a`), and `CE_Dup` (if `x` occurs in `a` multiple times).

The header of a procedure that doesn't throw any exceptions must also state this fact, by including an empty throw clause:

```
throw ( )
```

For example, all `intset` operations except `choose` should have an empty throw clause as illustrated in Figure 5.1. Requiring the empty throw clause allows us to ensure that specifications indicate exactly what a caller of a procedure can expect, including whether any exceptions are raised; this issue is discussed further in Section 5.2.5.

The specification of a procedure that throws exceptions must explain exactly when this happens, in the effects clause. As before, the effects clause should define the behavior of the procedure for all inputs not ruled out by the requires clause. Since this behavior includes exceptions, the effects section must define what causes the procedure to terminate with each exception, and what its behavior is in each case. Furthermore, if a procedure signals an exception for arguments in some D_i, inputs in D_i should *not* be excluded in the requires clause. Termination by throwing an exception is part of the ordinary behavior of the procedure.

Figure 5.2 shows specifications of `choose` and `search`. Note that the specification of `search` contains a requires clause and that, as usual, its

```
class intset {
public:

  intset( ) throw( ); //default constructor

  void insert(int x) throw( );
  void remove(int x) throw( );
  bool is_in(int x) const throw( );
  int size( ) const throw( );
  int choose( ) const throw (CE_Empty);
  bool rep_ok( ) const throw ( );

}; //end intset
```

Figure 5.1 Revised intset headers.

```
int choose ( ) const throw (CE_Empty);
  // effects: if this is empty throws CE_Empty else returns
  //   an arbitrary member of this.

int search (array<int> const * a, int x, int &cnt) throw (CE_Notin, CE_Dup);
  // requires: a is sorted
  // modifies: cnt
  // effects: if x is not in a, throws CE_Notin; if x occurs in
  //   a multiple times, stores the count of occurrences in cnt
  //   and throws CE_Dup; otherwise returns i such that a[i] = x.
```

Figure 5.2 Some specifications with exceptions.

effects section assumes that the requires clause is satisfied. The domains of `search` are

$D_0 = \{ \langle a, x \rangle \mid x \in$ a exactly once $\}$
$D_1 = \{ \langle a, x \rangle \mid x \notin a \}$
$D_2 = \{ \langle a, x \rangle \mid x \in$ a more than once $\}$

Here the domain subsets are pairs consisting of an array and an integer; in each case the array, a, is sorted.

5.2 The C++ Exception Mechanism

An exception mechanism must provide a way for a procedure to *signal* exceptions, that is, bring them to the attention of the caller. We also need a way for callers to *handle* the exceptions.

5.2.1 Signaling Exceptions

A C++ procedure (function, constructor, or method) can terminate by throwing an exception. To terminate its execution in the "normal" way, a procedure either falls off the end of its body or uses the **return** statement. The **throw** statement is used to terminate a procedure with an exception. For example, in `choose` we might have

```
throw CE_Empty("call of intset::choose with empty set");
```

Here we are throwing an object of type `CE_Empty`; we actually construct this object as part of the throw, by calling one of `CE_Empty`'s constructors. The argument is a string that is intended to indicate why the exception is being raised. As explained in Section 5.2.3, this information can be used by a programmer to figure out what the problem is, in the case where the exception is not handled by the procedure's caller.

If a procedure is being used in a limited context in which you know the exception *will* be handled, you can omit the string, e.g.,

```
throw CE_Empty( );
```

This creates the exception object using the default constructor.

5.2.2 Handling Exceptions

When a procedure terminates with an exception, execution does *not* continue right after the call. Instead control is transferred to some code that handles the exception.

The program controls where the exception is handled by using the `try` statement:

```
try {try_body} catch (declaration) {catch_body}
```

If an exception with the type indicated in the declaration of the catch clause is thrown by the *try_body*, control transfers to the *catch_body*; the variable defined in the *declaration* can be used to access the exception object. For example,

```
try {x = s -> choose( );}
  catch (CE_Empty e) {// in here can use e}
```

The declaration need not provide a variable name; in this case the *catch_body* will not be able to access the exception object. For example,

```
try {x = s → choose( );}
  catch (CE_Empty) {// cannot access exception object in here}
```

Several catch clauses can be attached to the `try` statement so that several different exceptions can be handled. Only exceptions thrown by the *try_body* can be caught in these clauses. In particular, a catch clause can throw another exception, or it can throw the exception it is catching, e.g.,

```
try {...} catch (CE_Foo e) {...; throw e;}
```

However, such an exception cannot be caught by another catch clause of this `try` statement since it was not thrown by the *try_body*. For example:

```
try {x = s → choose( );}
  catch (CE_Empty e) {throw CE_Bound( );}
  catch (CE_bound b) {...}
```

The `CE_Bound` exception thrown by the first catch clause will *not* be caught in the second catch clause.

`Try` statements can be nested (i.e., one of the statements in a *try_body* can be a `try` statement). If an exception thrown by the body of the inner `try` statement is not caught by one of its catch clauses, it can be caught by one of the catch clauses of the outer `try` statement. Thus the `CE_Bound` exception could be caught in a catch clause of an outer `try` statement (assuming there was one), as illustrated in the following code:

```
try { ...;
  try {x = s → choose( );}
    catch (CE_Empty e) {throw CE_Bound( );}
    catch (CE_bound b) {...}
  } // end of outer try
  catch (CE_bound b) {...}
```

Figure 5.3 illustrates various uses of the `try` statement. The procedure `index_sum` removes all elements of `s` that are also in array `a`. If all elements of `s` are in `a`, it returns a count of their occurrences in `a`; if some element of `s` is not in `a`, it throws `CE_Notin`. In the figure, possible signals are shown as comments on each statement. The exceptions from `search` are handled in the inner `try` statement. The `CE_Empty` exception from the call of `choose` is used to terminate the loop.

```
int index_sum (intset *s, array<int> const* a) throw (CE_Notin) {
  int sum = 0;
  int cnt;
  bool not_in = FALSE;
  try {
  while (TRUE) {
    int x = s → choose( );  // can signal CE_Empty
    try {search(a, x, cnt); sum++; } // can signal CE_Notin and CE_Dup
      catch (CE_Dup) {sum = sum + cnt;}
      catch (CE_Notin) {not_in = TRUE; continue;}
      s → remove(x);}} end of while and outer try
    catch (CE_Empty) {if (not_in) throw
          CE_Notin( "missing element in call of index_sum"); }
    return sum;
  } // end index_sum
```

Figure 5.3 Examples of catching exceptions.

5.2.3 Creating Exception Classes

Now we address the question of how to create exception classes such as `CE_Dup`.

All exception classes that we will use have a similar form: they provide two constructors and two methods. The specifications of these operations are given in figure 5.4.

The idea behind our approach to exception classes is the following: when an exception is *not* handled, it is desirable to provide information about what happened. For example, this information is used in our special `main` to provide the meaningful error message, and is also available when an abort occurs due to an unhandled exception. Note that the information in the exception object is typically *not* used by programs that handle exceptions; for handling exceptions the exception type is typically sufficient to determine what to do.

To allow a program like our `main` to be written, we need a way to catch objects from any of our exception classes. We accomplish this by using the *subclass* mechanism; this mechanism and the precise details of how to define our exception classes will be discussed in a later chapter. We define a class `C6170Exception`; all of our exception classes are subclasses of `C6170Exception`.

To make it easy for you to use exceptions, we have predefined a number of them in `6170_except.h`. For example, we have defined `CE_Bounds` and `CE_Empty`. If you need an exception class not contained there, you can create one using a macro `MAKE_EXCEPTION`:

```
// overview: Exception objects are pairs of strings <name, info> where
//   name identifies the exception type, and info provides further information
//   (such as what caused the exception to be raised). Exception
//   objects are immutable and on the stack.

// constructors

  CE_ename ( ); // default constructor
  // effects: initializes this to have name "ename" and info "none".

  CE_ename (string s);
  // effects: initializes this to have name "ename" and info s.

// methods

  string name ( );
  // effects: returns the name of this.

  string info ( );
  // effects: returns the info of this.
```

Figure 5.4 Operations of exception class CE_name.

```
MAKE_EXCEPTION(Foo);
```

creates an exception class named `CE_Foo`, with the constructors and methods shown in Figure 5.4. This macro can be used in the `.h` file of the procedure or class that uses that exception class. Alternatively, you could create a special `.h` file that contains all of your exception classes. That file could include `6170.h`, and then all your files would just include it.

Your programs are unlikely to make use of `C6170Exception` directly. For example, you should not throw objects of this class, nor should you list `C6170Exception` in your throw clauses. You might occasionally catch `C6170Exception`, however; this is useful when you want to catch objects from several different subclasses and treat them all uniformly. For example,

```
try {x = a → fetch(i); y = s → choose(); ...;}
  catch (C6170Exception)
    {// in here can handle both CE_Bounds and CE_Empty}
```

Sometimes we would like exception objects to contain additional information, useful for programs handling the exception. We will discuss how to define such exception classes in a later chapter.

5.2.4 Catching Local Exceptions

Exceptions can be thrown and caught within a procedure to effect local transfers of control. To make such transfers easy, we have defined an exception class `CE_Exit` that can be used for this purpose. The following example uses this exception to terminate a search loop:

```
int x;
int i = 0;
try {
   while (i < a → length( )) {
      x = a → fetch(i);
      if (special(x)) throw CE_Exit();
      i++;}
   x = make_new_one( );} // didn't find one so make one
   catch (CE_Exit e) {}
   // at this point have a suitable value for x
```

Note that the body of the catch clause is empty; in this case the exception is being used solely to transfer control, without having any special work to do.

5.2.5 Unhandled Exceptions

Now we must address the question of what happens if a procedure attempts to throw an exception not listed in its throw clause.

First consider the case of a procedure that does throw some exceptions, e.g., `intset::choose`. If such a procedure attempts to throw an exception that isn't listed in its header, this will cause the program to abort. Note that the attempt could occur in one of two ways: either the code of the procedure contains an explicit throw of a type not listed in the throw clause, or it neglects to catch an exception thrown by some procedure it calls. For example, if `intset::choose` neglected to catch a `CE_Bounds` exception arising from its call of `array::fetch`, this would constitute an attempt by it to terminate with an exception type not listed in its header.

Having an abort occur at this point is a good idea, since there is a serious error in the program. For example, either `intset::choose` isn't implemented correctly, since it should have caught the bounds exception, or it only calls `array::fetch` when the array is not empty, in which case the error is in `array::fetch`.

Now let's consider the second case, a procedure that does not signal exceptions. If the header of such a procedure does *not* contain a throw clause, it is allowed to throw any exception. Thus if `intset::is_in` had the header

```
bool is_in(int x) const;
```

and its implementation did not handle a bounds exception thrown by its call of `array::fetch`, that exception would propagate to its caller.

Having this kind of propagation is unfortunate for several reasons. From a specification viewpoint, it means calls of procedures without throw clauses can result in behavior not captured by the specification. Furthermore, it means that procedures with throw clauses behave differently than those that do not have throw clauses: in one case aborts due to unhandled exceptions happen, while in the other case they do not. A final point is that, as mentioned above, unintended signals almost always indicate serious program errors; in such a case you want the program to abort, since otherwise it might do damage, e.g., to online data bases.

It may seem that the using code can avoid the damage by handling the problem later, e.g., at the top level, as we do in our special definition of `main`. However, there is a danger that the exception will be "captured" before it reaches that point. For example, consider the following code:

```
try {x = a → fetch(i);
  if (s → is_in(x)) ...;}
catch (CE_Bounds e) {// handle bounds exception from call of fetch }
```

The problem here is that the bounds exception unintentionally signaled by `is_in` is mistakenly caught and handled in the catch clause — and then the program continues its execution.

These problems can be avoided by requiring that all procedure headers contain a throw clause. This is why we require the clause. The rule is that the throw clause is either empty, or it lists specific exception types such as `CE_Bounds`. It *should not* list the type `C6170Exception`, since this would allow it to throw any of our exception types as discussed in Section 5.2.3.

When an abort occurs due to an unhandled exception, you will get an error message that will allow you to figure out what exception wasn't handled and where the problem arose.

5.3 Using Exceptions in Programs

When implementing a procedure with exceptions, the programmer's job, as always, is to provide the behavior defined by the specification. If this behavior includes exceptions, the program must signal the proper exceptions at the proper times with the meaning as described in the specification. To accomplish this task, the program may need to handle exceptions that are thrown by procedures it calls.

Exceptions can be handled in two different ways. Sometimes an exception is *propagated* up another level; that is, the caller also terminates by signaling an exception. Before propagating the exception, the caller may do some local processing. Such processing is sometimes needed to satisfy the caller's specification. For example, an operation of a data type must ensure that objects satisfy the rep invariant before it terminates.

A second possibility is that the caller *masks* the exception, that is, handles the exception itself. For example, a procedure might fetch the elements of an array until there are none left. At this point the call to the `fetch` method throws the `CE_Bounds` exception. However, to the caller this simply means that all elements of the array have been processed and it is time to finish up.

This section illustrates the use of exceptions by showing the implementation of `intset`. The implementation, given in figure 5.5, uses the same rep as before (Figure 15.3). Note that we have changed `get_index` to signal `CE_Notin` when `x` is not in `this`. Also note that none of the catch clauses makes use of the exception object; instead they simply make use of the information that an exception of the listed type has occurred.

The implementation of `get_index` uses the `CE_Bounds` exception signaled by the array `fetch` operation to indicate that it has checked all the elements of the array. When it catches this signal, it throws `CE_Notin` — an example of exception propagation. The implementation of `choose` provides another example of propagation. Note that `get_index` does not provide a meaningful `info` string for the exception; this is acceptable because we can prove the exception is always handled.

This `CE_Notin` exception thrown by `get_index` is caught in `remove`, which simply masks the exception and returns normally. The implementations of `is_in` and `insert` also mask the exception.

The implementation of `remove` contains an example of possibly unhandled exceptions. If `get_index` returns normally, we know the index is within bounds and the array is not empty. Therefore we do not bother to catch the `CE_Bound` exception from the calls of array methods `store`, `top`, and `remove`. If these exceptions should occur, the array implementation is broken. Therefore we want the program to halt; this is accomplished because `CE_Bound` is not listed in the throw clause of `intset::remove`.

In our sample implementation, we did not try to avoid raising exceptions; instead we used them to control program flow. Exceptions can improve performance by reducing the number of calls *provided* the exception mechanism is implemented efficiently. For example, the `choose` implementation in figure 5.5 uses the `CE_Bound` exception to avoid calling the array `length` method. Not making extra calls saves work — both the cost of the call and return, and the cost of executing the called procedure. On the other hand, handling exceptions may be expensive, depending on the

```
intset::intset ( ) throw ( ) {
  els = new array<int>( );
  } // end

void intset::insert (int x) throw ( ) {
  try {get_index(x);} catch (CE_Notin) { els → append(x); }
  } // end insert

void intset::remove (int x) throw ( ) {
  int i;
  try {i = get_index(x);} catch (CE_Notin) {return;}
  els → store(i, els → top( ));
  els → remove( );
  } // end remove

bool intset::is_in (int x) const throw ( ) {
  try {get_index(x); return TRUE;}
    catch (CE_Notin) {return FALSE;}
  } // end is_in

int intset::get_index (int x) const throw (CE_Notin) {
  for (int i = 0; i < els → length( ); i++)
    if (x == els → fetch(i))  return i;
  throw CE_Notin( );
  } // end get_index

int intset::size ( ) const throw ( ) {
  return els → length( );
  } // end size

int intset::choose ( ) const throw (CE_Empty) {
  try {return els → fetch(0);}
    catch (CE_Bounds)
      {throw CE_Empty("call of intset::choose with empty set");}
  } // end choose

bool intset::rep_ok ( ) const throw ( ) {
 for (int i = 0; i < els → length( ); i++) {
   int x = els → fetch(i); // check for dups of x
   int j = i + 1;
   try {while (TRUE)
       if (x == els → fetch(j)) return FALSE; else j++;}
       catch (CE_Bounds) {} } // end for
 return TRUE;
 }  // end rep_ok
```

Figure 5.5 Implementation of intset using exceptions.

cleverness of their implementation. So there are costs either way. Probably it is not a good idea to uses exceptions to avoid a single call of just one, inexpensive procedure, but otherwise it is good to use exceptions.

5.4 Design Issues

Exceptions should be used to eliminate most constraints listed in requires clauses. The requires clause should remain only for efficiency reasons or if the context of use is so limited that we can be sure the constraint is satisfied. For example, `search` might still require that the array be sorted, since it can then be implemented much more efficiently. Also, the `merge` procedure used in merge sort (figure 3.7) should require its arguments to be sorted, for efficiency and also because the context of use is so limited. Of course, as discussed earlier, when implementing a procedure a requires clause it is best to check the requirement *if possible* and cause the program to halt if it is not satisfied. For example, in `merge` (figure 3.8) we could compare array elements as we look at them to see whether they are ordered properly and cause a failure if they are not. Such a test costs little, but improves the robustness of the program considerably.

Exceptions should also be used to avoid encoding information in ordinary results. For example, instead of returning -1 if the element is not in the array, `search` signals an exception. It is better to convey this information with an exception, since the result returned in this case cannot be used like a regular result. By using an exception, we make it easy to distinguish this result from a regular one, thus avoiding a potential error. Using exceptions is particularly important for procedures intended for general use. For example, since `get_index` is used in a limited context in might be acceptable for it to return -1 when the element isn't in the `els` array.

In addition, exceptions can be used to distinguish among classes of legal inputs in cases where distinctions might not be considered in the absence of exceptions. For example, consider the behavior of `intset::insert` when the object to be inserted is already in the set. Perhaps this situation should cause an exception to be raised? The decision should depend on expectations about the needs of users of the abstraction. Thus the current specification of `insert` is right provided users do not want to be informed about the re-insertion. Another alternative is to provide two `insert` methods (with different names!), one of which signals in this case and the other of which does not. Then users can choose which one they want to use.

Exceptions provide information that can usually be obtained by calling operations directly. For example, the fact that an `intset` is empty can be learned either by calling `choose` and having it signal or by calling `size`. Such redundancy is not required. For example, there might be a type in which the only way to find out whether an object is empty is through an exception. Such a type is sensible, however, only when it is expensive to do the empty test, since otherwise users might prefer to call an `empty` method to handling the exception.

Not all errors give rise to exceptions. Consider an erroneous record in a large input file, where it is possible to continue processing the file by skipping that record. In such a case, it may be appropriate to inform a person (not a program) about the error. Exceptions are a mechanism for communication among programs, not for communication from programs to people. To communicate with people, an error message might be written on some output device. Note, by the way, that what is done with the error is defined in the abstraction's specification.

Exceptions are not always associated with errors, though. For some abstractions there may be more than one kind of normal behavior, and here exceptions are a convenient tool. They provide a means for allowing several kinds of behavior and enabling the caller to distinguish among the different cases.

For example, a `lookup` method on a symbol table has a twofold purpose. Given an identifier, it determines whether a declaration for this identifier has already been processed and, if it has, returns information about the identifier. A header for this method might be

```
info* lookup (string s) const throw (CE_Notin);
```

Here we have chosen to treat the case in which the declaration exists as the normal case, but we could easily have made the opposite choice. One rationale for this choice is efficiency. It is likely that catching exceptions will be more expensive than normal returns. Therefore the case that is expected to occur most frequently should be considered the normal case. When a distinction is made between cases based on performance considerations, it is obvious that no notion of "error" can be attached to the exception case.

The relationship of modification of arguments to termination in an exception condition is worth further discussion. The modifies section of a specification indicates that an argument may be modified, but does not say when this will happen. If there are exceptions, it is likely that the modification will happen only for some of them. Exactly what happens must be described in the effects section. Modifications must be described explicitly in each case where they occur; if no modifications are described, this means none happens. For example, suppose we had the following method for `intset`

```
void insert_max ( int x) throw (CE_Small);
  // modifies: this
  // effects: if this contains any element larger than x throws CE_Small
  //   else adds x to this
```

Since no modification is described when `insert_max` signals `CE_Small`, we know that `this` is modified only when `insert_max` returns normally.

Note that it is possible for any procedure to cause the program to abort, yet no mention of this fact is made in the specification. A specification describes the behavior of a procedure when it is working and when its arguments satisfy the requires clause. Failure occurs when the procedure is broken and no longer meets its specification or when the arguments are improper.

The exceptions signaled by operations of data types are related to the type as a whole and the same exception name should be used for similar situations. For example, array operations `fetch`, `store`, and `top` all signal `CE_Bounds` when a call is made on an empty array.

5.5 Very Robust Programs

The approach to unhandled exceptions described above causes the program to fail as soon as an error, in the form of an unhandled exception, occurs. This is a satisfactory approach only under the assumption that a person is available to fix the problem.

A very robust program, with redundant ways of doing things, may be able to recover from the error itself, without a person being involved. Typically the program would do a clean shut-down, followed by a restart.

To write such code it is necessary to propagate exceptions to the shut-down/restart handler. This requires a different approach to exceptions than we have been using — and one that is more work. In this case every procedure must explicitly check for all unhandled exceptions and, if they occur, throw a special exception that the shut-down/restart handler will catch.

One way to write robust code is the following: Every header must contain a throw clause that mentions the special exception, e.g.,

```
throw (CE_Failure,...)
```

Also every procedure body must be a try statement with a catch clause that catches every exception, e.g.,

```
try {//procedure body}
  catch (C6170Exception e) {throw CE_Failure( );}
```

The `CE_Failure` exception may have some arguments that can be used by the shut-down/restart code to pinpoint the source of the problem.

5.6 Summary

In this chapter, we have generalized procedures to include exceptions. Exceptions are needed in robust programs because they provide a way to respond to errors. If an argument is not what is expected, a procedure can notify the caller of this fact rather than simply fail. Since this notification is distinct from the normal case, the caller cannot confuse the two.

Exceptions are introduced when procedures are designed. Most procedures should be defined over the entire domain; exceptions are used to take care of situations in which the "normal" behavior cannot happen. Partial procedures are suitable only when it is either too expensive or not possible to check the condition, or when the procedure is used in a limited context in which it can be proved that all calls have proper arguments. In either case, when the procedures are implemented, it is a good idea to practice defensive programming, checking in as many cases as possible for inputs outside the accepted domain subset.

In implementing a procedure, the programmer must ensure that it terminates as specified in all situations. Only exceptions permitted by the specification should be signaled, and each should be signaled for the right reason. In doing the implementation, it is reasonable to use exceptions of called procedures and to mask or propagate as appropriate.

Further Reading

Goodenough, John B., 1975. Exception handling: issues and a proposed notation. *Communications of the ACM* 18(12): 683–696.

Liskov, Barbara, H., and Alan Snyder, 1979. Exception handling in CLU. *IEEE Transactions on Software Engineering* SE-5(6): 546–558.

Exercises

5.1 Modify the `poly` data abstraction defined in chapter 4 (figure 4.3) to take advantage of exceptions. Specify the new abstraction and then implement it.

5.2 Exercise 7 in chapter 4 concerned a bounded queue abstraction. Redefine this abstraction using exceptions and provide an implementation of the modified abstraction.

5.3 A map is a table that associates integer elements with strings. Each string is mapped to at most one associated element. Maps have a constructor to create an empty map; the `insert` method adds a string and its associated element, the `change` method changes the element associated with a string; the `remove` method deletes a string and its associated element; and the `eval` methods looks up the element associated with a string. Specify maps, being careful to include appropriate exceptions and to provide extra operations as needed for adequacy. Then implement your specification and provide the rep invariant and abstraction function. Your implementation must be efficient: The running time of `eval` must be much less than Order(n), where n is the number of entries in the map.

5.4 Discuss the adequacy of the map abstraction defined in exercise 3. Is it possible always to avoid having calls of map operations raise exceptions, for example, by calling other appropriate map operations? Discuss whether the decisions made in this regard were right or wrong.

5.5 A procedure to compute the minimum value of an array might require a nonempty array, return the smallest integer if the array is empty, or signal an exception if the array is empty. Discuss which alternative is best.

5.6 Instead of computing the minimum value, suppose that the procedure of exercise 5 were adding the elements of the array. Would this change your idea of which alternative is best?

Chapter 6

More General Abstractions

The procedures and types discussed previously are somewhat limited in the kinds of arguments they accept. For example, the `sort` procedure works only for arrays of integers. If it applied to other kinds of arrays, such as arrays of characters, strings, or reals, it would be more generally useful. We can achieve this extra generality, which comes from carrying abstraction by parameterization further than we have done so far, by using data types as parameters.

Types are clearly useful as parameters. Evidence for this is the fact that many built-in data types in programming languages, such as arrays, records, and procedures, are parameterized by types. It is true in general that whatever is useful at the level of a programming language is probably also useful for its users, and there is no doubt that type parameters fall into this category.

When types are used as parameters, some parameter values may not be meaningful. For example, arrays can be sorted only if the elements belong to a type that is totally ordered; `remove_dupls` can be called on an array only if there is a way to test the equality of the array elements. We need a way to capture such constraints.

In this chapter, we discuss how to generalize abstractions to be *polymorphic*: to work over many types of inputs. We being by describing the C++ *template* mechanism, which allows polymorphic abstractions to be defined. Then we discuss how to specify and implement such abstractions.

```
template <class T>
  int search (array<T> * a, T x) throw (CE_Notin);
  // effects: if x is not in a, throws CE_Notin else returns i
  //   such that a[i] = x.

template <class T>
  int search (array<T> * a, T x) throw (CE_Notin) {
     T el;
     for (int i = 0; i < a->length(); i++) {
       el = a -> fetch(i); // S1
       if (el == x) return i;  } // S2
    throw CE_Notin("call of search when x is not in");
  } // end search
```

Figure 6.1 Parameterized search.

6.1 The Template Mechanism

The C++ template mechanism can be used to define parameterized procedures and classes. Although parameters can be of various types, e.g., ints, we will focus on the most interesting case in which the parameters themselves are types. When used this way, templates permit the definition of polymorphic abstractions.

This section describes the C++ template mechanism using a simple example. This example will illustrate a difficulty with defining constraints. Then we will explain how to overcome this difficulty — by using functions as arguments.

As an example of where polymorphism should be useful, consider the `search` routine discussed previously. This procedure ought to be useful for arrays containing other types of elements than just integers.

Figure 6.1 shows a specification and implementation of a generalized `search` procedure that works on arbitrary types. The information inside the brackets `<` and `>` names the parameters and indicates their types. In this case the single parameter is named `T` and it is a type; this is indicated by declaring it to be a `class`.

Careful examination of this code indicates several places where calls are made on methods of `T` objects. In the line labeled *S1*, the `T` object returned by the call of `fetch` is assigned to `el`; thus we have a call on the `operator =` method of `T`. In the line labeled *S2*, `el` and `x` are compared by calling `operator ==` of `T`.

For a call of `search` to run, these calls on methods of `T` must be resolved somehow. The way this works is as follows. Whenever code that calls `search` is compiled, the C++ compiler *deduces* what type `T` must be

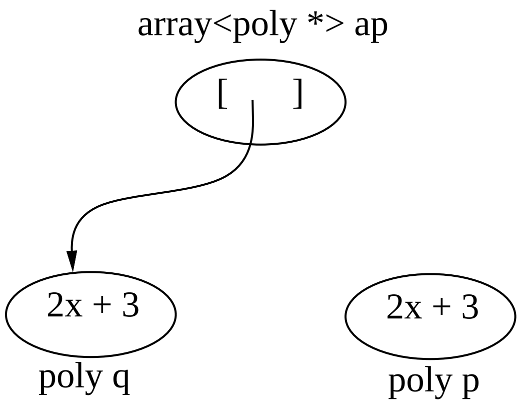

Figure 6.2 Poly example.

for this particular call (and sometimes it makes the wrong deduction!). For example, consider the call `search(z,3)` where `z` is an `array<int>*`; then the compiler knows that `T` is `int` in this call. Next it substitutes the particular type for each occurrence of `T` in the code. Thus if it is substituting `int` for `T` it rewrites the above code, obtaining the non-parameterized version of `search` that we saw earlier. Finally it compiles the new code. In doing so it selects the methods for the substituted type (e.g., `int`); if the substituted type doesn't have one of the used methods, there is a compile-time error.

Now consider a call `search(ap,p)` where `ap` is an `array<poly*>` and `p` is a `poly`. In this case `T` is `poly*`, i.e., it is a pointer to a `poly`. When the compiler does the substitution, the result will be calls to `operator =` and `operator ==` of pointers. The first of these calls, to `operator =` is what we want: it makes `el` refer to the `poly` just fetched from the array. But the second call is *wrong*! What we want here is a match if `x` and `el` both represent the same abstract `poly` object; instead we only have a match if they are the exact same object. Figure 6.2 illustrates a situation in which a call to `search` will throw `CE_Notin` when it actually ought to return 0.

We could make `search` work properly for `poly` by changing its definition; in this case, line *S2y* would become:

```
if (el → equal(x)) return i;
```

However, now `search` won't work properly when `T` is `int`! It may appear that the problem here is that in the case of `int` the method is called "`operator ==`," while in the case of `poly` it is called "`equal`." Unfortunately the problem is more fundamental than this: it has to do not with the fact that the syntax for calling methods is different for objects on the stack than for objects on the heap. Nevertheless, we would like to provide a single parameterized definition that works in either case!

We will solve this problem by passing procedures as arguments. The procedure arguments will encapsulate information about whether objects of the parameter type are on the stack or heap; this will allow the parameterized abstraction to ignore these details. It will be up to the caller to pass a procedure that does the "right" thing. The specification of the parameterized abstraction will state what the "right" thing is, i.e., how the procedure argument must behave; this information will be given in the `requires` clause.

A polymorphic abstraction is not supposed to use methods of objects of the parameter types except for the copy constructor and `operator =`. If other methods are needed, these are almost always accessed indirectly, by calling procedures passed as arguments. Occasionally some other method must be used (we will see an example in Section 6.2.2). The specification describes the required behavior of the argument procedures and the other methods, in the requires clause. Nothing is said in the specification about the behavior of `operator =` and the copy constructor, since these are assumed to be defined and do the right thing for all types: they make `this` be *identical* to their argument. Section 6.5 discusses what it means for objects to be identical.

The combination of the procedure arguments and the requires clause specify *constraints* on both using and implementing code. Users must provide the required procedures, and must make sure they behave as indicated in the requires clause; if any additional methods are used, the user must check that they also behave as indicated in the requires clause. Implementations of polymorphic abstractions interact with objects of the parameter type only via `operator =` and the copy constructor, explicitly mentioned methods, and by calls of the procedure arguments. If the implementor follows the rules, we can guarantee that no compile-time errors will arise from a use of the abstraction that satisfies the constraints. If the user follows the rules, we can guarantee that the abstraction will behave as indicated in its specification.

6.2 Polymorphic Procedures

Specifications for some polymorphic procedures are given in Figure 6.3. The parameterized `search` procedure takes an extra argument, the `eq` routine that it will use to find where `x` is in `a`. Note that the requires clause states the expected behavior of `eq` and the effects clause indicates how `eq` affects the behavior of `search`. The specification of `sort` is similar except that in this case we need a routine (`lt`) to determine which is the smaller of two array elements.

```
template <class T>
  int search (array<T> const * a, T x, bool (eq) (T, T)) throw (CE_Notin);
    // requires: eq defines an equality relation on T
    // effects: if x is not in a, throws CE_Notin; otherwise returns i
    //   such that a[i] = x, where equality is tested using eq.

template <class T>
  void sort (array<T> * a, bool (lt) (T, T)) throw ( );
    // requires: lt defines a total order on T
    // modifies: a
    // effects: sorts a into ascending order as determined by lt.

template <class S, class T>
  array<T>* map_array(array<S>* const a, T (m) (S)) throw ( );
    // modifies: elements of a if m modifies them.
    // effects: returns a new array b = [m(a[0]),...,m(a[a.length−1])].
```

Figure 6.3 Specifications of polymorphic procedures.

Note that for both `search` and `sort`, there is a potential for modifying `T` objects (since they might be mutable). However, the specifications make it clear that this will not happen since they do not indicate any modifications to these objects. A further point is that implementations will access these objects only by calling their routine arguments (plus they might use `operator =` or the copy constructor). Therefore they have no way to modify objects *unless* these routines do modifications. As mentioned, `operator =` and the copy constructor are assumed to not do modifications (except to make `this` identical to their argument). Modifications caused by the routine arguments or other required methods would be described explicitly in the specification.

The `map_array` abstraction is different from the other two: it has two parameters, and it does not constrain its argument `m`. In fact, `m` is not intended to be a particular operation of objects of type `S`. Rather it is just a procedure chosen by the user to map objects of type `S` to objects of type `T`. It is even acceptable for `m` to modify its arguments, as indicated in the modifies clause of `map_array`.

6.2.1 Implementing Polymorphic Procedures

Figure 6.4 gives implementations of polymorphic `search` and `map_array`. These implementations are straightforward. They are just like the implementations of non-parameterized versions of the same functionality except that they call the routine arguments rather than using methods. Thus the implementation of `search` calls `eq` where the implementation of the non-

```
template <class T>
  int search (array<T> const * a, T x, bool (eq) (T,T)) throw (CE_Notin) {
     for (int i = 0; i < a→length(); i++) {
       T el = a → fetch(i);
       if (eq(el,x)) return i;} // call of eq
     throw CE_Notin("call of search when x is not in");
  } // end search

template <class S, class T>
  array<T>* map_array(array<S> const *a, T (m) (S)) throw ( ) {
     array<T> * res = new array<T>( );
     for (int i = 0; i < a → length(); i++)
       res → append(m(a→fetch(i)));
     return res;
  } // end map_array
```

Figure 6.4 Implementations of polymorphic procedures.

polymorphic version of `search` called a method (e.g., `operator ==` in the case of `search` for an `array<int>`).

Implementations of polymorphic procedures must *not* call any T methods except for `operator =` and the copy constructor unless the methods are required in the requires clause. C++ will not find an error, however, if you violate the rule. Instead, the error will occur on some use of the polymorphic procedure, when it is called with a type that is missing the method. The programmer using the polymorphic procedure will get an unpleasant surprise if this happens, since it will be evident that the specification is not satisfied.

In fact, implementations of parameterized abstractions are only compiled when they are used, after the substitution of the actual types for the parameters. Their implementations are put in a `.t` file instead of a `.cpp` file. Furthermore, instead of having the `.t` file include the associated `.h` file, the `.h` file includes the `.t` file *at its end*. When you compile a `.cpp` file that uses the abstraction (and thus includes its associated `.h` file), the parameterized abstraction will be compiled.

6.2.2 Using Polymorphic Procedures

Polymorphic procedures are used just by calling them. Here is an example of a call to `search`:

```
array<poly*> *b;
// code here adds some elements to b
int i = search(b, new poly(3, 2), poly_equal);
```

Here poly_equal is a function that tests the equality of two polys by calling the poly::equal method:

```
bool poly_equal(poly * p, poly * q) throw ( ) {
  return p → equal(q);
} // end poly_equal
```

For the call on a polymorphic procedure to work correctly, it is not sufficient to pass arbitrary procedures that match the signatures given in the header. Thus search won't work right unless the procedure passed to it provides a meaningful equality test for elements of type T. Similarly, sort won't work right unless the procedure passed to it provides a meaningful ordering on T objects. Therefore it is important to pass routines that meet the constraints stated in the requires clause. Also, if the polymorphic routine requires any T methods, it is important to use it only if those methods have the required behavior.

To call a polymorphic routine, you need to provide the procedures it requires. Of course you can always implement these procedures. Typically they are very simple, like the poly_equal function shown above. However, a good methodology is to define the needed procedures when data abstractions are defined. We can expect a need for certain routines — to test for equality, or, in the case of a numeric type, to test for "less-than", etc. These routines can be declared as related routines of a new type, and implemented at the same time the type is implemented. If the type is adequate, it should not be necessary for them to be friends. Providing these related routines has an additional advantage (over making it convenient to use the type): it allows the definer of the type to provide the right definition for the routines. This makes it more likely that when the routines are passed as arguments to polymorphic abstractions, they will meet the requirements.

For example, the following related routines might be provided for poly.

```
bool poly_eq(poly *p, *q) throw ( );
bool poly_lt(poly *p, *q) throw ( );
bool poly_gt(poly *p, *q) throw ( );
```

It will not be necessary to write out the specifications for such routines, since their meanings are standard. For example, poly_lt returns true if p < q.

Related routines are provided for the built-in types and strings; you can get these routines by including 6170.h. We provide T_eq, T_lt, and T_gt for T being int, char, bool,and string. Related routines for arrays are discussed in Section 6.5.

Figure 6.5 Two similar (but unequal) intsets.

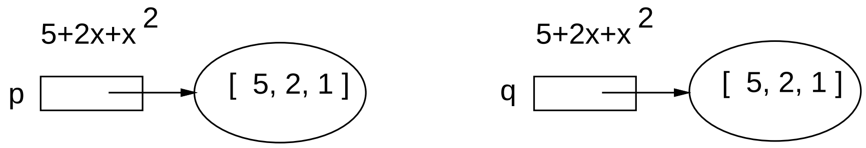

Figure 6.6 Two poly objects with the same state.

6.3 The Meaning of Equality

To provide related routines that test for equality, we need to understand what the proper definition of equality ought to be.

We would like to consider two objects to be equal if and only if they are *identical*: *it is not possible to distinguish between them either now or in the future.* The distinguishing occurs by calling methods on the objects. In other words, whatever methods a program calls on these objects after discovering they are equal, it won't be able to notice a difference between them.

Two mutable objects can be considered equal if and only if they are the very same object (i.e., if the pointers that refer to them are `==`). The reason for this strong definition is that otherwise a program could distinguish the objects by calling a mutator and then an observer. For example, consider the `intsets` `x` and `y` shown in Figure 6.5. These `intsets` both have the same state {`1, 4, 7`}, but they are not equal: if `x` is modified by inserting 3 into it, it can now be distinguished from `y`.

Immutable objects are equal if they are the very same object, but also if they are different objects that contain the same state. If two immutable objects have the same state, they cannot be distinguished now or in the future, since their state doesn't change. Thus the two `poly` objects shown in Figure 6.6 are equal.

For mutable objects, there is a weaker notion of equality that we refer to as *similarity.* Two mutable objects are *similar* if it is not possible to distinguish between them right now, even though we might be able to distinguish between them later. Thus the `intsets` `x` and `y` in Figure 6.5 are similar (but not equal). Note that the sets are similar even though their reps differ. Equality for immutable types, and similarity for mutable types, are abstract notions: two objects are equal (similar) if they represent the same abstract object even though their reps are different.

For immutable type `T`, a single related equality operator `T_eq` ought to be provided. For mutable type `S`, both equality (`S_eq`) and similarity (`S_sim`) can be provided. Thus `intset` might provide the following related routines:

```
bool intset_eq(intset const *s, intset const *t) throw( ); // same object
bool intset_sim(intset const *s, intset const *t) throw( ); // same state
```

Having an "similarity" test like `intset_sim` does not make sense for immutable types like `poly`. The implementation of `intset_eq` is simply:

```
bool intset_eq(intset const *s, intset const *t) throw( ) {
  return s == t; }
```

The implementation of `intset_sim` must go through the two sets to see whether they contain the same `ints`. (Note that to implement `intset_sim` efficiently, the routine needs to be a friend of `intset`. The reason is that `intset` as defined so far is *not* adequate. We will correct this problem in the chapter on iterators.)

6.4 Polymorphic Data Abstractions

In addition to parameterized procedures, it is also possible to define parameterized data abstractions. For example, the notion behind `intset` can easily be generalized to allow other kinds of set elements.

The specification of the `set` abstraction is given in figure 6.7. Since sets cannot contain duplicate elements, the type needs to receive an equality operator from its user. This operator is passed as an argument to the constructor. Since the requirement about the meaning of the equality routine applies to the entire data abstraction, rather than just the constructor, we have stated it in a requires clause in the overview section, and the requirement is not repeated in the specification of the constructors.

Note that the private section disallows the use of the default constructor. It doesn't make sense to have a default constructor for `sets` (since then the implementation wouldn't know how to do the equality test).

6.4.1 Using Polymorphic Data Abstractions

To use a parameterized data abstraction it is necessary to *instantiate* it explicitly, by providing actual values for the parameters. For example `set<int>` and `set<poly*>` are instantiations. Here are examples of instantiations together with creations of `set` objects:

```
template <class T>
class set {
public:
// Overview: sets are mutable, unbounded sets containing elements of type T.
//      A typical set is {x_1, ..., x_n}. set is a heap abstraction.
// Requires: T has an equality operator used to determine set membership.

// constructors:

   set (bool (eq) (T, T)) throw ( );
      // effects: Initializes this to be empty and to use eq
      //      to test for element equality.

// methods:

   void insert (T x) throw ( );
      // modifies: this
      // effects: Adds x to the elements of this, i.e., s_post = s ⋃ x

   void remove (T x) throw ( );
      // modifies: this
      // effects: Removes x from this, i.e., s_post = s - x

   bool is_in (T x) const throw( );
      // effects: If x is in this returns true else returns false.

   int size ( ) const throw ( );
      // effects: Returns the cardinality of this.

   T choose ( ) const throw (CE_Empty);
      // effects: If this is empty throws CE_Empty else returns an
      //    arbitrary element of this.

   bool rep_ok ( ) const throw ( );

private:
   void operator = (set<T> const &); //disallowed
   set<T> ( ); // disallowed
   set<T> (set<T> const &); // disallowed

}; // end set
```

Figure 6.7 Specification of the set data abstraction.

```
template <class T>
class set {
public:

// as in Figure 6.7

private:

   array<T> *els; // the set elements without duplicates
   bool (eqp) (T, T); // the equality operator

   int get_index (T x) const throw ( );
     // effects: If x is in this returns i such that els[i] = x, else returns -1

   void operator = (set<T> const &); //disallowed
   set<T> ( ); // disallowed
   set<T> (set<T> const &); // disallowed

}; // end set
```

Figure 6.8 Private section of the set class.

```
set<int> * s = new set<int>(int_eq);
set<poly*> * sp = new set<poly*>(poly_eq);
```

An instantiation defines a type suitable for use in declarations. The implementation of the type is obtained by rewriting the class declaration and implementation with the actual parameters replacing the formals.

Note that instantiation defines a type, but the procedure arguments are supplied only when objects of the type are created. It is important that the same procedure be passed in for each constructor call. For example, suppose we also created a `set<poly*>` object using a procedure that found two `polys` to be equal if they had the same degree. That set object would behave quite differently than one based on `poly_eq`!

6.4.2 Implementing Polymorphic Data Abstractions

Figure 6.8 shows the private section of the `set` class, using the same rep as the one shown in Figures 4.5 and 15.3, except that the rep now has a second instance variable — to hold the equality operator. The abstraction function and rep invariant are the same as before, except that now `eqp` is used to determine the matching of elements in the rep invariant:

```
// the rep invariant is:
//   for all 0 <= i < j < els.length (!eqp(els[i],els[j]))
```

Figure 6.9 shows an implementation of `set`. Note that every method must be implemented by a template. Note how the constructor stores the passed procedure in the rep, and how that procedure is used in the other methods (e.g., in `get_index`).

6.5 More on Equality

Since sets are mutable, the related routine `set_eq` must return true only if the two sets are the very same object:

```
template <class T>
  bool set_eq (set<T> const *s, set<T> const *t) throw () {
    return s == t; }
```

Note that we need to provide the routine using a template, since we want a version of `set_eq` for each actual parameter.

We might also want a routine to test for similarity. The problem here is that if the set contains mutable objects, or stack objects like `int`, then we want to use `operator ==` to do the test. But if the set contains immutable, heap objects, we want to call the `equal` method. The following routine will do the right thing:

```
template <class T>
  bool set_test(set<T>const *s, set<T> const *t,
        bool (eq)(T, T, bool (T, T))) throw ( );
    // requires: eq is an equality test on T
    // effects: returns true if s and t contain the same elements
    //   in the same order, as determined by calling eq
```

Note that this routine will provide either meaning for similarity, depending on what routine it gets as an argument. However `set_test` isn't suitable for use as an argument to polymorphic abstractions since it takes three arguments and they expect a routine that takes two arguments.

A practical way to proceed is to provide a similarity routine that returns true if the two sets are equal using `operator ==` to compare the elements:

```
template<class T>
  bool set_sim ( set<T> const *s, set<T> const *t) throw ( );
  // requires: uses operator == for T
  // effects: returns true if s and t contain the same objects,
  //   where objects are compared using operator == for T
```

```
template <class T>
  set<T>::set (bool (eq) (T, T)) throw ( ) {
    els = new array<T>( );
    eqp = eq; } // end

template <class T>
  void set<T>::insert (T x) throw ( ) {
    if (get_index(x) < 0) els → append(x);
    } // end insert

template <class T>
  void set<T>::remove (T x) throw ( ) {
    int i = get_index(x);
    if (i ≥ 0) {els → store(i, els → top( )); els → remove( );}
    } // end remove

template <class T>
  bool set<T>::is_in (T x) const throw ( ) {
    return (get_index(x) ≥ 0);
    } // end is_in

template <class T>
  int set<T>::get_index (T x) const throw ( ) {
    for (int i = 0; i < els → length( ); i++)
      if (eqp(x, els → fetch(i))) return i;
    return −1;
    } // end get_index

template <class T>
  int set<T>::size ( ) const throw ( ) {
    return els → length( );
    } // end size

template <class T>
  T set<T>::choose ( ) const throw (CE_Empty) {
    if (els → length( ) == 0) throw CE_Empty("call of set::choose on empty set");
    return els → fetch(0);
    } // end choose

template <class T>
  bool set::rep_ok( ) const throw ( ) {
   for (int i = 0; i < els → length( ); i++) {
     T x = els → fetch(i); // check for dups of x
     for (int j = i + 1; j < els → length ( ); j++)
        if (eqp(x, els → fetch(j))) return false; }
   return true;
   } // end rep_ok
```

Figure 6.9 Implementation of set.

This kind of similarity routine is sometimes referred to as *shallow* similarity since it doesn't look at the structure of the contained objects. For example, it may find two sets containing equal `poly` objects to be unequal — because they contain different `poly` objects.

We provide equality and shallow similarity routines for arrays:

```
template <class T>
  bool array_eq (array<T> * const a, array<T> *b) const throw ( );
    // effects: returns true if a and b are the same object.

template <class T>
  bool array_sim (array<T> const * a, array<T> *b) const throw ( );
    // requires: operator == for T
    // effects: returns true if a and b contain the same elements
    //   in the same order, as determined by operator == for T
```

For polymorphic types like `set` that depend on an equality operator, we can also provide another similarity routine, one that uses the `eqp` operator to implement *deep similarity* (i.e., similarity based on the structure of the objects). This operator needs to be a friend of set, since it will access the rep of its set object to get the equality operator:

```
friend bool set_deep_sim (set<T> const *s, set<T> const *t) throw ( );
  // effects: returns true if s and t contain the same elements
  //   as determined using the equality operator for s
```

Note that if `s` and `t` contain different equality operators the result here may be surprising! Note also that this approach is not possible for polymorphic types such as arrays that do not depend on an equality operator.

One final point: if the polymorphic type is immutable, then the equality operator must compare the elements. For example, suppose we generalized the `list` abstraction of Chapter 4 (see Figure 4.14). Then we could provide:

```
template<class T>
  bool list_eq(list<T> *x, list<T> *y) throw ( );
  // requires: uses operator == of T
  // effects: returns true if and only if x and y contain equal objects in
  //   the same order, as determined using operator == of T
```

This operator will not be what you want for a `list<poly *>`: if two such lists are determined to be equal, they really are, but they might still be equal even when `list_eq` returns false. However, since lists do not require an equality operator, there is no way to provide an equality routine that uses such an operator. In such a situation, users must define their own equality operator when they need one. The operator would *not* be polymorphic, but instead would be defined specifically for the particular element type. Thus a user might define a `list_poly_eq` if he or she needed to compare `list<poly>` objects.

6.6 Ordered Lists

This section presents another example of a polymorphic data abstraction, `ordered_list`. Ordered lists are mutable objects that contain elements of a parameter type. Their type parameter must provide both an ordering on its elements and an equality test.

The specification of `ordered_list` is given in Figure 6.10. The comparison and equality routines are arguments to the constructor; in this way methods of `ordered_list` objects will have access to the routines. Note that we have disallowed the default constructor as well as the copy constructor and `operator =`.

Note that `add_el` throws an exception when the element is already in the ordered list. (The situation with `rem_el` is similar.) This choice reflects a belief that users will want to know about this, and it would be expensive for them to have to call `is_in` to find out about it. If we thought users didn't want to know, we might have defined the methods to do nothing, e.g., `add_el` would simply not add the new element to the ordered list if it were already there. Another possible approach is to provide more than one form for these methods, e.g., in addition to `add_el` there might also be an `insert` method that does nothing if the element is already in the list. This would provide a more useful abstraction since it could support a wider variety of users.

The implementation of `ordered_list` uses a sorted tree. The idea is that each node of the tree contains a value, and also two subnodes, one on the left and one on the right. The two subnodes are themselves `ordered_lists` and therefore the rep is recursive. The tree is sorted so that all the values in the left subnode are less than the value in the parent node, and all values in the right subnode are greater than the value in the parent node.

Figure 6.11 gives the full class declaration for ordered lists. The private section contains the rep and also comments defining the rep invariant and abstraction function.

Figures 6.12 and 6.13 give the implementation of ordered lists. Note that the implementations of `add_el` and `rem_el` propagate the `CE_Notin` exceptions raised by their recursive calls implicitly.

This definition of `ordered_list` is not *adequate* because there is no convenient way to find out what is in the list without removing elements from the list. For example, it would not be convenient to print an ordered

```
template<class T>
class ordered_list {
public:
// Overview: An ordered list contains a mutable ordered list of elements,
//     of parameter type T. A typical list is a sequence
//     [x_1, ..., x_n] where x_i < x_j if i < j. Ordered_list is a heap abstraction.
// requires: routines that provide a total ordering on T and an
//     equality relation on T

// constructors:
  ordered_list(bool (lt) (T, T), bool (eq) (T, T)) throw ( );
    // effects: Initializes this to be an empty ordered list, where
    //     the ordering on the list will be determined by lt and
    //     eq will be used to determine equality.

// methods:
  void add_el(T el) throw (CE_Isin);
    // modifies: this
    // effects: if el is in this, throws CE_Isin; otherwise, adds
    //   el to this

  void rem_el(T el) throw (CE_Notin);
    // modifies: this
    // effects: If el is not in this, throws CE_Notin;
    //   otherwise, removes el from this.

  bool is_in(T el) const throw ( );
    // effects: if el is in this returns true else returns false.

  bool empty( ) const throw( );
    // effects: if this is empty returns true else returns false.

  T least( ) const throw (CE_Empty);
    // effects: if this is empty, throws CE_Empty;
    //   otherwise, returns the smallest element of this.

  bool rep_ok( ) const throw ( );

private:
  void operator = (ordered_list const &); // disallowed
  ordered_list(ordered_list const&); // disallowed
  ordered_list( ); // disallowed

}; // end ordered_list
```

Figure 6.10 Specification of ordered lists.

```
template<T>
class ordered_list {
public:

  // as in Figure 6.10

private:

  bool (ltp) (T, T); // the comparison operator
  bool (eqp) (T, T); // the equality operator
  bool is_empty; // true if list is empty
  ordered_list *left, *right; // the descendants
  T val; // the value

// the abstraction function is:
//   A(this) = if is_empty then [ ] else A(left) || val || A(right)

// the rep invariant is:
//   is_empty or
//       all elements of A(left) are < val and
//       all elements of A(right) are > val
//       where the comparison is done using ltp.

  ordered_list ( ); // disallowed
  void operator = (ordered_list<T> const&); // disallowed
  ordered_list(ordered<list> const&); // disallowed
```

Figure 6.11 Full class for ordered list.

list using these methods. This situation will be remedied in the chapter on iteration abstraction.

6.7 Summary

In this chapter, we have generalized procedures and data abstractions to be polymorphic, i.e., work for many types. This allows us to define more general abstractions, e.g., a `sort` procedure that works on `array<int>`, `array<char>`, and so on, or a `set` data type such that individual sets are homogeneous, but contain different types of elements, such as `set<int>`, `set<poly>`, and so on.

Polymorphic abstractions are defined using C++ templates. However, we identified a problem with implementing polymorphic abstractions: since the syntax for calling methods is different for objects on the stack than for objects on the heap, a single piece of code cannot handle both kinds of objects by calling their methods directly. We solved this problem by using procedure arguments. These arguments encapsulate details of where the object of the particular parameter type resides; the parameterized code just calls the procedures. Most of the needed procedures can be provided as related routines when new types are defined; this makes polymorphic abstractions easier to use and makes it more likely that the routines passed in as arguments provide the required behavior.

Exercises

6.1 Implement

```
template<class T>
  T maxt(array<T> * x, bool (lt) (T, T)) throw (CE_Empty);
  // requires: lt defines a total order on T
  // effects: if x is empty throws CE_Empty else
  //   returns the largest element of x as determined by lt.
```

Also implement a program that calls `maxt` on an `array<int>`.

6.2 Generalize the `remove_dupls` procedure of figure 3.3 so that it works on an array of arbitrary type. Give a specification and an implementation.

6.3 Define a procedure

```
template<class etype>
  void key_sort(array<struct{int key1; int key2; etype el;} * a) throw ( );
```

`key_sort` modifies its input array `a` by sorting `a`'s elements into increasing order according to the value of the two key fields. `key1` is the primary key; it is used for a preliminary sort of `a`. Then `key2` is used as a secondary key to sort those elements containing the same value for `key1` (for example, elements `x` and `y` such that `x.key1 == y.key1`). Give a specification and implementation of this procedure.

6.4 The specification of `key_sort` may or may not require the sort to be *stable*. A sort is stable if all elements that are the same for all keys retain their relative positions in the input array. Does your specification in exercise 4 imply anything about stability?

6.5 Generalize the `list` abstraction defined in Figure 4.14 so that we can obtain `list<int>`, `list<poly*>`, and so on. Specify and implement the new abstraction and its related routines.

6.6 Suppose you knew that `polys` (figure 4.3) would usually be sparse. Give an implementation that is efficient in such a situation. (Hint: Ordered lists may be useful.)

```
template<class T>
  ordered_list<T>::ordered_list(bool (lt) (T, T), bool (eq) (T, T)) throw() {
    ltp = lt; eqp = eq;
    is_empty = TRUE;
    }

template<class T>
  void ordered_list<T>::add_el(T el) throw(CE_Isin) {
    if (is_empty) {
      left = new ordered_list<T>(ltp,eqp); right = new ordered_list<T>(ltp,eqp);
      val = el; is_empty = FALSE; return;}
    if (eqp(el,val)) throw CE_Isin("call of ordered_list::add_el when el is in");
    if (ltp(el,val)) left → add_el(el); else right → add_el(el); // re−throw
    } // end add_el

template<class T>
  void ordered_list<T>::rem_el(T el) throw (CE_Notin) {
    if (is_empty) throw CE_Notin("call of ordered_list::rem_el when el not in");
    if (eqp(el,val)) {
      if (right → empty( )) { is_empty = left → is_empty; val = left → val;
        right = left → right; left = left → left; return;}
      val = right → least( );
      right → rem_el(val); } // end el = val
    else if (ltp(el,val)) left → rem_el(el); else right → rem_el(el);
    } // end rem_el

template<class T>
  bool ordered_list<T>::is_in(T el) const throw ( ) {
    if (is_empty) return FALSE;
    if (eqp(el,val)) return TRUE;
    if (ltp(el,val)) return left → is_in(el); else return right → is_in(el);
    } // end is_in
```

Figure 6.12 First part of implementation of ordered list.

```
template<class T>
  bool ordered_list<T>::empty( ) const throw ( ) {
    return is_empty;
    } // end empty

template<class T>
  T ordered_list<T>::least( ) const throw (CE_Empty) {
    if (is_empty) throw CE_Empty("call of ordered_list::least on empty list);
    try {return left → least( );}
      catch (CE_Empty) {return val;}
    } // end least

template<class T>
  bool ordered_list<T>::rep_ok( ) const throw ( ) {
    if (is_empty) return TRUE;
    if (!(left → is_empty))  {
      if (!ltp(left → val, val)) return FALSE;
      if (!left → rep_ok( )) return FALSE;
      } // end left checking
    if (right → is_empty) return TRUE;
    if (!(ltp(val, right → val))) return FALSE;
    return right → rep_ok( );
    }  // end rep_ok
```

Figure 6.13 Rest of implementation of ordered list.

Chapter 7

Iteration Abstraction

This chapter discusses our final abstraction mechanism, the *iteration abstraction*, or *iterator* for short. Iterators are a generalization of the iteration methods available in most programming languages. They permit users to iterate over arbitrary types of data in a convenient and efficient way.

For example, an obvious use of a set is to perform some action for each of its elements:

for all elements of the set
  do action

Such a loop might go through the set completely—for example, to print all elements of a set. Or it might search for an element that satisfies some criterion, in which case the loop can stop as soon as the desired element has been found.

`Intsets` as we have defined them so far provide no convenient way to perform such loops. For example, suppose we want to compute the sum of the elements in an `intset`:

```
int setsum (intset const * s) throw ( );
// effects: Returns the sum of the elements of s.
```

The implementation of `setsum` shown in figure 7.1 illustrates the two main defects of our `intset` abstraction. First, to loop through all elements, we delete each element returned by `choose` so that it will not be chosen again. Thus two operations, `choose` and `remove`, must be called on each iteration. This inefficiency could be avoided by having `choose` remove the chosen element, but we still have the second problem, which is that iterating over an `intset` destroys it by removing all its elements. Such destruction may

```
int setsum(intset const * s) throw ( ) {
  array<int> *a = new array<int>();
  int sum = 0;
  try {while (TRUE) {
    int x = s → choose( );
    sum = sum + x;
    s → remove(x);
    a → append(x); } } // end while and try
    catch (CE_Empty) { }
  // restore elements of s
  for (int i = 0; i < a → length(); i++)
    s → insert (a → fetch(i));
  return sum
  } // end setsum
```

Figure 7.1 An implementation of *setsum*.

be acceptable at times, but cannot be satisfactory in general. Although we can collect the removed elements and reinsert them later, as is done in figure 7.1, such a method is clumsy and inefficient.

If `setsum` were an `intset` operation, we could implement it efficiently using array operations. However, `setsum` does not really make sense as an `intset` operation; it seems peripheral to the concept of a set. Furthermore, even if we could justify making it an operation, what about other similar procedures we might want, such as printing all the elements? There must be a way to implement such procedures efficiently outside the type.

To support such iteration adequately, we need to access all elements in a collection efficiently and without destroying the collection. How might we do this for `intsets`? One possibility is to provide a `members` methods:

```
array<int> * members ( ) throw ( );
// effects: returns an array containing the elements of this,
//   each exactly once, in some arbitrary order.
```

Given this operation, we can implement `setsum` as shown in figure 7.2. Since `members` does not modify its argument, we no longer need to rebuild the `intset` after iterating.

Although `members` makes it easier to use `intsets`, it is inefficient, especially if the `intset` is large. First, we have two data structures—the `intset` itself and the array, and, if the set is large, so is the array. Second, in the case of a search loop we have probably done too much work since such a loop need not examine all elements of the collection being searched. For example, if we were searching an `intset` for a negative element, we

```
int setsum (intset const * s) throw ( ) {
  array<int> *a = s → members( );
  for (int i = 0; i < a → length( ); i++)
    sum = sum + a → fetch(i);
  return sum;
  } // end setsum
```

Figure 7.2 Implementation of setsum using the members method.

could stop as soon as we encountered the first negative element. However, we must process the entire collection to build the array.

An alternative to `members` is an operation that simply returns the representing array. However, this solution is very bad since it destroys abstraction by exposing the rep.

What is needed is a general method of iteration that is convenient and efficient and that preserves abstraction. Iterators provides the needed support. An iterator is a special kind of procedure that causes the items we want to iterate over to be produced incrementally. The produced items can be used in other modules that specify actions to be performed for each item. The using module will contain some sort of looping structure, such as

for each result item i produced by iterator A
  do perform some action on i

Each iteration of the loop produces a new item, which is then acted on by the body of the loop.

Note the separation of concerns in such a form. The iterator is responsible for producing the items, while the module containing the loop defines the action to be performed on them. The iterator can be used in different modules that perform different actions on the items, and it can be implemented in different ways without affecting these modules.

Since the iterator causes items to be produced one at a time, it avoids the space and time problems discussed earlier. We need not construct a potentially large data structure to contain the items. Moreover, if the using module is performing a search loop, the iterator can be stopped as soon as the item of interest is found.

As mentioned earlier, iterators are a generalization of the iteration methods available in most programming languages. In addition to some form of "while" loop, programming languages typically provide a "for" loop for iterating over integers. Such iteration is useful in conjunction with arrays, which are indexed, but does not mesh well with non-indexed collections like `intset`. Iterators provide convenient iteration even over non-indexed collections.

```
bool get(T &value);
  // modifies: this, value
  // effects: If there are more results to yield, store the
  //     next result in value, advance past this result, and return true.
  //     Else, do not modify this or value, and return false.
```

Figure 7.3 Specification of the get method for generators.

```
generator<int> *g = primes_lt_100( );
int v;
while (g → get(v)) {
  // a loop body that uses v
  } // end while
```

Figure 7.4 Using an iterator.

7.1 Iteration in C++

C++ does not provide direct support for iteration abstraction. Instead we will provide iteration by a combination of a procedure and a data abstraction. We will refer to the procedure as an *iterator*. An iterator returns a special kind of data object called a *generator*. A generator keeps track of the state of an iteration in its rep. It has a `get` method that is used to get the next element in the iteration; in addition, `get` advances the state of the generator to record the returning of that element. Generators belong to a parameterized type where the parameter determines the type of element produced by `get`. Generator objects reside on the heap.

The specification of `get` is given in Figure 7.3. Note that `get` produces the next value (if any) by modifying its by-reference argument. The specification provides a generic description of all generators; we will see specifications of specific generators in the next section.

Figure 7.4 shows how to use an iterator. First a generator is created by calling the iterator; in this case the iterator `primes_lt_100` returns a `generator<int>` that will produce all prime numbers less than 100. The generator is typically used in a while loop. The loop will continue to iterate as long as `get` returns `true`, i.e., as long as values are being assigned to `v`. Each time the loop body runs, `v` will contain the most recently generated value.

Generators can also be used outside of loops. For example, they can be passed as arguments to routines. Some examples are given in Section 7.4.

```
// a poly iterator method
  generator<int> *terms( ) throw ( );
    // effects: returns a generator that will produce the exponents of all
    //     terms of this up to the degree, in order of increasing exponent.

// a set<T> iterator method
template<T>
class set {
// as in Figure 6.7 except that choose is replaced by elements
public:
  generator<T> *elements( ) const throw ( );
    // effects: returns a generator that will produce all the elements of
    //     this, each exactly once, in arbitrary order.
    // requires: this must not be modified while the generator is in use.
}; // end set
```

Figure 7.5 Two iterator methods.

7.2 Specifying Iterators

The specification of an iterator explains the whole iteration: how the iterator uses its arguments to produce a generator *and* the behavior of the generator. The specification of the `get` method given in Figure 7.3 explains operationally what `get` does, but is otherwise generic: it doesn't explain exactly what any particular generator does. We capture the missing information in the specification of the iterator.

Figure 7.5 gives the specifications of two iterators, both of which are methods of data abstractions. The `terms` iterator is a method of `poly` that provides the ability to iterate through the terms of the `poly`. The specification explains that the returned generator allows iteration over `this`, producing all the exponents of its `poly` for terms up to the degree.

The figure also specifies an `elements` iterator method for sets; this method would replace the `choose` method described earlier. There are three points of interest here. First note that the type parameter of `set` is also used to instantiate the generator. What is going on here is that when `set` is instantiated, whatever actual type is provided for `T` will also be provided for the generator parameter. Thus, for a `set<int>`, the `elements` iterator will return a `generator<int> *`, and so on.

The second point is that the specification of `elements` defines a requirement on how the generator it returns can be used. It is not clear what the generator would do if the set were modified while the generator was being used. Therefore we rule out such modifications in the specification of `elements`. Almost always a generator over a mutable object will have such

```
generator<int> *primes( ) throw ( );
  // effects: returns a generator that will produce all primes,
  //     each exactly once, in increasing order.
```

Figure 7.6 Specification of primes iterator.

a requirement. We state the requirement in a `requires` clause, but since this is a requirement on the use of the generator, rather than on the call to the iterator, we place the `requires` clause at the end of the specification. Normally a `requires` clause is the very first part of a specification; in fact, an iterator might have two requires clauses, one ruling out certain arguments, and the other stating contraints on using the returned generator.

The third point is that unlike the `choose` method, the `elements` iterator does not throw any exceptions. It is typical that the use of iterators eliminates problems associated with certain arguments (like the empty set) that would arise for related procedures such as `choose`.

Although both of these data abstractions provide only one iterator, a data abstraction can have many iterators. Also, neither `terms` nor `elements` modifies anything: the iterator doesn't modify `this` and neither does the generator it returns. Usually iterators are like this, but occasionally modifications are useful. If there is a modification, the iterator specification must explain what it is, and whether the iterator or the generator does the modification.

In addition to iterator methods, it is also possible to have stand-alone iterators. Figure 7.6 gives a specification of such an iterator. The generator returned by `primes` will keep producing results without any bound and therefore it will need to be used in a loop that bounds the iteration.

7.3 Using Iterators

Figure 7.7 gives some examples of using iterators and the generators they return. The routine `set_sum` returns the sum of elements in the set. The routine `big_poly` returns a `poly` of high degree if there is one in the set; otherwise it returns the zero `poly`. This routine uses a searching loop: the iteration stops as soon as a big enough `poly` is found. The `diff` routine differentiates a `poly`. Note that all using code is written in terms of the generic generator type described in Figure 7.3; as we shall see in Section 7.5, iterators actually return objects belonging to subclasses of this type, but the subclasses are never visible to using code.

```
int set_sum(set<int> const *s) throw ( ) {
  generator<int> *g = s → elements( );
  int sum = 0;
  int el;
  while (g → get(el)) sum = sum + el;
  return sum;
  } // end set_sum

poly * big_poly(set<poly *> const *s) throw ( ) {
  // effects: if s contains a poly with degree ≥ 100 returns such
  //     a poly else returns the zero poly.
  generator<poly *> *g = s → elements( );
  poly *p;
  while (g → get(p))
    if (p → degree( ) ≥ 100) return p;
  return new poly( );
  } // end big_poly

poly *diff(poly *p) throw ( ){
  poly *q = new poly( );
  int exp;
  generator<int> *g = p → terms( );
  while (g → get(exp))
    if (exp == 0) continue;
    q = q → add(new poly(exp*p → coeff(exp), exp−1));
  return q;
  } // end diff
```

Figure 7.7 Using iterators and generators.

7.4 Predefined Iterators

Both `string` and `array` have iterator methods, and both have constructors that take generators as arguments. Their specifications are given in Figure 7.8. The constructors are examples of places where generators are used outside of loops.

7.5 Implementing Iterators

To implement an iterator, one needs to write its code *and* define and implement a class for its generator. There will be a separate class for each iterator. These classes aren't visible to users: users don't see their class declaration. Instead, user code is written in terms of the generic `generator` class, as was illustrated in Figure 7.7.

```
// additional string constructor
  string(generator<char>* gen);
    // modifies: gen
    // effects: Extracts all remaining characters from gen and creates
    //     a string that contains the extracted characters in order.

// Iterator method for string:
  generator<char>* chars( ) const;
    // effects: Returns a generator that will produce the characters of
    //     this in order.

// additional array constructor
  array(generator<T>* gen);
    // modifies: gen
    // effects:   Extracts all remaining elements from gen and creates
    //     a new array that contains the extracted elements in order.

// Iterator methods for arrays:
  generator<T>* elements( ) const;
    // effects: Returns a generator that will produce the elements of this
    //   in order.
    // requires: this is not modified while the generator is in use.
```

Figure 7.8 Additional string and array operations.

Each new class is defined to be a *public subclass* of the `generator` class. Such a definition means that any code written to use `generator` objects will be able to use objects of the subclass. This is why users don't need to know about the new class — knowing about the generator class is sufficient.

Figure 7.9 shows the implementation of the `terms` iterator for `poly`. This code would be part of the `.cpp` file that implements `poly`. The implementation is based on the same rep that was given in Figure 4.7. The figure repeats the rep as comments.

The iterator creates and returns an object of type `poly_gen`. The generator class, `poly_gen`, is declared to be a `public` subclass of `generator<int>`, meaning that its objects can be used anywhere where a `generator<int>` is needed. `poly_gen` is *not* described in the `.h` file for `poly`. Instead, using code is supposed to get a `poly_gen` object only by calling the `terms` method.

The `terms` method passes the `trms` array as an argument to the constructor for `poly_gen`. Thus the generator shares the array with the `poly`. This sharing does not constitute exposing the rep, however, because the `poly_gen` class is really part of the implementation of `poly`. Note that the array is passed `const` and therefore cannot (easily) be modified by the

```
class poly_gen: public generator<int> {
  array<int> const *tms; // the terms of the poly
  int next; // the next one to yield

public:
  poly_gen(array<int> const *terms) throw ( ) {
    tms = terms;
    next = 0;
    }

  bool get (int &c) throw ( ) {
    for (; next < tms → length( ); next++)
      if (tms → fetch(next) != 0)
        {c = next; next++; return TRUE;};
    return FALSE;
    } // end get
}; // end poly_gen

// the poly rep is: array<int> * trms;
generator<int> * poly::terms( ) const throw ( ) {
  return new poly_gen(trms);
  } // end terms
```

Figure 7.9 Implementation of the terms iterator.

implementation of `poly_gen`.

Since there is no separate `.h` file for `poly_gen`, its implementation is included in its class definition. Furthermore, the private section is given first since this is convenient. (When giving a specification, we put the private section second as a convention; a specification's purpose is to define the interface and therefore it is better to put that part of the class (the public section) first.)

Figure 7.10 contains an implementation of the `primes` iterator whose specification was given in Figure 7.6. The implementation keeps all the primes generated so far in the `ps` array and uses them to determine whether the next candidate is a prime. Since all primes are odd except for 2, only odd candidates are produced and 2 is handled separately.

7.6 Rep Invariants and Abstraction Functions for Generators

We need to define rep invariants and abstractions functions for generators, just as we do for ordinary abstract types. This section explains how to do

```
class pgen: public generator<int> {
    array<int> *ps;  // primes yielded
    int next; // next candidate to try
public:

  pgen ( ) throw ( ) {
    next = 2;
    ps = new array<int>( );
    }

  bool get(int &v) throw ( ) {
    if (next == 2) {ps → append(2); next = 3; v = 2; return TRUE;}
    int el;
    for (; TRUE; next = next + 2)
      for (int i = 0; i < ps → length( ); i++) {
        el = ps → fetch(i);
        if (next%el == 0) break; // not a prime
        if (el*el > next) // have a prime
          {ps → append(next); v = next; next = next + 2; return TRUE;}
        } // end both fors
    } // end get
}; // end pgen

generator<int> *primes( ) throw ( ) {
  return new pgen( );
  } // end primes
```

Figure 7.10 Implementation of primes iterator.

this.

Rep invariants for generators are similar to those for ordinary abstract types; the only difference is that we will not provide a method to check them. The rep invariant for `poly_gen` is

// $0 \leq$ `next` $\leq$ `length(tms)`

The rep invariant for `pgen` is

// *all elements of* `ps` *are primes, and they appear in sorted order,*
// *and they include all primes* < `next`

// *for all x an element of* `ps` . *x is prime*
// *for all primes* `p` < `next` . `p` *is in* `ps`
// *for all* `i, j` *indexes of* `ps` . `i` < `j` => `ps[i]` < `ps[j]`

Here the invariant is given both informally, and a little more precisely. Note that this invariant would be quite expensive to check.

To define the abstraction function for a generator, we need to understand what the abstract state of a generator is. All generators have the same abstract state: a sequence of the items that remain to be generated. The abstraction function thus needs to map the rep to this sequence.

Here are the abstraction functions for `poly_gen` and `pgen`:

```
// abstraction function for poly_gen
//   A(this) = mk_seq(next, tms)
//     where mk_seq(n, t) =
//        if n = length(t) then [ ] else [t[n]] || mk_seq(n+1, t)

// abstraction function for pgen
//   A(this) = [ p | p is prime and p ≥ next ] such that
//     the sequence is sorted in ascending order
```

The abstraction function for `poly_gen` is defined by giving a way of computing the sequence, whereas the abstraction function for `pgen` is given by describing the sequence that is wanted. Both approaches are legitimate; you should choose the one that is easiest for the particular abstraction function that you want to define.

7.7 Another Example

This section provides another example of an iterator, an `elements` method of the `ordered_list` type. This iterator will produce the elements of the list in sorted order. Its specification is given in Figure 7.11.

In this case, it is useful to give the associated generator access to the rep of `ordered_list` objects. This is accomplished by making it a `friend` of `ordered_list`. The fact that it is a friend is stated in the class shown in Figure 7.11. However, since users of `ordered_list` don't have access to the generator, the information is given in the private section of the class.

The implementation of the iterator and its associated generator is given in Figure 7.12. The generator starts by producing the elements of the left subtree. When all these elements have been produced, it produces the value of the top node of the tree, and then produces the elements of the right subtree. The rep invariant and abstraction function are:

```
// the rep invariant is:
//  child = 0 or (on_left & (child.me = me.left))
//        or (!on_left & (child.me = me.right))
// the abstraction function is:
//  A(this) = mk_seq(on_left, child, me)
```

```
template<T>
class ordered_list {
  // as defined in Figure 6.11

  generator<T> * elements( ) const throw ( );
    // effects: Returns a generator that will produce the elements of this
    //     each exactly once, in order from smallest to largest.
    // requires: this must not be modified while the generator is in use.

private:
  bool (*ltp) (T, T); // the comparison operator
  bool (*eqp) (T, T); // the equality test
  bool is_empty; // true if list is empty
  ordered_list<T> *left, *right; // the descendants
  T val; // the value

  friend class olist_gen; // for implementing the elements iterator
} // end ordered_list
```

Figure 7.11 Specification of the elements iterator for ordered lists.

```
//      where mk_seq(lf, c, m) = if c = 0 then [ ]
//        else if lf then A(c) || [m.val] || A(olist_gen(m.right))
//        else A(c)
```

Note the use of `.` where a C++ program would use `->`, e.g., `child.me`.

7.8 Design Issues

Most data types will include iterators among their operations, especially types like `intset` and `ordered_list` whose objects are collections of other objects. Iterators are frequently needed for adequacy; they make elements of a collection accessible in a way that is both efficient and convenient.

A type might have several iterators. For example, `ordered_list` might have the method

```
generator<T> * big_to_small ( ) const throw ( );
// effects: returns a generator that will produce the elements of this
//   each exactly once, in order from largest to smallest.
// requires: this must not be modified while the generator is in use.
```

in addition to the `elements` method discussed earlier.

For mutable collections we have consistently required that the loop body not modify the collection being iterated over. If we omit this requirement,

```
template<class T>
  generator<T> *ordered_list<T>::elements( ) const throw ( ) {
    return new olist_gen<T>(this);
    } // end elements

template<class T>
class olist_gen: public generator<T> {
    bool on_left; // true if generating from left subtree
    olist_gen<T> * child; // the current sub-generator
    ordered_list<T> const *me; // my node
public:
    olist_gen(ordered_list<T> const *o) throw ( ) {
      me = o;
      if (o -> is_empty) { child = 0; return;}
      on_left = TRUE;
      child = new olist_gen(o -> left); }

    bool get (T &val) throw ( ) {
      if (child == 0) return FALSE;
      if (child -> get(val)) return TRUE;
      if (!on_left) {child = 0; return FALSE;}
      on_left = FALSE;
      child = new olist_gen(me -> right);
      val = me -> val;
      return TRUE; } // end get
}; // end olist_gen
```

Figure 7.12 The elements iterator for ordered list.

the generator returned by the iterator must behave in a well-defined way even when modifications occur. For example, suppose integer `n` is deleted from an `intset` while the generator returned by `elements` is in use; should `n` be produced by the generator or not?

One approach is to require that a generator produce the elements contained in its collection argument at the time it is created by the iterator, even if modifications occur later. The behavior of a generator specified in this way is well defined, but the implementation is likely to be inefficient. For example, if the `elements` iterator had to return this sort of generator, its implementation would have to provide the generator with a copy of the `els` array — just what we objected to in the `members` method. Because the approach of constraining the loop body avoids such inefficiencies, it will be preferred most of the time. A related issue is whether the iterator or the generator it returns can modify the collection. As a general convention, such modifications should be avoided.

Modifications by the loop body or the iterator or generator can sometimes be useful. For example, consider a program that performs tasks waiting on a task queue:

```
generator<task*> *g = q → all_tasks( );
task* t;
while (g → get(t)) {
  // perform t
  // if t generates a new task nt, enqueue it by performing
  //   q → enq(nt)
  }
```

When the task being performed generates another task, we simply enqueue it to be performed later; the generator returned by iterator `all_tasks` will present it for execution at the appropriate time. However, examples like this are rare; usually neither iterator nor loop body will modify the collection.

7.9 Summary

This chapter identified a problem in the adequacy of data types that are collections of objects. Since a common use of a collection is to perform some action for its elements, we need a way to access all elements. This method should be efficient in space and time, convenient to use, and nondestructive of the collection. In addition, it should support abstraction by specification.

Iterators are a mechanism that solves this problem. They return a special kind of object, called a generator, which produces the items in the collection one at a time. Producing items incrementally like this means that extra space to store the items is not needed, and production can be stopped as soon as the desired object has been found. Iterators support abstraction by specification for the containing type by encapsulating the method of producing the items; the method depends on knowledge of the rep, but using programs are shielded from this knowledge.

Iterators and their associated generators are efficient to use. There is simply one call of the generator `get` method per iteration of the loop.

Iterators are useful in their own right, as was illustrated by the `primes` example. However, their main use is as operations of data types. We shall see other examples of such use in the rest of the book.

Further Reading

Atkinson, Russell R., Barbara H. Liskov, and Robert W. Scheifler, 1978. Aspects of implementing CLU. In *Proceedings of the ACM 1978 Annual Conference*, pp. 123–129.

Exercises

7.1 Specify a procedure `is_primes`, which determines whether an integer is prime, and then implement it using `primes` (figure 7.6).

7.2 Implement the `big_to_small` iterator for `ordered_lists` that was specified in section 7.7.

7.3 It would be useful to have an iterator for `poly` (figure 4.3) that yielded all the nonzero terms. Give a specification for this iterator. Should it be a `poly` operation?

7.4 Implement the iterator specified in exercise 3, either as a `poly` operation or not as you prefer. Then use it to implement a procedure that differentiates a `poly`.

7.5 Give a specification of a `stack` abstraction that provides access to its elements in last-in/first-out order. Stacks are mutable. Note that for stacks to be useful in some applications (for example, to implement the type in exercise 4), there must be some way to access all the stack elements. We could provide an iterator that yields the stack elements from the most recently pushed to the least recently pushed, or we could provide a `fetch` method that returns the `i`th element for `i` between 1 and the number of elements on the stack. Which is the better choice?

7.6 Consider the following iterator:

```
template <class T>
  generator<T> * filter (generator<T> *g, bool (f) (T)) throw ( );
  // effects: returns a generator that produces in order, each exactly
  //   once, all elements of g for which f is true.
```

This specification does not require that the generator `g` not be modified (by `get`ting some of its elements) while the returned generator is in use. Should this be disallowed? If it is not disallowed, what should the specification of `filter` say about it?

7.7 Implement the `filter` iterator (using whatever specification you selected in the preceding exercise).

7.8 Implement the following iterator:

```
generator<array<int>*> * perms (array<int> const * a) throw ( );
  // effects: produces the permutations of a, each exactly once,
  //   in arbitrary order
  // requires: a not be modified while generator is in use.
```

7.9 Define and implement an iterator for `list<T>` that produces all the elements of the list in the order in which they occur.

7.10 Provide a specification for a block-structured symbol table data abstraction that provides operations well-suited to compiling programs written in a block-structured language. The table maps names to information; it is parameterized by `T`, the type of this information. It provides methods to enter and leave a new scope, to add names and associated information to a scope, and to look up the information associated with a name. In doing the look up, it should find the most recent definition of the name, i.e., the one in the innermost scope.

7.11 Implement the symbol table abstraction and give the rep invariant and abstraction function. (Hint: Use stacks (see exercise 5) and maps (see exercise 5 in chapter 5); add extra operations to stack and map as needed, and give specifications for any such added operations.)

Chapter 8

Testing and Debugging

So far we have talked a bit about program design and quite a lot about program specification and implementation. We now turn to the related issues of ascertaining whether or not a program works as we hope it will and discovering why not when it does not.

We use the word *validation* to refer to a process designed to increase our confidence that a program will function as we intend it to. Most commonly we do validation through a combination of testing and some form of reasoning about why we believe the program to be correct. We shall use the term *debugging* to refer to the process of ascertaining why a program is not functioning properly, and *defensive programming* to refer to the practice of writing programs in a way designed specifically to ease the process of validation and debugging.

Before we can say much about how to validate a program, we need to discuss what we hope to accomplish by that process. The most desirable outcome would be an ironclad guarantee that all users of the program will be happy at all times with all aspects of its behavior. This is not an attainable goal. Such a guarantee presumes an ability to know exactly what it would mean to make all users happy. The best result we can hope for is a guarantee that a program satisfies its specification. Experience indicates that even this modest goal can be difficult to attain. Most of the time we settle for doing things to increase our confidence that a program meets its specification.

There are two ways to go about validation. We can argue that the program will work on all possible inputs. This must involve careful reasoning about the text of the program, and is generally referred to as *verification*. We shall discuss techniques that allow us to reason rigorously about programs in a later chapter. As we shall see then, formal program verification is generally too tedious to do successfully without machine aids. Unfortunately, because only relatively primitive aids exist today, most program verification is still rather informal. Even informal verification, however, can be a difficult process.

The alternative to verification is testing. We can easily be convinced that a program works on some set of inputs merely by running it on each member of the set and checking the results. If the set of possible inputs is small, exhaustive testing (checking every input) is possible. For most programs, however, the set of possible inputs is so large (indeed, it is often infinite) that exhaustive testing is impossible. Nevertheless, a carefully chosen set of test cases can greatly increase our confidence that the program works as specified. If well done, testing can detect most of the errors in programs.

In this chapter we focus on testing as a method of validating programs. We discuss how to select test cases and how to organize the testing process. We also discuss debugging and defensive programming.

8.1 Testing

Testing is the process of executing a program on a set of test cases and comparing the actual results with the expected results. Its purpose is to reveal the existence of errors. Testing does not pinpoint the location of errors, however; this is done through debugging. When we test a program, we examine the relationship between its inputs and outputs. When we debug a program, we worry about this relationship but also pay close attention to the intermediate states of the computation.

The key to successful testing is choosing the proper test data. As mentioned earlier, exhaustive testing is impossible for almost all programs. For example, if a program has three integer inputs, each of which ranges over the values 1 to 1,000, exhaustive testing would require running the program 1 billion times. If each run took 1 second, this would take slightly more than 31 years.

Faced with the impossibility of exhausting the input space, what do we do? Our goal must be to find a reasonably small set of tests that will allow us to approximate the information we would have obtained through exhaustive testing. For example, suppose a program accepts a single integer as its argument and happens to work in one way on all odd integers and in a second way on all even ones; in this case testing it on any even integer, any odd integer, and zero is a pretty good approximation to exhaustive testing.

8.1.1 Black-Box Testing

Test cases are generated by considering both the specification and the implementation. In black-box testing we generate test data from the specification alone, without regard for the internal structure of the module being tested. This approach, which is common across many engineering disciplines, has several significant advantages. The most important advantage is that the testing procedure is not adversely influenced by the component being tested. For example, suppose the author of a program made the implicit invalid assumption that the program would never be called with a certain class of inputs. Acting upon this assumption, the author might fail to include any code dealing with that class. If test data were generated by examining the program, one might easily be misled into generating data based upon the invalid assumption. A second advantage of black-box testing is that it is robust with respect to changes in the implementation. Good black-box test data need not be changed even when major changes are made to the program being tested. A final advantage is that the results of a test can be interpreted by people unfamiliar with the internals of the program being tested.

Testing Paths through the Specification

A good way to generate black-box test data is to explore alternate paths through the specification. These paths can be through both the requires and effects clauses. As an example of a path through the requires clause, consider the specification

```
float sqrt (float x, float epsilon) throw( );
// requires: x ≥ 0 & .00001 < epsilon < .001
// effects: returns sq such that x − epsilon ≤ sq*sq ≤ x + epsilon
```

The requires clause of this specification is the conjunction of two terms:

1. $x \geq 0$.
2. $.00001 <$ `epsilon` $< .001$

To explore the distinct ways in which the requires clause might be satisfied, we must explore the pairwise combinations of the ways each conjunct might be satisfied. Since the first conjunct is a disjunct of two primitive terms ($x \geq 0$ is just a shorthand for $x = 0 \mid x > 0$), it can be satisfied in one of two ways. This leaves us with two interesting ways to satisfy the requires clause:

1. $x = 0$ and $.00001 <$ `epsilon` $< .001$

2. `x` > 0 and $.00001 <$ `epsilon` $< .001$

Any set of test data for `sqrt` should certainly test each of these cases.

It can be difficult to formulate test data that explore many different paths through the effects clause of the specification. It may be difficult even to know which paths can be explored. For example, given the above specification of `sqrt`, we might expect the program sometimes to return an exact result, sometimes a result a little less than the square root, and sometimes a result a little greater. However, a program that always returned a result greater than or equal to the actual square root would be a perfectly acceptable implementation. We would not be able to find test data that forced this program to return a result less than the square root, but we could not know this without examining the code. In fact, without examining the code, we would have no idea which classes of inputs would lead to results in the three categories.

Nevertheless, we should always examine the effects clause carefully and try to find test data that exercise different ways to satisfy it. For example, consider the `intset::is_in` operation:

```
bool is_in (int x) throw ( );
// effects: returns true if x is in this else returns false
```

The effects clause of this specification is a conjunction: Either `x` is in `s` or it is not. Both conjuncts should be tested by the test cases.

Often paths through the effects clause pertain to error handling. Failing to signal an exception when called with exceptional input is just as serious as failing to do the proper thing with normal input. Therefore the test data should cause every possible signal to be raised. For example, consider the specification

```
int search (array<int> const *a, int x) throw (CE_Notin);
// effects: If x is in a, returns i such that a[i] = x, else throws CE_Notin.
```

Here we must include test cases for both the case in which `x` is in `a` and the case in which it is not. Similarly, if `sqrt` signaled exceptions rather than having a requires clause, we would want to include test data that should cause the exceptions.

Testing Boundary Conditions

A program should always be tested on "typical" input values—for example, an array or a set containing several elements, or an integer between the smallest and largest values expected by a program. It is also important to test atypical inputs, though; these tend to show up as boundary conditions.

Considering all paths through the requires clause tests certain kinds of boundary conditions—for example, the case in which `sqrt` is asked to find the square root of zero. A lot of boundary conditions, however, do not emerge from such analysis. It is important to check as many boundary conditions as possible. Such checks catch two very common kind of errors:

1. logical errors, in which a path to handle a special case presented by a boundary condition is omitted, and

2. failure to check for conditions that may cause the underlying language or hardware system to raise an exception (for example, arithmetic overflow).

To generate tests designed to detect the latter kind of error, it is a good idea to use test data that cover all combinations of the largest and smallest allowable values for all bounded numerical arguments. For example, tests for `sqrt` should include cases for `epsilon` very close to .001 and .00001. For strings, tests should include the empty string and a one-character string; for arrays, we should test the empty array and a one-element array.

Aliasing Errors

Another kind of boundary condition occurs when two different formals both refer to the same mutable object. For example, suppose procedure

```
void append_array (array<int> *a1, array<int> *a2) throw ( );
// modifies: a1 and a2.
// effects: Removes all elements of a2 and appends them in
//   reverse order to the end of a1.
```

were implemented by

```
void append_array (array<int> *a1, array<int> *a2) throw ( ) {
  while (a2 → length( ) > 0) {
    a1 → append(a2→top( ));
    a2 → remove( ); }
  } // end append_array
```

Any test data that did not include an input in which `a1` and `a2` refer to the same nonempty array would fail to turn up a very serious error in `append_array`.

8.1.2 Testing Based on the Program Text

While black-box testing is generally the best place to start when attempting to test a program thoroughly, it is rarely sufficient. Without looking at the internal structure of a program it is impossible to know which test cases are likely to give new information. It is therefore impossible to tell how much coverage we get from a set of black-box test data. For example, suppose a program relies on table lookup for some inputs and computation for others. If the black-box test data happened to include only those values for which table lookup is used, the tests would give no information about the part of the program that computed values.

A good way to supplement black-box testing is with inputs that exercise the different paths through the program. The goal here is to have a test set such that each path is exercised by at least one member of the set. We say that such a test set is *path-complete*. In a later chapter we discuss a rigorous technique for enumerating paths through a program. Here we rely on informal arguments.

Consider the program

```
int max_of_three (int x, int y, int z) throw ( ) {
  if (x > y) {
    if (x > z) return x; else return z;
    if (y > z) return y; else return z; }
  } // end max_of_three
```

Despite the fact that there are n^3 inputs, where n is the range of integers allowed by the programming language, there are are only four paths through the program. Therefore the path-complete property leads us to partition the test data into four classes. In one class x is greater than y and z. In another, x is greater than z but smaller than y, and so forth. Representatives of the four classes are

3, 2, 1   3, 2, 4   1, 2, 1   1, 2, 3

It is easy to show that path-completeness is not sufficient to catch all errors. Consider the program

```
int max_of_three (int x, int y, int z) throw ( ) {
  return x;
  } // end max_of_three
```

The test set containing just the input

2, 1, 1

is path-complete for this program. Using this test might mislead us into believing that our program was correct, since the test would certainly fail to uncover any error. The problem is that a testing strategy based upon exercising all paths through a program is not likely to reveal the existence of

```
j = k;
for (int i = 1; i ≤ 100; i++)
  if (pred(i*j)) j++;
```

Figure 8.1 A program with many paths.

missing paths, and omitting a path is a fairly common programming error. This problem is a specific instance of the general fact mentioned earlier: No set of test data based solely upon analysis of the program text is going to be sufficient. One must always take the specification into account.

Another potential problem with a testing strategy based upon selecting path-complete test data is that there are often too many different paths through a program to make that practical. Consider the program fragment in figure 8.1. There are 2^{100} different paths through this program, as can be seen from the following analysis. The **if** statement causes either the true or the false branch to be taken, and both of these paths go on to the next iteration of the loop. Thus for each path entering the ith iteration, there are two paths entering the $(\mathtt{i}+1)$st iteration. Since there is one path entering the first iteration, the number of paths leaving the ith iteration is 2^i. Therefore there are 2^{100} paths leaving the 100th iteration.

Testing each of 2^{100} paths is not likely to be practical. In such cases we generally settle for an approximation to path-complete test data. The most common approximation is based upon considering two or more iterations through a loop as equivalent and two or more recursive calls to a procedure as equivalent. To derive a set of test data for the above program, for example, we find a path-complete set of test data for the program

```
j = k;
for (int i = 1; i ≤ 2; i++)
  if (pred(i*j)) j++;
```

There are only four paths through this program. A path-complete set of test data would have representatives in the following categories:

1. pred(k) and pred(2k+2)
2. pred(k) and ¬pred(2k+2)
3. ¬pred(k) and pred(2k)
4. ¬pred(k) and ¬pred(2k)

To sum up, we always include test cases for each branch of a conditional. However, we approximate path-complete testing for loops and recursion as follows:

1. For loops with a fixed amount of iteration, as in the example just shown, we use two iterations. We choose to go through the loop twice rather than once because failing to reinitialize after the first time through a loop is a common programming error. We also make certain to include among our tests all possible ways to terminate the loop.

2. For loops with a variable amount of iteration, we include zero, one, and two iterations, and in addition we include test cases for all possible ways to terminate the loop. For example, consider

```
while (x > 0) {
  // do something
  }
```

With a loop like this, it is possible that no iterations will be performed. This situation should always be handled by the test cases because not executing the loop is another situation that is likely to be a source of program error.

3. For recursive procedures we include test cases that cause the procedure to return without any recursive calls and test cases that cause exactly one recursive call.

This approximation to path-complete testing is, of course, far from fail-safe. Like engineer's induction, "One, two, three—that's good enough for me," it frequently uncovers errors, but offers no guarantees.

8.1.3 An Example

To illustrate these points, we consider a simple procedure for determining whether a string is a palindrome. A palindrome is a string the reads the same backward and forward (an example is "deed"). Figure 8.2 gives a specification and implementation of this procedure. By looking at the specification we can see that we need tests that cause both true and false to be returned. In addition, we must include the empty string as a boundary condition. This might lead to the strings " ", "deed", and "ceed". Examination of the code indicates that we should test the following cases:

1. not executing the loop;

2. returning false in the first iteration;

```
bool palindrome (string s) throw ( );
// effects: Returns true if s reads the same forward and backward;
//    otherwise returns false. E.g., "deed" and " " are both palindromes.


bool palindrome (string s) throw ( ) {
  int low = 0;
  int high = s.length( ) −1;
  while (high > low) {
    if (s[low] != s[high]) return FALSE;
    low ++;
    high −−; }
  return TRUE;
  } // end palindrome
```

Figure 8.2 The *palindrome* procedure.

3. returning true after the first iteration;
4. returning false in the second iteration;
5. returning true after the second iteration.

Cases 1, 2, 3, and 5 are covered already. For case 4 we might add "aaba". At this point we should ask ourselves whether there is any case we have missed, and we might notice that all our test strings have an even number of characters. Therefore we should add a number of odd-length test strings. These strings should include the boundary case of a string of length 1. Finally, we should arrange the test cases in a sensible order, with the shortest first. Such an arrangement helps in finding errors (see section 8.4).

8.1.4 Testing Iterators

Generating test cases for iterators is similar to generating them for procedures. The only point of interest is that iterators have paths in their specifications that are similar to those for loops. In other words, we must be sure to include cases in which the generator returned by the iterator produces exactly one result, and produces two results, and if it is possible for the generator to not produce any results, we should include a case for that. For example, consider the `primes` iterator:

```
generator<int> * primes(int n) throw ( );
// effects: Returns a generator that produces all primes less than
//   or equal to n; if there are no such primes (i.e., n < 2)
```

// *the generator produces nothing.*

Test cases here could include calls with `n` equal to 1, 2, and 3. Whether more tests are needed can be determined by looking at the iterator's implementation.

8.1.5 Testing Data Types

In testing data types, as usual we generate test cases by considering the specifications and implementations of each of the operations. We must now test the operations as a group rather than individually, however, because some operations produce the objects that are used in testing others. In the `intset` operations, for example, the constructor and the `insert` and `remove` methods must be used to generate the arguments for the other operations and for each other. (The specification for `intset` is repeated in figure 8.3.)

We begin by looking at paths in specifications. The specifications of `is_in` and `elements` have obvious paths to explore. For `is_in` we must generate cases that produce both true and false as results. Because `elements` is an iterator, we must look at least at paths of lengths zero, one, and two. Therefore we shall need `intset`s containing zero, one, and two elements. The empty `intset` and the one-element `intset` also test boundary conditions. Thus to test the observers, we might start with the following `intset`s:

1. the empty `intset` produced by calling the `intset` constructor;
2. the one-element `intset` produced by inserting 3 into the empty set; and
3. the two-element `intset` produced by inserting 3 and 4 into the empty set.

For each, we would do calls on `is_in`, `size`, and `elements` and check the results. In the case of `is_in`, we would do calls in which the element is in the set, and others in which it is not.

We obviously do not yet have enough cases. For example, `remove` is not tested at all, and there are also paths in other specifications that have not yet been discussed. These paths are somewhat hidden in our informal specifications; they would be more evident if we had used formal specifications, which is another of their virtues.

The size of an `intset` remains the same when we insert an element that is already in the set, and we must therefore look at a case in which we insert the same element twice. Similarly, the size decreases when we remove an element only if it is in the set, so that we must look at one case in which

```
class intset {
public:

// Overview: intsets are mutable, unbounded sets of integers.
//     A typical intset is {x1, ..., xn}. intset is a heap abstraction.

// Constructors:

  intset ( ) throw ( );
    // effects: Initializes this to be empty

// Methods:

  void insert (int x) throw ( );
    // modifies: this
    // effects: Adds x to the elements of this, i.e., s_post = s ⋃ x

  void remove (int x) throw ( );
    // modifies: this
    // effects: Removes x from this, i.e., s_post = s − x

  bool is_in (int x) const throw ( );
    // effects: If x is in this returns true else returns false.

  int size ( ) const throw ( );
    // effects: Returns the cardinality of this.

  generator<int> * elements ( ) const throw ( );
    // effects: returns a generator that produces all the elements of
    //   this, each exactly once, in arbitrary order.
    // requires: this not be modified while generator is in use

  bool rep_ok ( ) const throw ( );

private:

  intset(intset const&); // disallowed
  void operator = (intset const&); // disallowed

  array<int> * els; // the elements without duplicates
  int get_index(int x) const throw ( );
  // effects: if x not in this returns −1 else returns i such that
  //    els[i] = x.
}; // end intset
```

Figure 8.3 Specification of the intset data abstraction.

we remove an element after inserting it and another in which we remove an element that is not in the set. We might use these additional `intset`s:

1. the set obtained by inserting 3 twice into the empty set;

2. the set obtained by inserting and then removing 3; and

3. the set obtained by inserting 3 and removing 4.

To find these hidden paths we must look explicitly for paths in the mutators. Thus `insert` must work properly whether the element being inserted is already in the set or not, and similarly for `remove`. This simple approach will produce the three cases just given.

In addition, of course, we must look for paths in the implementations of the operations. The cases identified so far provide quite good coverage for the implementation using the array with no duplicates (see figures 8.4 and 8.5). One possible problem is in `is_in`, which contains a loop. To cover all paths in this loop, we must test the case of a two-element array with either no match or a match with the first or the second element. (It is not possible to find such tests cases by considering only the specification. At the level of the specification we are concerned only with whether or not the element is in the set; its position in the array is not of interest.) Similarly, in `remove` we must be sure to delete both the first and second elements of the array. In doing so we cover the potential error of not handling the last element of the array properly. The implementation of `remove` moves the last element of the array into the location of the deleted element before shrinking the array. If we reversed the order of these two actions, the program would not work when the element being deleted was stored at the high bound of the array.

8.2 Unit and Integration Testing

Testing typically occurs in two phases. During *unit testing* we attempt to convince ourselves that each individual module functions properly in isolation. During *integration testing* we attempt to convince ourselves that when all the modules are put together, the entire program functions properly.

Integration testing is generally more difficult than unit testing. First, the intended behavior of an entire program is often much harder to characterize than the intended behavior of its parts. Second, problems of scale tend to arise in integration testing that do not arise in unit testing; for example, it may take much longer to run a test. Finally, specifications play rather different roles in the two kinds of validation.

The acceptance of the specification as a given is a key factor that distinguishes unit testing from integration testing. During unit testing, when

```
intset::intset ( ) throw ( ) {
  els = new array<int>( ); } // end

void intset::insert (int x) throw ( ) {
  if (get_index(x) == −1) els → append(x);
  } // end insert

void intset::remove (int x) throw ( ) {
  int i = get_index(x);
  if (i == −1) return;
  els → store(i, els → top( ));
  els → remove( );
  } // end remove

bool intset::is_in (int x) const throw ( ) {
  if (get_index(x) > −1) return TRUE; else return FALSE;
  } // end is_in

int intset::get_index (int x) const throw ( ) {
  for (int i = 0; i < els → length( ); i++)
    if (x == els → fetch(i))  return i;
  return −1;
  } // end get_index

int intset::size ( ) const throw ( ) {
  return els → length( ); } // end size

generator<int> * intset::elements ( ) const throw ( ) {
  return intset_gen(els); } // end elements

bool intset::rep_ok ( ) const throw ( ) {
 for (int i = 0; i < els → length( ); i++) {
   int x = els → fetch(i); // check for dups of x
   for (int j = i + 1; j < els → length( ); j++)
     if (x == els → fetch(j)) return FALSE; }
 return TRUE;
 }  // end rep_ok
```

Figure 8.4 Implementation of intset.

```
class intset_gen {

  int next; // the next one to yield
  array<int> * els; // the array containing the elements

public
  intset_gen(array<int> const * e) throw ( ) {
    next = 0;
    els = e; }

  bool get (int & v) throw ( ) {
    if (next ≥ els → length( )) return FALSE;
    v = els → fetch(next − 1);
    next ++;
    return TRUE; }
}; // end intset_gen
```

Figure 8.5 The intset_gen generator.

a module fails to meet its specification, we generally conclude that it is incorrect. When validating a whole program, we must accept the fact that the most serious errors are often errors of specification. In these cases, each unit does what it is supposed to, but the program as a whole does not. A prime cause of this kind of problem is ambiguous specifications. When this occurs, a module may perform as expected by those doing its unit testing while failing to meet the expectations of those writing some of the modules that call it. This makes errors detected during integration testing particularly difficult to isolate.

Consider a program implemented by module P, which calls module Q. During unit testing, P and Q are tested individually. (To test either individually, it is necessary to simulate the behavior of the other, as will be discussed.) When each of them has run correctly on its own test cases, we test them together to see whether they jointly conform to P's specification. In doing this joint test, we use P's test cases. Now suppose that an error is discovered. These are the possibilities:

1. Q is being tested on an input that was not covered in its test cases.

2. Q does not behave as was assumed in testing P.

It is tempting when dealing with multiple modules like P and Q to test them jointly rather than to do unit tests for each first. Sometimes such joint tests are a reasonable approach, but unit testing is usually better. For example, to test the program shown in figure 8.1, we only care that each of the four paths be covered; the various ways in which `pred` produces its results are not of concern. However, testing `pred` thoroughly probably involves many test cases. Combining all these test cases has many disadvantages: There are more tests to run, tests may take longer to run, and if either of these modules is reimplemented, we shall have to rethink the whole set of test cases. Testing each module individually is more efficient.

8.3 Tools for Testing

It is useful to automate as much of the testing process as possible. Usually we cannot automate the generation of test data; generating appropriate inputs for testing a program is a nonalgorithmic process requiring serious thought. Furthermore, to automate the process of deciding what outputs are appropriate for any set of inputs is often as difficult and error prone as writing the program being tested.

What we can automate are the processes of invoking a program with a predefined sequence of inputs and checking the results with a predefined sequence of tests for the acceptability of outputs. A mechanism that does this is called a *driver.* A driver should call the unit being tested and keep track of how it performs. More specifically, it should

1. Set up the environment needed to call the unit being tested. In some languages this may involve creating and initializing certain global variables. In most languages it may involve setting up and perhaps opening some files.

2. Make a series of calls. The arguments for these calls could be read from a file or embedded in the code of the driver. If arguments are read from a file, they should be checked for appropriateness, if possible.

3. Save the results and check their appropriateness.

The most common way to check the appropriateness of results is to compare them to a sequence of expected results that has been stored in a file. Sometimes, however, it is better to write a program that compares the results directly to the input. For example, if a program is supposed to find the roots of a polynomial, it is easy enough to write a driver that checks whether or not the values returned are indeed roots. Similarly, it is easy to check the results of `sqrt` by a computation. A driver for testing an implementation of the `sqrt` specification given previously is shown in figure 8.6.

In addition to drivers, testing often involves the use of *stubs.* A driver simulates the parts of the program that call the unit being tested. Stubs simulate the parts of the program called by the unit being tested. A stub must

```
// accept as inputs the files:
//      file_of_tests, bad_tests, correct_results, and incorrect_results

for // each test in file_of_tests
  { if (test.square < 0 || test.epsilon < .00001 || test.epsilon > .001)
     { // add test to bad_tests }
     else {result = sqrt(test.square, test.epsilon);
       if (fabsf(square − result*result) ≤ epsilon )
          { // add <test, result> to correct_results }
          else { // add <test, result> to incorrect_results } }}
```

Figure 8.6 Driver for *sqrt*.

1. Check the reasonableness of the environment provided by the caller.

2. Check the reasonableness of the arguments passed by the caller.

3. Modify arguments and the environment and return values in such a way that the caller can proceed. It is best if these effects match the specification of the unit the stub is simulating. Unfortunately, this is not always possible. Sometimes the "right" value can only be found by writing the program the stub is supposed to replace. In such cases we must settle for a "reasonable" value.

(If all communication is only via arguments and results, then it is not necessary to check or modify the environment.)

Drivers are clearly necessary when testing modules before the modules that invoke them have been written. Stubs are necessary when testing modules before the modules that they invoke have been written. Both are needed for unit testing, in which we want to isolate the unit being tested as much as possible from the other parts of the program.

In practice, it is common to implement drivers and stubs that rely on interaction with a person. A very simple implementation of a stub might merely print out the arguments it was called with and ask the person doing the testing to supply the values that should be returned. Similarly, a simple driver might rely on the person doing the testing to verify the correctness of the results returned by the unit being tested. Although drivers and stubs of this nature are easy to implement, they should be avoided whenever possible. They are far more prone to error than automated drivers and stubs, they make it hard to build up a good database of test data, and they make it hard to reproduce tests.

The reproducibility of tests is particularly important. The following testing scenario is all too typical:

1. The program is tested on inputs 1 through `n` without uncovering an error.

2. Testing the program on input $\mathtt{n}+1$ reveals the existence of an error.

3. Debugging leads to a fix that makes the program work on input $\mathtt{n}+1$.

4. Testing continues at input $\mathtt{n}+2$.

This is an unwise practice, for there is a nonnegligible probability that the change that made the program work on input $\mathtt{n}+1$ will cause it to fail on some input between 1 and `n`. Whenever any change is made, no matter how small, it is important to make sure that the program still passes all the tests it used to pass. This is called *regression testing*. Regression testing is practical only when there are tools available that make it relatively easy to rerun old tests.

It is important to implement drivers and stubs with care. When an error is detected, we want it to be in the code being tested. If drivers and stubs are implemented carelessly, however, they are at least as likely to contain errors as the program being tested. In this case the programmer wastes lots of time testing and debugging the testing environment.

8.4 Debugging

Testing tells us that something is wrong with a program, but knowing the symptom is a far cry from knowing its cause. Once we know that a problem exists, the tactics to be used in locating and fixing the problem—in debugging—are extremely important. The variance in the efficiency with which people debug is quite high, and we can offer no magic nostrums to make debugging easy. Most of what we have to say on the subject is simple common sense.

Debugging is the process of understanding and correcting errors. In debugging we try to narrow the scope of the problem by looking for simple test cases that manifest the bug and by looking at intermediate values to locate the responsible region in the code. As we collect evidence about the bug, we formulate hypotheses and attempt to refute them by running further tests. When we think we understand the cause of the bug, we study the appropriate region of code to find and correct the error. In chapter 11 we discuss ways to annotate and reason about code that facilitate such study.

The word "bug" is in many ways misleading. Bugs do not crawl unbidden into programs. We put them there. *Do not think of your program as "having bugs"; think of yourself as having made a mistake.* Bugs do not breed in programs. If there are many bugs in a program, it is because the programmer has made many mistakes.

Always keep in mind that debugging consumes more time than programming. It is worth trying very hard to get your program right the first time. Read your code very carefully and *understand exactly why you expect it to work* before you begin to test it. No matter how hard you try and no matter how clever you are, though, the odds against your program working properly the first time are very long. Consequently, you should *design, write, and document your programs in ways that will make them easier to test and debug.* The key is making sure that you have relatively small modules that can be tested independently of the rest of your program. To a large extent, this can be achieved by following the design paradigms outlined earlier in this book. Introduce data abstractions and associate with each the most restrictive possible rep invariant. Write careful specifications for each procedure, so that when it comes time to test it, you know both what input values it should be prepared to deal with and what it should do in response to each of the possible inputs.

Just as you need an overall testing strategy, you also need a careful plan for every debugging session. *Before beginning, decide exactly what you want to accomplish and how you plan to accomplish it.* Know what input you are going to give your program, and exactly what you expect it to do with that input. If you have not thought carefully about your inputs, you will probably waste a lot of time doing things that are not likely to help isolate the problem.

The so-called scientific method provides a good paradigm for systematic debugging. The crux of the scientific method is to

1. begin by studying already available data,

2. form a hypothesis that is consistent with those data, and

3. design and run a repeatable experiment that has the potential to refute the hypothesis.

Consider a program that accepts a positive integer as input and is supposed to return true if the number is prime and false otherwise. As our first test case we try 2, and our program returns the correct answer, true. We next try 3, and the program returns the incorrect result, false. We now have two pieces of data on which to form a hypothesis. One plausible hypothesis is that somehow we have failed to reinitialize something after the first input and that the program will always work on the first input and fail on the second. To check this hypothesis we can test the program on the same two arguments but reverse the order of the tests; that is, we can try 3 and then 2. Before running the tests, we decide which results would support our hypothesis and which would refute it:

1. results supporting hypothesis: (true, false);

2. results refuting hypothesis: (false, true), (false, false), (true, true).

When we try the experiment, the program returns false and then true. We immediately reject our first hypothesis and look for another, for example, that the program will fail on all odd primes.

When debugging, a good starting goal is to *find a simple input that causes the problem to occur.* This input may not be the test data that first revealed the existence of the bug. It is often possible to find simpler input that is sufficient to provoke a manifestation of the bug. Thus a good way to start is to pare down the test data and then run the program on variants of that subset.

For example, suppose we are testing the `palindrome` procedure of figure 8.2, and when we run it on the famous (allegedly Napoleonic) palindrome "able was I ere I saw elba", it returns false. This is a rather long palindrome, so we should try to find a shorter one on which the program fails. We might begin by taking the middle character of this palindrome and seeing whether the program succeeds in recognizing that the single character "r" is itself a palindrome. If it fails on that, we might hypothesize that the program does not work on palindromes containing an odd number of characters, and we should examine our other tests to see whether they support this hypothesis. If it succeeds in recognizing that "r" is a palindrome, we might try "ere" on the hypothesis that it will fail on odd palindromes containing more than one character. If "ere" fails to provoke the error, we should probably try "I ere I". Suppose the program fails on this input. Two hypotheses come to mind: perhaps the blanks are the root problem, or perhaps it is the uppercase letters. We should now test our program on the shortest inputs that might confirm or refute each hypothesis, for example, " " and "I".

Once we have found a small input that causes the error to occur, we use this information to locate where in the program the bug is likely to be. Finding the kind of input necessary to provoke a symptom is often tantamount to locating a bug. If not, however, the next step is to narrow the scope of the problem by examining intermediate results.

The goal is to rule out parts of the program that cannot be causing the problem and then look in more detail at what is left. We do this by *tracing* the program, that is, running it and looking at the values of variables at specific points in its control flow. If the program consists of several modules, our first goal is to discover which module is the source of the bug. We do this by tracing all calls and returns of procedures. For each call, we ask whether the arguments are what they should be; the arguments should satisfy the requires clause of the called procedure and should also follow from what we have learned in the trace so far. If the arguments are not right, then the error is in the calling module. Otherwise we ask whether the results of the call follow from the arguments. If not, the error is in the called procedure.

Localizing the problem to a single procedure is often enough, since we can then discover the error by examining the code of the faulty module. Sometimes, however, it is useful to narrow the bug to a subpart of the faulty module. To do this, we continue the trace and examine the values of local variables of the module. The goal is to detect the first manifestation of incorrect behavior. It is particularly important to *check the appropriateness of intermediate results* against values computed prior to beginning the trace. If you wait until you see the intermediate results before thinking about what they should be, you run the risk of being unduly influenced by your (erroneous) program.

Consider the following incorrect implementation of `palindrome`:

```
bool palindrome (string s) throw ( ) {
  int low = 0;
  int high = s.length( ) −1;
  while (high > low) {
    if (s[low] != s[high]) return FALSE;
    low ++;
    if (high > low + 1) high −−; }
  return TRUE;
  } // end palindrome
```

This implementation does not work properly on odd-length palindromes that have more than one character. Therefore we might trace it on the string "ere". Suppose we expect that `low` and `high` equal the low and high bounds of the array at the end of initialization and that `low` increases and `high` decreases in each iteration. At the end of initialization, `low` = 0 and `high` = 2, as expected. However, at the end of the first iteration, we notice that `low` = 1 and `high` = 2, which is not expected. We should be able recognize the error at this point.

If you have access to an excellent debugger, it may be possible to examine intermediate results relatively conveniently, for example, merely by telling the debugger which variables you wish to trace. If not, it is worth your while to *write a considerable amount of code whose only purpose is to help you examine intermediate results.* One piece of code that should be written for each type is a routine that displays the objects in abstract form. When debugging polynomials, for example, it is much easier to understand what is happening if the poly is displayed as the string

"3 + x ** 5"

instead of the array

[3, 0, 0, 0, 1]

Such a routine can be called either by a user of an interactive debugger or by a print statement. Also, it is sometimes useful to have the reverse routine that accepts an abstract form for an object and produces an appropriate object of the rep type. (We have been providing output ability by overloading `operator <<`; input ability can be providing by overloading `operator >>`.)

In planning debugging sessions, keep in mind that *the bug is probably not where you think it is.* If it were, you would have found it by now. It is all too easy to develop and cling to a fixed idea about the location of a bug. The first obvious manifestation of a bug can occur far from the code where the error lies. If you think that you know which procedure contains the error, and you have spent a significant amount of time examining that procedure, you are probably wrong. Keep an open mind. Examine the reasoning that led you to that procedure, and ask yourself whether there is any possibility that it is flawed.

One way to get a slightly different perspective on the problem is to *ask yourself where the bug is not.* It can be much easier to understand where a bug could not possibly be than where it is. Often, trying to demonstrate that a bug could not possibly be in a particular place will lead to the discovery that that is exactly where it is. In any event, the systematic elimination of possibilities is frequently the best way to home in on a bug.

While trying to eliminate possible locations for a bug, *take a careful look at your input* as well as at your code. Every programmer has spent time hunting for a bug in a program when in fact the problem is in the input. As mentioned, you should write your drivers and stubs carefully so as to avoid as many such errors as possible.

Looking carefully at the input is a good example of the more general principle: *Try the simple things first.* Most errors are not particularly subtle. Simple errors that occur frequently include

1. reversing the order of input arguments,

2. failing to initialize a variable,

3. failing to reinitialize a variable the second time through a program segment,

4. copying only the top level of a data structure when you intended to copy all levels, and

5. failing to parenthesize an expression correctly.

Remember that in designing your program you went through a reasoning process not unlike that involved in debugging it. The existence of a bug is evidence that your initial reasoning was flawed. It is easy to convince yourself that a procedure does not contain a bug by using the same reasoning that led you to introduce the bug in the first place. It can therefore be invaluable to *get somebody else to help you.* Asking for help is not an admission of failure; it is merely good practice. Try to explain your problem to somebody else. Often, the mere attempt to articulate your reasoning will lead you to discover the source of the problem. Failing that, a fresh viewpoint is almost certain to prevent you from getting stuck in too deep a rut.

One of the hardest problems in debugging is *deciding what to take for granted.* The naive thing to say is that one should take nothing for granted, but this is generally counterproductive. When a test yields a faulty result, the problem might lie in one of the modules the program calls, the compiler used to compile the program, the operating system, the hardware, the electrical system of the building housing the hardware, and so on. The most likely location, however, is your program. Begin by taking everything else for granted and looking for a bug in the program. If after a reasonable amount of effort you fail to find any problem with your program, start worrying about the modules your program calls. If after a reasonable amount of effort you can find no problem there, find out whether or not the compiler or operating system has been changed recently. If not, you should be very reluctant to attribute your problems to either of them.

When you encounter a bug that you just cannot track down, *make sure that you have the right source code.* In putting together large systems, you will almost always have to rely on separate compilation facilities. This can easily lead to a situation in which the object code exhibiting the bug does not match the source in which you are trying to find the bug. A particularly vexing variant of this problem can occur when either the compiler or the operating system is changed. When your program's behavior has changed and you are absolutely sure that you have not changed anything, make sure that you are not using code that has been compiled to run on a different version of the operating system.

When you have tried everything you can think of and still have not found the bug, *go away.* The goal of any programming project is to complete the program (including its documentation and testing) expeditiously. The goal is not to find a particular bug as soon as possible. The obsessive pursuit of a particular bug is almost always counterproductive. If you spend too long looking for the same bug, there is a high probability that you will become stuck in a rut. If you try to debug when you are overly tired, you will at best work inefficiently. At worst, you will make mistakes, such as making ill-considered changes to the program or accidentally deleting a crucial file.

When you do find a bug, *try to understand why you put it there.* Was it a clerical error, does it reflect a lack of understanding of the programming language, or is it indicative of some logical problem? Knowing why you inserted a bug may help you understand how to fix your program. It may also help you to discover other bugs and even to avoid bugs in the future.

Finally, when you think that you have found a bug and that you know how it got there, *do not be in too much of a rush to fix "the bug."* Make sure that the bug you found could indeed have caused the symptoms that you observed. If you have already spent a lot of time observing the behavior of your program, it may be counterproductive to change that behavior before you have completed your detective work. Not only is it often easier to repair many bugs at once than to repair many bugs one at a time, but it almost always leads to a cleaner and more efficient program.

When you do decide to make a change, *think through all of its ramifications.* Convince yourself that the change really will both cure the problem and not introduce new problems. The hardest bugs to find are often those we insert while fixing other bugs. This is because we are often not as systematic in designing these "patches" as in our original designs. We try to make local changes, when a more global approach might well be called for. It is often more efficient to reimplement a small procedure than to patch an old one.

8.5 Defensive Programming

In preparing to cope with mistakes, it pays to program defensively. In every good programmer there is a streak of suspicion. Assume that your program will be called with incorrect inputs, that files that are supposed to be open may be closed, that files that are supposed to be closed may be open, and so forth. Write your program in a way designed to call these mistakes to your attention as soon as possible. C++ provides some help here via its compile-time type checking and other compiler warnings. In addition, our `string` and `array` types help by doing checks for out-of-bounds subscripts.

Two standard defensive programming methods not built into any programming language are checking requirements and rep invariants, and exhaustive testing of all conditionals. The violation of a rep invariant or a procedure's requirements is often the first manifestation of a bug. If code to check these explicitly is not included, the first observable symptom of the bug may occur quite far from the place where the mistake actually occurred.

We make it easy to check the rep invariant by including the `rep_ok` method in every type. This method can be used to test the rep invariant at the beginning of each operation, and before returning if `this` is new or has been modified. The checks can either be put in your test driver, or in the type's operations. The latter is particularly robust, but you may need to disable the checks when in production to improve performance.

As an example of requirements violation, consider a procedure with the specification

```
bool in_range (int x, int y, array<int> * a, int e) throw ( );
// requires: x ≤ y
// effects: Returns true if e is an element of a[x], . . . ,a[y], i.e.,
//   b = ∃ z [ x ≤ z ≤ y & a[z] = e ]
```

Suppose that a caller of this procedure reverses the order of the first two arguments. `in_range` will probably return false whether or not `e` is in `a`. The first observable symptom of this incorrect call might appear arbitrarily far from the call. In the worst case, the error would never be detected, and the program in which `in_range` occurs would simply return an incorrect answer. However, if `in_range` checks the requirement and signals an exception if it isn't satisfied, the error can usually be found immediately.

Failure to perform exhaustive testing in conditionals can have a similar effect. For example, suppose the `receive` procedure delivers a string that has been sent over a communications network in a message and that for this particular call only the values "deliver" and "examine" are meaningful. The implementation

```
s = receive( );
if (s == "deliver") { // carry out the deliver request}
  else if (s == "examine") {// carry out the examine request}
  else {// handle error case }
```

is far superior to the marginally more efficient implementation

```
s = receive( );
if (s == "deliver") { // carry out the deliver request}
  else {// carry out the examine request}
```

Rep invariants and requirements are two examples of assertions about intermediate states of a computation. In a later chapter we shall discuss several other kinds of assertions. It is frequently useful to write code to check these.

Defensive programming generally involves a certain amount of extra overhead—both for the programmer and at runtime. Most of the time, the programming overhead is not an issue, since defensive programming almost always reduces the total amount of programmer time over the course of a programming project. The runtime overhead cannot be dismissed so easily. For programs in which performance is an issue, some defensive programming methods can be prohibitively expensive. If, for example, the hardware does not detect arithmetic overflow, detecting it in software can more than double the cost of doing arithmetic. On a more abstract level, a binary search procedure can hardly afford to check that every array given to it is indeed sorted.

When it seems that defensive programming will be prohibitively expensive in a production version of a program, we should still give serious thought to putting the checks in while the program is under development. Removing error detection code just before a program is put into production use is much easier than inserting it during debugging. Removing a program's defenses, however, should not be done as a matter of course. If at all possible, these should be left in the production version. It is almost certain that when the program first goes into production use, it will still contain some bugs and that other bugs will be introduced as it is modified. It is important that these bugs be detected and repaired as expeditiously as possible. The actual economic cost of an undiscovered error in a program may exceed the cost of keeping the checks in during production runs. It is usually worthwhile to retain at least the inexpensive checks.

8.6 Summary

This chapter has discussed the related issues of testing and debugging. Testing is a method of validating a program's correctness. We have described a way to develop test cases methodically by examining both a module's specification and its implementation. The test cases should then be run by a driver that checks the results of each case; the driver either produces the inputs or reads them from a file, and either checks the results by computations or compares them to outputs in a file. If the test being run is a unit test, then lower-level modules are replaced by stubs that simulate their effects. Later, during integration testing, the stubs are replaced by the implementation.

Testing can exhibit the presence of a bug. Debugging is the process of understanding and correcting the cause of the bug. In debugging we try to narrow the scope of the problem by searching for simple test cases that manifest the bug and by looking at intermediate values to locate the responsible region in the code. As we collect evidence about the bug, we formulate hypotheses and attempt to refute them by running further tests. When we think we understand the cause of the bug, we study the responsible region of code to find and correct the error.

Debugging can be made easier if we practice defensive programming. This consists of inserting checks in the program to detect errors that are likely to occur. In particular, we should check that the requires clause is satisfied. It is also a good idea to check the rep invariant. These checks should be retained in the production code if possible.

The outcome of being methodical about testing, debugging, and defensive programming is a reduction of programmer effort. This work pays off not only when the program is written, but also later when it is modified.

Further Reading

Beizer, B., 1983. *Software Testing Techniques.* New York: Van Nostrand Reinhold.

Goodenough, John B., and Susan L. Gerhart, 1975. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* SE-1(2): 156–173.

Howden, William E., 1981. A survey of dynamic analysis methods. In *Tutorial: Software Testing and Validation Techniques* (New York: IEEE Computer Society Press), pp. 209–231.

Weinberg, Gerald, M., 1971. *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold.

Exercises

8.1 Develop a set of test cases for `merge` using the specification given in figure 3.7 and the implementation given in figure 3.8. Do the same thing for `merge_sort` and `sort`. Write a driver for `merge`.

8.2 Develop a set of test cases for `poly` (figures 4.3, 4.7, and 4.8). Write a driver for `poly`.

8.3 Develop a set of test cases and write a driver for `permutations` (see exercise 8 in Chapter 7).

8.4 Suppose `intsets` were implemented using ordered lists (see figures 7.11 and 7.12). Discuss what kind of stub you would use for ordered lists in testing your implementation of `intset`.

8.5 The block-structured symbol table in exercise 10 of chapter 7 can be implemented using stacks and maps. Sketch such an implementation. Then discuss how you would use stubs for these two abstractions in testing `symtab`.

8.6 Implement an iterator that yields all Fibonacci numbers. (A Fibonacci number is the sum of the preceding two Fibonacci numbers, and the first Fibonacci number is 0. For example, the first seven Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8.) Define test cases in advance of debugging. Then debug your program and report on how successful your tests were.

8.7 Develop an error profile for yourself. Keep a log in which you record errors in your programs. For each error, record the reason for it and look for patterns.

Chapter 9

More on Specifications

Throughout this book we emphasize the importance of specifications in all stages of program development. Our main premise is that the proper use of abstraction is the key to good programming. Without specifications, abstractions are too intangible to be helpful. In this chapter we discuss the meaning of specifications and some criteria to consider when writing them. We also discuss two primary uses of specifications.

9.1 Specifications and Specificand Sets

The purpose of a specification is to define the behavior of an abstraction. Users will rely on this behavior, while implementors must provide it. An implementation that provides the described behavior is said to *satisfy* the specification.

We define the meaning of a specification to be the set of all program modules that satisfy it. We call this the *specificand set* of the specification. As an example, consider the specification

```
int p (int y);
// requires: y > 0
// effects: returns an int x such that x > y
```

This is satisfied by any procedure named p that, when called with an argument greater than zero, returns a value greater than its argument. Members of the specificand set include

```
int p (int y) { return y+1; }
int p (int y) { return y*2; }
```

```
int p (int y) { return y*3; }
```

Like every specification, this one is satisfied by an infinite number of programs.

It is important to remember that a specification, its specificand set, and a particular member of the specificand set are very different kinds of things, as different as a program, the set of all possible executions of that program, and an execution of that program on a single set of data.

9.2 Some Criteria for Specifications

Good specifications take many forms, but all of them have certain attributes in common. Three important attributes—restrictiveness, generality, and clarity—are discussed in this section.

9.2.1 Restrictiveness

There is a vast difference between knowing that some members of a specification's specificand set are appropriate and knowing that all members are appropriate. This is similar to the difference between knowing that a program works on some inputs and knowing that it works on all inputs, a difference we emphasize when we talk about testing programs (see the chapter on testing). A good specification should be restrictive enough to rule out any implementation that is unacceptable to its abstraction's users. This requirement is the basis of almost all uses of specifications.

In general, discussing whether or not a specification is sufficiently restrictive involves discussing the uses to which members of the specificand set might be put. There are certain common mistakes, however, that almost always lead to inadequately restrictive specifications. One such mistake is failing to state needed requirements in the requires clause. For example, figure 9.1 gives three specifications for an `elems` iterator for a bag of integers. (A bag is like a set except that elements can occur in it more than once. For example, a bag[int] could contain 3 twice. Bags are sometimes called multisets.) The first specification fails to address the question of what happens if `b` is changed within a loop using `elems`. It therefore allows implementations exhibiting radically different behavior. For example, does changing `b` affect the values returned by `elems`?

One way to deal with this particular problem is to require that `b` not be changed within the loop using `elems`, as is done in the second specification. This specification may or may not be sufficiently restrictive, since it does not constrain the order in which the elements are returned. It would be better if it either defined an order or included the phrase "in arbitrary order." In addition, the specification fails to make clear what is done when

```
template <class T>
  generator<T> * elems ( ) const throw ( );
  // effects: returns a generator that produces every element of this,
  //   one at a time.

template <class T>
  generator<T> * elems ( ) const throw ( );
  // effects: returns a generator that produces every element of this,
  //   one at a time.
  // requires: this not be modified while generator is in use

template <class T>
  generator<T> * elems ( ) const throw ( );
  // effects: returns a generator that produces every element of this,
  //   one at a time, in arbitrary order. Each element is
  //   produced exactly the number of times it occurs in this.
  // requires: this not be modified while generator is in use
```

Figure 9.1 Three specifications of `elems`.

```
int index_string(string s1, string s2) throw( );
// effects: If s1 occurs as a substring in s2, then returns the least index
//   at which s1 occurs. Returns -1 if s1 does not occur in s2, and 0 if s1
//   is the empty string.
//   E.g.,
//     indexs("bc", "abcbc") = 1
//      indexs("", "a") = 0
```

Figure 9.2 Specification of index_string operation.

an element is contained in `b` more than once. For that matter, it does not even say explicitly that `elems` returns only elements that are in `b`. The third specification corrects these deficiencies.

Other mistakes are failing to identify when exceptions should be signaled and failing to specify behavior at boundary cases. For example, consider a procedure `index_string` that takes strings `s1` and `s2` and, if `s1` is a substring of `s2`, returns the index at which `s1`'s first character occurs in `s2`; for example `index_string(''ab'', ''babc'')` returns 1. A specification that contained only this information would not be restrictive enough because it does not explain what would happen if `s1` were not a substring of `s2` or if it occurred multiple times in `s2`, or if `s1` or `s2` were empty. The specification in figure 9.2 is restrictive enough.

The moral is that it takes considerable care to write sufficiently restrictive specifications. A formal specification language can play a helpful role here (formal specifications will be discussed in a later chapter).

9.2.2 Generality

A good specification should be general enough to ensure that few, if any, acceptable programs are precluded. The importance of the generality criterion may be less obvious than that of restrictiveness. It is not essential to ensure that no acceptable implementation is precluded, but the more desirable (that is, efficient or elegant) implementations should not be ruled out. For example, the specification

```
float sqrt (float sq, float e) throw ( );
// requires: sq ≥ 0 & e > .001
// effects: returns rt such that 0 ≤ (rt*rt − sq) ≤ e.
```

constrains the implementor to algorithms that find approximations that are greater than or equal to the actual square root. The constraint may well result in a needless loss of efficiency.

It is our desire to make specifications as general as possible that has led us to the *definitional* style of specification used in this book. A definitional specification explicitly lists properties that the members of the specificand set are to exhibit. The alternative to a definitional specification is an *operational* one. An operational specification, instead of describing the properties of the specificands, gives a recipe for constructing them. For example,

```
int search (array<int> const *a, int x) throw (CE_Notin);
// effects: Examines a[0], a[1], ..., in turn and returns the index of
//   the first one that is equal to x. Signals CE_Notin if none equals x.
```

is an operational specification of `search`, while

```
int search (array<int> const *a, int x) throw (CE_Notin);
// effects: Returns i such that a[i] = x;
//   signals CE_Notin if there is no such i.
```

is definitional. The first specification explains how to implement `search`, while the second merely describes a property that its inputs and outputs must satisfy. Not only is the definitional specification shorter, but it also allows greater freedom to the implementor, who may choose to examine the array elements in some order other than first to last.

Operational specifications have some advantages. Most significantly, they seem to be relatively easily constructed by trained programmers—chiefly because their construction so closely resembles programming. They are generally longer than definitional specifications, however, and they often lead to overspecification. The operational specification of `search`, for example, specifies which index is to be returned if `x` occurs more than once in `a`. As another example, consider trying to write an operational specification for a square root procedure.

A good check for generality is to examine every property required by a specification, in both the requires and effects clauses, and ask whether it is really needed. If it is not, then it should be eliminated or weakened. Also, any portion of a specification that is operational rather than definitional should be viewed with suspicion.

9.2.3 Clarity

When we talk about what makes a program "good," we consider not only the computations it describes but also properties of the program text itself—for example, whether it is well modularized and nicely commented. Similarly, when we evaluate a specification, we must consider not only properties of the specificand set but also properties of the specification itself—for example, whether it is easy to read.

A good specification should facilitate communication among people. A specification may be sufficiently restrictive and sufficiently general—that is, it may have exactly the right meaning—but this is not enough. If this meaning is hard for readers to discover, the specification's utility is severely limited.

There are two distinct ways in which people may fail to understand a specification. They may study it and come away knowing that they do not understand it. For example, a reader of the second specification of `elems` in figure 9.1 may be confused about what to do if an element occurs in the bag more than once. This is troublesome, but not as dangerous as when people come away thinking that they understand a specification when in fact they do not. In such a case, the user and the implementor of an abstraction may each interpret its specification differently, leading to modules that cannot work together. For example, the implementor of `elems` may decide to yield each element the number of times it occurs in the bag, while the user expects each element to be yielded only once.

Clarity is an important but amorphous criterion. It is easy enough to say that a good specification should be easy to understand, but much harder to say how to achieve this. There are many factors contributing to clarity, of which conciseness, redundancy, and structure are perhaps the most important.

The most concise presentation may not always be the best specification, but it is frequently the best starting point. There are good reasons to increase the size of a specification by adding redundant information or levels of structure, as we shall discuss, but it is important to avoid pointless verbosity. Generally, as a specification grows longer it is more likely to contain errors, less likely to be completely and carefully read, and more likely to be misunderstood.

It is important not to confuse length with completeness. A stream-of-consciousness technique will easily lead to specifications that are both long and incomplete. Like programs, specifications that grow by accretion are often longer than they need to be. Instead of just adding to a specification, it is important to step back from that local change and see whether there is a way to consolidate the information. It takes more time to write a complete short description than to write a complete long one, but the author of a specification owes it to the readers to make this investment.

Any specification containing redundant text is less concise than it could be. Redundancy should not be introduced without a good reason, but it can be justified in two ways: to reduce the likelihood that a specification will be misunderstood by its readers and to catch errors.

In many respects, the role of a specification is like that of a textbook. It should be designed not merely to contain information, but to communicate that information effectively. Redundancy can be used to reduce the likelihood that an important point will be missed. The old dictum, "Tell 'em what you're gonna tell 'em, tell 'em, then tell 'em what you told 'em," has some pedagogical validity. The key is to present the same information in more than one way, to be redundant without being repetitious. Consider, for example,

```
bool p (intset *s1, intset *s2) throw ( );
// effects: returns true if s1 is a subset of s2 else returns false

bool p (intset *s1, intset *s2) throw ( );
// effects: returns b = ∀x [x ∈ s1 ⇒ x ∈ s2].

bool p (intset *s1, intset *s2) throw ( );
// effects: returns true if s1 is a subset of s2 else returns false
//    i.e., b = ∀x [x∈ s1 ⇒ x ∈ s2].
```

The first specification is concise and for most readers quite clear. However, some readers might be left with a nagging doubt: Was the word "subset" carefully chosen, or might the the author have meant *proper* subset? The second specification, while a bit harder to read than the first, leaves no doubt on this point. The question it raises is why, if the specifier intended that p be a subset test, was this not stated explicitly? The third specification, of course, answers both of these questions.

By stating the same thing in more than one way, a specification provides readers with a benchmark against which they can check their understanding. This helps to prevent misunderstandings and thus allows readers to spend less time studying a specification. A particularly useful kind of redundancy in this regard is one or more well-chosen examples, such as those given in the specification of `string_index` in figure 9.2.

A specification that states the same thing in more than one way also allows for the fact that different readers will find different presentations of the same information easier to understand. Frequently, some critical part of a specification is a concept with a name that will be meaningful to some readers, but not to others. For example, consider

```
float pv (float inc, float r, int n) throw ( );
// requires: inc > 0 & r > 0 & n > 0.
// effects: Returns the present value of an annual income of inc for
//     n years at a risk-free interest rate of r.
//      I.e., pv(inc,r,n) = inc + (inc/(1+r)) + ... + (inc/(1+r)^(n-1)).
//      E.g., pv(100, .10, 3) = 100 + 100/1.1 + 100/1.21
```

For readers well versed in financial matters, a specification that did not use the phrase "present value" would not be as easy to understand as one that did. For readers lacking that background, the part of the specification following "I.e." is invaluable. The part following "E.g." can be used by either group of readers to confirm their understanding.

If readers are to benefit from redundancy, it is critical that all redundant information be clearly marked as such. Otherwise a reader can waste a lot of time trying to understand what new information is being presented when, in fact, none is. A good way to indicate that information is redundant is to preface it with "i.e." or "e.g."

Redundancy does not reduce the number of mistakes in a specification. Instead it makes them more evident and provides the reader with the opportunity to notice them. For example, consider

```
bool too_cold (int temp) throw ( );
// effects: returns true if temp is ≤ 0 degrees
//    Fahrenheit; otherwise returns false.

bool too_cold (int temp) throw ( );
// effects: returns true if temp is ≤ 0 degrees
//    Fahrenheit; otherwise returns false. I.e., returns true
//    exactly when temp is not greater than the freezing point of
//    water at standard temperature and pressure.
```

The first specification offers a reader no reason for suspicion, but the second should ring a useful warning bell for most readers. This warning should eventually lead to a revised specification.

One of the primary problems with informal specifications is that each reader brings a somewhat different knowledge and perspective to the task of reading a specification. Introducing redundancy can go a long way toward coping with this. For example, consider

```
int billion ( ) throw ( );
// effects: Returns the integer one billion.

int billion ( ) throw ( );
// effects: Returns the integer one billion, i.e., 10^9.
```

Both American and British readers are likely to find the first specification perfectly unambiguous. Unfortunately, they are also likely to interpret it in completely different ways, for in the United States a billion is 10^9, whereas in Britain it is 10^{12}. The insertion of what an American author might consider redundant information in the second specification precludes any confusion.

9.3 Why Specifications?

Specifications are essential for achieving program modularity. Abstraction is used to decompose a program into modules. An abstraction is intangible, though. Without some description, we have no way to know what it is or to distinguish it from one of its implementations. The specification is this description.

A specification describes an agreement between providers and users of a service. The provider agrees to write a module that belongs to the specificand set. The user agrees not to rely on knowing which member of this set is provided, that is, not to assume anything except what is stated by the specification. This agreement makes it possible to separate consideration of the implementation from the use of a program unit. Specifications provide the logical fire walls that permit divide-and-conquer to succeed.

Specifications are obviously useful for program documentation. The very act of writing a specification is also beneficial because it sheds light on the abstraction being specified. Our experience is that we often profit as much from this activity as from our use of the result. Writing a specification almost always teaches us something important about the specificand set being described. It does this by encouraging prompt attention to inconsistencies, incompleteness, and ambiguities. In some cases, such improved understanding is the most important result of a specification effort.

The goal is to write specifications that are both restrictive enough and general enough. Thus we pay special attention to requirements, exceptions, and boundary conditions. Doing this involves posing questions about the behavior of the abstractions, questions like those posed about `string_index` — for example, what to do if either string is empty. The point is that posing and answering such questions forces us to think carefully about an abstraction and its intended use.

The construction of a specification focuses attention on what is required of a program. It serves as a mechanism for generating questions that should be answered in consultation with users of a program or a module, rather than by implementors. By encouraging the asking of these questions in the early stages of system development, specification helps us debug our understanding of a system's requirements and design before we start implementation.

As we shall discuss in later chapters, specifications should be written as soon as the decisions they record have been made. Since specifications become irrelevant only when their abstraction is obsolete, they should continue to evolve as long as the program evolves. It is a serious mistake to treat the process of writing specifications as a separate phase of a software project.

Once written, specifications can serve many different purposes. They are helpful to designers, implementors, and maintainers alike. During the implementation phase of the software life cycle, the presence of a good specification helps both those implementing the specified module and those implementing modules that use it. As discussed previously, a good specification strikes a careful balance between restrictiveness and generality. It tells the implementor what service to provide, but does not place any unnecessary constraints on how that service is provided. In this way it allows the implementor as much flexibility as is consistent with the needs of users. Of course, specifications are crucial for users, who otherwise would have no way to know what they can rely on in implementing their modules. Without specifications, all that exists is the code, and it is unclear how much of that code will remain unchanged over time. During testing, specifications provide information that can be used in generating test data and in building stubs that simulate the specified module. (We discussed this use in chapter 8.) During the system-integration phase, the existence of good specifications can reduce the number and severity of interfacing problems by reducing the number of implicit assumptions about module interfaces. When an error does appear, specifications can be used to pinpoint where the fault lies. Moreover, they define the constraints that must be observed in correcting the error, which helps us to avoid introducing new errors while correcting old ones.

Finally, a specification can be a helpful maintenance tool. The existence of clear and accurate documentation is a prerequisite for efficient and effective maintenance. We need to know what each module does and, if it is at all complex, how it does it. All too often, these two aspects of documentation are intimately intertwined. The use of specifications as documentation, however, helps to keep them separate and makes it easier to see the ramifications of proposed modifications. For example, a proposed modification that requires us only to reimplement a single abstraction without changing its specification has a much smaller impact than one that changes the specification as well.

9.4 Summary

This chapter has discussed specifications and offered criteria to follow in writing them. We defined the meaning of a specification to be the set of all program modules that satisfy it. This definition captures the intuitive purpose of a specification, namely, to state what all legal implementations of an abstraction have in common. Such a specification tells users what they can rely on and tells implementors what they must provide.

Good specifications should be restrictive, general, and clear. Restrictiveness and generality involve the set of modules that satisfy the specification: No implementations that would be unacceptable to users of an abstraction should be permitted, and desirable implementations (ones that are efficient or elegant, for example) should not be ruled out. Generality is made easier when specifications are written using a definitional approach, which just states properties of the specificand set. An operational approach, which explains a way to implement the abstraction, tends to yield specifications that are too restrictive.

Clarity refers to the ease with which users understand the specification. The main way to enhance clarity is to start with a concise statement and then add some redundancy, often in the form of an example. Redundancy allows readers to check their understanding of the specification. It also makes errors more evident, since these often show up as inconsistencies in the redundant descriptions. To make the reader's job as simple as possible, all redundant information should be clearly marked as such.

Specifications have two main uses. First, the act of writing a specification sheds light on the abstraction being specified by focusing attention on the properties of that abstraction. This use can be enhanced by careful attention to properties that might be overlooked, including what should be stated in the requires clause, exactly when exceptions should be signaled, and the treatment of boundary cases. Sometimes this use is the main benefit of a specification because it points out a problem with the abstraction that requires further study.

The second use is as documentation. Specifications are valuable during every phase of software development, from design to maintenance. Of course, they are not the only program documentation required. A specification describes what a module does, but any module whose implementation is clever should also have documentation that explains how it works.

Program modification and maintenance are eased if these two forms of documentation are clearly distinguished.

A specification is the only tangible record of an abstraction. Specifications are a crucial part of our methodology, since without them abstractions would be too imprecise to be useful. We shall continue to emphasize them in the chapters that follow.

Further Reading

Parnas, David L., 1977. The use of precise specifications in the development of software. In *Proceedings of IFIP Congress 77*, pp. 861–868.

Exercises

9.1 Provide a concise but readable specification of a bag-of-integers abstraction, with operations to create an empty bag, insert and remove an element, test an element for membership, give the size of the bag, give the number of times an element occurs in a bag, and produce the elements of the bag.

9.2 Take a specification you have given for a problem in an earlier chapter and discuss its restrictiveness, generality, and clarity.

9.3 Is it meaningful to ask whether a specification is correct? Explain.

9.4 Discuss how specifications can be used during system integration.

9.5 Discuss the relationship between an abstraction, its specification, and its implementation.

Chapter 10

Hierarchy and Inheritance

This chapter discusses a way to extend the utility of data abstraction by defining families of related types. All members of a family have similar behavior: they all have certain methods, and calls on those methods behave in similar ways. Family members may differ by extending the behavior of the common methods, or by providing additional methods.

A type family might correspond to the kind of hierarchy found in the real world, e.g., `busses` and `cars` are both specialized kinds of `vehicles`, or `dogs` and `cats` are special kinds of `mammals`. Or it might correspond to concepts that only exist within programs, e.g., a `bordered_window` is a specialized kind of `window`, or an `ifstream` is a specialized kind of `istream`.

A type family is defined by a *type hierarchy*. At the top of the hierarchy is a *base type*; this type's specification defines the behavior common to all family members, including the signatures and behavior of all the common methods. Other family members are defined to be *subtypes* of the base type, which is referred to as their *supertype*. Both base types and subtypes are defined using classes, as discussed in previous chapters, but with certain additional features. The hierarchy can be more than two levels: subtypes of the base type can themselves have subtypes, and so on.

Type families are used in two different ways. First they can be used to define *multiple implementations* of a type. In this case the subtypes do not add any new behavior, except that each of them has its own constructors. Rather the class implementing the subtype implements the behavior defined by the supertype. For example, we could use a type family to provide both sparse and dense polynomials, so that the most efficient representation could be used for each polynomial object. A type family providing multiple implementations has only two levels: all the implementations are direct subtypes of the base type.

More generally, though, the subtypes in a type family extend the behavior of their supertypes, for example, by providing additional methods. The class implementing a subtype implements the extended behavior. The hierarchy defining such a type family can be multilevel. Furthermore, at the bottom of the hierarchy there could be a final level providing multiple implementations of some subtype.

Type hierarchies are defined in C++ using the *inheritance* mechanism. This mechanism allows a class to be declared a *subclass* of some other class or classes. Subclassing can be either *public* or *private*. In this book we will be interested primarily in public subclassing, which is the mechanism used to define type families. Furthermore, we will be interested primarily in *single inheritance*, i.e., in the case where a class has at most one superclass. Private subclassing and multiple superclasses are discussed briefly at the end of the chapter.

10.1 Type Hierarchy and Assignment

The utility of type hierarchy rests on a loosening of the rules governing assignment and argument passing: a variable declared to belong to one type can actually refer to an object belonging to some subtype of that type. In particular, if `S` is a subtype of `T`, then pointers to `S` objects can be assigned to variables of type `T*`, and they can be passed as arguments or results where a `T*` or a `T&` is expected.

For example, suppose that `dense_poly` and `sparse_poly` are subtypes of `poly` (the idea is that `dense_poly` provides a good implementation of `poly`s that have relatively few zero coefficients below the degree term, and `sparse_poly` is good for the `poly`s that don't match this criterion). Then the following code is permitted:

```
poly * p1 = new dense_poly( ); // the zero poly
poly * p2 = new sparse_poly(3, 20); // the poly 3x^20.
```

Thus variables of type `poly*` can refer to `dense_poly` and `sparse_poly` objects.

Having assignments like these leads to a situation in which the type of object referred to by a variable is not necessarily the same as what is declared for the variable. E.g., `p1` is declared to have type `poly*` but in fact refers to a `dense_poly` object. To distinguish these two types, we refer to an object's *apparent* type and its *actual* type. The apparent type is what the compiler can deduce given the information available to it (i.e., from declarations); the actual type is the type the object really has. For example, the object referred to by `p1` has apparent type `poly`, but actual type `dense_poly`. The actual type of an object will always be a subtype of its apparent type.

Note that the compiler does type checking based on the information available to it: it uses the apparent type, not the actual types, to do the checking. In particular, it determines what method calls are legal based on the apparent type. For example,

```
int d = p1 → degree( );
```

is considered to be legal provided `poly`, the apparent type of `p1`, has a method named `degree` that returns an `int`.

For this to make sense, it is essential that the object `p1` refers to have all the methods indicated by the supertype, with the expected signatures. Thus `dense_poly` and `sparse_poly` must have all the methods declared for `poly` with the expected signatures. As we shall see, C++ ensures that this condition is satisfied.

It is also essential that methods of subtypes *behave* as indicated in the specification of the supertype. This is not a condition that a programming language or a compiler can ensure. Instead, we will need to ensure it by reasoning about the specifications of the sub- and supertypes. This issue is discussed further in Section 10.5.

A further problem is that the compiler may not be able to determine what code to run when a method is called. The code to run may depend on the actual type of the object, while the compiler knows only the apparent type. For example, consider the compilation of:

```
poly * diff(poly * p) {
  // differentiates p
  generator<int> * g = p → terms( );
  ...
  } // end diff
```

When this routine is compiled, the compiler does not know whether the actual type of the object `p` refers to is a `dense_poly` or a `sparse_poly`, yet it must call the implementation of `terms` for `dense_poly` if `p` is a `dense_poly`, and the implementation of `terms` for `sparse_poly` if `p` is a `sparse_poly`. (It must call the code determined by the actual type since the representations are different and the code works differently in the two cases.)

Calling the right method is achieved by a runtime mechanism called *dispatching*. The compiler does not generate code to call the method directly. Instead it generates code to look in a table to find out what code to call, and then branch to that code.

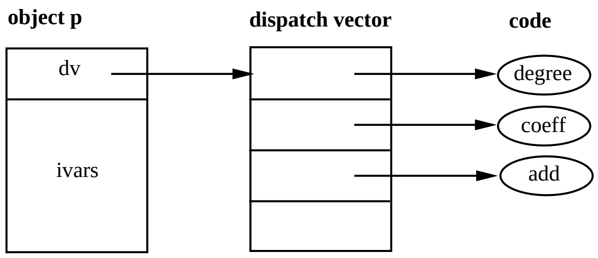

Figure 10.1 A poly object with a dispatch vector.

There are several ways to implement dispatching. One approach is to have objects contain, in addition to their instance variables, a pointer to a *dispatch vector*, which contains pointers to the implementations of the object's methods. This structure is illustrated in Figure 10.1. Code to call a method retrieves the dispatch vector from the object, retrieves the address of the method's code from the appropriate slot of the dispatch vector, and then branches to that address. For example, to call the `add` method of `poly p`, the calling code would call the code pointed at by the third slot of `p`'s dispatch vector.

10.2 Defining a Type Hierarchy

The first step in defining a type hierarchy is to define the class at the top of the hierarchy. This class is called a *base class*. A base class will usually have some methods that could be implemented by a subclass. Such methods are indicated by declaring them to be `virtual`. Marking a method `virtual` tells the compiler that it will not know what code to run at compile time but instead must find this out at runtime (by dispatching). A base class can also be *abstract*, meaning that it exists solely as a specification and doesn't have any objects. Abstract classes are discussed in Section 10.4.

For example suppose we wanted to have a family of integer set types. The base type, `intset`, will provide a minimal (adequate) set of methods; subtypes will provide additional methods. A specification for `intset` is given in Figure 10.2. In this case the intention is that any method can be implemented by a subclass; that is why they are all marked `virtual`.

We can implement `intset` in the usual way. For example, we could use the implementation given in Figure 15.3, in which the elements of the `intset` are stored in the `els` array.

```
class intset {
public:
// overview: intsets are mutable sets of integers. A typical
//   intset is {x1, x2, ...}. intsets reside on the heap.

// constructors:
  intset ( ) throw ( );
  // effects: makes this be the empty set.

// methods:
  virtual void insert (int x) throw ( );
    // modifies: this
    // effects: adds x to this

  virtual void remove (int x) throw ( );
    // modifies: this
    // effects: Removes x from this

  virtual bool is_in (int x) const throw ( );
    // effects: If x is in this returns true else returns false.

  virtual int size ( ) const throw ( );
    // effects: Returns the size of this.

  virtual generator<int> * elements ( ) const throw ( );
    // effects: Returns an generator that will produce the elements of
    //      this, each exactly once, in arbitrary order.

  virtual bool subset (intset *s) const throw ( );
    // effects: Returns true if this is a subset of s else returns false.

  virtual bool rep_ok( ) const throw ( );

private:
  void operator = (intset const &); // disallowed
  intset (intset const &); // disallowed
}; // end intset
```

Figure 10.2 Specification for intset.

```
class intset_max: public intset {
public:
// overview: intset_max is a subtype of intset with an additional
//   method, max, to determine the maximum element of the intset.

//constructors:
  intset_max ( ) const throw ( );
  // effects: makes this be the empty intset_max.

// methods:
  virtual void insert (int x) throw ( );
  virtual void remove (int x) throw ( );
  virtual bool rep_ok( ) const throw ( );

  virtual int max ( ) const throw (CE_Empty);
  // effects: if this is empty throws CE_Empty else returns largest
  //   element of this

private:
  void operator = (intset const &); // disallowed
  intset_max (intset const &); // disallowed
}; // end intset
```

Figure 10.3 Specification for intset_max.

10.2.1 Defining Subtypes

A subtype is defined by declaring a class to be a *public* subclass of another class. For example, we might define `intset_max` to be a subtype of `intset`. An `intset_max` object behaves just like an `intset` except that it has a method that returns the largest element of the set.

The class declaration for `intset_max` is given in Figure 10.3. Note that the specification does not attempt to define what an `intset` is; instead it relies on the supertype specification for this, and only defines the additional behavior, both in the overview clause, and in the specifications of operations. Thus only the new method, `max`, and the `intset_max` constructor have been given specifications; all other methods have the specifications provided for them in the specification of `intset`.

The second point to note about `intset_max` is that headers are provided for some of `intset` methods but not for others. Headers *must* be provided for any supertype methods that will be reimplemented in the subtype implementation; headers *must not* be provided for any supertype methods that will not be reimplemented in the subtype implementation. Thus the class declaration for `intset_max` indicates that we intend to reimplement `insert`, `remove`, and `rep_ok`, but not any of the other methods. (As usual, a class declaration includes implementation information in addition to specification information!)

As mentioned in Section 10.1, subtypes must have all the methods of supertypes, with expected signatures. C++ guarantees that this condition is satisfied. The subclass is allowed to define methods with the same names as superclass methods; these are the *overridden* methods. The subclass declaration of an overridden method must have the same signature as the superclass method or there will be a compiler warning. You should pay attention to such a warning; it means code written in terms of the supertype will break when using an objecct of the subtype. (You can provide additional methods with the same name but different signatures as the overridden method; an example of such overloading will be given in Section 10.5.)

Any superclass methods that are not overridden become methods of the subclass automatically; these are the *inherited* methods. In addition, the subclass can contain headers for additional methods not in the superclass; these are the *extra* methods.

Actually, C++ does not quite guarantee that the signatures of the overridden methods are right since it ignores throw clauses. Therefore the definer of a subtype must be careful to do the right thing: when a subtype overrides a method, the overriding definition may not throw any exceptions not listed in the throw clause of the overridden definition. The reason for this rule is that, as discussed further in Section 10.5, we want users to be able to reason about runtime behavior based on the supertype specification. Therefore, only the exceptions listed there will be expected.

A public subclass should only override virtual superclass methods. This constraint makes sense because the compiler will bind directly to the code of non-virtual superclass methods (in programs that use the supertype). Therefore overriding non-virtual methods would have no effect on runtime behavior.

Typically a subtype object will be created and used in code that knows it is dealing with the subtype. That code will be able to make use of the extra methods. But in addition, the object will be passed as an argument to code written in terms of the supertype, and that code will use only the supertype methods.

10.2.2 Implementing a Subtype

The representation of a subclass object consists of the instance variables declared for the superclass *and* those declared for the subclass. Furthermore, to implement the subclass, it may be necessary to access the superclass instance variables.

For example, an easy way to implement `intset_max` is to allow `intset` to keep track of the elements of the set. However, to make it easy to keep track of the maximum element, it is desirable for the subclass to have an instance variable:

```
int biggest; // stores the maximum element (if the set is not empty)
```

Thus `intset_max` objects have two instance variables, `els` (from the superclass), and `biggest` (from the subclass). When a new element is inserted, if it is bigger than the current `biggest`, the value of `biggest` is changed. When an element is removed, if it is the `biggest`, we need to reset `biggest` to hold the new maximum. This latter activity can be accomplished using the `elements` iterator.

When you design a class that is intended to have subclasses you need to think about the interface it provides to these subclasses. This interface should encapsulate information about the class's implementation, since that will allow the implementation to be changed later. The interface must also provide subclasses with enough power that they can be implemented efficiently.

It's always best if subclasses can interact with superclasses entirely via the public interface since this preserves full abstraction and allows the superclass to be reimplemented without affecting the implementation of the subclass. If the public interface is not sufficient, we can provide additional access in special section of the superclass declaration called the *protected* section. Instance variables and operations declared in this section are visible to subclasses but not to users.

Providing access via protected methods and constructors is usually better than providing direct access to instance variables because then you can change implementation details later. However, it does reduce abstraction over just using the public interface. For example, it might seem harmless to provide access to the `els` array by a protected `fetch` method

```
virtual int fetch (int i) const throw (CE_Bounds);
// effects: if i is in bounds of els, return els[i], else throw CE_Bounds
```

However, having this method means the superclass implementation must implement it efficiently, which certainly reduces flexibility in choosing the implementation.

In the case of `intset_max` the public interface of `intset` is sufficient. We will give examples of protected access in later sections.

Figure 10.4 shows the implementation of `intset_max`. First note the implementation of the constructor. When a constructor of a subclass is called, C++ automatically calls the constructor of the superclass before entering the code of the subclass. Therefore by the time the `intset_max` constructor starts to run, the `els` array has already been initialized. At this point there is no work for the `intset_max` constructor to do, since `biggest` has no value when the set is empty.

To implement the methods of `intset_max`, we need to make use of the overridden methods of `intset`. For example, for `insert`, we want `intset::insert` to do the actual work; all `intset_max::insert` does is update `biggest`. Within a subclass all the overridden methods of the superclass are available. However, there is a naming issue: how do we distinguish the overriding method (the one being implemented for `insert_max`) from the overridden method. C++ resolves this issue by requiring the compound form, e.g,, `intset::insert`, to name the overridden method; a name without the prefix, e.g., `insert`, means the overriding one. Examples of uses of overridden methods occur in the implementations of `insert`, `remove`, and `rep_ok`. Note that the only checking done directly in `intset_max::rep_ok` is that `biggest` is the largest element when the set is not empty; all the other checking is done in `intset::rep_ok`. In other words, the details of the `intset` representation are localized to the `intset` implementation, as we wanted.

Although the overridden methods are visible to the subclasses, they are *not* accessible to users of subclass objects. Thus there is no way a user of `intset_max` can call the `intset::insert` method.

The rep invariant and abstraction function for a subclass are typically defined in terms of those for the superclass. Thus we have

```
// The abstraction function is:
//   A_intset_max(c) = A_intset(c)
```

Here we have introduced notation to distinguish the two abstraction functions; the one for `intset` is `A_intset`, and the one for `intset_max` is `A_intset_max`. In this example, the `biggest` field does not affect what set an `intset_max` object represents; therefore `A_intset_max` simply returns the same set as `A_intset`. (Note that it is reasonable to apply `A_intset` to an `intset_max` object since such an object has all the `intset` instance variables.)

The rep invariant for `intset_max` is also defined using the rep invariant for `intset` (and also the abstraction function for `intset`):

```
// the rep invariant is:
//   I_intset_max(c) = I_intset(c) & (size(c) > 0 =>
//      [c.biggest ∈ A_intset(c) & ∀ x ∈ A_intset(c) [x ≤ biggest] ]
```

A subclass implementation should *only* make use of methods and other accessible information of direct superclasses and not superclasses of super-

```
class intset_max: public intset {
// as in Figure 10.3
private
   int biggest; // the biggest element if set not empty
}; // end intset_max

intset_max::intset_max ( ) throw ( ) { }

void intset_max::insert (int x) throw ( ) {
   if (size ( ) == 0 || x > biggest) biggest = x;
   intset::insert(x); } // end insert

void intset_max::remove (int x) throw ( ) {
   if (size( ) == 0) return;
   intset::remove(x);
   if (x == biggest && size( ) > 0) {
      generator<int>*g = elements( );
      int v;
      g → get(biggest);
      while (g → get(v))
         if (v > biggest) biggest = v; }
   } // end remove

int intset_max::max ( ) const throw (CE_Empty) {
   if (size( ) == 0) throw ("call of intset_max::max on empty set");
   return biggest; } // end max

bool intset_max::rep_ok ( ) const throw ( ) {
   if (size( ) == 0) return intset::rep_ok( );
   bool max_in = FALSE;
   generator<int> *g = elements ( );
   val v;
   while (g → get (v)) {
      if (v > biggest) return FALSE;
      if (v == biggest) max_in = TRUE; }
   return (max_in && intset::rep_ok( )); } // end rep_ok
```

Figure 10.4 Implementation of intset_max.

classes. C++ does *not* enforce this constraint. From a methodological point of view, however, the constraint makes sense, since the purpose of a subclass is to support its specification by extending the implementation of its superclass. Furthermore, the constraint hides information so that changes can happen later. In particular we can change the hierarchy and implementation above the superclass without having to worry that this will affect its subclasses.

10.2.3 Casting Down

When there is a type hierarchy, using code sometimes needs to find out the actual type of an object. For example, suppose using code has an `intset*` variable `s`, and it needs to find the largest element of `s`. If `s` is actually an `intset_max` object, this can be determined by calling the `max` method; otherwise it must be computed.

Many C++ compilers provide no way to find out the actual type of an object at runtime. Therefore we need to provide this ability by means of an extra method called `force` with the following specification:

```
virtual void *force (string s) const throw (CE_Wrongtype);
  // effects: if s is the name of my type or one of my supertypes,
  //    return a pointer to this else throw CE_Wrongtype.
```

Every base type and subtype will have a `force` method. Thus `intset` and `intset_max` must each have a `force` method; the header for the method needs to be added to their class declarations (in Figures 10.2 and 10.3). It is not necessary to give a specification for `force`, however; instead we rely on a "standard" specification — just as we did for `rep_ok`.

In the case of a base type such as `intset`, the implementation of `force` simply checks for a direct match, e.g.:

```
void * intset::force(string s) const throw (CE_Wrongtype) {
  if (s == "intset") return this; else throw CE_Wrongtype;
    } // end force
```

In the case of a subtype, e.g., `intset_max`, if the string argument does match, the `force` method passes the buck to the superclass, e.g.:

```
void * intset_max::force(string s) const throw (CE_Wrongtype) {
  if (s == "intset_max") return this; else return intset::force(s);
    } // end force
```

Of course the superclass might also pass the buck — to its superclass. Thus this approach obeys our constraint of having the hierarchy above a superclass be hidden from its subclasses, yet it allows matching at intermediate levels of a hierarchy. For example, suppose `intset_max` had a subtype `intset_max_plus`, and a user of `s` with apparent type `intset` wanted to know whether the actual object was an `intset_max`. The user would thus call `s -> force(''intset_max'')`. If the actual object were of type `intset_max_plus` the call would go to its `force` method, where there would be no match; then the call would go to the `force` method of `intset_max` and there would be a match.

When the `force` succeeds we get back a `void*` pointer which can then be cast to the needed type, e.g.,

```
try {intset_max *m = (intset_max*) s → force ("intset_max");
  //use max }
  catch (CE_Wrongtype) {//compute max}
```

The assignment to `m` is a cast (from the `void*` to `intset_max*`). Note how the code determines whether the cast is legal by using `force`; in this way it determines that the cast is safe. Sometimes it won't be necessary to do the cast. For example, suppose `intset` had a subtype `sorted_intset` in which the `elements` iterator produced the elements in sorted order. Then we might have the following code:

```
generator<int> *g = s → elements( );
try {s → force("sorted_intset");
     // in here can assume sorted order }
  catch (CE_Wrongtype) { // in here cannot assume sorted order }
```

Doing the `force` is useful because code based on the assumption of sorted order can be more efficient that code not based on this assumption.

Even when we already know the actual type of an object, `force` must be used in order to properly convert to a pointer of the type. It might seem that we could just use a C++ cast to change the type of the pointer in this case, but this does not work for technical reasons concerning how dispatching is implemented in some compilers.

10.2.4 Polymorphic Hierarchy

It is possible to combine polymorphism and type hierarchy. For example, instead of having `intset` and `intset_max`, we could have `set` and `set_max`. In this case the subtype would itself be parameterized; for any type `foo`, `set_max<foo>` would be a subtype of `set<foo>`. It is also possible to define a subtype of a particular instantiation, e.g., `intset_max` might be a subtype of `set<int>`.

For a polymorphic type, the `force` method does not give information about particular instantiations. For example, consider the following code:

```
class C6170Exception {
public:
// overview: exception objects have two fields, name, which contains the string
//   name of the type of the object, and info, which contains optional
//   information about the reason for the exception.
//   Exception objects are immutable and are on the stack.

// constructors:
  C6170Exception ( );
  // effects: makes this have name "C6170Exception" and info "none".

  C6170Exception (string s);
  // effects: makes this have name "C6170Exception" and info s.

// methods:
  string name ( );
  // effects: returns the name of this.

  string info ( );
  // effects: returns the info of this.

protected:
  C6170Exception (string s, string n);
  // effects: makes this have name n and info s.

private:
  string n; // the name
  string i; // the info
}; // end C6170Exception
```

Figure 10.5 Specification for C6170Exception.

```
set<int> * s;
try { set_max<int> *m = (set_max<int>*) s → force ("set_max")
      ... }
```

The call on `force` is able to tell us the `s` actually refers to a `set_max`, but not what kind of element is in the set. However, this information is readily available in the using code.

10.3 Exception Types

```
class CE_foo: public C6170Exception {
public:
// constructors:
  CE_foo ( );
  // effects: makes this have name "CE_foo" and info "none".

  CE_foo (string s);
  // effects: makes this have name "CE_foo" and info s.
}; // end CE_foo

// implementation of CE_foo
CE_foo::CE_foo ( ): C6170Exception("none", "CE_foo") { }
CE_foo::CE_foo (string s): C6170Exception(s, "CE_foo") { }
```

Figure 10.6 Class and implementation for CE_foo.

Now we have introduced enough material to explain our hierarchy of exception types. The exception types we use in programs are all subtypes of `C6170Exception`, the base type of the hierarchy.

The specification and representation of `C6170Exception` are given in Figure 10.5. Its implementation provides instance variables to store the `name` and `info` fields and implements the methods that provide access to this information. These methods are *not* virtual: they are implemented in the superclass and are not intended to be overridden by subclasses. These subtypes simply provide their own constructors and inherit the `name` and `info` methods from `C6170Exception`. The subtypes need a way to initialize the `name` and `info` fields; the protected constructor of `C6170Exception` provides this ability.

For example, the class declaration and implementation for `CE_foo` are given in Figure 10.6. What is shown in the figure is what is provided by using the macro `MAKE_EXCEPTION(''foo'')`. The main point to notice here is how the `CE_foo` constructors use the protected `C6170Exception` constructor to initialize `this`. The call to the `C6170Exception` constructor occurs in the header, before the body of the `CE_Foo` constructor (and in fact the body is empty). As mentioned earlier, before calling the code of a constructor for a subtype, C++ calls a constructor of the supertype. It will call the default constructor by default; the subtype can override this by indicating explicitly what constructor to call, as shown in the figure.

Sometimes it is useful for exception types to provide additional information, similar to the results returned by procedures that terminate normally. This can be accomplished by defining the subclass yourself. Such a subclass will have fields of its own (to store the extra information), its constructors will take additional arguments (the extra information), and it will provide additional methods (to access the extra information).

The exception type family is rather unusual. The reason for having the family is to provide lots of exception types, so that we can do proper discrimination in catch clauses. The objects of these exception types are all implemented alike (except if an exception subtype provides extra fields and methods).

10.4 Abstract Classes

Suppose instead of just keeping track of the maximum element of the `intset`, we wanted to provide access to the elements of the set in sorted order. The type could be a subtype of `intset` or even of `intset_max`; it would still provide the `max` method and, in addition, it would redefine the `elements` iterator.

To implement `sorted_intset` we might like to use an ordered list. However, if `sorted_intset` is a subtype of `intset` as defined in Figure 10.2, we have a problem. Every `sorted_intset` object will contain within it instance variables inherited from `intset`. These instance variables are no longer interesting, since we do not want to keep elements of a `sorted_intset` in the `els` array.

The problem we have run into here is that we want to inherit the behavior of `intset` without also inheriting its instance variables. The only way to avoid inheriting instance variables is for `intset` to not have any. This means that `intset` can't actually have any objects. (In general a class without instance variables can have objects, but in the case of `intset` there would be no objects since we need instance variables to record the set elements.)

A class without any objects is called an *abstract* class. Such a class might actually provide implementations of some methods, and it might have some instance variables. However, it is defined in the expectation that part of the implementation will be provided by subclasses.

An abstract class contains one or more *pure methods*. A method is pure if its declaration ends in `= 0`; this means the method *cannot* be implemented when the class is implemented. Instead, implementations will be provided by subclasses. A class with pure methods must be abstract, i.e., have no objects, since if such objects existed, they would be missing some methods. An abstract class may have constructors, but these cannot be called by using code (such a call will lead to a compile-time error); instead the constructors may only be called as part of constructing subclass objects.

An example of an abstract class declaration is given in Figure 10.7, which shows the headers of an abstract class for `intset`. Here all methods except for `is_in` and `force` are pure. The `force` method will never be pure; each type is responsible for checking for a match with its name. Note the

```
class intset {
public:
// methods:
   virtual bool is_in (int x) const throw ( );
   virtual int size ( ) const throw ( );
   virtual bool subset(intset* s) const throw ( );
   virtual void* force(string s) const throw (CE_Wrongtype);
// pure methods:
   virtual void insert (int x) throw ( ) = 0;
   virtual void remove (int x) throw ( ) = 0;
   virtual generator<int> * elements ( ) const throw ( ) = 0;
   virtual bool rep_ok( ) const throw ( ) = 0;
protected:
   int sz; // the intset size
private:
   void operator = (intset const &); // disallowed
   intset (intset const &); // disallowed
}; // end intset
```

Figure 10.7 Declaration for abstract intset.

convention of giving the declarations of the non-pure methods before those of the pure methods. Note also that we have not disallowed the default constructor, since any attempt by using code to call it is automatically an error.

The implementation of an abstract type must implement all the non-pure methods. It may seem that there is no point in an abstract class having any non-pure methods (other than `force`) but this is not true. Implementing methods in the superclass is good: we implement them just once, even though there may be many subclasses. Often they can be implemented abstractly, by making calls on the other methods. For example, `is_in` can be implemented using `elements`:

```
bool intset::is_in (int x) throw ( ) {
   generator<int> * g = elements ( );
   int v;
   while (g → get(v)) if (x == v) return TRUE;
   return FALSE; } // end is_in
```

The `elements` iterator is sufficient to allow `is_in` to be implemented efficiently (in general, iterators often provide the needed power to allow abstract implementations). It is not sufficient to implement `size`, however. In this case we need an instance variable, `sz`, and need to decide whether to make it accessible to subclasses, or to hide it, providing access through

```
class sorted_intset: public intset {
public:
//overview: sorted sets are sets of integers providing access in sorted order.

//constructors:
  sorted_intset ( ) throw( );
  // effects: makes this be the empty set

//methods:
  virtual generator<int>* elements ( ) const throw ( );
  // effects: returns a generator that produces the elements of this,
  //   each exactly once, in sorted order.
  // requires: this must not be modified while the generator is in use.

  virtual int max ( ) const throw (CE_Empty);
  // effects: if this is empty throws CE_Empty else returns its largest element.

  virtual void insert (int x) throw ( );
  virtual void remove (int x) throw ( );
  virtual bool subset (intset * s) const throw ( );
  virtual bool rep_ok ( ) const throw ( );
  virtual void * force ( ) const throw (CE_Wrongtype);
private:
  void operator = (sorted_intset const &); // disallowed
  sorted_intset (sorted_intset const &); // disallowed
  ordered_list<int> * els; // the elements in sorted order
}; // end sorted_intset
```

Figure 10.8 Specification of Sorted Intset.

protected methods. If we hide it, `intset` can maintain the invariant `sz >= 0`. However, this is quite uninteresting: what matters is that `sz` is the size of the set, and that can only be maintained by the subclasses. Therefore, we will allow subclasses direct access to `sz`; this is why its declaration is in the protected section of the `intset` class.

The specification of `sorted_intset` is given in Figure 10.8. The main point of interest is that specifications are provided for `max` and the constructor, and also for `elements`. The reason `elements` is specified is that its behavior has changed: `intset::elements` produces the elements in arbitrary order, while `sorted_intset::elements` produces them in sorted order. Also note that no header is provided for `is_in` since it is inherited from `intset`. Finally note that we override `subset`. The reason is that when `this` and `s` are both `sorted_intset`s the implementation can be order `n` rather than order n^2. The implementation can use `force` to determine whether the argument is a `sorted_intset`.

10.5 The Meaning of Subtypes

To define subtypes properly requires an understanding of what it means for a subtype specification to be correct. Note that the question here concerns the specification of the subtype, not its implementation: at this point we are interested in the relationship between the subtype specification and the specification of its supertype(s).

To understand what behavior is required of subtypes, we need to consider what we hope to accomplish with them in a program. The goal is for programmers to be able to write code and reason about that code in terms of a supertype, i.e., using the supertype specification. When the code runs, the objects it uses may belong to subtypes; nevertheless, we want the code to behave just as it would have if it used supertype objects, and we want the reasoning done based on the supertype specification to still be valid. For example, the **`stream`** hierarchy is used in this way; I/O routines such as **`operator`** $\ll$ are written in terms of `ostream`, but they work properly when using objects belonging to subtypes of `ostream`.

To support reasoning using the supertype specification requires:

1. The subtype objects must have all the methods of the supertype.

2. These subtype methods must have *compatible* signatures to those of the corresponding supertype methods.

3. Calls of these subtype methods must also "behave like" calls to the corresponding supertype methods.

4. Any extra methods provided by the subtype, or extra information obtained from constructors or producers of the subtype, must not cause a detectable change in the behavior of objects as observed via calls of the superclass methods. (Recall that extra methods are the ones the subtype provides in addition to those of the supertype).

The first two requirements guarantee that every call made using the specification of the supertype will be legal if the actual call goes to a subtype object. These requirements can be checked by a compiler. The C++ compiler does enforce the first requirement and it almost enforces the second, but unfortunately it ignores throws clauses. Therefore the definer of a subtype is responsible for ensuring that the subtype method throws *at most* the exceptions that can be thrown by the corresponding supertype method. Note that this requirement implies that if the supertype method throws nothing, then so must the corresponding subtype method.

In C++, exact equality of signatures is required, i.e., the subtype and supertype methods must have identical signatures. However, the subtype can always provide extra methods that overload the supertype method name if this is needed. (Other programming languages may allow a more permissive rule: arguments of subtype methods can be supertypes of what the corresponding supertype method requires (this is called *contravariance*), and results of subtype methods can be subtypes of what the supertype method requires (this is called *covariance*). However, overloading gives you almost the same power.)

The third and fourth requirements guarantee that programs written in terms of the supertype won't notice different behavior when using objects of the subtype. To write these programs, programmers rely on facts they can deduce by reasoning about the supertype specification. There are two kinds of reasoning: reasoning concerning specific method calls, and reasoning concerning all future states.

Examples of the first kind of reasoning are the following:

1. For any `intset`, if we call `insert` and then `size`, the size of the set will have increased at most by one.

2. For any `intset`, if we call `remove(3)` followed by `is_in(3)`, the answer will be "false."

This kind of reasoning is entirely concerned with specific sequences of calls using superclass methods. Of course, when the objects concerned belong to subclasses, the calls actually go to the code provided by the subclass. Requirement 3 says that all deductions of this sort are valid even though the subclass code is running.

All the examples given so far have obeyed the third requirement. In fact, our subtype methods all have exactly the same specification as the corresponding supertype method, with one exception, the `elements` method of `sorted_intset`. Whenever a method is re-specified, there is a potential for doing things wrong. In the case `elements`, the new behavior is acceptable because we have taken advantage of the non-determinism in the `intset::elements` specification. That specification allows various orders for producing the elements; one of these orders is the sorted order produced by `sorted_intset::elements`. When we give new specifications for supertype methods in subtypes, we often take advantage of non-determinism like this.

However, suppose that we had defined a subtype of `intset` in which we redefined `insert`:

```
void insert ( int x) throw ( );
// effects: if x is odd adds it to this else does nothing
```

In this case we have violated the requirement. A program written in terms of the `intset` specification would clearly expect even numbers to be added to the set as well as odd ones!

Mistakes like this definition of `insert` are easy to see, but sometimes things are more subtle. For example, one might think that `intbag` is a subtype of `intset`: it has all the `intset` methods (plus some extra ones) with what appear to be identical specifications. Thus the specification of the `insert` methods for both `intset` and `intbag` is

```
void insert(int x);
  // modifies: this
  // effects: adds x to this
```

Yet the method behaves differently in the two cases. The `insert` method for `intset` does not add its argument `x` to the set if it is already in the set; the `insert` method for `intbag` does add `x` to the bag. Furthermore, the difference in behavior in noticeable, e.g., by calling the `size` method. The fact is that the specifications are different in the two cases, even though they "look" the same, because they interpret the word "add" differently; for sets it means set union, while for bags it means multiset union.

Extra methods can sometimes cause this kind of proof to fail. For example, suppose suppose a subtype of `intset` had an extra method

```
void insert_dup (int x) throw ( );
// modifies: this
// effects: inserts two copies of x in this
```

yet the specification of `remove` removes only one copy of its argument. Note that this kind of problem can also arise from extra creators and producers. For example, the same problem could occur if the purported subtype had a constructor that initialized `this` to contain two copies of some integer.

Extra methods can also cause problems with the other kind of proofs, the ones that concern all future states. Some of these are abstract invariants; for example, we can deduce that the size of an `intset` is always non-negative. Here is another example. Suppose we had a type `simple_set`, which just had methods `insert` and `is_in`. For such a type we can prove that simple sets never shrink: for all simple sets `s`, if we discover that some integer `x` is in `s` at some point, then we know that `x` will be in `s` at any future point. Since `intset` violates this condition, it is not a subtype of `simple_set`. It is the mutator method `remove` that causes the problem; all violations of this kind of proof are caused by mutators, since only mutators cause state changes.

Extra and overriding mutators are allowed to cause changes to subtype objects that wouldn't have occurred in supertype objects, but these changes cannot be observable by supertype methods; instead they can only be observed by extra subtype methods. For example, consider a type `intset_dup` that maintains two sets of elements, a *main* copy and a *shadow* copy. Extra method `insert_dup` adds `x` to both copies and other `intset` mutators also modify both copies, e.g., `insert` might add `x` only to the main copy, but `remove` might remove `x` from both copies. Code written in terms of `intset_dup` can observe the shadow copy via extra observers, e.g., `is_in_shadow` can be used find out about elements of the shadow. This subtype will be permissible provided all `intset` observers when called on `intset_dup` objects only provide information about the main copy, since in this case the presence of the shadow copy is invisible to any code written in terms of `intset`.

As a final example, suppose we have two types, `point2`, which represents points in two-space, and `point3`, which represents points in three-space, and suppose `point3` is defined to be a subtype of `point2`. `point2` has the following operations:

```
point2(int x, int y); // effects: makes this have coordinates x and y
virtual float get_x( ); // effects: returns x coordinate
virtual float get_y( ); // effects: returns y coordinate
virtual bool equal (point2 * p);
  // effects: returns true if this and p have the same coordinates.
```

and suppose its subclass `point3` declares

```
point3 (intx, int y, int z); // effects: makes this have coordinates x, y, and z
virtual float get_z( ); // effects: returns z coordinate
virtual bool equal (point2 * p);
  // effects: returns true if this and p have the same coordinates
```

I.e., `point3` overrides `equal`, inherits the other methods, and adds the `get_z` method. It overrides `equal` so that it can test for equality in 3-space.

Now consider the following code, where `p` and `q` are declared to be `point2`:

```
if (p → get_x( ) == q → get_x( ))
  if (p → get_y( ) == q → get_y( ))
    check (p → equal(q)), "error: two equal points found to be unequal");
```

If `p` and `q` are actually `point3` objects that differ in the `z` coordinate, the specification of `point3::equal` given above indicates that it should return `false`. But this response seems erroneous from the perspective of the code above, since it just determined (based on the `point2` specification) that the two points are identical.

The problem here is that the specification of `equal` should actually be different in the two types. If you compare two `point3` objects in a context in which the caller knows they are objects in 3-space, you do want to consider all three coordinates; otherwise, you should consider only the `x` and `y` coordinates. This problem can be solved by providing two `equal` methods in `point3`.

```
virtual bool equal (point2 * p);
  // effects: returns true if p and this have the same x and y
  //     coordinates else returns false

virtual bool equal (point3 *p);
  // effects: returns true if p and this have the same x, y, and z
  //     coordinates else returns false
```

The first will be override the `point2` method, and its implementation will simply call the overridden method. The other is an extra method. A call `x -> equal(y)` will go to the extra method only when the apparent types of both `x` and `y` are `point3`. For example,

```
point2 * p2 = new point3(x, y, z);
point3 * p3 = new point3(x, y, z);
if (p2 → equal (p2)) // calls overridden method
if (p2 → subset (p3)) // calls overridden method
if (p3 → subset (p3)) // calls extra method.
```

Thus in the situation above, the right thing will happen, and the right thing will also happen when code is written in terms of `point3`; in this case the code can observe the difference in the `z` coordinates, and the call to `equal` will return a result consistent with that observation. (Note that we cannot get the right behavior by having a single `point3::equal` which uses `force` to do the 3-space comparison if the argument is a `point3` since we want the answer to depend on the apparent type, not on the actual type.)

10.6 Multiple Implementations

Type hierarchy can be also used to provide multiple implementations of a type. This use can be thought of as defining a very constrained type family, in which all members have exactly the same methods and behavior.

For example there might be both sparse and dense implementations of polynomials. Furthermore, in a program that uses polynomials, it might be desirable to use both implementations. This allows each `poly` to be represented in the way that is best for it. Yet one wants to consider objects from different implementations as belonging to the same type.

When inheritance is used to provide multiple implementations, the superclass will be abstract: the whole point is to defer implementation details to subclass. Furthermore, the subclasses will provide exactly the same methods as the superclass, with exactly the specifications defined in the superclass. The subclasses thus provide *no* additional behavior, except that they must provide constructors. Some methods might be implemented by the superclass; in fact the superclass might even have some instance variables. But the bulk of the implementation will be provided by the subclasses.

For example, Figure 10.9 gives the class declaration for the `poly` base class. (The `exps` iterator produces the exponents of all the terms with non-zero coefficients, in order up to the degree.) Note that although most methods are pure, some are not; the figure also provides implementations of the non-pure methods. Note that we have elected to keep the degree as an instance variable of poly, since this is useful information for all poly subclasses. Furthermore, we have made `deg` protected, so that subclasses can access it directly, although we have provided a constructor to initialize `deg`. We provide direct access to `deg` because `poly` cannot by itself preserve any interesting rep invariant on it.

The class declaration of `dense_poly` is given in Figure 10.10. The implementation of `dense_poly` is similar to what we saw before (in Figure 4.8), including the use of `deg`. The main difference is that we don't need to implement the methods provided by the superclass.

The implementation of the `dense_poly` constructor is:

```
dense_poly::dense_poly(int c, n) throw (CE_Neg): poly((c==0)?0:n) {
  if (n < 0) throw CE_Neg;
  trms = new array<int> ( );
  if (c == 0) {trms → append(0); return;}
  for (int i = 0; i < n; i++) trms → append(0);
  trms → append(c); }
```

The header contains a call to the `poly` constructor — in an if expression that passes in the right information provided `n` is not negative Therefore, the `dense_poly` constructor need not initialize `deg` directly.

When type hierarchy is used to define multiple implementations, the subtypes are largely invisible to users. The only place where users need to be aware of subtypes is when they create new objects. At that point the code must call the constructor of the appropriate subtype. For example the programmer of code that uses `poly` must decide whether to create a sparse or dense poly whenever a new poly object is created.

We can use subclasses to define multiple implementations for parameterized types. For example, we could provide implementations of `set<T>` that are intended to be used for small sets (`small_set<T>`) and large sets

```
class poly {
public:
// constructor:
   poly (int n) throw ( ); // sets degree to n
// methods:
   virtual int degree ( ) throw ( );
   virtual poly* sub (poly* p) throw ( );
   virtual poly* equal (poly* p) throw ( );
   virtual void* force ( ) throw (CE_Wrongtype);
// pure methods:
   virtual int coeff (int n) throw ( ) = 0;
   virtual poly* add (poly* p) throw ( ) = 0;
   virtual poly* mul (poly* p) throw ( ) = 0;
   virtual poly* minus ( ) throw ( ) = 0;
   virtual generator<int> * exps ( ) throw ( ) = 0;
   virtual bool rep_ok ( ) throw ( ) = 0;
protected:
   int deg; // the degree of the poly
private:
   void operator = (poly &); // disallowed
   poly (poly &); // disallowed
   poly ( ); // disallowed
}; // end poly

poly::poly (int n) throw ( ) { deg = n; }

int poly::degree ( ) throw ( ) { return deg }

poly* poly::sub (poly* q) throw ( ) { return add(q → minus( )); }

bool poly::equal (poly* q) throw ( ) {
   if (deg != q → deg) return FALSE;
   generator<int> * gp = exps ( );
   generator<int> * gq = q → exps ( );
   int np, nq;
   while (gp → get(np), gq → get(nq)) {
      if (nq != np || coeff(np) != q → coeff(nq)) return FALSE; }
   return TRUE; }

void * poly::force (string s) throw (CE_Wrongtype) {
   if (s == "poly") return this; else throw CE_Wrongtype; }
```

Figure 10.9 Poly base class and implementation.

```
class dense_poly: public poly {
public:
// constructor
  dense_poly(int c, n);

// methods
  virtual int coeff (int n) throw ( );
  virtual poly* add (poly* p) throw ( ;
  virtual poly* mul (poly* p) throw ( ;
  virtual poly* minus ( ) throw ( );
  virtual generator<int> * ( ) throw ( );
  virtual void* force ( ) throw (CE_Wrongtype);
  virtual bool rep_ok ( ) throw ( );
private:
  array<int>* trms; // the coefficients up to the degree
  void operator = (poly &); // disallowed
  dense_poly (poly &); // disallowed
  dense_poly ( ); // disallowed
}; // end dense_poly
```

Figure 10.10 Declaration of dense_poly.

(`hashed_set<T>`). In addition we could provide implementations for specific instantiations. For example we might have

```
class char_set: public set<char> { ... };
```

This class could use a very efficient implementation: a 256-bit array (to handle 8-bit `chars`).

10.6.1 Constructing New Objects

The polynomial example illustrates a problem that can arise when hierarchy is used to define multiple implementations; it concerns methods that construct new objects. Polynomials have several such methods, the arithmetic methods. The problem we are concerned with is: when we add (or multiply) two polynomials, should the resulting `poly` be dense or sparse.

Note that there are two issues that must be resolved here: how does the code decide what to do, and if it decides that it must create a representation belonging to some other class, how does it do this? Note also that underlying these questions is a basic fact: to do a good job of multiple implementations, it is often necessary to know what the implementation possibilities are. Thus in implementing `dense_poly` it is not sufficient to just know about this class and its superclass; we also need to know about the other subclasses of `poly`.

Once we acknowledge that different implementations know about each other, we can accomplish what we need using mechanisms that have already been described. For example, in implementing the `add` method we might decide to do the addition using the representation of `this`. Then at the end, we decide whether the newly-created `poly` has the right rep or not, and if it doesn't we convert it to the other rep. The decision might be based on the relationship between the number of non-zero coefficients and the degree. For example, if we are in the dense implementation, and we discover that the new `poly` has mostly zero coefficients, we would decide to produce a sparse representation instead. Similarly, in the sparse implementation we will convert to the other representation if the new poly has lots of non-zero coefficients for terms below the degree.

These conversions mean that `dense_poly` must have an efficient way of creating a `sparse_poly` and vice versa. This can be accomplished by having each subclass have an additional constructor. Here is one possible way of defining such a constructor. Suppose `poly` has the following (public) iterator method:

```
typedef struct{int coeff, int exp} pair;
virtual generator<pair> *terms( ) = 0;
  // effects: returns a generator that will yield each exponent of
  //    of a non−zero term with its associated coefficient, in
  //    order of increasing exponent
```

In addition, each subclass has a constructor that takes the generator produced by `terms` as an argument. For example, here is the specification of the constructor for `sparse_poly`:

```
sparse_poly (generator<poly::pair> * g);
  // effects: initializes this to contain the terms produced by g
```

Of course, to implement the subclasses, each must know about the other. This can be accomplished by having their source include the header file of the other.

One final point about this example: It might be nice to completely hide the existence of the subclasses of `poly` from users, since it is not clear they should be burdened with making decisions about dense vs. sparse representations for `polys`. If we do this, however, users will need a generic way to create new monomials. This can be provided by a procedure that creates new `polys`, e.g.,

```
poly * make_poly(int c, int n);
```

```
// effects: if n < 0 returns the zero poly else returns
//     the monomial cx^n.
```

The procedure might choose between a sparse and dense representation based on what `n` is. It is related to `poly` and would be declared in the same header file as `poly`.

Other implementation hierarchies can also use this approach of hiding the implementation details by providing similar procedures.

10.7 Generators

Our parameterized type `generator` is actually an abstract type with multiple implementations. Each time we define a public subclass of `generator` we are providing an implementation for it.

Sometimes we implement just a single instantiation of `generator`; for example, the implementation of `elements` for `intset` will use a class that is a subclass of `generator<int>`. Other times we provide an implementation that is itself parameterized; for example this is what happened for ordered list (see Figure 7.12); the `elements` iterator was defined using:

```
template<class T>
  class olist_gen: public generator<T> {
  ...
  }; // end olist_gen
```

Each instantiation of `olist_gen` is an implementation of the related instantiation of `generator`. Thus `olist_gen<int>` is an implementation of `generator<int>`, etc.

10.8 Subtype vs. Parametric Polymorphism

Both type hierarchies and parameterized types achieve similar things: they define a class of related data abstractions. Furthermore both allow the writing of generic code that works for many different types. However, there are important differences between the two mechanisms that determine when each should be used.

Parametric polymorphism should be used to define groups of related, homogeneous collection types, e.g., sets, bags, ordered lists, etc. For example, the parameterized `bag` abstraction allows us to define `bag<int>`, `bag<char>`, and so on. Each particular bag type is homogeneous, yet we are able to define all bag types with a single definition.

It is possible to define collections like this using inheritance. For example, the supertype could be heterogeneous bags, and the subtypes would

```
class T {
public:
  int foo ( );
private:
  array<int> * els;
}; // end T

class S: private T {
public:
  int foo ( );
  int bar ( );
private:
  int z;
}; // end S
```

Figure 10.11 Private Inheritance.

each define a particular kind of element, e.g., `intbag` would be the subtype in which all elements are integers. There are a number of problems in doing things this way. One important one is that it's clearly very inconvenient to have to define all these subtypes! In fact, you will never be done defining them. Every time a new type `foo` is defined, you may need to define `foobag` as well.

On the other hand, you should use a type hierarchy when you want multiple implementations, or when you expect there to be subtypes that all have the same base behavior, but that extend that behavior in various ways.

10.9 Private and Protected Subclasses

As mentioned, it is also possible to declare a *private subclass.* The reason for doing this is *only* to share code. Objects of a private subclass *cannot* be viewed as belonging to a subtype of the type defined by the superclass. For example, consider the situation shown in Figure 10.11. Here `T` is the superclass and `S` is the private subclass. (Actually, C++ does not require the use of the word `private`; the *lack* of the word `public` means the subclass is private.)

The rules for private subclasses are a little different from those for public subclasses. First, using code cannot use `S` objects in contexts that accept a `T *` or `T &`. For example, the following is illegal:

```
S * x;
```

```
T * y;
y = x;
```

The C++ compiler does not allow uses like this; an attempt to use `S` objects in a `T*` or `T&` context will result in a compile time error.

Second, the inherited superclass methods need not be virtual. (E.g., `S` inherits `foo` from `T` even though `foo` is not declared to be virtual.) This follows because using code written in terms of the type defined by the superclass would never be able to call the implementation of that method given in the subclass, since the subclass doesn't define a subtype of the type defined by the superclass.

Third, the private subclass must include in its public section declarations of all methods that its users can call. Even if one of these methods is provided by its superclass, and is just what is wanted, it needs to be declared. For example, suppose that `T`'s `foo` is exactly the code that is needed for `S`'s `foo`. We still have to declare it in `S`'s class declaration, as shown in the figure. In addition we have to implement it for `S`, e.g.,

```
int S::foo( ) { return T::foo( ) } // end foo
```

Private subclasses really aren't all that useful. You can do just as well using instance variables. For example, rather than making `S` a subtype of `T`, it could just have a `T` instance variable:

```
class S {
  ...
private:
  T * stuff;
  int z;
}; // end S
```

and the implementation of the `foo` method would be:

```
int S::foo( ) { return stuff → foo( ) } // end foo
```

(There appears to be a slight efficiency advantage to private subclasses, because when `S` is a subclass of `T` its rep contains the `T` instance variables, while in the case just shown, its rep contains a pointer to a `T` object. However, in fact the rep of S could contain a `T` rather than a `T*`.)

We will not make use of private subclasses in this course. We will also not use protected subclasses; they are also not very useful.

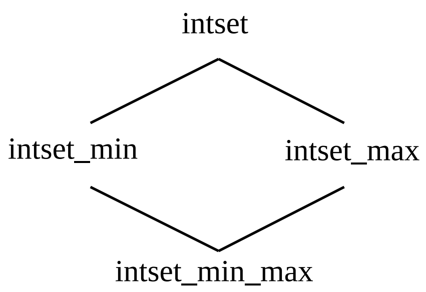

Figure 10.12 Hierarchy with a diamond.

10.10 Multiple Inheritance

In all the examples we have given, a subclass has a single superclass. However, C++ allows a class to have multiple superclasses.

Most of the motivation for multiple superclasses arose because inheritance was being used in several different ways and a particular class might need to be a subclass of one class, say `T`, for one of these reasons, and of another class `T2`, for another reason. However, we are using hierarchy *only* to define type families, and when its use is limited like this the need for multiple superclasses is greatly reduced. We have not eliminated uses just to avoid multiple superclasses; in each case we have a better way of doing things. The two uses we have eliminated so far are:

1. Before there was a template mechanism, inheritance was the only way to define collections whose objects contained elements of distinct types, e.g., `bag`, with subtypes `intbag`, `charbag`, etc. As discussed in Section 10.8, this is not a very good way to define such collections.

2. Private inheritance was used to build on some other class's implementation; as discussed in Section 10.9, this really buys you nothing over just making objects of that class be part of your rep.

A third use of inheritance is to express constraints; we support constraints by using procedure arguments. (Recall that a constraint occurs when a polymorphic abstraction requires some methods from its parameter type.)

If inheritance is used only for defining type families, it is rare that multiple inheritance will be needed. This is good because multiple inheritance has some semantic problems. For example, suppose that in addition to `intset_max`, we also had a subtype of `intset` called `intset_min`, and we then decided to have a subtype `intset_min_max`:

```
class intset_min_max: public intset_min,intset_max {
virtual void insert (int x) throw ( );
... };
```

Now we have a diamond in the hierarchy domain as shown in Figure 10.12. Such a structure raises a number of questions including: (1) is there one copy of the `els` array in the rep of `intset_min_max` objects or two? (2) Does the `insert` method override `insert` from `intset_min` or `intset_max`? (3) Does inherited method `is_in` come from `intset_min` or `intset_max`?

Although all of these questions have answers, they are not always the answers you want! Therefore in general it is good to avoid multiple inheritance, and fortunate that it is almost never needed.

10.11 Summary

This chapter has discussed the C++ inheritance mechanism and how it can be used to define type families and multiple implementations.

The use of type families in program design allows a new kind of abstraction: the designer abstracts from the properties of a related group of types to identify what all of those types have in common. When used appropriately, this kind of abstraction can improve the structure of programs in several ways:

1. By grouping the related types into a family, the designer makes the relationship among them clear, thus making the program as a whole easier to understand. For example, a program that treats the different kinds of windows as a family is easier to understand than one that just has a bunch of different window types because the similarities among the set of types have been carefully delineated.

2. Hierarchy allows the definition of routines and other types that work over the entire family. I.e., a routine works on windows; it will do its job no matter what kind of specialized window is passed it as an argument.

3. Hierarchy provides a kind of *extensibility*. New kinds of subtypes can be added later if necessary to provide extended behavior. Yet all the code defined to work using objects of the existing types in the family will continue to work when it actually uses objects belonging to subtypes. This kind of extensibility is similar to that provided by other abstraction mechanisms. For example, it's like the ability to replace the implementation of a data abstraction without affecting the correctness of using code. But it allows the invention of new abstractions, rather than just new implementations. It is most similar to the kind of extensibility provided by parameterization; the

differences between the two types of extensibility was discussed in Section 10.8.

Thus, type families can be a valuable tool in designing and structuring programs. However they must be used properly. This requires understanding when to use hierarchy, and what it means for one type to be a subtype of another. Such understanding allows type families to be used both correctly and appropriately.

Chapter 11

A Preamble to Program Design

11.1 The Software Life Cycle

So far we have concentrated on the specification, implementation, and validation of program modules. The next three chapters deal with issues related to programs as a whole and with the process of program development.

Program development is usually broken up into a number of phases: requirements analysis; design; implementation and testing; acceptance testing; production; and modification and maintenance. Typically, the process begins with someone we shall call the *customer* who wants a program to provide a particular service. Sometimes the service is well understood and described a priori in a complete and precise manner, but this is quite rare. More often customers do not fully understand what they want the program to do. Even if the desired service is well understood, it is probably not described precisely enough to serve as a basis for constructing a program. The purpose of the requirements analysis phase is to analyze the needs of the customer and produce a *requirements specification* of a program that will meet those needs.

The specification that results from requirements analysis is the input to the design phase. In this phase a modular decomposition of a program satisfying the specification is developed. In the next phase the individual modules are implemented and then tested to ensure that they perform as intended. As discussed in chapter 8, we use two kinds of tests: unit tests, in which individual modules are tested in isolation, and integration tests, in which modules are tested in combination.

At best, integration testing shows that the modules together satisfy the implementer's interpretation of the specification. The implementer may have misinterpreted the specification or neglected to test some portion of its behavior, though, and the customer therefore needs some other basis for deciding whether or not the program does what it is supposed to do. This typically takes the form of *acceptance tests*. Acceptance tests provide an evaluation of the program behavior that is independent of the design, and they are generally performed by an organization different from the one that worked on the design and implementation. They should include both trial runs under conditions approximating those the customer will actually encounter and tests derived directly from the requirements specification.

When the program has passed the acceptance tests, it enters the production phase and becomes a product that the customer can use. The useful life of the program occurs during this phase, but the program is unlikely to remain unchanged even here. First, it almost certainly harbors undetected errors that must be corrected during production. Correcting such errors is called *program maintenance*. Second, the customer's requirements are likely to change. Responding to such changes requires *program modification*.

Errors are a problem throughout the entire development process. Those made in the earlier stages, however, are likely to be harder to fix and to result in more wasted effort. An error made in the requirements analysis can lead to a totally useless program, since it is not the program the customer wants. If this is not discovered until the acceptance tests, an enormous amount of design and implementation may have to be redone. An error made during design can result in the implementation of unusable modules and failure to create needed modules. By contrast, an error made in implementing a single module affects only that module and can be corrected simply by reimplementing it.

The earlier an error is detected, the less serious its consequences. If an error in the requirements analysis is caught during that phase, it may be necessary to rethink parts of the requirements specification, but not to discard design work. Similarly, if an error in design is caught before implementation begins, it may be necessary to rethink a number of design decisions, but not to discard work done in implementing and testing unneeded modules. It is clearly important to make use of error-detection methods and techniques during requirements analysis and design, and not just during implementation.

It is worth noting once again that there are no guarantees of correctness. Program verification can ensure that a program meets its specification, but it is not generally practical today, and even if it were, it would still find errors much too late in the program development process. As was just noted, the errors made in the early phases matter most, and it is important to find these errors quickly to minimize wasted effort. One can never be absolutely certain that a requirements specification describes the exact product the customer wants. Nor are there existing techniques that can tell us whether the modular decomposition produced by the design phase will satisfy the specification when the modules are implemented. There are methods that can be used to catch errors in these early phases; they depend on explicit documentation of decisions and careful review of these decisions. While better than nothing, however, they are far from adequate. Identification and invention of better methods is an important area for research in programming methodology.

Although we have emphasized the importance of doing the first three phases in the proper order, it is important to recognize that inevitably a certain amount of iteration through these phases will be necessary. During design we often uncover problems with the requirements specification. When this happens, it is necessary to redo part of the requirements analysis. Similarly, if a problem with the design is found during implementation, the relevant parts of the system must be redesigned.

This chapter focuses on the requirements analysis. We shall give an overview of the issues that must be addressed during this phase and provide a short example. The topics covered are complicated ones. Our discussion of them is abbreviated, oversimplified, and intended to serve only as an introduction.

11.2 Requirements Analysis Overview

The original product description produced by a customer is unlikely to be either complete or precise. The purpose of requirements analysis is to analyze the customer's needs so that we can identify and carefully describe the customer's requirements. This analysis must involve consultation, since the customer is the ultimate judge of what is wanted.

Often, a good way to get started on requirements analysis is to examine how the customer does things at present. If a system has been in place for a while, it will have developed methods for normal processing and for coping with errors and a variety of contingencies. The present system can be a source of ideas not only about how to do things, but about what needs to be done. Studying the customer's organization can also help ensure that the program developed will fit smoothly into that organization. It must be compatible with other systems already in use and also with the current practices of the organization.

The requirements analysis must consider both normal situations and errors. Studying normal case behavior means defining the effect of all nonerroneous user interactions with a program under the assumption that the program is in a normal state. Normal case behavior is often the easiest part of the problem, so that focusing on it is a good way to get started. A good approach is to work out scenarios of typical interactions with the system. We shall give an example of this in the next section.

Cases with no errors represent only a small part of program behavior, and it is essential to consider and describe how a program behaves in the presence of errors. This part of the analysis should never be neglected or underemphasized. The analyst must try to uncover all possible errors that might occur and develop the appropriate responses for each case. Errors come from two sources: users interacting with the program, and hardware and software malfunctions. Scenarios are a good way to study user errors, since they allow us to pinpoint the ways in which these errors might be made. Sometimes proper interface design can prevent user errors—for example, by presenting an interactive user with a menu containing only legal commands. Another possibility is for the system to recognize erroneous input and reject it. Of course, it is not possible to avoid all errors. For example, if a bank clerk makes an error entering an account holder's deposit, it might not be noticed until the account holder examines a monthly statement. The analyst must identify all situations like this and decide what should be done to handle them. Often these decisions require consultation with the customer, since business policy is involved (for example, deciding how to compensate an account holder whose check has bounced because of a bank error).

As far as software errors are concerned, the analyst must make a decision about the amount of effort to be expended during program design and implementation to avoid, remove, or limit the scope of such errors. For example, if it is important to limit the scope of software errors, the output of critical modules can be checked for reasonableness, and the system shut down if the checks fail. The analyst must also decide what to do about hardware failures. It may be important for a system to be *highly available*; that is, it should be highly likely that the system is up and running all the time (or at certain times). Satisfying such a requirement may involve the use of redundant hardware and software. A related requirement is that the system be *highly reliable*. Here we are concerned with avoiding loss of information because of failures. Determining how ambitious the system must be in trying to recover from hardware and software malfunctions is an important aspect of requirements analysis.

In addition to functional requirements, a program must satisfy efficiency requirements. Time and space efficiency should be considered together since it is often necessary to trade one off against the other. The first thing to do is to find out whether or not there is a hard limit in either dimension. For example, the program might have to run on a microcomputer with 128K of memory and one floppy disk. Alternatively, it might have to satisfy some real-time constraint (for example, computing the current altitude of a plane every tenth of a second). In considering time efficiency, it is important to distinguish between *throughput* and *response time*. Throughput refers to the amount of data processed by a system over an interval of time. Response time refers to the amount of time between interactions with the system. Often, optimizing one of these characteristics has a deleterious effect on the other. A communications network, for example, might maximize throughput at the cost of response time by batching messages.

The customer's space and time requirements must be checked for compatibility with the functional requirements, the hardware the customer intends to use, and the price the customer is willing to pay for the system. For example, the customer may want some activity to satisfy efficiency requirements that are either not possible given the hardware or can be satisfied only with very sophisticated software. If there is an incompatibility, negotiations may be necessary to produce new requirements.

These issues affect the content of the requirements specification directly. There are a number of other issues that should be considered during requirements analysis, not because they affect the specification, but because they provide useful input for the designers. Two such issues are *modifiability* and *reusability*. Usually there will be areas with fixed requirements and others in which changes are likely. Information about likely changes is useful because a design can be shaped in such a way as to make certain changes easy. This information is often best obtained by careful study of earlier systems designed to do a similar job. Changes are likely in places where those systems differ from the current one. The specified system might also be intended to be the first of many systems that will be similar but will differ in details. The job of building the additional systems can be simplified if all or part of the software produced for the first system is designed to be reusable. If the designers know what the similarities are, they can shape the design to accommodate them. An outstanding example of this is the isolation of target-machine dependencies in a compiler.

Pinning down constraints on the delivery schedule is another important part of the requirements analysis. Knowing that the customer is in a hurry, for example, may encourage the designer of the software to trade extendability for simplicity. Knowing that a program providing a proper subset of the functionality of the entire system should be available early would have an impact on the implementation schedule as well as on the design.

Ideally, the requirements specification should be precise and complete. A precise specification is one that defines behavior unambiguously, while a complete specification is one that describes all relevant aspects of that behavior. Since requirements specifications tend to be written in an informal language, ambiguities and incompleteness are unavoidable and hard to notice. The analyst should look hard for ambiguities and incompleteness in all parts of the specification.

The requirements specification can be the input to two activities in addition to the design. It can be used to produce acceptance tests and as a basis for a system user's manual. The user's manual is something that must be produced anyway, but its production can provide an independent check on the suitability of the specification. If the system is hard to use, this may be evident when the manual is written. Also, by reading the manual, the customer may notice deficiencies in the specification that were overlooked earlier.

In addition to the specification, the analyst should also produce a document explaining the decisions made during analysis and, if reasonable, the alternatives that were rejected (and why they were rejected). This information is useful when requirements must be rethought because of errors or changing customer needs.

11.3 A Sample Problem

To illustrate this discussion, we now investigate an example, a program for generating KWIC indexes for sets of titles, with the goal of developing a partial requirements specification for this program. KWIC indexes are used to scan a database of titles to find titles of interest. In general, the database is restricted to titles in a particular domain, such as computer science research. The purpose of the index is to allow someone scanning the titles to identify those of interest quickly. Some words in a title are more useful than others for this purpose. Words like "the" and "and" are not useful at all, whereas words like "compiler" and "C++" are clearly useful. The words that are useful are called *keywords*. In the output the titles are presented to the user sorted by keywords. The user can then scan the sorted list, looking for keywords related to the topic of interest. Once a set of titles containing a relevant keyword has been found, the user can choose specific titles to consider in more detail.

There are two ways to present the index: Keywords can be presented "in context" or "out of context." In a KWIC (Key Word In Context) index the keywords stay in their position in the title, but are highlighted. For example, a KWIC index for the titles

A Simple Database Language for Personal Computers
A Practical Tool Kit for Making Portable Compilers
Triply Redundant Databases
Compiling Made Easy

would begin with

A Practical Tool Kit for Making Portable <u>Compilers</u>
<u>Compiling</u> Made Easy
A Simple Database Language for Personal <u>Computers</u>

A Simple Database Language for Personal Computers
Triply Redundant Databases

We want a program to construct a KWIC index for a database of titles supplied by the user. It should go through the database to find all keywords, using information supplied by the user, and should print the index. This description is typical of the kind of problem description that exists prior to the requirements analysis phase. Enough information has been given to provide a general idea of what is wanted. However, as soon as we try to define what is wanted in any detail, we can see that the problem description is vague and imprecise.

We begin by considering the normal case behavior. A good way to start is to enumerate all of the information that must be communicated between the user and the program. The program must find out from the user:

1. where to find the database of titles,
2. which words are keywords, and
3. where to put the KWIC index it has produced.

The user must find out from the program

1. what information the program wants him to provide and
2. whether or not it has succeeded in building a KWIC index.

As mentioned, a good way to determine how this information should be communicated is to sketch a sample session. Before doing this, however, we should decide what style of interaction to use. In general, there are two methods of interacting with a program: The user either waits for the program to demand information or offers it spontaneously—for example, as arguments when the program is called. Let us make the provisional decision that our KWIC program will request responses from the user.

We now start to sketch a sample session. It seems reasonable to assume that the user must start by invoking the program. How this is done almost certainly depends upon the computing environment in which KWIC is to run. Let us assume that KWIC is invoked by typing

```
>run kwic
```

(Here > is the prompt from the operating system. We shall follow the convention of printing input from the user in roman and messages sent by the program in italic.)

Note that we have already made a decision to invoke KWIC without passing it any arguments. There are two reasons behind this choice. First, it preserves the narrowest possible interface between the operating system and KWIC; for example, we need not concern ourselves with the syntax used for passing arguments to programs or even with whether or not it is possible to pass arguments to programs. This will make it easier to move our program from one system to another and will make it more robust to changes in the environment. The second reason is that we have decided to let user interaction be demand driven.

Next the program should ask for some of the information it needs. It could ask about how to find the database of titles or keywords. Since there is no obvious a priori reason to start with either one, we should ask whether getting one kind of information first might make it easier to get the other. Knowing something about keywords is unlikely to help us find out where the database is, but having access to the database may help with keywords. Let us assume, then, that the program asks the user where the database can be found, and the user supplies this information. We shall also assume that the database is in a file that can be accessed by its name:

>`run kwic`
Enter name of database to be indexed: `cs.titles`

Here the user enters the file name followed by a carriage return to indicate that the name is complete.

The program now needs information that will allow it to decide which words are keywords. One possibility is to ask the user to list the keywords and assume that all words not listed are not keywords. Alternatively, the program could ask the user to list the nonkeywords. Of these alternatives, it would probably be better to supply a list of nonkeywords. In either case, errors of omission seem inevitable. When errors cannot be avoided, we should choose the alternative in which the consequences of an error are less severe. Forgetting to list one nonkeyword only results in some titles being listed in a uninteresting way. On the other hand, if one interesting keyword is forgotten, then the user of the listing may miss a title of interest. Assuming that KWIC asks for nonkeywords, our sample session might continue:

Please enter the list of nonkeywords.
The words should be separated by a comma or carriage return, and the list terminated by esc.
Nonkeywords: `a, as, at,` ...

As we construct this sample session, it becomes clear that compiling this list is going to be a long and not terribly easy process, and errors of omission are going to be quite a problem. We should clearly consider other possibilities. For example, if the user were to supply both a list of keywords and a list of nonkeywords, the program could look up every word occurring in a title in these lists. If it found a word that was not on either list, it could then ask the user which list to put it on. In this way we could avoid errors of omission.

The question now arises as to why the user should enter lists at all. Perhaps it would be better for the program to go through the titles and every time it encounters a new word to ask the user which list to put it on. This approach has two important assets: It eliminates a class of user errors and it reduces the amount of typing a user must do. If we adopt this approach, our sample session might continue

> *For the following sequence of words, type k if the word is a keyword and n if it is not.*
> *A*: `n`
> *Simple*: `n`
> *Database*: `k`
> *Language*: `k`
> *For*: `n`
> *Personal*: `k`
> *Computers*: `k`
> *Practical*: `k`
> .
> .
> .

In looking at this sample session we discover that if our database has a lot of different words, the enumeration will be tedious. It would be particularly annoying if the user were producing more than one index and had to repeat the process for each set of titles. A little thought, however, should convince us that this last problem is easily surmountable. KWIC could store the lists of keywords and nonkeywords in such a way that they can be reused. It could ask at the start of a run whether it should use a predefined dictionary, and if so in what file it can be found. When it finds a word that is not in the dictionary, it could ask whether or not it is a keyword and then augment the dictionary with that information. At the end of a run, KWIC could ask the user whether or not to save the dictionary, and if so in what file. A sample session might then look like

> \>`run kwic`
> *Enter name of database to be indexed:* `cs.titles`
> *Enter name of initial dictionary:* `cs.dict`
> *For the following sequence of words, enter k if the word is a keyword and n if it is not.*
> .
> .
> .

If you want to save the amended dictionary, enter y else enter n: `y`
Enter name of file in which to store new dictionary: `mycs.dict`

We now have several ways to obtain information about keywords, so this is probably a good point at which to consult with the customer. We can prepare a summary of the advantages and disadvantages of each approach and present that and the sample sessions to the customer. Let us assume that the customer prefers the last alternative discussed.

Another point to discuss is how sophisticated the program should be at recognizing variants of words. If the user has already indicated that "assembler" is a keyword, should the program ask about "assemblers," or should it assume that "assemblers" is also a keyword? If so, how about "assembled," "assembly," "assembly-line," and so forth? Similarly, if the user tells the program that "is" is not a keyword, should the program assume the same thing about "are," "was," "were," and so forth? This is a difficult and open-ended problem. There is a clear trade-off between ease of use and the difficulty and expense of building the program. The customer should be apprised of this trade-off and asked to make a decision. We shall assume that the unsophisticated approach to recognizing words is chosen.

This gets us to the final piece of information that must be supplied by the user: where to put the output of KWIC. Let us assume that the output is to be written to a file and that the user will be asked for the name of that file:

Enter name of output file: `cs.titles.out`

The user can print this file or read it online using whatever utilities are provided by the operating system.

We have now completed a pass outlining how information is to be communicated from the user to KWIC. To complete our normal case requirements analysis, we still need to clarify the contents of the database of titles (assuming that it has no errors in it) and the form of the output (assuming that no errors occurred during processing). We shall not define these formats here. The key question to ask about the database is how much control we have over its contents. Is it an existing database with contents and organization beyond our control or is it to be built up in a way we may specify? As far as the output is concerned, its format is an important issue, but one that is difficult to deal with before the event. It is hard to anticipate what formats users will like or dislike. Often the best approach is to do something quite simple initially, but to design the program in such a way that it is easy to respond to suggestions for improvements.

We now consider errors. One possible error is for the user to supply an invalid file name, perhaps one that KWIC is unable to open in the appropriate mode. In response we can simply ask the user to supply another name.

Another error might be imperfect data in the database, such as titles with misspelled words. Suppose KWIC is not allowed to alter this database, so that it is not possible for the user to correct the error immediately. What should be done about further processing of the input and about production of the output in this case? The user should have the option of continuing to process the input, so that other errors can be found. To decide about the output, we need to know how much computation would be lost if we discarded it, and also how useful it would be given the errors. KWIC indexes are usually run over very large collections of titles, and the reading in of those titles is a substantial part of what the program does. Furthermore, an error in one word is unlikely to affect the processing of surrounding words or titles. The information in the index about the title containing the error may be misleading, but information about other titles should be correct. Therefore let us make the following decision. If the user decides to continue, we shall skip the word containing the error insofar as producing keyword listings in the index is concerned, but continue normal processing of subsequent words. When input is finished, we shall give the user the option of canceling the output through a further interaction:

If you want to produce the index, enter y else enter n:

Finally, we arrange for titles containing skipped words to appear at the end of the output (if any). The advantage of appending these titles to the regular output is that someone using the index can look at them too.

How does the user indicate that processing should stop? Probably a special user response should be permitted at every interaction with KWIC. Termination should happen gracefully; for example, the user should be given the option of producing the new dictionary and the output.

What about error checking of the dictionary of keywords and nonkeywords? Since the dictionary is produced by KWIC, in theory there should be no errors in it, but, of course, this may not be the case. The user might supply a file that was not produced by KWIC, a file that was originally produced by KWIC but has since been altered accidentally, or a file produced by a version of KWIC that had a bug in it. How easily such problems can be detected will depend upon the format of the dictionary. This format should not be part of the requirements specification, but should be left to the designers. However, we shall require that the dictionary be an ASCII file so that users can inspect it easily. We should probably also require that an auxiliary program be available that allows users to inspect a dictionary and repair errors. (An existing text editor may be sufficient for this purpose.) The effects of an error in the dictionary are likely to be serious. They will lead to either spurious or missing entries in the index. Once an error in the dictionary has been discovered, it should be reported to the user, who should be given the option of providing another dictionary.

Finally, suppose KWIC gets into an unanticipated state because of a hardware or software malfunction. In this case, it should try to write out the amended dictionary to a temporary file (this will not always be possible) and then stop processing. If we have reason to believe that hardware or software malfunctions are going to be moderately frequent, KWIC should probably be required to save the dictionary periodically in a temporary file.

To complete the requirements specification for KWIC, we must define the details that have been omitted and specify efficiency constraints. Potential modifications should include easy reformatting of the output.

11.4 Summary

We began this chapter with a description of the software life cycle. While conceding that it is indeed a cycle, we emphasized the importance of doing a careful requirements analysis before starting design and a careful design before starting implementation. The bulk of the chapter was devoted to a discussion of the requirements analysis. The next chapter takes up design.

We usually start the requirements analysis with an incomplete understanding of what the customer really needs. The goal of analysis is to deepen our understanding and describe the needed system in a moderately complete and precise requirements specification. Customers should be consulted during analysis because they are the ultimate judges of what is needed.

A number of issues must be considered during requirements analysis:

1. The program's behavior must be defined for both correct and incorrect inputs.

2. Issues related to hardware and software errors, such as availability and reliability constraints, must be explored.

3. Constraints on time and space efficiency must be pinned down. Efficiency is not an add-on feature. It must be designed in from the start.

4. Scheduling constraints must be addressed. The customer's desired delivery schedule for the software or part of the software may well have an impact on the design and implementation.

5. It is useful to try to pin down those parts of the requirements that are most likely to change.

6. It is useful to know whether the system being specified is the first of a number of similar systems, so that it can be designed in a way that will allow components to be reused.

The requirements analysis process is a difficult one. We suggested an approach based on sample sessions or scenarios and illustrated that approach with an example. Our discussion was far from thorough. Involving customers in a productive way is perhaps the most difficult and critical aspect of requirements analysis. It involves many issues that are beyond the scope of this book.

Further Reading

Boehm, Barry W., 1981. *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall.

De Marco, T., 1979. *Structured Analysis and System Specification.* Englewood Cliffs, N.J.: Prentice-Hall.

Heninger, Kathryn L., 1980. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering* SE-6(1): 2–13.

Exercises

11.1 Prepare a complete and precise requirements specification for KWIC. Be sure to address issues related to the format of its input and output and the handling of errors. State reasonable efficiency requirements. You may assume that there is enough space to fit all of KWIC's data in main memory, assuming that space is not wasted.

11.2 Program *xref* produces an index for a document: For each word containing more than one letter, it lists the word followed by the lines in which it appeared, for example,

compiler 3, 17, 25, ...

Prepare a complete and precise requirements specification for *xref.*

Chapter 12

Design

In preceding chapters we have discussed the specification and implementation of individual abstractions. We have emphasized abstractions because they are the building blocks out of which programs are constructed. We now discuss how to invent abstractions and how to put them together to build good programs.

12.1 An Overview of the Design Process

The purpose of design is to define a program structure consisting of a number of modules that can be implemented independently and that, when implemented, will together satisfy the requirements specification. This structure must provide the required behavior and also meet the performance constraints, including space and time efficiency, availability, and reliability. Furthermore, the structure should be a good one: it should be reasonably simple and avoid duplication of effort (for example, it should not contain two modules where one suffices). Finally, the structure should support both initial program development and maintenance and modification. The ease of modification will depend on the sort of change desired. A particular decomposition cannot accommodate all changes with equal ease. This is why likely modifications should be identified prior to design—so that a decomposition can be developed that facilitates them.

We start with the requirements specification. Sometimes it describes a single abstraction, sometimes several that together make up the system. We pick one of these abstractions to start work on. This becomes the initial *target abstraction*. In designing a program, we carry out three steps for each target:

1. Identify *helping abstractions*, or *helpers* for short, that will be useful in implementing the target and that facilitate decomposition of the problem.

2. Specify the behavior of each helper.

3. Outline a program to implement the target in terms of the helpers.

The first step consists of inventing a number of helping abstractions that are useful in the problem domain of the target. The helpers can be thought of as constituting an abstract machine that provides objects and operations tailored to implementing the target. The idea is that if the machine were available, implementing the target to run on it would be straightforward. Next, we define the helpers precisely by providing a specification for each one. Usually when an abstraction is first identified, its meaning is a bit hazy. The second step involves pinning down the details and then documenting the decisions in a specification that is as complete and unambiguous as is feasible. Once the behavior of each helper is defined precisely, we can use them to write programs. In principle, the target can now be implemented, but we generally do not do this at design time. Instead we merely outline enough of an implementation to convince ourselves that an acceptably efficient and modular implementation can be constructed. If the abstract machine existed, we would now be finished, but in reality the helpers need to be considered in more detail. The next step is to select one helper and design its implementation. This process continues until all the helpers have been studied.

For large programs, we usually do not know in advance what the structure of the program ought to be. Instead, the discovery of this structure is a major goal of the design. As the design progresses, sometimes a choice must be made between a number of structures, none of which is well understood. Later, the decision may be found to be wrong. This is especially likely for choices made early in design, when the structure of the program is least understood and least constrained by other decisions, and when the effect of an error on the global structure is largest.

When errors are discovered, we must correct them by changing the design. Usually we must discard all later work that depends on the error. That is why we delay the start of implementation until after we have a complete design. Of course, no matter how careful we are about our design, during implementation we are likely to uncover problems with it. When this happens, we must rethink the part of the design related to the problem.

Our discussion so far has outlined how design occurs, but has neglected a number of questions, such as:

1. How is decomposition accomplished? That is, how do we identify subsidiary abstractions that will help to decompose the problem?

2. How do we select the next target?

3. How do we know whether we are making progress? For example, are the helpers easier to implement than the target that caused them to be introduced?

4. How are performance and modification requirements factored into a design?

5. How much decomposition should be done?

These and other, similar questions will be addressed as we carry out an example. First, however, we discuss how to document a design.

12.2 The Design Notebook

The decisions made during design must be recorded. This documentation should be done in a systematic manner and kept in a *design notebook*. The notebook contains an introductory section describing the overall design of the program and a section for each abstraction.

12.2.1 The Introductory Section

The introductory section lists all the abstractions identified so far and indicates where information about them can be found in the notebook. It also indicates which helpers are to be used in the implementations of which targets. This documentation takes the form of a *module dependency diagram*, which identifies all the abstractions encountered during design (those present in the requirements specification or introduced as helpers) and shows their relationships.

A module dependency diagram consists of nodes and arcs. The nodes represent abstractions, and each contains the name of its abstraction. There are three kinds of nodes, one for each of the three kinds of abstractions we use. Nodes like the ones labeled P and Q in figure 12.1 represent procedures, ones like E and D represent data types, and ones like I represent iterators.

The arcs show which abstractions are to be used in implementing which other abstractions. There is an arc from a target to each helper. We say that the target *uses* or *depends on* the helpers. A target uses a procedure or iterator by calling it. A data type is used if one or more of its operations is called. The diagram does not show the specific operations used; the use of any operation is sufficient to set up the dependency. To record more detailed information would result in an overly cluttered diagram that would be hard to use. If this extra information is needed, it is recorded in notes accompanying the diagram.

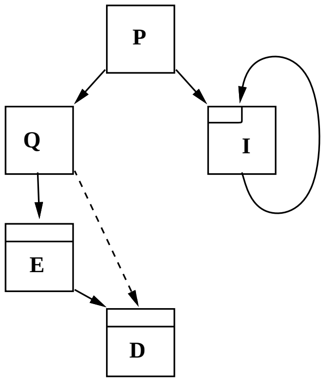

Figure 12.1 A module dependency diagram.

Sometimes, an abstraction uses an object of a type without calling any of the type's operations. For example, it might receive an object as an argument and simply pass the object on to another abstraction. In such a case we say that the target *weakly uses* the type. A special dashed arc is used to indicate this kind of dependency.

The dependencies determine the level of the nodes. An abstraction A is at a higher level than B if A depends on B and B does not depend on A (that is, there is no direct or indirect recursion). If A and B depend on each other, they must appear at the same level.

In figure 12.1 procedure P is to be implemented by the three helpers Q, D, and I. Q is another procedure; its implementation uses E and weakly uses D. E and D are data types; the implementation of E uses D, but D uses no helpers. I is an iterator; it is implemented in terms of itself; that is, it is recursive. Note that the entire structure is a directed graph. It is not a tree, since one abstraction can be used in implementing several others. Note also that the graph may have cycles in it. Cycles occur when there is recursion. If an abstraction is to be used directly in implementing itself, there is an arc pointing from that node to itself (as is the case with I in figure 12.1). Cycles involving more than one arc represent mutual recursion. In general, recursion is reasonable for a program if the problem being solved has a naturally recursive structure. We saw examples of natural recursion in the implementations of `list` and `sorted_list` in earlier chapters. However, if the cycles get very long and involve many arcs, it is wise to be suspicious of the program structure.

The module dependency diagram is useful when errors are detected. A design error shows up as a flaw in an abstraction; for example, an efficient implementation becomes impossible or needed arguments are missing. The potential impact of the error can be determined from the diagram. Implementations of all abstractions that use the erroneous one must be reconsidered. (That reconsideration may find some of those abstractions erroneous, and so on.) An abstraction that weakly uses an erroneous type is affected if the type disappears entirely, but not if the type changes or is replaced by another type. For example, in figure 12.1, if we found a problem with D, we would have to rethink the implementations of E and P, but Q and I would not be affected. Of course, if rethinking E prompted us to change its specification, we would be forced to reexamine the implementation of Q.

The introductory section of the notebook should also contain information about the overall process of implementing the design. It should suggest how many people should work on the implementation, the order in which various abstractions should be implemented, and how long various phases of the implementation should take. It should also propose a set of tests to be used on the completed implementation. Strategies to be used in the implementation process will be discussed in the next chapter.

12.2.2 The Abstraction Sections

We partition the notebook entry for an individual abstraction into five pieces:

1. a specification of its functional behavior;

2. a description of efficiency constraints it must observe;

3. information about how it is to be implemented;

4. information about how it is to be tested; and

5. miscellaneous information that does not fit into any of the previous four categories.

The specification is, of course, the most important item in this list. However, we have already said a great deal about such specifications and will say no more about them here.

Efficiency constraints are generally propagated top-down through a program. The requirements specification may constrain the time the program can take to perform certain tasks and the amount of primary and secondary memory it can use. To convince ourselves that the implementation of a target will meet its efficiency constraints, we have to make assumptions about the efficiency of its helpers. These assumptions show up as efficiency constraints in the parts of the design notebook detailing each helper.

Efficiency constraints can be expressed in a variety of ways. Frequently we express them as functions of the size of the input. We might, for example, write as part of the description of a `sort` procedure with formal argument `x` of type `list<int>*`

efficiency:
worst case time = Order(length(x) $*$ log(length(x)))
main memory added = Order(length(x))
max temporary main memory = Order(length(x))

This says

1. The time required to sort list `x` should never exceed some constant multiplied by the product of the length of `x` and the log (base two) of the length of `x`.

2. When `sort` returns, the total amount of main memory currently allocated for the program in which `sort` is embedded should not have been increased by more than a constant multiplied by the length of `x`.

3. During the execution of `sort`, the amount of main memory allocated for `sort` should not exceed some constant multiplied by the length of `x`.

For many applications, it suffices to supply relative efficiency constraints such as those just given. Sometimes, however, it is useful to bound the multiplicative constants or even to impose an absolute bound. An absolute upper bound on time is needed in real-time applications. Analogously, we put absolute bounds on the space used in programs that are to be run on small machines or machines that do not support virtual memory. If `sort` were to be embedded in a real-time system that was intended to run on a machine with no secondary memory, we might write

efficiency:
worst case time = 1 second
main memory added = Order(length(x))
max temporary main memory = Order(length(x))
secondary store added = 0
max temp. secondary store = 0

(For `sort`, an absolute constraint on time would be satisfiable only if the requires clause of its specification put some upper bound on the length of the input list `x`.)

The section of the notebook entry containing information about how the abstraction is to be implemented should include a list of its helpers. For data types, the entry should describe the representation, representation invariant, and abstraction function associated with the intended implementation. This section might also include a description of how the implementation works. This description can be omitted if the implementation is straightforward, but it is necessary if the implementation is clever. Frequently it is helpful to give a sketch of the intended algorithm, such as

```
// Implement sort(x) by a recursive algorithm of the form:
  if length(x) ≥ 1 then x
    else merge(merge_sort(first_half(x)), merge_sort(second_half(x)))
```

Alternatively we might give an appropriate reference, such as

> Implement sort using one of the merge sorts in section 5.2.4 of *The Art of Computer Programming, Volume 3*, by Donald Knuth, Addison-Wesley, 1973

An important criterion for evaluating a design is how easy or hard it will be to test its implementation. The notebook entry for each abstraction should contain information on how the designer believes its implementations should be tested. This information might include suggested test data, a description of a driver that could be used to supply the test data and check the results, and suggestions about stubs that could be used to replace the helpers.

As its name implies, the section of the abstraction entry labeled "miscellaneous" might contain almost anything. Typical items are

1. a justification for the decisions documented elsewhere in the entry,
2. a discussion of alternatives that were considered and rejected,
3. potential extensions or other modifications of the abstraction, and
4. information about the context in which the designers expect the abstraction to be used.

In closing this section we should note that if the design is very large, it may be useful to structure the notebook by introducing subsidiary notebooks. In a module dependency diagram, any subgraph can be viewed as an independent subsystem. However, the most convenient choice is a subgraph in which only one node is used from outside.

12.3 Problem Description

In the remainder of this chapter, we carry out a complete design of a simplified text formatter. This program is small and is intended primarily to serve as an illustration of the design process. This section gives a specification of the formatter; subsequent sections describe its design. The input to the text formatter consists of a sequence of unformatted text lines mixed with command lines. Each line (except perhaps the last) is terminated by a newline character, and command lines begin with a period to distinguish them from text lines. An example is

```
Justification only occurs in fill mode.
In nofill mode, each input text line is output without
modification.
The .br command causes a line-break.
.br
Just like this.
```

The program produces justified, indented, and paginated text:

```
Justification only occurs in fill mode.  In nofill mode,
each input text line is output without modification.  The .br
command causes a line-break.
Just like this.
```

The output text is indented 10 spaces from the left margin and is divided into pages of 50 text lines each. A header of 5 lines is output at the beginning of each page. The header consists of 2 blank lines, a line of the form

```
Page n
```

where n is the page number, and 2 more blank lines.

An input text line consists of a sequence of words and word-break characters. The word-break characters are space, tab, and newline; all other characters are constituents of words. Tab stops are considered to be every eight spaces. The tabs and spaces are accumulated in the current output line along with the words. For example, if two spaces occur in the input between two words and those words appear on the same output line, they should be separated by at least two spaces. If two words appear on different output lines, the spaces between them need not be preserved.

The formatter has two basic modes of operation. In "nofill" mode, each input text line is output without modification. The formatter is initially in "fill" mode, in which it produces output lines of 60 characters and accepts input until no more words can fit on the current output line (newline characters are treated as spaces). The line is then justified. Justification is performed by adding spaces between words until the last word has its last character in the rightmost position of the line. Spaces should be added as evenly as possible, in a way that also avoids "rivers" of white space in the text. (This could occur, for example, if all extra spaces were added to the same side of each line.) Spaces can be added only between words to the right of all tabs. If there is no place to add spaces, then no justification is performed and no error message is produced. Thus if a word is encountered that is longer than the line length, it is output as given on a line by itself with no error message.

Any input line that starts with a word-break character causes a *line break*: If the current output line is not empty, it is neither filled nor adjusted,

```
void format(istream & ins, ostream & outs, ostream & errs) throw (CE_Badarg);
// modifies: ins, outs, errs
// effects If ins is not readable, or outs or errs is not
//    writable, throws CE_Badarg(s), where s identifies the bad argument;
//    otherwise proceeds as described in the text, taking input from ins,
//    and producing output on outs and error messages on errs.
```

Figure 12.2 Specification of the formatter.

but output as is. If the current output line is empty, a line break has no effect. An empty input text line (one containing no words, tabs, or spaces) causes a line break and then causes a blank line to be output.

The formatter accepts three commands:

.br	causes a line break
.nf	causes a line break and changes the mode to nofill
.fi	causes a line break and changes the mode to fill

An unrecognized command name causes an error message and is otherwise ignored. The message describes the error and gives the line number on which it occurred.

The specification of the formatter is given in figure 12.2. Since the formatter is intended for general use, we have not made any assumptions about its inputs. (That is, it has no requires clause.) The specification describes the effects by referring to the text of this section; a specification that appears in the design notebook would contain this text. The efficiency constraints are as follows: The time used should be Order(`n`), where `n` is the number of characters in `ins`. The space added should be Order(`n`) (this is the storage for `outs`), but the temporary space used should be small, much less than Order(`n`). Finally, the design must support such potential modifications as additional commands (for example, for figures and different fonts) and different kinds of output devices. And it must be reasonably simple and modular.

12.4 Getting Started on a Design

In the next several sections, we illustrate the design process by developing a C++ program to implement the formatter. Our presentation idealizes the design process; for example, no design errors are made. We discuss this point further in section 12.9.

When design begins, we have a module dependency diagram containing the abstractions identified during requirements analysis. The graph may contain arcs, since some of the identified abstractions may depend on some of the others. (For example, the requirements specification for an accounting system might have identified a command processor—a procedure—and an account database—a data type; the command processor depends on the database since it makes calls on the database operations.) We begin our design by entering this diagram and the specifications of the abstractions in the design notebook. Then we choose our first target. Since in the case of the formatter there is just one abstraction, `format`, this automatically becomes our first target.

The first step is to invent helpers. There is one main heuristic to guide us in this process: Let the problem structure determine the program structure. This means that we should concentrate on understanding the problem being solved (namely, how to implement the target) and use the insights we gain into the problem to help us develop the structure of the program.

A good way to study the problem structure is to make a list of the tasks that must be accomplished. Here is a list for `format`:

1. Read input.
2. Interpret input.
3. Produce output.

We do not assume that the final program will have subparts that correspond to the listed tasks. Listing the tasks is just a first step toward a design. The next step is to use the list as a guide in inventing the abstractions that will determine the program structure.

Although we have listed the tasks in approximately the order in which they might be carried out, we do not assume that this order will exist in the final program. In fact, for the tasks just given we have a major choice to make. We could complete each task before starting the next, but this problem does not require that all input be interpreted before output can be produced. We can thus do the job incrementally, interpreting the input and producing the output as we go. Since an incremental solution is needed to satisfy our space constraints, we shall make this choice.

In looking for abstractions, we seek to hide details of processing that are not of interest at the current level of the design. Although we can use procedures and iterators to hide details, data types (and sometimes families of types) are most useful for this. We shall look for types in likely places: input, output, internal and external databases, and individual data items. For example, we can use a type to hide details of output processing (task 3). First we have to decide what level of encapsulation to provide. We could have a line abstraction that hides the formatting of a single line or a page abstraction that hides the formatting of a single page. Neither of these completely encapsulates the output, though, and complete encapsulation is a good idea, because as production of output becomes fancier (a likely modification), it would be useful if we only had to change one abstraction. Our solution is to introduce a data type, `doc`, that is informed about details of the input and produces the output accordingly. Doc is a kind of abstract output device, with high-level operations well matched to formatting.

Just as doc encapsulates details of output processing, we would like to encapsulate the details of processing input. Here we could invent a token abstraction to represent the meaningful items in the input text, such as words and tabs. However, to produce the proper kind of token, we must know whether we are at the start of a line, since a period is interpreted differently in this position. In fact, the problem as stated is line oriented. If we adopt this problem structure in our program, we can capture knowledge about current position in a line in a natural way. We therefore define a procedure, `do_line`, that processes a single line of input. `format` will be responsible for processing all the lines of the input in order; `do_line` will be responsible for processing individual lines.

Finally, we must consider task 2, interpreting input, and the relationship between `do_line` and `doc`. A simple solution is to let `do_line` interpret the input and call appropriate doc operations. This solution requires that we pass a `doc` object as an argument to `do_line`. `format` is responsible for creating and finishing the output; doc provides constructor to start the output and a `terminate` method to finish it.

Now we are ready for the second step of the design, namely, firming up and documenting these abstractions. In the process of producing the documentation, we often uncover loose ends. Some are simply details that need to be worked out, but some might be design errors that require modifications. That is to say, a lot of design work is done during documentation. For example, in specifying `do_line`, we must decide exactly where the input stream is positioned when `do_line` is called and when it returns. We must also decide which module is responsible for checking that the various argument streams are in the proper mode. Furthermore, we must decide whether `do_line` writes error messages and how `format` finds out that all input has been processed.

The specification of `do_line` is shown in figure 12.3, which also shows the efficiency constraints. This specification contains the answers to our questions. `do_line` is responsible for error messages, but not for checking the mode of `ins` and `errs` or for closing them; `ins` is positioned at the start of a line both when `do_line` is called and when it returns; and `do_line` notifies `format` of the end of input by raising an exception.

The specification of `doc` is given in figure 12.4. Specifications are given only the constructor and the `terminate` method; more operations will be specified later. Note that `doc` relies on `format` to check that `outs` is writable. Also, `doc` requires that other modules in the formatter not mod-

```
void do_line (istream &ins, doc *d, ostream &errs) throw (CE_Alldone);
// requires: can read ins and write errs and d has not been terminated.
// modifies: ins, errs, d
// effects: If ins is empty throw CE_Alldone. Otherwise processes one input
//    line from ins to d as defined in the specification of format, writing
//    error messages on errs. The entire line is processed including the
//    end-of-line character.
// efficiency: The time and space taken to scan and parse a line should be
//    proportional to the number of characters in the line.
```

Figure 12.3 Specification of `do_line`.

ify `outs`; this requirement is needed so that `doc` has control over what is written to the stream.

Our picture of `doc` is rather incomplete at this point. When inventing data abstractions, we define only the operations used by the implementation we are studying at the time. Additional operations will be defined when we study other implementations that use the type.

Note that the specifications of `doc` and `do_line` both refer to that of `format`. It is typical to have such references in the specifications of helpers. References are better than copying text, since they lead to shorter specifications that highlight how an abstraction fits into the overall structure.

The extended module dependency diagram is shown in figure 12.5. The diagram contains nodes for `doc` and `do_line` and arcs that indicate use. At this stage of the design, we know that `format` uses `doc` and `do_line` and also that a `doc` object is passed to `do_line`. If an abstraction takes an object of some type as an argument or result, then it uses that type. Here `do_line` uses `doc`, and there is thus an arc from `do_line` to `doc` and `do_line` appears at a higher level in the diagram than `doc`.

A sketch of the implementation of `format` is shown in figure 12.6.

12.5 Discussion of the Method

In constructing our design, we have used a single method for focusing our attention on what needed to be done—we broke the work into subtasks and then investigated how we might accomplish the subtasks. We did not simply introduce a procedure for each subtask, however. Instead we looked for abstractions, especially data types, to take care of the details of the tasks. We believe that this results in a better design, by making it easier to hide details until later stages of the design and to see and exploit connections among different tasks.

```
class doc {
public:
// requires: the ostream passed in when doc is constructed must be readable and
//    must not be modified outside of doc between the time the doc object is
//    created and the time it is terminated.
// overview: Doc is an output abstraction that produces formatted output
//   (as defined in the specification of format) corresponding to the information
//    passed to it via its operations. A doc object is connected to the
//    ostream passed to its constructor. All output will be written to this
//    stream by the time the terminate operation returns.
// efficiency: time = Order(n), where n is the number of characters written to
//   the connected ostream. Space added = Order(n). Temporary space use
//   much less than Order(n).
// modifies: the ostream passed to the constructor

// constructors:
  doc (ostream &outs) throw ( );
  // effects: Prepares to produce the document on outs, starting on page
  //   1, in "fill" mode.

// methods
  void terminate ( ) throw ( );
  // modifies: this
  // effects: Finishes writing information in this to the connected ostream
};
```

Figure 12.4 Partial specification of doc.

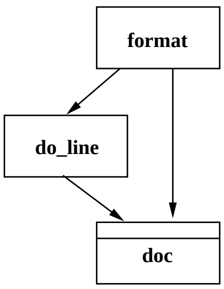

Figure 12.5 Extended module dependency diagram.

```
check modes of arguments
doc *d = new doc(outs);
call do_line (ins, d, errs) repeatedly until it throws CE_Alldone
terminate d
```

Figure 12.6 Sketch of the `format` implementation.

We introduced abstractions to hide details that we deemed inappropriate at the current level. This raises the question of how to decide what is appropriate at a given level. Such a decision is largely a matter of judgment, but there are some guidelines. An implementation should accomplish something, but it should not do too much. Our goal is to end up with small modules; implementations of procedures and iterators, both within and outside of data types, should be no more than a page or two in length, and usually shorter. If in `format` we had decided to take care of the details of input and output, we would have ended up with too large a module. In addition, the different parts of the implementation of a module should be at roughly the same level of detail. Even though it deals with input line by line, while output is one big abstraction, `format` is reasonably well balanced in this regard because these decisions match the structure of the problem.

The design of `format` is typical of the design of a top-level abstraction in a system. The implementations of such abstractions are concerned primarily with organizing the computation, while the details of carrying out the steps are handled by helpers. Introducing partially specified data types like `doc` is also typical. In the early stages of design we frequently know that two modules must communicate with each other, but we do not know exactly how this communication is to take place. In particular, we know that the modules are to communicate through objects of some type, but we do not know what operations on those objects will be useful. Therefore the specification of the shared type is necessarily incomplete. It will be completed as we continue with the design.

12.6 Continuing the Design

We now have a module dependency diagram containing several abstractions. How do we select the next target? The first thing to notice is that not all abstractions are suitable as the next target; those that are suitable we shall call the *candidates*. Clearly, `format` itself is not a candidate since we have already designed its implementation. In addition, `doc` is not a candidate, since it is incomplete. (Actually, we shall occasionally select an incomplete abstraction as a candidate—see section 12.9.) Therefore we have a single candidate, `do_line`.

`do_line` must process either a text line or a command line. The processing is quite different in the two cases, and we shall provide two procedures, `do_text_line` and `do_command`, to handle them. Although the work of processing command and text lines could be done directly in `do_line`, providing these two procedures may simplify future modifications by localizing what would have to be changed if, for example, a new command were added.

```
void do_text_line (istream &ins, doc *d) throw ( );
// requires: can read ins and ins is not empty and d is not terminated
// modifies: ins, d
// effects: Reads one line from ins, including the end−of−line character,
//   and processes it as text, as defined in the specification of do_line.

void do_command (istream &ins, doc *d) throw (CE_Error);
// requires: can read ins and ins not empty and d has not been terminated
// modifies: ins
// effects: Reads one line from ins, including the end−of−line character.
//   Discards the initial character and processes the rest of the line as a
//   command line as defined in the specification of do_line. If the command
//   is erroneous, throws CE_Error(s), where s describes the error;
//   however, the entire line is processed even if the command is erroneous.
```

Figure 12.7 Specifications of *do_text_line* and *do_command*.

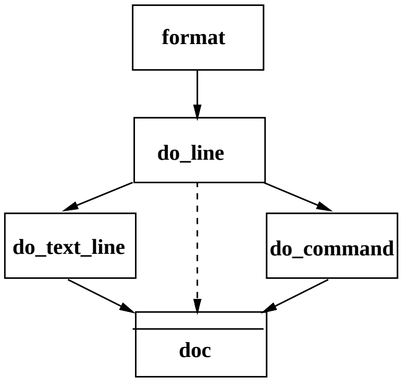

Figure 12.8 Module dependency diagram after design of *do_line*.

Specifications for the two procedures are given in figure 12.7. Note that both `do_text_line` and `do_command` assume that `ins` is positioned at the first character of the line when they are called. This means that `do_line` must only peek at and not read the first character of the line; this can be accomplished using the `istream::peek` method.

The new module dependency diagram is given in figure 12.8. Now there is a dashed arc from `do_line` to `doc` because `do_line` does not call any `doc` operations. Before studying `do_line`, we did not know whether its use of `doc` would be weak or strong, so we assumed it would be strong. We are now making the same assumptions about `do_line` and `do_command`.

Now we have two candidates, `do_line` and `do_command`. How do we choose between them? There are no hard and fast rules here; either candidate could be studied. However, there are several reasons why we might prefer one of several candidates:

1. We are uncertain about how to implement it. For example, we might need to implement an abstraction very efficiently, and we are not sure how to achieve this efficiency or even whether it can be achieved.

2. We are uncertain about its appropriateness.

3. It may be more central to the design than another, so that studying its implementation will provide more insight into the design.

4. We may have been working on some area of the design and would like to finish that area.

The first two rules concern the investigation of areas considered to be questionable; choosing an abstraction for one of these two reasons is likely to expose design errors quickly. If such an uncertainty exists, it is almost always best to investigate it immediately. The other two rules are really opposites: Finishing up an area is unlikely to lead to further insights, but it may nevertheless be useful (not to mention psychologically comforting) to get part of the design fully out of the way.

In our example, there is little difference between the two candidates, so we shall arbitrarily choose `do_text_line`. The main question here is how to convey information about the input to `doc`. One possibility is to pass a token that encodes the information. However, such an approach involves unnecessary work: `do_text_line` must encode information in a token that must then be decoded in `doc`. A better approach is to provide a `doc` method for each case. Such a solution will be more efficient in both space and time.

The following kinds of information must be passed to `doc`: words, spaces, tabs, end of line, line break (if the first character in the line is a space or tab), and empty line (if the entire line is empty). Each of these will be handled by a separate method: `add_word`, `add_space`, `add_tab`, `add_newline`. `break_line`, and `skip_line`. This scheme has the advantage of being easily extensible.

Another question concerns the representation of words: Should words be strings, or do we need a data abstraction? We should use a data abstraction if it will ease potential modifications, but that appears not to be the case here. For example, a word abstraction would not help in adding different fonts, since a font change typically affects more than one word at a time. Therefore we shall just use strings for words. The rest of `do_text_line` involves scanning the input, but this is relatively straightforward and no helpers are needed.

`do_command` must scan the command from the stream and discard the rest of the line. The main decision here is whether it should recognize

```
class doc {
public:
// as in Figure 12.4 plus the following methods
// requires: all methods require this not terminated
// modifies: all methods modify this

  void add_word (string w) throw ( )
  //  effects  Adds w to this.

  void add_space ( ) throw ( );
  // effects: adds a space to this.

  void add_tab ( ) throw ( );
  // effects: adds a tab to this.

  void add_newline ( ) throw ( );
  // effects: adds a newline character to this.

  void skip_line ( ) throw ( );
  // effects: adds an empty line to this.

  void break_line ( ) throw ( );
  // effects: adds a line break to this.

  void set_fill ( ) throw ( );
  // effects: adds a line break to this and enters fill mode.

  void set_nofill ( ) throw ( );
  // effects: adds a line break to this and enters no−fill mode.
};
```

Figure 12.9 Complete specification of doc.

the command or just pass the string to a `doc` method. Either decision is satisfactory for now. However, as more commands are added, it is likely that fancy parsing will be required (since commands may have arguments). It is appropriate for `do_command` to do this parsing, since we should keep `doc` free of such considerations. Therefore `do_command` will have to recognize the commands. It can then call specific `doc` methods. The `doc` methods needed are `break_line`, `set_fill`, and `set_nofill`.

The specification of `doc` is given in figure 12.9. The module dependency diagram is still that shown in figure 12.8.

12.7 The Doc Abstraction

Now `doc` is the only candidate, and we begin by considering its rep. To decide on the rep, we must make a fundamental decision: Do we produce output incrementally or wait until the end? Since it is not necessary to wait, we shall choose the incremental approach. Also, this choice is necessary to support the space constraints of the formatter.

The rep must contain at least the following: the connected `ostream`, the current page number, the current line number, the current mode (fill or nofill), and a buffer. The buffer is needed because output cannot be completely incremental; when we are in fill mode, we must collect a line's worth of characters before doing the output. (Note that the rep will need to point to the `ostream`, which is passed in by reference. Therefore it will point to a location on the stack, which is fundamentally dangerous. It is safe in this case, however, because the lifetime of the `ostream` variable is longer than the call to `format` and therefore its storage will exist as long as `doc` is in use.)

`doc` must perform two different kinds of tasks: processing lines and arranging lines in pages. Most of the complications are involved in line processing, namely, filling and adjusting a line when in fill mode. If we introduce a `line` abstraction, we may be able to deal with the details of line processing separately. By encapsulating these details, we can avoid considering them in the implementation of `doc`.

We could encapsulate the details of line processing at several different levels. One possibility is a high-level abstraction that justifies and outputs the characters of the line automatically when the line becomes full. Such an abstraction would have to know the current mode of operation (fill or nofill) and also the maximum line length. Note that outputting a line cannot be completely invisible; the `doc` abstraction must be informed when line output happens so that it can take care of page formatting. An alternative approach is to have a low-level abstraction that keeps track of words and the spacing between them, but does not do justification. However, such an abstraction provides little benefit over simply keeping the information in an array.

We shall use a third approach, a `line` abstraction at a level between the two just discussed. This type will provide justification, but only on command. Information about the current processing mode and line length will be kept in `doc`, not in `line`. This decision seems appropriate because such information concerns the document as a whole; that is, it is information at the document level. Note that we have achieved a good separation of concerns: `line` takes care of all details of managing lines, while `doc` is concerned with all other details of formatting the document.

The intermediate-level abstraction has potential for future flexibility. For example, we might want to vary the size of the left margin, or to prepare several lines in advance of output. These possibilities can be accommodated provided we decouple the line abstraction from physical lines; in particular, outputting a line should not happen automatically, and should not include producing the left margin.

To determine the `line` operations, we must look at the implementations of `doc` operations. To take care of spaces and tabs, `line::add_space` and `line::add_tab` will be useful. However, most of the interaction between line and `doc` takes place in `doc::add_word`. A rough sketch of what this operation does is

> **if** fill mode and no room for word on line
>   **then** justify and output line
> add word to line

Given this sketch, we can define the `line` operations we need. We need methods for adding a word to the `line` and for finding out whether a word can fit in a `line`. We must also be able to justify and output the `line`. Justification and output should be separate operations, so that later we could, for example, arrange to justify several lines before outputting any of them. The output method should write the `line` directly onto the stream rather than return a string, since this may be more efficient for some `line` representations (for example, if the rep is not a string).

Justification must be done in such a way as to avoid rivers of white space in the text; since rivers concern adjacent lines, this is an issue we must consider in doc. It is not always possible to add spaces evenly, so some pairs of words will have more spaces added between them than will others. We can avoid rivers by adding these extra spaces alternately from the right and left in subsequent lines. We must remember in the rep of `doc` where the extra spaces should go for the current line. We can begin at the right at the start of each page.

One last question concerns the creation of `line` objects. Obviously we need a constructor to get started. After we have output a `line`, we could create another object for the next `line`, but such an approach is wasteful because it creates garbage (old `lines`) that may need to be collected. It is better to reinitialize the old object and reuse it. The `clear` method will allow us to do this. A specification of `line` is given in figure 12.10.

`doc::add_word` is not the only `doc` method that causes `lines` to be output; methods `add_newline`, `break_line`, `skip_line`, and `terminate` also do this. Furthermore, outputting lines involves page formatting. A procedure `output_line` is useful for handling these details. Since `output_line` is unlikely to be useful outside the implementation of `doc`, it can be private to `doc`, and we do not add it to the module dependency diagram. Instead we design its implementation as part of designing the implementation of `doc`.

```
class line {
public:
// overview: A line is a mutable line of text consisting of the characters
//    corresponding to words, tabs, and spaces. The justify method justifies
//    the line to a given length, and output writes the line to a given ostream.
// efficiency: The space added should be Order(n), where n is the number of characters
//   in the line. The time used to justify and output should be Order(n).
//   The time used for other operations should be constant.

// constructors:
  line ( ) throw ( );
  // effects: makes this be the empty line.

// methods:
  void add_word (string w) throw ( );
  // modifies: this
  // effects: adds characters of w to the end of this

  void add_space ( ) throw ( );
  // modifies: this
  // effects: adds a space character to the end of this.

  void add_tab (  ) throw ( );
  // modifies: this
  // effects: adds a tab to the end of this.

  int length ( ) const throw ( );
  // effects: returns the length of this

  void justify (int len, bool from_right) throw ( );
  // modifies: this
  // effects: justifies this as described in the specification of format to contain
  //   len characters if possible. Justification occurs by enlarging spaces occurring
  //    to the right of the rightmost tab. Spaces are added as evenly as possible, with
  //    extra spaces added from the right if from_right is true and otherwise from the left.
  //    If justification is not possible (e.g., because this contains just one word),
  //    this is left unchanged.

  void output (ostream &outs) throw ( );
  // requires: outs is writable
  // modifies: outs
  // effects: writes characters of this to outs.

  void clear ( ) throw ( );
  // modifies: this
  // effects: reinitializes this to be empty.
}; // end line
```

Figure 12.10 Specification of line.

```
class doc {
// as in Figure 12.9

private:
// doc produces output incrementally, making use of a buffer consisting of a
//   single line object.

  line *l; // the current line
  bool fill; // true if in fill mode
  bool from_to; // true if extra spaces should be added from the right
  int lineno; // number of lines on current page
  int pageno; // number of current page
  ostream *o; // the ostream
  lines_per_page; // number of lines per page

  // A typical doc object is a sequence of characters of all the lines completed
  //   so far, followed by the characters in the current line.
  // The abstraction function is:
  //   A(c) = Characters in c.outs || Characters in c.line
  // The rep invariant is:
  //   can write c.outs & 1 < c.lineno ≤ c.lines_per_page & 0 ≤ c.pageno
  //   & c.page_no > 0 => the current page has the proper header & c.line_no
  //     lines, not counting the header
  //   & all previous pages p have the proper header and c.lines_per_page lines
  //     not counting the header and p is followed by a page break.

  void output_line ( ) throw ( );
  // modifies: this
  // effects: writes this and an eol character on outs, takes care of page
  //   headers and page breaks, and increments lineno and pageno appropriately.
}; // end doc
```

Figure 12.11 Implementation documentation for doc.

`output_line` is in charge of all page formatting, so we must make decisions about how to keep track of page numbers and how to recognize the need for a new page. A reasonable choice is to store the number of the current page and also the number of lines on the current page in the rep. When we have produced all lines on this page, we can prepare for the next. However, we should not produce the page break and page header for the next page unless there is something on it.

The implementation of `doc` is described in figure 12.11. Note that we have included not only the rep, rep invariant, and abstraction function, but also the specification of the private `output_line` method. The extended

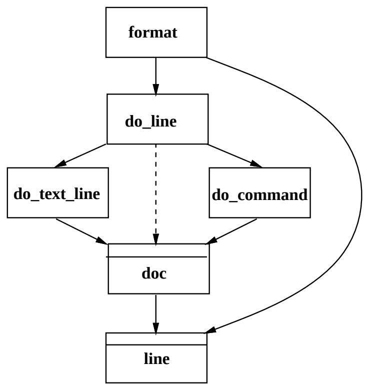

Figure 12.12 Module dependency diagram including line.

module dependency diagram is shown in figure 12.12.

12.8 The Line Abstraction

As usual when designing the implementation of a data type, we begin our study of `line` by considering its rep. The major issue here is to store the words, spaces, and tabs in a way that makes justification relatively straightforward and efficient; in particular, we should avoid moving characters around any more than necessary. Since `justify` is the operation method affected by our choice, we should study its implementation first.

Justification consists of the following tasks:

1. Find leftmost space subject to justification.

2. Enlarge spaces.

A plausible approach is to keep words, tabs, and spaces separate in the rep, so that `justify` can easily locate the leftmost space subject to justification and all the spaces it must enlarge. For example, we could use

```
array<struct{int info; string wd}> * items;
```

If `info = 0` there is a word, which is stored in `wd`; otherwise there is a space (`info < 0`) or tab (`info > 0`), and the absolute value of `info` tells how many blank characters should be output. For efficiency, the current line length should be stored redundantly in the rep. Also, justification will be faster if we keep track of the rightmost tab as tabs are added to the `line`. This optimization is worthwhile if tabs are rare, which is likely. The implementation of `line` is described in figure 12.13.

```
// the rep of a line keeps spaces, tabs, and words separately, although adjoining spaces
//  are merged. Tabs and spaces are kept as a count of the number of spaces to output
//  The line length and the position of the rightmost tab are also kept.

   int length;  // the current line length
   int right_tab; // index of rightmost tab or -1 if none
   array<struct{int info; string wd}> *items; // the words, tabs, and spaces
   int max_tab_width; // the maximum tab width

// a typical line is a sequence of characters.
//  The abstraction function is:
//     A(c) = if c.length = 0, then " " else
//        chars(c.items[1]) || ... || chars(top(c.items))
//  where chars(item) is
//      if the item is a word, then the characters in the word
//      otherwise |n| spaces, where n is the integer in the tab or space item
// The rep invariant is
//      c.length = the sum of the ints in the tab and space items + the
//        sizes of the words
//      & c.right_tab >= -1 & c.right_tab > -1 => c.items[c.right_tab] is a tab
//         & there are no adjacent spaces in c.items
//         & the int in each tab is <= c.max_tab_width
//         & for each tab in c.items, count mod max_tab_width = 1, where
//             count is the number of characters from the beginning
//             of c.items up to and including this tab
```

Figure 12.13 Implementation documentation for line.

12.9 Review and Discussion

The design of `format` was accomplished in stages. At each stage we selected a new target and investigated its implementation. There were two main activities in this investigation. First, we invented some helping abstractions that were useful in the problem domain of the target. Then we defined the properties of the helpers precisely and described those properties in specifications.

The hardest part of program design, and the place where the greatest creativity is required, is in the invention of the helpers. The primary method used here is to study the problem structure and derive from it a structure for the program. Our study of the problem structure takes the form of a list of tasks to be accomplished. However, we do not simply invent a program structure that carries out those tasks one after another. Instead we use the list as a way of focusing attention on what needs to be done, and then invent abstractions to accomplish the needed work.

Abstractions are introduced in each case to hide detail. For example, doc hides the details of the way in which the output is formatted, while do_line hides the processing and interpretation of input. It is desirable to hide detail for two reasons: control of complexity and modifiability. We control complexity in order to delay making decisions and thus limit the number of concerns that must be dealt with at any particular level of the design. We support modifiability because the hidden details can be changed later without affecting other abstractions.

In developing the design we are guided by several factors:

1. Knowledge of what abstractions are already available. This includes abstractions present in the programming language and in any available library of programs as well as those already identified during this design.

2. Knowledge of preexisting algorithms and data structures. It is necessary to know about the methods that have already been developed. For example, the designer should be familiar with existing sorting and searching techniques and not have to reinvent them.

3. Knowledge about related programs and their structure. As we get deeper into the design, this knowledge is likely to become more and more useful, since even radically different applications are likely to have some similar subtasks.

For example, our decision to process input and output incrementally was based on our knowledge that one-pass algorithms are usually more space efficient than two-pass algorithms. Also, we relied on the existence of C++ and 6.170 types and made explicit choices between them and data abstractions—for example, strings versus a word abstraction and streams versus an output abstraction. We did not use knowledge of data structures in the formatter, but we would do so in any program that needed to store large amounts of data. For example, if we had many formatting commands, we might store information about them in a command table that provides fast lookup.

Types usually have the most impact on the shape of the design. Of course, procedures and iterators are also important, but they tend to appear as operations of types rather than independently. (The formatter was unusual in not having any iterators in its design.) We look for types in four areas: input, output, internal data structures, and individual data items. The design of format contains examples of two of these four uses. doc is an abstraction of output, while line is an abstract data structure. The word abstraction would have been an abstract item, but we decided it was not needed. However, a command abstraction might be useful in the future, especially if users are permitted to define their own commands.

Encapsulating input and output permits us to consume input or produce output in terms of abstract quantities. Input is often handled with procedures or iterators, since typically we just read the next item. The buffering provided by `istreams` is then adequate. A data abstraction allows fancier buffering and also permits us to produce or consume the next item whenever it is convenient. We made use of both these features in `doc`.

As the design of the implementation becomes better understood, we can use sketches of the program as a guide to our study of abstractions. These sketches are written in English and contain descriptions of subtasks to be accomplished. We can then note for each subtask what procedure or iterator, or operation of a type is to be used and what its arguments are. Unlike the list of tasks used to start the design, the order in which subtasks are carried out in the sketch is significant. We made such a sketch in studying the `doc::add_word` method.

In inventing helpers we are concerned only with concepts; the details of those concepts are usually left ill-defined. In the second step of design, we firm up those details and document them by means of specifications. During this activity we commonly uncover cases that have been overlooked, especially error cases and end cases (for example, the empty input). We may discover missing arguments or results, or find that arguments or results are of the wrong type. In the case of data types, we may discover missing operations. Moreover, if an abstraction has become overly complex, this may be evident when we try to write the specification. A complicated specification leads naturally to the question of how things can be simplified. A better, simpler abstraction is often the result.

After we have specified all the helpers precisely, we document our design decisions in the notebook by extending the module dependency diagram, adding or extending definitions of the helpers, and sometimes recording a description of the implementation of the target. Such information can be omitted unless the implementation is clever, although for data types will will often provide a sketch of the rep (what it will contain).

One issue we have not yet discussed is how thorough the design of an abstraction implementation must be. At each stage, the investigation must be thorough enough that all helpers are identified and specified precisely. It is not necessary to go further than this. For example, in `do_text_line` we did not discuss scanning, since it was clear it could be done easily using stream operations. By contrast, the design of `doc` was carried out in more detail. This extra detail was not really necessary; for example, we need not have considered the details of producing pages. If we had stopped the design sooner, the rep would only be sketched, and the rep invariant and abstraction function would be omitted. In such a case, more design (but of a localized nature) would be needed during implementation—for example, to define the exact meanings of `lineno` and `pageno` in the rep of `doc`, and possibly even to identify and specify the private method `output_line`.

As soon as one target has been studied, the next step is to select a new target. First, the candidates are identified; these are abstractions whose implementations have not yet been studied but whose using abstractions have all been analyzed. One candidate is then chosen as the next target. While there are guidelines for making this choice, there are no rules. For example, there is no requirement that all abstractions at one level in the design must be studied before those at lower levels. Instead, common sense should be used, with the goal of finishing the entire design as quickly as possible. This is why we look at questionable abstractions as soon as possible and even study them before they are complete.

If, while studying the implementation of some target, we discover an error in the abstraction itself (i.e., a change in its specification), we must correct the error before proceeding further with the current target. We use the arcs in the module dependency diagram to discover all implementations affected by the error. Then we correct the design for those implementations. In the process, we may discover more errors that must also be corrected.

For example, suppose we had forgotten to pass the argument `from_right` to `line::justify`? This error might have been discovered while we were figuring out what to do with the extra spaces. By consulting the module dependency diagram, we would discover that the implementation of `doc` must be reconsidered. So we would back up, correct the error (perhaps we had forgotten to include the information in the rep of `doc`), correct the specification of `justify`, and then continue with the implementation of `line`.

When there are no more candidates, then ordinarily the design is finished. There is one special case, however. When two or more abstractions are mutually recursive, then none can be a candidate, since each is used by another abstraction that has not yet been studied. When dealing with mutually recursive abstractions, we must proceed with caution. One must be selected as a candidate and studied first. Since this candidate is used by another abstraction that has not been studied, we may discover later that its behavior is not what is needed. This problem is another indication of why mutual recursion must be viewed with suspicion.

The result of a finished design is a design notebook containing the completed module dependency diagram and sections for each abstraction. It is important that this notebook contain not only the decisions that were made but also the reasons for them. Ideally, this part of the documentation will explain both what problems are being solved by a particular structure and what problems are being avoided that would have been introduced by other structures studied during the design.

Because such documentation is difficult and time consuming to produce, it is often neglected. It is important in the later stages of program development, however, whenever a situation arises in which a design decision must be reconsidered or changed. The new decision can best be made by someone who fully understands the design. The documentation makes the needed information available, both to people other than the original designers and even to the original designers, who will forget it as time goes by.

Note that design is a top-down process; that is, we always reason from what is wanted to how to achieve it. In this way we keep our goal (the required program) firmly in view and are free to use our intuition about programs to guide us to a solution. Through experience in writing programs, programmers develop intuition about what is implementable with what efficiency. They also come to know what program structures are appropriate to various problems. This knowledge can be used to good effect in top-down design.

An alternative to top-down design is bottom-up design, which starts with what is known to be implementable and somehow proceeds from there to achieve what is wanted. Bottom-up design is not really a tenable process for any but the smallest programs. For large programs the gap between what is available and what is wanted is large and must be bridged by the introduction of many abstractions. This gap is easier to bridge top-down because we can concentrate on the thing that is least understood (the program that is wanted) and use what we know to help us. With top-down design, our intuition about what is implementable tells us whether we are making progress: The helpers should be easier to implement than the target. With bottom-up design, it is much harder to measure progress, since it is hard to evaluate how close we are to what is wanted.

In reality, we tend to go back and forth between top-down and bottom-up design. For example, we may investigate the implementation of key abstractions or questionable abstractions even when all uses are not yet understood. However, it is important that the design be driven from the top and that we avoid actually implementing any abstractions; doing implementation is truly a waste of effort, since the chances of all the details being right are small.

12.10 Summary

This chapter has discussed program design. Design progresses by modular decomposition based on the recognition of useful abstractions. We discussed how this decomposition happens and illustrated the design process with an example. We also discussed a method of documenting a design in a notebook.

The example used was a simple text formatter, and the resulting program was small, containing only six modules. In addition, the presentation of the design process was unrealistic—we did not make errors as we went along. In the real world, any design, even of a simple program like the formatter, requires a great deal of iteration, and many errors will be introduced and corrected as it progresses. Nevertheless, the basic methods we presented still apply, even to much larger programs. We have used it in such programs ourselves.

We do not claim that the formatter design is the best possible. In fact, the goal of the design process is never a "best" design. Instead it is an "adequate" design, one that satisfies the requirements and design goals and has a reasonably good structure. We discuss this issue in the next chapter.

Further Reading

Bentley, Jon L., 1982. *Writing Efficient Programs.* Englewood Cliffs, N.J.: Prentice-Hall.

Hester, S. D., David L. Parnas, and D. F. Utter, 1981. Using documentation as a software design medium. *Bell System Technical Journal* 60(8): 1941–1977.

Kernighan, Brian W., and P. J. Plauger, 1981. *Software Tools in Pascal.* Reading, Mass.: Addison-Wesley.

Parnas, David L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12): 1053–1058.

Exercises

12.1 Design and implement the KWIC program specified in chapter 11 and exercise 1 of chapter 11.

12.2 Form a team of three and design and implement a moderately large program.

Chapter 13

Between Design and Implementation

In this chapter we discuss briefly the two considerations that arise between the completion of a design and the start of implementation, namely evaluation of the design and choice of a program development strategy. We also discuss the relationship of abstraction to efficiency.

13.1 Evaluating a Design

Periodically during the design of a large program it is worthwhile to step back and attempt a comprehensive evaluation of the design so far. This process is called *design review*. Design reviews should always be conducted by a team composed of some people involved in the design and others who are not. Those not involved in the design should be familiar both with the program requirements and with the technology that will be used in implementing the design. They should also be familiar with the design itself.

It is important that both the designers and the outside reviewers understand that the point of a design review is to find problems with the design. While the designers will inevitably find themselves attempting to justify their design decisions to the outside reviewers, they should not view this as their primary goal. In addition, both the reviewers and the designers must keep in mind that the purpose of the review is only to find errors, not to correct them. Errors should be recorded in an error log, and then the review should continue (unless so many errors have been found that continuing is no longer productive).

It is useful for the designers to present not only the design but also the alternatives that were considered and rejected. This will give the outside reviewers a context for evaluating the chosen design. It may also help the reviewers to find flaws in the chosen design. A common problem is failure to apply design criteria uniformly. Explaining that an alternative was rejected because it failed to meet some criterion may well prompt the reviewers to notice that some other part of the design fails to meet that same criterion.

There are three critical issues to address in evaluating a design:

1. Will all implementations of the design exhibit the desired functionality? That is, will the program be "correct"?

2. Are there implementations of the design that will be acceptably efficient?

3. Does the design describe a program structure that will make implementations reasonably easy to build, test, and maintain? Also, how difficult will it be to enhance the design to accommodate future modifications, especially those identified during the requirements phase?

13.1.1 Correctness and Efficiency

Earlier in this book we discussed two approaches, testing and verification, to increasing our confidence that a program will behave as desired. Unfortunately, neither of these approaches can be applied to designs. Since designs cannot be run, testing is out of the question. If designs were presented completely in a formal language, some verification might be possible. However, formal specification of designs of large programs is beyond the current state of the art.

While there are no completely rigorous techniques for reviewing a design, it is important that design reviews be systematic. They should examine both local and global properties of the design. Local properties can be examined by studying the specifications of individual modules; global properties, by studying how the modules fit together.

Two important local properties are consistency and completeness. These were discussed in chapter 9. Another important local property is efficiency. The first step in estimating the overall efficiency of a system is to construct for each abstraction an expression relating its running time and storage consumption to its arguments. How accurately this can be done depends upon the completeness of the design. Consider the `sort` procedure:

```
void sort (array<int> *a) throw ( );
// modifies: a
// effects: Permutes the elements of a so that a is sorted in increasing order.
```

If the design does not specify any efficiency constraints for `sort`, relatively little can be said about the efficiency of its implementations or about the efficiency of abstractions to be implemented using `sort`. The problem is that implementations of `sort` span a wide range with respect to efficiency. Considerably more can be said about the efficiency of implementations if the design includes the following criterion:

> **efficiency:** worst case time = $n * \log(n)$ comparisons, where n is the size of `a`.

and more yet if the design states:

> **efficiency:** worst case time = $n * \log(n)$ comparisons, where n is the size of `a`. Maximum temporary main memory allocated is a small constant. No main memory added.

An important function of a design review is to discover places where the design needs to be more specific about what is required of implementations.

After each module has been evaluated in isolation, we examine the design as a whole. A good way to begin is by tracing paths through the design that correspond to various uses of the program. We select some test data and then trace how both control and data would flow through an implementation based on the design. This tracing process is sometimes called a *walk-through*. The test data are chosen in much the same way as described in chapter 8. However, since "testing" a design is much more labor intensive than testing an implementation, we must be more selective in choosing our test data.

Since the point of tracing the design is to convince ourselves that all implementations of the design will have the desired functionality, the success of this method is related to the completeness of the test cases. We are using the test cases to carry out an informal verification process. During the design review it is also important to discuss the completeness of the process, that is, to argue that all cases have been considered. Normal and exceptional cases should be considered separately.

Picking test cases for a design review is somewhat simplified by the fact that the data can be symbolic. We need only identify properties that the test data should have; we do not need to invent data with those properties. As an example, consider the `format` program designed in chapter 12. We start with normal input; that is, we assume that `format`'s `istream` argument can be read and its `ostream` arguments can be written. Since the initial contents of the output and error files are irrelevant, we need only consider what the contents of the input file, `ins`, should look like. The test should include cases involving various combinations of

1. short and long words, including words too long to fit on a single output line,

2. single and multiple spaces between words,

3. tabs and spaces at different positions on output lines—for example, the front, the middle, and the end,

4. short and long input lines, including input lines that are longer than output lines,

5. output lines where no justification is necessary,

6. output lines where more spaces have to be inserted than there are word breaks in the line, and

7. output lines containing exactly one word.

The following symbolic data cover some of these cases:

wd1 space wd2 space wd3 space wd4

where the sum of the lengths of wd1 and wd2 is less than the length of an output line, but the sum of the lengths of wd1, wd2, and wd3 is greater than the length of an output line.

We use these data to walk through the design of `format`. The walk-through is, in effect, a hand simulation of the design. The main thing we want to examine is the flow of information through the program. Here is how we might start a walk-through based upon the previous data:

1. After checking the modes of its arguments, `format` creates a new `doc`, `d`, by calling the `doc(outs)`. The specification of the `doc` constructor tells us that this `doc` is in "fill" mode. Nothing is written to `outs` by the constructor.

2. `format` next calls `do_line(ins,d,errs)`. At this point we note that `do_line` does not have direct access to `outs` and ask ourselves whether this will present a problem. It does not seem to.

3. `do_line` peeks at the first character of the first line of `ins`, discovers that it is not a period, and then calls `do_text_liine(ins,d)`. We note that `do_text_line` is not being passed `errs`. We note also that it has no way to pass information about any errors back to `do_line`. This presents no problem for `format` as currently specified, but is likely to present a problem if the specification is modified to allow something other than words, for example, information about changes in type face, within text lines. We should note this in our design documentation.

The walk-through continues in this fashion until the behavior of the entire program has been explored. In the process we estimate the efficiency of each module, so that we can construct estimates of worst-case and average efficiencies for the whole program.

Walk-throughs are a laborious and imprecise process. Experience indicates that designers are seldom able to examine their own designs adequately. The process works best when it is performed by a team of people including, but not dominated by, the designers.

Tracing through the entire design with a few sets of inputs helps us to uncover gross errors in the way the abstractions comprising the design are fit together. A good next step is to work bottom-up through the module dependency diagram, isolating subsystems that can be meaningfully evaluated independently of the context in which they will be used. Since these are likely to be considerably smaller than the system as a whole, we can trace more sets of test data through them.

The modifiability of the design should be addressed explicitly during the review. There should be a discussion of how the design must be changed to accommodate each expected modification. A plausible measure of how well the design accommodates modifications is how many abstractions must be reimplemented or respecified in each case. The best situation is one in which only a single abstraction needs to be reimplemented.

A walk-through forces us to look at the design from a different perspective than the one that characterized the design process. During design, we focused on identifying abstractions and specifying their interfaces. These abstractions arose from considering what steps were to be carried out, but our attention was focused on parts of the program separately. Now we go back over the steps carried out by the whole program as it uses the abstractions, and this exercise forces us to address the question of whether the abstractions can be composed to solve the original problem.

13.1.2 Structure

The most important structural issue to address in evaluating a design is the appropriateness of the module boundaries. There are two key questions to ask:

1. Have we failed to identify an abstraction that would lead to a better modularization?

2. Have we grouped together things that really do not belong in the same module?

We can provide no formula for answering these questions. What we can provide is a list of symptoms that occur when a program has been badly modularized. We shall concentrate on local symptoms, that is, problems that can be detected by looking at a single module or at the interface between two modules.

Coherence of Procedures

Each module in a design should represent a single coherent abstraction. The coherence of an abstraction can be examined by looking at its specification. A procedure should perform a single abstract operation on its arguments. (Our discussion applies to iterators too. An iterator maps its inputs to a sequence of items; this mapping should be a single abstract operation.)

Some procedures have no apparent coherence. They are held together by nothing more than some arbitrarily placed bracketing mechanism. In the early days of "structured programming" many people failed to understand that good program structure is basically a semantic notion. They looked for simple syntactic definitions of "well-structured." Many of these simplistic definitions included an arbitrary upper bound, such as one page, on the size of procedures. Another example of an arbitrary size restriction occurs in programs that must manage their own memory and are divided into modules to facilitate overlays. Such arbitrary restrictions frequently led to procedures totally lacking in coherence.

A second cause of lack of coherence is hand optimization of programs. An eagle-eyed programmer may notice that some arbitrary group of statements appears several times. In an attempt to save space, the programmer may bundle these statements into a procedure. In the long run, however, such optimizations are generally counterproductive, since they make the program harder to modify.

There are two reliable indicators of a total lack of coherence. If it seems that the best way to specify a procedure involves describing its internal structure (that is, how it works), the procedure is probably incoherent. The second good tip-off is difficulty in finding a suitable name for a procedure. If the best name we can come up with is "procedure1," there is probably something wrong with our design. If there is no apparent coherence to a procedural abstraction, we should rethink the design with the goal of eliminating that procedure.

Conjunctive coherence is a step up from no coherence at all. It is indicated by a specification of the form

A & B & C & ...

Conjunctive coherence usually occurs when a sequence of temporally contiguous actions is combined into a single procedure. A typical example is an abstraction whose job is to initialize all data structures. The specification of such an abstraction is likely to be a conjunction:

initialize A & initialize B & ...

Note that such a structure can make it more difficult to identify data abstractions, since part of the job of each type is taken over by the procedure.

In an environment in which procedure calls are unduly expensive, conjunctive coherence may be useful, since it can eliminate some calls. However, unless there is a strong logical connection between the actions, it is generally better not to combine procedures. The more we put into a procedure, the harder it will be to debug and maintain it. Furthermore, as we maintain a program, we are likely to discover occasions when it would be useful to perform some subset of the conjuncts. If this happens, the appropriate thing to do is probably to break up the original procedure. What people often do instead, however, is to add another procedural abstraction. This leads to more code to debug and maintain, and to a program that occupies more space at runtime.

Disjunctive coherence is indicated by a specification with an effects clause of the form

A | B | ...

often in the guise of an if-then-else or a conjunction of implications. A robust procedure is likely to involve some disjunctive coherence, to separate the normal return from exceptions. However, if the specification of what happens when the procedure returns normally involves disjunction, one should be concerned. Consider the specification in figure 13.1. The abstraction `get_end` can raise the signal `CE_Empty` or it can return normally and do one of two different things. Each of the these two would have been a perfectly reasonable abstraction in its own right; that is to say, we could have had the two abstractions of figure 13.2. Combining these two procedures into one has no advantage and several disadvantages. First, a call of the form `get_end(a, 1)` is likely to be harder for a reader to understand than a call of the form `get_first(a)`. Second, a new class of errors is possible—for example, a call of the form `get_end(a, 3)`. Third, a program using `get_end` is less efficient than one using the two abstractions of figure 13.2. Whenever a call to `get_end` is made, the caller knows which of the procedures is wanted. However, this information must be encoded into the second argument of the call, and `get_end` must test this argument to figure out what to do. This extra work requires both time and space. Also, we may implement `get_end` using subsidiary abstractions such as `get_first` and `get_last`, thus increasing the number of procedure calls that get executed.

Disjunctive coherence often arises from a misguided attempt to generalize abstractions. When a program design contains two or more similar abstractions, it is always worthwhile to consider whether a single more general abstraction might replace all or some of the similar ones. If successful, generalization saves space and programmer effort with little cost in execution

```
int get_end (list<int> * a, int j) throw (CE_Empty);
// requires: 0 < j < 3
// effects: if size(a) < 1 then throws CE_Empty
//   else if j = 1 then returns the first element of a
//   else if j = 2 then returns the last element of a
```

Figure 13.1 An example of disjunctive coherence.

```
int get_first(list<int> *a) throw (CE_Empty);
// effects: if size(a) < 1 then throws CE_Empty
//   else returns the first element of a

int get_last( list<int> *a) throw (CE_Empty);
// effects: if size(a) < 1 then throws CE_Empty
//   else returns the last element of a
```

Figure 13.2 Two coherent procedures.

speed or complexity in the implementation of the generalized abstraction. However, if the result is an abstraction with disjunctive coherence, then it is usually better not to do the replacement.

The method `line::justify` in the design of `format` is an example of successful generalization. We could have had two methods, one to justify from the left and the other to justify from the right, but this would have bought us little, except a larger program and more code to maintain. While there is a disjunction in `justify`'s specification, it is buried rather deeply. Most of the specification, and most of the code, is independent of the value of the argument `from_right`.

A generalization we did not make in `line` was combining `add_word`, `add_space`, and `add_tab` into a single method, such as

```
void add_token(token t) throw ( );
```

where `token` contains information about the token to be added. This would not have saved any code, and it would have been slower than having the three operations.

Occasionally the appearance of excessive disjunctive coherence indicates failure to introduce appropriate data abstractions into the design. In such cases combining several distinct functions into one procedure may be an attempt to encapsulate representation information that should have been encapsulated in a missing type. In effect, the type is implemented by a single procedure, and extra arguments are used to identify the operation being called.

Coherence of Types

Each operation of a type should be a coherent procedure or iterator. In addition, a type should provide an abstraction that its users can conveniently think of as a set of values and a set of operations intimately associated with those values. One way of judging the coherence of a type is to examine each operation to see whether it really belongs in the type. As discussed in chapter 4, a type should be *adequate*, that is, should provide enough operations so that common uses are efficient. In badly designed types one frequently finds operations that do not seem particularly relevant to the abstraction and whose implementation can take little or no advantage of direct access to the representation. It is generally better to move such operations out of the type. If fewer operations have access to the representation, it is easier to modify the representation if it becomes desirable to do so.

Consider, for example, a stack type containing `sqrt_top` method:

```
float sqrt_top ( ) throw (CE_Empty);
// effects: if size(this) = 0 then throws CE_Empty else returns the
//   square root of top(this).
```

`sqrt_top` has little to do with stacks, and most of its implementation will not depend upon the representation of stacks. Therefore this operation should be moved out of the stack abstraction.

Communication between Modules

A careful examination of how much and what kind of information is exchanged between modules can uncover important flaws in a design. Throughout this book we have stressed the importance of narrow interfaces: A module should have access to only as much information as it needs to do its job. Our methodology is designed to encourage narrow interfaces—for example, by not allowing procedures to refer to global variables—but it is still possible to pass too much information to a module.

A module may be passed too much information because a type has not been identified. In the absence of a type, all modules that would have communicated in terms of the abstract objects instead communicate in terms of the representation. The result is modules that have much wider interfaces than necessary; instead of being related only through the type, they share knowledge of how that type is implemented. Note that this includes the abstraction function and representation invariant in addition to the representation itself. Note also that all using modules must be considered in reasoning that the implementation of the (missing) type is correct. Furthermore, if the implementation of the type changes, every using module must also change.

Even if all needed types have been identified and are implemented by their own clusters, some interfaces may still be wider than necessary. Well-designed programs frequently have types that include a great deal of information. Some modules may not need to access all of this information. Yet many designs call for passing the entire abstract object when a small piece of it would be sufficient. For example, we might have a type `student_record` that includes, among other things, a student's name, social security number, residence, and transcript. A procedure, `print_address`, that prints an address label might need only a student's name and residence. Such a procedure should not be passed the whole student record; instead the information it needs should be extracted from the student record by its caller and passed to it explicitly. There are several good reasons for this:

1. If `print_address` is passed an object of type `student_record`, its implementor will have to know how to extract the needed information—that is, what methods to call. If the specification of `student_record` is changed, the implementation of `print_address` may have to be changed. None of this would be necessary if the needed information were passed explicitly.

2. The implementation of `print_address` may have a bug that causes it to mutate the `student_record`. Such bugs can be very hard to find.

13.2 Ordering the Program Development Process

Throughout this book we have stressed a program development strategy that is basically top-down. While we recognize that program development is an iterative process, we have argued that in each iteration, specification and design should precede implementation. We have not yet discussed how to order the process of going from a design to an implementation. The basic choice is between a top-down strategy and a bottom-up strategy.

The traditional mode of development is bottom-up. In bottom-up development we implement and test all modules used by a module M before we implement and test M. Consider, for example, implementing a design with the module dependency diagram shown in figure 13.3. We might begin by implementing and unit testing D and E. We might then implement and test B and C. In testing B and C we might use D and E in order to avoid writing stubs. Doing this implies that we are no longer performing unit testing, since we are not testing a single module. Finally, we would implement and test A. Note, by the way, that this is not the only possible

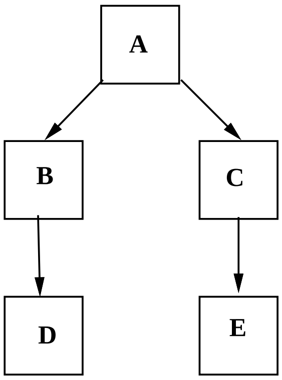

Figure 13.3 A simple module dependency diagram.

bottom-up order. We might equally well have used the order D, B, E, C, A.

In top-down development, we implement and test all modules that use a module M before implementing and testing M. Possible top-down orders for the above example include A, B, C, D, E and A, C, E, B, D. Just as bottom-up development reduces our dependence on stubs, top-down development reduces our dependence on drivers. It is important that top-down development be accompanied by careful unit testing of all modules. If we tested B only as it is used by A, we might see only part of B's specified behavior. Were we to change A later, new bugs might be revealed in B. Therefore if we choose to use A as a driver for B, we must make sure that A tests B thoroughly.

Neither development strategy strictly dominates the other. Most of the time, it seems best to work top-down, but there can be compelling reasons to pursue a bottom-up approach. We advocate a mixed strategy in which one works top-down on some parts of the system and bottom-up on others.

Top-down development has the advantage of helping us to catch serious design errors early on. When we test a module, we are testing not only the implementation of that module but also the specifications of the modules that it uses. If we follow a bottom-up strategy, we might easily spend a great deal of effort implementing and testing modules that are not useful because there is a problem with the design of one of their ancestors in the module dependency diagram. A similar problem can occur in top-down development if we discover that some crucial descendent module is unimplementable or cannot be implemented with acceptable efficiency. Experience indicates, however, that this problem occurs less often. This may be because lower-level abstractions are often similar to things we have built before, whereas higher-level abstractions tend to be more idiosyncratic.

In top-down development it is always possible to integrate one module at

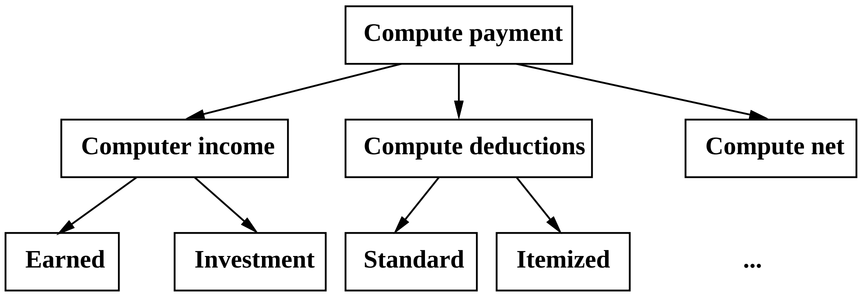

Figure 13.4 Module dependency diagram for an income tax program.

a time. We merely replace a stub by the module it is intended to simulate. In bottom-up development, on the other hand, we tend to integrate several modules at once; in most cases a single higher-level module corresponds not to one but to several drivers. For example, when A is integrated into the program of figure 13.3, it will be replacing drivers for both B and C. Since system integration tends to proceed more smoothly if we can add one interface at a time, top-down development has a significant advantage in this regard.

Top-down development also increases the likelihood of bringing up useful partial versions of the program being developed. Suppose a program to compute income tax payments has the (partial) module dependency diagram of figure 13.4. By working top-down, it is possible to bring up many useful partial implementations of this design. A system without `investment` and `itemized` procedures, for example, might be quite useful to some people.

Even if partial versions of the system cannot be used productively by the customer, bringing partial versions up early in the development process may have some important benefits. If implementors are able to demonstrate something early, they can get feedback that may reveal problems with the requirements for the program. Furthermore, on lengthy projects the ability to produce a series of working partial systems seems to help the morale of both the customer and the developers.

While bottom-up development delays the point at which one has a working partial system, it does lead to the earlier completion of useful subsystems. These subsystems generally have wider applicability than do the partial systems we get when working top-down. This is particularly true of low-level subsystems (for example, input/output subsystems). In an environment in which several related systems are being developed simultaneously, it can be helpful to bring up shared subsystems by working bottom-up. Moreover, it is sometimes easier to build a low-level subsystem than to build a stub to simulate it.

Another potential advantage of bottom-up development is that it may place less of a load on machine resources than does top-down development. This is particularly true early in the development process. Top-down development tends to use more machine time because we are running a larger part of the system during many phases of development. Typically, full tests of the system are developed and then run each time a new module is added. Such tests may exercise many other parts of the system besides the module being added, and thus consume much time and space that really is not needed for this step of the integration. On the other hand, such a systematic approach to testing is bound to uncover more errors and be more thorough than a more ad hoc approach.

A related advantage of bottom-up development is that it may allow us to proceed in the absence of some computational resource. Consider building a system that is to run on a machine with 32MB of memory. If that machine has not yet arrived, but there is a similar machine with 16MB available, we might be able to do quite a bit of bottom-up development before memory becomes a problem. We might run out of memory sooner working top-down.

In deciding when to work top-down and when to work bottom-up, we must consider nontechnical as well as technical issues. Sometimes large implementation efforts must begin before an adequate team of programmers has been assembled. In such cases top-down development seems to work best. Since module dependency diagrams are always narrower at the top than at the bottom, the need for programmers tends to grow as we get deeper into the implementation. If, on the other hand, the programming team is already assembled and ready to go to work, bottom-up development may be better.

Generally, we recommend a mixed strategy in which top-down development is favored but not followed entirely. In working out this strategy, both technical and nontechnical factors should be taken into account. The most important point is that a development strategy should be defined explicitly before implementation begins. Some insight into implementation and testing order can be gained by considering `format`:

1. `format` and `do_line` should probably be implemented first. They are simple to implement and together can be used as drivers to test lower-level abstractions. They can be tested easily since stubs for `do_text_line`, `do_command_line`, and the `doc` constructor and `terminate` method, need only print the arguments with which they are called.

2. `do_command` and `do_text_line` should probably be implemented next. Again, they can be easily tested, since stubs for the procedures they call need only print their arguments.

3. At this point we have only two abstractions left, `doc` and `line`. A

strictly top-down development would lead us to implement `doc` next, using stubs for the `line` operations that just print their arguments. A problem with this approach is that to test `doc` thoroughly we need to run through a fair amount of input, and it might be easier to deal with a version of `line` that approximates the final implementation. For this reason it probably makes sense to consider implementing and testing parts of each of `doc` and `line` at the same time. A reasonable approach to just implement the no-fill case. This will allow us to ignore the complications caused by justification for now, yet produce a running version of the formatter.

4. We conclude the implementation by finishing up `doc` and `line`, including the implementation of `line::justify`. It will be the most complicated procedure to implement and require the most testing. Testing it will be easier once the rest of the program is in place.

13.3 The Relationship of Abstraction to Efficiency

Throughout this book we have argued that well-chosen abstractions simplify programming, testing, debugging, and maintenance. We have put less emphasis on their impact on program efficiency.

Most of the time, well-chosen abstractions have a salutary effect on efficiency. They can be a big help in designing efficient algorithms. Moreover, by increasing the modularity of programs, they make it far easier to introduce modifications. This, in turn, helps us to eliminate bottlenecks and respond to changes in efficiency requirements. Occasionally, however, the introduction of abstractions can have an adverse impact on efficiency. An example of a serious problem is a procedure that could take advantage of accessing the rep of some type. When such a problem arises, there is nothing to do but consider reworking the design.

Less serious problems arise when the programming language being used imposes an unacceptably high cost on the use of abstractions. The style of programming we advocate leads to programs that make a large number of procedure and method calls. Calling a procedure or method always involves some runtime overhead; how much depends upon the programming language and the compiler. Additionally, there may be overhead stemming from the introduction of redundant computation. The problem is that for any particular use of a procedure or method there may be information available that is not available for all uses. In figure 13.5, for example, the check for the empty stack in the body of `P` is redundant for `Q`'s call of `P`.

The solution to this problem is an optimization called *inline substitution*. Doing an inline substitution means replacing a procedure call with the body of the procedure being called. This involves replacing occurrences of

```
void Q (stack<int>*s) ... {
  ...;
  if (!(s → empty( )) {
    int y = P(s);
    ...;} // use y in here
  } // end Q

int P (stack<int>*s) throw ( ) {
  if (!(s → empty( ))
    {int x = s → pop( );
    // use x and return it; }
  } // end P
```

Figure 13.5 A procedure call of P by Q.

```
void Q (stack<int>*s) ... {
  ...
  if (!(s → empty( ))
    { t = s; // body of P goes here
      if (!(t → empty( )) {
        int x = t → pop( );
        // use x;
        y = x; }
    ...; } // this is Q's use of y
  } // end Q
```

Figure 13.6 Replacing P's call by its body.

arguments in the body by the corresponding actuals in the call. If the body has local identifiers that conflict with identifiers in the calling environment, these must be renamed. Figure 13.6 shows the result of applying inline substitution to the code in figure 13.5. Given the code in figure 13.6, a reasonably good optimizing compiler will remove the redundant code.

Some compilers perform inline optimization automatically, but most do not. Since inline substitution is a source-level optimization, it can be done by programmers. Hand optimizations of this sort can be useful when applied on a small scale to critical sections of code. Attempting to do it on a large scale, however, is not a good idea. If this is desired, the best approach is to write a program to do it automatically. Unfortunately such a program is moderately difficult to write.

13.4 Summary

In this chapter we discussed some things that should be done between the completion of a design and the start of implementation. The key points to take away are the importance of conducting a systematic evaluation of the design and developing a precise plan specifying the order of implementation and testing of the modules comprising the design.

In a design review one considers whether or not implementations of the design will exhibit the desired behavior and efficiency, and whether or not the program structure described by the design will be reasonably easy to build, test, and maintain. We suggested conducting design reviews by tracing the path of symbolic test data through the design. We also suggested several criteria that could be used in evaluating structural issues. All of the criteria were related to the appropriateness of module boundaries.

We did not give fixed rules for picking an order of implementation and testing. We discussed the relative merits of top-down and bottom-up development and testing. Our conclusion was that most of the time it is best to follow a mixed strategy, but with an emphasis on proceeding top-down.

In our discussion of design reviews we presented an extremely abbreviated review of the text formatter designed in chapter 12. The reader should not infer from this that design reviews should be conducted only after a design is complete. For larger programs it is imperative that we conduct careful reviews during the design phase. It is also important to start considering how the implementation effort is to be organized. A need for early completion of subsystems, for example, can have a significant impact on design decisions.

Further Reading

Lampson, Butler W., 1984. Hints for computer system design. *IEEE Software* 1(1): 11–28.

Myers, G. J., 1979. *The Art of Software Testing.* New York: John Wiley & Sons.

Exercises

13.1 Perform a design review for some program that you have designed. Be sure to include a discussion of the structure and modifiability of the program as well as a discussion of its correctness.

13.2 Define an implementation strategy for a program that you have designed.

13.3 A student proposes that in the design of `format` (chapter 12) the `line` methods `justify`, `clear`, and `output` should be replaced by a single operation that does all three things. A second student objects that this would lead to one of the bad forms of coherence. What kind of coherence would result? What are the advantages and disadvantages of merging the three operations, or of merging just `clear` and `output`?

13.4 Suppose that in the design of `format` (chapter 12) we had invented a `command` data type that encapsulated all information about commands, including their format and their meaning. We could then change the `doc` abstraction so that it would have a single method in place of `set_fill` and `set_nofill`. Evaluate this alternative design with respect to both the current specification of the formatter and future modifications involving more commands.

13.5 Suppose that in the design of `format` (chapter 12) we had not introduced the `line` abstraction. Evaluate this alternative design with respect to both the current specification of the formatter and future modifications.

13.6 Suppose that in the design of `format` (chapter 12) we had introduced neither `line` nor `doc`. Evaluate this alternative. Discuss the coherence of the design and also the issues of module communication discussed in section 13.1.2.

13.7 The `map` abstraction was discussed in exercise 3 of chapter 5. It provided an `insert` method to add a string with its associated element to the `map`, and a `change` method to change the element associated with the string. Suppose these two operations were replaced by a single method that added the association if it did not already exist, and changed it if it did. Discuss the coherence of this modified abstraction. How does the modified abstraction compare with the original?

Chapter 14

A Quick Look at Program Verification

This chapter is a short introduction to a complex topic. We hope to give the flavor of the process of program verification, without getting bogged down in the myriad details involved in actually constructing rigorous proofs of program properties. We believe that an understanding of what is involved in the formal verification of programs is an asset in informal reasoning about programs, and the most important thing to carry away from this chapter is thus simply a sense of the basic techniques involved in such formal reasoning. The structures of the proof rules and of the proofs are important; the details are not.

Program verification involves reasoning about program texts. This distinguishes verification from testing, which is always based upon observing computations. In verification we examine the program text and make inferences about the set of computations described by the program. We frequently refer to the set of computations to justify our method of reasoning, but we never construct this set or any part of it.

We start this chapter by introducing the basic program verification paradigm and reasoning about straight-line code. We next discuss programs containing **if** statements and **while** loops. Then we turn to the process of proving that procedures satisfy their specifications and of reasoning about programs that invoke procedures.

We shall generally avoid precise definitions in favor of illustrative examples. Moreover, we shall not treat all statement and expression forms of C++; instead we shall rewrite code whenever necessary to make use of just the part of C++ that is covered. Other issues that will not be discussed include aliasing, recursive procedures, iterators, and exceptions.

14.1 Reasoning about Straight-Line Code

We shall use *total correctness formulas* to state what we wish to verify. Each formula includes two predicates, a *pre-condition* and a *post-condition*, and a segment of code. The domain of the predicates is the set of all states of computations. A total correctness formula has the form

$$\mathrm{P} \quad \{\mathrm{S}_1;\ \mathrm{S}_2; \ldots \mathrm{S}_n;\} \quad \mathrm{Q}$$

Such a formula says that if the state of the computation before executing the statements $\mathrm{S}_1;\ \mathrm{S}_2; \ldots \mathrm{S}_n;$ satisfies the pre-condition P, we want the state following execution to satisfy the post-condition Q. Of course, if the pre-condition is false in the initial state, then the whole formula is (vacuously) true. This is analogous to the rule of logic that

$$(\mathrm{FALSE} \Rightarrow \mathrm{P}) = \mathrm{TRUE}.$$

Here we follow the convention of using uppercase letters in formulas for identifiers, such as TRUE and FALSE, that do not appear in programs. Note, by the way, that the pair of predicates in these formulas bears a striking resemblance to the body of a procedure specification. The pre-condition is like the requires clause, and the post-condition like the effects clause.

Consider an example:

```
TRUE
{ x = 0;
  y = 1; }
y > x
```

How can we convince ourselves of the validity of this formula? That is, how do we know that this program fragment terminates, and that when it is finished, the value of y is greater than the value of x? We begin by arguing that exactly two statements get executed. After the second statement is executed, the value of y is 1 and the value of x is whatever it was after the first statement was executed, that is, 0. We complete the argument by observing that $1 > 0$. Note that in this argument we reasoned about

1. the flow of control through the program,
2. the meaning of the assignment statement, and
3. the meaning of the $>$ relation on integers.

The basic paradigm we shall use in all our reasoning about programs is as follows:

1. Locate all paths between the two predicates.

2. For each path, work backward from the final predicate to discover a derived predicate R that must be true prior to the first statement in the path if the final predicate is to hold. Do this one statement at a time.

3. Use our knowledge about the types of the values of the variables occurring in the predicates to convince ourselves that for each path the pre-condition implies the derived predicate R.

The simple proof just outlined can now be restated more carefully. Pushing $y > x$ back across the second assignment simplifies the program fragment to

TRUE {x = 0;} 1 > x

and pushing $1 > x$ across the remaining assignment gives

TRUE $\Rightarrow 1 > 0$

which, by the rules of logic and what we know about integers, simplifies to TRUE. In constructing this proof, we used the following *rule of assignment* to push predicates backward across assignment statements:

> Let P be any predicate and e any side-effect-free expression: If P is to be true after the assignment $x = e$, P with e substituted for all free occurrences of x (that is, occurrences not bound to a quantifier) must be true before the assignment. That is to say, P with e substituted for all free occurrences of x is the *weakest pre-condition* that will ensure that P holds after the assignment $x = e$.

(This rule is valid only when there is no *aliasing* between `x` and the variables in `e`. Aliasing occurs when there are two or more distinct names for the same location. One way in which this happens in C++ is via call by reference. We will not treat programs that use call by reference in this chapter.)

At first glance, it may seem a bit odd to be working backward through the program. The advantage of working this way is that our reasoning is goal directed: At each step, knowing what we wish to be true at the conclusion of some part of the program, we compute what needs to be true prior to that part. When we reach the start of the program, we have the weakest pre-condition sufficient to imply the desired post-condition ("weakest" in the sense that any other sufficient pre-condition implies it). We complete the proof by showing that the given pre-condition implies the weakest pre-condition.

If we were to work forward, we would begin with what was known to be true at the start of some part of the program and compute what must be true at the conclusion of that part. When we reached the end of the program, we would have the strongest post-condition implied by the known pre-condition ("strongest" in the sense that it implied all other valid post-conditions). We would complete the proof by showing that this strongest post-condition implied the given post-condition. The problem with this approach is that as we work forward through the program, we have no systematic way to prune irrelevant information. We might accumulate a large number of things that happen to be true but are totally unrelated to what we wish to prove.

We observed earlier that the two predicates in a total correctness formula are similar to the requires and effects clauses in procedure specifications. There is, however, a crucial difference. In a procedure specification the effects predicate deals with two states, the one before and the one after the call. When we refer to the value of a formal, we qualify that formal (sometimes implicitly) with *pre* or *post*. In a total correctness formula each predicate deals with only one state. This presents a slight technical problem when we need to express a relationship between initial and final values.

As an example, consider a total correctness formula asserting that the statement $x = x + 1$ increments x. The post-condition needs to refer to both the initial and final values of x. To do this, we use the trick of introducing a *fresh variable* x_0 in the pre-condition and asserting that it has the same value as x. It is "fresh" in the sense that it does not already occur anywhere in either the program text or the given pre- or post-condition. To assert that $x = x + 1$; increments x we write

$$\mathrm{x_0 = x} \quad \{\mathrm{x = x + 1;}\} \quad \mathrm{x > x_0}$$

To verify this formula, we push the predicate $x > x_0$ across the assignment statement to yield

$$\mathrm{x_0 = x \Rightarrow (x + 1 > x_0).}$$

Substituting x_0 for x in the conclusion gives $x_0 + 1 > x_0$, which, assuming x is of type integer, simplifies to TRUE.

14.2 Reasoning about Programs with Multiple Paths

Thus far we have looked at programs that have only one path. Now consider a program with two paths through it, such as

if (x == 1) y = 0; **else** y = x;

Here there is one path for the **then** clause (the clause executed when the predicate is true) and one for the **else** clause. In reasoning about the **then** clause, we can assume that the predicate of the conditional is true; in reasoning about the **else** clause, that the predicate is false. In general, to reason about programs with **if** statements we use the following *case analysis rule:*

> If the evaluation of b is side-effect-free, proving
> P {**if** (b) s1; **else** s2;} Q
> is equivalent to proving both P & b {s1;} Q and P & ¬b {s2;} Q.

A concise way of writing a *rule of inference* such as this is

$$\frac{\text{side-effect-free (b)},\ \text{P \& b \{s1;\} Q},\ \text{P \& } \neg\text{b \{s2;\} Q}}{\text{P \{}\textbf{if}\text{ (b) s1; }\textbf{else}\text{ s2;\} Q}}$$

In this notation the clauses separated by commas above the line are called *hypotheses*. The single clause below the line is called the *conclusion*. Such a rule can be read as stating that demonstrating the truth of all the hypotheses is sufficient to prove the truth of the conclusion.

The case analysis rule is the first of a number of rules we shall use to reduce the verification of a complex total correctness formula to a set of simpler formulas or *verification conditions*. For example, to prove

> (x = 1) | (x = 0)
> { **if** (x == 1) y = 0; **else** x = y; }
> (y = 0)

we need to prove three verification conditions:

1. side-effect-free (x == 1)
2. ((x = 1) | (x = 0)) & (x = 1) { y = 0; } y = 0
3. ((x =1) | (x = 0)) & (x ≠ 1) { y = x; } y = 0

The validity of the second and third verification conditions can be shown using the rule of assignment. The validity of the first verification condition follows from the fact that the only procedure invoked by the expression is the `operator ==` method of *int*, which modifies nothing.

The kind of case analysis used above is applicable to any program segment with a relatively small number of paths through it. Unfortunately, as noted in chapter 8, most programs have a great many paths through them. Consider, for example,

$x \geq 0$ & $y = 0$
{ **while** (y == x) y = y + 1; }
$y = x$

There are an infinite number of paths through the program fragment in this formula, one for each possible value of x. If $x = 0$, the path contains no assignments; if $x = 1$, the path has one assignment; if $x = 2$, it has two assignments; and so forth. Clearly we cannot reason about this program by enumerating all paths through it. An ad hoc argument about the validity of this formula would probably go:

1. The pre-condition guarantees that $x \geq y$.

2. Every time through the loop the program increments y, thus bringing it closer to x.

3. Since there are only a finite number of integers between x and y, y will eventually equal x.

4. The program exits the loop and terminates as soon as $y = x$.

This is analogous to proving a theorem about all positive integers by ordinary induction. We first prove that the theorem holds for 1. We then show that if it holds for an arbitrary positive integer n, it must also hold for $n+1$. We conclude that the theorem must hold for all positive integers. In reasoning about programs with loops, we resort to induction over the number of iterations through each loop.

A crucial step in constructing an inductive proof is choosing the induction hypothesis. In reasoning about loops, we call this hypothesis the *loop invariant*. Choosing a suitable loop invariant can be challenging. It must satisfy four conditions. Suppose we wish to prove that

P { **while** (b) S; } Q

We must be able to show that

1. The invariant is true at the start of the first iteration of the loop; that is, P implies the invariant.

2. If the invariant is true before evaluating the predicate controlling the loop, it is true after evaluating this predicate.

3. If the invariant and the predicate controlling the loop are true at the start of any iteration through the loop, the invariant is true at end of that iteration.

4. The conjunction of the loop invariant and the negation of the predicate controlling the loop implies Q.

The fourth condition implies that if the loop terminates in a state satisfying the invariant, Q will hold upon termination. The first three conditions guarantee that the invariant will indeed hold upon termination. Thus if we succeed in finding a loop invariant that satisfies these four conditions and in showing that the loop always terminates, we can conclude that the loop terminates in a state satisfying Q. This is expressed by the following *loop invariant rule*:

$$\frac{\begin{array}{l} \mathrm{P} \Rightarrow \mathrm{Inv}, \\ \mathrm{Inv}\ \{\mathrm{b}\}\ \mathrm{Inv}, \\ \mathrm{Inv}\ \&\ \mathrm{b}\ \{\mathrm{S}\}\ \mathrm{Inv}, \\ \mathrm{Inv}\ \&\ \neg\mathrm{b} \Rightarrow \mathrm{Q}, \\ \textbf{while}\ (\mathrm{b})\ \mathrm{S};\ \mathrm{terminates} \end{array}}{\mathrm{P}\ \{\textbf{while}\ (\mathrm{b})\ \mathrm{S};\}\ \mathrm{Q}}$$

For our example, $x \geq y$ is a suitable loop invariant. It is good programming practice to include the loop invariant as a comment at the start of loop. To prove that

$$\begin{array}{l} \mathrm{x} \geq 0\ \&\ \mathrm{y} = 0 \\ \quad \{\ //\ \mathrm{invariant}\ \mathrm{x} \geq \mathrm{y} \\ \qquad \textbf{while}\ (\mathrm{y}! = \mathrm{x})\ \mathrm{y} = \mathrm{y} + 1;\ \} \\ \mathrm{y} = \mathrm{x} \end{array}$$

we show that the loop always terminates and demonstrate the validity of each of the following:

1. The loop invariant holds on entry to the loop:

$$(\mathrm{x} \geq 0\ \&\ \mathrm{y} = 0) \Rightarrow \mathrm{x} \geq \mathrm{y}$$

2. The loop invariant is maintained by evaluation of the test controlling the loop:

$$\mathrm{x} \geq \mathrm{y}\ \{(\mathrm{y}! = \mathrm{x})\}\ \mathrm{x} \geq \mathrm{y}$$

3. The loop invariant is maintained by the body of the loop:

$$\mathrm{x} \geq \mathrm{y}\ \&\ \neg(\mathrm{y} = \mathrm{x})\ \{\mathrm{y} = \mathrm{y} + 1;\}\ \mathrm{x} \geq \mathrm{y}$$

4. The loop invariant and the negation of the test imply the post-condition:

$$((\mathrm{x} \geq \mathrm{y}) \ \& \ \neg(\neg(\mathrm{y} = \mathrm{x}))) \Rightarrow (\mathrm{y} = \mathrm{x})$$

The first and fourth conditions simplify immediately to TRUE. The second follows directly from the fact that the evaluation of $(x \mathrel{!=} y)$ is side-effect-free. By pushing the final predicate across the assignment statement, we can simplify the third verification condition to the predicate

$$\mathrm{x} \geq \mathrm{y} \ \& \ \neg(\mathrm{y} = \mathrm{x}) \Rightarrow (\mathrm{x} \geq \mathrm{y} + 1)$$

Since $x \geq y$ and $\neg(y = x)$ imply $x > y$, this predicate also simplifies to TRUE. Note, by the way, that this part of the proof is the inductive step. We use the fact that the loop invariant holds on entry to the loop body to show that it holds on completion of the loop body.

To complete our total correctness proof, we need to show that the loop always terminates. We do this by associating a *decrementing function* with the loop. This function maps a subset of the program variables to some well-ordered set. (A set is *totally ordered* if it is possible to compare every member of the set to every other member. A totally ordered set is *well-ordered* if every nonempty subset of it has a least element.) The nonnegative and positive integers are well-ordered. The set of all integers is not, since it has no least element. The set of nonnegative rationals is not, since there are nonempty subsets with no least element—for example, all positive rationals. In this chapter we shall use the nonnegative integers as the range of our decrementing functions.

The decrementing function must satisfy two conditions:

1. We must be able to show that every iteration of the loop reduces the value to which the decrementing function maps.

2. We must be able to show that the conjunction of the loop invariant and the fact that the decrementing function maps to the least element of the well-ordered set implies the negation of the predicate controlling the loop; that is, it implies that the loop will be exited.

If we can exhibit a decrementing function with these two properties, we can conclude that the loop always terminates. This is so because each time through the loop the decrementing function must map to a smaller value (condition 1); there are only a finite number of values between the initial value of the decrementing function and the least element of its domain (this is a property of well-ordered sets); and when the program reaches a state in which the decrementing functions maps to the least element, we exit the loop (condition 2). The result is analogous to the loop invariant rule: *Even though there may be infinitely many paths through a program, we can reason about the program's termination by examining only a finite number of them.*

For our example, $(x - y)$ is a suitable decrementing function. To prove that the loop in

$\mathrm{x} \geq 0$ & $\mathrm{y} = 0$
    { // invariant $\mathrm{x} \geq \mathrm{y}$
      // decrements $(\mathrm{x} - \mathrm{y})$, with range nonnegative integers
    while (y != x) y = y + 1; }
$\mathrm{y} = \mathrm{x}$

always terminates we must show that

1. The decrementing function is decremented by the body:

$$(\mathrm{x} - \mathrm{y}) = \mathrm{d}_0 \; \& \; \neg(\mathrm{y} = \mathrm{x}) \; \& \; \mathrm{x} \geq \mathrm{y} \; \{\mathrm{y} = \mathrm{y} + 1;\} \; (\mathrm{x} - \mathrm{y}) < \mathrm{d}_0$$

2. The decrementing function is not incremented by the test:

$$(\mathrm{x} - \mathrm{y}) = \mathrm{d}_0 \; \{\neg(\mathrm{x} = \mathrm{y})\} \; (\mathrm{x} - \mathrm{y}) \leq \mathrm{d}_0$$

3. When the decrementing function reaches 0, the loop is terminated:

$$((\mathrm{x} - \mathrm{y}) = 0) \; \& \; (\mathrm{x} \geq \mathrm{y}) \Rightarrow \neg(\neg(\mathrm{y} = \mathrm{x}))$$

All of these simplify easily to TRUE.

We started out by observing that the basic problem presented by a program with a loop is that it is not possible to enumerate all the paths through the program. This has two ramifications:

1. The proof of termination is no longer trivial.

2. It is not immediately obvious how to reduce arguments about the program's behavior to a finite number of steps.

We have dealt with these two issues separately. First, we argued that if the loop were to terminate, it would terminate in a state satisfying the post-condition. This is called a *partial correctness* argument, and the crucial step is inventing a loop invariant. We then argued that the loop would indeed terminate. The crucial step here was exhibiting a decrementing function. The combination of partial correctness and termination is called *total correctness*.

Students frequently have difficulty learning to write loop invariants. Fortunately, it is easy to tell whether or not one has an appropriate invariant. If the invariant is inappropriate, at least one step of the verification will fail. Most of the time we can quickly deduce the problem with the invariant by studying how the verification failed.

14.3 Reasoning about Procedures

The introduction of procedures complicates the proof system needed to reason about programs, but it also reduces the complexity of proofs constructed in that system. This is similar to the introduction of procedures, which complicates the definition of a programming language but simplifies the writing of programs in that language. Appropriate use of procedures makes it easier to understand a program and to reason formally about it. We reason separately about the correctness of a procedure's implementation and about parts of the program that call the procedure. To prove the correctness of a procedure definition, we show that the procedure's body satisfies its specification. When reasoning about invocations of a procedure, we use only the specification.

We shall limit our discussion of procedures to procedures that return values but do not mutate their arguments. Furthermore, we consider only procedures that take all their arguments by value.

14.3.1 Reasoning about Procedures with No Side Effects

To prove that a procedure satisfies its specification, we first convert its body and its specification into a total correctness formula of the form we have been looking at,

P {body} Q

We then use the techniques discussed earlier to reason about this formula.

To make it easy to turn the specification into a formula, we will describe the result of the procedure as being the value of an extra formal, named `res`. An example is given in the specification for the `factorial` procedure shown in Figure 14.1.

The pre-condition of the formula is based upon the requires clause of the specification. We construct it by first conjoining to the requires predicate a predicate $x = x_0$, where x_0 is a fresh variable, for each formal argument x. This "saves" the initial values of the arguments. We then conjoin the predicate $\sigma_0 = \sigma$. This is introduced to allow us to deal with the modifies

```
int factorial (int x);
// requires: x ≥ 0
// modifies: nothing
// effects: res = x!

int factorial (int x) {
  int ans;
  ans = 1;
  while (x > 0) {
    ans = ans * x;
    x = x − 1; }
  return ans; } // end factorial
```

Figure 14.1 A specification and implementation of *factorial*.

clause of the specification. The reserved symbol σ stands for the current mapping from objects to their values. The fresh variable σ_0 is introduced to save this mapping.

The post-condition is based upon the effects clause of the specification. We construct it by first substituting the saved initial value of each argument formal for every free occurrence of that formal in the effects predicate. (Recall that in the body of a specification the appearance of an unqualified argument formal stands for the initial value of that formal.) We then account for the *modifies nothing* by conjoining the predicate $\sigma_0 \subseteq \sigma$. By using a subset here, we permit the procedure to create new objects but guarantee that no existing objects are modified.

The statement part of the formula is constructed from the body of the implementation and the header of the specification. To transform the body we first delete the variable declarations. (Strictly speaking, we should not do this. In a completely formal treatment, all syntactic objects, including declarations, are dealt with in the proof rules. The rules associated with declarations are used to govern what axioms may be applied in reasoning about the values of variables. For example, the declaration

```
int x;
```

indicates that we can use the theory associated with integers in reasoning about the value of x.) Next, we connect the value returned by the procedure with the returned value in the specification; we do this by replacing any **return** statement of the form **return** *ans* by an assignment statement of the form *res* = *ans*. (We assume that there is only one **return** statement and that it is always the last statement in the procedure body.)

Figure 14.1 contains a specification and an implementation of a factorial function. When applied to this example, the transformations just described yield the verification condition

$$
\begin{array}{l}
x \geq 0 \ \&\ x_0 = x \ \&\ \sigma_0 = \sigma \\
\quad \{\ \text{ans} = 1; \\
\quad\ \ \textbf{while}\ (x > 0)\ \{ \\
\qquad\quad \text{ans} = \text{ans} * x; \\
\qquad\quad x = x - 1;\ \} \\
\quad\ \ \text{res} = \text{ans};\ \} \\
\text{res} = x_0!\ \&\ \sigma_0 \subseteq \sigma
\end{array}
$$

From here we proceed as with any total correctness formula. We begin by annotating the loop with an invariant and a decrementing function. The invariant will include the predicate $\sigma_0 \subseteq \sigma$; we shall need this to show that the post-condition holds on exiting the loop. Pushing the final predicate across the assignment statement that replaced the **return** statement gives us

$$
\begin{array}{l}
x \geq 0 \ \&\ x_0 = x \ \&\ \sigma_0 = \sigma \\
\quad \{\ \text{ans} = 1; \\
\quad\ \ \textbf{while}\ (x > 0)\ \{ \\
\qquad\quad //\ \text{invariant ans} = (x_0!/x!)\ \&\ x \geq 0\ \&\ \sigma_0 \subseteq \sigma \\
\qquad\quad //\ \text{decrements x, range a nonnegative integer} \\
\qquad\quad \text{ans} = \text{ans} * x; \\
\qquad\quad x = x - 1;\ \}\ \} \\
\text{ans} = x_0!\ \&\ \sigma_0 \subseteq \sigma
\end{array}
$$

We can apply the loop invariant theorem to transform the above into four verification conditions:

1. The loop invariant holds on entry to the loop:

$$
\begin{array}{l}
x \geq 0 \ \&\ x_0 = x \ \&\ \sigma_0 = \sigma \\
\quad \{\ \text{ans} = 1;\ \} \\
\text{ans} = x_0!/x!\ \&\ x \geq 0\ \&\ \sigma_0 \subseteq \sigma
\end{array}
$$

2. The loop invariant is maintained by the test:

$$
\begin{array}{l}
\text{ans} = (x_0!/x!)\ \&\ x \geq 0\ \&\ \sigma_0 \subseteq \sigma \\
\quad \{x > 0\} \\
\text{ans} = (x_0!/x!)\ \&\ x \geq 0\ \&\ \sigma_0 \subseteq \sigma
\end{array}
$$

3. The loop invariant is maintained by the body of the loop when the test is true:

$$
x > 0\ \&\ \text{ans} = (x_0!/x!)\ \&\ x \geq 0\ \&\ \sigma_0 \subseteq \sigma
$$

$$\begin{array}{l} \quad \{ \text{ ans} = \text{ans} * \text{x}; \\ \quad\;\; \text{x} = \text{x} - 1; \} \\ \text{ans} = (\text{x}_0!/\text{x}!)\ \&\ \text{x} \geq 0\ \&\ \sigma_0 \subseteq \sigma \end{array}$$

4. The loop invariant and the negation of the test imply the post-condition:

$$\begin{array}{l} ((\text{ans} = (\text{x}_0!/\text{x}!)\ \&\ \text{x} \geq 0\ \&\ \sigma_0 \subseteq \sigma)\ \&\ \neg(\text{x} > 0)) \Rightarrow \\ \quad (\text{ans} = \text{x}_0!\ \&\ \sigma_0 \subseteq \sigma) \end{array}$$

Consider verification condition 1. Since the evaluation of the expression on the right-hand side of the assignment involves no side effects, we apply the assignment rule to yield

$$\begin{array}{l} (\text{x} \geq 0\ \&\ \text{x}_0 = \text{x}\ \&\ \sigma_0 = \sigma) \Rightarrow \\ \quad (1 = \text{x}_0!/\text{x}!\ \&\ \text{x} \geq 0\ \&\ \sigma_0 \subseteq \sigma) \end{array}$$

Since $x \geq 0$ occurs on both sides of the implication, and since $\sigma_0 = \sigma$ implies $\sigma_0 \subseteq \sigma$, we can reduce this to

$$(\text{x} \geq 0\ \&\ \text{x}_0 = \text{x}) \Rightarrow (1 = \text{x}_0!/\text{x}!)$$

Since $x_0 = x$ occurs on the left of the implication, we can reduce the right side to

$$\text{x} \geq 0 \Rightarrow 1 = (\text{x}!/\text{x}!)$$

that is, to TRUE.

We discharge the second verification condition by observing that the test $x > 0$ is side-effect-free. Using the assignment rule twice, we simplify verification condition 3 to

$$\begin{array}{l} (\text{x} > 0\ \&\ \text{ans} = (\text{x}_0!/\text{x}!)\ \&\ \text{x} \geq 0\ \&\ \sigma_0 \subseteq \sigma) \Rightarrow \\ \quad (\text{ans} * \text{x} = (\text{x}_0!/(\text{x} - 1)!)\ \&\ (\text{x} - 1) \geq 0\ \&\ \sigma_0 \subseteq \sigma) \end{array}$$

which, deleting $\sigma_0 = \sigma$ from both sides, reduces to

$$\begin{array}{l} (\text{x} > 0\ \&\ \text{ans} = (\text{x}_0!/\text{x}!)\ \&\ \text{x} \geq 0) \Rightarrow \\ \quad (\text{ans} * \text{x} = (\text{x}_0!/(\text{x} - 1)!)\ \&\ (\text{x} - 1) \geq 0) \end{array}$$

Since $x > 0$ implies $(x - 1 \geq 0)$, we can reduce this to

$$(\text{x} > 0\ \&\ \text{ans} = (\text{x}_0!/\text{x}!)) \Rightarrow (\text{ans} * \text{x} = (\text{x}_0!/(\text{x} - 1)!))$$

Since $ans = (x_0!/x!)$ occurs on the left of the implication, we can reduce this to

$$\text{x} > 0 \Rightarrow ((\text{x}_0!/\text{x}!) * \text{x} = \text{x}_0!/(\text{x} - 1)!)$$

which simplifies to

$$\mathrm{x} > 0 \Rightarrow (((\mathrm{x}_0! * \mathrm{x})/\mathrm{x}!) = \mathrm{x}_0!/(\mathrm{x}-1)!)$$

which, using the fact that $v > 0 \Rightarrow v! = v * (v - 1!)$, simplifies to TRUE. Finally, consider verification condition 4. Since $(x \geq 0 \ \& \ \neg(x > 0))$, we know that $x = 0$. The condition can therefore be reduced to

$$(\mathrm{ans} = \mathrm{x}_0!/0!) \Rightarrow \mathrm{ans} = \mathrm{x}_0!$$

Since $0! = 1$, this reduces to TRUE.

Suppose we had omitted the $x \geq 0$ part of the invariant. We would then have been able to simplify condition 4 to

$$(\mathrm{ans} = \mathrm{x}_0!/\mathrm{x}! \ \& \ \neg(\mathrm{x} > 0)) \Rightarrow \mathrm{ans} = \mathrm{x}_0!$$

At this point we would have concluded that this reduces to TRUE only when $x! = 1$, that is, when $x = 0$ or $x = 1$. We know that x is not equal to 1, since $\neg(x > 0)$. However, since we cannot deduce $x = 0$ from the hypothesis of the verification condition, we would be led to conclude that our loop invariant is too weak. Presumably this would lead us to strengthen it appropriately. Having changed the invariant, we would need to generate and prove the new verification conditions for the loop.

That completes our proof of partial correctness. To prove total correctness we use the decrementing function to show that the loop terminates, as follows:

1. The body reduces the decrementing function:

$$\begin{array}{l}\mathrm{x} = \mathrm{d}_0 \ \& \ \mathrm{x} > 0 \ \& \ \mathrm{ans} = \mathrm{x}_0!/\mathrm{x}! \ \& \ \mathrm{x} \geq 0 \ \& \ \sigma_0 \subseteq \sigma \\ \quad \{ \ \mathrm{ans} = \mathrm{ans} * \mathrm{x}; \\ \quad\ \ \mathrm{x} = \mathrm{x} - 1; \ \} \\ \mathrm{x} < \mathrm{d}_0\end{array}$$

2. The decrementing function is not incremented by the test:

$$\mathrm{x} = \mathrm{d}_0 \quad \{\mathrm{x} > 0\} \quad \mathrm{x} \leq \mathrm{d}_0$$

3. The loop is exited when the decrementing function reaches 0:

$$(\mathrm{ans} = \mathrm{x}_0!/\mathrm{x}! \ \& \ \mathrm{x} \geq 0 \ \& \ \sigma_0 \subseteq \sigma \ \& \ \mathrm{x} = 0) \Rightarrow \neg(\mathrm{x} > 0)$$

Using the assignment rule twice, we can simplify verification condition 1 to $(x = d_0 \Rightarrow (x - 1) < d_0)$, that is, to TRUE. Since $x > 0$ is side-effect-free, condition 2 simplifies to TRUE. Condition 3 also simplifies immediately to TRUE.

This completes our proof that the implementation of *factorial* satisfies its specification. The only thing that distinguishes this proof from those earlier in the chapter is the introduction of σ, which allowed us to reason about the absence of mutation in the same way we reasoned about other properties. In practice, it is often simpler to reason separately about mutability, since it is frequently possible to use some sort of meta-argument to show that the body of a procedure cannot possibly mutate anything. For *factorial*, we could have argued that since its only argument is an int that is passed by value, the body cannot access any mutable objects and therefore must modify nothing.

We use only the specification to reason about procedure invocations, and such reasoning is independent of the proof that a procedure satisfies its specification. For example, to reason about a program that invokes *factorial*, we assume *factorial*'s specification—that its requires implies its effects. It is convenient to reason about *factorial* using two different rules, each of which is valid because of *factorial*'s specification. First, since *factorial* is side-effect-free, we use it in formulas as an expression, as in

$$\mathrm{x} = 3 \Rightarrow \mathrm{factorial(x)} = 6$$

Such an expression stands for the value returned by the *factorial* procedure, just as the expression $a + b$ stands for the value returned by the int `operator +` method. We can reason about this value using the rule

1. $\forall$ w: INT [$\mathrm{w} \geq 0 \Rightarrow (\mathrm{factorial(w)} = \mathrm{w!})$]

where ! stands for mathematical factorial on integers. In addition, we sometimes need to reason about aspects of *factorial*'s behavior that are not captured by the returned value. To do this we use the rule

2. $\mathrm{x} \geq 0$ & $\sigma_0 = \sigma$ {factorial(x)} $\sigma_0 \subset \sigma$

In this formula we treat *factorial*(x) as a command.

When we want to reason about a program fragment that invokes *factorial*, we can use these rules just as we use other components of our proof system, such as the assignment rule or the case analysis rule. (A slight technical problem arises if the first formula turns out to be always false. However, in practice this is not a serious problem, and we shall ignore it.) Consider a proof of the formula

x $\geq$ 0
  { **if** (factorial(x) == 6) c = TRUE; **else** c = FALSE; }
c = (x=3)

Since rule 2 for *factorial* tells us that it has no side effects, we begin by applying the case analysis rule. This yields two verification conditions:

4. $x \geq 0$ & $\text{factorial}(x) = 6 \ \{c = \text{TRUE};\}\ c = (x = 3)$
5. $x \geq 0$ & $\neg(\text{factorial}(x) = 6) \ \{c = \text{FALSE};\}\ c = (x = 3)$

We use *factorial*(x) here to stand for the value returned by the *factorial* procedure when it is called with x as an argument. Pushing across the assignment statement of the first verification condition, we get

$$x \geq 0 \ \& \ \text{factorial}(x) = 6 \Rightarrow ((x = 3) = \text{TRUE})$$

or

$$x \geq 0 \ \& \ \text{factorial}(x) = 6 \Rightarrow (x = 3)$$

Rule 1 for *factorial* tells us that

$$\forall\, w : \text{INT}[\ w \geq 0 \Rightarrow (\text{factorial}(w) = w!\ )\ ]$$

Since the hypothesis of the verification condition asserts that $x \geq 0$, we can use this rule to reduce the verification condition to

$$x! = 6 \Rightarrow x = 3$$

We then use the mathematical definition of ! to reduce this to TRUE. The second formula simplifies similarly.

14.3.2 Reasoning about Code that Uses Arrays

To reason about code that uses arrays requires that we provide some analog to what we have used to reason about integers. In this chapter we present a very limited way of doing such reasoning. In particular we consider the use of only two array methods, `fetch` and `length`. Furthermore we will consider only arrays containing integers.

The state of an array object can be thought of as a mathematical sequence. We will assume that a formal definition of sequences of integers (`INT_SEQ`) has been provided and that it defines two operations on sequences:

LENGTH(s)	gives the length of the sequence s
FETCH(s, i)	gives the ith element of the sequence s if $0 \leq i <$ LENGTH(s); undefined otherwise

```
int mina (array<int> * a) {
// requires: LENGTH(a) > 0
// modifies: nothing
// effects: returns res such that
//    ∃ j (0 ≤ j < LENGTH(a) & res = FETCH(a, j)) &
//    ∀ k [0 ≤ k < LENGTH(a) ⇒ res ≤ FETCH(a, k)]
  int m, i;
  m = a → fetch(0);
  i = 1;
  while (i < a → length()) {
    if (m > a → fetch(i)) m = a → fetch(i);
    i = i + 1; } } // end mina
```

Figure 14.2 Specification and Implementation of mina.

We do not discuss how such definitions are given in this chapter.

To allow us to reason about code that calls the `length` and `fetch` methods, we introduce rules similar to the rules for `factorial`. Here are the rules that allow us to reason about the values obtained by calling these two methods:

$$\forall \text{ a: INT\_SEQ } [\text{ a} \rightarrow \text{length( )} = \text{LENGTH(a)}]$$

$$\forall \text{ a: INT\_SEQ, i: INT } [\ 0 \leq \text{i} \leq \text{LENGTH(a)} \Rightarrow \text{a} \rightarrow \text{fetch(i)} = \text{FETCH(a,i)}\ ]$$

In addition we have the rules that allow us to deduce that calls on these methods have no side effects:

$$\sigma_0 = \sigma \ \{ \text{ a} \rightarrow \text{length( ) } \} \ \sigma_0 \subseteq \sigma$$

$$0 \leq \text{i} \leq \text{LENGTH(a)} \ \& \ \sigma_0 = \sigma \ \{ \text{ a} \rightarrow \text{fetch(a, i) } \} \ \sigma_0 \subseteq \sigma$$

Using these rules we can reason about code that uses arrays. For example consider the procedure to compute the minimum element of an array; the specification and implementation are given in Figure 14.2.

We begin by producing the pre- and post-conditions from the specification, and we also rewrite the code to obtain the verification condition:

```
LENGTH(a) > 0 & σ0 = σ
  { m = a → fetch(0);
    i = 1;
    while (i < a → length()) {
      if (m > a → fetch(i)) m = a → fetch(i);
```

$$
\begin{array}{l}
\quad\quad \mathsf{i = i + 1; \}} \\
\quad \mathsf{res = m; \}} \\
\exists\ \mathsf{j}\ (\ 0 \le \mathsf{j} < \mathsf{LENGTH(a)}\ \&\ \mathsf{res} = \mathsf{FETCH(a, j)})\ \& \\
\forall\ \mathsf{k}\ (0 \le \mathsf{k} < \mathsf{LENGTH(a)} \Rightarrow \mathsf{res} \le \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}
$$

Next we push back across the assignment that represents the return statement to obtain:

$$
\begin{array}{l}
\mathsf{LENGTH(a)} > 0\ \&\ \sigma_0 = \sigma \\
\quad \{\ \mathsf{m = a \rightarrow fetch(0);} \\
\quad\quad \mathsf{i = 1;} \\
\quad\quad \textbf{while}\ (\mathsf{i < a \rightarrow length()})\ \{ \\
\quad\quad\quad \textbf{if}\ (\mathsf{m > a \rightarrow fetch(i)})\ \mathsf{m = a \rightarrow fetch(i);} \\
\quad\quad\quad \mathsf{i = i + 1; \}\ \}} \\
\exists\ \mathsf{j}\ (\ 0 \le \mathsf{j} < \mathsf{LENGTH(a)}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\ \mathsf{k}\ (0 \le \mathsf{k} < \mathsf{LENGTH(a)} \Rightarrow \mathsf{m} \le \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}
$$

Now we can proceed in the usual way. As before we begin by inventing the loop invariant and decrementing function. A good way to come up with the loop invariant is to think about modifying the post condition so that it represents what has happened so far. In other words, the post condition states what must be true at the end, and each iteration gets us closer to the final result; the loop invariant should capture this progress. In this case, we want `m` to be the minimum when we have processed the entire array; we achieve this by having it be the minimum of the part of the array `a` that we have examined so far. This gives us the following invariant:

$$
\begin{array}{l}
0 \le \mathsf{i} \le \mathsf{LENGTH(a)}\ \& \\
\exists\ \mathsf{j}\ (\ 0 \le \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\ \mathsf{k}\ (0 \le \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \le \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}
$$

The first clause is needed to keep track of the constraints on `i`; for example, we will use it to satisfy the precondition of the call to `fetch` inside the loop body.

Now we can generate the four verification conditions:

1. The loop invariant holds on entry to the loop:

$$
\begin{array}{l}
\mathsf{LENGTH(a)} > 0\ \&\ \sigma_0 = \sigma \\
\quad \{\mathsf{m = a \rightarrow fetch(0);} \\
\quad\quad \mathsf{i = 1;\}} \\
0 \le \mathsf{i} \le \mathsf{LENGTH(a)}\ \&
\end{array}
$$

$$\begin{array}{l}
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}$$

2. The loop invariant is maintained by the test:

$$\begin{array}{l}
0 \leq \mathsf{i} \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma \\
\quad \{\ (\mathsf{i} < \mathsf{a} \rightarrow \mathsf{length(\ )})\ \} \\
0 \leq \mathsf{i} \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}$$

3. The loop invariant is maintained by the body of the loop when the test is true:

$$\begin{array}{l}
\mathsf{i} < \mathsf{LENGTH(a)}\ \& \\
0 \leq \mathsf{i} \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma \\
\quad \{\ \textbf{if}\ (\mathsf{m} > \mathsf{a} \rightarrow \mathsf{fetch(i)})\ \mathsf{m} = \mathsf{a} \rightarrow \mathsf{fetch(i)}; \\
\qquad \mathsf{i} = \mathsf{i} + 1;\ \} \\
0 \leq \mathsf{i} \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}$$

4. The loop invariant and the negation of the test imply the post-condition:

$$\begin{array}{l}
\neg\ (\mathsf{i} < \mathsf{LENGTH(a)})\ \& \\
0 \leq \mathsf{i} \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{i}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{i} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma \\
\quad \Rightarrow \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < \mathsf{LENGTH(a)}\ \&\ \mathsf{m} = \mathsf{FETCH(a, j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < \mathsf{LENGTH(a)} \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a, k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}$$

Consider the proof of the first verification condition. Pushing back over the second assignment statement give us

$$
\begin{array}{l}
0 \leq 1 \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < 1\ \&\ \mathsf{m} = \mathsf{FETCH(a,\ j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < 1 \Rightarrow \mathsf{m} \leq \mathsf{FETCH(a,\ k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}
$$

Then we push back over the first assignment statement, using the rules for `fetch` to obtain

$$
\begin{array}{l}
0 \leq 1 \leq \mathsf{LENGTH(a)}\ \& \\
\exists\, \mathsf{j}\ (\ 0 \leq \mathsf{j} < 1\ \&\ \mathsf{FETCH(a,\ 0)} = \mathsf{FETCH(a,\ j)})\ \& \\
\forall\, \mathsf{k}\ (0 \leq \mathsf{k} < 1 \Rightarrow \mathsf{FETCH(a,\ 0)} \leq \mathsf{FETCH(a,\ k)})\ \& \\
\sigma_0 \subseteq \sigma
\end{array}
$$

which follows from the precondition.

The next verification condition is trivial since the test is side-effect-free. Here we use the second rule for the `length` method.

The third verification condition requires us to deal with an `if` statement that doesn't have an `else` clause; we will handle this by simply assuming the clause is there with an empty body. Informally, the condition holds because (1) we know that `0 <= i < LENGTH(a)` at the start of the loop, and therefore at the end we must have `0 <= i <= LENGTH(a)`; (2) `m` was the minimum between 0 and `i` at the start of the loop, and also at the end because we update `m` if the `i`th element is smaller; and (3) the array isn't modified.

Finally the fourth verification condition follows from simple substitution.

Now we need to show that the loop terminates. The decrementing function is:

$$\mathsf{LENGTH(a)}{-}\mathsf{i},\ \mathsf{range\ a\ non{-}negative\ integer}$$

We must show:

1. The loop body reduces the decrementing function:

$$
\begin{array}{l}
(\mathsf{LENGTH(a)} - \mathsf{i}) = \mathsf{d}_0\ \&\ \mathsf{i} < \mathsf{LENGTH(a)}\ \&\ \mathsf{I} \\
\quad \{\textbf{if}\ (\mathsf{m} > \mathsf{a} \rightarrow \mathsf{fetch(i)})\ \mathsf{m} = \mathsf{a} \rightarrow \mathsf{fetch(i)}; \\
\quad\ \ \mathsf{i} = \mathsf{i} - 1;\ \} \\
(\mathsf{LENGTH(a)} - \mathsf{i}) < \mathsf{d}_0
\end{array}
$$

2. The decrementing function is not incremented by the test:

$$(\mathsf{LENGTH(a)} - \mathsf{i}) = \mathsf{d}_0\ \{\mathsf{i} < \mathsf{a} \rightarrow \mathsf{length(\ )}\}\ \mathsf{LENGTH(a)} - \mathsf{i}) < \mathsf{d}_0$$

3. The loop terminates when the decrementing function reaches 0:

$$\mathsf{I}\ \&\ (\mathsf{LENGTH(a)} - \mathsf{i}) = 0 \Rightarrow \neg\ (\mathsf{i} < \mathsf{LENGTH(a)})$$

The first condition follows because incrementing `i` when it was below `LENGTH(a)` causes it to become closer to `LENGTH(a)` and therefore we reduce the value of the decrementing function. The second condition follows because the loop predicate is side-effect free; the third simplifies immediately to `TRUE`.

14.4 A Few Remarks about Formal Reasoning

From the foregoing it should be obvious that formal reasoning about programs involves a large amount of tedious symbol manipulation. This is not a process that people usually enjoy or do very well. Clearly this is a perfect job for a machine. People do need to provide

1. a specification of the program to be verified,
2. a putative implementation of that program,
3. specifications of the procedures and types used in the implementation,
4. a loop invariant and decrementing function for each loop, and
5. a rep invariant and abstraction function for each cluster.

It is important not to think of this information as being generated during an ex post facto program verification phase. Our whole approach to programming is based on developing specifications during program design. Invariants, decrementing functions, and abstraction functions should all be included as part of a program's documentation.

Given the above information, the second stage of program verification—the construction of verification conditions—is algorithmic and therefore best left entirely to a program. Much of the symbol manipulation done so laboriously in this chapter involved the generation of verification conditions.

The final stage of the verification—the simplification of verification conditions—can be partially automated. Almost all of the simplification done in this chapter can be done by theorem-proving programs. In general, however, we shall not always be so fortunate. A program verifier can fail to prove that a program satisfies its specification under a number of conditions:

1. The deductive power of the simplifier is insufficient to allow it to derive a valid predicate as a theorem. There are a number of interesting domains for which it is impossible to write theorem provers that can answer all relevant questions.

2. The program being verified satisfies its specification, but the specifications of internal procedures are too weak to allow this to be shown.

3. The program being verified satisfies its specification, but the loop invariants, decrementing functions, rep invariants, or abstraction functions are inappropriate.

4. The correctness of the program depends upon some fact about the application area. Ideally, one should incorporate such information into the specification or other documentation.

5. The program being specified does not satisfy its specification. The program, the specification, or both may be "wrong."

The first of these causes of failure is a serious problem in program verification. It can generally be overcome by guiding the verifier, giving it appropriate lemmas to prove. Unfortunately, deducing the lemmas that need to be supplied is often difficult if one is unfamiliar with the theorem-proving technology used by the verifier. The second, third, and fourth causes often represent failures in the design or documentation of the program. Discovering such failures is an important benefit of program verification. The fifth cause represents a failure in either the specification or the implementation of the program. There is no a priori way to know whether it is the program or the specification that is "wrong." Uncovering such problems is the primary purpose of program verification.

14.5 Summary

Throughout this chapter we treated programs as text and reasoned about them as such. We associate with a programming language a set of proof rules that allow us to reduce a total correctness formula of the form

Pre-condition {program text} Post-condition

to a predicate whose meaning is independent of the programming language. In order to make this possible, we embed various kinds of auxiliary information in the program text.

To deal with loops, we introduce loop invariants and decrementing functions. Decrementing functions are used to prove termination. Loop invariants reduce reasoning about the arbitrarily large number of paths through a program containing a loop to reasoning about a relatively small number of paths.

In dealing with procedures, we rely heavily upon specifications. Although procedures have complicated proof rules (simplified versions of these rules were presented here), they play a vital role in modularizing reasoning about programs. The key point is that reasoning about the implementation of a procedure is independent of reasoning about its invocations.

The usefulness of loop invariants, decrementing functions, rep invariants, and abstraction functions is not limited to formal verification. They are useful to the implementer since they are statements of assumptions on which the implementation depends. They are useful during program testing as a source of test cases, and during debugging as evidence of errors. They also provide useful program comments for those who must try to understand the program later.

Understanding a program means reasoning informally about its meaning. Informal reasoning is like formal reasoning; it follows the same steps and uses the same information. Thus it is no surprise that concepts developed for formal reasoning are useful for understanding programs. Perhaps the most important outcome of work on formal verification has been the identification and definition of these concepts.

Further Reading

Gries, David, 1981. *The Science of Programming.* New York: Springer-Verlag.

Hoare, C. A. R., 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12(10): 576–583. Reprinted in *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*, edited by David Gries (New York: Springer-Verlag, 1978).

Jones, Cliff B., 1980. *Software Development, A Rigorous Approach.* Englewood Cliffs, N.J.: Prentice-Hall International.

London, Ralph L., Mary Shaw, and William A. Wulf, 1981. Abstraction and verification in Alphard: A symbol table example. In *Alphard: Form and Content*, edited by Mary Shaw (New York: Springer-Verlag), pp. 161–190.

Exercises

14.1 Specify a procedure, *max_elem*, that accepts a nonempty array of integers as an argument and returns the largest element in the array. Specify a procedure, *max_array*, that accepts two nonempty arrays of integers as arguments and returns the array that contains the largest element. Implement *max_array* using *max_elem* and prove that your implementation satisfies the specification. (Note: You need not implement *max_elem*.)

14.2 Discuss the relationship between program verification and the material on test data selection presented in chapter 8.

Chapter 15

Extending our Approach

This book has presented a methodology for program construction using a restricted subset of C++. We constrained the use of C++ in two main ways: we required that objects reside in the heap, and and we assumed the existence of a garbage collector. This chapter discusses how to relax these constraints.

15.1 Stack vs. Heap

So far all the data abstractions you have defined were intended to have objects in the heap, and furthermore all programs using these types were required to place their objects in the heap. However, it is also possible to define types whose objects are "on the stack". In this section we discuss advantages and disadvantages of placing objects in the stack or the heap.

When we discuss objects on the stack, what we mean are objects stored directly in variables. This is in contrast to "heap objects", which do *not* reside in variables; instead the variables point to the objects. In either case the variables might reside in the stack, or they might be contained in objects, e.g., they are instance variables. In the latter case the storage for the variables might actually be on the heap (because the containing object is in the heap). However, in the case of stack objects, we won't explicitly allocate heap space for them, but rather the space that contains them will already have been created (when the declaration of the containing variable was executed). In the case of heap objects, explicit allocation of space for the object in the heap is required.

Stack objects are attractive for two main reasons. The first has to do with storage allocation. As mentioned, stack objects are created when the declaration of the associated variable is executed. The execution occurs within a certain *scope* within a program (e.g., the body of a procedure or a loop), or when an object is created via a call to `new`. In the case of a scope, space to store the variables of the scope is provided on the program stack, which grows when the scope is entered and shrinks when control exits from the scope. In the case of instance variables, space to store them is provided when `new` is called, and deallocated when the containing object is deallocated. In either case, space for stack objects is deallocated automatically as part of deallocating the containing storage.

Because we provide a garbage collector, you haven't had to be concerned with deallocating storage for heap objects. When there is no garbage collector, which is the situation for most C++ programs, the question of when to deallocate storage for heap objects is a major issue in writing code; we will discuss this issue in Section 15.2. Not having to worry about deallocation is therefore attractive.

Furthermore, even when there is a garbage collector, stack allocation is still attractive because it is cheaper. Deallocation of heap objects requires work. The programmer will need to delete the heap object explicitly, and furthermore, the runtime system will need to record the deletion, e.g., entering the freed space in a free list so that it can be used again later. By contrast deallocation of stack objects requires no extra work (in addition to whatever was needed to deallocate the containing variable); furthermore, when that variable is itself on the stack, its storage goes away when control exits the containing scope, just by shrinking the stack.

The second reason why stack objects are attractive is that the operator syntax works conveniently for them. For example, consider the following code:

```
int x = 3;
int *p = new(3);
x = x + x; // call on operator +
p = *p + *p; // call on operator +
```

Because `x` is a stack variable, we can call the `operator +` method on it very conveniently. Using this method for `p` is less convenient; because it is a pointer to an `int` we need to dereference it before using the method.

However, there are also a number of reasons that stack objects are unattractive. First, stack storage is less general than heap storage because once space has been allocated for an object on the stack, it isn't possible to change the allocation. Therefore dynamic objects (whose size changes during program execution) cannot be allocated on the stack. Such objects can easily be accommodated on the heap; for example, in the case of a linked list we just allocate a new object to hold the new list member and link it onto the chain of objects on the list.

Second, C++ additionally requires that the size of stack objects be known at compile time. The reason is that the compiler determines how much space to allocate for the containing variable (on the stack or in the containing object in the case of an instance variable); obviously it couldn't do this determination if the size weren't compile-time known. (Some languages allow the size of stack objects to be determined at creation time, i.e., when the declaration of the containing variable is executed, but C++ does not support this.)

Therefore only objects of fixed, compile-time known size can be allocated on the stack. Additionally, these objects should be small. The reason is that stack objects are frequently copied. This is what happens in an assignment, and also when they are passed by value. Copying is expensive when objects are large. (A copy *must* have the abstract effect of providing a disjoint copy, without any sharing with the object being copied that is visible to using code. If the object being copied is mutable there is no choice but to copy every piece of it. If the object is immutable, there are ways to avoid the copy, but this is not easy in the absence of a garbage collector. This issue is discussed further in Section 15.2.2.)

A fourth point concerns sharing. Mutable objects are frequently shared, since this is needed to ensure that modifications are visible to all users. If a stack object is mutable, the only way to share it is to have pointers to its location on the stack. Such pointers are no problem when the sharing happens because the object is passed by reference to some called routine. They are problematical, however, when we want the object to be pointed at from some other object since care is needed to ensure that the object being pointed at exists at least as long as the pointer. Objects that are going to be pointed at from other objects *ought* to be placed in the heap.

A related point concerns the use of routines that create new objects of the type. For example, consider a type `intset100` whose objects are integer sets containing a maximum of 100 elements. These objects could be stack objects (e.g., their rep might contain exactly enough room for the 100 elements). Now consider the following signatures for procedure `make_set`, which creates and returns a new `intset100` object:

```
intset100 * make_set (...);
intset100 make_set (...);
void make_set (intset100 &s, ...);
```

In the first definition, `make_set` is creating the new `intset100` on the heap and returning a pointer to it; this is a suitable definition if `intset100`s are on the heap but not if they are on the stack. The other two definitions are intended for use when `intset100`s are on the stack. In the second definition, the new `intset100` is copied into a variable in the caller's scope when the call on `make_set` returns. The problem here is that this copy is expensive. The third definition avoids the expense by providing a by-reference argument `s`; now `make_set` can store the new `intset100` directly in s. Note that make_set cannot return a intset100 & to an object created in its stack frame since that storage will disappear when it returns.

A final point concerns subtyping. Any variables that are going to be assigned objects of a subtype should be pointers. If these variables instead contain their object directly, program execution is unlikely to do what you expect. For example, suppose that extended_intset is a subtype of intset, and consider the following:

```
intset *s;
intset s2;
extended_intset *e = new extended_intset( );
extended_intset e2 = extended_intset( );
s = e;
s2 = e2;
```

In the first assignment, s points to an extended_intset object, and calls on methods of s go the implementation provided for extended_intset. In the second assignment, the values of the inherited instance variables of e2 are copied into their counterparts in s2; all other instance variables of e2 are ignored. Furthermore, when methods are called on s2 they go to the implementation provided for intset.

15.1.1 Defining Data Abstractions

Although using code ultimately determines where objects reside, definers of data abstractions get to decide whether stack storage is convenient or not. Usually if there are objects on the stack, programs will need to assign them and to pass them by value. The former requires the use of the operator = method; the later requires the use of the copy constructor. In addition, if the new type has any methods that take arguments of the type or return results of the type, the signatures of these methods will be biased to either heap or stack objects. If the definer of the type expects objects to be on the heap, the methods will take and return T*'s; if instead objects are expected to be on the stack, the methods will take arguments of either type T or T&, and will either return results of type T, or take an extra by-reference T& argument.

Therefore, when you define a new type, you need to decide where you expect its objects to reside. A good rule is the following: make the new type be a heap abstraction (i.e., convenient to use when its objects are on the heap) *unless* its objects are immutable, small, and have a compile-time known size. An additional motivation for stack types is allowing the use of operators. For example a numeric type will be very convenient to use if the operators are available.

Rational numbers are a plausible stack abstraction. They are immutable and small and have a predefined size, e.g., two fields, to record the numerator and denominator. Furthermore, they have many associated arithmetic operators.

Figure 15.1 gives the specification for rationals. The overview section states explicitly that a stack abstraction is being defined. The methods and operators (e.g., `operator +`) that take rationals as arguments or return them as results are defined in a way that is consistent with stack objects. Results are returned by value; arguments are taken by reference to avoid needless copying.

Note that all the operators shown (and presumably the ones that aren't shown) have meanings that are consistent with their symbol. For example, `operator +` does addition. It is very important to obey this rule since otherwise programs that use objects of the type will be difficult to understand.

Even though conceptually rational numbers are immutable, it appears that they can be mutated by `operator =`. However, really this is not a mutation (even though C++ views it as a modification to `this`); rather it is changing the object that a particular variable denotes. This interpretation is illustrated by the following code:

```
intset *s, *t;
rational r1, r2;
...
s = t;  // s now denotes the same object as t
r1 = r2; // r1 now denotes a rational whose value is the same as r2
```

Therefore we will consider a stack abstraction to be immutable if the only method that modifies `this` is `operator =`. (Recall that constructors don't mutate their objects because the object isn't visible to using code until after the constructor returns.)

The rational number abstraction provides a definition of the default constructor. It is not necessary for stack abstractions to provide this constructor, any more than heap abstractions need to provide it. In either case, if the constructor is *not* provided, it must be disallowed by including its header in the private section of the class declaration.

15.1.2 Stack-or-Heap Abstractions

Sometimes it is appropriate to provide an abstraction whose objects are convenient to use either on the stack or on the heap. This is what we did for 6.170 arrays. There were two reasons for this. When arrays are on the stack, the bracket notation (`[ ]`) can be used to fetch and store their elements, which is convenient. Additionally, arrays are often used in reps,

```
class rational {
public:
// overview: rational numbers. An immutable stack abstraction.

// constructors:
  rational(int n, int d) throw (CE_Zero);
    // effects: if d = 0 then throws CE_Zero else
    //   initializes this to be the rational n/d.

  rational(rational const & r) throw ( );
    // effects: initializes this to be the same rational as r

  rational ( ) throw ( );
    // effects: initializes this to be the zero rational

// methods
  int num( ) const throw ( );
    // effects: returns the numerator of this (in reduced form).

  int denom( ) const throw ( );
    // effects: returns the denominator of this (in reduced form).

  void operator = (rational const & r) throw ( );
    // modifies:   this
    // effects: makes this be the same rational as r

  rational operator + (rational const & r) const throw ( );
    // effects: returns the sum of this and r

  rational operator * (rational const & r) const throw ( );
    // effects: returns the product of this and r

  bool operator == (rational const & r) const throw ( );
    // effects: returns true if this = r else returns false.

  // other rational methods

private:
  int numer; // the numerator
  int denomin; // the denominator
}; // end rational
```

Figure 15.1 Specification of the rational number abstraction.

```
// overview: arrays can be used on either the stack or the heap.

// additional constructor
  array<T> (array<T> const &a) throw ( );
    // requires: T has a default constructor and operator =
    // effects: changes this to contain the same elements as a
    //      in the same order.

// additional methods
  T operator [ ] (int i) const throw (CE_Bounds);
    // effects:    if i is in bounds, returns the ith element of this
    //   else throws CE_Bounds

  T & operator [ ] (int i) throw (CE_Bounds);
    // modifies: this
    // effects: if i is in bounds, returns a reference to the ith location
    //   in this else throws CE_Bounds.

  void operator = (array<T> const & a);
    // requires:     T has operator =
    // modifies:   this
    // effects: changes this to contain the same elements as a
    //      in the same order.
```

Figure 15.2 Additional array methods.

and storing them directly in instance variables can avoid an extra level of indirection and thus improve performance.

When an type is defined to be used on either the stack or heap, it will have `operator =` and the copy constructor. Furthermore, if it has any methods that take arguments of the type or return results of the type, it will probably have two versions of the method, one for use with stack objects and one for use with heap objects. This is what happens with arrays. We have already seen the `fetch` and `store` methods, which are intended for use with heap objects; for stack objects we provide `operator [ ]`. Figure 15.2 provides the specifications of the additional array operations.

The first definition of `operator [ ]` will be called whenever the object is a `const` array. The definition indicates that the operator returns a value of type `T`. Alternatively we could have defined the operator to return a `T const &`, i.e., a location that holds a `T`, but that cannot be assigned to. The latter form is useful if `T` is large, since it allows the location to be passed by reference (to a `T const &` argument); it does not allow an assignment to the location (unless the caller does a cast!).

The second definition of `operator [ ]` will be called whenever the array object is not `const`. It returns a reference to the ith location as a `T &`, meaning that the caller can assign to it. Note that the method does *not* modify its object. However, it enables the caller to do modifications; this is why we consider it a mutator.

These distinctions are illustrated by the following code:

```
array<int> a;
array<int> const b;
x = b[i]; // calls first definition of operator [ ]
b[i] = x; // illegal at compile time
x = a[i]; // calls second definition of operator [ ]
a[i] = x; // calls second definition of operator [ ]
```

Users need to be careful in their use of pointers to locations within objects. No problems result if the location is used immediately as the target of an assignment. But consider the following use

```
void p (int & x, array<int> & z);
```

and suppose that `p` modifies array `z` in such a way that its size changes. The rep of an array `a` contains a pointer to a C++ array of a certain size; if `a` grows bigger than this size, a new C++ array is allocated, the elements of `a` are copied into it, the old C++ array is discarded, and the rep of `a` is changed to point to the new C++ array. Therefore when `p` causes the size of `z` to change, the C++ array contained within `z` might be deallocated. Since `x` points to a location within this array, the deallocation will cause `x` to become a dangling pointer.

The specifications of the copy constructor and `operator =` indicate that these calls do a complete copy of their argument, which will be expensive if the array is large. However the copy is necessary; the alternative of just copying the pointer to the C++ array that holds the elements is incorrect since it would introduce hidden sharing.

The copy constructor, `operator =`, and the first definition of `operator [ ]` all make use of the either the copy constructor or `operator =` (or both) of parameter type `T`. As discussed in Chapter 6, we do not pass in routines for these operations (since the syntax for calling them is identical whether `T` is a pointer type or not). Instead we rely on the type checking in the C++ compiler at the time an instantiation of `array` is compiled to guarantee that `T` has the needed operations.

An example of the use of stack arrays is given in Figure 15.3. The figure shows how `intset` can be implemented with an instance variable that contains an array rather than pointing to one. This techniques avoids a level of indirection whenever the `els` component is accessed, and it allows use of the bracket notation for accessing elements of `els`. Problems with

```
// part of .h file
class intset {
private:
   array<int> els;  // elements of the set
   int sz; // size of the set
   int get_loc(int x) const;  // private method
} // end intset

// part of the .cc file
   intset::intset ( ) throw ( ) {
      sz = 0; } // end

   intset::insert (int x) throw ( ) {
      int loc = get_loc(x);
      if (loc >= 0) return;
      els.append(x);
      sz = sz+1; } // end insert

   int intset::get_loc (int x) const throw ( ) {
      for (int i = 0; i < els.length( ); i++)
         if (els[i] = x) return (i);
      return (-1); } // end get_loc
```

Figure 15.3 Part of implementation of intset.

the expense of copying do not arise since `intset` doesn't provide a copy operation; furthermore, if `intset` had such an operation, its implementation would have to do a complete copy of the `els` array, whether it were in the heap or not.

One point to note about this implementation is the following. Suppose that your program library contained a `search` routine with the following signature:

```
int search (array<int> const * a, int x) throw ( );
```

This routine could be used in place of `get_loc` inside the implementation of `intset` but it isn't very convenient to use (because its header is biased toward objects in the heap). In general it is a good idea to use each type in a standard way, i.e., always place its objects in the stack, or always place them in the heap. Otherwise you are likely to end up with helping routines that are mismatched for one of the placements.

15.2 Explicit Memory Management

In 6.170 we have provided a garbage collector. This frees the programmer from concerns about storage management. Instead, objects continue to exist as long as they are accessible from program variables.

In the absence of a garbage collector, programmers must deallocate storage explicitly. This raises two issues: how to know when to deallocate storage, and what should happen when storage is deallocated.

Actually it's difficult to know when to deallocate storage; this is why we provided a garbage collector for 6.170. Storage deallocation is a rich source of program errors. These errors take two forms, dangling references, and storage leaks.

A *dangling reference* occurs when storage is deallocated too early. In this case the deallocated object still needs to be used. However the pointer used to access it now points to nothing, i.e., it "dangles." When the access occurs, something strange will happen. Bugs of this sort tend to manifest themselves in obscure ways and can take a long time to find.

A *storage leak* occurs when storage isn't deallocated soon enough. In this case, space is tied up for an object that is no longer being used. The result is that storage "leaks", i.e., the amount of storage available to the program becomes less and less. This sort of bug shows up in the inability to allocate an object at some later time. Again it can be difficult to figure out the source of the error. In addition, a storage leak may very well not show up during testing, but instead occurs during production when much larger data sets are in use.

A garbage collector avoids the first kind of error entirely. It avoids some storage leaks: if the object isn't accessible, its storage disappears. But some storage leaks can remain even when there is a collector, since the path to an object might remain even though it is never followed. Therefore, one must remember to remove paths to objects in either case.

However, when there is no garbage collector, one must also remember to deallocate objects *before* the last pointer to them disappears, and it can be difficult to know when this is. There may be many ways to get to an object while it was in use. Some of these are direct references from program variables. Others are indirect references through data structures. Knowing when all the references are gone can be difficult; it might even require computation. For example, a routine `q` may have destroyed a pointer to an object `x`, and later in some other scope another pointer to `x` is about to disappear. It may be necessary to look at some other variable, e.g., one recording a count of references to `x`, in order to know whether to explicitly deallocate `x`.

Here are some rules of thumb for doing explicit deallocation correctly: The basic rule is to think about deallocation whenever pointers disappear. Such a pointer provides access to some heap object; the object should be deallocated when the last pointer to it goes away. Note that failure to deallocate the object at this point will result in a storage leak, since after this point there will be no way to name the object, and therefore no way to deallocate it.

Pointers to objects go away in two ways: by assignment (which overwrites a pointer) and by leaving a scope. At each such point in your program, you should think about whether to do an explicit deallocation. If the pointer refers to a local object, i.e., one that was created in the current scope (or in some contained scope or by some routine that was called) and that is used only in that scope, it is relatively easy to figure out what to do since at that point you know all about how the object is being used. Much harder is figuring out what to do when the object is used in many parts of the program. In this case, knowing about later use requires global knowledge about the program. The right way to tackle this problem is at the specification level: the specification of a routine should address the question of later use for all global objects used by the routine. In other words, to do explicit deallocation right requires adding information to specifications, and sometime also adding runtime structures to record the status of paths to particular global objects.

For example, suppose you have a long-running procedure with the following form:

```
while (true) {
  // create a heap object and store a pointer to it in table T
  // do stuff
  p(T, ...)
}
```

The idea here is that `p` finds some object in table `T` according to some filter, uses the object and then deletes it from the table; the question is whether the object should be deallocated at that point. The best structure is for the long-running procedure to do this. In this case, `p` must provide information that can be used by the caller to delete the object, e.g., the call above becomes:

```
foo *x = p(t, ...); // here x is the removed object
delete x;
```

The alternative is to include the fact that the object is deleted in the specification of `p`, e.g.:

```
void p (table *t, ...);
  // modifies: t, the heap
  // effects: ... and deletes the object and its entry in t
```

This approach is less desirable because `p`'s caller may still be using the object.

15.2.1 Explicit Deallocation

Deallocating the storage of an object is done by using the `delete` primitive. For example, consider the following code:

```
poly * p;
// use p in here
delete p; // deallocates the storage for p
```

Your code should include calls on `delete` whenever you have determined that the last pointer to an object is disappearing as discussed above.

However, when a pointer to an abstract object disappears, it is essential to deallocate *all* the storage of the object, or else there will be a storage leak. For example, `poly` contains a pointer to an `array`. When a `poly` is deallocated, the `array` must also be deallocated. Note that `C++` can't do the right thing automatically, since it cannot distinguish the case of a contained pointer that refers to storage that will never be used again from the case of a contained pointer that refers to storage that is also referred to in some other way. Therefore user intervention is required.

However, the user intervention must not break the abstraction barrier. A user of `poly` doesn't know about the `poly` rep (and even if the user knew about the rep, the using code can't access the instance variables). Therefore the user cannot be expected to deallocate the array pointed at by a `poly`.

The way to handle this is to add a *destructor* to the data abstraction. When an object is deleted in `C++`, its destructor is called. The job of the destructor is to deallocate any storage pointed at by its object.

Destructors get called in two different ways, corresponding to heap and stack objects. In the case of heap objects, there must be an explicit call on `delete`. In the case of stack objects, the object disappears implicitly when control leaves its scope or when the object that contains it (within an instance variable) is deleted; the call on the destructor happens at this point.

Figure 15.4 gives the specification and implementation of the destructor for `poly`. The specification does not give a description of what the destructor does since this is generic: a destructor always gets rid of the storage for its object. Note that the implementation of the destructor explicitly deletes the `trms` array.

If no specification is given for a destructor in the public section of a class, `C++` will provide a default implementation that implicitly calls the destructor for each instance variable. If one of the instance variables is a pointer, however, the destructor does *not* delete the object that pointer

```
// the .h file
class poly {
public: // as before plus
  ~poly( ); // the destructor

private:
  int deg; // the degree
  array<int> * trms; // the coefficients
}; // end poly

// the .cc file
  poly::~poly ( ) {
    delete trms; } // end destructor
```

Figure 15.4 Example of a destructor.

refers to. Therefore if some of your instance variables are pointers, you will need to provide your own code for the destructor. Thus it is necessary to provide a destructor for `poly`, but not for the implementation of `intset` given in Figure 15.3; in the case of `intset`, the default implementation of the destructor will call the destructor for the `els` array, causing its contained `C++` array to be deallocated.

15.2.2 Implementing Copying

In the presence of a garbage collector, a copy operation on an immutable heap object can be implemented by just returning a pointer to that object. Since the object is immutable, the sharing introduced by this approach will not be visible to any using code. Furthermore, since there is a garbage collector, the using code will not have to keep track of the sharing (to avoid deleting the object too early). However, in the absence of a garbage collector, the sharing can be troublesome, since the using code must keep track of the sharing.

For example, consider the following `poly` method:

```
poly * poly::copy ( ) throw ( ) {
  // effects: returns a new poly with the same degree and
  //    coefficients as this
  return this;
  } // end copy
```

This code takes advantage of the fact that `polys` are immutable to avoid creating separate storage for the returned `poly`, which can result in con-

```
// the .h file
class poly {
private:
  int deg; // the degree
  struct {int cnt; array<int> * trms} *els;
}; // end poly

// the .cc file
  poly * poly::copy ( ) throw ( ) {
    poly * r = new poly ( );
    r -> deg = deg;
    r -> els = els;
    els -> cnt = els -> cnt + 1;
    return r; } // end copy

  poly::~poly ( ) {
    if (els -> cnt == 1) delete els;
    else els -> cnt = els -> cnt - 1; } // end destructor
```

Figure 15.5 Hidden Sharing.

siderable storage saving if the `poly` is large. However now consider the following code:

```
poly *p;
// use p
poly *q = p -> copy( );
// use p and q
delete q; // q is no longer needed
```

The problem is that deleting `q` causes `p`'s object to be deleted, leading to a dangling reference.

It is possible to take advantage of shared storage in conjunction with immutable heap objects, but the implementation of the type needs to be more careful. Effectively, it needs to do reference counting: it needs to keep track of how many objects use the shared storage. Figure 15.5 shows how this might work for `poly`. Note that the reference counter needs to be kept on the heap along with the shared information; it would not work to instead make the counter be an instance variable!

15.3 Summary

Earlier chapters of this book made use of a restricted subset of C++. Two major restrictions were imposed: objects of user-defined data abstractions were constrained to reside in the heap, and a garbage collector was provided so that explicit memory management was not needed. Neither of these restrictions is necessary for use of the programming methodology. Therefore this chapter has explained how to use C++ in a way that is consistent with the methodology while removing the restrictions.

Definers of data abstractions are now free to define three different kinds of abstractions: those whose objects should be on the heap, those whose objects should be on the stack, and those whose objects can be allocated in either place. In all cases, what the definer chooses is only a suggestion. Users are free to place objects where convenient.

In the absence of a garbage collector, definers of data abstractions must define what it means to discard an object of the type, by providing an implementation of the destructor that discards *all* storage belonging to the object. Users are responsible for explicitly deallocating heap objects. As is always the case when explicit deallocation is in use, care must be taken delete objects at just the right moment, when the last pointer to the object is about to go away.

Further Reading

Meyers, S, 1992 *Effective C++: 50 Specific Ways to Improve Your Programs and Design*, Addison-Wesley.

Exercises

15.1 Discuss the relative advantages and disadvantages of stack-based and heap-based approaches to providing data abstraction.

Chapter 16

Using Other Languages

The methodology presented in this book is language independent. However, not all programming languages support all the kinds of abstractions we use, nor do they support abstractions of our exact form. When faced with such a situation, we do not abandon the methodology. Instead we bridge the gap between the methodology and the language in two ways:

1. Adapt the forms of abstractions to better fit the language.
2. Invent conventions for implementing abstractions in the language.

Here we illustrate how to use our methodology in conjunction with C. C is missing several features that we used in C++: the most important are exception handling, classes, and templates. The following sections discuss how to adapt to these differences.

16.1 Procedural Abstractions in C

C does not have call by reference and it does not have exceptions. The absence of call by reference is not a serious problem because C allows us to pass pointers, even to stack locations. Thus we might have:

```
bool p (int x, int *y);
```

Here the second argument is a pointer to an int. To call p we might say `p(3, &i)` where `i` is an int variable; this passes the location of `i` in that argument position. Within the body of `p`, to get the value of `y` we say `*y`, e.g.,

```
int factorial (int x);
/* checks: x ≥ 0
   effects: returns i such that a[i] = x */

int factorial (int x) {
   int f = 0;
   assert (x ≥ 0);
   for ( ; x > 0; x−−) f = f * x;
   return f; }
```

Figure 16.1 Specification and implementation of factorial procedure.

```
x = x + *y;
```

The absence of exceptions is more serious. What to do here depends on the routine being defined and what we think users need when that routine is unable to do the "normal" thing. If we think users what to be informed about the problem, we should do this, possibly by having an extra result. If we think users are treating the situation as an error, i.e., they never expect it to arise, then we should cause the program to fail if the situation arises (just as it would in C++ if the exception weren't handled).

We have already discussed how to convey information about the fact that inputs aren't in the "normal" domain in previous chapters, so here we limit discussion to the latter situation. This is easy to handle by introducing an extra clause in the specification, the *checks* clause. This clause is like the requires clause in that it specifies a predicate that is expected to hold when the procedure is called. However, in the case of the checks clause the implementation of the procedure *must* check that predicate, and cause execution to halt if the predicate is false.

For example, Figure 16.1 gives the specification and part of the implementation of a `factorial` routine whose argument must be non-negative. The implementation calls the `assert` primitive; this primitive causes the program to halt if its boolean argument is false.

It is often the case that some users want a routine to tell them about exceptional conditions while others want these to be considered as errors. Therefore, it may be necessary to define two versions of a procedure, one for each set of users. An exception mechanism avoids this problem since the user can decide whether to handle the exception or not.

16.2 Data Abstraction

C provides no mechanism like a class for defining and implementing a data abstraction. Therefore we will need to invent our own mechanism here. There are two main ways that the mechanism could work: As in C++ we could have the operations be methods of an object. This would require us to put the methods into a `struct` that the object would point to; then to call a method we could dispatch through the structure. Alternatively, we could have the operations simply be procedures that work on objects of the new type. These procedures do not belong to a particular object; instead they are passed the object to work on as an explicit argument.

The second approach is much simpler than the first and therefore is the one that will be discussed in this section. This approach provides fine support for data abstraction but not for inheritance. Of the two, data abstraction is by far the more important. Furthermore, supporting inheritance is not really practical in C (or in any language without language support for it).

As in C++, the specification for the data abstraction will be placed in a header file and the implementation will be placed in a source file that includes that header file. Note that C does not require us to organize things like this; instead this is a convention we are adopting.

The source file will need to contain information about the rep (so that when the compiler compiles code that uses the abstraction, it knows how much space to allocate for variables of the new type). Since there is no class mechanism with public and private sections, we will indicate what is what by means of comments. The general form will be:

/* *datatype dname*
overview: .../

/* *PRIVATE* */
(some information about the rep and **private** routine headers goes here)

/* *PUBLIC* */
(specifications of the operations go here)

We need to put the private section first because it will define the new type by giving its rep. This information needs to precede the headers for the operations since they are defined in terms of the type.

16.2.1 Issues

C++ classes provide a number of abilities that are lacking in C. We will need to design conventions to compensate for this lack. There are a number of issues to consider when designing these conventions.

An absolute requirement is the following: *To reimplement a type it should not be necessary to modify the source code of any modules that use the type.* Obviously, if using code had to be modified, this would be a rich source of errors. It is also desirable (but less important) that reimplementation of a data abstraction not require recompilation of any using modules.

Additionally, the conventions must be *easy to use.* Making a convention easy to use not only increases the probability that it will be followed, but also decreases the likelihood that errors will arise from its misuse.

Two additional criteria, *type safety* and *protection*, are desirable, but can be difficult to achieve in some languages. Type safety means that objects belonging to different types are kept distinct: An object of one type cannot be used in a context where some other type is required. For example, it should not be possible to pass a polynomial to a set operation. C++ supports security by compile-time type checking (assuming using code avoids casts). Ansi C provides a similar level of security.

Protection means that access to an object's representation is possible only from inside an implementation of the object's type. C++ classes provide protection. C has no mechanisms that will provide it. Note that in the absence of protection it is not possible to guarantee local control for a type's implementation, and therefore reasoning about its correctness is no longer local. Unfortunately, protection is very hard to achieve unless the language provides an encapsulation mechanism. If it does not, it is better to use management techniques such as having people read one another's code to enforce protection.

16.2.2 Stack vs. Heap

As in C++, the definer of a new type needs to decide whether its objects should be on the stack or on the heap. In C++ this decision was merely a guide to users. For example, a user could decide to place objects of a heap type on the stack (although the using code would not be able to assign to such objects nor to pass them by value).

In C, however, the decision is more binding. Users *must not* place heap objects on the stack. The reason is that there is no way in C to redefine the assignment operator (or call by value) to have a type-dependent meaning, nor is there any way to make assignment unavailable. Instead, assignment simply means copying the fields of the rep, which will *not* do the right thing if some of these fields are pointers.

A second point is that *only* types for which the built-in meaning of assignment is correct should be stack types. One implication of this point is that it will not be possible to use any heap storage in conjunction with a stack type. In C++ a stack type was one whose objects could be allocated on the stack *but* those objects might contain pointers to storage on the

```
#include "6170.h"
/* datatype poly
overview: polys are immutable polynomials over the integers.
  poly is a heap abstraction */

/* PRIVATE */
  typedef struct { int deg; int_array trms; } poly;

/* PUBLIC */
  poly * poly_new (int c, int n);
  /* checks: n >= 0
    effects: returns a new poly cx^n */

  int poly_degree (poly * p);
  /* effects: returns the degree of p */

  int poly_coeff (poly * p, int n);
  /* effects: returns the coefficient of the n term of p */

  poly * poly_add (poly *p, poly *q);
  /* effects: returns a new poly that is the sum of p and q */

  void poly_delete (poly * p); /* the destructor */
```

Figure 16.2 Partial specification for poly.

heap; for example, strings are like this. In C doing things this way will not work, again because there is no way to control the meaning of assignment.

16.2.3 Heap Abstractions

This section illustrates how to define heap abstractions by means of a simple example, a specification and implementation of type `poly`. The `poly` type must be a heap abstraction because its objects are not of fixed size.

Figure 16.2 gives a partial specification of `poly`. The private section gives the `poly` rep as a `struct`; the fields of this `struct` correspond to the instance variables of a C++ class. The representation is as in C++. Note that `trms` is an `int_array`; we assume this type (which is like our 6.170 arrays) is specified in `6170.h`.

There are several things to notice about the specifications of the operations. First, note that all the operations have "poly" as a prefix. This convention avoids name conflicts among operations of many different types, e.g., `poly_add`, `rational_add`, etc. Second, note that instead of constructors, we have `creators` that return new objects of the type. A type can

```
poly * poly_new (int c, int n) {
  int i;
  poly * p = (poly*) malloc (sizeof (poly));
  assert (n ≥ 0);
  p → trms = int_array_new ( );
  if (c == 0) n = 0;
  trms → deg = n;
  for ( i = 0; i < n; i++) int_array_append(p → trms, 0);
  int_array_append(p → trms, c);
  return p; }

int poly_degree (poly * p) {
  return p → deg; }

int poly_coeff (poly * p, int d) {
  if (d < 0 || d > p → deg) return 0;
  return int_array_fetch(p → trms, d); }

void poly_delete (poly * p) {
  int_array_delete(p → trms);
  free (p); }
```

Figure 16.3 Part of poly implementation.

have several creators; `poly` has just one, `poly_new`. Next, note that the operations that correspond to C++ methods all take the objects being operated on as an extra argument. Thus `degree` and `coeff` both take a `poly` as an argument, and `add` takes two `polys` as arguments. An additional point is that the type is geared for use on the heap, since all operations that take `polys` as arguments or return them as results actually deal with pointers to `polys`.

Finally, note that `poly` has a destructor operation to deallocate the storage of a `poly` object. No specification need be given for the destructor since it has the predefined meaning (*all* storage for the argument object must be deallocated). The destructor must be called explicitly by using code.

Figure 16.3 shows part of the implementation of `poly` (which would be placed in a source file that included the associated header file). The creator allocates space for the new object on the heap and then initializes that space. It uses the C `malloc` primitive to allocate the space and then casts the `void*` result to a `poly*`. `malloc` requires an argument that tells it how much storage to allocate; the `sizeof` builtin function is used to determine the proper size.

The other `poly` operations are implemented similarly to their C++

implementations. The main differences are that they take a `poly` as an extra argument, and they call operations of the `int_array` type, passing the array as an extra argument in the calls.

Now we consider briefly how to use `poly`. The following is an example of correct use, in which `poly`s are placed in the heap:

```
poly *p, *q, *r;
p = poly_new(2, 3);
q = poly_new(3, 4);
p = poly_add(p, q);
...
poly_delete(p); /* must delete explicitly */
```

Note in particular the explicit call on the destructor. This call must be explicit; C will not generate *any* implicit calls, e.g., when execution exits a scope.

The following use is incorrect:

```
poly x, y; /* polys should not be on the stack */
```

This declaration will *not* be found to be illegal in C. Instead, C will allocate sufficient stack storage for the two fields of the rep, but the fields will be uninitialized, and there will be no way to initialize them without breaking protection, i.e., without using code accessing fields of the rep directly. If using code does not violate protection, heap objects cannot usefully be placed on the stack. However, note that C does *not* enforce protection. Instead programmers must voluntarily follow the convention that only code implementing an abstraction can access the rep.

Now consider the following code:

```
poly * p;
intset * s;
...
intset_insert(p, 3); /* compile time error */
```

Because the reps of `poly` and `intset` are defined as `struct`s, C views them as belonging to different types, even if they have the same format (the same number of fields in the same order with the same names and types). Since the `.h` file for `intset` says that the first argument to `intset_insert` must be an `intset`, a compile time error occurs when an attempt is made to instead pass a `poly`. Thus our way of defining types in C does provide type safety.

Note that it is essential for type safety that the rep be a `struct`, since otherwise C will view the new type as identical to any other type with that

```
#include "6170.h"
/* datatype rat
overview: objects of type rat are rational numbers. These objects
  are immutable and on the stack. */

/* PRIVATE */
typedef struct {int n, int d} rat;

/* PUBLIC */
  rat_init (rat *r, int n, int d);
  /* checks: d is non-zero
    effects: initializes r to be the rational n/d. */

  int rat_num(rat r);
  /* effects: returns the numerator of r (in reduced form) */

  int rat_denom(rat r);
  /* effects: returns the denominator of r (in reduced form). */

  rat rat_add (rat r, rat s);
  /* effects: returns the sum of r and s */

  rat rat_mul (rat r, rat s);
  /* effects: returns the product of r and s */

  bool rat_equal (rat r, rat s);
  /* effects: returns true if r = s else returns false. */
```

Figure 16.4 Partial specification of the rational number abstraction.

rep. Therefore, you should always use a `struct` to define the rep, even if only one field is needed!

16.2.4 The Stack Approach

Now we illustrate stack abstractions in C by means of a simple example, rational numbers. The specification is given in Figure 16.4; Figure 16.5 shows part of the implementation. There are several points of interest here. Note first that instead of a creator, type `rat` has an "initializer" that initializes previously allocated storage for a rational. All stack abstractions have initializers instead of creators. The argument to an initializer is a pointer to the storage for the object; a pointer is necessary here, since an initializer must have access to the actual storage in order for the initialization to take effect. Second, note that the other operations take their rational arguments by value. This means they will be copied into the formal; the

```
rat_init (rat *r, int n, int d) {
  assert (d != 0);
  r → n = n;
  r → d = d; }

int rat_num (rat r) {
  int g = gcd (r.n, r.d);
  return r.n/g; }

bool rat_equal (rat r, rat s) {
  return rat_num(r) == rat_num(s) && rat_denom(r) == rat_denom(s); }
```

Figure 16.5 Partial implementation of rational abstraction.

copy is acceptable since rationals are small. Alternatively, the operations could take their arguments by reference, but that makes them somewhat less convenient to use. Finally, note that there is no destructor; destructors aren't needed for stack abstractions, since the storage for objects on the stack is deallocated automatically when control exits the containing scope.

Here is an example of correct use of rationals:

```
rat r, s, t;
rat_init (&r, 2, 3);
rat_init (&s, 3, 4);
t = rat_add (r, s);
if (rat_equal(r, s)) ...
```

The declaration creates but does not initialize the storage for the rationals. Note the possibility for error here: nothing ensures that storage is properly initialized before use. In this case, however, the storage is used properly. The initializer is called to initialize the storage of `r` and `s`, and `t` is initialized by the assignment to hold the sum of `r` and `s`. Note that it is not possible to say `r + s`; C provides no way to link operations to the expression syntax. Finally, note the use of the `rat_equal` operation in the `if` statement. This operations must be called to check for equality. If using code says `r == s` where `r` and `s` are rationals, the components of the two arguments will be checked for equality, which may give the wrong answer (since the reps are not maintained in reduced form).

It isn't possible to use rationals on the heap because they haven't got any creators and there is no destructor. If these operations were provided, rationals could be allocated on the heap, although they would be less convenient to use than when they are allocated on the stack.

```
/* datatype T_set
overview: T_sets are sets containing elements of parameter type T.
  T_sets are mutable, on the heap. */

/* REQUIRES: bool T_equal (T, T) which is an equality operation on T */

/* PRIVATE */
  typedef struct { int sz; T_array els; } T_set;

/* PUBLIC */
  T_set * T_set_new ( );
  /* effects: returns a new, empty set */

  ...

/* IMPLEMENTATION */
  T_set * T_set_new ( ) {
    T_set * s = (T_set *) malloc (sizeof (T_set));
    s → sz = 0;
    s → els = T_array_new( ); }

  void T_insert (T_set * s, T x) {
    int i;
    for ( i = 0; i < T_array_length(s → els); i++)
      if (T_equal(x, T_array_fetch(s → els, i)) return;
    s → sz = s → sz + 1;
    T_array_append(s → els, x); }
```

Figure 16.6 Partial specification and implementation of set.

16.3 Parameterized Abstractions

C has no mechanism like templates that makes it possible to define parameterized abstractions. In the absence of such a mechanism a simple approach is to use a text editor to do instantiation. For example, suppose there is a file containing the specification and implementation of a set abstraction, as shown in Figure 16.6. Here `T` is used throughout the file to stand for the parameter type, including within the names of `set` operations. Furthermore, the name `T_equal` is used for the required `T` operation.

To produce code for an instantiation of `set` we:

1. For each occurrence of the name of a required operation (`T_equal` in this case) substitute the name of the procedure that should be used.

2. Substitute the name of the actual parameter type for all occurrences of

the associated formal (`T` in this case).

We then use the specification to produce the header file, and the implementation to produce the source file. Both files are then compiled.

For example, to produce a `set` of `int`, we might substitute `equalp` for `T_equal`, and then substitute `int` for `T`, obtaining a definition and implementation for `int_set`. Sometimes the first step can be eliminated. For example, to produce a `set` of `poly` we just substitute `poly` for `T` everywhere, obtaining a specification and implementation for `poly_set` (since `poly_equal` is what we want for the required operation).

In this chapter we have assumed that text editing is being used for instantiation. In particular, `int_array` is an instantiation of a general array abstraction, and `6170.h` includes several common instantiations of `array`. If others are needed, the user would need to provide the instantiation explicitly as discussed above.

Using text editing for instantiation has two problems:

1. It doesn't work well when the implementation of the parameterized abstraction is modified, because then you must remember to recreate all the instantiations.

2. It leads to code blowup, since there is a different implementation for each instantiation.

The former problem can be eased by using macros; C does support macros, but many languages do not. The latter problem still exists when macros are used (and in fact the problem exists in C++ as well).

16.4 Summary

When using our programming method with some language other than C++, there will be areas in which the language does not fit the method very well. In such a case we offer two suggestions:

1. Adapt the methodology to the language—for example, by changing the form of procedural abstractions to include a checks clause.

2. Invent conventions for implementing abstractions in the language, e.g., to name operations of data abstractions.

By changing the methodology, we ensure that the programs we design are suitable for implementation in the language. The conventions then ensure that implementations proceed in a straightforward way.

Choosing the right language for a programming project is important. Some languages make it much easier to do modular design and implementation than others. The methodology is more important than the programming language, though, and it can be used with any language. It is the programming method, not the language, that provides the power of modularity and abstraction.

Further Reading

Shaw, Mary, 1984. Abstraction techniques in modern programming languages. *IEEE Software* 1(4): 10–26.

Exercises

16.1 Specify and implement in C a bounded first-in/first-out queue with room for 100 elements. Be sure to provide a generator to iterate over the elements of the queue. Use a stack-based implementation.

16.2 Define a technique for using the methodology for some language other than C++ or C. Do an example program for each part of the method. Be sure to include exceptions, an iterator, a heap data type, and a stack data type.