C++ Solutions

Companion to *The C++ Programming Language, Third Edition*

David Vandevoorde

ADDISON–WESLEY
An Imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Computer and Engineering Publishing Group
Addison Wesley Longman, Inc.
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data

Vandevoorde, David.
C++ solutions : companion to The C++ programming language, Third edition / David Vandevoorde.
p. cm.
Includes index.
ISBN 0-201-30965-3
1. C++ (Computer program language) I. Stroustrup, Bjarne. C++ programming language. II. Title.
QA76.73.C153V36 1998
005.13'3—DC21 98–20523
CIP

ISBN 0-201-30965-3
1 2 3 4 5 6 7 8 9—MA—0201009998
First printing, July 1998

To Franklin T. Luk,
friend and mentor

[blank page]

Contents

Acknowledgments

Thank You

This book owes its existence to the contributions of many people.

It all started when Deborah Lafferty (an editor at Addison-Wesley) and Bjarne Stroustrup contacted me to write this book—I owe both of them for their initiative and for the suggestions they offered throughout the writing process. Marina Lang and Mike Hendrickson were also instrumental in the book design, and I must thank Mike for demonstrating useful Adobe FrameMaker procedures.

Or perhaps it all started when I bought my first copy of a book by Bjarne Stroustrup, or even when he decided to design a language that truly enhances my enjoyment of programming. Thus, I must thank those people who directly or indirectly taught me elements of C++: Bjarne Stroustrup, Josee Lajoie, Bill Gibbons, John Spicer, and Steve Adamczyk, in particular. Much of what is written in this book I also learned from discussions in the various technical C++ Usenet forums. This learning process continued through the many hours of voluntary work put in by the moderators of `comp.lang.c++.moderated` and `comp.std.c++`: Matt Austern, Stephen Clamage, Domenic De Vitto, Howard Harkness, Fergus Henderson, James Kanze, Dietmar Kühl, Robert Martin, John Potter, and Herb Sutter.

Eva Wong put up with my many hours of typing and grumbling around our home. Also many thanks to my parents and parents-in-law for their continuous encouragement.

Michael Beckmann, Sassan Hazeghi, and Chris Maitz of Hewlett-Packard were most helpful and supportive as I combined this project with my other professional responsibilities.

Franklin Luk deserves credit for his patience in teaching me—among other things—to write technical texts. I hope his efforts weren't wasted.

This book benefitted *immensely* from the corrections and suggestions by early reviewers: David Francis, Jeffrey Oldham, Srikanth Sankaran, Ed Schiebel, Jay Shain, Bjarne Stroustrup, Clovis Tondo, and Chris Van Wyk. Their contributions cannot be overstated.

David Vandevoorde
Belmont, California

[blank page]

Chapter 1

Introduction

Regardless of which programming languages are selected to write practical, effective programs, developing software is never trivial and often complex. Bjarne Stroustrup's *The C++ Programming Language (Third Edition)* shows how C++ provides a number of tools that can be applied to divide and conquer the challenges faced by modern software designers.

This conquest does requires a certain amount of experience, which in turn requires practice. No book can "supply practice," but Stroustrup provides a large number of exercises that can be used to review the knowledge and acquire experience. This book provides sample answers and discussions for a selection of these exercises.

Overall Organization

The exercises selected for discussion in this book generally have the following three properties in common:

1. They illustrate important concepts or common uses and methods of C++.
2. Their solution is largely independent of a particular platform.
3. They can be solved fairly concisely.

I occasionally deviate from these rules when I feel the discussion warrants it. For example, a few exercises come with solutions that span several pages.

For easy reference, I follow Bjarne Stroustrup's chapter organization—for example, you will find the discussions about exercises presented in Chapter 15 of *The C++ Programming Language* in Chapter 15 of this book.

To accommodate the organization of this book, I had to decide what to do for the second and third chapters. In *The C++ Programming Language,* these chapters take the reader on a tour of the language and the library, but they do not present exercises. I chose to use these two chapters to provide a short presentation of fundamental C++ concepts and some

consequences of language evolution. Chapters 23 and 24 do not present exercises either; we used placeholders in this book.

This book is not a complete tutorial or a reference. Instead, the discussions frequently refer back to the third edition of *The C++ Programming Language*. References to specific paragraphs will use the § symbol. For instance, §C.3.1 refers to paragraph C.3.1 in appendix C of Stroustrup's book. However, I often elaborate on less-common C++ language constructs and even on common ones when they may result in subtle situations.

Almost every exercise statement is followed by a short Hints paragraph. These paragraphs point to related exercises, indicate relevant material in *The C++ Programming Language*, and suggest how fundamental or advanced the topic covered by the exercise is.

Most exercises can be solved in many ways. When no method appears clearly superior, I often suggest various alternative approaches. Even when I do think a particular solution is technically superior, I may still introduce a simpler one for the sake of illustration. Many problems—even the simpler ones—can easily lead to tangential experiments and discussions. I therefore sometimes point in these tangential directions using "supplementary exercises," which are new exercises not proposed in Stroustrup's book but which can be used to further explore various software development challenges. Note, however, that these exercises may involve solutions that are well beyond the scope of the answers in this book.

Short Guide to the Exercises

The exercises in *The C++ Programming Language* were crafted as a vehicle to expand one's understanding of the possibilities of the language. Together, they cover a broad spectrum of C++ programming knowledge and solution components. Yet I still find that most of the selected exercises fall in one or a few of the following categories:

- *Facts and constructs:* Exercises that could be used as tests of one's familiarity with the language specifications. The following belong to this category: 4.2, 4.3, 4.7, 5.1, 5.5, 5.6, 6.1, 6.2, 6.4, 6.5, 6.6, 6.7, 6.9, 7.1, 7.2, 7.14, 8.4, 8.5, 8.7, 9.9, 10.1, 10.4, 11.1, 11.7, 12.1, 13.1, 13.15, 15.2, 16.15, 20.1.
- *Common idioms and good practices:* Illustrations of how C++ is often used to address various small programming problems. See Exercises 5.3, 5.4, 5.7, 5.10, 5.11, 5.12, 5.13, 6.3, 6.10, 6.11, 6.12, 6.13, 6.14, 6.16, 6.17, 6.19, 6.22, 7.3, 7.4, 7.6, 7.7, 7.9, 7.11, 7.16, 7.18, 7.19, 8.1, 8.10, 9.5, 10.2, 10.5, 10.12, 10.19, 11.3, 11.4, 11.8, 14.1, 14.2, 14.9, 15.3, 16.1, 16.2, 16.3, 16.6, 17.3, 17.6, 18.2, 18.5, 18.10, 18.11, 18.14, 19.1, 19.2, 19.3, 20.5, 20.16, 21.1, 21.2, 21.12, 21.18, 21.24, 22.1, 22.2, 22.4, 25.1, 25.5, 25.7, 25.19.
- *Experiments:* Exercises that include programs that illustrate implementation-dependent behavior or performance. Several exercises contain little benchmarks that are very useful in acquiring some intuition about the cost of various C++ features and techniques. See Exercises 4.3, 4.4, 4.5, 4.6, 5.8, 6.8, 8.6, 8.8, 9.1, 17.1, 18.19, 20.15.
- *Advanced idioms and design techniques:* Discussions of more specialized problems. Usually, I try to emphasize the general thought process leading to a solution in these

cases. Applying the resulting ideas to your own situation will often require some non-trivial adaptation. See Exercises 7.19, 12.10, 13.2, 13.16, 14.10, 21.13, 22.8.

- *Trivia and horrors:* Every programming language has its own little peculiarities and can be abused to produce unintelligible programs. A few exercises explore this darker side of software development, as well as pecularities that are not harmful but not fundamental either, as in Exercises 4.6, 5.9, 6.15, 10.15, 11.10, 11.20, 11.21, 12.9.

The fact that most exercises fall into one of the first two categories testifies to the fact that this book focuses primarily on C++ fundamentals. Once that is established, however, the more advanced material should become more accessible and lead to added productivity. In contrast, the exercises in the category *Experiments* do not usually require familiarity with the advanced C++ features.

Suggestions

Don't feel compelled to work through the exercises in their numeric order. However, it may be useful to start with the exercises in the first category to refresh your fundamentals and then proceed with those in the second category (*Common idioms and good practices*) to get fluent with the language elements presented in the corresponding chapter. As is the case with *The C++ Programming Language,* the index in this book provides quick and easy access to topics of interest.

It is not always necessary to get completely familiar with the material in a chapter of *The C++ Programming Language* before trying out the exercises. However, it is useful to get a bird's eye view of the language by reading the second and third chapters of Stroustrup's book (the "Tour" chapters).

Of special interest are the slightly longer exercises. Exercise 15.3 is a complete object-oriented design and implementation of a popular board game. For an example implementation of standard-compliant elements, see 18.2 for an algorithm and 19.3 for a complete standard iterator adaptor. Exercise 20.5 is shorter but very useful for this purpose, as well. For a demonstration of the power of the standard libary, consider Exercise 6.12, which computes a variety of statistics on a set of data. If you're looking for an exercise to sharpen your skills with the more traditional features (statements and expressions), try 6.22.

I believe that to get the most out of this book, it is best to first try the exercises without reading the proposed solutions. Once you have found a solution to an exercise, you can compare your solution to the one in this book. In some cases, this comparison may suggest that there is really one "natural answer" to the exercise. But more likely, you will find that there are many ways to accomplish the same task. If a solution cannot be found within a reasonable time, reading the discussion of the corresponding exercise in this book will provide clarification. Often, this discussion will lead you to consider alternatives to those I propose, which is valuable practice.

Many exercises are best solved by actually implementing their solutions. There is much to be gained by sketching a solution and convincing yourself that the approach taken is valid. However, the actual implementation requires an additional level of attention to detail, and

good software design skills are acquired by cultivating a healthy respect for details. This is doubly true for the exercises in the *Experiments* category; do not hesitate to modify these experiments to further your insights into what makes C++ program tick and tick well.

Of course, good judgement is also needed to discriminate between the fundamental mechanisms and the details of the scaffolding. In some cases, you may find that one of the more annoying aspects of implementing solutions to the proposed exercises stems from having a C++ implementation that does not strictly comply with the ISO definition of the language. Chapter 3 deals with the most common issues encountered in this respect.

The C++ Programming Language (Third Edition) is, of course, highly recommended as a source of information to study the proposed software solutions. Once you are familiar with most techniques, you might want to sink your teeth into more literature. There are many excellent books on C++ out there that present material beyond the scope of *The C++ Programming Language*. I want to emphasize only a few:

Effective C++: 50 Specific Ways to Improve Your Programs and Designs (Second Edition). Scott Meyers. Reading, MA: Addison Wesley Longman, 1997.

More Effective C++: 35 New Ways to Improve Your Programs and Designs. Scott Meyers. Reading, MA: Addison Wesley Longman, 1996.

The Design and Evolution of C++. Bjarne Stroustrup. Reading, MA: Addison Wesley Longman, 1994.

The first two books describe a large collection of rules of thumb and techniques that will lead to more robust and often more efficient C++ programs. General design principles and specific coding techniques are covered.

The third book describes the evolution of the C++ language up until about the end of 1993. I have found that understanding this evolution helps in determining how the rich set of C++ constructs can be best used. It also helps to make some of the subtler properties of C++ more intuitive and manageable.

Finally, I would like to point out that the C++ community is large. Discussion of techniques, semantics, and design principles with friends and colleagues is invaluable. In addition, there are many conferences and electronic forums that pertain to C++ programming. In particular, I have found that the Usenet forums `comp.lang.c++.moderated` and `comp.std.c++` provide enlightening discussions on a large variety of C++ issues.

Chapter 2

C++ Concepts

This book is not intended as a stand-alone tutorial for the C++ programming language. Instead, *The C++ Programming Language* is an excellent resource for that purpose. However, it is always useful and convenient to summarize some fundamental concepts and agree on terminology.

You can safely skip this chapter and return to it when you need a quick review of some fundamental C++ topic. For more complete and in-depth information refer to Stroustrup's book.

Language versus Implementation

A programming language is an abstract thing. It is a set of rules and conventions to describe the behavior of programs. The concretization of this is a *language implementation*—a device, often a program itself, that translates a program description into actual behavior. The translation process always enforces some of the rules set by the programming language.

For C++, the implementation consists of the vast majority of cases of a *compiler*, which is a program that translates text (*source code*) into an *executable program*. Once translated, the program can then be run repeatedly on a *target platform* (a computer).

This book assumes that these concepts are not totally foreign to you. Some practical considerations are also discussed in Exercise 4.1.

Tokens

C++ source code is usually divided up in multiple files (§9.1). When such a file is *preprocessed*, comments are stripped and macros are substituted. The result is a so-called *translation unit*. A translation unit is a sequence of *tokens*: words, numbers, and symbols. They are

the smallest entities that can have a meaning by themselves in C++. For example, the following snippet of C++ code:

```
int const hundred = 100 ;
```

consists of six tokens: '`int`', '`const`', '`hundred`', '`=`', '`100`', and '`;`'. *Identifiers* are tokens whose "spelling" can be freely chosen by the programmer. They consist of an arbitrary sequence of letters, digits, and underscores ('`_`'), but cannot start with a digit. Note that the capitalization of letters matters. Some sequences are, however, not allowed as identifiers:

- *Keywords* (§A.2) and *alternative representations* for certain symbols: '`int`' and '`const`' are examples of keywords. '`and`' is an example of an alternative token for the symbol '`&&`' (*logical and* operator). The C++ programming language has 63 keywords and 11 alternative representations.
- *Reserved names*: It is sometimes convenient for an implementation to introduce names of its own. Names containing two consecutive underscores (for instance, '`re__served`') and names starting with an underscore followed by a capital letter (for instance, '`_Reserved`') should, therefore, not be used as user-defined identifiers.

Tokens are often separated by *whitespace*, although that whitespace has no meaning of itself. In many cases, the transition from one token to the next does not require such whitespace and there are no restrictions on the amount of whitespace allowed between tokens. For example, the declaration above could appear in any of the following ways:

```
int const hundred=100;
int
   const
      hundred
         =
    100                                    ;
```

But the following would not work:

```
intconst hundred = 100;
```

Without intervening whitespace, `intconst` is a single—presumably user-defined—identifier rather than two keywords.

Certain uses of whitespace are detrimental to source code clarity. Our third example illustrates this. Nonetheless, individual tastes vary when it comes to source code style, and it would be a futile exercise for this book to try to impose a set of rules in this regard. Many projects do follow certain guidelines for formatting C++ source code in an attempt to ease the understanding of source files (see also §4.9.3 and Exercise 6.23).

Names, Declarations, and Scopes

Many entities in C++ are referred to by using a *name*. The introduction or *re*introduction of such a name is called a *declaration*. The part of a translation unit in which a declaration can

be referred to by its plain associated name is called the *scope* of that declaration. For example:

```
int a = 3;
int b = 2; // <- scope of global b starts here

void f() {
   int b = 0; // <- scope of global b stops here,
   ++a;       //    scope of local b starts
   ++b; // Refers to local b
} // <- scope of local b ends, scope of global b resumes
```

As the example shows, two different declarations can be associated with a single name. When that happens, the "innermost" name hides the outer one. In some cases, hidden names and names in nonactive scopes can be accessed by using so-called *qualified names*—names containing the *scope-resolution operator* '`::`'. For example:

```
int a;

namespace N {
   int a, b;
   void f() {
      ::a = 1; // Qualified name for (hidden) global 'a'
      a = 10; // Unqualified name accesses inner 'a'
    }
}

void g() {
   a = 10; // Global 'a'
   N::a = 7; // 'a' in namespace scope (qualified name)
}
```

A *qualifier* on the left of the scope-resolution operator must be either the name of a namespace (§8.2) or that of a class (§10.2). This implies that names local to a function cannot be accessed through a qualified name.

Objects, Types, References, and Functions

Objects are things that occupy a contiguous region of storage (computer memory). However, a random set of contiguous bits of information is usually not considered an object. Instead, an object has a *type* associated with its physical bits. A type is thus a collection of attributes and applicable operations that determine how the bits constituting an object must be interpreted. For example, on many systems the definitions

```
char c = 1;
```

and

```
bool b = true;
```

result in exactly the same set of physical binary values. Yet because their types are different, ++c and ++b will most likely have different physical values.

C++ has a number of built-in *fundamental types*; `bool`, `char`, `unsigned int`, and `double` are examples of these (§4.2, §4.3, §4.4, §4.5). Additional types are called *compound types* and can be created in a variety of ways. These compound types fall into one of the following categories:

- Enumerations (§4.8)
- Classes (including structures; §5.7, §10)
- Unions (§10.4.12)
- Arrays (§5.2)
- Pointers (§5.1, §5.3)
- References (§5.5)
- Functions (§7)

Although functions and references have types, they are not *objects* in the C++ sense of the word. References are "aliases" for objects—as such, they do not always require storage at run-time; instead, the alias can sometimes be substituted by the C++ translator. Functions typically do require run-time storage for their representation, but C++—like most other high-level programming languages—does not make that storage available to the programmer. This prevents us, for example, from writing so-called self-modifying code.

Lvalue and Rvalue Expressions

Objects can exist because you explicitly define them or allocate them (for example, by using the `new` operation). If an expression refers to such an object, the expression is called an *lvalue*. Otherwise, it is called an *rvalue*. For example:

```
int a;
int *b = &a;
a = 2; // 'a' is an lvalue, '2' is an rvalue
*b = a; // Both '*b' and 'a' are lvalues
b = new int; // 'new int' is an rvalue; 'b' is an lvalue
int const &ri = *new int; // '*new int' is an lvalue
a = ri; // 'ri' is an lvalue
int f();
*b = f(); // 'f()' is an rvalue
int& g();
a = g(); // 'g()' is an lvalue
```

The *l* in the term *lvalue* was inherited from ancestors of the C language (for example, BCPL, §1) in which lvalue could appear on the *left* side of an assignment, whereas rvalues could not. In modern C and in C++, lvalues may not always appear on the left of an assignment operator because the referenced object is `const`. Such lvalues are called *nonmodifiable lvalues*. C++ also provides ways to modify rvalues, but it is rarely necessary to do this.

The results of arithmetic operators (+, -, %, *, ...) and the results of function calls are often rvalues unless they return references.

Initialization versus Assignment

Initialization (§4.9.5) is the process that creates the first value of an object. *Assignment* (§2.3.1), on the other hand, is an operation that usually changes the value of an object—a previous value existed. Assignment is expressed using the token '=,' but sometimes initialization can also be indicated with that symbol:

```
int a = 12; // Initialization (definition)
a = 13; // Assignment
```

Basically, the '=' symbol represents initialization when it appears as part of a declaration; otherwise, it is assignment.

Initialization can also occur without the '=' token, as well. For example:

```
int a(12); // Same as above
```

Also, the transfer of function arguments happens through initialization. For example,

```
void f(X x) { /* ... */ }

void g() {
   X a; // Assume X is some type
   f(a); // parameter 'x' acquires 'a' value
}        // by initialization
```

For classes, you can define your own initialization procedure by providing constructors (§10.2.3, §10.4) and your own assignment operation by providing an `operator=` (§10.4.4.1). Remember that your `operator=` *will not* be called for initialization even if that initialization is expressed using the '=' symbol.

Declaration Syntax

The syntax of the most commonly occurring declarations is relatively intuitive, but the general syntax rules can sometimes be surprising. The discussion below summarizes some of the fundamentals explained in §5, §6, §7, and §10.

Variables and Constants

A declaration consists of four parts (§4.9.1):

1. A set of "specifiers" (optional)
2. A base type
3. A declarator
4. An optional initializer

The specifiers correspond to a variety of properties associated with the entity being declared—for example, `extern` (§9.2), `virtual` (§12.2.6), and `explicit` (§11.7.1)—or with the type of that entity (for example, `const`, §5.4). Interestingly, there are few limitations to the order in which specifiers appear, and, in fact, they may even appear after the base type. For example, all of the following are legal and equivalent:

```
extern int const c;
const int extern c;
const extern int c;
int extern const c;
```

Almost all programmers prefer to keep the specifiers associated with the type (the *type specifiers*) close to the base type; the other specifiers usually precede the base type and the type specifiers. The above declaration is thus generally ordered in one of the following ways:

```
extern const int c; // "const" is a type-specifier,
extern int const c; // "extern" is not
```

The former ordering has been the most popular, but the latter is sometimes preferred for reasons explained in Exercise 5.1.

The *base type* can be a fundamental type (for example, `float`) or the name of an `enum`, a `class` (or `struct`), or a `union`. If the base type is not a fundamental type, its name can be qualified (see above) and *elaborated*. The latter means that the—possibly qualified—name is preceded by one of the keywords `enum`, `struct`, `class`, or `union`. For example:

```
namespace N {
   enum E { two = 2, seven = 7 };
   E a; // Name E is not qualified and not elaborated
   enum E b; // Elaborated, but not qualified name E
}
enum N::E c; // Elaborated and qualified name E
```

The *declarator* part of a declaration consists of the declared name optionally surrounded by *declarator operators* that modify the base type with which the name is being associated. These operators are used to create references, pointers, arrays, functions, and combinations of these things. Let's first concentrate on pointers, which are declared using a (prefix) asterisk '`*`.' A pointer `p` to an integer can thus be declared as follows:

```
int *p;
```

Above, we saw how the keyword `const` can be associated with a base type. For example,

```
int const *p;
const int *q;
```

imply that `p` and `q` are pointers to integers but that they cannot be used to modify the integers they point to. Sometimes, however, we want to express that the pointers themselves should not be modified. This is achieved by following the declarator operator '*' with the keyword `const`:

```
int i;
int *const cp = &i;
int const *const cpc = &i;
```

The latter declaration specifies that the pointer `cpc` may not be modified and that what it points to cannot be modified through `cpc` either:

```
*cp = 3; // OK: not a pointer to a const base-type
int j;
cp = cpc = &j; // Error: cannot modify const pointers
*cpc = 2; // Error: pointer to a const base-type
```

The declaration syntax for references closely follows that of pointers, with the declarator operator spelled as an ampersand, '&', instead of an asterisk. However, references are never `const`:

```
int const &r = j, &s = j; // OK: references to const ints
int &const cr; // Error: no const references
```

C++ also includes a qualifier `volatile` that can appear wherever `const` can appear (syntactically, at least). It denotes that the quantity being qualified may change asynchronously—that is, the change is not caused by the current thread of execution. The use of `volatile` is much less frequent than that of `const`; its meaning is also much less precise but is intended to tell the C++ compiler to be more conservative when optimizing accesses to the qualified entity. `const` and `volatile` can be applied simultaneously, but that is even less common:

```
int volatile strange;
int volatile *const cpv = &strange;
int const volatile &rcv = strange;
```

This last declaration says that `rcv` cannot be used to modify what it refers to, but that "an outside influence" may, in fact, alter the value referred to. For example, the variable may map onto a hardware clock that is updated every microsecond, but that cannot be modified by your program. The qualifiers `const` and `volatile` are sometimes also called *cv-qualifiers*.

Arrays

Arrays are declared by using a postfix declarator operator consisting of brackets surrounding an optional constant (or a constant expression). Here are some examples:

```
double pt[3];
double coords[] = { 1.0, 1.2 }; // coords[2] deduced
int const vals[2] = { 0, 1 };
```

Although you can have arrays of `const` values, there is no syntax to apply `const` to the declarator operator itself. For example:

```
int a[2] const; // Error: cannot const-qualify arrays
```

Just as we can have pointers to pointers, we can declare arrays of arrays (but not references to references!):

```
int *const *pp; // a pointer to a const
                // pointer to an int
int aa[10][20]; // an array of 10 arrays of 20 ints
int &&rr; // Error!
```

The declaration of `aa` illustrates that the left brackets '`[10]`' apply first—that is, `aa` is an array of 10 items, not an array of 20 items. On the other hand, for prefix declarator operators, the right-most one applies first: `pp` is a non-`const` pointer.

Different declarator operators can be mixed:

```
int *ap[10]; // Array of pointers
```

You might wonder: Is this an array of pointers, or a pointer to an array? The former is true. Postfix declarator operators always take precedence over prefix operators. To override this order, we can always use parentheses:

```
int (*pa)[10]; // Pointer to an array
```

I recommend staying away from using a single base-type with multiple but different declarator operators. For example:

```
double const pi = 3.141592654,
             *ppi = &pi; // Easily confusing
```

Instead, separate this as

```
double const pi = 3.141592654;
double const *ppi = &pi;
```

Functions

The syntax for declaring a function (§7) without defining it is similar to that of declaring an array; the postfix bracket operator is replaced by a parenthesized list of parameters:

```
int f(int, double); // Declarator operator
                    // is (int, double)
```

The type underlying such a declarator operator is the return type of the function being declared. Remember that postfix operators take precedence over prefix operators:

```
int* f(int, double); // Function returning
                     // a pointer to an int
int (*pf)(int, double); // Pointer to a function
                        // returning an int
```

Function declarations can easily become unwieldy, as shown in the following example:

```
int (*g(int))[10]; // Function returning a pointer
                   // to an array of 10 integers
```

Therefore, it is easier to declare such functions using intermediate *typedef* declarations (see also Exercises 5.1 and 5.3):

```
typedef int IntA10[10];
IntA10* g(int);
```

A typedef declaration is a way to specify that the declared name is not the name of an object, reference, or function, but just a synonym for the type underlying the declaration.

In principle, we can declare multiple functions as follows:

```
void fun1(int), fun2(double, int); // Confusing
```

This is not recommended.

Unlike arrays, some functions can be qualified with `const` and/or `volatile`. However, only nonstatic member functions (§10.2.1, §10.2.4, §10.2.6) can be declared that way. When declaring a function without defining it, the names of the parameters can be omitted:

```
int f(int, double); // Unnamed parameters
int g(int n, double d); // Named parameters
```

However, it is a good idea to name those parameters whose meaning may not be obvious. For example,

```
void copy_chars(char *dest, char const *src, size_t);
```

Be careful to distinguish the concepts of parameters (sometimes called *formals*) and arguments (sometimes called *actuals*); the latter substitute the former when calling a function. Function declarations that are definitions have more restrictions—for example, only one function can be defined at the time.

Special "specifiers" are applicable to functions. For example, the specifier `inline` is a hint for the C++ compiler that the function should be substituted inline wherever it is called, instead of using the usual function-call mechanism (§7.1.1).

Class Definitions

Classes (§10) can be defined equivalently using the keyword `class` and the keyword `struct`. The two are exactly equivalent, except that the default access protection for members and bases of a class defined with `class` is `private`, while with `struct` it is `public`. Hence, the following two definitions of `Stack` are esentially identical:

```
class Stack {
public:
   Stack();
   ~Stack();
   void push(StackItem*);
   void pop(int n = 1);
   StackItem& top() const;
private:
   StackNode *top_;
};

struct Stack {
   Stack();
   ~Stack();
   void push(StackItem*);
   void pop(int n = 1);
   StackItem& top() const;
private:
   StackNode *top_;
};
```

Member functions of a class can be defined inside the class (§10.2.9). Doing this implies that the member function is `inline`. For example, the definition

```
struct Array {
   int size() const { return s_; }
   // ...
};
```

is equivalent to

```
struct Array {
   inline int size() const;
   // ...
};
```

```
inline int Array::size() {
   return s_;
}
```

Chapters 10, 15, and 25 contain a large variety of exercises relating to classes. Because the class concept is so central to C++, you will also find many exercises dealing with classes in the other chapters.

Overloading

C++ allows you to *overload* function names—the same name can be used for multiple functions. These functions must differ in their number of parameters and/or the types of some of these parameters. When the function name is used, the particular function selected is the one that best matches the argument types. The exact rules for "best match" are numerous and complex but will fortunately match intuition in the majority of cases.

```
f(int); // (1)
f(int, long); // (2)
f(long, long, int = 0); // (3)
f(int, int, long = 0L); // (4)

f(0, 0); // Matches (4) best
f(0, 0, 0); // Ambiguous: (3) or (4)?
```

The one principle to remember is that the selected function cannot have a parameter that has a worse match than the corresponding parameter in another callable function. For example, '`f(0, 0, 0)`' has three (signed) `int` arguments and, therefore, matches two out of three parameters of declaration (4) above. This may seem a better match than candidate (3), which only matches exactly in the third argument while the first two require an implicit promotion of `int` to `long`. Yet because declaration (3) has a better match than (4) in that third argument, declaration (4) cannot be selected as the best match.

The detailed rules for what is a better match in a single argument are a little unwieldy, but, in general, the following rules hold:

- An *exact match* is better than a trivial adjustment.
- A trivial adjustment (for example, adding `const` qualification and/or decaying an array to a pointer to its first element, §5.3) is preferred over an *integral promotion*.
- Integral promotions—such as conversions from `char` to `int`—beat other *standard conversions*.
- These standard conversions are a better match than *user-defined conversions*.
- Matching "ellipsis" arguments (§7.6) is the worst kind of match.

See §7.4 and §11 for additional details.

Operator Precedence

If you do not have parentheses in an expression, certain operators take precedence over others. For example, arithmetic operators (add, multiply, divide, and so forth) follow the usual rules of mathematics. This implies that

```
int a = 2+5*8;
```

is really equivalent to

```
int a = 2+(5*8); // Not (2+5)*8
```

Because of the richness of C++'s set of operators, it is convenient to reproduce the table of operators grouped by precedence, shown in §6.2. Each box in the table holds operators with the same precedence. Unary operators and assignment operators are right associative. All others are left-associative. Table 2.1 shows the table of operators in decreasing order of precedence.

Table 2.1 *Operators in decreasing order of precedence*

Name	***Syntax***
scope resolution	class-name **::** member-name namespace-name **::** member-name **::** global-name **::** qualified-name
member selection	object **.** member-name pointer **->** member-name
subscripting	pointer **[** expression **]**
function call	function **(** expression-list **)**
value construction	type-name **(** expression-list **)**
post increment/decrement	lvalue**++** lvalue**--**
type identification	**typeid (** type-name **)**
run-time type identification	**typeid (** expression **)**
run-time checked conversion	**dynamic_cast <** type-name **> (** expression **)**
compile-time checked conversion	**static_cast <** type-name **> (** expression **)**
unchecked conversion	**reinterpret_cast <** type-name **> (** expression **)**
const/volatile conversion	**const_cast <** type-name **> (** expression **)**

Continued

Table 2.1 *Operators in decreasing order of precedence (Continued)*

Name	***Syntax***
size of type or object	**sizeof** (type) **sizeof** expression
pre increment/decrement	++lvalue --lvalue
binary (bit) complement	~ expression
logical not	!expression **not** expression
unary plus and minus	+expression -expression
address of	&lvalue
dereference	*expression
create (allocate)	**new** type-name
create (allocate and initialize)	**new** type-name (expression-list)
create (place)	**new** (expression-list) type-name
create (place and initialize)	**new** (expression-list) type-name (expression-list)
destroy single object or array	**delete** pointer **delete** [ ] pointer
cast (type conversion)	(type-name) expression
member selection	object .* pointer-to-member pointer ->* pointer-to-member
multiply divide modulo (remainder after division)	expression * expression expression / expression expression % expression
add (plus) subtract (minus)	expression + expression expression - expression
shift left/right	expression << expression expression >> expression
less than, less than or equal greater than, greater than or equal	expression < expression expression <= expression expression > expression expression >= expression
equal, not equal	expression == expression expression != expression
bitwise and	expresson & expression
bitwise exclusive or	expression ^ expression
bitwise inclusive or	expression | expression
logical and	expression && expression expression **and** expression
logical inclusive or	expression || expression expression **or** expression

Continued

Table 2.1 *Operators in decreasing order of precedence (Continued)*

Name	***Syntax***	
assignment	lvalue = expression	
computed assignment	lvalue *= expression	lvalue /= expression
	lvalue %= expression	
	lvalue += expression	lvalue -= expression
	lvalue <<= expression	lvalue >>= expression
	lvalue &= expression	lvalue ^= expression
	lvalue |= expression	
conditional expression	expression ? expression : expression	
throw exception	**throw** expression	
comma (sequencing)	expression , expression	

Chapter 3

C++ Evolution and Compatibility

The C++ programming language evolved with the needs of its user community and the capabilities of its supporting technology. C++ in 1997—the year in which this text was written and the year the last substantial modifications were made to the language definition—is not the same as C++ was in 1990, which in turn was different from C++ in 1985.

This implies that for a short period from the time this is written, compilers that do not completely implement the C++ language will be commonplace. Similarly, many systems will implement certain language features slightly differently than the standard definition. This should not alarm you; overall, the changes are rarely significant. Nonetheless, if you are working with a pre-ISO C++ compiler, you should find this chapter useful in working around the small incompatibilities you may encounter. If your compiler *is* fully compliant with the C++ standard, this chapter may still be useful to understand why legacy code contains constructs that seem anomalous in light of the standard.

The remainder of this chapter lists a selection of issues that arise when porting C++ from or to pre-ISO C++ compilers. The presentation does not attempt to be complete, but instead focuses on those items that may apply to the sample solutions that appear in this book.

More information about the evolution of the language can be found in §B and *The Design and Evolution of C++* by Bjarne Stroustrup.

Standard Headers

Most C++ programs use the C++ standard library, which includes an extension of the C standard library (§3, §16). To enable this use, we must make declarations of items in these libraries visible in our programs. That, in turn, is achieved by `#include`-ing *standard headers*. For example,

```
#include <iostream>
#include <cstdio>
#include <math.h>
```

```
int main() {
   std::printf("Hello,\n");
   std::cout << "The natural logarithm of 10 is"
             << log(10.0) << '\n';
   return 0;
}
```

This little program illustrates three kinds of header name. The first header, `<iostream>`, brings declaration of the C++ standard library into the *namespace* `std` (explained below). The second header, `<cstdio>`, does something similar—for example, for `std::printf`—but the leading letter *c* indicates that this header brings in declarations inherited from the C standard library. In C and in earlier versions of C++ however, all header names ended with the suffix '.h.' To improve the embedding of C code in C++ programs, C++ provides headers with the traditional names used for the C standard library. Hence, the header `<stdio.h>` corresponds to the header `<cstdio>` used in the preceding program, and the header `<math.h>` has a counterpart, `<cmath>`. The difference between the headers with a '.h' suffix and their counterparts with a 'c' prefix is that the former not only brings name in the `std` namespace, but also in the global namespace. If your C++ system does not yet provide the version of the headers with a 'c' prefix, you can always use the '.h' versions. They have always been in the C++ language, but older versions brought only the C standard library names in the global namespace and not in namespace `std`.

C++ specific headers do not have '.h' suffixes, but in the early days of C++, they did. For example, `<iostream>` used to be called `<iostream.h>` and would bring names in the global namespace. Again, if your system still adheres to the older conventions, you can use those in the sample solutions presented in this book and remove the references to namespace `std` (see the next section).

Namespaces

A namespace (§8.2) is a C++ feature that allows you to group a logically coherent set of C++ classes and functions in a sort of module. This modularization is useful to diminish the risk of unwanted interactions (that is, "name clashes") between two or more independent software components.

If your system does not support namespaces, you will still be able to make use of almost all solutions presented in this book, with a few minor modifications. Of these modifications, the majority relate to the use of the standard library namespace `std`. Using a name from a namespace, such as `std`, can be achieved in four ways. The first way consists of simply putting the use in the namespace where the name is declared:

```
namespace MyNames {
   void f(int); // Declaration of f
   void g() {
      f(3); // Use of MyNames::f(int)
   }
}
```

Of course, this first use is only a generalization of how names are looked up in C++ and does not illustrate why namespaces were introduced. The second mechanism to select a name from a namespace prefixes the name with the namespace qualifier. Here is an example:

```
namespace MyNames {
   void f(int);
}

namespace YourNames {
   void f(int);
}

void g() {
   YourNames::f(1);
   MyNames::f(2);
}
```

Thus, a rather straightforward mechanism permits us to have identical names coexist and cooperate in the same program. Explicitly qualifying namespace names in this way is, in principle, sufficient for any sort of namespace use. However, the designers of C++ recognized that such repeated qualifications can become unwieldy. Hence, if you find that you use the same qualified name many times, you can use a *using-declaration*:

```
#include <iostream>

int main() {
   using std::cout; // A using-declaration
   cout << "Hello, ";
   cout << "Earth!\n";
   return 0;
}
```

This example involves the standard namespace `std`. The name `cout` is brought in the scope of the function `main` with a using-declaration, after which it no longer requires qualification.

The final way to get at a name in a namespace is to use a *using-directive*. Unlike using-declarations, using-directives do not bring namespace names into a given scope, but instead direct the name lookup mechanism of C++ to consider additional namespaces. For example,

```
#include <iostream>

int main() {
   using namespace std; // A using directive
   cout << "Hello, ";
   cout << "again.";
   return 0;
}
```

Using-directives must be used with care because they can make a great number of—often undocumented—names visible, thereby increasing the risk of a conflict between identical names in different namespaces.

If your C++ compiler does not yet support namespaces, names used in this book will instead have to be global (for example, names normally occuring in `std` will have been made global by your vendor). It is, therefore, usually sufficient to drop the explicit namespace qualification, using-declaration, or using-directive from the sample programs in this book. If doing so were to cause identical names to conflict, you would also need to modify at least one of these names to ensure uniqueness. See Exercise 16.1 for an example involving the adaptation of a program to an older C++ program.

The bool Type

At the time of this writing, there still exist a few popular C++ compilers that do not support the `bool` type and the associated `true` and `false` literals. When C++ code that uses the `bool` type must be ported to these compilers, you will need to introduce a suitable substitute. The two popular alternatives are

```
typedef int bool;
int const false = 0;
int const true = 1;
```

and

```
enum bool { false = 0, true = 1 };
```

The former solution better models the close connection between `bool` and integers, but the second alternative allows functions to be overloaded for bool arguments (that is, if the compiler supports overloading on enumeration types).

You may also find a few programs that were unaware of this type in C++ and that defined the names `bool`, `false`, or `true` for their own purposes. For example, one popular C library used in C++ programs declares

```
typedef char bool; // Error! Cannot typedef keyword.
```

Such programs are easily made ISO-compliant by replacing the occurences of `bool`, `true`, and `false` by alternative names (for example, `Bool`, `kTrue` and `kFalse`).

The introduction of a few other keywords (for example, `export`) and reserved words (for example, `compl`) also increases the potential for clashing with user-defined names, but those cases are much rarer.

Alternative Tokens

Some characters that can have a special meaning in C++ (such as '&') can be hard to produce with certain non-U.S. keyboards. To ease C++ programming on such systems, alternative tokens were introduced. Table 3.1 is taken from §C.3.1.

Table 3.1 *Alternative tokens*

Keywords		***Digraphs***		***Trigraphs***	
`and`	`&&`	`<%`	`{`	`??=`	`#`
`and_eq`	`&=`	`%>`	`}`	`??(`	`[`
`bitand`	`&`	`<:`	`[`	`??<`	`{`
`bitor`	`|`	`:>`	`]`	`??/`	`\`
`compl`	`~`	`%:`	`#`	`??)`	`]`
`not`	`!`	`%:%:`	`##`	`??>`	`}`
`or`	`||`			`??'`	`^`
`or_eq`	`|=`			`??!`	`|`
`xor`	`^`			`??-`	`~`
`xor_eq`	`^=`				
`not_eq`	`!=`				

In most cases, this book uses the original tokens. Hence, if your system does not allow for the required characters, you will need to use one of the alternative tokens from Table 3.1. However, for the logical operators '&&', '||' and '!,' I find the alternatives '`and`', '`or`,' and '`not`' more readable and do not hesitate to use them on occasion. If your system does not support these alternative tokens, you will need to replace them with the traditional symbols or by trigraphs.

Templates

Hint: If you are totally unfamiliar with C++ templates, consider skipping this section. Do not hesitate, however, to read through §2.7 and §13 to learn about this important element of C++. Athough C++ templates are often presented as an advanced topic, they are, in fact, a fundamental software building block and form a cornerstone of the C++ standard library.

Templates are one of the most powerful, but also one of the more complex, concepts in C++. Furthermore, templates have evolved rapidly to meet the needs of software library designers and also to give precise meaning to every template construct.

Templates allow you to create generic code by parameterizing classes, functions, and member functions. For example, the simple class template:

```
template<typename T, typename U = T>
struct Pair {
   T first_;
   U second_;
};
```

has two *type parameters,* T and U. This means you can subsitute these parameters to obtain actual classes:

```
Pair<char, int> a;
Pair<int> b; // equivalent to 'Pair<int, int> a;'
```

Note how class template parameters can have *default template arguments*. These default arguments are not supported by certain older C++ implementations. Furthermore, they are not allowed for function and member function templates.

The keyword `typename` above can be replaced by the keyword `class` without any change in meaning. In particular, the keyword `class` in this context *does not imply* that only class types can be substituted; built-in types, pointer types, enumeration types, and so forth are acceptable, as well. Note also that some compilers will *only* accept the keyword `class` here.

The keyword `typename` has another important use discussed in Exercise 14.1 and §13.5.

A class, function, or member function obtained by substituting all the parameters of a template is called a *specialization*. Usually specializations are generated automatically from the template, but it is possible to override this by declaring and defining a *explicit specialization*:

```
template<> struct Pair<bool, bool>; // declaration

template<>
struct Pair<bool, bool> { // definition
   bool first_: 1; (bitfields, §C.8.1)
   bool second_: 1;
};
```

The declaration of an explicit specialization is sufficient to prevent a compiler from attempting to generate a specialization from the generic form, but a definition is, of course, needed if the compiler needs to know the size or the internal structure of the specialization. Although C++ mandates that explicit specializations be prefixed by `template<>`, some compilers expect no prefix at all. For example:

```
struct Pair<bool, bool>; // old-style explicit
                         // specialization
```

Three kinds of template parameters exist:

1. Template type parameters (§13.2.3)
2. Template non-type parameters (§13.2.3)
3. Template template parameters (§C.13.3)

The first kind—illustrated in the `Pair` template above—is by far the most commonly used and usually has the best support in older compilers. They are used in many exercises throughout this book. Template non-type parameters are illustrated in the rather advanced discussion of Exercise 14.10 and the slightly simpler presentation of Exercise 25.7. At the time of this writing, however, template template parameters are only available in experimental compilers (not generally available) and are not discussed further in this book.

Function templates are also possible and common. For example,

```
template<class T>
T max(T const &a, T const &b) {
   return a<b? b: a;
}
```

When using a function template, the template arguments are often automatically *deduced* (§13.3.1, §C.13.4) from the function call arguments. For example:

```
int a = 3, b = -7;
int c = max(a, b);
// equivalent to: int c = max<int>(a, b)
```

In fact, older C++ compilers will not accept the explicit specification of function template arguments and, therefore, require that all template arguments be deducible from the function call arguments.

Member function templates and nested class templates are also possible but, again, are not always implemented by 1997 compilers:

```
struct Outer {
   template<class T> // or: template<typename T>
   class Inner {
      // ...
   };
   template<class T>
   void f(T);
};
```

These concepts are often useful and are used throughout the C++ standard library.

Template Instantiation

The process of creating a specialization from the generic description of a template is called *template instantiation*. Any kind of template can be instantiated but the creation of specializations that take up storage at run-time (code and data) is the most delicate for C++ implementations. These include instantiations of:

- Function templates
- Member function template
- Static data members of class templates

The difficulty with these entities is that their nontemplate counterparts can be defined only in a single translation unit—violating this aspect of the *one definition rule* (ODR, §9.2.3) usually results in a link-time error. However, instantiations for the same specialization may be required by multiple translation units.

Various C++ implementations have chosen different solutions to this problem, but they can generally be classified in one of the following three categories:

- *Greedy instantiation*. A number of C++ compilers will instantiate template specializations in every translation unit in which they are needed. These compilers then work with a linker that is able to recognize multiple instantiations of the same specialization and eliminate all but one of them.

  This approach to template instantiation is flexible in practice and fits well in the traditional model that produces exactly one object file for every translation unit (see also Exercise 4.1). The downside is that a specialization may need to be processed and optimized several times, thereby increasing the overall build-time for a given software product.

- *Queried instantiation*. An elegant approach to template instantiation is to associate a database of already available specializations with the compiler. When the compiler encounters the need for a specialization, it queries the database to determine whether the specialization must be instantiated or whether it is already available in the database.

  Queried instantiation ensures that every specialization is processed only once. However, it requires some adaptation in the organization and distribution of source files and software libraries.

- *Iterated instantiation*. This instantiation policy delays the production of specializations until link-time. At that time, the linker or a pre-linker determines which specializations are required, reinvokes the C++ compiler on selected source files, and directs it to produce the needed specializations. Because the production of a specialization can reveal the need for additional instantiations, this process must be iterated until closure is achieved.

  Several variations on this instantiation method exist. It has the advantage of being able to exploit traditional linkers, and it is possible to keep a model in which each

translation unit corresponds to exactly one object file. However, the need to repeat compilations can lead to severe degradation of build times.

C++ provides syntax called an *explicit instantiation* directive:

```
template int max(int, int);
```

This indicates that an instantiation is requested at that point. It does not, however, guarantee that the instantiation will be associated with that translation unit. For example, in an implementation using queried instantiation, this directive might be ignored if the database already contains the requested specialization.

All the members of a given class can be instantiated with an explicit instantiation directive of the following form:

```
template struct Pair<double, int>;
```

A few older compilers do not support this syntax but achieve the same effect with certain typedef declarations:

```
typedef Pair<double, int> anyname;
```

However, standard C++ does not consider that a typedef declaration creates a need for instantiation. In fact, standard C++ does not create points of instantiations unless really needed. Consider the following example:

```
template<class T>
class Example {
   T data_;
public:
   bool is_large() const
      { return sizeof(T)>sizeof(double); }
   void dereference()
      { *data_; }
};

int main() {
   Example<int> a;
   return a.is_large();
}
```

Notice that the member `Example<int>::dereference` cannot be instantiated because an `int` cannot be dereferenced (that is, `*data_` would be illegal if `data_` has type `int`). Some implementations will nevertheless attempt instantiation of every member of a class specialization as soon as that specialization is used—if any of these instantiations fail, the compilation is interrupted. Sometimes, this is only a problem for inline members.

A (not quite satisfactory) workaround for compilers that instantiate members too hastily is to provide nonmember functions instead:

```
template<class T> class Example;

template<class T>
void dereference(Example<T> &e) {
   *e.data_;
}

template<class T>
class Example {
   friend void dereference(Example<T>&);
   T data_;
public:
   bool is_large() const
      { return sizeof(T)>sizeof(double); }
};

int main() {
   Example<int> a;
   return a.is_large();
}
```

Fortunately, C++ implementations are rapidly catching on with the C++ standard and these contortions are becoming curiosities of the past.

Chapter 4

Types and Declarations

A C++ *declaration* (§4.9) introduces or reintroduces a name in a program. At the same time, it provides at least a partial description of what the name refers to (for example, "the name `Color` refers to the name of a class"). A *type* (§4.1) is a set of properties that turns a set of raw bits into an *object*. Although C++ includes a collection of built-in types, it also provides a rich set of mechanisms to extend the type system for your specific purposes. Making the most of these constructs is the key to successful software development in C++.

Chapter 2 summarizes some essential elements of the anatomy of a declaration. Although mastering the complete declaration syntax and type system of C++ takes some effort, much useful work can be achieved by understanding the basic principles illustrated in this chapter.

Exercise 4.1

Get the "Hello, world!" program (§3.2) to run. If that program doesn't compile as written, look at §B.3.1.

Hint: This exercise is where everything begins. Its purpose is to get you to familiarize yourself with your C++ implementation.

The process of turning the human-readable text of a C++ program into executable software is highly dependent on the environment in which the software is being developed.

In almost all cases, the first step consists of creating a file with the text of the program. Next, this *source file* must somehow be fed to a C++ translation system. Usually, this system is built around a *compiler*—a piece of software that translates the C++ program text file into a form more readily executable by the target machine. However, a few so-called *C++ interpreters* exist; they take a given source file and *input* for the associated program and directly produce the desired output.

If a compiler is used, performing the translation may be as simple as issuing a command similar to

```
CC helloworld.C
```

where `helloworld.C` is the name of the source file (and `CC` is the name of this particular C++ compiler). This command will typically perform two tasks:

1. Translate the source file into an *object file* (compilation)
2. Link the resulting object file with the software libraries it relies on (in this case, the standard C++ library)

The result is executable software stored in a file whose name will depend on the platform in use. Options to change the name of the executable file are also available.

More and more commonly, compilers are integrated in environments endowed with rich graphical interfaces. Such environments may be helpful in the development of more complex software, but they may also require additional learning.

Many difficulties—most of which are not specific to C++—can crop up when you try to run your first few programs. First, you'll need a correct installation of the translation system. Even if the system you used had such a system in place, you might find that installation was not flawless, or that the installed configuration does not correspond to your documentation. Next you may have to fight little quirks of the translation software. A common one is the requirement that the name of source files end with a particular "suffix" (`.C` and `.cpp` are not unusual). Finally, you might find that your C++ compiler vendor does not yet fully implement the language (for example, lack of namespace support, incomplete template facilities, and so forth). See Chapter 3 for discussions about working around some of these problems.

However, the time you invest early on in gaining familiarity with your development software is likely to pay for your productivity and the quality of your work as you move to larger projects. To further ease this learning process, seeking advise from your friends and colleagues is also invaluable.

Exercise 4.2

For each of the declarations in §4.9, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.

Hint: This is a fundamental exercise illustrating the distinction between *declarations* and *definitions*. A short summary of basic declaration syntax is given in Chapter 2.

The list of declarations that are not definitions is smaller:

```
extern int error_number;
double sqrt(double);
struct User;
```

For objects (variables), a declaration is a definition when it is not preceded by the keyword `extern` or when it is followed by an initializer (a '=' symbol followed by an expression, or a function-style initializer). For example:

```
int i; // Definition: no extern keyword
extern complex<double> z = 3.0; // Definition: initializer
extern complex<double> w(1.0, 1.0);
   // Definition: function-style initializer
extern double score; // Declaration only
```

Function-declarations are definitions if they have a body. For example:

```
double sqrt(double); // Declaration only
double dabs(double x) { // Definition
   return x<0.0? -x: x;
}
```

A `class`, `struct`, `union`, or `enum` is a definition if its declaration includes a declaration of all its members. Hence,

```
class Unknown;
```

is only a declaration, whereas

```
class Pixel {
   int x_, y_, color_;
};
```

is a definition.

Class templates and function templates essentially follow the rules of classes and functions. These few rules explain most of the following definitions:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
char *name = "Njal";
char *season[] = {"spring", "summer", "fall", "winter"};
struct Date { int d, m, y; };
int day(Date *p) { return p->d; }
template<class T> T abs(T a) { return a<0? -a: a; }
typedef complex<short> Point;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

If you are avid for trivia and anecdotes, you may like to hunt for the curious references sprinkled throughout Stroustrup's book. There you will find the hero of an Icelandic saga (Njal) and a triplet of Danish beers (Carlsberg, Tuborg, and Thor).

Namespace definitions are syntactically somewhat similar to class definitions, but not all members must be declared. For example:

```
namespace N {
} // Definition: other members of N may be declared elsewhere
```

It is not possible to declare a namespace without making it a definition.

Armed with this knowledge, changing the definition above in declarations only, and vice versa, is relatively straightforward (but not always possible). First, the plain declarations that are no longer definitions:

```
extern char ch;
extern string s;
extern int count;
extern const double pi;
extern char *name;
extern char *season[];
struct Date;
int day(Date*);
template<class T> T abs(T a);
typedef Date Point;
enum Beer;
```

And here are the definitions:

```
int error_number;
double sqrt(double) {
   return 0.0;
}

struct User {
};
```

As explained earlier, it is not possible to declare a namespace without defining one.

Exercise 4.3

Write a program that prints the sizes of the fundamental types, a few pointer types, and a few enumerations of your choice. Use the `sizeof` operator.

Hint: The use of `sizeof` is fairly fundamental in C++. Remember, non-`char` types do not have the same size on every C++ implementation. See also Exercises 4.5, 4.7, and 5.2.

The key to this exercise is to find all the fundamental types, as well as interesting variations on pointer types and enumeration types. Let me first present a straightforward solution that essentially consists of a series of output operations:

```
#include <iostream>
// Omit the following using-declaration if your compiler
// doesn't place cout in namespace std:
using std::cout;

struct Polymorph {
   virtual ~Polymorph() {}
};

enum Bit { zero, one };
enum Intensity { black = 0, brightest = 1000 };

int main() {
   // Fundamental integral types:
   cout << "sizeof(bool)==" << sizeof(bool) << '\n';
      // sizeof(bool) may not be one (but it can be).
   cout << "sizeof(char)==" << sizeof(char) << '\n';
   cout << "sizeof(signed char)=="
        << sizeof(signed char) << '\n';
   cout << "sizeof(unsigned char)=="
        << sizeof(unsigned char) << '\n';
   cout << "sizeof(wchar_t)==" << sizeof(wchar_t) << '\n';
      // There are no signed/unsigned variants of wchar_t.
   cout << "sizeof(signed short)=="
        << sizeof(signed short) << '\n';
   cout << "sizeof(unsigned short)=="
        << sizeof(unsigned short) << '\n';
   cout << "sizeof(signed int)=="
        << sizeof(signed int) << '\n';
   cout << "sizeof(unsigned int)=="
        << sizeof(unsigned int) << '\n';
   cout << "sizeof(signed long)=="
        << sizeof(signed long) << '\n';
   cout << "sizeof(unsigned long)=="
        << sizeof(unsigned long) << '\n';
#ifdef LONGLONG_EXT
   cout << "sizeof(signed long long)=="
        << sizeof(signed long long) << '\n';
   cout << "sizeof(unsigned long long)=="
        << sizeof(unsigned long long) << '\n';
#endif
```

```
    // Fundamental floating-point types:
    cout << "sizeof(float)==" << sizeof(float) << '\n';
    cout << "sizeof(double)==" << sizeof(double) << '\n';
    cout << "sizeof(long double)=="
         << sizeof(long double) << '\n';

    // Pointer types:
    cout << "sizeof(int*)==" << sizeof(int*) << '\n';
    cout << "sizeof(int (*)())=="
         << sizeof(int (*)()) << '\n';
       // The above corresponds to a pointer to a function.
       // Next is a pointer to a member function:
    cout << "sizeof(void (Polymorph::*)())=="
         << sizeof(void (Polymorph::*)()) << '\n';
    cout << "sizeof(void*)==" << sizeof(void*) << '\n';

    // Enumeration types:
    cout << "sizeof(Bit)==" << sizeof(Bit) << '\n';
    cout << "sizeof(Intensity)=="
         << sizeof(Intensity) << '\n';
    return 0;
}
```

To ensure that I didn't forget any fundamental type, I looked them up in the working paper of the ISO C++ standardization committee (WG21); the subsection "Simple type specifiers" contains an exhaustive table. Because many compilers now provide an additional type `long long` as an extension (likely to be standardized for the C language), I also added a line to support such types.

Also note the line dealing with a *pointer-to-member function*. This sort of pointer usually requires some extra storage when pointing to a member of a polymorphic class (a class with virtual members) or to a member of a class with virtual bases. Indeed, such a pointer need not only keep information about which member is indicated, but also for which derived type the member should be looked up.

Here is the output produced by a compiler that I occasionally use:

```
sizeof(bool)==1
sizeof(char)==1
sizeof(signed char)==1
sizeof(unsigned char)==1
sizeof(wchar_t)==2
sizeof(signed short)==2
sizeof(unsigned short)==2
sizeof(signed int)==4
sizeof(unsigned int)==4
sizeof(signed long)==4
sizeof(unsigned long)==4
```

```
sizeof(signed long long)==8
sizeof(unsigned long long)==8
sizeof(float)==4
sizeof(double)==8
sizeof(long double)==10
sizeof(int*)==4
sizeof(int (*)())==4
sizeof(void (Polymorph::*)())==12
sizeof(void*)==4
sizeof(Bit)==4
sizeof(Intensity)==4
```

Your favorite compiler may give you different results.

For the sake of experimentation, let's develop a somewhat more sophisticated version of the above program. This one uses the `typeid` operator (§15.4.4), which returns a reference to a `type_info` object that contains some information about the argument type. It is not guaranteed that the `name()` member of this object returns a meaningful value, but in practice the returned string (actually a `char const*`) describes the type accurately:

```
#include <iostream>
#include <typeinfo>

template<typename T>
struct Type {
   static void print() {
      std::cout << "sizeof(" << typeid(T).name() << ") = "
                << sizeof(T) << std::endl;
   }
};

struct Polymorph {
   virtual ~Polymorph() {}
};

enum Bit { zero, one };
enum Intensity { black = 0, brightest = 1000 };

int main() {
   // Fundamental integral types:
   Type<bool>::print(); // May not be one (but can be).
   Type<char>::print(); // This must be one!
   Type<signed char>::print();
   Type<unsigned char>::print();
   Type<wchar_t>::print(); // No signed/unsigned variants.
   Type<signed short>::print();
   Type<unsigned short>::print();
   Type<signed int>::print();
```

```
    Type<unsigned int>::print();
    Type<signed long>::print();
    Type<unsigned long>::print();
#ifdef LONGLONG_EXT
    Type<signed long long>::print();
    Type<unsigned long long>::print();
#endif

    // Fundamental floating-point types:
    Type<float>::print();
    Type<double>::print();
    Type<long double>::print();

    // Pointer types:
    Type<int*>::print();
    Type<int (*)()>::print(); // Pointer to function
      // Pointer to member function:
    Type<void (Polymorph::*)()>::print();
    Type<void*>::print(); // May be larger than int* (rare)

    // Enumeration types:
    Type<Bit>::print();
    Type<Intensity>::print();
    return 0;
}
```

This program may need to be adapted for compilers that do not support namespaces and will need many more modifications if the `typeid` operator is not supported.

Exercise 4.4

Write a program that prints out the letters* a, ..., z *and the digits 0, ..., 9, and their integer values. Do the same for other printable characters. Do the same again, but use hexadecimal notation.

Hint: More portable code is obtained by not assuming universal values for any character. In fact, it is not even universally true that alphabetically successive letters have successive values, although it is assumed in some solutions (for example, Exercise 15.3).

The following program is fairly self-explanatory. It uses a character string to select the characters to be displayed:

```
#include <iostream>

char const char_table[] =
   "abcdefghijklmnopqrstuvwxyz0123456789-*&@. \~";
```

```
int main() {
   // Do not include the terminating null of chartable!
   // Decimal first:
   for (int k = 0; k<sizeof(char_table)-1; ++k)
      std::cout << char_table[k] << '\t'
                << dec << int(char_table[k]) << std::endl;
    // Or, since dec is the default:
    for (int k = 0; k<sizeof(char_table)-1; ++k)
       std::cout << char_table[k] << '\t'
                 << int(char_table[k]) << std::endl;
   // Hexadecimal:
   for (int k = 0; k<sizeof(char_table); ++k)
      std::cout << char_table[k] << '\t'
                << hex << int(char_table[k]) << std::endl;
   return 0;
}
```

As already explained, you may need to adapt this if your compiler does not place the standard library in namespace `std` (by omitting `std::`). You may also find that your compiler requires you to use the older `<iostream.h>` header name. See Chapter 2 for suggestions.

Exercise 4.5

What, on your system, are the largest and the smallest values of the following types: `char`, `short`, `int`, `long`, `float`, `double`, `long double`, and `unsigned`?

Hint: Consider reading §22.2 before trying this fundamental exercise. These limits determine when overflow and underflow occur. Exercise 8.10 also illustrates this.

If you have a hard time finding this in your system's documentation, you can make use of the `numeric_limits` template (§22.2) provided by the standard library header `<limits>`. Here is a little program that produces the answer to this question:

```
#include <iostream>
#include <limits>
#include <typeinfo>

template<typename T>
struct Type {
   static void print() {
      std::cout << typeid(T).name() << ": range is ("
                << std::numeric_limits<T>::min() << ", "
                << std::numeric_limits<T>::max() << ")\n";
   }
};
```

```
int main() {
   Type<char>::print();
   Type<short>::print();
   Type<int>::print();
   Type<long>::print();
   Type<float>::print();
   Type<double>::print();
   Type<long double>::print();
   Type<unsigned>::print();
   return 0;
}
```

With one compiler, this produces

```
char: range is (é, ⌂)
short: range is (-32768, 32767)
int: range is (-2147483648, 2147483647)
long: range is (-2147483648, 2147483647)
float: range is (1.17549e-38, 3.40282e+38)
double: range is (2.22507e-308, 1.79769e+308)
long double: range is (3.3621e-4932, 1.18973e+4932)
unsigned: range is (0, 4294967295)
```

Note the limited precision on floating-point types and the somewhat strange representation of the character range. The latter range could be cast to another integral type if desired.

Some compilers do not yet provide the `<limits>` header. A less-general alternative is provided by the C language header `<limits.h>`, which provides preprocessor symbols such as `CHAR_MIN` and `UINT_MAX`, which describe the extreme values of fundamental types. The advantage of the C++ specific library is that it can mesh well with your templates. See also Exercise 4.3 and §15.4.4 for the meaning and use of the `typeid` operator. Do these results relate to those of Exercise 4.3?

Exercise 4.6

What is the longest local name you can use in a C++ program on your system? What is the longest external name you can use in a C++ program on your system? Are there any restrictions on the characters you can use in a name?

Hint: You can safely skip this exercise on a first read. Exercise 22.8 demonstrates an advanced technique that can exhibit a phenomenon described in the last part of this exercise.

In general, you will need to glean this information from your system's documentation. The following comments may nonetheless be of some use.

The draft C++ standard recommends in its appendix that at least 1,024 initial characters be allowed in every identifier (external or internal). Identifiers can be composed of the lowercase letters *a* through *z*, the uppercase letters *A* through *Z*, the digits 0 through 9 and the

underscore '_' character; they cannot start with a digit, however. Furthermore, identifiers can contain *universal character names*; these generally correspond to special-purpose symbols and characters or glyphs used in non-English languages.

Some implementations might allow other characters as extensions (for example, the '$' character). Other systems may not be able to distinguish between uppercase and lowercase characters when object files are linked together (which involves resolving external names).

Also, most systems transform (or *mangle*) certain external names that appear in the C++ source to associate them with types. This can be useful to ensure typesafe linkage and allow overloading, for example. This means that if you define a function taking an argument of type `int` in one translation unit and attempt to call that function in another translation unit with two arguments instead, the two units will not successfully link because the external names for the call and for the definition do not match up. However, this also lengthens the identifiers seen by the linker. Therefore, if the linker has, for example, a limit of 4,096 characters per identifier, you might in fact hit this limit with a name that has 3,000 or fewer characters in the source.

If you cannot find these limitations in your system's documentation, you might consider performing a number of experiments to determine them. For example, you may try using an identifier of 32 characters at first. If this is too large, you can repeatedly halve the value until you reach a too-small value. To find a too-large value, you can repeat experiments doubling the identifier length every time. Once you have both a too-large and too-small value, you can halve the interval every time until you find the transition point. Be aware, however, that modern systems are sometimes limited only by the amount of memory available to them. In that case, finding the too-large value may cause a memory resource error.

The following illustration uses advanced programming techniques and may not be of interest if you're working through this material for the first time.

You might think that you will never create names that are so long. However, names might automatically be generated internally to denote template instantiations. Consider, for example, the following piece of code:

```
template<typename T, typename U>
struct Doublify {};
template<int N>
struct Exponential {
   typedef Doublify<typename Exponential<N-1>::LongType,
                    typename Exponential<N-1>::LongType>
           LongType;
};

template<>
struct Exponential<0> {
   typedef double LongType;
};

int f(Exponential<10>::LongType*) {
   return 3;
}
```

Try compiling this source and determine the external name of the function `f`. This may not be possible on every platform, and if it succeeds, the name is likely to take up more than 10,000 characters. The reason is that the typedef `LongType` in `Exponential<10>::LongType` must be expanded to

```
Doublify<Exponential<9>::LongType, Exponential<9>::LongType>
```

to form the external name. In turn, the two embedded `LongType` typedefs must be expanded revealing double the number of `typedef`s. This doubling repeats 10 times until the special case `Exponential<0>::LongType` that expands to `double` is hit.

Exercise 4.7

Draw a graph of the integer and fundamental types where a type points to another type if all values of the first can be represented as values of the second on every standards-conforming implementation. Draw the same graph for the types on your favorite implementation.

Hint: Portable code assumes that both graphs are indentical. See Exercises 4.3 and 4.5.

I draw the first graph, followed by some comments on which types may be subsets of others on various implementations. First a few notes.

There are, in general, no subset-property guarantees between the ranges for unsigned and signed integral types. So these two families will appear as separate connected subgraphs. Also, although type `char` has values that can be represented with the values of either `signed char` or `unsigned char`, the forthcoming standard does not specify which of these two types underlies `char`. Similarly, there is an integral type underlying `wchar_t`, but the standard does not say which type this is.

Floating-point types also form a separate subgraph, and their situation is fairly straightforward: The set of `long double` values encompasses the set of `double` values, which itself contains the set of `float` values.

Because it is not guaranteed that there is a nonzero floating-point value, it is not guaranteed that a `bool` can be fit in the representation of a floating-point type. However, such a guarantee does exist for the other integral types with respect to `bool`.

In Figure 4.1, we omit edges (arrows) from a node A to a node C if A already points to B, which in turn points to C (that is, use transitive closure to find all the relationships).

On a particular implementation, it is likely that more edges can be drawn. For example, in practice, the `long` type often encompasses all the values of the `unsigned short` type. Usually, a similar relation exists between either `long` and `unsigned int`, or between `int` and `unsigned short`. More arrows will also come into `unsigned char`.

In addition, there will be a map between `wchar_t` and some integral type, which in turn can justify several arrows on the graph. Similarly, the plain `char` type will correspond to either the signed or the unsigned `char` type.

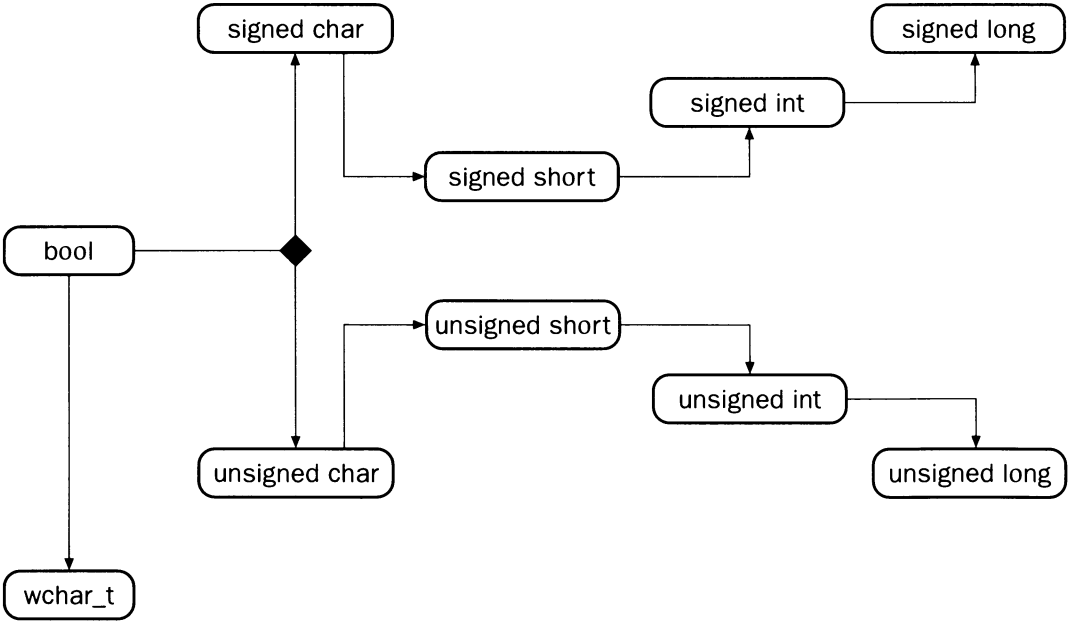

Figure 4.1 *One type points to another if all of its values can be represented with that other type.*

[blank page]

Chapter 5

Pointers, Arrays, and Structures

Pointers, arrays, and structures are the basic glue that C++ inherited from the C language to assemble more complex data structures. To these, C++ added *references* (§5.5). The understanding of the concepts they embody is essential for any nontrivial program development. The exercises in this chapter illustrate the syntax and semantics of these elements, as well as some typical uses and platform-specific issues.

Exercise 5.1

Write declarations for the following: a pointer to a character; an array of 10 integers, a reference to an array of 10 integers, a pointer to an array of character strings, a pointer to a pointer to a character, a constant integer, a pointer to a constant integer, and a constant pointer to an integer. Initialize each one.

Hint: Chapter 2 and §4.9 discuss the structure of declarations and the concept of initialization (as opposed to assignment). Exercises 5.3 and 7.1 complement this fundamental exercise.

You will note that some of these declarations *must* be initialized (for example, references). Here are the declarations in requested order:

```
char *pc = 0;
int ai[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
int (&rai)[10] = ai;
std::string (*pas)[5] = 0;
char **ppc = &pc;
int const ci = 42;
int const *pci = &ci;
int *const = ai+3;
```

Most of these declarations are straightforward. One item to remember is that declarator operators that come on the right of the declared name have higher precedence than those that come on the left of that name, unless parentheses are used. For example,

```
std::string *aps[5];
```

is an array of pointers to strings (array comes before pointers). If you want a pointer to an array of strings, you can use parentheses as in the declaration of `pas`.

Still, when writing C++, I prefer not to force everyone to remember the exact precedence rules of declarator operators. Instead, I use `typedef` declarations to build up the required type. In particular, I prefer to declare `rai` and `pas` as follows:

```
typedef int Array10Int[10];
Array10Int &rai = ai;
typedef std::string Array5String[5];
Array5String *pas = 0;
```

Note that the declaration

```
int const ci = 42;
```

is equivalent to the more traditional

```
const int ci = 42;
```

However, this does not generally apply when declaring `const` pointers or member functions. For example, `int const*` is not a "constant pointer to an integer," but a "pointer to a constant integer" (equivalent to `const int*`, in fact).

I also view the more traditional convention as less consistent when combined with `typedef` declarations or template arguments. For example,

```
typedef Object *Handle;
const Handle x = 0;
```

is equivalent to

```
Object *const x = 0;
```

but not to

```
const Object *x = 0;
```

Ultimately, such minor stylistic issues are of little importance, but they can lead to heated debates—even among experienced programmers.

Exercise 5.2

What, on your system, are the restrictions on the pointer types `char*`, `int*`, and `void*`? For example, may an `int*` have an odd value?

Hint: Alignment is an important concept in systems programming. Exercise 13.2 (which is rather advanced) shows how hard it can be to deal with alignment in generic code.

Typically, a pointer to an object corresponds to a machine representation of a byte address and machine bytes usually match `char`s. (C and C++ define a *byte* to be the basic unit of storage and to be large enough to hold a `char`. It is, therefore, not impossible for a `char` to be larger than a machine byte.) So you might expect the internal values underlying a `char*` to have relatively few limitations. Nevertheless, some architectures may reserve the most significant bits of an address for various reasons—for example, they are sometimes used to indicate whether the address maps to a user program's address space or to the system's address space. Because `char*` pointers can be cast to `void*` and back without loss of information, the restrictions on the machine representation of `void*` and `char*` will often be identical.

On many platforms, the compiler will assume that integers are at addresses that are multiples of their size. For example, on a 32-bit machine, integers will almost always occupy 4 bytes and, therefore, pointers to integers will assume only values that are multiples of 4. This convention allows the compiler to access an integer using a single machine instruction (instead of, for example, using multiple instructions to load the integer one byte at a time).

Many compilers have an option not to assume that integers are aligned at multiples of their sizes. An interesting experiment consists of measuring the run-time of an application with and without that option turned on.

Exercise 5.3

Use `typedef` to define the types unsigned `char`, `const` unsigned `char`, pointer to integer, pointer to pointer to `char`, pointer to arrays of `char`, array of 7 pointers to `int`, pointer to an array of 7 pointers to `int`, and array of 8 arrays of 7 pointers to `int`.

Hint: Compare this to Exercise 5.1. Also try the supplementary exercise proposed below.

This exercise illustrates how the progressive build-up of a type using simple `typedef`s requires much less subtlety than producing the type with a single declaration:

```
typedef unsigned char UnsignedChar;
typedef UnsignedChar const ConstantUnsignedChar;
typedef int* IntegerPointer;
typedef char** PointerPointerChar;
typedef char* PointerArrayChar;
typedef IntegerPointer Array7IntegerPointer[7];
typedef Array7Integer* PointerArray7IntegerPointer;
typedef IntegerPointer Array8Array7IntegerPointer[8][7];
```

Supplementary Exercise

Write declarations of objects of these types without using `typedef`s.

Exercise 5.4

Write a function that swaps (exchanges the values of) two integers. Use `int*` as the argument type. Write another swap function using `int&` as the argument type.

Hint: *Argument passing* is a fundamental concept explained in §7.2. The C++ standard library has a `swap` function (header `<algorithm>`, §18.6). Also see Exercise 5.6.

Here is a straightforward solution using a temporary variable in each case:

```
void swap_values(int *p1, int *p2) {
   int tmp = *p1;
   *p1 = *p2;
   *p2 = tmp;
}

void swap(int &v1, int &v2) {
   int tmp = v1;
   v1 = v2;
   v2 = tmp;
}
```

Some developers prefer the former version because the call site explicitly shows that the callee might change the arguments' values. Others prefer to reserve pointer arguments for those situations in which the callee will store the pointer value (thereby "taking possession" of the object pointed). Passing a pointer may then remind whoever analyzes the code that care must be taken with the management of the object and the memory it occupies.

Although consistency in this area is desirable, you may find that the various libraries you rely on use different conventions in this respect.

Supplementary Exercise

(Not fundamental, but short) You may find implementations of `swap` that look like the following:

```
void swap(int &v1, int &v2) {
   v1 = v1^v2;
   v2 = v1^v2;
   v1 = v1^v2;
}
```

Convince yourself that, in most cases, this will indeed swap the given values without the need for a temporary variable. Compare the performance of this function with the previous ones by timing a little benchmark of your own. Be sure to turn on the best optimizations available in your compiler. Independently of performance, can you see why this technique is usually not recommended?

Hint: When will it *not* work?

Exercise 5.5

What is the size of the array `str` in the following example:

```
char str[] = "a short string";
```

What is the length of the string "a short string?"

Hint: For an explaination of the important relationship *and* the distinction between arrays and pointers, see §5.3.

As explained in §5.2.2, `str` is an array whose length is determined by its string-literal initializer. Including the spaces and the terminating null character, there are 15 characters in this string, which answers the first question. Remember that the commonly used function `strlen` will not count the terminating null. Hence, `strlen(str)==14`, which answers the second question.

Exercise 5.6

Define functions `f(char)`, `g(char&)`, and `h(const char&)` and call them with the arguments 'a', 49, 3,300, `c`, `uc`, and `sc`, where `c` is a `char`, `uc` is an unsigned `char`, and `sc` is a signed `char`. Which calls are legal? Which calls cause the compiler to introduce a temporary variable?

Hint: This is a slightly subtler exercise about argument passing (see Exercise 5.4). Be sure to understand the answer to this fundamental exercise.

Some of the proposed calls are illegal because they involve binding an rvalue (for example, a temporary or a literal) to a non-`const` reference. In this exercise, this can only be the case for calls to the function `g(char&)`. Other calls might be illegal because the argument and parameter types do not match in signedness. The following program annotates the answer:

```
#include <iostream>

void f(char c) { std::cout << c << std::endl; }
void g(char &c) { std::cout << c << std::endl; }
void h(char const &c) { std::cout << c << std::endl; }

int main() {
   char c;
   unsigned char uc;
   signed char c;
   f('a');
   f(49);
   f(3300); // OK, but probably dangerous
   f(c);
   f(uc);
   f(sc);
```

```
    g('a'); // Error
    g(49); // Error
    g(3300); // Error
    g(c);
    g(uc); // Error: unsigned char->char produces rvalue
    g(sc); // Error: signed char->char produces rvalue
    h('a'); // Temporary
    h(49); // Temporary
    h(3300); // Temporary, probably dangerous
    h(c);
    h(uc); // Temporary (possibly dangerous)
    h(sc); // Temporary (possibly dangerous)
    return 0;
}
```

As explained in §7.2, temporaries may be introduced when rvalues (the concept complementing lvalues—see §4.9.6 and Chapter 2) are passed through references to `const` types. The temporary might be required because, for example, the function to which the reference is passed may take the address of that reference. Because literals have no address, they must first be loaded into a temporary, which does have an address. The creation of temporaries can sometimes lead to serious performance degradation. It may, therefore, be worthwhile to familiarize yourself with situations in which these temporaries appear.

The types of the arguments 49, 3,300, `uc`, and `sc` are `int`, `int`, `unsigned char`, and `signed char`, respectively. They must be converted to plain `char`. These conversions—which may be purely conceptual, without any actual computation—result in rvalues, and rvalues cannot bind to non-`const` references. Hence, most calls to `g` above are illegal. Conversions between integral types are generally possible, but they can change the value of the converted expression. For example, 49 is guaranteed to be representable by the type `char`, but 3,300 is not. The conversion of the integer 3,300 into a `char` is likely to result in a value different from 3,300. Similarly, one of the types `signed char` or `unsigned char` can represent values that are not representable by plain `char` (see also §C.3.4). Fortunately, good compilers will warn you about the possibility of unintentional conversions.

Exercise 5.7

Define a table of the names of the months of the year and the number of days in each month. Write out that table. Do this twice, once using an array of `char` for the names and an array for the number of days, and once using an array of structures, with each structure holding the name of a month and the number of days in it.

Hint: This is a good exercise to practice the fundamental material in §5.2, §5.3, and §5.7.

Ignoring the issue of leap years, here is the first solution:

```
#include <iostream>

int const n_months = 12;

char const *month[n_months]
   = {"January", "February", "March", "April",
      "May", "June", "July", "August",
      "September", "October", "November", "December"};

int const month_length[n_months]
   = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int main() {
   for (int k = 0; k!=n_months; ++k)
      std::cout << month[k] << ":\t"
                << month_length[k] << std::endl;
   return 0;
}
```

The second solution is not extremely different:

```
#include <iostream>

struct Month {
   char const *name_;
   int length_;
};

int const n_months = 12;

Month const month[n_months]
   = {{"January", 31}, {"February", 28}, {"March", 31},
      {"April", 30}, {"May", 31}, {"June", 30},
      {"July", 31}, {"August", 31}, {"September", 30},
      {"October", 31}, {"November", 30}, {"December", 31}};

int main() {
   for (int k = 0; k!=n_months; ++k)
      std::cout << month[k].name_ << ":\t"
                << month[k].length_ << std::endl;
   return 0;
}
```

Remember that Chapter 3 covers various aspects of getting programs, such as the one above, to work with C++ systems that do not yet fully support the standardized language

definition. Appending an underscore to the names of `struct`, `class`, and `union` members is purely conventional.

Exercise 5.8

Run some tests to see if your compiler really generates equivalent code for iteration using pointers and iteration using indexing (§5.3.1). If different degrees of optimization can be requested, see if and how that affects the quality of the generated code.

Hint: Although this is an interesting experiment, it can be safely skipped on a first reading. Remember that neither method is fundamentally superior.

I wrote the following little program to study this exercise:

```
#include <iostream>
#include <time.h> // For clock()
#include <stddef.h> // For size_t
#include <string.h> // For memcpy()

void copy_with_pointers(char const *src, char *dst,
                        size_t n) {
   for (size_t k = 0; k!=n; ++k)
      *dst++ = *src++;
}

void copy_with_indices(char const *src, char *dst,
                       size_t n) {
   for (size_t k = 0; k!=n; ++k)
      dst[k] = src[k];
}

int main() {
   int const n_bytes = 100000;
   int const n_calls = 100;
   char *src = new char[n_bytes];
   char *dst = new char[n_bytes];

   // Reduce paging effects by accessing all bytes:
   memcpy(dst, src, n_bytes);
   clock_t start, end, reftime;

   // Test 1 (reference time)
   start = clock();
   for (int k = 0; k!=n_calls; ++k)
      memcpy(dst, src, n_bytes);
```

```
    end = clock();
    reftime = end-start;

    // Test 2 (pointers)
    start = clock();
    for (int k = 0; k!=n_calls; ++k)
       copy_with_pointers(src, dst, n_bytes);
    end = clock();
    cout << "With pointers: "
         << static_cast<double>(end-start)/reftime
         << " times slower than memcpy.\n";

    // Test 3 (indices)
    start = clock();
    for (int k = 0; k!=n_calls; ++k)
       copy_with_indices(src, dst, n_bytes);
    end = clock();
    cout << "With indices: "
         << static_cast<double>(end-start)/reftime
         << " times slower than memcpy.\n";

    delete[] src;
    delete[] dst;
    return 0;
}
```

The program exploits a standard function `clock()` to determine the CPU time spent on each test. The time-units return by `clock()` varies from one machine to another, but for our purposes it is sufficient to compare times on a relative scale. To add more interest to the experiment, I compared my handwritten memory-copying functions with `memcpy` (provided by the standard library).

With one compiler, the two copying functions used the same amount of CPU time, and both were about twice as slow as `memcpy` (independent of whether optimization was turned on). The latter observation relates somewhat to Exercise 5.2: `memcpy` is typically implemented to take advantage of alignment. Rather than copying one byte at a time, `memcpy` will often copy words at a time if the source and destination arrays have similar alignments.

It may come as a surprise that the two functions used exactly the same amount of CPU cycles. An assembly language listing of the generated code reveals that the two functions were not implemented using identical code. The pointer version included two more instructions (to move the pointers) but used a simpler addressing mode. This CPU also had the capability to issue more than one instruction at a time (so that the additional instructions may not affect run-time). This illustrates some of the problems one may encounter when trying to evaluate code-generation quality based on small computational kernels.

Exercise 5.9

Find an example where it would make sense to use a name in its own initializer.

Hint: An illustration of a fine point in declaration syntax. See §4.9.4.

Such initializers are rarely needed. Occasionally, they appear when initializing self-referencing data structures. For example, it is sometimes convenient to have a linked list structure end with a guard node that points to itself. This could be done as follows:

```
struct ListNode {
   int value_;
   ListNode *next_;
};

ListNode const list_terminator = {0, &list_terminator };
```

Exercise 5.10

Define an array of strings where the strings contain the names of the months. Print those strings. Pass the array to a function that prints those strings.

Hint: Try Exercise 5.7 first. Read §3.5 for an introduction to C++ strings.

Let's show off the standard library for this one:

```
#include <iostream>
#include <algorithm>
#include <string>

using std::copy;
using std::cout;
using std::ostream_iterator;
using std::string;

int const n_months = 12;

string month[n_months]
    = { string("January"), string("February"),
        string("March"), string("April"),
        string("May"), string("June"),
        string("July"), string("August"),
        string("September"), string("October"),
        string("November"), string("December") };
```

```
int main() {
   copy(&month[0], &month[0]+n_months,
         ostream_iterator<string>(cout, "\n"));
   return 0;
}
```

The standard library algorithm `copy` (§18.6.1) takes two iterators describing a source range and another iterator that determines the beginning of a destination range. Its effect is to copy the elements of the source range to the destination range. An `ostream_iterator` (§19.2.6) provides an iterator-like interface for a output stream (first argument) and takes a second argument that allows us to configure a separator string for the elements.

In this case, the solution can also be expressed concisely with C-style strings (`char` arrays):

```
#include <iostream>

int const n_months = 12;

char const *month[n_months]
   = {"January", "February", "March", "April", "May", "June",
      "July", "August", "September", "October", "November",
      December"};

int main() {
   for (int k = 0; k!=n_months; ++k)
      std::cout << month[k] << '\n';
   return 0;
}
```

Exercise 5.11

Read a sequence of words from input. Use* Quit *as a word that terminates the input. Print the words in the order they were entered. Don't print a word twice. Modify the program to sort the words before printing them.

Hint: This exercise does not focus on a single aspect of C++ but, instead, requires some combination of basic C++ idioms. The tour chapters of *The C++ Programming Language* (§2, §3) form a great introduction for this exercise.

The standard library facilities worked well last time (Exercise 5.10), so let's try to use them again. First a word about our strategy.

To eliminate duplicates, we'll make use of a `set` (§17.4.3). Sets are sorted, so it's actually easier to provide for the second part of the exercise. To perform the first part (output the names in the order of input), we'll add a list of iterators into the sorted set that we will update every time a new word is input.

```
#include <algorithm>
#include <iostream>
#include <list>
#include <set>
#include <string>

int main() {
   using std::copy;
   using std::cout;
   using std::ostream_iterator;
   using std::pair;
   using std::string;
   typedef std::set<string> WordSet;
   typedef WordSet::iterator WordIter;
   typedef std::list<WordIter> Index;
   WordSet words;
   Index input_order;
   // Input the words uniquely
   for(string new_word;
       cin >> new_word, new_word!="Quit";) {
      pair<WordIter, bool> &trace = words.insert(new_word);
      if (trace.second)
         input_order.push_back(trace.first);
   }
   // Output unique words in order of input:
   copy(input_order.begin(), input_order.end(),
        ostream_iterator<string>(cout, "\n"));
   // Output unique words in default set<string> order:
   copy(words.begin(), words.end(),
        ostream_iterator<string>(cout, "\n"));
   return 0;
}
```

A compiler I tried had difficulties processing this code because it too eagerly instantiated certain members of `input_order`. A simple "hack" to make the program work on that compiler consisted of changing the type of `input_order` to `list<string>` and dereferencing the `push_back` argument (that is, use '`*trace.first`' instead of just '`trace.first`' above). This may affect efficiency, but many `std::string` implementations use sophisticated techniques to reduce the effective number of copy operations.

Note how the member function `insert` of a `set` or a `map` returns a `pair` (§17.4.1.2; a standard `struct` that holds two members, `first` and `second`). The first element of that pair is an iterator pointing to the value inserted in the set. The second is a Boolean value that equals `true` if and only if the value had not previously been present in the set.

Supplementary Exercise

(Just slightly more tedious than the above solution) Implement an equivalent program as compactly as possible without using the standard library's data structures (you can use the input/output facilities of `<iostream>`).

Exercise 5.12

Write a function that counts the number of occurrences of a pair of letters in a string and another that does the same in a zero-terminated array of `char` (a C-style string).

Hint: Consider reading §3.5 first. This exercise shows how many basic string and C-style string manipulations can be very similar.

I will consider that the pair 'aa' occurs three times in 'aaaa,' and the pair 'ab' occurs only once in 'abcba.'

```
#include <string>
using std::string;

size_t count_charpair(string const &s, char a, char b) {
   size_t result = 0;
   string::iterator p = s.begin();
   while(p!=s.end())
      if (*p++==a)
         if (p!=s.end && *p==b)
            ++result;
   return result;
}

size_t count_charpair(char const *s, char a, char b) {
   assert(s);
   size_t result = 0;
   while (*s) {
      if (*s++==a)
         if (*s==b)
            ++result;
   return result;
}
```

The strong similarity between these two is a consequence of a deliberate choice in the design of the standard library. It was deemed desirable to stay close to some idioms typical of C-style programming. Among other things, this makes it easy to switch between built-in arrays and library containers.

Exercise 5.13

Define a `struct Date` to keep track of dates. Provide functions that read `Date`s from input, write `Date`s to output, and initialize a `Date` with a date.

Hint: This is a great exercise to transition from the basic idea of a structure (§5.7) to the richer class concept (§10). The problem can be solved in not too much time. Try Exercise 7.19 after this.

I'm interpreting the term `struct` in this question as C-style struct. Clearly, it could also be taken to mean a class-type with constructors, destructors, and other bells and whistles. The following is a very minimal implementation, not taking into account, for example, the possibility of multiple locales for reading and writing `Date` objects. It also does not perform any form of error detection.

```
#include <iostream>

struct Date {
   unsigned day_: 5; // Bitfield: use only 5 bits to
                     //  represent this integer member
   unsigned month_: 4;
   int year_: 15;
};

std::istream& operator>>(std::istream &input, Date &d) {
   int const bufsize = 6;
   char buffer[bufsize];
   input.getline(buffer, bufsize, '/');
   d.month_ = atoi(buffer);
   input.getline(buffer, bufsize, '/');
   d.day_ = atoi(buffer);
   input >> d.year_;
   return input;
}

std::ostream& operator<<(std::ostream &output, Date &d) {
   output << d.month_ << '/' << d.day_ << '/' << d.year_;
   return output;
}

Date& init(Date &d, unsigned day, unsigned month, int year) {
   d.day_ = day;
   d.month_ = month;
   d.year_ = year;
   return d;
}
```

I chose to illustrate *bitfields* (§C.8.1) for the solution to this exercise. This makes the `Date` type potentially very compact but will often slow down operations on it. Unless your application must store a large number of `Date` objects, using bitfields in this context probably has only didactic value. Bitfields are sometimes useful to interface with legacy structures at a low level of abstraction. This code would be equally correct without bitfields:

```
struct Date {
   int day_;
   int month_;
   int year_;
};
```

The function `init` is similar to a constructor and returns its argument by reference, which allows the initialization to be chained with another operation. Note that this is standard practice for streaming operators (<< and >>), even though it is not, strictly speaking, mandatory.

[blank page]

Chapter 6

Expressions and Statements

Expressions and statements are the heart of C++ programs. They describe the computations and actions to be performed by your software. This chapter illustrates code fragments, making use of the various constructs available in C++. It also demonstrates how some constructs have behaviors that cannot be accurately predicted without knowing which platform will be used to compile and run the program. In most situations, such expressions or statements should be avoided. Occasionally, they may be used to squeeze the available computing power. Finally, one sometimes needs many expressions and statements to achieve a goal. It pays to reuse as much work as possible when that is the case. Hence, many programs in this chapter make use of the C++ standard library (§3, §16).

Exercise 6.1

Rewrite the following `for` statement as an equivalent `while` statement:

```
for (i = 0; i<max_length; i++)
   if (input_line[i]=='?') quest_count++;
```

Rewrite it to use a pointer as the controlled variable—that is, so that the test is of the form `*p=='?.'`

Hint: Reading §5.3.1, §6.2.5, and §6.3.3 may help in solving this question. The duality of the pointer idiom versus the array indexing idiom is very popular among C and C++ programmers. See also Exercise 5.8.

The first part is, in fact, quite mechanical (that is, it can always be done in a similar way):

```
i = 0; // Initialization
while (i<max_length) { // Test
   // Body of for-statement:
   if (input_line[i]=='?') quest_count++;
```

```
    // Updating expression:
    i++;
}
```

The answer to the second question is easily found as well (refer to §5.3.1 for a discussion of these complementary idioms). Assume p is a pointer to char:

```
for (p = input_line; p!=input_line+max_length; ++p)
    if (*p=='?')
        ++quest_count;
```

Note that I chose to use pre-incrementation where possible in this last code fragment. In this case, it hardly matters whether pre- or post-incrementation is used. However, because post-incrementation implies the formation (at least conceptually) of a temporary, it does occasionally pay to choose pre-incrementation over post-incrementation. The same argumentation applies to the decrement operators, of course.

Exercise 6.2

Fully parenthesize the following expressions:

```
a = b+c*d<<2&8
a&077!=3
a==b || a==c && c<5
c = x!=0
0<=i<7
f(1, 2)+3
a = -1+ +b-- -5
a = b==c++
a = b = c = 0
a[4][2] *= *b? c: *d*2
a-b, c = d
```

Hint: Use the table at the end of Chapter 2 (reproduced from §6.2). Do not hesitate to add redundant parentheses in your code if that makes the expressions easier to read.

Remember that all operators, except unary and assignment operators (including *=, +=, etc.), are left-associative.

```
a = (((b+(c*d))<<2)&8)
a&(077!=3)
(a==b) || ((a==c) && (c<5))
c = (x!=0)
(0<=i)<7
(f(1, 2))+3   //Argument list, not comma expression.
a = (((-1)+(+(b--)))-5)
a = (b==(c++))
```

```
a = (b = (c = 0))
((a[4])[2]) *= ((*b)? (c): ((*d)*2))
(a-b), (c = d)
```

The parentheses in the second expression illustrate an operator precedence rule that is sometimes surprising; the bit-manipulation operators have lower precedence than the relational operators.

In the seventh expression, not all the whitespace is optional. Indeed, the expression

```
a=-1++b---5
```

is parsed as

```
a = (-1)((++(b--))-5)
```

but is not legal syntax. This is a consequence of the *maximum munch principle*—as many characters as possible are accumulated to make a single token. Hence, when the compiler is fetching a new token and the next input characters are '`---`,' it will not stop at the first '`-`' because '`--`' is also a valid token. For built-in types, '`++(b--)`' is illegal because the pre-increment operator requires a modifiable lvalue (§4.9.6), while '`b--`' is not an lvalue (that is, it is an rvalue).

Exercise 6.3

Read a sequence of (name, value) pairs, where the name is a single whitespace-separated word and the value is a floating-point value. Compute and print the sum and mean for each name, and the sum and mean for all names. Hint: §6.1.8.

Hint: Familiarize yourself with standard library `map` (§3.7.4, §17.4.1) and `pair` (§17.4.1.2) templates. See Exercise 6.12 for a more sophisticated extension of this exercise.

It turns out to be convenient to use a `map` (§17.4.1) to associate two values with every name that is input: the sum of all values seen with that name, and the number of times that name has been seen.

```
#include <iostream>
#include <map>
#include <string>

struct Stat {
   // Make sure the default constructor does the right thing
   // (used internally by the map data-structure)
   Stat(): sum_(0.0), count_(0) {}
   double sum_;
   int count_;
};
```

```
using std::string;
typedef std::map<string, Stat> Data;

void collect_data(Data &stats) {
   string name;
   while (cin >> name) {
      double datum;
      std::cin >> datum;
      stats[name].sum_ += datum;
      ++stats[name].count_;
   }
}

void print_stats(Data const &stats) {
   double global_sum = 0.0;
   int global_count = 0;
   for (Data::iterator p = stats.begin(); p!=stats.end(); ++p) {
      std::cout << (*p).first // the 'key' element
                << ": sum = " << (*p).second.sum_
                << ", mean = "
                << (*p).second.sum_/(*p).second.count_
                << '\n';
      global_sum += (*p).second.sum_;
      global_count += (*p).second.count_;
   }
   std::cout << "Global sum: " << global_sum
             << ", Global mean: " << global_sum/global_count
             << std::endl;
}

int main() {
   Data stats;
   collect_data(stats);
   print_stats(stats);
   return 0;
}
```

Maps are linear containers that maintain (key, value) pairs in key-sorted order. Hence, iterators in maps point to `pair` objects (§17.4.1.2); in this case, `pair<string, Stat>` objects. That explains why the `second` member of `pair` is used to access the `Stat` values stored in the `Data` map.

An alternative implementation can make use of `multimaps` to associate the list of values read with each name. For this particular problem, however, this is a less efficient approach in terms of both storage and run-time.

Exercise 6.4

Write a table of values for the bitwise logical operations (§6.2.4) for all possible combinations of 0 and 1 operands.

Hint: See §6.2.4. Although two's complement representation is by far the most common integer representation on modern computers, there are a number of important architectures that use other representations. Also beware of the low precedence of bitwise operators.

Let's assume in Table 6.1 that the operands have type `int` and that a two's complement representation is used internally.

Table 6.1 *Values for bitwise logical operations*

a	*b*	*~a*	*a|b*	*a&b*	*a^b*	*a>>b*	*a<<b*
0	0	-1	0	0	0	0	0
0	1	-1	1	0	1	0	0
1	0	-2	1	0	1	1	1
1	1	-2	1	1	0	0	2

Exercise 6.5

Find five different C++ constructs for which the meaning is undefined (§C.2). Find five different C++ constructs for which the meaning is implementation-defined (§C.2).

Hint: See §C.2. *Undefined behavior* can lead, literally, to *anything*: a compile-time or run-time diagnostic, catastrophic failure, noncatastrophic glitches, or just what you intended. This and the following exercises teach you to recognize situations to avoid.

There are many ways to trigger undefined behavior in C++. The following are a few that I have encountered in programs.

1. Access outside the bounds of an array:

```
int a[10];
int *p = &a[10]; // a[0]..a[9] OK, a+10 also, but not a[10]
```

2. Use of a destroyed object:

```
int &r = *new int;
delete &r;
r = r+1; // reference no longer valid
```

3. Attempting to reinterpret variables:

```
int i;
*(float*)&i = 1.0; // access through undeclared type
```

4. Casting to a type that is not the real type of the source object:

```
struct B { int i; };
struct D: B {};
B* bp = new B;
D* dp = static_cast<D*>(bp); // invalid cast
```

5. Casting away constness of an object that was defined `const`:

```
void f(int const &r) { const_cast<int&>(r) = 2; }
int const i = 3;
f(i); // invalid const_cast
```

Tripping on implementation-defined constructs is also rather easy.

1. The size of certain types:

```
sizeof(int);
```

2. The output of `type_info::name()`:

```
std::cout << typeid(double).name();
```

3. The effect of bitwise operators:

```
int mistery = ~0;
```

4. The extreme values of fundamental types:

```
int mx = numeric_limits<int>::max();
```

5. The number of temporary copies:

```
struct S { S(S const&) { std::cout << "copy.\n"; } };
S f() { S a; return a; }
int main() { f(); return 0; }
```

Undefined behavior is almost always undesirable and can usually be avoided. Implementation-defined behavior usually cannot be avoided, but it is often possible to write programs in such a way that they are insensitive to the implementation-defined behavior. For example, the `sizeof` operator and the `numeric_limits` template can be used to parameterize your code in terms of various implementation-specific quantities.

Exercise 6.6

Find 10 different examples of nonportable C++ code.

Hint: Aim for portability (§23.4.2). A moderate amount of conditional compilation (§7.8.1) can help when nonportable code is unavoidable.

See the sample answer to Exercise 6.5. Writing portable code is often desirable because the systems on which the code will eventually be deployed are rarely fully determined in advance. Even when it is decided in advance, market realities may dictate that the target platforms be reconsidered. Hence, it is important to be aware of which constructs rely on nonportable implementation features.

Yet programs do not exist in a vacuum, and most significant projects will need to rely to a certain extent on nonportable behavior. Even in these cases, there are good practices. First, undefined behavior should be avoided at almost any cost. The other two categories of nonportable constructs are implementation-defined (the actual behavior is documented) and unspecified (no documentation may be available). In either case, it is often convenient for the purpose of maintenance to isolate the code invoking these behaviors in separate code units (namespaces, files, classes). Furthermore, it may be a good idea to use the preprocessor facilities to ensure that the code will not compile on an unrecognized platform. For example:

```
#if defined __SYSTEM_X__ // predefined on "System X"
// nonportable code here
#elif defined __SYSTEM_Y__ || defined __SYSTEM_Z__
// other nonportable code
#else
#error "Unrecognized system"
#endif
```

This practice will help avoid unknowingly triggering unexpected behavior. Of course, this does not preclude the need for thorough testing.

Exercise 6.7

Write five expressions for which the order of evaluation is undefined. Execute them to see what one or—preferably—more implementations do with them.

Hint: See §6.2.2.

```
#include <iostream>

int gi = 0;
int f(int a, int b) {
   std::cout << "(a b gi) = (" << a << ' ' << b
                                  << ' ' << gi << ")\n";
   return a+b+gi;
}
```

```
int main() {
   f(++gi, ++gi);
   f(1, 2)+f(3, 4);
   if (f(5, 6)==++gi);
   f((gi++)+(gi++), 0);
   gi & 32/gi;
   return 0;
}
```

The operative technical term in this context is *sequence point*. A sequence point is a point during the execution of a program at which all the side effects of previous expressions have completed while side effects of following expressions have not yet taken place. Many components of an expression do not introduce a sequence point and, therefore, do not guarantee that the effect of this component will have taken place by the time another component is being evaluated. In the last case above, for example, it is not guaranteed that the bitwise and-operator will first evaluate its first operand and then skip the evaluation of the second operand if the former is zero (unlike the logical and-operator, which introduces a sequence point after the evaluation of its first operand).

Here is the output of one implementation:

```
(a b gi) = (2 1 2)
(a b gi) = (1 2 2)
(a b gi) = (3 4 2)
(a b gi) = (5 6 2)
(a b gi) = (7 0 5)
```

The first line indicates that the right-most argument of the call to function `f` was evaluated first. The last line demonstrates that the post-increment operation completes before control is transferred to function `f`. This is, in fact, guaranteed by the language (there is a sequence point after the evaluation of function arguments). However, it is unspecified that one post-increment will complete before the other initiates. In this implementation, this was the case: `f` passed value 7 == 3 + 4 as its first argument (otherwise, value 6 == 3 + 3 would have been passed).

Another implementation produced just slightly different output:

```
(a b gi) = (2 2 2)
(a b gi) = (1 2 2)
(a b gi) = (3 4 2)
(a b gi) = (5 6 3)
(a b gi) = (7 0 5)
```

Exercise 6.8

What happens if you divide by zero on your system? What happens in case of overflow and underflow?

Hint: See also Exercise 8.10.

The most common behavior when dividing by zero in an integer division is a premature abortion of the program (often with a run-time diagnostic provided by the underlying operating system, and possibly with a file that can be used for post-mortem analysis of the program state at the point of division). Many systems provide ways to intercept this abortion process, thereby allowing more graceful handling (and possibly recovery) of such situations. A few recent systems even provide mechanisms that allow such an error to be transformed into a C++ exception; however, the language specifications certainly do not require this possibility.

Similar remarks can be made about floating-point divisions. In addition, it is noteworthy that many floating-point representations provide for the representation of nonreal values (infinities or nonnumerical values used, for example, to represent domain errors such as computing the square root of a negative value.)

Overflow and underflow are more typically ignored by default, but certain platforms can also be configured to abort and/or diagnose when such situations occur.

Exercise 6.9

Fully parenthesize the following expressions:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

Hint: See Exercise 6.2 and Table 2.1.

The suggestion made in Exercise 6.2 still holds: Use parentheses to make precedence explicit when it is not otherwise obvious.

```
*(p++)
*(--p)
++(a--)
(int*)(p->m)
*(p.m)
*(a[i])
```

Note that the third expression is illegal if a has a built-in type (see also Exercise 6.2). Even if a has a user-defined type, it is usually considered poor practice to overload ++/-- in a way that makes this expression legal.

Exercise 6.10

Write these functions: `strlen()`, which returns the length of a C-style string; `strcpy()`, which copies a string into another; and `strcmp()`, which compares two strings. Consider what the argument types and return types ought to be. Then compare with the standard library versions as declared in <cstring> (<string.h>) and specified in §20.4.1.

Hint: C-style strings are introduced in §3.5.1 and §5.2.2.

Here are definitions and declarations that make sense:

```
#include <cstddef> // For std::size_t

std::size_t strlen(char const *s) {
   assert(s);
   std::size_t len = 0;
   while (s[len])
      ++len;
   return len;
}

void strcpy(char *d, char const *s) {
   assert(s && d);
   do {
      *d++ = *s;
   } while (*s++);
}

enum CStringOrder { FirstBeforeSecond = -1, Same = 0,
                    SecondBeforeFirst = 1};

CStringOrder strcmp(char const *first, char const *second) {
   assert(first && second);
   unsigned char const *f
      = static_cast<unsigned char const*>(first);
   unsigned char const *s
      = static_cast<unsigned char const*>(second);
   do {
      if ((*f)<(*s))
         return FirstBeforeSecond;
```

```
        else
        if ((*s)<(*f))
           return SecondBeforeFirst;
    } while (*(s++) && *(f++));
    assert((*(s-1)==0) && (*(f-1)==0));
    return Same;
}
```

The standard version of `strcpy` returns a pointer to its first argument (`char*`), which makes it convenient to chain various C-string operations. The `assert` macro (explained in §24.3.7.2) causes the program to abort if the specified condition is false.

The standard library version of `strcmp` does not return an enumeration type, but an integer. Furthermore, in case of inequality, only the sign of this integer is guaranteed. This might gain cycles on certain implementations because the loop could be coded as:

```
do {
   int diff = (*f)-(*s);
   if (diff)
      return diff;
} while (*(s++) && *(f++));
```

Otherwise, these declarations match the ones of the standard library and the declarations found on my local implementation. You may find that certain implementations have not incorporated the `const` keyword (which was added later in the evolution of the C language) or have replaced `size_t` by another fundamental type (`size_t` was also added later in the evolution of the C language). On the other hand, you may find that your implementation has added the qualifier `restrict` to certain parameter declarations. This is a new (and not yet fully standardized at the time of this writing) keyword of the C language that indicates that two pointers point to distinct objects. Although the C++ language does not include this keyword, it is likely to be provided by vendors as an extension at some time in the future.

Supplementary Exercise

Compare the above implementations' performance with those of the library supplied with your compiler. Make sure that you enable all the optimizations that your compiler is capable of. Make an educated guess about to what causes the difference (if any).

Exercise 6.11

See how your compiler reacts to these errors:

```
void f(int a, int b) {
   if (a = 3) // ...
   if (a&077==0) // ...
   a := b+1;
}
```

Hint: Compilers can drastically differ in the quality of their diagnostics, and good diagnostics can be a tremendous asset for coding productivity.

Note that the `if` statements contain valid expressions that were probably unintended. The first contains an assignment instead of a comparison; the second will apply the bitwise operation after the equality comparison. The implementation on which I initially developed many of the answers to these exercises warns about both of these dubious constructs. Note that certain programmers in fact enjoy on-the-fly assignments such as the one used in the first if statement. To accommodate this style, implementations that, by default, warn about such usage often allow the programmer to turn off this kind of warning.

Finally, the expression '`a := b+1`' is ill-formed in C++. However, it is the correct syntax in a variety of other programming languages (for example, Pascal). The error messages produced by several C++ compilers reflect that the colon is used for goto labels in C++ (§6.3.4). However, the first C++ compiler correctly diagnosed all cases and would say that '`:=`' is not a C++ operator.

Exercise 6.12

Modify the program from Exercise 6.3 to also compute the median.

Hint: Try Exercise 6.3 first. It is similar to, but simpler than, this one.

I interpret the question to require the computation of both medians of each name's values, as well as the median of the total collection of values. Remember that the median of a sequence of values is its middle value m; if there are an even number of elements in the sequence, then m is the mean of the two middle elements. Unlike Exercise 6.3, we must now keep a recording of all the data seen in order to compute the median. At first, I thought I should use a multimap (§17.4.2) rather than a plain map (§17.4.1). However, after trying to complete the code, I realized that I wanted not only the names sorted, but also the values. Hence, I decided to use a map of multisets (§17.4.4). Each name has an associated set of possibly distinct values rather than the sum and count of these values as was done in Exercise 6.3.

```
#include <iostream>
#include <map>
#include <numeric>
#include <set>
#include <string>

using std::string;
typedef std::multiset<double> SubData;
typedef std::map<string, SubData> Data;

void collect_data(Data &stats) {
   string name;
```

```
   while (std::cin >> name) {
      double datum;
      std::cin >> datum;
      stats[name].insert(datum);
   }
}

void print_setstats(SubData const &stats) {
   std::size_t count = 0;
   double sum = 0.0, median;
   for (SubData::const_iterator p = stats.begin();
        p!=stats.end(); ++p, ++count) {
      if (count==(stats.size()-1)/2) {
         median = *p;
         if (stats.size()%2==0) {
            SubData::const_iterator q = p;
            median = (median+(*++q))/2.0;
         }
      }
      sum += *p;
   }
   std::cout << "sum = " << sum
             << ", mean = " << sum/double(count)
             << ", median = " << median << std::endl;
}

// merge_stats: move items from the per-name sets to
// a global set, so that we can eventually compute the
// overall statistics.
void merge_stats(SubData &dest, SubData &src) {
   // Associative containers can take "insertion hints":
   SubData::iterator hint = dest.begin();
   for (SubData::iterator p = src.begin();
        p!=src.end(); ++p) {
      hint = dest.insert(hint, *p);
      src.erase(p);
   }
}

void print_stats(Data const &stats) {
   SubData global_set;
   for (Data::const_iterator p = stats.begin();
        p!=stats.end(); ++p) {
      cout << (*p).first << ": ";
```

```
        print_setstats((*p).second);
        merge_stats(global_set, (*p).second);
    }
    cout << "Global stats: ";
    print_setstats(global_set);
}

int main() {
    Data stats;
    collect_data(stats);
    print_stats(stats);
    return 0;
}
```

Note the use of a version of `multiset::insert` that takes a hint about where searching for the insertion point should start. This allows for merging the data associated with a certain name into the global set more efficiently.

Exercise 6.13

Write a function `cat()` that takes two C-style string arguments and returns a string that is the concatenation of the arguments. Use `new` to find storage for the result.

Hint: See §20.4.

This is relatively straightforward using the standard C-style string facilities:

```
#include <cstring>

char* cat(char const *a, char const *b) {
    std::size_t la = std::strlen(a);
    char *s = new char[la+std::strlen(b)+1];
    std::strcpy(s, a);
    std::strcpy(s+la, b);
    return s;
}
```

Remember that C-style strings have a trailing null character not counted by `std::strlen`. This explains the '+1' in the `new` expression.

Note that this routine allocates memory for the string but that caller is responsible for deallocation. It is often undesirable to distribute allocation and deallocation this way. Here is an alternative concatenation routine that leaves both allocation and deallocation to the client:

```
void concatenate(char const *a, char const *b, char *dest) {
    while (*a!='\0') *dest++ = *a++;
```

```
    while (*b!='\0') *dest++ = *b++;
    *dest = '\0';
}
```

If the `while` loops in this code appear too cryptic, read §6.2.5 for a thorough discussion of similar constructs.

Exercise 6.14

Write a function `rev()` that takes a string argument and reverses the characters in it. That is, after `rev(p)` the last character of `p` will be the first, etc.

A one-liner solves this using the C++ standard library facilities:

```
void rev(string &s) { std::reverse(s.begin(), s.end()); }
```

or for C-style strings:

```
void rev(char *s) { std::reverse(s, s+std::strlen(s)-1); }
```

The `reverse` algorithm (from header `<algorithm>`) is described in §18.6.7. It takes two *bidirectional iterators* (§19.2.1) delimiting the sequence to be reversed in place.

Here is a solution that does not involve the standard library:

```
void rev(char *s) {
    for (char *e = s+std::strlen(s)-1; s<e; ++s, --e) {
        char tmp = *s; // Swap *s and *e
        *s = *e;
        *e = tmp;
    }
}
```

Although the `std::reverse` algorithms can be replaced by a few lines of code (as shown in this last example), it is remarkable how much these little algorithms improve overall readability.

Exercise 6.15

What does the following example do?

```
void send(int *to, int *from, int count)
// Duff's device. Helpful comment deliberately deleted.
{
    int n = (count+7)/8;
    switch (count%8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
```

```
        case 6:     *to++ = *from++;
        case 5:     *to++ = *from++;
        case 4:     *to++ = *from++;
        case 3:     *to++ = *from++;
        case 2:     *to++ = *from++;
        case 1:     *to++ = *from++;
                } while (--n>0);
        }
    }
```

Hint: Don't do this in real code. This exercise illustrates an obscure syntax rule.

This convoluted piece of code, in fact, does nothing but copy `count` integers from the array pointed to by `from` to the array pointed to by `to`. You might also notice that this particular version will not work if `count` is zero on entry (moving the '`case 0:`' label to the end of the `while` loop would resolve that particular problem).

This *Duff device* is sometimes used to explicitly "unroll loops." The desire to do this stems from the fact that in tight loops, most of the time is spent in the branch-back code. The above implementation presumably reduces the number of branch-back instructions (generated by the compiler) by a factor of eight.

On modern compilers, However, there is no justification for this sort of hard-to-maintain construct. Indeed, most modern compilers will unroll loops for you and do so with an optimal unroll factor (a factor of eight may very well decrease performance overall). This observation can be generalized. Many low-level optimizations should be left to the compiler, which usually has a more up-to-date knowledge of the timing properties of the target platform.

Finally, for the case above, one should also remember that the standard library includes ready-to-use copying routines. In this case, we could use (see §18.6.1):

```
std::copy(to, to+count, from);
```

Exercise 6.16

Write a function `atoi(const char*)` that takes a string containing digits and returns the corresponding `int`. For example, `atoi("123")` is 123. Modify `atoi()` to handle C++ octal and hexadecimal notation in addition to plain decimal numbers. Modify `atoi()` to handle the C++ character constant notation.

Hint: The C++ standard library provides this function, but it is instructive to try to implement it using core C++ statements and expressions. It may also be useful to try Exercise 6.17 first.

I show only the final version of the function with all the bells and whistles. We must decide how to handle unexpected conditions: range errors (values whose magnitudes are too large to be contained in an `int`) and domain errors (unexpected characters). This version throws an appropriate exception among the ones provided by the standard library. Also, to keep

things clear, helper functions are placed in an unnamed namespace (so that clients of `atoi` can use these names for other things).

```
#include <stdexcept>
#include <string>
#include <limits>

using std::domain_error;
using std::range_error;
using std::string;

namespace {
   inline int digit(char c, int base) {
      int value;
      switch (c) {
         case '0': value = 0; break;
         case '1': value = 1; break;
         case '2': value = 2; break;
         case '3': value = 3; break;
         case '4': value = 4; break;
         case '5': value = 5; break;
         case '6': value = 6; break;
         case '7': value = 7; break;
         case '8': value = 8; break;
         case '9': value = 9; break;
         case 'a': case 'A': value = 10; break;
         case 'b': case 'B': value = 11; break;
         case 'c': case 'C': value = 12; break;
         case 'd': case 'D': value = 13; break;
         case 'e': case 'E': value = 14; break;
         case 'f': case 'F': value = 15; break;
         default:
            throw domain_error(string("invalid digit"));
      }
      if (value>=base)
         throw domain_error(string("invalid digit"));
      return value;
   }

   inline char next_char(char const *&p) {
      if (*p!='\\') // \ is special; hence '\\'
         return *p++;
      else { // 3 octal digits:
         int char_value = digit(p[1], 8)*64
                        +digit(p[2], 8)*8
                        +digit(p[3], 8);
```

```
            if (char_value>std::numeric_limits<char>::max()
             or char_value<std::numeric_limits<char>::min())
               throw domain_error(string("not a char"));
            p += 4; // backslash and 3 octal digits
            return char_value;
         }
      }

      void load_first_digit(char const *&s, int &value,
                            bool &is_negative, int &base) {
         char c1 = next_char(s);
         is_negative = c1=='-';
         if (c1=='-' || c1=='+')
            c1 = next_char(s);
         if (c1=='\0') { // "", "-" and "+" are illegal
            throw domain_error(string("invalid input"));
         } else if (c1!='0') {
            base = 10;
         } else {
            char const *p = s;
            char c2 = next_char(p);
            if (c2=='x' || c2=='X') { // "0x..."?
               base = 16;
               s = p;
               c1 = next_char(s);
            } else { // c2!='x' and c2!='X'
                base = 8; // I.e., even "0" is treated as octal
            }
         }
         value = digit(c1, base);
      }
   } // End unnamed namespace (helper functions)

   int atoi(char const *s) {
      int value, base;
      bool is_negative;
      load_first_digit(s, value, is_negative, base);
      while (char c = next_char(s)) {
         if (value>std::numeric_limits<int>::max()/base)
            throw range_error(string("out-of-range"));
         value *= base;
         int d = digit(c, base);
         if (value>std::numeric_limits<int>::max()-d)
            throw range_error(string("out-of-range"));
```

```
        value += d;
    }
    return is_negative? -value: value;
}
```

Observe how there are many corner cases in this problem. For example, one must be careful when testing that the value represented by the string does not exceed the range of integers. Again, there exists a similar function (with the same name) in the standard library. However, it will not throw an exception when it is passed an invalid string. As an experiment, you might want to compare the performance of `atoi` as coded above (with the best optimization available on your compiler) with that of the function `atoi` available from the standard library.

Supplementary Exercise

(Somewhat subtle) Why will the above `atoi` function fail on a system using a two's complement integer encoding when the input represents `numeric_limits<int>::min()`? Correct this.

Exercise 6.17

Write a function `itoa(int i, char b[])` that creates a string representation of `i` in `b` and returns `b`.

Hint: See Exercise 6.16.

This is essentially the inverse operation of the previous exercise. It is also considerably simpler. Because it is easiest to determine the digits starting with the least significant (right-most), a loop was inserted to first move to the right-most position of the final representation.

```
namespace { char const digit[] = "0123456789"; }

char* itoa(int i, char b[]) {
    char *p = b;
    if (i<0) { // Add in the sign if needed
        *p++ = '-';
        i = -i;
    }
    int shifter = i;
    do { // Move to where the representation ends
        ++p;
        shifter = shifter/10;
    } while (shifter);
    *p = '\0';
    do { // Move back, inserting digits as you go
```

```
        *--p = digit[i%10];
        i = i/10;
    } while (i);
    return b;
}
```

This function does not exist as is in the C/C++ standard library. However, the library does provide `sprintf` (formatted print to C-style string) to achieve the same effect:

```
char* itoa(int i, char b[]) {
    sprintf(b, "%d", i);
    return b;
}
```

`sprintf` behaves like `printf` (§21.8) but places the output in the array pointed to by its first argument. See also §21.5.3 for the related `stringstream` facility of the C++ standard library.

Exercise 6.19

Modify the calculator to report line numbers for errors.

Hint: This discussion assumes you are familiar with the presentation of a desk calculator in §6.1.

We need to make two modifications to the calculator code. We need a device that keeps track of line numbers, and we need the error-reporting function to make use of that information. Fortunately, the calculator design has kept input local to a single function, and, similarly, there is only one error-reporting function. We could use a global `int` variable to keep track of which line we're on. But to allow for more-complex functionality in the future, let's introduce a special-purpose class variable instead:

```
struct LineCount {
    LineCount(): line_(1) {} // There is no line zero
    unsigned long current() { return line_; }
    void new_line() { ++line_; }
private:
    unsigned long line_;
};

LineCounter line_count;
```

Now we can adapt the low-level input function to count lines:

```
Token_value get_token() {
    char ch;
```

```
    do { // skip whitespace except '\n'
       if (!cin.get(ch))
          return curr_tok = END;
    } while (ch!='\n' && isspace(ch));
    switch (ch) {
       case '\n': line_count.new_line(); // Fall through!
       case ';': return curr_tok = PRINT;
       case '*': // ...
}
```

and the error reporting function is straightforward:

```
double error(string const &msg) {
   ++no_of_errors;
   cerr << "ERROR (line " << line_count.current() << "): "
        << msg << '\n';
   return 1;
}
```

Observe how the well-defined structure of the calculator program made the task of this exercise rather easy. In general, you will find that the importance of this principle grows more than linearly with the size of the software.

Exercise 6.22

Write a program that strips comments out of a C++ program. That is, read from `cin`, remove both // comments and /* */ comments, and write the result to `cout`. Do not worry about making the layout of the output look nice (that would be another, and much harder, exercise). Do not worry about incorrect programs. Beware of //, /*, and */ in comments, strings, and character constants.

Hint: This is a good exercise to practice your understanding of C/C++ control statements.

For this exercise, C++ source code can be seen as consisting of a sequence of *contexts* that can be C-style comments, C++-style comments, string literals, character literals, and "code." The latter three contexts should be copied without modification; the former two should be filtered out. We cannot clump these contexts together, however, because they are delimited by different character sequences.

Central to our implementation will be the idea that switching from one context to another always occurs through the code context. So the main loop calls a function that traverses the code context and returns a value that indicates what the next context is (or `file_end` if the end of the file was encountered). This returned value is then used to decide which function to use to handle (skip or copy) the next context.

```
enum Context { c_comment, cpp_comment,
               string_literal, char_literal, file_end };

void handle_c_comment() {
   char ch;
   while (cin.get(ch)) {
      if (ch=='*') {
         // Skip additional asterisks:
         while(cin.get(ch) && ch=='*');
         if (ch=='/')
            break;
      }
   }
}

void handle_cpp_comment() {
   // Skip until end-of-line:
   char ch;
   while (cin.get(ch) && ch!='\n');
}

void handle_literal(char delimiter) {
   cout << delimiter; // Read left delimiter
   char ch;
   while (cin.get(ch)) {
      cout << ch;
      if (ch==delimiter) // Right delimiter found?
         break;
      elseif (ch=='\') // Next character passed unmodified
         cin.get(ch) && cout << ch;
   }
}

Context handle_code() {
   char ch;
   while (cin.get(ch)) {
      switch (ch) {
         case '/':
            if (!cin.getch(ch)) { // This would in fact be
               cout << '/';       //  erroneous C++ code
               return file_end;
            } else {
               if (ch=='*')
                  return c_comment;
```

```
                else if (ch=='/')
                    return cpp_comment;
                else {
                    cout << '/';
                    cin.putback(ch);
                    break;
                }
            }
        case '\"': return string_literal;
        case '\'': return char_literal;
        default: cout << ch;
      }
   }
   return file_end;
}

int main(int argc, char *argv[]) {
   if (argc!=0) {
      cerr << "This program takes no arguments.\n";
      return -1;
   } else {
      Context context;
      while ((context = handle_code())!=file_end)
         switch(context) {
            case c_comment:
               handle_c_comment();
               break;
            case cpp_comment:
               handle_cpp_comment();
               break;
            case string_literal:
               handle_literal('\"');
               break;
            case char_literal:
               handle_literal('\'');
               break;
         }
      return 0;
   }
}
```

The `switch` in the main function and the one in `handle_code` are somewhat redundant; you could have the `while` loop of `handle_code` replace the one of `main`. However, I chose to make `handle_code` look more like the other context-handling functions.

Exercise 6.23

Look at some programs to get an idea of the variety of indentation, naming, and commenting styles actually used.

Hint: Naming entities in a program is an art with major impact on maintainability.

Indentation, naming, and commenting styles do not make a good program. However, the introduction of reasonably consistent rules makes it easier for you and people who must deal with your code to understand what you meant to do when you wrote the code. You may also find that your environment includes some tools to test or enforce such conventional rules (style guides).

Style guides often include conventions regarding the use of whitespace. Many programmers like to introduce whitespace very liberally. Personally, I prefer to limit the length of my source-code lines to 78 characters or less (to easily fit 80-character screens) and the length of my functions to what fits on a screen (ideally, 24 lines, but I often work in an environment in which a screen has 40 lines and I tend to indulge that length).

Indentation reflects the block-structure of your programs. I like the resulting staircase effects to have a reasonably strong contrast, and I usually choose to add three blank characters for each level of indentation. Like so many style items, this is a matter of personal taste (although I have not yet found an experienced programmer happy with less than two blank characters per indentation level).

Certain popular naming conventions use names (like `apple_f`) in which the type associated with the name (here, `f` stands for `fruit`) is encoded. There have been endless debates over whether this is appropriate or not. In C and C++, however, this style is made more tedious when user-defined types are used extensively (typically the case in large software systems). Other developers like to have an indication of the scope of certain names—for example, I append an underscore character to the names of data members. Perhaps the most common characteristic of a name is its function: Does the name denote a variable, a typename, a constant, a function? Very often, each of these functions will be indicated with special capitalization rules (for example, `lower_case_with_underscores`, `UpperCaseWithoutUnderscores`, `ALL_CAPS`, or `somethingMoreExotic`) or prefix/postfix conventions (`k_prefix_for_konstants`, `data_members_`, or `m_dataMembers`). Because the `ALL_CAPS` notation is so widely used for macros, it is probably best to stick to that convention.

Finally, you might find even more variety in the number of commenting styles. For example, there are thousands of cute little ways to delimit larger comments from preceding or following segments of code. Again, the details of your preferred style matter little, but it is a good idea to strive toward informative comments that (a) explain the role of a little piece of code in its larger context, and (b) point out the subtleties when they're unavoidable. Although the design of subtle techniques may feel elegant, they often hinder maintenance unless they have evolved to become a widespread idiom.

Chapter 7

Functions

C++ functions—sometimes called *procedures* or *subroutines* in other programming languages—are the primary method for hiding the details of combinations of statements and expressions. In conjunction with classes, they lead to the concept of member functions and provide the essential interface element to enable software libraries. This chapter builds on the practices of Chapter 6 to exercise your skills in developing usable and *re*usable functions.

Exercise 7.1

Write declarations for the following: a function taking arguments of type* pointer to character *and* reference to integer *and returning no value; a pointer to such a function; a function taking such a pointer as an argument; and a function returning such a pointer. Write the definition of a function that takes such a pointer as argument and returns its argument as the return value. Hint: Use `typedef`.

Hint: This fundamental exercise complements Exercises 5.1 and 5.3.

Let's call the first function `f`:

```
void f(char*, int&);
```

Now, let's typedef `PF` to be a pointer to such a function. Because the parameter list '(...)' binds more tightly to the name `PF` than the pointer-to-declarator operator, we need to surround `*PF` with parentheses:

```
typedef void (*PF)(char*, int&);
PF pf; // The requested pointer declaration
```

The next two functions are declared concisely:

```
void g1(PF); // Pointer to function (e.g., f) as argument
PF g2(); // Returns such a pointer
```

Finally, the requested definition:

```
PF g(PF funptr) {
   return funptr;
}
```

Exercise 7.2

What does the following mean? What would it be good for?

```
typedef int (&rifii)(int, int);
```

Hint: Pointers and references to functions are essentially similar. For nonfunctions, this is not the case, however!

Compare this to the type `PF` in the previous exercise. The essential difference is that we now have a reference operator instead of a pointer operator; `rifii` now names the type "reference to function taking two `int` arguments, and returning an `int`."

This type can be used much like pointers to functions are used—to pass functions as arguments to other functions or to store references to functions away for further use. For example,

```
int reduce_intarray(rifii f, int a[], std::size_t n) {
   assert(n);
   int result = a[0];
   for (size_t k = 1; k!=n; ++k)
      result = f(result, a[k]);
   return result;
}
```

Exercise 7.3

Write a program like "Hello, world!" that takes a name as a command-line argument and writes "Hello, name!" Modify this program to take any number of names as arguments to say hello to each.

Hint: See §6.1.7 for a discussion of command-line argument processing.

Because the second variation also handles the first requested program, we will look at only the more general version. This exercise is clearly intended to experiment with the optional function parameters of `main`.

```
#include <iostream>
int main(int argc, char *argv[]) {
   if (argc<2) {
      cout << "Hi! Why not pass me arguments?\n";
   } else {
```

```
      for (int k = 1; k!=argc; ++k)
         cout << "Hello, " << argv[k] << "!\n";
   }
   return 0;
}
```

Exercise 7.4

Write a program that reads an arbitrary number of files whose names are given as command-line arguments and writes them one after another on `cout`. Because this program concatenates its arguments to produce its output, you might call it `cat`.

Hint: A good exercise to illustrate basic file input/output.

A family of fairly popular operating systems includes a utility program called `cat` with this functionality. Our aim here is primarily to develop a program that works and is portable. The versions that come with these operating systems, however, almost certainly make use of nonstandard functions (*system calls*) to improve performance. They also usually add some bells and whistles (that is, command-line options) to this bare-bones specification:

```
#include <iostream>

void cat_stream(istream &input) {
   char ch;
   while (input.get(ch))
      cout.put(ch);
}

int main(int argc, char *argv[]) {
   if (argc<2) {
      cat_stream(cin);
   } else {
      for (int k = 1; k!=argc; ++k) {
         ifstream infile(argv[k]);
         cat_stream(infile);
      }
   }
   return 0;
}
```

If no command-line arguments are given to this program, it will instead copy `cin` to `cout`.

Note also that the destructor of the `infile` objects inside the for loop closes the associated files after every call to `cat_stream`.

Exercise 7.6

Implement `ssort()` (§7.7) using a more efficient sorting algorithm.

Hint: *`qsort`.*

Hint: C++ provides a variety of ways to sort sequences (§17.2.2.1, §18.7, §18.11). In addition, the associative containers (§17.4) also maintain their contents in sorted order.

Because `qsort` is already available to us in the standard C library, we can use it:

```
#include <cstdlib>
#include <stddef.h> // For size_t

typedef  int (*CFT)(void const*, void const*);

void ssort(void *base, size_t n, size_t sz, CFT cmp) {
   std::qsort(base, n, sz, cmp);
}
```

Although the `qsort` routine (§18.11) is well-tuned on most systems, other choices are sometimes preferable. For example, if the comparison function is simple, the cost of a call to `cmp` can be prohibitive. Furthermore, `std::qsort` moves elements around as chunks of bytes. This is sometimes undesirable; the copy-constructor for the objects being sorted should be used instead. These shortcomings can be circumvented by using the standard template `std::sort`.

Also, `qsort` is usually implemented using a variant of the QuickSort algorithm.[1] C.A.R. Hoare[2] showed that this algorithm usually requires less than $k_1\ n\ log\ n$ elementary operations to complete, with k_1 a rather small constant and n the number of elements to sort. However, in the worst case, more than $k_2\ n^2$ operations are needed for some other constant k_2. David Musser (who, along with Alexander Stepanov, was also instrumental in the development of the standard C++ library) devised an interesting combination of QuickSort and HeapSort[1,3], which exploits the strong features of both algorithms: the low value of k_1 for QuickSort in the average case and the $O(n\ log\ n)$ worst-case behavior of HeapSort. Some implementations of the C++ standard library use this IntroSort[4] for `std::sort`.

A simple version of the QuickSort algorithm is implemented in the next chapter.

1. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
2. Hoare, Charles Anthony Richard. "Quicksort." *Computer Journal*, 1962, vol. 5, issue 1, pp. 10–15.
3. Williams, J. W. J. "Algorithms 232 (Heapsort)." *Communications of the ACM*, 1964, vol. 7, pp. 347–348.
4. Musser, David R. "Introspective Sorting and Selection Algorithms." *Software—Practice and Experience*, 1997, vol. 8, pp. 983–993.

Supplementary Exercise

(Simple experiment) Find various implementations of C++ array-sorting algorithms and compare their performance.

Exercise 7.7

Consider the following:

```
struct Tnode {
   string word;
   int count;
   Tnode *left;
   Tnode *right;
};
```

Write a function for entering new words into a tree of Tnodes. Write a function to write out a tree of Tnodes. Write a function to write out a tree of Tnodes with the words in alphabetical order. Modify Tnode so that it stores (only) a pointer to an arbitrarily long word stored as an array of characters on free store using new. Modify the functions to use the new definition of Tnode.

Hint: An exercise about manipulating linked structures. This does not discuss the general algorithmic issue of balanced binary trees.

We'll assume that the tree structure maintains at all times the property that a subtree to the left of a node `n` contains only strings lexicographically (a generalization of alphabetically) preceding the string in node `n`. Similarly, the subtree to the right of node `n` contains only strings that lexicographically come after the string held by `n`.

The first function `enter_word(Tnode*&, string const&)` takes a reference to a pointer to the tree root and a reference to the word (string) that is to be entered. Note that the pointer passed as the root to the tree may need to be modified if the tree was originally empty (null root):

```
Tnode* enter_word(Tnode *&root, string const &word) {
   if (root!=0) {
      // First search for the node in the tree:
      Tnode *node = root;
      do {
         int order = word.compare(node->word);
         if (order==0) { // Word is in the tree already
            ++node->count;
            break;
         } else {
            Tnode *&next = order<0? node->left
                                  : node->right;
```

```
            if (next==0) { // Word not yet in tree?
               next = new_Tnode(word);
               break;
            } else
               node = next;
         }
      }
   } else { // Create the first node at the root
      root = new_Tnode(word);
   }
}
```

The helper function `new_Tnode(string const&)` fulfills a role akin to a constructor. (If `Tnode` hadn't been a given, you would probably have equipped it with such a constructor).

```
Tnode* new_Tnode(string const *word) {
   Tnode *node = new Tnode;
   node->word = word;
   node->count = 1;
   node->left = node->right = 0;
   return node;
}
```

We'll write out the tree in depth-first infix order, which means that each node first asks its left node to be printed, prints itself, and then asks its right node to be printed. Because we maintain the lexicographical ordering of the nodes, printing them out in this manner also satisfies the requirement to write the words in alphabetical order:

```
void write(std::ostream &output, Tnode *node,
           bool indent, int spaces = 2) {
   if (node) {
      write(output, node->left, indent, spaces+2);
      if (indent)
         for (int k = 0; k!=spaces; ++k)
            cout << ' ';
      output << node->word
             << " (" << node->count << ")\n";
      write(output, node->right, indent, spaces+2);
   }
}
```

To allow a form of output that displays the tree structure, we added the Boolean parameter `indent`, which specifies whether an indentation based on the level of a word inside the tree should be used. The parameter `spaces` specifies how deep the indentation should be if `indent` is true.

Observe that our implementation of `enter_word` used the member function `compare` for standard `strings`. This member function will return a value smaller than, equal to, or

larger than zero, depending on whether the target string precedes, equals, or comes after the argument string. This is essentially equivalent to the function `strcmp` for C-style strings (null-terminated arrays of characters).

Supplementary Exercise

(A simple, nonfundamental adaptation) Convert the functions developed above to use C-style strings (null-terminated character arrays) instead of the C++ `string` type.

Exercise 7.9

Write an encryption program that reads from `cin` and writes the encoded characters to `cout`. You might use this simple encryption scheme: The encrypted form of a character `c` is `c^key[i]` where `key` is a string passed as a command-line argument. The program uses the characters in `key` in a cyclic manner until all the input has been read. Reencrypting encoded text with the same `key` produces the original text. If no `key` (or a null string) is passed, then no encryption is done.

Hint: An illustration of the bitwise exclusive-or operator.

This is not as hard as it may look:

```
int main(int argc, char *argv[]) {
   char *key = (argc>=2)? argv[1]: "";
   size_t key_length = (argc>=2)? std::strlen(key): 1;
   for (std::size_t k = 0;
        std::cin.getc(ch); k = (k+1)%key_length)
      std::cout.putc(ch^key[k]);
   return 0;
}
```

The cyclic effect is achieved using the remainder-after-division operator (%). The program also relies on the fact that the only element of an empty string is '\0' and that the exclusive-or operation with '\0' has no effect.

Exercise 7.11

Write an error function that takes a `printf`-style format string containing `%s`, `%c`, and `%d` directives and an arbitrary number of arguments. Don't use `printf()`. Look at §21.8 if you don't know the meaning of `%s`, `%c`, and `%d`. Use `<cstdarg>`.

Hint: See §7.6 for details on how to handle variable-length argument lists.

Remember that if your compiler does not support the header `<cstdarg>`, it probably does support `<stdarg.h>` which is equivalent, except that the latter places names in the global

namespace. However, because <cstdarg> really consists of macro names, and because macro names do not belong to namespaces, there is little difference between <cstdarg> and <stdarg.h>.

Variable-argument lists are handled using an iterator-like device of type va_list. The macro va_start initializes this device, while the macro va_end must be used to indicate that the device will not be used again. To obtain the value of the current argument and move the device to the next argument, use the macro va_arg. The end of the argument list must be determined from the arguments visible so far (typically, from one of the arguments not grouped under the ellipsis notation).

```
void error(char const *fmt, ...) {
   assert(fmt);
   va_list p;
   va_start(p, fmt);
   for (char const *s = fmt; *s; ++s)
      if (*s!='%')
         cerr.put(*s);
      else
         switch (*(++s)) {
         case '%': cerr.put('%'); break;
         case 's': cerr << va_arg(p, char const*); break;
         case 'c': cerr << va_arg(p, char); break;
         case 'd': cerr << va_arg(p, int); break;
         default: throw std::domain_error(std::string ("panic"));
         }
   va_end(p);
}
```

Note that the format specifiers for printf have many more options that you also might want to integrate into the above function. Although this is not particularly difficult, it will add a considerable amount of code.

Exercise 7.14

What is wrong with these macro definitions?

```
#define PI = 3.141593;
#define MAX(a, b) a>b? a: b
#define fac(a) (a)*fac((a)-1)
```

Hint: See §7.8 for information about macros. Macros ignore C++ typing and scoping rules and are, therefore, best avoided when reasonable alternatives exist.

The first macro says that each instance of PI should be replaced by '= 3.141593;', which is almost certainly unintentional. More likely, PI should be substituted by '3.141593', which is achieved with

```
#define PI 3.141593
```

However, it is safer to use

```
double const PI = 3.141593;
```

instead, because this tells the compiler that `PI` is really the name of a double precision constant rather than just a sequence of characters.

The second macro is dangerous because the macro parameters are not parenthesized. This may result in unexpected interpretations when the macro gets expanded. A safer alternative is

```
#define MAX(a, b) ((a>b)? (a): (b))
```

but even so, one may be surprised when side effects of the macro arguments occur more than once. We're almost always better off using a solution that fits in the language itself:

```
template<typename T> inline
T max(T const &a, T const &b) {
   return a>b? a: b;
}
```

Finally, the macro `fac` is recursive, so that if we tried to expand it, it would result in an infinite replacement. However, your compiler will report an error when it sees the macro. Again, a better alternative is to use a function (or a template).

Exercise 7.16

Implement `print()` from §7.5.

Hint: Try Exercise 6.17 first.

This exercise is similar to Exercise 6.17. Let's assume that the largest integer can be represented with no more than 129 characters (one for the sign and at most 128 bits in base 2 representation).

```
namespace { digit[] = "0123456789ABCDEF"; }

void print(int value, int base = 10) {
   int const MXS = 128; // Maximum number of digits
   assert((numeric_limits<int>::radix==2)
      and (numeric_limits<int>::digits<=MXS));
   if (base<2 or base>16)
      throw std::domain_error(std::string ("Print error"));
   char rep[MXS+2];
   rep[MXS+1] = '\0'; // Place an end-of-string marker
   if (value<0) { // Add in the sign if needed
      rep[0] = '-'; // This may have to move later
      value = -value;
```

```
    } else
        rep[0] = '+';
    char *p = rep+MXS; // Start at the end
    do { // Move back, inserting digits as you go
        *p-- = digit[value%base];
        value = value/base;
    } while (value);
    *p = rep[0]; // Place the sign just before the most
                 // significant digit.
    cout << p;
}
```

To easily fix the size of the array in which a representation of the value is composed, I made the additional assumption that the internal representation of integers uses base 2. Platforms on which this is not the case are extremely rare.

Exercise 7.18

Write a factorial function that does not use recursion. See Exercise §11.6.

Hint: See §7.3 for an example of a recursive factorial function.

Here is a solution that checks its input argument:

```
int factorial(int a) {
    if (a<0) throw std::domain_error(string("negative factorial"));
    int result = 1;
    while (a>1)
        result *= a--;
    return result;
}
```

Remember that very deep recursion is undesirable (because, for example, most systems have a limit on how deeply function calls can be nested). Recursion does provide a natural and concise solution to many problems in which the depth of recursion can be reasonably bounded. Compilers for certain programming languages automatically convert some forms of recursion into a simple iteration, but this is still not common for C++ compilers.

Exercise 7.19

Write functions to add one day, one month, and one year to a `Date` as defined in Exercise 5.13. Write a function that gives the day of the week for a given `Date`. Write a function that gives the `Date` of the first Monday following a given `Date`.

Hint: Try Exercise 5.13 first. The discussion of Exercise 10.1 is also pertinent. The last part of this exercise delves a little into calendar theory. Also see Exercise 10.2.

Here is the `Date` class as we defined it for the exercise in Chapter 5:

```
struct Date {
   unsigned day_: 5;
   unsigned month_: 4;
   int year_: 15;
};
```

Let's add a `Month` class to it:

```
struct Month {
   enum { Jan = 1, Feb, Mar, Apr, May, Jun,
          Jul, Aug, Sep, Oct, Nov, Dec };
   static int n_days(int m, int y) {
      return m!=Feb? n_days_[m-1]
                   : (is_leapyear(y)? 29: 28);
   }
private:
   static int const n_days_[12];
};

int Month::n_days_[12]
   = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

The implementation of `Month::n_days` reveals the need for a function that discriminates leap years:

```
bool is_leapyear(int y) {
   return !(y%4) and ((y%100) or !(y%400));
} // y is a multiple of 4, but not a multiple of 100
  // that is not a multiple of 400.
```

With all those preparations, adding the functions that add a day, month, or year requires little additional code:

```
Date next_year(Date const &d) {
   Date n;
   n.day_ = d.day_;
   n.month_ = d.month_;
   n.year_ = d.year_+1;
   // February 29, but not a leap year?
   if (n.day_>Month::n_days(n.month_, n.year_)) {
      n.day_ = 1; n.month_ = 3;
   }
   return n;
}
```

```
Date next_month(Date const &d) {
   Date n;
   n.day_ = d.day_;
   n.month_ = d.month_%12+1;
   n.year_ = d.year_;
   if (n.month_==1)
      n = next_year(n);
   if (n.day_>Month::n_days(n.month_, n.year_))
      n.day_ = Month::n_days(n.month_, n.year_);
   return n;
}

Date next_day(Date const &d) {
   Date n;
   n.day_ = (d.day_)%Month::n_days(d.month_, d.year_)+1;
   n.month_ = d.month_;
   n.year_ = d.year_;
   if (n.day_==1)
      return next_month(n);
   else
      return n;
}
```

I again make use of the remainder-after-division operator to handle the cyclic character of days and months. Note how this operator naturally expresses cycles over ranges starting at zero (for example, 0...11, not 1...12). Hence, we must be careful when applying it to our numbering of days or months: The first month or day is '1.'

Finding the day of the week, given a `Date`, can be done concisely. Every year shifts January 1 forward by one day of the week except for leap years, which shift it by two days. For example, the Western New Year's Day in 1996 (a leap year) fell on a Monday. So in 1997, it fell on a Wednesday (that is, two days later); in 1998, January 1 was a Thursday, one day later because 1997 was not a leap year. So if we're in nonleap year *Y*, we have shifted $\left[Y-1+\left(\frac{Y-1}{4}-\frac{Y-1}{100}+\frac{Y-1}{400}\right)\right]\ \%\ 7$ weekdays since the same date of year 1; the expression in parentheses is the number of leap years since that date (assuming integer division). Had we been in a leap year on a day in March or later, we would have to shift one more day to account for last February 29. You can also verify that in a non-leap year, the first day of month *M* is shifted `MS[`*M*`-1]` weekdays with respect to January 1 of that year, where `MS` is the array:

```
int MS[] = { 0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5 };
```

Again, for the months March through December of a leap year, you'd have to shift by one more weekday. In this system, year 1 started on a Monday (day 1); ignore the fact our Gregorian calendar system did not exist back then. Our algorithm will not work for dates much

more than two centuries or so back in time (see Exercise 10.1). Armed with all this, the required function can be written as:

```
enum DayOfWeek { Sunday = 0, Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday };

DayOfWeek day_of_week(Date const &dt) {
   int const y = dt.year_, m = dt.month_, d = dt.day_;
   int const j1 = (1+y-1+(y-1)/4-(y-1)/100+(y-1)/400)%7;
   if (m<=2 or !is_leap_year(y))
      return DayOfWeek((j1+MS[m-1]+d-1)%7);
   else
      return DayOfWeek((j1+MS[m-1]+d)%7);
}
```

Finding the Monday following a given `Date` is then a matter of combining the functions we already have:

```
Date next_Monday(Date const &d) {
   DayOfWeek dday = day_of_week(d);
   int const n_days = (dday==Sunday)? 1: 7-dday;
   Date Monday(d);
   for (int k = 0; k!=n_days; ++k)
      Monday = next_day(Monday);
   return Monday;
}
```

The solutions presented here rely on "first principles" of the Gregorian calendar. Although the resulting implementation is rather efficient, it is easy to make mistakes in working out the detailed computations. The C standard library provides a powerful alternative in the function `mktime`. Exercise 10.2 shows how it can be used; it also has the ability to determine the day of the week (like `day_of_week`).

Supplementary Exercise

Prove that the following version of `day_of_week` is equivalent to the one above.

Hint: For positive integers `a`, `b`, and `c`, `(a+b)%c==(a%c+b%c)%c`.

```
DayOfWeek day_of_week(Date const &d) {
   static int const MS[] = {
      0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
   int const m = dt.month_, d = dt.day_;
   int const y = (m<=2)? dt.year_-1: dt.year_;
   int const day = (y+y/4-y/100+y/400+MS[m-1]+d)%7;
   return DayOfWeek(day);
}
```

[blank page]

Chapter 8

Namespaces and Exceptions

Classes and functions are powerful building blocks for software development. They provide mechanisms to separate cooperating interfaces from the details of the implementation. The logistics of very large software do not always allow for tight cooperation, however. Although classes and functions still play a dominant role in the organization of software, the need for new tools becomes apparent. These tools must allow teams of developers to contribute to a single software system without necessarily knowing the details of each other's interfaces. Namespaces (§8.2), exceptions (§8.3), and templates (§13) are such tools. Namespaces deal primarily with the problem of *name collision*. Certain terms are so natural and commonplace in program sources that two teams in a larger project may decide to use identical names for different entities. Namespaces allow us to resolve potential conflicts arising from such double uses. Exceptions—unlike namespaces—are a run-time mechanism. They allow a component that has control of the thread of execution to transmit the control back to another—a priori unknown—component. This is typically required only when exceptional situations that cannot be dealt with locally are encountered. Templates are discussed further in another chapter.

Exercise 8.1

Write a doubly-linked list of `string` module in the style of the `Stack` module from §2.4. Exercise it by creating a list of names of programming languages. Provide a `sort()` function for that list, and provide a function that reverses the order of the strings in it.

Hint: Modular programing is introduced in §2.4. You might prefer to implement the `sort()` function last because that is the hardest part to get right. See Exercises 18.5 and 18.19.

Here is the code for the module, with interspersed comments:

```
#include <assert.h>
#include <string>
#include <algorithm> // for std::swap

namespace StringList {
   using std::string;
   using std::swap;
```

This module will manage a single list of strings consisting of individual nodes described by the following structure. Two nodes—`head_node` and `tail_node`—delimiting the beginning and the end of the list will always be present (but will not hold a user-inserted string). This technique occasionally simplifies coding by eliminating boundary conditions.

```
   struct Node {
      string string_;
      Node *prev_;
      Node *next_;
   } head_node, tail_node;
```

`Cursor` is the type that will be used to indicate position in the list. In this case, a simple pointer to a node will suffice. `Cursor` is similar to a *standard iterator* (§19.2).

```
   typedef Node *Cursor;
```

Before using the list in this module, its head and tail nodes must be primed to point at each other. It is the user's responsibility to call this function before other operations are applied to the list.

```
   void init() {
      head_node.prev_ = tail_node.next_ = 0;
      head_node.next_ = &tail_node;
      tail_node.prev_ = &head_node;
   }
```

A few simple functions are all that is needed to allow bidirectional navigation through the list:

```
   inline Cursor next(Cursor c) {
      assert(c);
      return c->next_;
   }

   inline Cursor prev(Cursor c) {
      assert(c);
      return c->prev_;
   }
```

```
inline Cursor head() {
   return &head_node;
}

inline Cursor tail() {
   return &tail_node;
}
```

Extracting strings from the list and inserting new ones into it is a little trickier. In particular, two kinds of insertions at a given location can make sense: before and after the node indicated by the argument `Cursor`.

```
string extract(Cursor c) {
   assert(c!=head() and c!=tail());
   string s = c->string_;
   c->prev_->next_ = c->next_;
   c->next_->prev_ = c->prev_;
   delete c;
   return s;
}

Cursor insert_after(Cursor c, string s) {
   assert(c!=tail());
   Node *n = new Node;
   n->string_ = s;
   n->prev_ = c; n->next_ = c->next_;
   c->next_ = n; n->next_->prev_ = n;
   return n;
}

Cursor insert_before(Cursor c, string s) {
   assert(c!=head());
   Node *n = new Node;
   n->string_ = s;
   n->prev_ = c->prev_; n->next_ = c;
   c->prev_ = n; n->prev_->next_ = n;
   return n;
}
```

Reversing a list is a traditional algorithm that turns out to be relatively straightforward when bidirectional navigation is available. See Exercise 5.4 for a discussion of functions that swap their arguments, and see §18.6.8 for a description of the corresponding standard function.

```
void reverse() {
   // First exchange the prev_ and next_ pointers
   // inside each internal node:
```

```
    for (Cursor c = next(head()); c!=tail(); c = c->prev_)
        swap(c->prev_, c->next_);
    // Then exchange the head and tail positions:
    swap(head()->next_, tail()->prev_);
}
```

For the `sort` function, I use a simple version of the QuickSort algorithm of C.A.R. Hoare. It is often described as a method to sort arrays. Here you can see that, in fact, it is also applicable to doubly linked lists. However, the crucial choice of selecting a *pivot element* is less flexible. The `sort` function makes use of a helper function that is placed in an *unnamed namespace* (§8.2.5.1) to hide it from client code altogether.

```
namespace { // unnamed namespace
    void quicksort(Cursor left, Cursor right) {
        // The input partition is described by cursors
        // pointing to nodes just outside its boundaries:
        Cursor p = left->next_; // The pivot
        Cursor l = left->next_, r = right->prev_;
        // Zero- or one-element lists are trivially sorted:
        if (left==right || l==right || l==r)
            return;
        // Partition the list into elements smaller and
        // larger than the pivot value:
        while (true) {
            while (l!=r && l->string_<p->string_)
                l = next(l);
            while (l!=r && p->string_<=r->string_)
                r = prev(r);
            if (l==r)
                break;
            swap(l->string_, r->string_);
        }
        // Establising the boundaries of both partitions
        // is more delicate when dealing with doubly linked
        // than the version for arrays:
        if (l->string_<p->string_) {
            if (r->next_!=right)
                l = next(l);
        } else {
            if (left->next_!=r)
                r = prev(r);
        }
        // Recursively partition these partitions:
        quicksort(left, l);
        quicksort(r, right);
    }
} // End unnamed namespace
```

The details of the QuickSort algorithm are not so important to illustrate the use of namespaces in this exercise. If you're unfamiliar with the algorithm, you will find its description in many introductory algorithms textbooks.

```
    void sort() {
       if (head()->next_!=tail()
       and head()->next_->next_!=tail())
          quicksort(head()->next_, tail()->prev_);
    }
} // End namespace StringList
```

Supplementary Exercise

(Moderately tedious, but interesting experiment) See if you can eliminate one or both of the guard nodes without degrading the performance of common operations. Rather than exchange `string` values, you could exchange the nodes within the list by modifying `prev_` and `next_` pointers. Implement this alternative and compare its performance to the above.

This little piece of code would become much more useful if the module organization were replaced by an object-based organization.

Supplementary Exercise

(Fundamental; see §2.5) Reorganize the code above into a class or into multiple cooperating classes. This should make it convenient to have multiple `StringLists` in your program.

Here is some code that exercises some of the preceding functions:

```
#include <iostream>

void print_stringlist() {
   StringList::Cursor p = StringList::head();
   while ((p = StringList::next(p))!=StringList::tail())
      std::cout << p->string_ << std::endl;
}

int main() {
   StringList::init();
   insert_before(StringList::tail(), string("Cobol"));
   insert_before(StringList::tail(), string("Fortran"));
   insert_before(StringList::tail(), string("Lisp"));
   insert_before(StringList::tail(), string("Algol 68"));
   insert_before(StringList::tail(), string("Basic"));
   insert_before(StringList::tail(), string("Simula"));
```

```
    insert_before(StringList::tail(), string("C"));
    insert_before(StringList::tail(), string("Ada"));
    insert_before(StringList::tail(), string("Modula-2"));
    insert_before(StringList::tail(), string("C++"));
    print_stringlist();
    StringList::reverse();
    print_stringlist();
    StringList::sort();
    print_stringlist();
    return 0;
}
```

Notice how the calls to `StringList::insert_before` are not qualified with the '`StringList::`' prefix. This is possible because of *argument-dependent lookup*—unqualified function names, such as `insert_before`, are looked up in the namespaces associated with the types of the arguments (if any). In this case, the `StringList::tail()` argument returns the type `Node` associated with the `StringList` namespace. Hence, `insert_before` is looked up in that namespace.

Exercise 8.4

Write a program that throws an exception in one function and catches it in another.

Hint: Try variations on this fundamental exercise. See Exercises 8.5 through 8.8.

A question with a brief answer:

```
#include <iostream>

void f() {
    throw "Bomb!";
}

int main() {
    try {
        f();
    } catch(char const* s) {
        std::cout << "Caught " << s << std::endl;
    }
    return 0;
}
```

It is usually preferable to use user-defined types when throwing exceptions (§8.3.2).

Exercise 8.5

Write a program consisting of functions calling each other to a calling depth of 10. Give each function an argument that determines at which level an exception is thrown. Have `main()` catch these exceptions and print out which exception is caught. Don't forget the case in which an exception is caught in the function that throws it.

Hint: Try the previous exercise first.

To keep the code compact, it's convenient to use a recursive function. Each time this function calls itself with an unmodified argument, it increments a global variable. Once the global variable equals the argument, an exception is thrown.

```
#include <iostream>
#include <cstdlib>

int level = 0;

void f(int throw_level) {
   ++level;
   if (throw_level==level)
      throw level;
   else
      f(throw_level);
}

int main(int argc, char* argv[]) {
   try {
      if (argc<2 or std::atoi(argv[1])<1)
         throw 0;
      else
         f(std::atoi(argv[1]));
   } catch (int depth) {
      std::cout << "Caught level "
                << depth << " exception.\n";
   }
   return 0;
}
```

The function `main` checks the parameter `argc` to see how many command-line arguments it is passed through `argv`. If none (`argc` < 2), it will throw an exception right away. Otherwise, it will convert the first argument (a `char` array) to an integer—with `std::atoi`— and use that to determine how deep recursion should progress before an exception is thrown.

Supplementary Exercise

(Fundamental skill) Eliminate the global variable `level` by introducing a second parameter for `f`.

Exercise 8.6

Modify the program from Exercise 8.5 to measure if there is a difference in the cost of catching exceptions depending on where in a class stack the exception is thrown. Add a `string` object to each function and measure again.

Hint: This is an enlightening experiment. Be sure to come back to this exercise if you decide to skip it at first.

To measure cost, we can use the C function `clock()` and measure how many catches we can perform in, say, five seconds:

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>

int level;

void f_without_string(int throw_level) {
   ++level;
   if (throw_level==level)
      throw level;
   else
      f_without_string(throw_level);
}

void f_with_string(int throw_level) {
   std::string s;
   ++level;
   if (throw_level==level)
      throw level;
   else
      f_with_string(throw_level);
}

int main(int argc, char* argv[]) {
   int throw_level = (argc<2 or std::atoi(argv[1])<1)?
                     0: std::atoi(argv[1]);
   unsigned long catch_count = 0;
   std::clock_t end = std::clock()+5*CLOCKS_PER_SEC;
```

```
    do {
       level = 0;
       try {
          if (throw_level==0)
             throw 0;
          else
             f_without_string(throw_level);
       } catch (int depth) {
          ++catch_count;
       }
    } while (std::clock()<end);
    std::cout << "Without string: " << catch_count/5
                                    << "catches/second.\n";
    catch_count = 0;
    end = std::clock()+5*CLOCKS_PER_SEC;
    do {
       level = 0;
       try {
          if (throw_level==0)
             throw 0;
          else
             f_with_string(throw_level);
       } catch (int depth) {
          ++catch_count;
       }
    } while (std::clock()<end);
    std::cout << "With string: " << catch_count/5
                                 << "catches/second\n";
    return 0;
}
```

In one of the environments I use, the number of catches without the local `string` is rather insensitive to the throw level (between 11,000 and 12,000 catches per second). Because I know the compiler I was using will not inline function calls before I ask for it, and because obviously increasing the recursion through the throw level increases the amount of work performed, I can only conclude that the cost of throwing and/or catching the exception dominates the cost of the recursive call by far. This is not unusual: C++ exceptions were designed with the intention that the addition of exception handling would not degrade performance by much if no exceptions are actually thrown. However, because a throw ought to be rather exceptional, it is usually not objectionable that such an operation takes some more time.

On the same platform, I found that the catch rate with the local `string` decreases gradually when the throw level increases above 0. When the throw level reaches 10, the catch rate is about half that of the version without the local `string`. This is a consequence of the increasing number of `string`s that must be destroyed.

Your environment could produce somewhat different results, but you're likely to see similar trends.

Exercise 8.7

Find the error in the first version of `main()` in §8.3.3.1.

Hint: A good illustration of how subtle programming logic can be (independent of the programming language).

Suppose `Parser::prim()` encounters a syntax error. An exception is thrown, which is caught by `main()`. The catch clause in `main()` issues an appropriate error message, and the program resumes from just after the offending token. However, it is very likely that the input starting at that point does not correspond to a valid expression, so we will quickly throw another exception (with an associated error message). This scenario may repeat itself several times until we finally hit the end-of-line marker, which brings us back to a sane state.

Exercise 8.8

Write a function that either returns a value or throws that value based on an argument. Measure the difference in run-time between the two ways.

Hint: Compare this to Exercise 8.6. The conclusion of this experiment is important.

This exercise is similar to Exercise 8.6. The code below uses the rate of returns as a measure of run-time speed.

```
#include <iostream>
#include <ctime>

int seven(bool do_throw) {
   if (do_throw)
      throw 7;
   else return 7;
}

int main() {
   unsigned long return_count = 0;
   std::clock_t end = std::clock()+5*CLOCKS_PER_SEC;
   do {
      int sept = seven(false);
      ++return_count;
   } while (std::clock()<end);
   std::cout << "Plain return: " << return_count/5
                                 << " times per second\n";
```

```
    return_count = 0;
    end = std::clock()+5*CLOCKS_PER_SEC;
    do {
       try {
          seven(true);
       } catch(int sept) {
          ++return_count;
       }
    } while (std::clock()<end);
    std::cout << "Thrown return: " << return_count/5
                                    << " times per second\n";
    return 0;
}
```

Using the same environment I used for Exercise 8.5, I find that my machine performs more than 2 million plain returns per second, whereas only about 12,000 throws per second are achieved.

This illustrates once more that using the exception-handling mechanism for non-exceptional situations is generally not a good idea.

Exercise 8.10

Write `plus()`, `minus()`, `multiply()`, and `divide()` functions that check for possible overflow and underflow and that throw exceptions if such errors happen.

Hint: See also Exercise 4.5.

Performing these tests in a portable way requires some care. It is easy to overflow or underflow during the check itself. I'll write functions for double precision floating-point arguments, but other types can be handled in a similar way. In fact, you could write function templates that provide this functionality for a variety of types in a compact way (although that would be trickier).

```
#include <limits>

string const op_err("arithmetic error");

double plus(double a, double b) {
   if ((b>0.0 && a>std::numeric_limits<double>::max()-b)
    || (b<0.0 && a<std::numeric_limits<double>::min()-b))
      throw std::overflow_error(op_error);
   else
      return a+b;
}
```

We could write almost exactly the same for minus, or take the following shortcut:

```
inline double minus(double a, double b) {
   return plus(a, -b);
}
```

For these additive operators, I've ignored the possibility of underflow (that is, the result of the operation being zero even though it should not mathematically be zero). Many architectures won't underflow with plus and minus. If you needed to, you could use the same method as that used below for the multiplicative operations:

```
double multiply(double a, double b) {
   if (std::fabs(b)>1.0
    && std::fabs(a)>std::numeric_limits<double>::max()
                   /std::fabs(b))
      throw std::overflow_error(op_error);
   double r = a*b;
   if (a!=0.0 && b!=0.0 && r==0.0)
      throw std::underflow_error(op_error);
   return r;
}
```

The header <cmath> (or <math.h>) defines the absolute value function `fabs`. Note also how the `min` static member of the `numeric_limits` template has very different meanings for integral and floating types. Indeed, for floating types, it returns the smallest strictly positive representable value; whereas, for integral types, it returns the algebraically smallest value, period—that is, a negative value. So, for floating types, we assume that the smallest representable value equals the negative of the largest representable value.

```
double divide(double a, double b) {
   if (std::fabs(b)<1.0
    && std::fabs(a)>std::numeric_limits<double>::max()
                  *std::fabs(b))
      throw std::overflow_error(op_error);
   double r = a/b;
   if (a!=0.0 && r==0.0)
      throw std::underflow_error(op_error);
   return r;
}
```

You'll find that these functions are generally much less efficient than the plain built-in operations on which they build. If you need these sorts of operations combined with maximum performance, many platforms offer ways to detect whether overflow or underflow occurred after the fact and at a much reduced cost. Exploiting this platform-specific functionality is outside the realm of standard C++.

Chapter 9

Source Files and Programs

Although learning abstract contructs, idioms, and techniques is an important part of learning a new programming language, it is just as important to acquaint yourself with the logistics of the *physical organization* of your software. One aspect of that side of software development is the mapping of your code onto your computer's file system for maximum productivity. Productivity is then a combination of the cost of maintenance, the speed of an *edit-compile-link-test cycle*, and so forth. The exercises in this chapter explore some of these issues for typical C++ systems. A facet that is not explored is the organization of the products of the build process. That problem depends largely on the development and target systems for the software, as well as on policies and customs of the team or organization producing the software.

Exercise 9.1

Find where the standard library headers are kept on your system. List their names. Are any nonstandard headers kept together with the standard ones? Can any nonstandard headers be `#include`d using the <> notation?

Hint: Familiarize yourself with your C++ programming environment. See Exercise 4.1.

The answer to this exercise depends on your particular system, but a few general remarks can be made. The first one is that the C++ standard does not mandate that the standard headers actually correspond to certain files on your system. The '`#include <stdheader>`' syntax may well trigger some magic inside your compiler that enables functionality that was there all along. Still, headers very often do map on specific files. Yet, sometimes, these files contain "magic words" that notify the compiler of some special functionality (this is often the case, for example, with the header `<stdarg.h>`).

The other observation worth making here is that the C and C++ programming languages have become the systems programming language of choice on many platforms. Various header files describing the interfaces to these systems' core libraries are then conveniently

treated using the same notation as the standard headers. They are also often placed in the same location as the files describing the standard C and C++ libraries. Finally, many compilation systems also allow their users to specify that certain locations on a file system be accessible using the <> notation.

Exercise 9.5

An external include guard is a construct that tests outside the file it is guarding and includes it only once per compilation. Define such a construct, devise a way of testing it, and discuss its advantages and disadvantages compared to the include guards described in §9.3.3. Is there any significant run-time advantage to external `#include` guards on your system?

Hint: Read §9.3.3. Include guards an essential technique in C and C++.

Such an external include guard can look like the following:

```
#ifndef FILE_H
#define FILE_H
#include "file.h"
#endif
```

This sort of construct gained popularity when it was observed that opening an included file, finding the internal guard at the top of that file, and skipping most of the file to the find end of the guarded file takes a significant amount of time. This appeared to be especially true when using distributed file systems (which are very convenient in team-oriented software development environments).

Unfortunately, external include guards not only make source *look* more complex, they also increase the complexity of their *maintenance*. Indeed, maintaining the guard is no longer restricted to one well-known place, but to all files that are clients to the header file. Furthermore, many systems now recognize the presence of an internal include guard. When they encounter another `#include` directive for such a previously encountered guarded file, they will simply ignore the directive. Hence, run-time analysis of the compilation process may not be a very good foundation to test the external guard device. Instead, you might try to rely on a side effect of multiple inclusion. For example, if the include file includes a definition of a non-inline function, multiple inclusion should trigger a diagnostic by your compiler.

Exercise 9.8

Modify the desk calculator so that it can be invoked from `main()` or from other functions as a simple function call.

Hint: The key issue here is the initialization of global objects. See §9.4.1.

This could be done in a variety of ways and with all kinds of bells and whistles (for example, to allow multiple calculators to be simultaneously active within a program). However, the most basic solution involves renaming the function `main` to something else—for example, `start_calculator`. That function would then presumably be placed in namespace `Driver`.

Next you will notice that the initializations (§9.4.1) of global objects that were formerly automatically performed before executing the first statement of `main` are not automatically performed when you call `start_calculator`. These initializations (for example, setting the initial value of `Driver::no_of_errors` to zero) can be added to the `start_calculator` function, or they can be placed in a separate function that must be called explicitly prior to invoking `start_calculator`.

Exercise 9.9

Draw the "module dependency diagrams" (§9.3.2) for the version of the calculator that used `error()` instead of exceptions (§8.2.2).

Hint: Some programming tools produce such diagrams automatically.

The only difference between the diagrams in §9.3.2 and the ones that answers this question is that we now must accommodate the function `error()`. A natural choice would be a file called `error.c`. Strictly speaking, this file needs to include only the standard `<iostream>` header (and no file needs to include `error.c`). However, I like to include its associated interface file (in this case, `error.h`) as an additional protection against inconsistencies between interface and implementation (which the compiler would diagnose when compiling `error.c`).

[blank page]

Chapter 10

Classes

The class notion is the foundation that makes C++ a programming language that supports *object-oriented design*. It blends data interfaces (C structs) and function interfaces (C functions) into a unit that more naturally *models* concepts from the application domain. Identifying which concepts to represent as classes and how detailed to model reality is a skill that leads to successful software development. This chapter contains a selection of exercises illustrating how to use existing classes, how to design class interfaces, and how to implement these interfaces.

Exercise 10.1

Find the error in `Date::add_year()` in §10.2.2. Then find two additional errors in the version in §10.2.7.

Hint: Leap years are one of the many irregularities that make calendar theory tricky. See Exercises 5.13, 7.19 and 10.2 for code examples.

The version in §10.2.2 does not take into account the possibility that the date before the operation is February 29, while the year after the operation is not a leap year. The version in §10.2.7 fixes this, making it suitable for most business situations. However, if you need to cover all times of known history, the problem becomes more complex. . . much more complex. The calendar used most in the world today (at least officially) is the Gregorian calendar. It replaced the Julian calendar, which had a slightly different definition of leap year, in a large part of Europe on October 15, 1582. In the Julian calendar, this was October 5, 1582, so about 10 days "disappeared" for those countries that adopted the Gregorian calendar at that time. Other countries adopted the calendar later, with similar strange effects. What the `Date` class in §10.2.7 also ignores is the fact that there is no year 0. So, if we decide that year -46 really is 46 B.C. (the year the Julian calendar was introduced), we will need to correct the `add_year` member to skip zero.

Other difficulties with historical dates are the introduction of new calendars as part of revolutions (for example, the French introduced a Revolutionary Calendar with 12 months containing 3 "weeks" of 10 days plus 5 "intercalary" days) or under influence of various religions (the Gregorian calendar was, in fact, pushed by the Roman Catholic church). Furthermore, many cultures still use lunar calendars today.

For extensive information on this topic, read Dershowitz and Reingold's *Calendrical Calculations*.[1]

Exercise 10.2

Complete and test `Date`. Reimplement it with "number of days after 1/1/1970" representation.

Hint: Compare this to the calculations performed in Exercise 7.19.

I will not provide a complete implementation of a `Date` class. However, it is probably worth pointing out that C++ inherits from the C library a very powerful function, `mktime`. As an illustration, let's see how simple functions can be written to convert between the day/month/year representation and the "number of days since 1/1/1970" representation.

```
#include <ctime>
void A1Jan1970_to_DMY(int days,
                      int &day, int &month, int &year) {
    std::tm tm;
    tm.tm_sec = 0;
    tm.tm_min = 0;
    tm tm_hour = 12;
    tm.tm_mday = 1+days;
    tm.tm_mon = 0;
    tm.tm_year = 70;
    std::mktime(&tm); // Will "normalize" the
                      // std::tm representation
    day = tm.tm_mday;
    month = tm.tm_mon;
    year = tm.tm_year;
}
```

The `std::tm` structure holds a representation of the date in which the year is represented as an offset with respect to the year 1900. The function `mktime` is primarily intended as a mechanism to convert the `std::tm` format into the number of seconds elapsed since the first moment of 1970. However, it also has the interesting side effect of normalizing its vari-

1. Dershowitz, Nachum and Edward M. Reingold. *Calendrical Calculations*. Cambridge, UK: Cambridge University Press, 1997.

ous components (that is, January 32, will essentially adjust to February 1). That is the property exploited above. We use the primary functionality to implement the inverse conversion:

```
void DMY_to_A1Jan1970(int day, int month, int year,
                      int& days) {
    std::tm tm;
    tm.tm_sec = 0;
    tm.tm_min = 0;
    tm tm_hour = 12;
    tm.tm_mday = day;
    tm.tm_mon = 1+month;
    tm.tm_year = year-1900;
    std::time_t seconds = std::mktime(&tm);
    days = (seconds/86400)-1;
}
```

Note that these routines usually have a limited scope because the `mktime` routine and `time_t` type cannot necessarily handle more than a few decades (a few billion seconds).

Exercise 10.4

How do you access `set_default` from class `Date` from namespace `Chrono` (§10.3.2)? Find at least three different ways.

Hint: This exercise allows you to verify your knowledge of fundamental C++ syntax. See §10.2.4.

The `set_default` member of the `Chrono::Date` class is a static member function. As such, it need not have a target object—that is, the use of a dot (as in `target.mf()`) or an arrow operator (as in `ptr->mf()`) is not required.

A first method to access `set_default` is to use fully qualified names everywhere:

```
Chrono::Date::set_default(1, Chrono::Date::jan, 1970);
```

A second exploits a using-declaration (§8.2.2), which is especially convenient if a namespace name is used repeatedly in the scope of that using declaration:

```
using Chrono::Date;
Date::set_default(1, Date::jan, 1970);
```

A very similar effect is achieved using a typedef declaration:

```
typedef Chrono::Date MyDate;
MyDate::set_default(1, MyDate::jan, 1970);
```

Finally, we could use the usual member access notation:

```
Chrono::Date d;
d.set_default(1, Chrono::Date::jan, 1970);
```

Exercise 10.5

Define a class `Histogram` that keeps count of numbers in some intervals specified as arguments to `Histogram`'s constructor. Provide functions to print out the histogram. Handle out-of-range values.

Hint: A useful illustration of reusable classes and a great addition to every programmer's toolbox.

Histograms are useful in various application areas. I have also found the availability of such a class to be extremely useful as a tool to analyze what sort of input, if any, is "typical" for my programs. This knowledge can then be used to tune the algorithms in my programs to improve the run-time of these typical cases (possibly at the expense of less frequent cases).

Here is a simple version (it ignores some problems that can occur when the histogram intervals touch the extreme values of the `ptrdiff_t` type):

```
#include <stddef.h>

struct Histogram {
   Histogram(ptrdiff_t minval, size_t gap, size_t n_bins);
   ~Histogram() { delete[] bin_; }
   void record(ptrdiff_t);
   void output_to(std::ostream&);
private:
   ptrdiff_t const minval_, maxval_;
   size_t const gap_;
   size_t *const bin_;
   size_t n_too_small_, n_too_large_;
};
```

Note the use of `size_t` wherever an unsigned integer is needed, and `ptrdiff_t` for signed integers. The `bin_` member is initialized to point to an array describing the intervals, or bins, counted by the histogram. The size of each bin is determined by a given gap parameter `g`. Because they do not change once initialized, the members `gap_`, `minval_` ,and `maxval_` are declared `const`. Two members keep track of the number of values that fall outside the range of counted values.

```
Histogram::Histogram(ptrdiff_t m, size_t g, size_t n)
   : minval_(m), maxval_(m+n*g-1), gap_(g),
     bin_(new size_t[n]), n_small_(0), n_large_(0) {
   assert(g!=0 && n!=0);
}

void Histogram::record(ptrdiff_t datapoint) {
   if (datapoint<minval_)
      ++n_small_;
```

```
    else if (datapoint>maxval_)
        ++n_large_;
    else
        ++bin_[(datapoint-minval_)/gap_];
}

void Histogram::output_to(std::ostream &output) {
    output << "< " << minval_ << ": " << n_small_ << '\n';
    for (ptrdiff_t left = minval_; left<maxval_; left += gap)
        output << left << ".." << left+gap_-1 << ": "
               << bin_[(left-minval_)/gap_] << '\n';
    output << "> " << maxval_ << ": " << n_large_ << '\n';
}
```

Exercise 10.12

Define a class `Char_queue` so that the public interface does not depend on the representation. Implement `Char_queue` (a) as a linked list and (b) as a vector. Do not worry about concurrency.

Hint: The C++ standard library provides a `queue` template. You might prefer to try the vector case first because it is simpler.

The ideal queue has but three operations: enqueue and dequeue operations and a function to test emptiness of the queue. In the real world, however, one must take into account the possibility of resource exhaustion—in this case, the resource is memory. We will, therefore, throw a `std::overflow` exception when attempting to add an element to a full queue. Because some implementations don't really need adhere to a predetermined maximum size, the interface will also contain a Boolean constant, `fixed_capacity`, which determines whether the capacity specified as a constructor argument is irrevocable. For implementations whose capacity is fixed, a member function `full` tests whether the maximum capacity has been reached.

```
struct Char_queue {
    Char_queue(unsigned capacity = default_capacity);
    ~Char_queue();
    bool empty() const;
    char dequeue();
    enqueue(char);
    bool full() const;
    static bool const fixed_capacity = true; // or = false.
private:
    // illustrative only:
    static unsigned const default_capacity = 20;
    // ...
};
```

The italic typeface for the default argument `default_capacity` is a reminder that this name is, in fact, not part of the `Char_queue` interface, nor is the value associated with this name.

Let's first solve the simpler case using a vector, where we take the term *vector* in its generic meaning rather than referring to the standard template with the same name. Specifically, we use an array of `char` to hold the elements in the queue:

```
#include <stdexcept>

struct Char_queue {
   inline Char_queue(unsigned capacity = default_capacity);
   ~Char_queue() { delete[] queue_; }
   bool empty() const { return head_==tail_; }
   inline char dequeue();
   inline void enqueue(char);
   bool full() const { return head_==(tail_+1)%capacity_; }
   static bool const fixed_capacity = true;
private:
   static unsigned const default_capacity = 32;
   char *queue_;
   unsigned head_, tail_;
   unsigned const capacity_;
};
```

Notice how the two integer cursors `tail_` and `head_` cycle through the array as elements are respectively added and removed from the queue. The remainder-after-division operator is used to achieve the cyclic effect.

```
inline
char Char_queue::Char_queue(unsigned n)
   : queue_(new char[n]), head_(0), tail_(0), capacity_(n) {
}

inline
char Char_queue::dequeue() {
   if (!empty()) {
      char c = queue_[head_];
      head_ = (head_+1)%capacity_;
      return c;
   } else
      throw std::underflow_error(std::string("queue"));
}

inline
void Char_queue::enqueue(char c) {
   if (!full()) {
      queue_[tail_] = c;
      tail_ = (tail_+1)%capacity_;
```

```
    } else
        throw std::overflow_error(std::string("queue"));
}
```

Because all the member functions are inline (for efficiency), there need not be a separate implementation file. Overall, this implementation should be efficient, but it may lack flexibility when an upper bound on the maximum number of elements in the queue is not known. A linked-list-based solution can provide that flexibility. However, it becomes complex enough to justify physically separating the interface from the implementation. Here is the header file:

```
struct Char_queue {
    Char_queue(unsigned capacity = 0): in_(&out_cell)
        { out_cell_.next_ = 0; }
    ~Char_queue();
    bool empty() const { return in_==&out_cell_; }
    char dequeue();
    void enqueue(char);
    bool full() const { return false; }
    static bool const fixed_capacity = false;
private:
    struct Cell {
        Cell *next_;
        char c_;
    };
    Cell out_cell_;
    Cell *in_;
};
```

As with the implementation of a linked list in Exercise 8.1, the code is simplified by adding a guard node, `out_cell_` , which guards the end of the queue on which dequeueing happens. The member `in_` points to the last enqueued cell or to `out_cell_` when the queue is empty. Note that the member function `full` can be misleading. It ignores the possibility of memory exhaustion.

The remainder of the implementation is placed in a separate implementation file:

```
#include <new>

Char_queue::~Char_queue() {
    // Delete all elements in the queue:
    for (Cell *p = out_cell_.next_; p;) {
          Cell *victim = p;
          p = p->next_;
        delete victim;
    }
}
```

```
char Char_queue::dequeue() {
   if (!empty()) {
      Cell *victim = out_cell_.next_;
      char c = victim->c_;
      out_cell_.next_ = victim.next_;
      delete victim;
      return c;
   } else
      throw std::underflow_error(std::string("queue"));
}

void Char_queue::enqueue(char c) {
   // the 'std::nothrow' prevents the bad_alloc exception:
   in_->next_ = new(std::nothrow) Cell;
   if (!in_->next_)
      throw std::overflow_error(std::string("queue"));
   in_ = in_->next_;
   in_->c_ = c;
}
```

Notice how this implementation is extremely inefficient on most platforms in terms of both run-time speed and memory usage. For example, for every character in the queue, a pointer is also allocated. However, pointers often require several times as much storage as `char`s do (typically four or eight times). Furthermore, the `new` operator usually allocates some bookkeeping overhead, as well.

Supplementary Exercise

(A slightly tedious, but realistic performance tuning exercise) Design a more space-efficient version of the above queue. Do this by linking chunks of several characters at a time. At least 75 percent of the storage required for a `Char_queue` containing 100 characters should be used for the storage of those characters themselves.

Supplementary Exercise

(Not trivial) In this list-based implementation, the fact that a call to `full` returns false is not a guarantee that the next enqueue operation will succeed. What would it take to have this guarantee?

Queues are often used as a synchronized communication primitive between processes running concurrently (for example, on a parallel computer). The exercise explicitly asks to ignore the complications arising in this context; access to the queue structures should be appropriately sequentialized otherwise.

Exercise 10.13

Design a symbol table class and a symbol table entry class for some language. Have a look at a compiler for that language to see what the symbol table really looks like.

Hint: Read about associative containers in §17.4. See also Exercises 6.3, 6.12, and 10.19.

Let me just briefly discuss the second part of this exercise. Even for a single programming language, you will find a variety of approaches to the symbol table problem. The fundamental data structure underlying a symbol table is usually a *hash table*, but there are some commercial compilers (also for C++) that use a balanced tree instead. The latter option is sometimes a little slower when looking up a symbol, but it keeps the symbols in order, and the implementation often scales well with the program size. In C++, such a symbol table might be implemented using an associative container (§17.4).

Most programming languages associate with each symbol at least two generic attributes: scope and type. This association need not be stored explicitly in the symbol entry. Often, scope information is kept by keeping one symbol table per scope, and using a stack of symbol tables to handle nested scopes. Some languages are structured in such a way that separate symbol tables can be used for each available type. However, for a language like C++, in which user-defined types are common, this is not generalizable. Still, certain symbol types could be kept separately (for example, this is often the case for `goto`-labels in C++).

Exercise 10.15

Given the program:

```
#include <iostream>
int main() {
   std::cout << "Hello, world!\n";
}
```

modify it to produce this output:

```
Initialize
Hello, world
Clean up
```

Do not change `main()` in any way.

Hint: Understand constructors and destructors (§10.4). Avoid the need to run constructors or destructors outside any function; such situations often lead to brittle software.

The only thing that occurs prior to the execution of the body of `main` is the initialization of global variables. The only thing that occurs after the `main()` program has completed is their destruction. Hence, the solution to this problem consists of adding a global variable whose

constructor and destructor have the appropriate side effects. This can be achieved by appending the following bit to the above C++ program:

```
struct GlobalBracket {
   GlobalBacket() { std::cout << "Initialize\n"; }
   ~GlobalBracket() { std::cout << "Clean up\n"; }
} global_bracket_variable;
```

Note that not all the global variables in your program will necessarily be constructed before the execution of the body of `main()`. That is only guaranteed for global variables defined in the same translation unit (source file) that contains the definition of `main()`. For other global variables, it is only guaranteed that they will be initialized before any of the functions in the same translation unit are called for the first time. This relatively weak guarantee is one of the many reasons for avoiding global variables requiring initialization.

Exercise 10.19

Write a function that, given an `istream` and a `vector<string>`, produces a `map<string, vector<int> >` holding each string and the numbers of the lines on which the string appears. Run the program on a text file with no fewer than 1,000 lines, looking for no fewer than 10 words.

Hint: See Exercises 6.3, 6.12, and 16.2. This is another nice illustration of the synergism in the C++ standard library.

We'll assume that the contents of the vector are, indeed, words—that is, things we would obtain by using expressions of the type '`cin >> some_string`.' Because we need to count lines, we'll read a line at a time using a `std::getline` function and then turn that line into a `stringstream` for further processing.

Again, `map` appears to be an extremely powerful standard library tool.

```
#include <map>
#include <sstream>
#include <string>
#include <vector>

using std::istream;
using std::string;
using std::vector;

typedef std::map<string, vector<int> > LineIndex;
typedef vector<string>::const_iterator EntryIt;

LineIndex* make_line_index(istream &input,
                           vector<string> const &entries) {
   // First create the index and enter the entries in it:
   LineIndex *index = new LineIndex;
```

```
    for (EntryIt p = entries.begin(); p!=entries.end(); ++p)
       (*index)[*p]; // Causes insertion

    // Now, read one line at a time:
    string line, word;
    int line_number = 0;
    while (std::getline(input, line)) {
       ++line_number;
       std::istringstream words(line);

       // Read each word in this line and update the index if needed:
       while (words >> word) {
          LineIndex::iterator p = index->find(word);
          if (p!=index->end()) // Found?
             (*p).second.push_back(line_number);
       }
    }
}
```

Observe again how the standard library allows fairly powerful programs to be written in a concise yet efficient manner. If you feed `make_line_index` a file containing the following three lines:

```
a dog
a cat
the dog
```

it will create a map containing the following (`string, vector<int>`) pairs:

```
("a", {1, 2}), ("cat", {2}), ("dog", {1, 3}), ("the", {3})
```

(in that order).

Here is a `main()` function that can make use of `make_line_index`:

```
#include <algorithm>
#include <iostream>
#include <iterator>

int main(int argc, char *argv[]) {
   // Words to look up: argv[1..argc-1]
   // Enter them in a vector:
   vector<string> entries;
   for (int k = 1; k!=argc; ++k)
      entries.push_back(string(argv[k]));
   // Compute the index using the above function:
   LineIndex *index = make_line_index(cin, entries);
```

```
    // Output the results:
    vector<string>::iterator p = entries->begin();
    for (; p!=entries->end(); ++p) {
       cout << "Word " << *p << " appears in lines ";
       LineIndex::iterator lines = index->find(*p);
       std::copy((*p).second.begin(), (*p).second.end(),
                 ostream_iterator<int>(cout, ", "));
       cout << ".\n";
    }
}
```

The `copy` algorithm is combined with the concept of an *output iterator* (§19.2.6) to provide a compact way of outputting a vector of line numbers. The words to be searched for are given on the command line of this program; the lines of text form the standard input `cin`.

Chapter 11

Operator Overloading

Abstraction captures the essential and relevant elements of the *concrete*. The essential elements in a function are the semantics of the transformation or procedure it realizes. Hence, it comes as no surprise that C++ allows multiple functions acting on different argument types to share the same name. Though the language does not mandate that two functions with the same name perform similar operations, it is clearly the primary use of overloading. Straying far from this principle is rarely wise. This chapter shows some examples of this. The fundamentals of function name overloading are explained in §7.4—the rules for operator overloading and function name overloading are mostly identical.

Exercise 11.1

In the following program, which conversions are used in each expression?

```
struct X {
   int I;
   X(int);
   X operator+(int);
};

struct Y {
   int I;
   Y(X);
   Y operator+(X);
   operator int();
};

extern X operator*(X, Y);
extern int f(X);
```

```
X x = 1;
Y y = x;
int i = 2;

int main() {
   i+10; y+10; y+10*y;
   x+y+i; x*x+i; f(7);
   f(y); y+y; 106+y;
};
```

Modify the program so that it will run, and print the values of each legal expression.

Hint: Chapter 2 and §7.4 introduce the fundamentals of overload resolution. This exercise leads to an important conclusion: Beware of implicit conversions.

Let's first look at the global initializations:

```
X x = 1;
Y y = x;
int i = 2;
```

The initialization of `i` needs no conversion because the literal 2 already has type `int`. However, the first two initializers (the literal 1 and the variable `x`) do not have the type of the object they're initializing. So they will be converted using a one-argument constructor. The following annotates this explicitly in the source:

```
X x = X(1);
Y y = Y(x);
int i = 2;
```

Observe how the necessary constructors are available. Similar annotations show which conversions are required in the expressions in `main()`:

```
int main() {
   i+10; // No conversion needed
   y+10; // Error: ambiguous!
      // y.operator int()+10 or y+X(10)
   y+10*y; // Error: ambiguous again!
      // y+X(10*y.operator int()) or y+X(10)*y
   (x+y.operator int())+i; // Uses X::operator+(int) twice;
                           // result has type X
   x*Y(x)+X(i); // Result has type X
   f(X(7));
   f(y); // Error: no valid implicit conversion from Y to X
   y.operator int()+y.operator int(); // Result has type int
   106+y.operator int(); // Result has type int
}
```

In the expression `i+10`, both arguments of the addition operator have integer type, and, therefore, the built-in operator applies without conversions. For `y+10`, there is a choice between using the built-in addition operator after converting `y` to an `int` using `Y`'s user-defined conversion operator and using `Y`'s addition-member operator following a conversion of the integer 10 to type `X` using `X`'s one-argument constructor. C++ does not prefer either of these choices over the other. Hence, the expression is invalid. The subexpression `10*y` in `y+10*y` is similarly ambiguous because there is no preference between converting the integer 10 to an `X` followed by a call to `operator*(X, Y)` and using the built-in multiplication operator after converting `y` to an integer.

Remember that only a single user-defined conversion—a nonexplicit one-argument constructor or a conversion operator—will be considered to achieve an overload resolution match (§11.4). The call `f(7)` is therefore not a problem and resolves to `f(X(7))` with a single user-defined conversion. On the other hand, the call `f(y)` is invalid. The conversion that would result in `f(X(y.operator int()))` cannot be considered because it involves a sequence of two user-defined conversions. The `operator+` member of `Y` cannot be considered for `y+y` for the same reason, but the built-in addition applies because both arguments can be converted to an `int` each with a single user-defined conversion.

Clearly, implicit conversions can become unwieldy and should be handled with care. Therefore, it pays to annotate single-argument constructors with the keyword `explicit` (§11.7.1) when the constructor is not intended to be considered for user-defined conversion.

Exercise 11.3

Define a class INT that behaves exactly like an `int`. Hint: Define `INT::operator int()`.

Hint: Conversion operators are explained in §11.4.

Consider the following solution:

```
struct INT {
   INT(int i): i_(i) {}
   INT& operator=(int i) { i_ = i; return *this; }
   operator int() { return i; }
private:
   int i_;
};
```

Superficially, this class will behave very much like the `int` type. One of the caveats, however, is that to achieve this we rely on a user-defined conversion operator. You can use, at most, *one* of these operators in any implicit conversion (§11.4.1). In the following scenario, we cannot, therefore just replace `int` by `INT`:

```
struct A {
   operator int();
} a;
```

```
char const *vowels = "aeiou";
char w = vowels[a];
```

Then we could not just replace `int` by `INT` in struct `A` beacuse the first user-defined conversion from struct `A` to struct `INT` would prevent a second one from `INT` to `int` from being considered in the expression '`vowels[a]`.'

Exercise 11.4

Define a class `RINT` that behaves exactly like an `int` except that the only operations allowed are + (unary and binary), - (unary and binary), *, /, and %. Hint: Do not define `RINT::operator int()`.

Hint: The development of a class `complex` (§11.1-3) can be mimicked for the solution of this exercise.

We will not try to emulate the various promotion and conversion rules that apply to objects of type `int`. Here is some sample code:

```
struct RINT {
   RINT(int i): i_(i) {}
   RINT& operator=(int i) { i_ = i; return *this; }
private:
   int i_;
   friend RINT operator+(RINT const&);
   friend RINT operator+(RINT const&, RINT const&);
   friend RINT operator-(RINT const&);
   friend RINT operator-(RINT const&, RINT const&);
   friend RINT operator*(RINT const&, RINT const&);
   friend RINT operator/(RINT const&, RINT const&);
   friend RINT operator%(RINT const&, RINT const&);
};

RINT operator+(RINT const &a) {
   return a;
}

RINT operator+(RINT cosnt &a, RINT const &b) {
   return RINT(a.i_+b.i_);
}

RINT operator-(RINT const &a) {
   return RINT(-a.i_);
}
```

```
RINT operator-(RINT const &a, RINT const &b) {
   return RINT(a.i_-b.i_);
}

RINT operator*(RINT const &a, RINT const &b) {
   return RINT(a.i_*b.i_);
}

RINT operator/(RINT const &a, RINT const &b) {
   return RINT(a.i_/b.i_);
}

RINT operator%(RINT const &a, RINT const &b) {
   return RINT(a.i_%b.i_);
}
```

If you want to support mixed-type operations (for example, the multiplication of an `RINT` with an `int`) you could add additional versions of the operator functions. For example:

```
RINT operator*(int const &a, RINT const &b) {
   return RINT(a*b.i_);
}
```

Clearly, you will need many overloaded versions of each operator if you want to support all combinations with built-in types.

Exercise 11.6

Define a class implementing arbitrary precision arithmetic. Test it by calculating factorial of 1,000.

Hint: You will need to manage storage in a way similar to what was done for class `String`.

The solution of this exercise is beyond the scope of this solution guide. However, it's worth noting that this problem has been tackled many times in both commercial and public domain C++ packages. You might want to try out some of these packages and compare their performance. In particular, it should be interesting to see how performance evolves as you multiply 100, 200, 400, 800, ... bit values. The better implementations will use an algorithm based on the *Fast Fourier Transform (FFT)* whose run-time cost grows like $O(n \log n)$, where n is the number of bits in the value representation.

Exercise 11.7

Define an external iterator for class `String`*:*

```
class String_iter {
   // refer to string and string element
public:
   String_iter(String& s); // iterator for s
   char& next();           // reference to next element
   // more operations of your choice
};
```

Compare this in utility, programming style, and efficiency to having an internal iterator for* `String` *(that is, a notion of a current element for the* `String` *and operations relating to that element).

Hint: The `String` class is developed in §11.12. This discussion touches on the important subject of resource ownership.

There are various approaches to this iterator. In particular, we can leave the user of the class responsible to ensure that a `String_iter` is not used past the lifetime of the `String` to which it refers, or we can handle this automatically. Similarly, because `String` uses an underlying reference-counted representation, we could let the user take care of this, or we could programmatically ensure correct behavior. I choose to leave the handling of lifetimes to the user, but I will use a conservative handling of the reference-counted representation when the iterator is dereferenced:

```
// Assume String_iter is a friend class of String:
struct String_iter {
   String_iter(String &s): s_(s), pos_(0) {}
   char& next() {
      s.rep = s.rep->get_own_copy();
      return s.rep->s[pos_++];
   }
   char next() const { return s.rep->s[pos_++]; }
private:
   String &s_;
   int pos_;
};
```

The use of external iterators is almost always preferable to internal iterators. They can usually be made as efficient as internal iterators, many of them can exist simultaneously for the same container, and they are often more convenient to pass around in your program. The C++ standard library is an example of a C++ library relying on a well-defined hierarchy of external iterators. If you haven't done so yet, you might want to study its detailed specification.

Exercise 11.8

Provide a substring operator for a string class by overloading (). ***What other operations would you like to be able to do on a string?***

Hint: Overloading the function-call operator is described in §11.9. It is also the operator of choice to produce *function objects* (§18.4).

Let's develop this for the `String` class discussed in the text (§11.12). The arguments to the operator indicate the left and right boundary of the substring to be extracted. If the left position exceeds the right position, an empty string is returned.

```
class String {
   // ...
public:
   String operator()(int left, int right) {
      check(left); check(right); // Ensure we're in-range
      if (left<=right) {
         char *a = new char[right-left+2];
         std::strncpy(a, rep->s+left, right-left+1);
         a[right-left+1] = '\0';
         String result(a);
         delete[] a;
         return result;
      } else // return an "empty string"
         return String("");
   }
};
```

Many other string operations are imaginable and can be found in commercial and public domain libraries: finding a substring, splitting the string up according to user-defined separators (for example, to split a sentence into words), finding a certain pattern in the given string, and so forth. Study the standard C++ string (§20) to identify a variety of basic string operations.

Exercise 11.10

Define an operation for `String` that produces a C-string representation of its value. Discuss the pros and cons of having that operation as a conversion operator. Discuss alternatives for allocating the memory for that C-string representation.

Hint: This discussion complements Exercise 11.7.

Because the `String` class already holds such a representation internally, we can simply provide access to it (as opposed to a copy of it):

```
class String {
   // ...
   char const* c_string() const {
      return rep->s;
   }
};
```

The disadvantage of this approach is the rather complicated *lifetime* of the returned C-string; it remains valid until the `String` is destroyed or until a `String` operation calling `Srep::get_own_copy` is applied to the source `String`. The advantage is that the user need not worry about releasing the memory for the returned C-string (because it is still *owned* by the `String` object). Alternatives would be to allocate a new array of `char`s and let the user take care of `delete[]`-ing this array when appropriate. Or more symmetrically, let the user provide an array in which the representation can be stored. (I find it more symmetric because the user handles both allocation and deallocation.)

Personally, I see no advantage at all to providing this functionality as a conversion operator. Some programmers like the ability to transparently use a `String` where a `char const*` is expected. In fact, doing so could lead to subtle overload resolution effects. For example, suppose we had the following `String` interface:

```
class String {
   // ...
   char operator[](int);
   operator char const*();
};
```

With the above, the code

```
String s;
for (long k = 0; s[k]; ++k)
   cout << s[k] << '.';
```

is ambiguous if `std::ptrdiff_t` is a synonym for `long`. Indeed, the type of the subscript index for built-in pointers is `std::ptrdiff_t`. Hence, for the expression `s[k]`, a correct compiler won't be able to choose between `(s.operator char const*())[k]` and `s.operator[](int(k))`. Note that on many platforms, `std::ptrdiff_t` is a synonym for `int`. On many others, it is a synonym for `long`. Code such as appears above is, therefore, not portable. Because of such effects, I generally advise against the use of conversion operators unless they positively contribute to the overall simplification of your software.

The standard `string` class has member functions `c_str` and `data` for this purpose.

Exercise 11.20

Given two structures:

```
struct S { int x, y; };
struct T { char *p; char *q; };
```

write a class C that allows use of x and p from some S and T much as if they had been members of C.

Hint: Remember that references act as synonyms.

This can be done simply using references (§5.5):

```
struct C {
   C(S &s, T &t): a(s.a), p(t.p) {}
   int &a;
   char *&q;
};
```

As with many of the answers in this chapter, just because you *can* do such things, it is not necessarily a good idea to do so.

Exercise 11.21

Define a class Index to hold the index for an exponentiation function mypow(double, Index). Find a way to have 2**I call mypow(2, I).

Hint: Remember that the asterisk ('*') is both a unary and a binary operator. Do not do this in real code.

You already know that there is no token ** in C++ (§11.2). However, the asterisk has two meanings in C++: the unary dereferencing operator and the binary multiplication operator. The former takes precedence over the latter so that

```
r = 2**I;
```

really means

```
r = 2*(*I);
```

Hence, we could overload the unary dereferencing operator to have no effect but to return its own target, and we could overload the multiplication operator to call mypow:

```
struct Index {
   Index(double p): p_(p) {}
   Index const& operator*() const { return *this; }
private:
   double p_;
};

double operator*(double a, Index const &b) {
   return mypow(a, b);
}
```

I should caution against performing this sort of trick in real code, however. One of its deficiencies is that it will behave counter intuitively in slightly more complex situations. For example,

```
r = 3*2**I;
```

is equivalent to

```
r = 6**I;
```

and not to

```
t = 3*(2**I).
```

The latter is the usual behavior of exponentiation operators.

Chapter 12

Derived Classes

Classes, overloading, and templates are compile-time mechanisms to improve program abstraction. Often, deciding which actions are appropriate for a certain data item cannot be decided until run-time. This is sometimes referred to as *run-time polymorphism*, or simply *polymorphism*. Class derivation and virtual functions are the basic tools to achieve polymorphic effects. Derivation can also be used simply as an implementation convenience—*private* inheritance is often the natural choice there. However, this chapter focuses on polymorphism and culminates in a discussion of the difficult problem of multiple dispatch in Exercise 12.10.

Exercise 12.1

Define

```
class base {
public:
   virtual void iam() { cout << "base\n"; }
};
```

Derive two classes from `base`. For each, define `iam()` to write out the name of the class. Create objects of these classes and call `iam()` for them. Assign pointers to objects of the derived classes to `base*` pointers and call `iam()` through those pointers.

Hint: A fundamental exercise illustrating the basic principle of virtual functions (§12.2.6). The second half of our discussion presents the `typeid` operator (§15.4.4).

Following the indicated steps, we obtain the following code:

```
#include <iostream>
using namespace std;

struct Northpole: public base {
   virtual void iam() { cout << "Northpole.\n"; }
};
```

```
struct Southpole: public base {
   virtual void iam() { cout << "Southpole.\n"; }
}

int main() {
   Northpole NP;
   Southpole SP;
   NP.iam();
   SP.iam();
   base *pole1= &NP, *pole2 = &SP;
   pole1->iam(); // or (*pole1).iam()
   (*pole2).iam(); // or pole2->iam()
   return 0;
}
```

which outputs

```
Northpole.
Southpole.
Northpole.
Southpole.
```

It is sometimes useful to have a textual representation of a type at run-time. C++, therefore, provides a built-in facility for this purpose: the `typeid` operator (§15.4.4), which returns a reference to a `type_info` structure. The program above might be rewritten as follows:

```
#include <iostream>
#include <typeinfo>
struct base {
   void iam() { std::cout << typeid(*this).name() << '\n'; }
private:
   virtual void polymorpher() const {}
};

struct Northpole: public base {};
struct Southpole: public base {};

int main() {
   Northpole NP;
   Southpole SP;
   NP.iam();
   SP.iam();
   base *pole1= &NP, *pole2 = &SP;
   pole1->iam(); // or (*pole1).iam()
   (*pole2).iam(); // or pole2->iam()
   return 0;
}
```

This solution assumes that the `type_info::name()` member function returns something reasonable. This need not be the case in theory, but in practice you should not encounter major surprises.

This solution no longer requires the `iam()` member to be virtual, but try removing the `polymorpher()` member and see how it affects the output of this program. Remember that the `typeid` operator will return a reference to the `type_info` object for the dynamic type of its argument only if you pass it a polymorphic object (that is, an object of a class type with at least one virtual function). Otherwise, the `type_info` object describes only the static type of its argument. Hence, if you leave out the `polymorpher()` member, `base` is nonpolymorphic and you always obtain the same output even though the complete object may have a type derived from `base`.

One last note: `typeid` can be very handy for analyzing and debugging code under development, as well as for the implementation of certain cross-program interfaces. However, you should be suspicious of code that uses this operator as a workaround for poorly designed class hierarchies. Virtual functions and templates are almost always preferable mechanisms to implement type-dependent behavior.

Exercise 12.9

Consider

```
class Char_vec {
   int sz;
   char element[1];
public:
   static Char_vec* new_char_vec(int s);
   char& operator[](int i) { return element[i]; }
   // ...
};
```

Define `new_char_vec()` to allocate contiguous memory for a `Char_vec` object so that the elements can be indexed through `element`, as shown. Under what circumstances does this trick cause serious problems?

Hint: Although this trick is nonportable and relatively dangerous, it is commonly found in existing programs.

Here is a definition that will work on many systems:

```
#include <new>

Char_vec* Char_vec::new_char_vec(int s) {
   char *rawbytes = ::operator new(sizeof(Char_vec)+s-1);
```

```
    Char_vec *vec = new(rawbytes) Char_vec;
    vec->sz = s;
    return vec;
};
```

Observe how we first allocate raw memory by calling the global operator `new` and then construct an object of type `Char_vec` at that location by using the *placement-new* (§10.4.11) syntax '`new(rawbytes) Char_vec`.' Note that one should not use the expression "`new char[sizeof(Char_vec)+s-1]`" to allocate raw memory because that memory may not satisfy the alignment requirements of a `Char_vec`, while an explicit call of the global operator `new` is guaranteed to yield storage that is sufficiently aligned for any C++ object.

Because most implementations do not check array bounds, you can often get away with this (then again, you may not). However, some code-analysis tools may warn you about this doubtful construct. Although unlikely, you may also find that you cannot successfully deallocate such a `Char_vec` object (because the deallocation directive '`delete Char_vec_ptr`' does not match the allocation directive). This technique will also fail for `Char_vec` objects not allocated on the heap, or possibly, if `Char_vec` has virtual member functions. Finally, this makes it hard to embed a `Char_vec` as a member or base class of another class.

This small example illustrates that memory allocation and object construction in C++ (and C) can be handled in exotic ways. Fortunately, this is rarely needed. Nevertheless, it is often a strength to be able to deviate from safe, well-defined practices and support platform-dependent, yet practical, solutions.

Exercise 12.10

Given classes `Circle`, `Square`, and `Triangle` derived from a class `Shape`, define a function `intersect()` that takes two `Shape*` arguments and calls suitable functions to determine if the two shapes overlap. It will be necessary to add suitable (virtual) functions to the classes to achieve this. Don't bother to write the code that checks for overlap; just make sure that the right functions are called. This is commonly referred to as double dispatch or a multi-method.

Hint: A difficult problem. It can be skipped on a first reading.

This exercise may look deceptively simple, but in the world of compiled languages it is a notoriously difficult problem—*multiple dispatch*. Basically, we would like the function intersect to be "virtual with respect to multiple classes." Unfortunately, such a concept does not exist in C++. (It does exist in languages that were not intended to map so closely to hardware implementations.)

The problem wouldn't be so hard if we didn't care about maintainability:

```
bool intersect(Shape *a, Shape *b) {
    if (Circle *c = dynamic_cast<Circle*>(a))
        return b->intersects(c);
```

```
    else
    if (Square *s = dynamic_cast<Square*>(a))
       return b->intersects(s);
    else
    if (Triangle *t = dynamic_cast<Triangle*>(a))
       return b->intersect(t);
    else
       throw "Unexpected Shape* for intersect().";
}
```

This would go hand in hand with a Shape class that includes the following interface elements:

```
struct Shape {
   virtual bool intersects(Circle*) = 0;
   virtual bool intersects(Square*) = 0;
   virtual bool intersects(Triangle*) = 0;
   // ...
};
```

If we ever add another kind of Shape, we must:

1. Add a test in the `intersect` function.
2. Add a pure virtual function in the abstract base Shape.
3. Add overriders for this pure virtual in every existing Shape derivation.
4. Add implementations of all the intersects members in the new Shape derivation.

Not exactly a scaleable design, is it?

Before exploring another fairly general alternative, there is an option that is often preferable to finding a multiple dispatch mechanism: a type-independent algorithm. For example, one geometric database has a small, but general, set of primitives to describe the boundaries of a shape. Each of these boundary primitives knows only about the concepts of "point" and "straight infinite line" and can answer simple queries about these. For example, given a point, the boundary can determine whether the point was on the boundary. A general intersection algorithm is then implemented using this relatively small, fixed set of primitives—no multiple dispatch is required. At first, this design looks expensive in terms of computational needs; many of the generic boundary query functions would have been much faster if they had been specialized for each kind of boundary element. However, because the design is so extensible, the project can focus on the client code—which is really the crux of the matter—rather than continuously maintain the geometric database primitives. In paying more attention to the client code, the project team also found ways to reduce the number of necessary queries, thereby achieving their performance goals.

Now, if you really need multiple dispatch, you can emulate a sort of multi-virtual mechanism using the `typeid` operator (§15.4.4), the standard library `map` facility (§17.4.1), and some home-brewed templates as follows:

```
#include <typeinfo>
#include <utility> // for pair<> and make_pair

// We will map pairs of type_info pointers to
// pointers to intersection functions:
using std::type_info;
typedef std::pair<type_info const*, type_info const*>
        TwoTypes;
typedef bool (*Intersector)(Shape const*, Shape const*);

// The map needs an ordering relation:
struct TwoTypesOrder {
   bool operator(TwoTypes const &a, TwoTypes const &b) {
      if ((a.first)->before(*b.first))
         return true;
      else
      if ((*a.first)==(*b.first))
         return (a.second)->before(*b.second);
      else
         return false;
   }
};
```

The standard library sometimes requires an ordering function. This could have been a pointer to a function. But, instead, the library usually expects an object that implements the function call operator: a function object (§18.4). Objects of class `TwoTypesOrder` satisfy this requirement and can be used to order two objects of type `TwoTypes`. For this design, that element will be the associative `map` container. For our purposes, it is not actually important what the ordering relation is; `TwoTypesOrder` uses the `type_info::before` ordering relation between `type_info` objects to achieve the desired effect.

Here is the map type:

```
#include <map>
typedef std::map<TwoTypes, Intersector, TwoTypesOrder>
        DispatchMap;
```

To avoid dependencies on the order of initialization, it is best not to create a global object to keep the dispatch information. Instead, use a function with a local static object:

```
DispatchMap& intersector_map() {
   static DispatchMap dispatch_table;
   return dispatch_table;
}
```

The function `intersector_map` illustrates a very useful C++ technique. If we were to define a global `DispatchMap` object, we might end up in situations in which we would use the map before it was actually constructed—an error whose origin would be hard to trace. Local static objects, on the other hand, are constructed on the first invocation of the function. A function returning a reference to a local static object is therefore often a better substitute for a global variable.

We still need a function to register intersection functions:

```
template<class Type1, class Type2>
void register_intersector(Intersector pf) {
   intersector_map().insert(
      make_pair(&typeid(Type1), &typeid(Type2)), pf);
}
```

Now the `intersect` function comes easily:

```
bool intersect(Shape *a, Shape *b) {
   DispatchMap::iterator p = intersector_map()
       .find(make_pair(&typeid(*a), &typeid(*b)));
   return (*p)(a, b);
}
```

This is a reasonably scalable approach; as new `Shapes` are added, you can write new intersection functions and add them to the map. The intersect function will automatically know about them. For example:

```
bool squares_intersect(Shape *a, Shape *b) {
   assert(a and b);
   Square *sq1 = dynamic_cast<Square*>(*a);
   Square *sq2 = dynamic_cast<Square*>(*b);
   // ... code to determine if *sq1 and *sq2 intersect
}

void register_all_intersectors() {
   register_intersector<Square, Square>(squares_intersect);
   // ...
}

int main() {
   register_all_intersectors();
   // ...
}
```

Still, as you add new `Shape` types, you may find that the number and complexity of intersection functions explode. This phenomenon would not occur (or would occur less so) if a less type-dependent approach to determine intersections had been chosen (see above).

[blank page]

Chapter 13

Templates

C++ templates brought about a revolution in computer programming. This revolution enabled the C++ standard template library to be hailed as a masterpiece of reusable design. Writing a solid, useful template is harder than writing nongeneric code—the unknown context in which the template will be instantiated must be anticipated. The first two exercises in this chapter illustrate this vividly.

Exercise 13.1

Fix the errors in the definition of `List` from §13.2.5 and write out C++ code equivalent to what the compiler must generate for the definition of `List` and the function `f()`. Run a small test case using your hand-generated code and the code generated by the compiler from the template version. If possible on your system, given your knowledge, compare the generated code.

Hint: This is a good exercise to sharpen your understanding of the template instantiation process. That, in turn, is useful for writing robust templates.

Here is the corrected `List` class template:

```
template<typename T>
class List {
   struct Link {
      Link(Link *p, Link *s, T const &v)
         : pre_(p), suc_(s), val_(v) {}
      Link *pre_, *suc_;
      T val_;
   };
```

```
   Link *head_;
public:
   List(): head_(0) {}
   List(T const &t): head_(new Link(0, 0, t)) {}
   // ...
   void print_all() {
      for(Link *p = head_; p; p = p->suc_)
         cout << p->val_ << '\n';
   }
};
```

So what must happen when the compiler encounters the function `f()`?

```
void f(List<int> &li, List<Rec> &lr) {
   li.print_all();
   lr.print_all();
}
```

The answer is that it must *instantiate* the class definition, as well as the member functions and static data members that are used by `f()`. The compiler might thus internally generate code equivalent to the following. (Don't worry about the substituted names. All that matters is that they somehow not cause conflicts with other names in the program.)

```
class __T1__i__List {
   struct Link {
      Link(Link *p, Link *s, int const &v)
         : pre_(p), suc_(s), val_(v) {}
      Link *pre_, *suc_;
      int val_;
   };
   Link *head_;
public:
   __T1__i__List();
   __T1__i__List(int const &t);
   inline void print_all();
};

inline void __T1__i__List::print_all() {
   for(Link *p = head_; p; p = p->suc_)
      cout << p->val_ << '\n';
}
```

```
class __T1__T3Rec__List {
   struct Link {
      Link(Link *p, Link *s, Rec const &v)
            : pre_(p), suc_(s), val_(v) {}
      Link *pre_, *suc_;
      Rec val_;
   };
   Link *head_;
public:
   inline __T1__T3Rec__List();
   inline __T1__T3Rec__List(Rec const &t);
   inline void print_all();
};

inline void __T1__T3Rec__List::print_all() {
   for(Link *p = head_; p; p = p->suc_)
      cout << p->val_ << '\n';
}
```

Note that the latter function will not be processed successfully if no `operator<<` is provided to output objects of type `Rec`. In that case, the following will also fail to be instantiated:

```
void f(__T1__i__List &li, __T1__T3Rec__List &lr) {
   li.print_all();
   lr.print_all();
}
```

Inserting the following declaration ahead of the definition of `f` would ensure successful instantiation of the above functions:

```
ostream& operator<<(ostream &output, Rec const&);
```

A definition for this function might look like this:

```
ostream& operator<<(ostream &output, Rec const&) {
   output << "<Rec object here>";
   return output;
}
```

Supplementary Exercise

(Subtle; safely skipped on a first reading) I failed to mention a rather subtle point in the above: I inserted the code resulting from the template instantiations before the function `f` that uses them. In this particular case, this procedure would not affect the meaning of the program. Yet the language specifications explicitly state that the "point of instantiation"

for a function **declaration** *occurs before the first point of use, but that of its* **definition** *occurs after the first point of use. Can you see why this may preferable? Hint: Play the substitution game on the following little program:*

```
#include <iostream>

template<class T>
T f(T k) (
   if (k%17) return k;
   else return g(k+13);
}

template<class T>
T g(T k) {
   if (k%13) return k;
   else return f(k+17);
}

int main() {
   std::cout << f(2) << '\n';
}
```

Exercise 13.2

Write a singly linked list class template that accepts elements of any type derived from a class `Link` that holds the information necessary to link elements. This is called an* intrusive *list. Using this list, write a singly linked list that accepts elements of any type (a nonintrusive list). Compare the performance of the two list classes and discuss the trade-offs between them.

Hint: This moderately complex exercise is a beautiful demonstration of the sort of complications that arise when creating templates.

The template facility may not be necessary to implement an intrusive list. For example,

```
struct Link {
   Link(Link *next): next_(next) {}
private:
   friend class LinkList;
   friend class Cursor;
   Link *next_;
};
```

```
struct Cursor {
   // Pre- and post-increment:
   Cursor& operator++() { p_ = p_->next_; return *this; }
   Cursor operator++(int)
      { Cursor old(*this); p_ = p_->next_; return old; }
private:
   friend class LinkList;
   friend bool operator==(Cursor const&, Cursor const&);
   friend bool operator!=(Cursor const&, Cursor const&);
   Cursor(Link *p): p_(p) {}
   Link *p_;
};
```

Remember that there are two distinct `operator++` functions. The post-increment version takes an additional "dummy `int` argument." A mnemonic that is sometimes used to remember this is that the *post*-increment has an argument *after* the `++` token. Back to the code:

```
bool operator==(Cursor const &a, Cursor const &b) {
   return a.p_==b.p_;
}

bool operator!=(Cursor const &a, Cursor const &b) {
   return a.p_!=b.p_;
}

struct LinkList {
   LinkList(): head_(&head_) {}
   ~LinkList() { assert(is_empty()); }

   // Element access functions:
   Link& operator[](Cursor const &p)
      { return *p.p_->next_; }
   Link const& operator[](Cursor const &p) const
      { return *p.p_->next_; }

   // Element insertion/extraction:
   void insert(Cursor p, Link *link) {
      assert(p.p_->next_ and link and (link->next_==0));
      link->next_ = p.p_->next_;
      p.p_->next_ = link;
   }
```

```
    Link* extract(Cursor p) {
        Link *link = p.p_->next_;
        assert(link and link!=head_.next_);
        p.p_->next_ = link->next_;
        link->next_ = 0;
        return link;
    }

    // Iteration functions:
    Cursor first() const { return Cursor(&head_); }
    bool is_last(Cursor const &p) const
        { return p.p_==&head_; }
    bool is_empty() const { return head_.next_==&head_; }
private:
    Link head_;
};
```

First, notice this code's simplicity. There is not a single if-statement here, and there are no special cases. This may seem to be of little importance, but on modern computer architectures maximum performance is often achieved by keeping a straight execution path. In order to achieve this simplicity, I opted to have the `Cursor` physically point one position ahead of the element that it logically designates. In addition, the list contains a fixed dummy `Link` element. This makes sure that the list is never physically empty and, hence, there is no need to introduce a special case for the insertion of the first element. Compare this to the approach taken in Exercise 8.1. This approach implies that every element access will cost an additional indirection, which in turn may degrade performance. Clearly, even a basic implementation of a basic data structure can involve a variety of trade-offs when seeking maximum performance. The trade-offs, with respect to usability, will often force the library writer to provide multiple implementations of a facility: a flexible one and a fast one.

Another design decision is to implement iterators using the "subscript idiom" (as opposed to the "pointer idiom" that is used, for example, in the standard library). The iterator type `Cursor` is responsible only for traversing the `LinkList` data structure, whereas actual element access requires the cooperation of the data structure itself. For example, access is achieved as follows:

```
void increment_first(LinkedList &list) {
    ++list[list.first()];
}
```

In contrast, the pointer idiom would allow the slightly more compact

```
void increment_first(LinkedList &list) {
    ++(*list.first()); // The iterator itself knows how to
}                      // access the element.
```

A consequence of the pointer idiom is that you need two distinct iterator classes to handle `const` and non-`const` `LinkedLists` (verify that this is the case in the C++ standard library).

There are some disadvantages to this `LinkList` design. Most noticeable is that there is no efficient and concise way to obtain an iterator designating the last element of the list. Consequently, it is also inconvenient to append an element to the end of the list (unless you manually maintain an end iterator). This is also reflected by the fact that there is no member function that is the exact opposite of `first()`. Instead, a function `is_last()` is provided to test that an iterator has reached the end of the list.

Supplementary Exercise

(A little tedious, but very interesting experiment) Implement a singly linked list that has an amortized constant-time way of appending an element to the end of the list. Compare the performance of the two list implementations for a mixture of random insertions and extractions.

Another aspect of the implementation above is that memory management is left entirely up to the user. To keep things simple in this presentation, we require the user of the list to manually remove all the elements of the list prior to its destruction. Memory management is often among the most delicate decisions for a library writer. Your users may have special allocation policies that could clash with a naive list design. Not providing any memory management policy at all is rarely adequate, but it is often better than imposing an inconvenient one. You will find that the standard library does provide automatic memory management, but it does so using a sophisticated approach that allows its users to customize the allocation and deallocation performed by the standard data structures.

We're now ready to make a first attempt at a nonintrusive singly linked list template based on the above intrusive version:

```
template<typename T>
struct TLink: Link {
   T& value() { return value_; }
   T const& value() const { return value_; }
private:
   T value_;
};

template<typename T>
struct TLinkList: private LinkList {
   TLinkList(): head_(&head_) {}
   ~TLinkList() { assert(is_empty()); }

   // Element access functions:
   T& operator[](Cursor const &p) {
      return *static_cast<TLink<T>*>(p.p_)->value();
   }
```

```
    T const& operator[](Cursor const &p) const {
        return *static_cast<TLink<T>*>(p.p_)->value();
    }

    // Element insertion/extraction:
    void insert(Cursor p, TLink *link)
        { LinkList::insert(p, link); }
    TLink* extract(Cursor p)
        { return static_cast<TLink*>(LinkList::extract(p)); }

    // Iteration functions:
    Cursor first() const { return Cursor(&head_); }
    bool is_last(Cursor const &p) const
        { return p.p_==&head_; }
    bool is_empty() const { return head_.next_==&head_; }
private:
    Link head_;
};
```

As you might have noticed, the memory allocation problem has become more severe. If the type for which you wish to instantiate this `TLinkList` does not have a default constructor, it is not so simple to create an appropriate `TLink` element. Can it be done at all? Perhaps not perfectly portably, but let's study an approach used in various commercial library implementations.

C++ provides a special operator, `new`, which allows us to place an object at a location of our choice. Here is an example of its use:

```
#include <new>

void f(void *p) {
    new(p) double(5.0);
}
```

In this code fragment, the placement-`new` (§10.4.11) syntax will construct a value of type `double` initialized with the value 5.0 at the location pointed to by `p`. On many computers, there are some limitations on the value that `p` can be given. Often, it must correspond to a memory address that is a multiple of the size of a `double`. Similar alignment requirements may exist for other types. If you violate such an alignment requirement, the program will usually abort at run-time in some fashion. (If your system has ever issued a "bus error" message at run-time, you may have encountered such a situation already.) Because of these alignment requirements, the following idea will often not work:

```
template<typename T>
struct TLink: Link {
    T& value() { return *(T*)(raw_mem_); }
    T const& value() const { return *(T const*)(raw_mem_); }
    unsigned char raw_mem_[sizeof(T)]; // Unaligned?
};
```

```
template<typename T>
TLink<T>* create_TLink() {
   TLink<T> *p = new TLink<T>; // OK
   new(p->raw_mem_) T; // May fail if strong alignment
}                      // requirements exist!
```

Fortunately, there is a technique to enforce the alignment of the member `char`-array:

```
template<typename T>
struct TLink: Link {
   union {
      unsigned char raw_mem_[sizeof(T)];
      aligner<T> dummy_;
   };
};
```

This uses an *anonymous union*. Such a union tells the implementation that its members will only exist one at a time and, therefore, space must be allocated only to accommodate the largest of them and/or the one with the most stringent alignment requirements. So we must ensure that the type `aligner<T>` requires alignment that is at least as strong as that required by the type T. There is no guaranteed solution for this within the specifications of the C++ language, but in practice, the following will often work:

```
template<typename T>
struct aligner {
   union {
      long double long_double_;
      long long_;
      void (*simple_function_ptr_)();
   };
};
```

We assume no type will require alignment stronger than the three types contained in this `aligner` template.

With all this machinery in place, we can write a little program that exploits the `TLinkList` template. This program assumes that all the elements developed above were placed in a header file `LinkList.h`.

```
#include "LinkList.h"
#include <iostream>

struct X {
   X(int i): i_(i) {} // No default constructor!
   void report() const { cout << i; }
private:
   int const i_;
};
```

```
TLink<X>* create_XLink(int i) {
   TLink<X> *p = new TLink<X>;
   new(p) X(i);
   return p;
}

void destroy_XLink(TLink<X> *p) {
   delete p;
}

template<typename T>
void reverse(TLinkList<T> &list) {
   for (Cursor p = list.first(); !list.is_last(++p);)
      list.insert(list.first(), list.extract(p));
};

void report(TLinkList<X> const &list) {
   Cursor p = list.first();
   if (!list.is_last(p))
      list[p].report();
   for (; !list.is_last(++p);) {
      cout << ", ";
      list[p].report();
   }
   cout << '\n';
}

int main() {
   TLinkList<X> list;
   Cursor end = list.first();
   int i;
   while ((cin >> i) and (i!=0)) {
      p = create_XLink(i);
      list.insert(end, p);
      ++end;
   };
   report(list);
   reverse(list);
   report(list);
   for (Cursor p = list.first(); not list.is_last(p);)
      destroy_XLink(list.extract(p));
}
```

This little program illustrates various techniques that are applicable to our template. It also illustrates some of its shortcomings. But many of these can be addressed by adding new function templates to the `LinkList.h` header file, such as the `reverse` template in the example above.

Supplementary Exercise

(Slightly subtle, but important) What happens when exceptions are thrown in the `create_XLink` function? Modify class `X` to exhibit this phenomenon. Then adapt the code for `create_XLink` to address the problem.

Supplementary Exercise

(Rather difficult) Develop a memory-management strategy that can conveniently be used with the `TLinkList` template developed above. You may wish to use the facilities provided in the standard library as a source of inspiration. However, you might also find value in keeping your strategy simple. Use this strategy to simplify the destruction of a `TLinkList`.

Writing templates is often more difficult than writing nontemplate code. The discussion around this exercise illustrates this clearly with respect to memory management and object construction issues. Exception handling is also often a difficult topic when designing templates, as is the maintenance of efficiency comparable to that which would be obtained with hand-specialized code. Fortunately, the standard library comes with a collection of useful templates that will allow you to focus on the more unique challenges of your software needs.

Exercise 13.15

Construct an example that demonstrates at least three differences between a function template and a macro (not counting the differences in definition syntax).

Templates are sometimes described as fancy macros, and indeed they can be seen in such a light to a modest extent. They both correspond to compile-time mechanisms that allow patterns to be expanded into actual code. However, the preprocessor facilities stand somewhat apart from the rest of the language. Ordinary lookup, linkage, and other fundamental language rules do not apply to them. Templates, on the other hand, are tightly and naturally integrated into the fundamental texture of C++. Function templates have, among other things, the following characteristics that macros do not have:

- Their instantiations have linkage (§9.2).
- They obey scoping rules (for example, a template may be visible only inside a namespace) (§4.9.4).

- There can be pointers to function-template specializations (§7.7).
- They can be overloaded or specialized (§13.3.2, §13.5).
- They can be recursive.

A short program can vividly illustrate some of these points.

```
#include <iostream>

namespace Template {
   template<typename T>
   void count_down(T const &count) {
      cout << count << '\n';
      if (count)
         count_down(count-1);
   }
   template<>
   void count_down(double const &count) {
      count_down<int>(count);
   }
}

int main() {
   Templates::count_down(3.4);
   return 0;
}
```

Nothing similar could be done with macros.

Exercise 13.16

Devise a scheme that ensures that the compiler tests general constraints on the template arguments for every template for which the object is constructed. It is not sufficient just to test constraints of the form "the argument T must be a class derived from `My_base`."

Hint: A sophisticated but interesting exercise. It can be skipped on a first reading.

In Exercise 13.1, we encountered a situation in which an instantiation would fail if a certain operator were not available. Often, the instantiation of one template will trigger that of another template, which in turn will trigger yet another one, and so forth. When one instantiation fails deep down this chain, the resulting error messages are often obscure, as you may have witnessed while crafting programs based on the standard C++ library templates.

Because of this phenomenon, it is desirable to devise constructs that cause instantiation failures to be diagnosed as early as possible. It has often been said that C++ ought to support a special-purpose mechanism to describe constraints for template parameters. However, it is

not so easy to come up with such a mechanism that is complete, compact, and simple. Fortunately, many kinds of constraints can be expressed using the language as it exists today.

Consider a simple example: We want to express that two types are not equal. One context in which two types cannot be equal is multiple inheritance. For example:

```
template<class X, class Y>
struct NonEqualTypes: X, Y {
};
```

This would fail to instantiate if the parameters X and Y were substituted by equal class types. However, it would also fail to instantiate if either X or Y were a nonclass type, or if, for example, X or Y did not have an accessible default constructor. Fortunately, there is a simple workaround to eliminate the latter failures:

```
template<typename T>
struct TypeHolder {
};

template<typename T, typename U>
struct NonEqualTypes: TypeHolder<T>, TypeHolder<U> {
};
```

Do we often need to establish that a template is not instantiated for a particular type? Probably not, but this little constraint tool will help us develop the next one—ensure that a type for which a template is instantiated inherits from a given type:

```
#define INHERITS_CLASSTYPE(D, B) {      \
   D *derived = 0;                      \
   B *base = derived;                   \
   NonEqualTypes<D*, void*> dummy;      \
}
```

Note that the use of macros is not essential for this technique. However, it allows us to hide odd-looking code behind a self-explanatory facade.

This constraint macro might be used as follows:

```
template<class X>
void destroy_link(X *p) {
   INHERITS_CLASSTYPE(X, Link) // (1)
   delete p;
}
```

If type X does not inherit from class `Link` (see Exercise 13.2 for an example situation in which this may be needed), then the local scope introduced by this macro will cause instantiation failures. The error messages that your compiler should produce for this situation will usually refer to the line marked (1) so that you can easily determine which template argument constraint was violated. By introducing the local scope—that is, enclosing the body of the macro definition with braces—we avoid causing conflicts with the unpredictable surrounding code. This is a rather general technique.

Let's add another example: We'll test the requirement that the template type have an accessible copy constructor. Here is a first idea:

```
#define ACCESSIBLE_COPYCTOR(T) {      \
   T x, y(x);                         \
}
```

The problem with this constraint test is that we enforce the additional constraint that the default constructor be accessible. Often this is undesirable. A general way of dealing with this is to copy reference parameters of a function. C++ does not allow local functions, but it allows member functions of a local class. So I can refine the macro above as follows:

```
#define ACCESSIBLE_COPYCTOR(T) {         \
   struct Dummy {                        \
      void f(T const &x) { T y(x); }     \
   };                                    \
}
```

This constraint macro is often useful, but it requires that the destructor be accessible as well, because code may be generated to destroy the object `y` when it goes out of scope. Sometimes, you may prefer that this is not the case. Dynamically allocating a copy of a `T` object addresses the shortcoming; dynamic objects do not "go out of scope" and, hence, the compiler need not call the destructor in this context.

```
#include <new>
#define ACCESIBLE_COPYCTOR(T) {          \
   struct Dummy {                        \
      void f(T const &x) { new T(x); }   \
   };                                    \
}
```

Constraints for other constructors can be handled similarly.

Supplementary Exercise

(Moderately difficult) Develop a constraint macro that will fail if a given type does not have an accessible destructor. It should not fail for any other reason.

Chapter 14

Exceptions

The fundamentals of exceptions were examined in Chapter 8. However, as with many language features, C++ exceptions are most effective when used in certain idioms. The exercises in this chapter explore techniques relating to both templates and exceptions.

Exercise 14.1

Generalize the STC class (§14.6.3.1) to a template that can use the "resource acquisition is initialization" technique to store and reset functions of a variety of types.

Hint: Restrict yourself to a small, fixed number of arguments. The solution presented uses relatively sophisticated template features. Be sure to understand the use of the keyword `typename` explained in the latter part of the discussion.

The variety-of-types requirement suggests the use of a template. Because templates cannot have variable argument lists, we'll set a fixed limit to the number of parameters a function can take—say, two, for the sake of brevity:

```
template<class R = void, class T1 = void, class T2 = void>
struct STC {
   typedef R (*PF)(T1, T2);
   STC(PF pf): prev_pf_(cur_pf_) { cur_pf_ = pf; }
   ~STC() { cur_pf_ = prev_pf_; }
   R operator()(T1 a, T2 b) { return cur_pf_(a, b); }
private:
   PF prev_pf_;
   static PF cur_pf_;
};

template<class R, class T1, class T2>
typename STC<R, T1, T2>::PF STC<R, T1, T2>::cur_pf_ = 0;
```

The STC template thus has three parameters: one that selects the return type of the functions to be stored and reset and the two others for the parameter types. On construction of one of these objects, we store the currently active function pointer away in the nonstatic data member. It will be restored when the object is destroyed (invocation of the destructor). The argument to the constructor then determines the newly active function. This function can be called by applying the overloaded function-call operator (§11.9) on an STC object. We will come back to the definition of the static member `cur_pf_` and the use of the keyword `typename` at the end of this sample solution.

The STC template arguments default to type `void`. For the return type, this should not be surprising; functions returning `void` are not uncommon. However, `void` function parameters are not allowed in C++. Instead, `void T1` and `T2` template arguments indicate the absence of the corresponding function parameters. As we learned previously, this will not work and the template will fail to instantiate when either of the types `T1` or `T2` is `void`. To resolve this, we can provide *partial template specializations* (§13.5), such as the following:

```
template<class R, class T1>
struct STC<R, T1, void> { // Partial specialization
   typedef R (*PF)(T1);
   STC(PF pf): prev_pf_(cur_pf_) { cur_pf_ = pf; }
   ~STC() { cur_pf_ = prev_pf_; }
   R operator()(T1 a) { return cur_pf_(a); }
private:
   PF prev_pf_;
   static PF cur_pf_;
};

template<typename R>
struct STC<R, void, void> { // Partial specialization
   typedef R (*PF)();
   STC(PF pf): prev_pf_(cur_pf_) { cur_pf_ = pf; }
   ~STC() { cur_pf_ = prev_pf_; }
   R operator()() { return cur_pf_(); }
private:
   PF prev_pf_;
   static PF cur_pf_;
};
```

When you instantiate STC template with either a `T2` or both `T1` and `T2` substituted by `void`, the C++ compiler will select the first or second specialization, respectively (because they are more specialized (§13.5.1) than the primary, more generic, template). If only `T1` is `void` but not `T2`, then none of the specializations will match and the compiler will attempt instantiation of the primary template. This instantiation will fail because the typedef of `PF` with a `void` parameter is illegal.

You might think that we'd also need to specialize the template for various combinations for which `R` is `void`. Fortunately, this is not the case; one is allowed to "return a `void` value." This possibility was introduced into the language specifically to make the implementation of templates such as the above more concise.

Supplementary Exercise

(A small extension) Add a unary & operator ("address-of") that returns a pointer to the currently active function.

The definition of the static data member of this template deserves a little extra discussion:

```
template<class R, class T1, class T2>
typename STC<R, T1, T2>::PF STC<R, T1, T2>::cur_pf_ = 0;
```

The keyword `typename` is used to indicate that the name following the keyword does, in fact, denote the name of a type (§C.13.5). However, you cannot just sprinkle your code with `typename` wherever you name a type. More precisely, you *must* use the keyword `typename` in front of a name that satisfies all of the following:

1. The name is qualified—that is, it contains a scope-resolution operator '`::`'
2. It appears in a template
3. The qualified name has a component left of this scope resolution operator that depends on a template *parameter* (*not* a template *argument*)
4. It denotes a type
5. It is not used in a list of base classes or as an item to be initialized by a constructor initializer list

You are *allowed* to use the keyword in this sense *only* if all of the above apply except perhaps the third point. To illustrate this rather subtle rule, consider this erroneous code fragment:

```
template<typename1 T>
struct S: typename2 X<T>::Base {
   S(): typename3 X<T>::Base(typename4 X<T>::Base(0)) {}
   typename5 X<T> f() {
      typename6 X<T>::C *p; // Declaration of pointer p
      X<T>::D *q; // Multiplication!
   }
   typename7 X<int>::C *s_;
};

struct U {
   typename8 X<int>::C *pc_;
};
```

Consider each occurrence of `typename` in this example. They are numbered with subscripts for easy reference. The first, $\texttt{typename}_1$, indicates a template argument. The rules above do not apply to this first usage. The second and third `typenames` are disallowed by the fifth item in the rules above. Base-class names in these two contexts cannot be preceded by `typename`. However, $\texttt{typename}_4$ is required. Here, the name of the base class is not used to

denote what is being initialized or derived from. Instead, the name is part of an expression to construct a temporary `X<T>::Base` from its argument 0 (a sort of conversion, if you will). The fifth `typename` is prohibited because the name that follows it, `X<T>`, is not a qualified name. The sixth occurrence is required if this statement is to declare a pointer. The next line omits the `typename` keyword and is, therefore, interpreted by the compiler as a multiplication. This illustrates the fundamental reason why `typename` was introduced into the language: It allows compilers to better understand the intent of your templates, thereby producing earlier and more accurate diagnostics when the code contains syntax errors. The seventh `typename` is optional because it satisfies all the above rules except the third. Finally, $\texttt{typename}_8$ is prohibited because it is not used inside a template.

Exercise 14.2

Complete the `Ptr_to_T` class from §11.11 as a template using exceptions to signal run-time errors.

Hint: A classic, relatively straightforward application of C++ templates.

The class below belongs to the category of "smart pointers." The "pointer" part of this name comes from the fact that we are attempting —to some extent— to emulate the behavior of built-in pointers. The "smart" qualifier indicates that we're also attempting to provide some extra functionality for this emulation. Smart pointers are typically implemented as templates to maintain a generality close to the concept of built-in pointers.

Let's first templatize the class-definition of §11.11:

```
template<typename T>
struct Ptr {
   // Pointer to single object:
   Ptr(T *obj);
   // Pointer to array-element:
   Ptr(T *obj, T *array, int size);
   T& operator*() const; // Dereference
   Ptr<T>& operator++(); // Pre-increment
   Ptr<T> operator++(int); // Post-increment
   Ptr<T>& operator--(); // Pre-decrement
   Ptr<T> operator--(int); // Post-decrement
private:
   T *obj_, *array_;
   int size_;
};
```

First, the dereference operator has been made a `const` member. After all, you can dereference an ordinary pointer that is `const`. Making the operator `const` allows it to be applied to both `const` and non-`const` objects of type `Ptr<T>`. Note that the increment and decrement operators were not modified in this way and will not work on const `Ptr<T>` objects.

The second significant change is that the increment and decrement operators now return a `Ptr<T>` to emulate the consistency of plain C++ pointers. Whether to return *by value* or *by reference* is a subtle point that was resolved here in favor of emulating the behavior of the built-in operators. Indeed, the built-in pre-decrement and pre-increment operators modify their argument and then produce that modified argument—not a copy of it—as the lvalue (§4.9.6) result. On the other hand, the post-decrement and post-increment operators first save a copy of their argument—which will be the result of the expression—and then modify the argument. Because the result of the post-operations is a temporary copy, they correspond to rvalues. All this is reflected in the template definition above: *Pre*-operations return references (to their argument); whereas, *post*-operations return a `Ptr<T>` object by value.

Here is how you could implement these operators:

```
#include <stdexcept>

template<typename T>
T& Ptr<T>::operator*() const {
   if (obj!=0)
      return *obj_;
   else
      throw std::invalid_argument(String("Ptr error"));
}

template<typename T>
Ptr<T>& Ptr<T>::operator++() {
   if (obj_!=0 and array_!=0) {
      if (obj_<array_+(size_-1)) {
         ++obj_;
         return *this;
      } else
         throw std::out_of_range(String("Ptr error"));
   } else
      throw std::invalid_argument(String("Ptr error"));
}

template<typename T>
Ptr<T> Ptr<T>::operator++(int) {
   if (obj_!=0 and array_!=0) {
      if (obj_<array_+(size_-1))
         return Ptr<T>(obj_++, array_, size_);
      else
         throw std::out_of_range(String("Ptr error"));
   } else
      throw std::invalid_argument(String("Ptr error"));
}
```

As expected, the increment should not be applied to pointers that do not point into an array. The decrement operators are totally similar to their incrementing counterparts:

```
template<typename T>
Ptr<T>& Ptr<T>::operator--() {
   if (obj_!=0 and array_!=0) {
      if (obj_>array_) {
         --obj_;
         return *this;
      } else
         throw std::out_of_range(String("Ptr error"));
   } else
      throw std::invalid_argument(String("Ptr error"));
}

template<typename T>
Ptr<T> Ptr<T>::operator--(int) {
   if (obj_!=0 and array_!=0) {
      if (obj_>array_)
         return Ptr<T>(obj_--, array_, size_);
      else
         throw std::out_of_range(String("Ptr error"));
   } else
      throw std::invalid_argument(String("Ptr error"));
}
```

Now that the operators are implemented, we are ready to handle the constructors:

```
template<typename T>
Ptr<T>::Ptr<T>(T *obj): obj_(obj), array_(0), size_(0) {}

template<typename T>
Ptr<T>::Ptr<T>(T *obj, T *array, int size):
   obj_(obj), array_(array), size_(size) {
   if (size_<0)
      throw std::invalid_argument();
   if (obj_!=0 and array_!=0
         and not(obj_>=array_ and obj_<array_+size_))
      throw std::out_of_range(String("Ptr error"));
}
```

Convince yourself that the default (compiler-generated) copy-constructor and copy-assignment operators do the right thing.

Supplementary Exercise

(A refinement leading to a useful addition for your C++ toolbox) This version of `Ptr` handles both single objects and objects within arrays. Implement, instead, a pair of templates, each with a dedicated use: `Ptr<T>` objects handle only single objects while `ArrayPtr<T>` objects point to objects within arrays only. Discuss advantages and/or disadvantages of this approach versus the one outlined above.

Smart pointers are used in many contexts but are especially useful when dealing with exceptions. In this context, they usually perform a special cleanup action in their destructors. In fact, the standard library includes such a smart pointer template called `auto_ptr`. I usually recommend using the standard library facilities unless there is a compelling circumstance to use an alternative. However, I have found that for my uses, `std::auto_ptr` incurs too large an overhead and has somewhat subtle ownership semantics (that is, you must be doubly careful when assigning or copying `auto_ptr` objects). Instead, I have crafted simpler, more specialized templates of my own.

Supplementary Exercise

(A simple but very usable implementation) Design and implement a simple smart pointer type (or a pair of these types to cover both simple objects and arrays of objects). The semantics of this type should be extremely basic: One or a few access functions, such as the dereference operator, should be provided, and the object that is pointed to should be `delete`d (or `delete[]`d) upon destruction. Inhibit copying such objects by making the copy-constructor and copy-assignment operators private. Evaluate the performance of your solution compared to that of `std::auto_ptr` with the C++ systems available to you. Describe advantages and disadvantages of both smart pointers.

Supplementary Exercise

(A much more sophisticated refinement) In the previous supplementary exercise, I kept the issue of object ownership simple by suggesting that no copy operations should be allowed. Ponder what it would take to allow meaningful copying of a smart pointer whose destructor needs to destroy and deallocate the object pointed to. Can you come up with a solution that you prefer over the approach taken with `std::auto_ptr`? The primary purpose for permitting copying of these smart pointers is to allow returning them as the result of a function call. You might, therefore, consider adding a helper class, `PtrCapsule<T>`, whose sole purpose is to transfer ownership of a smart pointer.

Exercise 14.9

Given a

```
int main() { /* ... */ }
```

change it so that it catches all exceptions, turns them into error messages and `abort()`s. Hint: `call_from_C()` in §14.9 doesn't handle all cases.

Hint: See Exercise 10.15. More evidence that objects whose nontrivial construction outside the scope of any function can lead to brittle software.

Here is how the `call_from_C` technique could be applied to `main()`:

```
int main() {
   try {
      /* ... */
   } catch (...) {
      cout << "Caught final exception.\n";
      return 1;
   }
}
```

The trouble with this scheme is that it may not catch exceptions that were thrown during the construction of objects with *static storage duration*. They may be constructed before the first statement of `main()` is initiated; this construction could produce an exception. There is, unfortunately, no simple and guaranteed way to catch these exceptions. If you need a portable partial solution, replace global objects of the type:

```
YourClass global_object;
```

by functions like

```
YourClass& global_object() {
   try {
      static YourClass object;
      return object;
   } catch (...) {
      // ...
   }
}
```

References to such global objects, as in

```
void f() {
   global_object.mf(); // call to member function
}
```

are then transformed as follows:

```
void f() {
   global_object().mf(); // call to member function
}
```

This does not take care of the object's destructor, however. If it throws an exception, there is no portable way to catch it. In particular, the specifications of the C++ language explicitly say that the use of a *function-try-block* on `main()` will not work:

```
int main() try {
   /* ... */
} catch (...) {
   cout << "Caught exception.\n"
           "(but not from static object's destructor)\n";
}
```

This is unlikely to be a major problem because there are many more reasons not to throw exceptions from destructors. Again, it may appear unfortunate that there are no guarantees about this behavior.

If an exception is not eventually caught, the C++ system will call the standard function `terminate` or alternatively the last function you installed with `set_terminate` (§14.7). You can thus customize catastrophic failure as follows:

```
void panic( ) {
spawn_recovery_process( );
   cerr << "Panic! Recovery process started. \n"
   "Main process aborting...\n";
   abort( );
}

int main( ) {
   set_terminate(panic);
   // ...
}
```

Exercise 14.10

Write a class or template suitable for implementing callbacks.

Hint: A fundamental exercise with many possible solutions. See Exercise 18.2.

In the traditional library software model, your application calls a library function, which performs some internal computations—perhaps with some side effects—and finally returns control to your application, possibly also producing a return value. Often, this library-is-endpoint model is satisfactory; but sometimes it is not. Instead, there may be a need for the library to control the execution of a piece of code provided by the application. Examples of this situation pervade the standard library. For example, the `std::sort` function template

takes a comparison object that is used by the sorting algorithm to determine the ordering of two elements (§18.7.1).

Note that the term *callback* is not always used in this context. In particular, it would seem that callback in C++ is more commonly used when an indirect function call is involved—that is, either a virtual function call (§12.2.6) or, a call through a pointer-to function (§7.7), or through a pointer-to-member function (§15.5). The comparison object for `std::sort` is thus usually not called a callback, but terminology can vary.

The simplest form of callback is a pointer-to-function. The C library routine `std::qsort` (§18.11), for example, takes such a pointer to determine how elements are ordered (a less flexible alternative to the approach taken in the more modern `std::sort` template). Often, there is a need to pass some context along with the function. In C++, this context is conveniently passed as an object, and the callback action could be a member function of that object. Therefore, a simple solution to the callback problem is:

```
struct Callback {
   virtual void operator()() = 0;
};
```

Here, `Callback` is an abstract base class from which specific kinds of `Callbacks` can be derived. For example, the regular pointer-to-function callback can be encapsulated as follows:

```
template<void (*PF)()>
struct CallbackFunction: Callback {
   virtual void operator()() { PF(); }
};
```

Let me digress for a paragraph to clarify what is happening here. Template arguments are most commonly used to parameterize types. However, they can also be used to parameterize values at compile time; such parameters are called *non-type template parameters* (§13.2.3). If such a non-type parameter has an integral type, it should be substituted by a constant value. However, if the template parameter has a pointer, including a pointer-to-function, or reference type, then it should be substituted for by a pointer or reference with *external linkage* (§9.2). Thus, pointers to local variables or to functions made local to a translation unit using the `static` keyword (§B.2.3) or using unnamed namespace (§8.2.5.1) can never be used as non-type template arguments. The template `CallbackFunction` uses such a non-type template parameter whose type is a pointer-to-function. The alternative would be to store the function pointer inside the object at run-time. The non-type template parameter approach has the advantage of keeping the object smaller (but this gain is rarely worthwhile), and usually results in a faster callback. The alternative has the advantage of allowing the callback function to be selected at run-time. There is, of course, nothing to prevent you from providing both alternatives simultaneously.

Here is how this callback mechanism could be used:

```
#include <stdio.h>

void a() { printf("Hello\n"); }
void b() { printf("World!\n"); }
```

```
void greet(Callback const &first, Callback const &second) {
   first();
   second();
}

int main() {
   greet(CallbackFunction<&a>(), CallbackFunction<&b>());
}
```

To get familiar with the syntax of member templates and advanced usage of templates, let's implement a `Callback` whose action consists of calling a member function of an object. To achieve this purpose, we must somehow maintain two pieces of information: the member function to call and the object to which this member function should be applied. Here is a possible solution:

```
template<typename T, void (T::*MF)()>
struct ObjectCallback {
   ObjectCallback(T &object): object_(object) {}
   void operator()() { object_.MF(); }
private:
   T &object_;
};
```

The comments that applied to the use of a non-type template parameter still apply here. Again, there is an alternative solution that selects the pointer-to-member function at run-time. Once you feel comfortable with these concepts and constructs, you may want to sink your teeth into the following supplementary exercises.

Supplementary Exercise

(Long and tedious) In the above development, the callbacks correspond to functions that take no argument and return no value. Modify the `Callback` base class to be a template with seven type parameters. The first type parameter determines the return type of the callback; the remaining parameters correspond to the first through sixth callback parameter.

Supplementary Exercise

(More of the same, but the result is a pearl.) Refine the template you developed for the previous supplementary exercise so that each template type parameter can be `void` (and is `void` by default). To make this possible, you will need to partially specialize the `Callback` template so that when, for example, its last template argument is `void`, the callback function does not take a sixth argument. Similarly, when the first template argument is `void`, the callback function will not return a value. See the sample answer to Exercise 14.1.

Supplementary Exercise

The standard library sometimes provides convenience functions, such as `std::make_pair`, that make the construction of objects of complex template class types simpler. Provide such functions that would make the use of your seven-parameter `Callback` template simpler.

Callbacks are not intrinsically related to exceptions. However, once you try designing more complex callbacks that also require robust error handling, you will find that exceptions are often the error-handling mechanism of choice. This, in turn, requires that the library be designed with the possibility of exceptions in mind. Such a library might even go a step further and make its error handling configurable through callbacks.

Chapter 15

Class Hierarchies

Chapter 12 reviewed the basics of run-time polymorphism in C++. This chapter builds on those concepts with two fairly extensive exercises. The first—Exercise 15.2—studies the delicate question: What is an object's identity while it is being constructed? This question, in turn, determines how calls to virtual functions dispatch while constructors are still active. Our study uses run-time type identification (§15.4) to trace an object's state as it is being constructed and subsequently destroyed.

Exercise 15.3 presents the longest solution in this book. It is the complete implementation of a popular board game. The presentation discusses the details of how the problem can be decomposed in objects that naturally reflect the solution components: the board, the players, and so forth. The solution also provides for configurability and extensibility, properties that are highly desirable in modern software.

Exercise 15.2

Write a program that illustrates the sequence of constructor calls at the state of an object relative to RTTI during construction. Similarly illustrate destruction.

Hint: RTTI (run-time type information) is described in §15.4. This exercise illustrates precisely defined, but slightly subtle, object construction issues.

We will work with the hierarchy sketched in Figure 15.1.

The diagram uses the common convention of having derived classes point to their direct base classes. Some of the base classes are plain concrete classes (§10.3) while others are polymorphic (§13.6.1)—that is, they have or inherit virtual members. Similarly, the hierarchy has regular and virtual base classes.

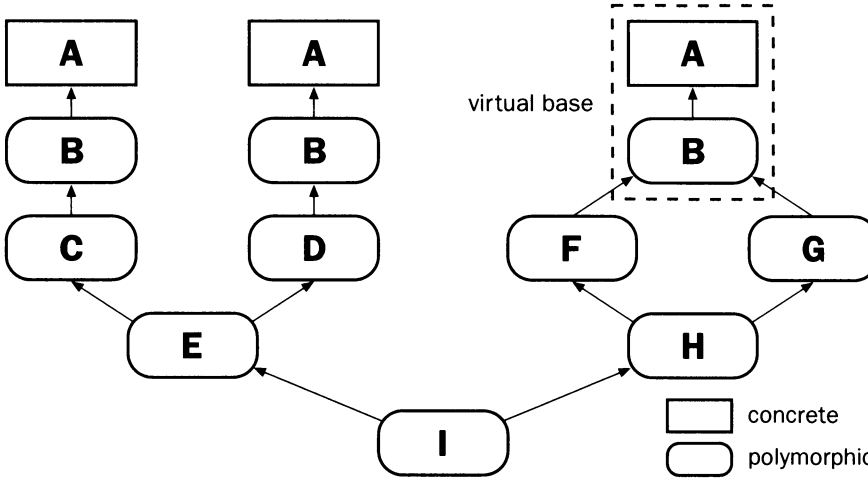

Figure 15.1 *A hierarchy of derived classes*

The following is the code that declares this hierarchy with constructors that use the `typeid` operator to determine the identity of the object at various stages of construction and destruction. The identity as seen from both outside and inside the object is examined.

```
#include <iostream>
#include <typeinfo>

using namespace std;
```

The following macros will allow more compact class definitions:

```
#define CTOR(CC) {                                    \
   cout << #CC " constructor: "                       \
        << typeid(*this).name() << '\n';              \
   from_outside(this);                                \
   cout << '\n';                                      \
}

#define DTOR(CC) {                                    \
   cout << #CC " destructor: "                        \
        << typeid(*this).name() << '\n';              \
   from_outside(this);                                \
    cout << '\n';                                     \
}
```

Within a macro '#,' is the *stringize* operator. It turns its argument into a string-literal. For example, when expanding `CTOR(C++)`, `#CC` will expand to `"C++"`, including the double-quote characters. The macros also rely on the string-literal concatenation property (§5.2.2), which makes `"He" "llo"` equal to `"Hello"`.

```
void from_outside(struct A*);

struct A {
   A() CTOR(A)
   ~A() DTOR(A)
};

void from_outside(A *object) {
   cout << "Located at address: " << (void*)object
        << "\nFrom outside: " << typeid(*object).name()
        << endl;
}
```

The `typeid` operator is most interesting when applied to polymorphic types. However, struct `A` is not polymorphic. So the function `from_outside` is overloaded for various polymorphic base class types in the hierarchy:

```
void from_outside(struct B *object);

struct B: A { // Single inheritance, non-polymorphic base
   B() CTOR(B)
   virtual ~B() DTOR(B) // makes the class polymorphic
};

void from_outside(B *object) {
   cout << "Located at address: " << (void*)object
        << "\nFrom outside: " << typeid(*object).name()
        << endl;
}
```

Were this version of `from_outside` omitted, calling `from_outside(rb)`, where `rb` is a reference to an object of type `B`, would still output only information about the `A` subobject of that `B`. Indeed, the `typeid` operator will return only type information for the complete object if the type of the argument refers to a polymorphic type (such as `B`).

```
struct C: B { // Single inheritance, polymorphic base
   C() CTOR(C)
   virtual ~C() DTOR(C) // make the class polymorphic
};
```

```
struct D: B { // Single inheritance, polymorphic base
   D() CTOR(D)
   virtual ~D() DTOR(D) // make the class polymorphic
};

void from_outside(struct E*);

struct E: C, D { // Multiple inheritance, regular bases
   E() CTOR(E)
   virtual ~E() DTOR(E) // make the class polymorphic
};

void from_outside(E *object) {
   from_outside((C*)object); // Resolve conversion to B*
   from_outside((D*)object); // Ditto
}
```

Looking at the sketch of the class hierarchy above, it appears that type E has class B twice as its base class. So a call to `from_outside(B*)` with an argument of type E* would fail because there is no hint about which subobject pointer should be passed—the base of C or the base of D? I resolve this by adding another overloaded version of this function, `from_outside(E*)`, which calls the version taking a B* after unambiguously converting the E* argument to C* and D* pointers.

```
struct F: virtual B {
   F() CTOR(F)
   virtual ~F() DTOR(F) // make the class polymorphic
};

struct G: virtual B {
   G() CTOR(G)
   virtual ~G() DTOR(G)
};

struct H: F, G { // Multiple inheritance, virtual bases
   H() CTOR(H)
   virtual ~H() DTOR(H)
};

void from_outside(struct I*);

struct I: E, H { // All of it
   I() CTOR(I)
   virtual ~I() DTOR(I)
};
```

```
void from_outside(I *object) {
   from_outside((C*)object); // Resolve conversion to B*
   from_outside((D*)object); // Ditto
   from_outside((H*)object); // And once more
}

int main() {
   I complex_object;
   cout << "Total size of I-object: " << sizeof(I) << endl;
}
```

The internal mechanics of constructing an object are far from trivial when the type of the object is so complex. Early C++ implementations differed in their behavior, and one could often not reliably call a virtual function on an object under construction or destruction. The function `from_outside` might reveal such problems. It would presumably produce unexpected output if the object did not appropriately update the internal structures that determine its identity. In fact, even a relatively recent compiler claimed through `typeid` that the completed object `complex_object` was of type `E` (it should have been type `I`). You might also find that some compilers produce an executable program that fails to execute to completion at all. (Read: It crashes.) You may then have to trim the program somewhat to identify the cause of failure.

In what order are the various parts or an object of type `I` constructed? This is, in fact, precisely defined by the language. First a definition:

> *A base class* `L` *is said to be left of another base class* `R` if `L` (or one of its derived classes) appeared before `R` (or one of its derivations) in a list of base classes. For example, the base class `B` of class `C` above *is left of* class `H`, because the class `E` that (indirectly) derives from that base class `B` appears before `H` in the list of base classes of `I`.

With this definition, we can enumerate the rules that determine the order of construction:

1. Base class subobjects are constructed before their derivations.
2. Virtual base class subobjects are constructed before other base classes; if there is more than one virtual base subobject, they are constructed left to right (using the definition above).
3. Nonvirtual base subobjects are constructed in left-to-right order.

Therefore, the program above should construct the base subobject of type `A` that is common to the subobjects of type `F` and `G` first (because it's the base class subobject of a virtual base). The virtual base class subobject of type `B` to which this `A`-subobject belongs is constructed next. At this point, all the virtual base subobjects have been constructed (only one, really). Next is the left-most nonvirtual base subobject, the `A` part of the unique `C` subobject. Then `C`'s `B` part and finally `C` itself are completed. The `D` subobject is constructed similarly (first its `A` part, then its `B` component). Once `C` and `D` are completed, the next left-most item is the unique `E` subobject. Next are `F`, `G`, and `H` (in that order) and finally the complete object of type `I`. As you can tell, the names of the various class types were chosen to match the

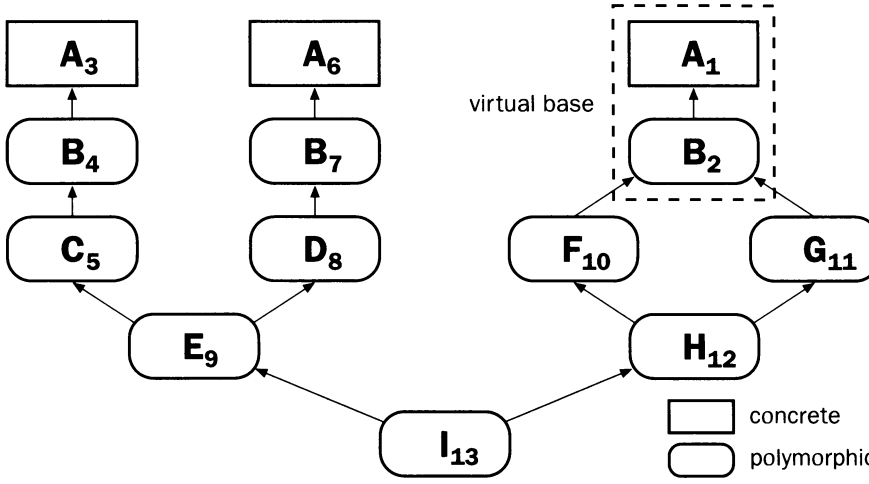

Figure 15.2 *Object structure indicating the order of construction*

order of construction. The object structure, seen in Figure 15.2, is annotated with ordinals indicating the order of construction—destruction occurs in reverse order.

Although the order of construction is well-defined, it is generally wise to minimize your code's dependencies on this order. Doing otherwise introduces additional subtleties that make understanding the program harder.

One final aspect of the above program is the size of the final object: How much memory is required to correctly handle all the object and subobject identity properties of class `I`? You'll find that this differs from system to system. I tried the program on six compilers, all of which implemented pointers and integers as 4-byte (32-bit) entities. Three of the compilers used 40 bytes for an `I`-object, one used 31 bytes, another used 28 bytes, and the most compact representation used just 20 bytes. Strangely, the compiler using the most compact representation was also the only one that handled the program correctly. You might wonder whether some 32-bit systems might use an even smaller representation for type `I`. Although not theoretically impossible, it is unlikely. Any efficient implementation probably needs space for at least five addresses (pointers) or memory offsets (integers). The exact reason for this is beyond the scope of this discussion.

Supplementary Exercise

(A small extension) The example does not actually demonstrate that virtual bases of different types are constructed in left-to-right order because only one type was used for virtual bases: struct B. Modify the example to alleviate this shortcoming.

Exercise 15.3

Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player smart enough to be worth playing against.

Hint: Perhaps the most "fun" exercise in this book. Requires a solid understanding of classes, expressions, and statements, and a little familiarity with the standard library (mostly input/output). No use of templates or exceptions.

Let's quickly review the rules of this deceptively simple game. It is played on an eight-by-eight grid of squares. One player plays black, the other plays white; black always opens the game. A move consists of placing a disc of your own color onto the board. Each straight-line sequence of pieces of the opposite color between your new piece and another piece of your own color then turns to your color. Figure 15.3 shows an example of this mechanism.

The gray piece indicates the next move of black. As you can see, this move causes three white pieces to turn black because they were trapped between the new black piece and previous black pieces. If there is no position in which you can capture a piece of your opponent, you must give up your turn. If the board is full or if neither player can capture a piece, the game terminates and the winner is the player who has the most pieces on the board. The initial state of the board is depicted in Figure 15.4.

In order to implement this game, it is useful to examine what the natural objects are: the pieces, the board, the players. These concepts will probably map to some user-defined types. The pieces are presumably quite simple—they are either black or white. The board could be represented as an array of states: black, white, or empty. Clearly, these states are close relatives of the pieces, and from an implementation point of view it may be more convenient to encode both in the same type, say:

```
enum Color { kEmpty = 0, kWhite, kBlack };
```

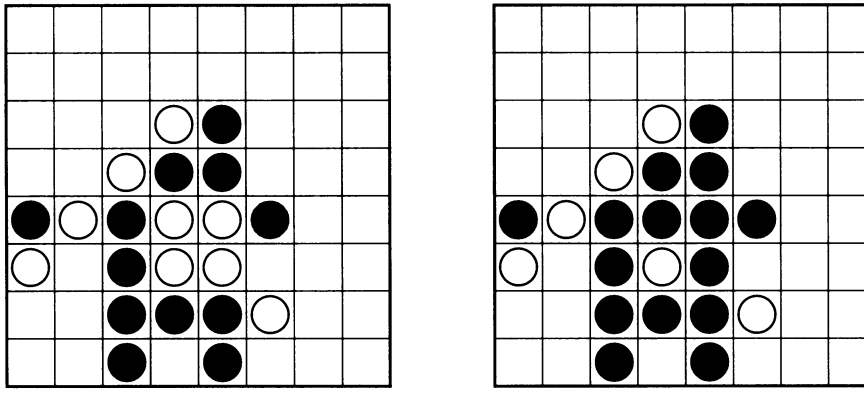

Figure 15.3 *Sample Reversi move: black plays the gray position*

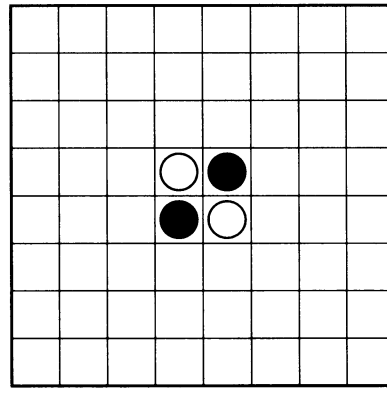

Figure 15.4 *Initial state of the board in a game of Reversi*

A board could then be defined as a two-dimensional array:

```
typedef Color Board[8][8];
```

A player is assigned a `Color` and is then repeatedly requested to make (or skip) a move until the game terminates. Hence, we could specify an abstract `Player` type (§12.3) as follows:

```
struct Player {
   virtual ~Player() {}
   virtual void get_move(Board const&,
                         int &row, int &col) = 0;
   virtual void skip_move(Board const&) {}
};
```

From this type, we could derive a `HumanTextPlayer` that implements `get_move` by taking input from `cin` and a `VirtualTextPlayer` that uses a simple algorithm to determine a move and reports it to `cout`. Other derivations that, for example, accept input from a graphical interface could also be developed.

We do not have all the elements to bring movement in the game yet. We need an entity that coordinates it all. This could be as simple as a function, but in the spirit of object orientation, let's map it onto a type `Game`:

```
struct Game {
   Game(Player *black, Player *white);
   void play();
private:
   // To be determined
};
```

We could place all the rule enforcing and administrative responsibilities in this type, but that would defeat our goal of trying to decompose a larger problem into smaller ones. Here are some of the responsibilities we may want to investigate further:

- Who initializes the board?
- Who draws the board at each stage of the game?
- Who keeps the score?
- Who determines whether a move is valid?
- Who handles situations in which a player must skip a turn?

Initializing the board sounds like the job of the `Board` constructor. However, our tentative `Board` type does not have a user-defined constructor. Hence, it may be worthwhile to make the board a class of its own. By making the board a full-fledged class, we open the door to adding more intelligence to it. For example, rather than treat the board type as an array in which we must validate all the moves and flip the colors when necessary, we can just "ask the board to attempt a move." If the move is valid, the board flips all necessary colors; if not, the move has no effect on the board state. The success of the move can be indicated by a conventional return value. Given such an approach, you may agree that the board is also the best place to keep the scores:

```
enum MoveResult { kUnsuccessful = 0, kSuccessful };

struct Board {
   Board();
   Color const& operator()(int r, int c) const
      { return b_[r][c]; }
   MoveResult move(int r, int c, Color color);
   int score(Color color) const
      { return color==kWhite? white_score_: black_score_; }
private:
   Color b_[8][8];
   int black_score_, white_score_;
};
```

I also decided to add an operator that provides a read-only interface to the state of each grid position. This is in preparation for the next question: Who draws the board at each step of the game?

You might take the view that this should also be the responsibility of the board. Because we do not want to exclude either a textual or a graphical interface, we could make `Board` an abstract class with a pure virtual member `draw()`. A derivation would then take care of rendering the board appropriately. However, what if you decide later that the same board can be viewed in multiple ways simultaneously? For example, one of the viewers could be a logging facility to a printer. The inheritance model could be restrictive in that case. Instead, I prefer having a separate `BoardViewer` abstract base class that only encapsulates a pointer

to a Board (which remains a concrete class). This separation of a model (the board) and its viewer turns out to be an essential foundation of modern user-interface design. We will place a BoardViewer under the control of the Game object.

The above is but a sketch and needs considerable development.

```
// File: Board.H
#ifndef BOARD_H
#define BOARD_H

#include <assert.h>

enum MoveResult { kUnsuccessful = 0, kSuccessful };
enum Color { kEmpty = 0, kWhite, kBlack };

inline Color opponent_of(Color color) {
   assert((color==kWhite) or (color==kBlack));
   return color==kWhite? kBlack: kWhite;
}

struct Board {
   Board();
   Color operator()(int r, int c) const { return b_[r][c]; }
   void move(Color, int r, int c);
   MoveResult get_possible_move(Color,
                                int &r, int &c) const;
   MoveResult valid_move(Color, int r, int c) const;
   int score(Color color) const
      { return color==kWhite? white_score_: black_score_; }
   enum { kNRows = 8, kNCols = 8 };
private:
   Color b_[kNRows+2][kNCols+2];
   int black_score_, white_score_;
};
#endif
```

The refined Board class—whose interface is placed in a file with include guards (§9.3.3)—encapsulates a slightly larger array to allow for a layer of empty "guard positions" around the actual board position, as illustrated in Figure 15.5.

These guard positions simplify the tests that validate moves. There is no need to verify that a running index stays within bounds. For example, the loop

```
while (b_[row][col]==opponent)
```

will fail when hitting the boundary of the grid because then b_[row][col]==kEmpty. The enumerators kNRows and kNCols could conceivably be modified to experiment with non-traditional board sizes.

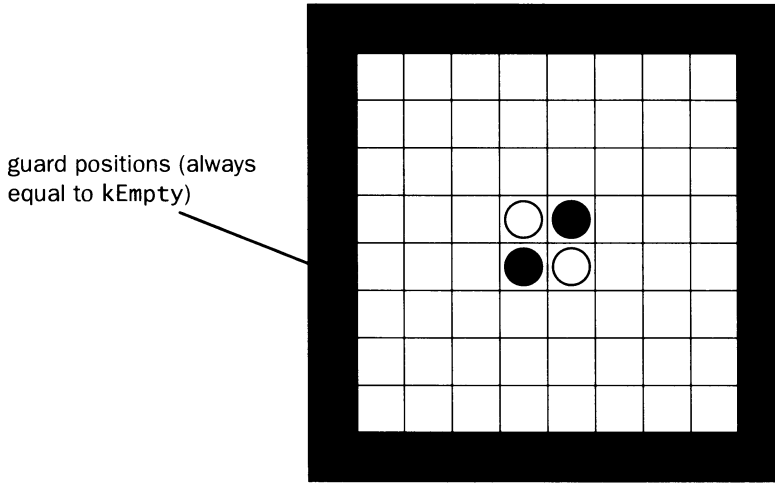

Figure 15.5 *Empty guard positions around the board*

I also added a public member function `get_possible_move()` that will, for example, help the `Game` object determine when a player must skip a turn. This function takes a color as its first argument. If a valid move exists for this color, `kSuccessful` is returned, and the `r` and `c` parameter are set to a valid move. Otherwise, `kUnsuccessful` is returned.

```
// File: Board.C
#include <assert.h>
#include "Board.H"

Board::Board() {
   for (int r = 0; r!=kNRows+2; ++r)
      for (int c = 0; c!=kNRows+2; ++c)
         b_[r][c] = kEmpty;
   b_[kNRows/2+1][kNRows/2]
      = b_[kNRows/2][kNRows/2+1] = kBlack;
   b_[kNRows/2][kNRows/2]
      = b_[kNRows/2+1][kNRows/2+1] = kWhite;
   black_score_ = white_score_ = 2;
}
```

The constructor initializes the board to contain only the initial center pieces.

The following member checks a move—a color and a position—for validity. To do this, the function starts off at the given (`r`, `c`) position and tries all eight possible directions to see if a line of consecutive opponent's pieces followed by an own piece exists. The eight directions are described by (`dr`, `dc`) pairs generated by the following code:

```
   for (int dr = -1; dr!=2; ++dr)
      for (int dc = -1; dc!=2; ++dc)
         if ((dr!=0) or (dc!=0)) {
            // ...
         }
```

Note how, in fact, nine pairs are generated, but the pair (`dr==0`, `dc==0`) is skipped because adding that pair to a position would not actually get us anywhere.

```
MoveResult Board::valid_move(Color color,
                             int r, int c) const {
   assert((r>=1) and (r<=kNRows)
      and (c>=1) and (c<=kNCols));
   if (b_[r][c]!=kEmpty)
      return kUnsuccessful;
   Color opponent = opponent_of(color);
   for (int dr = -1; dr!=2; ++dr)
      for (int dc = -1; dc!=2; ++dc)
         if ((dr!=0) or (dc!=0)) {
            int row = r, col = c;
            do {
               row += dr;
               col += dc;
            } while (b_[row][col]==opponent);
            if ((b_[row][col]==color)
            and ((row!=r+dr) or (col!=c+dc)))
               return kSuccessful;
         }
   return kUnsuccessful;
}
```

The next member function is similar but actually changes the color of the captured pieces. Note that the assertion '`assert(valid_move(color, r, c)`' is expensive, but it can be disabled by `#define`-ing the preprocessor symbol `NDEBUG` (§24.3.7.2).

```
void Board::move(Color color ,int r, int c) {
   assert(valid_move(color, r, c));
   Color opponent = opponent_of(color);
   int count = 0;
   for (int dr = -1; dr!=2; ++dr)
      for (int dc = -1; dc!=2; ++dc)
         if ((dr!=0) or (dc!=0)) {
            int row = r, col = c;
            do {
               row += dr;
               col += dc;
            } while (b_[row][col]==opponent);
```

```
            // If this direction captures something,
            // change the color of the captured pieces:
            if ((b_[row][col]==color)
            and ((row!=r+dr) or (col!=c+dc)))
               for (row = r+dr, col = c+dc;
                    b_[row][col]==opponent;
                    row += dr, col += dc) {
                  b_[row][col] = color;
                  ++count;
               }
         }
      b_[r][c] = color;
      if (color==kBlack) {
         black_score_ += count+1;
         white_score_ -= count;
      } else {
         white_score_ += count+1;
         black_score_ -= count;
      }
   }

   MoveResult Board::get_possible_move(Color color,
                                       int &r, int &c) const {
      for (r = 1; r!=kNRows+1; ++r)
         for (c = 1; c!=kNCols+1; ++c)
            if (valid_move(color, r, c))
               return kSuccessful;
      return kUnsuccessful;
   }
```

Let's add the code that determines the interface and the implementation of the "board-rendering system" (a big phrase for a little piece of code, but in the software industry using such phrases has occasionally proven to be successful):

```
// File: BoardViewer.H
#ifndef BOARDVIEWER_H
#define BOARDVIEWER_H

struct Board;
struct BoardViewer {
   virtual void draw(Board&) = 0;
};
#endif
```

The header file above defines an abstract class that introduces a simple interface. In this exercise, we'll limit ourselves to coding a single concrete implementation that renders a given board on cout.

```
// File: BoardTextViewer.H
#ifndef BOARDTEXTVIEWER_H
#define BOARDTEXTVIEWER_H

#include "BoardViewer.H"
#include <iostream>

using std::ostream;

struct BoardTextViewer: BoardViewer {
   BoardTextViewer(ostream &output): output_(output) {}
   virtual ~BoardTextViewer() {}
   virtual void draw(Board&);
private:
   ostream &output_;
};
#endif


// File: BoardTextViewer.C
#include "Board.H"
#include "BoardTextViewer.H"
#include <assert.h>

void BoardTextViewer::draw(Board &board) {
   assert((Board::kNCols<10) && (Board::kNRows<27));
   int r, c;
   output_ << "\n   ";
   for (c = 1; c!=Board::kNCols+1; ++c)
      output_ << ' ' << c;
   output_ << "\n  .-";
   for (c = 1; c!=Board::kNCols+1; ++c)
      output_ << "--";
   for (r = 1; r!=Board::kNRows+1; ++r) {
      output_ << "\n " << char('a'+r-1) << '|';
      for (c = 1; c!=Board::kNCols+1; ++c)
         if (board(r, c)==kWhite)
            output_ << " o";
```

```
            else if (board(r, c)==kBlack)
               output_ << " *";
            else
               output_ << "  ";
      }
      output_ << "\n[ Black(*): " << board.score(kBlack)
              << "    White(o): " << board.score(kWhite)
              << " ]\n\n";
   }
```

Not very fancy—black pieces are displayed as asterisks, white pieces as o characters—but it will do, and it has the advantage of being portable to a wide variety of platforms. Also note that this view handles only boards with no more than nine columns and no more than 26 rows because I wanted to use only one character for the coordinates (nine digits for the columns, 26 letters for the rows).

Next, we need to define the interface for a `Player` and add the implementation for a human player using a text interface. The implementation of a virtual player (computer) will be discussed later beacuse that's a little more intricate. Assume for now that a `VirtualTextPlayer` type is available.

```
// File: Player.H
#ifndef PLAYER_H
#define PLAYER_H

#include "Board.H"

struct Player {
   virtual ~Player() {}
   virtual void get_move(Board const&, int &r, int &c) = 0;
   virtual void skip_move(Board const&) {}
   virtual void win() {}
   virtual void lose() {}
   virtual void tie() {}
};
#endif
```

The `Game` object will repeatedly ask a player for a move through the member `Player::get_move` unless no valid move exists, in which case the `Player` is given a chance to react via an invocation of `Player::skip_move`. A player need not store the board; it is passed as an argument for each move. Once the game is over, the player is notified of the result through a call to `win`, `lose`, or `tie`.

```
// File: HumanTextPlayer.H
#ifndef HUMANTEXTPLAYER_H
#define HUMANTEXTPLAYER_H
```

```
#include <iostream>
#include "Board.H"
#include "Player.H"

using std::istream;
using std::ostream;

struct HumanTextPlayer: Player {
   HumanTextPlayer(ostream &output, istream &input,
      Color color)
         : output_(output), input_(input), color_(color) {}
   virtual void get_move(Board const&, int &row, int &col);
   virtual void skip_move(Board const&);
   virtual void win();
   virtual void tie();
private:
   ostream &output_;
   istream &input_;
   Color color_;
};
#endif
```

Notice how the input and output channels are passed to the `HumanTextPlayer`'s constructor. This should allow simple customization if `cout` and `cin` are not the desired input/output streams.

```
// File: HumanTextPlayer.C
#include <assert.h>
#include "HumanTextPlayer.H"

namespace { // anonymous namespace
   void convert(char &ch, int upper) {
      if (ch>='a' and ch<'a'+upper)
         ch = ch-'a'+1;
      else if (ch>='A' and ch<'A'+upper)
         ch = ch-'A'+1;
      else if (ch>='1' and ch<'1'+upper)
         ch = ch-'1'+1;
   }

   bool in_board(int row, int col) {
      return (row>0) and (row<=Board::kNRows)
         and (col>0) and (col<=Board::kNCols);
   }
```

```
   void print_color(ostream &output, Color color) {
      if (color==kBlack)
         output << "Black";
      else
         output << "White";
   }
} // end anonymous namespace

void HumanTextPlayer::get_move(Board const &board,
                               int &r, int &c) {
   char row, col;
   do {
      print_color(output_, color_);
      output_ << ", enter your move (e.g., a7) : ";
      input_ >> row >> col;
      convert(row, Board::kNRows);
      convert(col, Board::kNCols);
      if (in_board(row, col)
      and board.valid_move(color_, row, col))
         break;
      board.get_possible_move(color_, r, c);
      output_ << "Invalid move. Please try again.\n"
              << char('a'+r-1) << char('1'+c-1)
              << " is a valid example,"
                 " but other moves may be better.\n";
   } while (1);
   r = row; c = col;
}

void HumanTextPlayer::skip_move(Board const&) {
   print_color(output_, color_);
   output_ << ": no valid move possible.\n";
}

void HumanTextPlayer::win() {
   print_color(output_, color_);
   output_ << " wins!\n";
}

void HumanTextPlayer::tie() {
   if (color_==kBlack)
      output_ << "Game tied.\n";
   else
      output_ << "No winner!\n";
}
```

The member function `Player::lose` was intentionally left unchanged. If one player is said to win, it is not necessary to say that the other one loses.

Supplementary Exercise

(Minor correction and potential extensions) The text-based interface presented in `HumanTextPlayer` and `BoardTextViewer` makes unportable assumptions about the character set. For example, it is not guaranteed that all the lowercase letters are encoded using consecutive values. Change the code to remove that assumption. Also improve the interface in any other way you can imagine. Are there any changes you would make to the abstract (interface) classes `BoardViewer` and `Player`?

Now we're ready to show the `Game` class that will manage the board and the two players. As mentioned, you might just as well implement this as a function. The `Game` object also changed a little from the original discussion:

```
// File: Game.H
#ifndef GAME_H
#define GAME_H

#include "Board.H"

struct BoardViewer;
struct Player;

struct Game {
   Game(Player *black, Player *white)
      : black_(black), white_(white) {}
   void select_viewer(BoardViewer *viewer)
      { viewer_ = viewer; }
   void play();
private:
   Player *black_, *white_;
   Board board_;
   BoardViewer *viewer_;
};
#endif
```

The `BoardViewer` can be selected after the construction of the `Game` object. This makes it easier to have an interactive choice for board rendering (not exploited here).

```
// File: Game.C

#include "BoardViewer.H"
#include "Game.H"
#include "Player.H"
```

```
void Game::play() {
   bool more_moves;
   viewer_->draw(board_);
   do {
      int r, c;
      more_moves = false;
      if (board_.get_possible_move(kBlack, r, c)) {
         more_moves = true;
         black_->get_move(board_, r, c);
         board_.move(kBlack, r, c);
         viewer_->draw(board_);
      } else
         black_->skip_move(board_);
      if (board_.get_possible_move(kWhite, r, c)) {
         more_moves = true;
         white_->get_move(board_, r, c);
         board_.move(kWhite, r, c);
         viewer_->draw(board_);
      } else
         white_->skip_move(board_);
   } while (more_moves);
   if (board_.score(kWhite)>board_.score(kBlack)) {
      white_->win(); black_->lose();
   } else
   if (board_.score(kBlack)>board_.score(kWhite)) {
      black_->win(); white_->lose();
   } else {
      black_->tie(); white_->tie();
   }
}
```

Let's put all the pieces together (still assuming we will have a `VirtualTextPlayer` type):

```
// File: Reversi.C
#include "BoardTextViewer.H"
#include "Game.H"
#include "HumanTextPlayer.H"
#include "VirtualTextPlayer.H"
#include <iostream>

using std::cout;
using std::cin;
```

```
Player* select_player(Color color) {
   cout << (color==kWhite? "White: ": "Black: ")
        << "[H]uman or [C]omputer? ";
   char answer;
   cin >> answer;
   if (answer=='h' || answer=='H')
      return new HumanTextPlayer(cout, cin, color);
   else
      return new VirtualTextPlayer(cout, color);
}

int main() {
   Game game(select_player(kBlack), select_player(kWhite));
   game.select_viewer(new BoardTextViewer(cout));
   game.play();
   return 0;
}
```

Although we have spent several pages developing our game, you may agree that so far things have been conceptually relatively simple. At the very least, we've stayed within the domain of knowledge covered in *The C++ Programming Language*. I'm also hopeful that you'll find the decomposition of the problem natural and amenable to convenient modification and extension. You'll note the special attention to separating responsibilities and to isolating the implementations of all those responsibilities from each other. Yet I also made mistakes; I'll come back to them later on.

The last part of this implementation of the Reversi game is the code that allows the computer to choose moves. This will delve into a basic game strategy technique that you might want to skip on first reading.

An uninteresting strategy would be to evaluate every board position for validity and randomly pick any legal move. Such an opponent would be easily defeated. Instead, the computer should try to associate a value with each valid board position and select the best one. This is easier said than done. In reply to a move, the opponent usually has a variety of choices for the next move, but we do not know what the adversary's choice will be. A reasonable heuristic, however, is that the opponent will play as well as possible. Therefore, among all the valid moves we can make, we will pick (in order of preference):

- One that ends the game in our favor
- One for which the best reply of our opponent gives him or her the least advantage

The program will need to *simulate* our potential moves and the corresponding potential countermoves of our opponent on separate `Board` objects. The value associated with a simulated move of the opponent will be computed using the same method except that what is good for the opponent is bad for the player, and vice-versa. This simulation of moves and countermoves could theoretically be repeated to a large depth, but, in practice, you would find that only a few levels can be explored in a reasonable amount of time. At some point, we will stop the recursive evaluation and associate a numerical value with the board situation last simulated.

This method is often referred to as a *min-max* algorithm because it alternately selects the maximum value for our move and the minimum value for the opponent's countermove.

The following code realizes this idea:

```
// File: VirtualPlayer.H
#ifndef VIRTUALPLAYER_H
#define VIRTUALPLAYER_H

#include "Board.H"

struct VirtualPlayer {
   VirtualPlayer(Color c): color_(c), maxdepth_(6) {}
   void get_move(Board const&, int &row, int &col);
private:
   int eval_self(Board const&, int r, int c,
                 int best, int worst, int depth);
   int eval_opponent(Board const&, int r, int c,
                     int best, int worst, int depth);
   int eval_end_of_game(Board const&);
   int value_of(Board const&);
   Color color_;
   int maxdepth_;
};
#endif

// File: VirtualPlayer.C
#include "VirtualPlayer.H"
#include <limits>

void VirtualPlayer::get_move(Board const &board,
                             int &row, int &col) {
   int best = std::numeric_limits<int>::min(),
       worst = std::numeric_limits<int>::max();
   row = col = 0;
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (board.valid_move(color_, r, c)) {
            int score = eval_self(board, r, c,
                                  best, worst, 1);
            if (score>=best) {
               best = score;
               row = r; col = c;
            }
         }
}
```

```
int VirtualPlayer::eval_self(Board const &board,
                             int row, int col,
                             int best, int worst, int d) {
   if (d==maxdepth_) // Maximum depth of exploration?
      return value_of(board);
   Board copy(board);
```

It may be possible that we are evaluating the virtual player's position just after it had to forfeit its simulated turn. That cast would be indicated with `row==0`.

```
   if (row>0)
      copy.move(color_, row, col);
```

We must next test every board position to see if it could be a valid response for the opponent. If this is the case, we must find out how good a move it was (using `eval_opponent`). If we find that the opponent's countermove will place us in a worse situation than that which our other moves simulated so far lead to (described by the parameter `best`), we can discard this move right away. Otherwise, we remember what the worst score is (that is, the score of the best countermove of the opponent).

```
   bool move_result = kUnsuccessful;
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (copy.valid_move(opponent_of(color_), r, c)) {
            move_result = kSuccessful;
            int score = eval_opponent(copy, r, c,
                                      best, worst, d+1);
            if (score<=best) // This won't get any better:
               return score; // "prune" this branch now.
            if (score<worst)
               worst = score;
         }
```

If the opponent had a valid countermove, we return the score associated with the best one. If we can still make a move, we simulate the opponent forfeiting a turn by calling `eval_opponent` with a (0, 0) move. Otherwise, we have simulated the end of a game and return the evaluation of that.

```
   if (move_result==kSuccessful)
      return worst;
   else // The opponent had no valid response
      if (copy.get_possible_move(color_, row, col)
            ==kSuccessful)
         return eval_opponent(copy, 0, 0, best, worst, d+1);
      else // We reached an end-of-game situation
         return eval_end_of_game(copy);
}
```

Looking at the `eval_self` function above, you may have noticed a twist on the basic min-max algorithm. Remember that `eval_self` searches, for a given board situation, all the opponent's possible moves and selects the lowest possible score (indicating the best choice of the opponent). We are interested in finding the board situation that would force the highest such low point. Therefore, if we have already evaluated a low point for one board situation (the `best` parameter) and another situation reveals a lower score, we can abandon the search in that board right away because its low point can only decrease and we will never steer the game to that situation. The `eval_opponent` function will have a totally symmetric role and can, therefore, abandon searching a board as soon as that board situation promises to produce a high point that is higher than the lowest high point for previous situations. This early-termination technique is known as α–β pruning—α and β refer to the high points and low points that are carried along in the simulation (parameters `best` and `worst` in the evaluation functions). The implementation of `eval_opponent` below is identical to `eval_self` except that the roles of self and opponent are swapped.

```
int VirtualPlayer::eval_opponent(Board const &board,
                                 int row, int col,
                                 int best, int worst, int d)
{
   if (d==maxdepth_) // Maximum depth of exploration reached?
      return value_of(board);
   Board copy(board);
   if (row>0) // Row==0 when we have no valid move
      copy.move(opponent_of(color_), row, col);
   bool move_result = kUnsuccessful;
   // Explore own possible responses:
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (copy.valid_move(color_, r, c)) {
            move_result = kSuccessful;
            int score = eval_self(copy, r, c,
                                  best, worst, d+1);
            if (score>=worst) // This won't get any better:
               return score;  // "prune" this branch now.
            if (score>best)
               best = score;
         }
   if (move_result==kSuccessful)
      return best;
   else // We have no valid response
      if (copy.get_possible_move(opponent_of(color_),
                                 row, col)==kSuccessful)
         return eval_self(copy, 0, 0, best, worst, d+1);
      else // We reached an end-of-game situation
         return eval_end_of_game(copy);
}
```

To complete the virtual player, we must write the functions evaluating board situations that will not be explored further. This occurs for two reasons: The situation corresponds to the end of a game, or we have reached the maximum depth that we're willing to explore. The former case is relatively inflexible. If the situation corresponds to a win, we give it a very large score. If it's a tie, we return 0. Otherwise, we return a very negative value. The second case will be essential in the quality of the game played by the computer. If we can produce values that are truly representative of the potential development of the situation under evaluation, the computer will tend to steer the game towards the good developments and possibly win. Perhaps the most straightforward evaluation function is the one that returns the material advantage of the player—how many more pieces does the virtual player have on the board than its opponent. However, *position* is also crucial. For example, a corner position can never be taken away from a player and is, therefore, something to strive for. On the other hand, having a piece next to an empty corner position can sometimes be very dangerous because the opponent can use it to capture the powerful corner position. The following function can certainly be improved on:

```
int VirtualPlayer::eval_end_of_game(Board const &board) {
   if (board.score(color_)
          >board.score(opponent_of(color_)))
      return std::numeric_limits<int>::max()-1;
   if (board.score(color_)
          >board.score(opponent_of(color_)))
      return std::numeric_limits<int>::min()+1;
   else
      return 0; // The value of a tie could be biased
}               // if desired.

namespace { // unnamed namespace
   int corner_value(Color player, Color corner,
                    Color p1, Color p2, Color p3) {
      Color opp = opponent_of(player);
      if (corner==player)
         return 30;
      if (corner==opponent)
         return -30;
      int value = 0;
      if ((p1==player) || (p2==player) || (p3==player))
         value -= 30;
      if ((p1==opp) || (p2==opp) || (p3==opp))
         value += 30;
      return value;
   }
} // end unnamed namespace
```

```
int VirtualPlayer::value_of(Board const &s) {
   Color opponent = opponent_of(color_);
   // First component: material advantage
   int value = s.score(color_)-s.score(opponent);
   // Second component: corner positions
   int t = 1, l = 1, r = Board::kNCols, b = Board::kNRows;
   value += corner_value(color_, s(t, l),
                         s(t+1, l), s(t, l+1), s(t+1, l+1));
   value += corner_value(color_, s(t, r),
                         s(t+1, r), s(t, r-1), s(t+1, r-1));
   value += corner_value(color_, s(b, r),
                         s(b-1, r), s(b, r-1), s(b-1, r-1));
   value += corner_value(color_, s(b, l),
                         s(b-1, l), s(b, l+1), s(b-1, l+1));
   return value;
}
```

All that is left to do is to plug this virtual player into the `Player` interface. Note again how we did not combine a user interface in the `VirtualPlayer` class. It is, therefore, convenient to reuse `VirtualPlayer` for a fancy graphical implementation of the game or for any other desired interface.

```
// File: VirtualTextPlayer
#ifndef VIRTUALTEXTPLAYER_H
#define VIRTUALTEXTPLAYER_H

#include "Player.H"
#include <iostream>

struct VirtualTextPlayer: Player {
   VirtualTextPlayer(std::ostream &output, Color color)
      : output_(output), color_(color) {}
   virtual void get_move(Board const&, int &row, int &col);
   virtual void skip_move(Board const&);
   virtual void win();
   virtual void tie();
private:
   std::ostream &output_;
   Color color_;
};
#endif
```

```
// File: VirtualTextPlayer.C
#include "Board.H"
#include "VirtualPlayer.H"
#include "VirtualTextPlayer.H"

namespace { // unnamed namespace
   void print_color(std::ostream &output, Color color) {
      if (color==kBlack)
         output << "Black";
      else
         output << "White";
   }
} // end unnamed namespace

void VirtualTextPlayer::get_move(Board const &board,
                                 int &row, int &col) {
   VirtualPlayer machine(color_);
   machine.get_move(board, row, col);
   print_color(output_, color_);
   output_ << " plays ["
           << char('a'+row-1) << char('1'+col-1) << "]\n";
}

void VirtualTextPlayer::skip_move(Board const&) {
   print_color(output_, color_);
   output_ << ": no valid move possible.\n";
}

void VirtualTextPlayer::win() {
   print_color(output_, color_);
   output_ << " wins!\n";
}

void VirtualTextPlayer::tie() {
   if (color_==kBlack)
      output_ << "Game tied.\n";
   else
      output_ << "No winner!\n";
}
```

Voilà. We're done.

Unfortunately, the implementation has some minor flaws. Fixing them is the object of the following Supplementary Exercise.

Supplementary Exercise

(Minor adjustements) The design of the Reversi game failed to factor out some commonality between the two kinds of text-based players. It could also be more flexible with regard to the size of the board: Why not make it a quantity that can be chosen at runtime? A similar comment applies to the maximum depth of exploration of our simple virtual player. Address these issues. Also consider the possibility of merging `eval_self` and `eval_opponent` into a single function.

As it stands now, the computer plays well enough to beat me—which may not say much. However, it has some weaknesses that you can address.

Supplementary Exercise

(A realistic performance-tuning exercise) The strength of the machine's game can be improved in at least two ways: increasing the efficiency of the exploration algorithm (thereby allowing greater exploration depth) and tuning the `VirtualPlayer::value_of` function. Improve both.

[blank page]

Chapter 16

Library Organization and Containers

In just a few short years, the C++ standard library algorithm and containers have gained a tremendous reputation in terms of the fine-grained reusability they provide without compromising performance. The following exercises introduce a few practical idioms and "gotchas." Familiarity with these will make your use of the standard library so much more productive.

Exercise 16.1

Create a `vector<char>` containing the letters of the alphabet in order. Print the elements of that vector in order and in reverse order.

Hint: Fundamental idioms for accessing and outputting standard containers. See Exercise 16.2.

To fill the vector, we can make use of a literal string and take advantage of the fact that pointers in that array are valid iterators.

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

char const L[] = "abcdefghijklmnopqrstuvwxyz";

int main() {
   using std::copy;
   std::vector<char> alphabet(&L[0], &L[0]+sizeof(L));
   copy(alphabet.begin(), alphabet.end(),
        std::ostream_iterator<char, char>(std::cout, " "));
   copy(alphabet.rbegin(), alphabet.rend(),
        std::ostream_iterator<char, char>(std::cout, " "));
   return 0;
}
```

Remember that many standard library algorithms take a sequence described by an iterator pointing to the first element and one pointing just one position beyond the last element (§2.7.2). For a built-in array such as `L` above, these two pointers can be designated with the expressions `&L[0]` and `&L[0]+sizeof(L)/sizeof(L[0])`. In our example, `sizeof(L[0])==1` since `sizeof(char)` equals one by definition. The `copy` algorithm (§18.6.1) is generally useful to transfer data from one standard container to another. This example illustrates that output streams can also be treated as a sort of write-only sequence through stream iterators (see §19.2.6). In the above example, the second argument to the `std::ostream_iterator` constructor specifies which characters to use to separate the values that are output. The first template argument for that constructor indicates the type of the elements to be output; the second template argument corresponds to the type used by the underlying stream for the external representation of the streamed data (`char` or `wchar_t`). The input sequence to copy a vector was obtained through that vector's member functions. In particular, a *reversed* vector sequence is delimited by the iterators returned by the `rbegin` and `rend` members.

The above example cannot be handled by many C++ implementations available at the time of this writing. For the sake of illustration, here is what an adaptation for an earlier implementation may look like. Such adaptations will, hopefully, be unnecessary in the near future.

```
#include <algorith.h>
#include <iostream.h>
#include <iterator.h>
#include <vector.h>
using namespace std;

char const letters[] = "abcdefghijklmnopqrstuvwxyz";

int main() {
   vector<char> alphabet;
   copy(&letters[0], &letters[0]+sizeof(letters),
        back_inserter(alphabet));
   copy(alphabet.begin(), alphabet.end(),
        ostream_iterator<char>(cout, " "));
   copy(alphabet.rbegin(), alphabet.rend(),
        ostream_iterator<char>(cout, " "));
   return 0;
}
```

A first modification consisted of adding the '`.h`' suffix to all the included standard C++ headers. Many implementations have not yet moved to the newer header syntax, or have done so only partially. For example, a few implementations kept `<iostream.h>`, which is historically an older C++ header, but do provide `<vector>` and `<iterator>`, for example. Similarly, some implementations do not yet support namespaces at all, and for those we would have to omit the line

```
using namespace std;
```

Again, some implementations do support namespaces but have moved only part of the library inside namespace `std`. Vectors are in it, but streams (`cout, cin, ...`) are not. The implementation used for the adaptation had problems handling header names of more than eight characters (not including the suffix); that explains why `<algorith.h>` was used instead of `<algorithm.h>`.

You may find that the container constructors that take a range delimited by two iterators are either absent or severely limited (usually because the compiler cannot yet handle *member templates*, §13.6.2). The above code illustrates how such an absence could be compensated for by constructing an empty object followed by appending the source range using a `copy` operation. The `back_inserter` function (§3.8, §19.2.4) returns an input iterator that appends elements to the given container.

As mentioned, the `ostream_iterator` class template normally takes two template arguments. However, the second template argument, which selects the type for external representation of the stream, has no meaning on implementations that do not templatize streams with respect to the external representation. Consequently, some earlier `ostream_iterator` implementations accept only the first template argument.

Chapter 3 discusses a number of other issues that arise when dealing with C++ compilers that do not fully conform to the ISO standard.

Exercise 16.2

Create a `vector<string>` and read a list of names of fruits from `cin` into it. Sort the list and print it.

Hint: Some similarities with Exercise 16.1. A useful preparation for Exercise 10.19.

The following program accepts only single-word fruits (see Exercise 10.19 for an example of how to remove that limitation) and treats a word starting with a period as an indication of the end of the significant input. As usual, an unsuccessful stream extraction from `cin` is also considered to mean that no more input is available.

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

using std::string;

// Utility function-object to convert alphabetic characters
// to lower-case:
struct LowerCase {
   operator()(char const &c) const
      { return std::tolower(c); }
};
```

```
void process_data(std::vector<std::string> &fruits) {
   std::sort(fruits.begin(), fruits.end());
   std::copy(fruits.begin(), fruits.end(),
             std::ostream_iterator<string>(std::cout, "\n"));
}
int main() {
   string fruit;
   std::vector<string> fruits;
   while((std::cin >> fruit) and (fruit[0]!='.') {
      std::transform(fruit.begin(), fruit.end(),
                     fruit.begin(), LowerCase());
      fruits.push_back(fruit);
   }
   process_data(fruits);
   return 0;
}
```

This program was refined as follows: Whenever a fruit name is input, the program converts each character to lowercase using the standard `transform` algorithm. The first two arguments to that function describe the range to be transformed; the third argument indicates the first position at which the result should be deposited. The last argument is a *function object* (also known as functor, §18.4); it is an object of a type that provides a function call operator (`operator()(...)`). Here we created such a type `LowerCase` using the standard C library function `tolower` (§20.4.2).

Supplementary Exercise

(Fairly advanced) Adapt the above program to use the localization facilities (header <locale>) of the C++ standard library to lower the case of the input.

Note that some written languages do not have a concept of lowercase or uppercase; other languages do not deal with case on a symbol-by-symbol basis. This makes the design of a truly generic case-lowering function complicated.

Exercise 16.3

***Using the vector from Exercise 16.2, write a loop to print the names of all fruits with the initial letter* a.**

Hint: A small variation on a theme that is getting familiar.

Once the program in Exercise 16.2 is working, it requires comparatively little effort to add this sort of functionality. Only the function `process_data` needs modification:

```
void process_data(std::vector<string> &fruits) {
   for (std::vector<string>::iterator p = fruits.begin();
        p!=fruits.end(); ++p)
      if ((*p)[0]=='a')
         std::cout << *p << '\n';
}
```

Supplementary Exercise

(Not difficult) Rewrite the previous code using `copy_if` (§18.6.1).

Exercise 16.6

Using the vector from Exercise 16.2, write a loop to delete all fruits whose taste you don't like.

Hint: This exercise is too easy if you like all fruits. The solution presented includes a short elimination algorithm that it is easy to get wrong.

Again, we need not do much more than rewrite the `process_data` function:

```
char const *dont_like[] = {
   "kumquat", "pomelo", "starfruit", "walnut", "watermelon",
   0 };

void process_data(std::vector<string> &fruits) {
   std::sort(fruits.begin(), fruits.end());
   std::vector<string>::iterator p = fruits.begin();
   for (int i = 0; (p!=fruits.end()) and dont_like[i];) {
      int order = p->compare(dont_like[i]);
      if (order>0) // *p "comes after" dont_like[i]
         ++i;
      else
      if (order<0)
         ++p;
      else // order==0, i.e. *p==dont_like[i]
         p = fruits.erase(p);
   }
   using std::copy;
   copy(fruits.begin(), fruits.end(),
        std::ostream_iterator<string>(std::cout, "\n"));
}
```

It is assumed that the header files and using-declarations of Exercise 16.2 are visible. The `string::compare` function is described in §20.3.8. Note that the elimination loop assumes that both the vector `fruits` and the array `dont_like` are sorted.

Supplementary Exercise

(Not difficult) Rewrite the above using one of the standard `set_` functions (§18.7.4).

Exercise 16.15

Outline the possible behavior of `duplicate_elements()` from §16.3.6 for a `vector<string>` with the three elements "don't do this."

Hint: A good illustration of how the validity of certain iterators can expire.

Let's reproduce the `duplicate_elements` function for convenience:

```
#include <string>
#include <vector>
using std::string;
using std::vector;

void duplicate_elements(vector<string> &f) {
   for (vector<string>::iterator p = f.begin();
        p!=f.end(); ++p)
      f.insert(p, *p);
}
```

As explained in §16.3.6, this function can lead to undefined behavior because the `insert` member invalidates iterators into the container.

Assume for a moment that this vector implementation has allocated enough space to hold eight strings and that a `vector` iterator is, in fact, a plain C/C++ pointer. These assumptions reflect the reality of many C++ standard library implementations.

When entering the function `duplicate_elements`, only three of the eight allocated cells are occupied by actual strings. At the first call `f.insert(p, *p)`, `p` points to "don't." The result is a four-element vector `f`, but `p` was not modified. Hence, `p` points to the first copy of "don't" and at the next iteration it will point at the second copy. As we repeat the insertion, it should be clear that we are inserting multiple copies of the string "don't" into the vector.

At some point, the vector `f` will contain:

```
don't
don't
don't
don't
```

```
don't
don't   <- p
do
this
```

The position of p just before the next insertion is shown. This next insertion, however, will cause the vector to exceed its current capacity of eight cells; reallocation is necessary. Several things can happen when reallocation occurs:

- The reallocation can fail (usually with a thrown exception). This is not common but must be taken into account.
- The reallocation simply extends the memory block currently used by the vector. All the pointers into this block (vector iterators) remain valid. This is probably rare and purely coincidental; don't count on it.
- The reallocation copies the existing vector elements to a fresh block of memory. Existing vector iterators now point to unallocated memory. Trying to dereference these iterators can trigger random effects at random times in your program.

Eventually, a call to the function `duplicate_elements` will almost certainly lead to a catastrophic program failure. (Read: It will *crash*.) How could we obtain the intended functionality instead? One way it to pre-size the vector:

```
#include <string>
#include <vector>

void duplicate_elements(std::vector<std::string> &f) {
   std::vector<std::string> ff;
   ff.resize(2*f.size());
   for (int i = 0; i!=f.size(); ++i)
      ff[2*i] = ff[2*i+1] = f[i];
   f.swap(ff);
}
```

Another way is to use `push_back` to avoid iterators:

```
#include <string>
#include <vector>

void duplicate_elements(std::vector<std::string> &f) {
   std::vector<std::string> ff;
   for (int i = 0; i!=f.size(); ++i) {
      ff.push_back(f[i]);
      ff.push_back(f[i]);
   }
   f.swap(ff);
}
```

Supplementary Exercise

(A good experiment) This last version is generally less efficient than the previous one. Can you see why? Verify this with experiments. Hint: How do various strings need to be copied in both cases?

Chapter 17

Standard Containers

The C++ standard library offers a rich variety of containers. Clearly, the choice that this implies is not gratuituous. Various containers strike different compromises with the efficiency and flexibility of the operations they support. Exercise 17.1 presents a basic study of these efficiency issues as an illustration of the relativity of the *O()* notation. Compare your numbers for the proposed toy benchmark with those reported here.

Exercise 17.1

Understand the O() notation (§17.1.2). Do some measurements of operations on standard containers to determine the constant factors involved.

Hint: A fundamental exercise for effectively using the C++ standard library.

The constant hidden in the *O* notation corresponds to a somewhat abstract idea: a multiplicative factor for "the number of constant-time operations" on an unspecified computing machine. The concept of an operation, however, is not strictly defined in C++. For example, how many operations are triggered by the following statement?

```
a = b+c;
```

I could say there are just two: one addition and one assignment. Someone else might note that this requires, in fact, four elementary operations on machine platforms: two *read* operations (to load the values of `b` and `c`), one addition, and one *write* operation (to store the new value of `a`). Someone else might suspect that the generation of valid machine addresses to access the various variables incurs additional operations. And finally, it may turn out that the result of this operation is, in fact, not used further in the program and the compiler may decide to optimize the statement away. Even if this is not the case, the processing unit of your computer may be able to overlap the computations for this expression with those for other statements.

Clearly, trying to get a feeling for the hidden constant by counting whatever we decide to be operations is not an ideal approach. Instead, we will use *relative execution time* to find relative constants for a small set of operations on a small set of containers of `int`s.

The containers whose performance we will measure are:

- A C-style array
- An `std::vector`
- An `std::deque`
- An `std::list`
- An `std::set`

For each of these, we perform the following operations:

- Destruction of a previous container, creation of a new one, and insertion (append where applicable) of `n` integers out of a *random reference sequence*—the *insertion test*
- Sum all `n` elements—the *traversal test*
- Look up the first occurrence of the $n/2^{th}$ inserted element—the *lookup test*

Here is the program that implements the measurements:

```
#include <deque>
#include <list>
#include <numerics>
#include <set>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <vector>
```

The following two variables determine the range of container sizes that we will test. We will start with containers holding `kMinSize` elements and then repeatedly double the number of elements until we exceed `kMaxSize`.

```
int const kMinSize = 1000;
int const kMaxSize = 8000;
```

To ensure fairness and reproducibility, we cannot just fill each container with a new set of random numbers. Instead, we generate one sequence `data` of random numbers once and for all and then reuse that sequence for all our test cases.

```
int *data = new int[kMaxSize];

void generate_data() {
   for (int k = 0; k!=kMaxSize; ++k)
      data[k] = rand();
}
```

Most modern compilers have options to optimize the generated code. A benchmark program should almost always be compiled using those options. However, good optimizers may notice that some of our test cases don't have any observable effects and may, as a consequence, be eliminated altogether. However, few optimizers are able to see through calls to functions with ellipsis arguments (§7.6). Hence, we define a function `consume` that forces most optimizers to assume that its arguments cannot just be discarded. (Conversely, it is best to avoid ellipsis arguments for functions that require aggressive optimization.)

```
void consume(int, ...) {}
```

Next, we define our various containers. For each of these, we write three short functions that implement the insertion, traversal, and lookup tests. First for C arrays:

```
int *c_array = new int[kMaxSize];

void c_array_fill(int size) {
   delete[] c_array;
   c_array = new int[size];
   for (int k = 0; k!=size; ++k)
      c_array[k] = data[k];
}

void c_array_traverse(int size) {
   int sum = 0;
   for (int k = 0; k!=size; ++k)
      sum += c_array[k];
   consume(0, sum);
}

void c_array_lookup(int size) {
   int const desired_value = data[size/2];
   int k;
   for (k = 0; k!=size; ++k)
      if (c_array[k]==desired_value)
         break;
   consume(0, k);
}
```

Next come the tests for a `vector<int>`:

```
std::vector<int> *std_vector = new std::vector<int>;

void vector_fill(int size) {
   delete std_vector;
   std_vector = new std::vector<int>;
   for (int k = 0; k!=size; ++k)
      std_vector->push_back(data[k]);
}
```

```
void vector_traverse(int) {
   consume(0, std::accumulate(std_vector->begin(),
                              std_vector->end(), 0));
}

void vector_lookup(int size) {
   consume(0, std::find(std_vector->begin(),
                        std_vector->end(),
                        data[size/2]));
}
```

Note how the code for a `deque` (§17.2.3) is virtually identical to that of the `vector`. This is a consequence of successful generic programming, as realized by the C++ standard library.

```
std::deque<int> *std_deque = new std::deque<int>;

void deque_fill(int size) {
   delete std_deque;
   std_deque = new deque<int>;
   for (int k = 0; k!=size; ++k)
      std_deque->push_back(data[k]);
}

void deque_traverse(int) {
   consume(0, std::accumulate(std_deque->begin(),
                              std_deque->end(), 0));
}

void deque_lookup(int size) {
   consume(0, std::find(std_deque->begin(),
                        std_deque->end(),
                        data[size/2]));
}
```

Similar comments hold for the `list` container...

```
std::list<int> *std_list = new std::list<int>;

void list_fill(int size) {
   delete std_list;
   std_list = new std::list<int>;
   for (int k = 0; k!=size; ++k)
      std_list->push_back(data[k]);
}
```

```
void list_traverse(int) {
   consume(0, std::accumulate(std_list->begin(),
                              std_list->end(), 0));
}

void list_lookup(int size) {
   consume(0, std::find(std_list->begin(), std_list->end(),
                        data[size/2]));
}
```

... and even for an associative container such as `std::set<int>`:

```
std::set<int> *std_set = new Set;

void set_fill(int size) {
   delete std_set;
   std_set = new std::set<int>;
   for (int k = 0; k!=size; ++k)
      std_set->insert(data[k]);
}
void set_traverse(int) {
   consume(0, std::accumulate(std_set->begin(),
                              std_set->end(), 0));
}

void set_lookup(int size) {
   consume(0, std_set->find(data[size/2]));
}
```

Next we need a test harness to plug our tests into:

```
void print_header() {
   printf(
"Container kind    Size    Insertion  Traversal  Lookup\n"
"-------------------------------------------------------\n"
   );
}

typedef void (*Test)(int);
```

The `calibrate` function returns a rough estimate of how many times a test must be repeated to last for a given number of seconds. This function is used to ensure that the benchmark program doesn't run for too long but for long enough to have a reasonable resolution on performance.

```
int calibrate(int seconds, Test test, int size) {
   clock_t end = clock()+seconds*CLOCK_PER_SEC;
   int n;
   for (n = 0; clock()<end; ++n)
      test(size);
   return n;
}
```

The next function then performs the actual measurement for a single container type and a given size.

```
double time_in_seconds(int n , Test test, int size) {
   clock_t start = clock();
   for (int k = 0; k!=n; ++k)
      test(size);
   return (clock()-start)/double(CLOCKS_PER_SEC);
}
```

The `benchmark` function uses `time_in_seconds` above to test a container type for sizes between `kMinSize` and `kMaxSize`. After initializing the reference data and calibrating the number of times each test must run based on the performance of a lookup in a C array, `main` then proceeds to call `benchmark` for each container being tested.

```
void benchmark(char const *container_name, int n,
               Test filling, Test traversal, Test lookup) {
   for (int size = kMinSize; size<=kMaxSize; size *= 2) {
      filling(size); // Set up starting data set
      printf("%13s  %7d:   %7.2f   %7.2f   %7.2f\n",
             size==kMinSize? container_name: "", size,
             time_in_seconds(n, filling, size),
             time_in_seconds(n, traversal, size),
             time_in_seconds(n, lookup, size));
   }
   printf("\n");
}

int main() {
   generate_data();
   c_array_fill(kMinSize); // Initialize for calibration
   int n = calibrate(1 /* sec */, &c_array_lookup, kMinSize);
   print_header();
   benchmark("C array", n,
             c_array_fill, c_array_traverse, c_array_lookup);
   benchmark("std::vector", n,
             vector_fill, vector_traverse, vector_lookup);
   benchmark("std::deque", n,
             deque_fill, deque_traverse, deque_lookup);
```

```
    benchmark("std::list", n,
              list_fill, list_traverse, list_lookup);
    benchmark("std::set", n,
              set_fill, set_traverse, set_lookup);
return 0;
}
```

Though a bit lengthy, the code is relatively straightforward. Note that the function `consume` may not be sufficient to fool very smart optimizers; treat all extreme measurements you find with a healthy amount of suspicion.

Figure 17.1 shows the output from this program on one system and for one compiler.

The quantitative aspects of these numbers are sure to vary from one platform to another. However, the qualitative aspects can differ, too. One remarkable observation in the above table, for example, is that although the performance of a summation over a `vector` is as fast as a straightforward loop construct through a C-style array, finding a value is twice as expensive in the `vector`. A good optimizing compiler should probably be able to see

Container kind	Size	Insertion	Traversal	Lookup
C array	1000:	1.95	0.77	0.65
	2000:	3.73	1.53	1.28
	4000:	7.29	3.06	1.80
	8000:	14.40	6.10	5.09
std::vector	1000:	11.41	0.78	1.15
	2000:	32.15	1.54	2.30
	4000:	101.56	3.06	3.23
	8000:	353.40	6.11	9.15
std::deque	1000:	36.93	12.23	4.12
	2000:	73.23	24.46	8.21
	4000:	146.11	48.98	11.57
	8000:	291.81	98.08	32.71
std::list	1000:	32.74	5.61	2.56
	2000:	65.00	11.20	5.09
	4000:	129.59	22.37	7.18
	8000:	255.17	44.82	20.33
std::set	1000:	143.41	11.74	0.07
	2000:	295.08	23.13	0.08
	4000:	600.68	44.83	0.08
	8000:	1209.73	84.31	0.08

Figure 17.1. *Measurements of operations on standard containers*

through the different ways of performing the same task and generate essentially identical code for the inner loops over a C array and over a standard `vector`.

The insertion test is not fair; C arrays do not easily "grow" and the performance of successive `push_back` operations on standard `vectors` could be greatly enhanced using the `resize` or `reserve` member functions. I find it, therefore, more interesting to treat the numbers for the insertion test in C arrays as outliers and, instead, compare the numbers among the various standard containers.

This leads to a more general observation: Even for a given data structure, there exist multiple approaches to achieve the same result. It is not always the case that one of these methods yields indisputably better performance than all the others. This is one more reason to use the results shown above with special care.

The tests I chose do not reflect the advantages of having a container with more flexible iterators. For example, looking at the numbers in the table only, it would appear that a `list<int>` performs about as well or better than a `deque<int>` of the same size. However, deques provide random access iterators that can dramatically reduce the complexity of certain algorithms.

Supplementary Exercise

(Requires some thought) Add a simple test to the above program that clearly demonstrates the advantages of `deques` over both `vectors` and `lists`.

As expected, looking up a value in a `set` is extremely efficient compared to lookup in a unsorted nonassociative container. It is, in fact, so efficient that on the platform I tested, the operation did not last long enough to obtain an accurate measurement.

The traversal test illustrates perhaps best the main point of this exercise: For every container, summation of all the elements in the container is an $O(n)$ operation (with n as the number of elements). This growth factor is confirmed by the numbers above. Yet at the same time, you'll agree that the various containers do not exhibit equivalent performance for this task.

The moral of the story: Choose the best containers for the tasks at hand.

Exercise 17.3

Write a program that lists the distinct words in a file in alphabetical order. Make two versions: one in which a word is simply a whitespace-separated sequence of characters and one in which a word is a sequence of letters separated by any sequence of nonletters.

Hint: There are many ways to obtain sorted sequences using the C++ standard library. Associative containers (§17.4) maintain a sorted sequence at all times.

This is convenient to implement if a container that merges duplicate elements and sorts its items is available—a `std::set` is such a container.

```
#include <iostream>
#include <fstream>
#include <set>
#include <string>

using std::string;

bool load_nospace_word(istream &input, string &word) {
   return input >> word;
}

bool load_letter_word(istream &input, string &word) {
   bool result;
   char c;
   while (input.get(c))
      if (isalpha(c)) {
         word = c;
         break;
      }
   while (result = input.get(c)) {
      if (!isalpha(c)) {
         input.putback(c);
         break;
      }
      word += c;
   }
   return result;
}

typedef bool (*Loader)(istream&, string&);

main(int argc, char *argv[]) {
   istream *input;
   Loader loader;
   int name_pos;
   if (argc>1 && strcmp(argv[1], "-letters")==0) {
      name_pos = 2;
      loader = &load_letter_word;
   } else {
      name_pos = 1;
      loader = &load_nospace_word;
   }
```

```
    if (argc==name_pos)
       input = &cin;
    else
    if (argc==name_pos+1)
       input = new ifstream(argv[name_pos]);
    else {
       cerr << "Unexpected arguments.\n";
       exit(-1);
    }
    string word;
    std::set<string, std::less<string> > listing;
    while (loader(*input, word))
       listing.insert(word);
    std::copy(listing.begin(), listing.end(),
              std::ostream_iterator<string>(cout, "\n"));
    if (argc==name_pos+1)
       delete input;
    return 0;
}
```

This program allows up to two optional command-line arguments. If an option "`-letters`" is used, words consisting of letters only will be selected; otherwise, any consecutive sequence of characters delimited by whitespace is considered a word. By default, input will come from `cin`, but the last command-line argument can indicate the name of a file to be opened for input.

Note how the actual heavy processing—ordering and ensuring uniqueness of the words that are input—corresponds to very little source code. The hard work is conveniently described in just a few lines of standard library invocations. Here most of the code deals with parsing the command line and structuring the input. The standard library provides comparatively few facilities to perform these tasks, but other public domain or commercial libraries do. For larger projects, it often pays to rely on such third-party libraries.

Exercise 17.6

Define a queue using (only) two stacks.

Hint: A classic problem. A queue is sometimes called a first-in first-out (FIFO) buffer, whereas a stack is last-in first-out (LIFO).

The idea is to use one stack (the *instack*) for pushing new elements; the other (the *outstack*) is used for popping elements off the queue. If a *pop* operation finds an empty outstack, the instack will be transferred to the outstack one element at a time. The resulting outstack contains the elements that used to be on the instack in reverse order. The latter is the mechanism that realizes the LIFO to FIFO conversion.

The following implementation of Queue uses the standard stack adapter (§17.3.1) on a configurable container (determined by the second template argument to the Queue template). By default, this container is an std::vector.

```
#include <stack>

template<typename T, typename C = std::vector<T> >
struct Queue {
   typedef typename C::value_type value_type;
   typedef typename C::size_type size_type;
   typedef typename C container_type;
```

Our Queue can be pre-loaded with data by passing a container with the initial data to the constructor:

```
   explicit Queue(C const &contents = C())
      : outstack_(contents) {}
```

Both underlying stacks must be empty for the Queue to be empty:

```
   bool empty() const
      { return instack_.empty() && outstack_.empty(); }
```

Similarly, the number of elements in the Queue is the total number of elements in both stacks:

```
   size_type size() const
      { return instack_.size()+outstack_.size(); }
```

The member function front returns a reference to the oldest element in the queue—that is, the next element that can be popped. If the outstack is empty, accessing this element requires the transfer of the instack to the outstack:

```
   value_type& front()
      { transfer(); return outstack_.top(); }
   value_type const& front() const
      { transfer(); return outstack_.top(); }
```

The member function back requires us to treat the Queue as if elements were flowing in the direction opposite to what is achieved by pushing and popping. Hence, a reverse-transfer operation is needed—more on this below.

```
   value_type& back()
      { untransfer(); return instack_.top(); }
   value_type const& back() const
      { untransfer(); return instack_.top(); }
   void push(value_type const& x) { instack_.push(x); }
   void pop() { transfer(); outstack_.pop(); }
```

```
private:
   void transfer() {
      if (outstack_.empty())
         while (!instack_.empty()) {
            outstack_.push(instack_.top());
            instack_.pop();
         }
   }
   void untransfer() {
      assert(cerr << "WARNING: Queue::untransfer()'ed.\n");
      if (instack_.empty())
         while (!outstack_.empty()) {
            instack_.push(outstack_.top());
            outstack_.pop();
         }
   }
   std::stack<T, C> instack_, outstack_;
   T *outbottom_;
};
```

The private `untransfer` operation is somewhat of a "hack" to accommodate the `back` access function. It solves the problem that the standard stack adapter has no access to its bottom element, but it makes the time complexity of `back` nonconstant and introduces the danger that `back` and `front` operations waste many machine cycles transferring elements back and forth between the two stacks.

On a somewhat related note, the standard adapters—unlike the standard containers—have no support for iterators. This makes it inconvenient and/or inefficient to implement, for example, the equality and inequality operators for this `Queue` type. Remember that the standard library also provides a queue and a priority queue container adaptor.

Supplementary Exercise

(Illustrates a fundamental concept: amortized complexity) Prove to yourself that if `back` operations are never applied to the above `Queue` class, the* amortized complexity *of pushing and popping elements onto and off the queue is constant. The amortized complexity is the average complexity of an operation since the creation of the `Queue`. A single operation may be costly, but that cost is amortized over a number of previous operations.

Chapter 18

Algorithms and Function Objects

Function objects (§18.4) are one of the most dramatic illustrations of how the C++ standard library achieves reusability through configurability, without sacrificing performance. The number of pre-defined function objects multiplies the number of pre-defined algorithms to generate a vast selection of efficient solutions to common programming problems. Furthermore, this system can be extended with your own algorithms and function objects.

The exercises in this chapter illustrate the use of a few algorithms and function objects, and also show how to implement these things to meet your specific needs. Very often, the solutions show that a little code may lead to a lot of convenience down the road.

Exercise 18.2

Implement and test the four `mem_fun()` and `mem_fun_ref()` functions (§18.4.4.2).

Hint: See Exercises 14.10 and 18.10. The techniques used to solve this exercise are often useful when dealing with function objects.

We will implement and discuss only one of these because they all look very much alike. Let's handle the case in which we want to apply a *member function* taking one argument using an *ordinary function* taking two arguments. The first argument in this ordinary function call is the object to which we wish to apply a member function of our choice; the second argument is the argument for that member function.

This interface adaptation is easily achieved using a function object (§18.4):

```
template<typename R, typename T, typename P1>
struct mem_fun1_ref_t {
   typedef R return_type;
   typedef T first_argument_type;
   typedef P1 second_argument_type;
   typedef R (T::*PMF)(P1); // PMF is a pointer-to-member-
                            // function type (§15.5)
```

```
    explicit mem_fun1_ref_t(PMF pmf): pmf_(pmf) {}
    R operator()(T &obj, P1 arg) const
       { return obj.pmf_(arg); }
private:
    PMF pmf_;
};
```

In principle, this adapter is sufficient for the purpose of applying member functions through the standard library algorithms. In practice, you may find that the syntax that creates such adapter objects is unwieldy. For example (the `transform` algorithm is described in §18.6.2):

```
transform(squares.begin(), squares.end(),
          angles.begin(), squares.begin(),
          mem_fun1_ref_t<void, Square, int>(&Box::rotate));
```

This can be improved by exploiting the fact that for function templates—as opposed to class templates—the template arguments can be deduced from the function arguments. Thus, by defining the function template

```
template<typename R, typename T, typename P1>
mem_fun1_ref_t<R, T, P1> mem_fun_ref(R (T::*pmf)(P1)) {
    return mem_fun1_ref_t<R, T, P1>(pmf);
}
```

the example shortens to

```
transform(squares.begin(), squares.end(), angles.begin(),
          squares.begin(), mem_fun_ref(&Box::rotate));
```

Another advantage of this convenience function is that many versions of it can be overloaded. As a result, we need not remember the names of member functions taking different numbers of arguments; `mem_fun_ref` can be used in all cases.

Exercise 18.5

Sort a list using only standard library algorithms.

Hint: A good test of your algorithmic skills in the context of the standard library. Remember, however, that the standard library often already includes the algorithms you need. Exercise 18.19 complements this.

Note that `std::list` has a `sort` member function that would almost always be preferred in practice. However, the purpose of this exercise is to juggle a little with the available non-member algorithms.

For the following solution, I choose to rewrite the QuickSort implementation presented in Exercise 8.1 in terms of bidirectional iterators and the standard library algorithms. This task is greatly simplified by the availability of the `std::partition` algorithm:

```
template<typename T, typename Less, typename Cursor>
void quicksort(Less &fcn, Cursor left, Cursor right) {
   // Zero- or one-element lists are trivially sorted:
   Cursor probe = left;
   if (probe==right or ++probe==right)
      return;

   T pivot = *left;
   // Partition the list into elements less-than and not
   // less-than the pivot (less-than is defined by fcn):
   Cursor p = partition(left, right, bind2nd(fcn, pivot));
   if (p==left)
      ++p;
   else
   if (p==right)
      --p;

   // Recursively sort these partitions:
   quicksort<T>(fcn, left, p);
   quicksort<T>(fcn, p, right);
}

template<typename T, typename Less>
void sort(std::list<T> &container,
          Less &fcn = std::less<T>()) {
   quicksort<T>(fcn, container.begin(), container.end());
}
```

Note that the `quicksort` template implements a sorting algorithm on any sequence delimited by bidirectional iterators—that is, the type `Cursor` denotes a bidirectional iterator.

Supplementary Exercise

Test the performance of this code on a list of integers that is initially sorted. Improve this performance by using a more sophisticated choice for the pivot element.

Exercise 18.10

Write a `binder3()` that binds the second and third arguments of a three-argument function to produce a unary predicate. Give an example where `binder3()` is a useful function.

Hint: Notice the similarities with Exercise 18.2.

Let's first come up with an example. A predicate (implemented as a function object) that is occasionally useful is

```
template<typename T>
struct in_range {
   bool operator()(T const &value, T const &a, T const&b) {
      return (value<b) and not (value<a);
   }
};
```

which tests whether its first argument is in the range (a, b). Clearly, it may sometimes be useful to apply an operation to elements of a sequence that fall within a given range. With the above predicate, this could be achieved using constructs like

```
int n_candidates = count_if(heights.begin(), heights.end(),
                            binder3(in_range(), 400, 600));
```

Writing a `binder3` function template requires a helper class:

```
template<typename FO>
struct binder2_3 {
   typedef typename FO::result_type result_type;
   typedef typename FO::first_argument first_argument;
   // Constructor:
   binder2_3(FO const &fo,
             typename FO::second_argument const &a2,
             typename FO::third_argument const &a3)
      : fo_(fo), a2_(a2), a3(a3) {}
   // Overloaded function-call operator:
   result_type operator()(first_argument a1) {
      return fo_(a1, a2_, a3_);
   }
private:
   FO fo_;
   typename FO::second_argument const a2_;
   typename FO::third_argument const a3_;
};
```

One tricky aspect of this class is `const`ness: Which parameters and members should be `const` and which should not? Allowing the bound arguments to change from invocation to invocation would lead to subtle situations. So choosing to have the members `a2_` and `a3_` be `const` is not a hard decision.

On the other hand, in order to allow the function object to update its internal state, it makes sense to make it non-`const`. However, remember that the object stored in the binder object is a *copy* of the one passed to its constructor; changes made to the state of `fo_` will not be reflected in the state of the original function object.

The actual binder function becomes

```
template<typename FO, typename P2, typename P3> inline
binder2_3<FO> binder3(FO const &fo,
                      P2 const &a2, P3 const &a3) {
   return binder2_3<FO>(fo, a2, a3);
}
```

Notice the similarities with the development in Exercise 18.2. For a quick review of the uses of the keyword `typename`, see Exercise 14.1.

Exercise 18.11

Write a small program that that removes adjacent repeated words from from a file file. Hint: The program should remove a 'that', a 'from', and a 'file' from the previous statement.

Hint: `unique_copy` (§18.6.3).

There are various ways to do this, but in this chapter we are demonstrating the standard library algorithms. Here is a compact, library-intensive solution:

```
#include <algorithm>
#include <fstream>
#include <iostream>

void error(char const *name, char const *msg) {
   std::cerr << name << msg << '\n';
   std::exit(1);
}

int main(int argc, char *argv[]) {
   using std::string;
   if (argc!=3)
      error(argv[0], ": Unexpected number of arguments.\n");
   std::ifstream instream(argv[1]);
   std::ofstream outstream(argv[2]);
   if (not instream or not outstream)
      error(instream? argv[2]: argv[1],
            ": cannot open file.\n");
   std::istream_iterator<string, char> in(instream), in_end;
   std::ostream_iterator<string, char> out(outstream, " ");
   std::unique_copy(in, in_end, out);
   return 0;
}
```

The standard function `unique_copy` (§18.6.3) does exactly what we need for this exercise. It also combines nicely with *input stream iterators* and *output stream iterators* (§19.2.6), as do most other standard algorithms.

Supplementary Exercise

(Just a little harder) The program above does not output whitespace as it appeared in the input file. Improve the situation by creating your own input iterator that stores leading whitespace along with a word. Combine this with the version of `unique_copy` that takes a predicate. Your predicate should ignore the leading whitespace when comparing two strings (words).

Exercise 18.14

Produce all anagrams of the word food. That is, all four-letter combinations of the letters* f, o, o, *and* d. *Generalize this program to take a word as input and produce anagrams of that word.

Hint: `std::next_permutation` (§18.10).

The anagrams of food are dfoo, fdoo, dofo, odfo, fodo, ofdo, doof, odof, oodf, food, ofod, and oofd.

The `std::next_permutation` function would be sufficient to generate the set of anagrams of a word if it weren't for the fact that `std::next_permutation` can generate duplicates because the letter *o* appears twice in the word *food*. A simple way to remove duplicates is to sort the generated permutation and apply the `std::unique` algorithm.

```
#include <algorithm>
#include <cstring>
#include <iterator>
#include <new>
#include <vector>
#include <string>

void error(char const *name, char const *msg) {
   std::cerr << name << msg << '\n';
   std::exit(1);
}

int main(int argc, char *argv[]) {
   try {
      if (argc!=2)
         error(argv[0],
               ": unexpected number of arguments.\n");
      int length = std::strlen(argv[1]);
      std::vector<string> anagrams;
```

```
        // First: collect all permutations
        using std::next_permutation;
        do {
           anagrams.push_back(string(argv[1]));
        } while (next_permutation(argv[1], argv[1]+length))
        anagrams.sort();
        std::vector<std::string>::iterator last_anagram =
           std::unique(anagrams.begin(), anagrams.end());
        std::ostream_iterator<std::string, char>
           output(std::cout, " ")
        std::copy(anagrams.begin(), last_anagram, output);
     } catch {std::bad_alloc const&) {
        error(argv[0], ": out of memory.\n");
     }
     return 0;
  }
```

An iterator does not by itself determine an allocation or a deallocation strategy. Consequently, algorithms that work on sequences only determined by a pair of iterators cannot add (or remove) elements to (from) the given sequence. This has important consequences for the behavior of a function like `std::unique`. The duplicates are not removed but placed at the end of the sequence by *swapping* element values in appropriate ways. The `std::unique` algorithm thus returns an iterator marking the end of the cleaned-up sequence and the beginning of the sequence of displaced duplicates. Several other algorithms use a similar approach.

Even relatively short words can have a huge number of anagrams. This approach of storing them all is, therefore, limited by the available amount of memory. Note that the possibility of running out of memory was foreseen—an exception handler for the `std::bad_alloc` exception is in place (§14.4.5).

An alternative anagram-generation method, which works in constant memory, numbers each repeated letter. For the *food* example, this would result in the pairs

```
(f 0) (o 0) (o 1) (d 0)
```

Permutations of these pairs, in which a letter with a higher number comes before the same letter with a lower number, are then skipped. For example, the permutation '(o 1) (f 0) (o 0) (d 0)' would be skipped while '(o 0) (f 0) (o 1) (d 0)' would be a valid anagram.

Supplementary Exercise

(A worthwhile refinement) Implement the above scheme as a bidirectional iterator type.

Exercise 18.19

There is no `sort()` for bidirectional iterators. The conjecture is that copying to a vector and then sorting is faster than sorting a sequence using bidirectional iterators. Implement a general sort for bidirectional iterators and test the conjecture.

Hint: Try Exercise 18.5 first.

We implemented a general sort template for bidirectional iterators in Exercise 18.5. Here is a little program that compares the performance of the two approaches: the solution of Exercise 18.5 and copying the whole list into a vector for sorting.

```
#include <algorithm>
#include <ctime>
#include <fstream>
#include <function>
#include <iostream>
#include <list>
#include <string>
#include <vector>

void error(char const *name, char const *msg) {
   std::cerr << name << msg << '\n';
   std::exit(1);
}

// The following is copied from Exercise 18.5:
template<class T, class Less, class Cursor>
void quicksort(Less &fcn, Cursor left, Cursor right) {
   // Zero- or one-element lists are trivially sorted:
   Cursor probe = left;
   if (probe==right or ++probe==right)
      return;

   T pivot = *left;
   // Partition the list into elements less-than and not
   // less-than the pivot (less-than is defined by fcn):
   Cursor p = partition(left, right, bind2nd(fcn, pivot));
   if (p==left)
      ++p;
   else
   if (p==right)
      --p;
```

```
    // Recursively sort these partitions:
    quicksort<T>(fcn, left, p);
    quicksort<T>(fcn, p, right);
}

void test_quicksort(char *filename) {
    std::ifstream in(filename);
    std::list<std::string> words;
    std::string word;
    while (in >> word)
        words.push_back(word);
    std::clock_t start = std::clock();
    quicksort(std::less<std::string>(),
              words.begin(), words.end());
    std::cerr << "QuickSort: "
              << (std::clock()-start)/CLOCKS_PER_SEC << '\n';
}

void test_copysort(char *filename) {
    using std::string;
    std::ifstream in(filename);
    std::list<string> words;
    string word;
    while (in >> word)
        words.push_back(word);
    std::clock_t start = std::clock();
    std::vector<string> tmp(words.size(), string());
    std::copy(words.begin(), words.end(), tmp.begin());
    std::sort(tmp.begin(), tmp.end(), std::less<string>());
    std::copy(tmp.begin(), tmp.end(), words.begin());
    std::cerr << "Copy+Sort: "
              << (std::clock()-start)/CLOCKS_PER_SEC << '\n';
}

int main(int argc, char *argv[]) {
    if (argc!=2)
        error(argv[0], ": unexpected number of arguments.\n");
    test_quicksort(argv[1]);
    test_copysort(argv[1]);
    return 0;
}
```

I applied this program to an early draft of the C++ standard of about 185,000 words, producing the following output:

```
QuickSort: 302
Copy+Sort: 11
```

This would seem to support the conjecture of the standard library designers. We could argue that our implementation of the QuickSort algorithm is crude, whereas the `std::sort` and `std::copy` algorithms used to implement the Copy + Sort method are, presumably, highly tuned. However, the discrepancy is so large—almost a factor of 30—that I doubt simple tuning of the implementation can change the final conclusion.

The slower algorithm, nonetheless, has an advantage: It requires less storage to perform its task.

Supplementary Exercise

(Similar difficulty, but another scenario) Replace the faster algorithm with a similar one in which the list of strings is converted into a vector of strings by first saving the list to a file, then rereading that file into a vector. This reduces the main storage requirements of the Copy + Sort method. Is it still faster than the QuickSort approach under these circumstances?

Chapter 19

Iterators and Allocators

Software design is about *simplification*. This simplification is usually achieved by decomposing a problem into smaller subproblems. However, if these subproblems are still tightly dependent on each other, the simplification of the overall problem may be relatively modest. In contrast, major gains are achieved by *decoupling* the elements of a solution as far as possible.

In the C++ standard library, decoupling is achieved through *iterators*. Iterators are conceptually simple. They "point to values." In that sense, they are similar to the built-in C++ pointers, but they are more abstract and need not, in general, have *all* the capabilities of a pointer. Yet despite their simplicity, iterators are the key element that allow algorithms to operate on unknown containers. In addition, defining your own iterators often requires little work while adding new power to all of the standard algorithms.

Allocators decouple the data structure embodied by a container from the storage allocation policy desired for that data structure. Allocators are generally more complex than iterators, but fortunately, it turns out that the standard allocator is sufficient for most uses (§19.4.1). Furthermore, when defining your own container type, it is not necessary to include an allocator interface for that type to be able to work in conjunction with the standard algorithms. Hence, the exercises in this chapter deal with iterators only.

Exercise 19.1

Implement `reverse()` from §18.6.7. Hint: See §19.2.3.

Hint: A good exercise to sharpen your algorithm-writing skills.

In §18.6.7, we learn that the `reverse` algorithm reverses the order of elements in a sequence delimited by bidirectional iterators:

```
template<typename Bi> void reverse(Bi begin, Bi end);
```

Like most algorithms in the standard library, `reverse` works on *values* rather than on the internal node structure that implements the bidirectional sequence. This is a logical consequence of the abstract nature of bidirectional iterators. There are no specifications for their underlying structure and, hence, a generic algorithm cannot exploit any knowledge concerning such a structure. Furthermore, like many other standard algorithms, `reverse` uses the standard `swap` algorithm to move elements around. Here is a sample implementation of `std::swap`:

```
namespace std {
   template<typename T> inline
   void swap(T &a, T &b) {
      T tmp(a);
      a = b;
      b = tmp;
   }
}
```

Reversing a sequence then entails simultaneously iterating forward from the front and backward from the end until the two iterators meet, swapping the elements pointed to in the process:

```
namespace std {
   template<typename Bi> inline
   void reverse(Bi begin, Bi end) {
      while (begin!=end) {
         std::swap(*begin, *end);
         ++begin;
         if (begin==end)
            break;
         --end;
      }
   }
}
```

Clearly, this algorithm requires the capabilities of a bidirectional iterator; one iterator must iterate backwards. Could we reverse a sequence that can be traversed only using forward iterators (I call this a forward sequence)? Yes, but it requires O(n log n) operations as opposed to the O(n) solution shown above. A general algorithm is also somewhat more complicated.

Reversing a forward sequence can be achieved by splitting the sequence in two roughly equal halves and swapping these halves using the `std::swap_ranges` algorithm. If the number of elements in the sequence is odd, the middle element is not part of either of the halves. The two halves are then recursively reversed.

```
template<typename For>
void forward_reverse(For begin1, int len) {
   if (len>1) {
      int half_len = len/2;
      For end1 = begin1; // End of first half
      for (int k = 0; k!=half_len; ++k)
         ++end1;
      For begin2 = end1;
      if (len%2!=0) // Odd number of elements?
         ++begin2;
      std::swap_ranges(begin1, end1, begin2);
      forward_reverse(begin1, half_len);
      forward_reverse(begin2, half_len);
   }
}
```

Supplementary Exercise

(Not specific to C++) Prove that the total complexity of the function `forward_reverse` is O(len log len).

Hint: The complexity of `std::swap_ranges` is linear in the number of elements to swap.

Reversing a forward sequence is most easily specified by providing an iterator pointing to the first element of a sequence and the length of the sequence, as opposed to two iterators delimiting the forward sequence. However, if all that is available are two such iterators, it is relatively straightforward to recover the length. For example:

```
template<typename For>
int length(For begin, For end) {
   int result = 0;
   while (begin!=end) {
      ++begin;
      ++result;
   }
   return result;
}
```

Supplementary Exercise

(A realistic tuning exercise) Tune the implementation of `forward_reverse` to a reasonable extent (the memory consumption should remain small). For example, sequences shorter than 10 elements could be reversed using a helper array that can accommodate 10 elements. Compare the complexity of the `std::reverse` and the tuned `forward_reverse` algorithms by applying them to large arrays of `int`s.

Rather than force the programmer to remember to call the algorithm appropriate for the kind of iterator available, it is possible to automate this selection. One solution combines overloaded functions with *iterator traits* (§19.2.3).

An overloaded helper function allows compile-time selection:

```
template<typename For> inline
void tagged_reverse(For begin, For end,
                    std::forward_iterator_tag) {
   forward_reverse(begin, length(begin, end));
}

template<typename For> inline
void tagged_reverse(For begin, For end,
                    std::bidirectional_iterator_tag) {
   std::reverse(begin, end);
}
```

And all that is left to do is to determine a tag from the traits template:

```
template<typename It> inline
void flex_reverse(It begin, It end) {
   using std::iterator_traits;
   tagged_reverse(begin, end,
                  iterator_traits<It>::iterator_category());
}
```

A little verbose, but nifty, nonetheless, no?

Supplementary Exercise

(Somewhat advanced template usage) Instead of relying on overload resolution to decide between the two algorithms, it is also possible to use the template-specialization mechanism to achieve our purposes. Rewrite `flex_reverse` using such a template-based technique.

Exercise 19.2

Write an output iterator, `Sink`, that doesn't actually write anywhere. When can `Sink` be useful?

Hint: About the simplest possible output iterator, yet a useful tool.

Implementing `Sink` is relatively simple. The most remarkable part of it is that dereferencing the iterator should produce a type that is neutral with respect to assignment. This could be achieved by creating a separate type, but it is just as convenient to use the iterator type itself for this purpose.

```
#include <iterator>
using std::iterator;
using std::output_iterator_tag;

struct Sink
: iterator<output_iterator_tag, void, void, void, void> {
   template<typename T> Sink& operator=(T const&)
      { return *this; }
   Sink& operator*() const { return *this; }
   Sink& operator++() { return *this; } // Pre-increment
   Sink& operator++(int) { return *this; } // Post-increment
};
```

Note how the templated assignment operator ensures that `Sink` can be used with any kind of value type.

This iterator could conceivably be used to shorten the implementation of various algorithms. For example, the standard algorithm `for_each` could be written as follows:

```
namespace std {
   template<typename In, typename F> inline
   F for_each(In begin, In end, F f) {
      std::transform(begin, end, Sink(), f);
      return f;
   }
}
```

More generally, however, `Sink` can be used whenever an algorithm, such as `std::transform`, insists on assigning the result of an operation to an output iterator while only the side effects of that operator are desired.

Exercise 19.3

Implement `reverse_iterator` (§19.2.5).

Hint: A little tedious, but not hard. The result is a full-fledged standard iterator. Requires familiarity with templates.

The `reverse_iterator` template must be instantiated with a type that satisfies at least the requirements of a bidirectional iterator. It transforms its argument type into an iterator that reverses the meaning of "forward" and "backward." For example, a `reverse_iterator` redirects the '`++`' operation to mean '`--`' on its underlying type.

Standard iterators define a number of member types by inheriting the `std::iterator` template (§19.2.2) instantiated with types extracted from the `std::iterator_traits`

template (§19.2.2). The notations required to express all this quickly become unwieldy. Hence, the following shorthands:

```
#define RIT reverse_iterator
#define ITR typename iterator_traits
#define DIFF_T difference_type
```

The following things must appear in '`namespace std { ... }`':

```
template<typename Bi>
struct RIT: iterator<ITR<Bi>::iterator_category,
                     ITR<Bi>::value_type,
                     ITR<Bi>::DIFF_T,
                     ITR<Bi>::pointer,
                     ITR<Bi>::reference> {
   typedef Bi iterator_type;
   typedef ITR<Bi>::DIFF_T DIFF_T;
   typedef ITR<Bi>::pointer pointer;
   typedef ITR<Bi>::reference reference;
```

You might wonder why we repeat the `typedef`s when we inherit them from the `iterator` base class. The answer is that names that are independent of a template parameter, such as `pointer`, are not looked up in a base class that is dependent on a template parameter (§C.13.8.2). The `typedef`s thus ensure that such names are still found in the scope of `reverse_iterator`.

You may note that, for example,

```
typedef ITR<Bi>::pointer pointer;
```

expands to the following after macro-substitution:

```
typedef typename iterator_traits<Bi>::pointer pointer;
```

The requirement to use `typename` in this context is clarified in the second half of Exercise 14.1.

As you might expect, the various constructors just initialize the encapsulated member iterator `current`. A member function `base` is also provided to access that encapsulated member.

```
   RIT() {}
   explicit RIT(Bi x): current(x) {}
   template<typename B2>
      RIT(RIT<B2> const& it): current(it.current) {}
   Bi base() const { return current; }

   reference operator*() const
      { Bi result(current); --result; return *result; }
   pointer operator->() const
      { Bi result(current); --result; return result; }
   reference operator[](DIFF_T n) const
      { return current[-n-1]; }
```

Perhaps counter intuitively, the dereferencing operators do not just dereference the underlying iterator. To see why this would be wrong, consider the reverse of '`*begin()`'. Just saying '`*end()`' won't do because '`end()`' doesn't point to the last element but to *one position past* the last element. Hence, we must move one position backward before dereferencing a reverse iterator.

The incrementation and decrementation, on the other hand, come with no surprises:

```
RIT& operator++() { --current; return *this; }
RIT operator++(int) {
   { RIT result(*this); --current; return result; }
RIT& operator--() { ++current; return *this; }
RIT operator--(int)
   { RIT result(*this); ++current; return result; }
```

Some members, such as `operator[]` above and the following addition and subtraction operators, may be invoked only when the underlying iterator type is really a *random-access iterator.*

```
RIT operator+(DIFF_T n) const { return RIT(current-n); }
RIT& operator+=(DIFF_T n) { current -= n; return *this; }
RIT operator-(DIFF_T n) const { return RIT(current+n); }
RIT& operator-=(DIFF_T n) { current += n; return *this; }
```

Finally, we must define a number of nonmember operators. For convenience, we define them inside the class as friends (§11.5). As is the case with member functions (§10.2.9), defining the body of a friend function inside a class makes the friend function implicitly `inline`.

```
template<class B2> friend
   bool operator==(RIT<B2> const &x, RIT<B2> const &y)
      { return x==y; }
template<class B2> friend
   bool operator!=(RIT<B2> const &x, RIT<B2> const &y)
      { return x!=y; }
template<class B2> friend
   bool operator<(RIT<B2> const &x, RIT<B2> const &y)
      { return x>y; }
template<class B2> friend
   bool operator<=(RIT<B2> const &x, RIT<B2> const &y)
      { return x>=y; }
template<class B2> friend
   bool operator>=(RIT<B2> const &x, RIT<B2> const &y)
      { return x<=y; }
template<class B2> friend
   bool operator>(RIT<B2> const &x, RIT<B2> const &y)
      { return x<y; }
```

```
   template<class B2> friend
      typename RIT<B2>::DIFF_T operator-(RIT<B2> const &x,
                                         RIT<B2> const &y)
         { return y.current-x.current; }
   template <class B2> friend
      RIT<B2> operator+(typename RIT<B2>::DIFF_T n,
                        RIT<B2> const &x)
         { return RIT<B2>(x.current-n); }
protected:
   Bi current;
};
```

Finally, let's not forget to clean up the convenience macros we defined earlier:

```
#undef DIFF_T
#undef ITR
#undef RIT
```

Overall, implementing an *iterator adaptor* like `reverse_iterator` does not require too much effort. Yet the fact that the C++ standard library also takes care of those little things makes it significantly more pleasant to use.

Chapter 20

Strings

The idea of creating a C++ class that replaces the C-style null-terminated byte strings (NTBS) is extremely popular. Most commercial C++ libraries provide at least one such class; some C++ textbooks dedicate a lot of paper to designing and implementing such a class; and many projects decide to write their own string-like class, as well. This is probably a result of the fact that using C-style strings is often tedious and easily leads to silly, but insidious, errors.

Before deciding to write your own string-like class, consider reusing an existing one. Before committing to a nonstandard solution, verify that `std::string` is not adequate for your uses. The standard string facilities have a rich feature set and integrate well with the standard algorithms and the standard support for C++ locales. But best of all, they're standard and should be familiar to new C++ programmers joining the project.

This chapter shows you some of the many ways in which standard strings can be used and extended.

Exercise 20.1

Write a function that takes two strings and returns a string that is the concatenation of the strings with a dot (period) in the middle. For example, given "file" and "write," the function returns "file.write." Do the same exercise with C-style strings using only C facilities, such as `malloc()` and `strlen()`. Compare the two functions. What are reasonable criteria for a comparison?

Hint: Illustrates fundamental differences between null-terminated character arrays and C++ `string`s.

Writing such a function is relatively simple using both the C++ `string` facility and the plain C-style null-terminated character arrays. Here is a the C++ `string`-based solution:

```
#include <string>
using std::string;
```

```
string dotconnect(string const &a, string const &b) {
   return a+'.'+b;
}
```

Although compact, this version may involve more memory management than we care for. Indeed, every use of the '+' operator for `string`s is likely to require the allocation of at least one temporary object. Here is a slightly more involved implementation that may be more efficient:

```
#include <string>
using std::string;

string dotconnect(string const &a, string const &b) {
   string result(a.size()+b.size()+1, '.');
   result.replace(0, a.size(), a);
   result.replace(a.size()+1, b.size(), b);
   return result;
}
```

The latter function starts by constructing a string `result` large enough to hold the final assembly; the selected constructor also fills that string with period characters. It then suffices to replace the leading and trailing substrings of the `result` string by the first and second argument strings, respectively. The only character untouched by these operations remains a period, as required by the exercise.

A C-style version of the above may look like the following:

```
#include <string.h>
#include <stddef.h>
#include <stdlib.h>

char* dotconnect(char const *a, char const *b) {
   size_t alen = strlen(a), blen = strlen(b);
   char *result = (char*)malloc(alen+blen+2);
   memcpy(result, a, alen);
   result[alen] = '.';
   memcpy(result+alen+1, b, blen);
   result[alen+blen+2] = '\0';
   return result;
}
```

Let's briefly compare each of the following aspects of these implementations:

- Ease of implementation
- Ease of use
- Robustness/safety
- Performance

Ease of implementation is really a nonissue in this case; both functions are very simple.

Programmers generally find `string`-based functionality easier to use than the same functionality for C-style `strings`. For example, `strings` spare us in many cases from the burdens of explicit memory management yet provide a flexible means to control memory-management policies through allocators when that is really needed. Furthermore, the standard set of `string` manipulations is richer than the standard set provided for C-style `strings`.

Both standard C++ `strings` and C-style `strings` can lead to errors and can be abused in ways that increase the likelihood of errors. However, if performance is not critical, it is convenient to use the `std::string` functions in a way that greatly reduces the number of silly errors. This is a consequence of the higher level of abstraction presented by the `std::string` interface. The encapsulation of memory management also plays an important role in this respect. So overall, C++ `strings` are probably preferred in terms of robustness.

The performance comparison is more complex, and conclusions may vary from platform to platform and from application to application. In principle, many operations on `std::strings` can be made extremely efficient. However, in practice, you may find that your standard library provider did not tune C++ `strings` for maximum performance because doing so may increase either the memory consumption of the string class, the code size of programs using the standard string facilities, or both. Opportunities for better performance from `std::string` stem, among other things, from the fact that the length of the string is stored explicitly (C-style functions, usually, must traverse the string to determine its length, as in the calls to `strlen` above), and that memory management can easily be specialized. However, C++ `strings` also often use sophisticated schemes to reduce the cost of copying. In doing so, the cost of various other operations may increase.

A simple test revealed that with one popular C++ implementation, the first version of `dotconnect` for C++ `strings` was about twice as slow as the second one. However, the second one was almost four times slower than the version for C-style `strings` (my timings also included the deallocation of the character array returneed by the C version of `dotconnect`). I confirmed this observation for strings of various lengths.

Supplementary Exercise

(A moderately straightforward verification) Devise a fair performance testing strategy for the dot-connect functions. Do your results confirm my observations?

Exercise 20.5

Write a version of `back_inserter` (§19.2.4) that works for `basic_string`.

Hint: A nontrivial technique in the spirit of Exercises 18.2 and 18.10.

As explained in §19.2.4, `back_inserter` is basically a function that takes a container and returns an output iterator (a `back_insert_iterator`). This output iterator appends elements to the container using the `push_back` member of that container.

The `string::push_back` member was added late in the process of standardizing C++. Hence, to make things more interesting, the solution proposed below implements `back_inserter` without using the `push_back` member.

```
#include <string>

template<typename C, typename T, typename A>
struct string_appender;

template<typename C, typename T, typename A> inline
string_appender<C, T, A>
back_inserter(std::basic_string<C, T, A> const &s) {
   return string_appender<C, T, A>(s);
}
```

This requires a `string_appender` class template that might be written as follows:

```
#include <iterator>
using std::iterator;
using std::output_iterator_tag;

template<typename C, typename T, typename A>
struct string_appender
   : iterator<output_iterator_tag, void, void, void, void> {
   typedef std::basic_string<C, T, A> string;
   typedef typename string::value_type value_type;
   explicit string_appender(string const &s): s_(s) {}
   string_appender& operator=(value_type const &ch) {
      s_.append(ch);
      return *this;
   }
   string_appender& operator*() const { return *this; }
   string_appender& operator++() { return *this; }
   string_appender operator++(int) { return *this; }
private:
   std::string s_;
};
```

You might wonder why `std::back_insert_iterator` (§19.2.4) was not specialized for type `string` rather than introducing a template with a new name `string_appender`. The answer comes from the rule that C++ standard library templates may be specialized only if the declaration of the specialization depends on a user-defined name with external linkage—`std::string` is not such a user-defined name.

Supplementary Exercise

(A straightforward extension) The `string_appender` output iterator is an output iterator whose output operation appends characters to a given `basic_string`. Extend this so that the resulting output iterator can also be used to append `basic_strings` to the target `basic_string`.

Exercise 20.15

Imagine that reading medium-long strings (most are 5 to 25 characters long) from `cin` is the bottleneck in your system. Write an input function that reads such strings as fast as you can think of. You can choose the interface to that function to optimize for speed rather than for convenience. Compare the result to your implementation's `>>` for `strings`.

Hint: Another realistic (but relatively straightforward) performance tuning exercise.

The C++ `istream` `cin` uses the same input source as the C `FILE*` `stdin`. It has been my experience that operations using the C-style FILE structures are faster on many implementations than using the C++ stream facilities. Therefore, we will study two implementations of a `read_string` function: one using `<iostream>` and one using `<stdio.h>`.

First the `<iostream>`-based version:

```
#include <ctype.h>
#include <iostream>
#include <string>

bool read_string(std::string &s) {
   int const buffer_size = 100;
   char buffer[buffer_size+1];
   int i = 0, c;
```

Skip leading whitespace:

```
   while ((c = std::cin.get()) && (std::cin) && isspace(c));
```

Then load nonwhitespace into `buffer` and construct a `string` from this array:

```
   while ((c = std::cin.get()) && (std::cin) && !isspace(c)
          && i!=buffer_size) {
      buffer[i++] = char(c);
   }
   buffer[i] = '\0';
   s.assign(buffer, i);
```

We expect that most strings read from `cin` will fit in the `buffer` array. If that were not the case, we must read more input and append that to the `string`. Here we simply call `read_string` recursively to read the remainder of the input:

```
    if (cin) {
       std::cin.putback(c);
       if (!isspace(c)) {
          std::string remainder;
          read_string(remainder);
          s.append(remainder);
       }
       return true;
    } else
       return false;
};
```

The version using C-style input is almost identical:

```
#include <ctype.h>
#include <stdio.h>
#include <string>

bool read_string(std::string& s) {
   int const buffer_size = 100;
   char buffer[buffer_size+1];
   int i = 0, c;
   while ((c = getc(stdin))!=EOF && isspace(c));
   while ((c = getc(stdin))!=EOF && !isspace(c)
          && i!=buffer_size) {
      buffer[i++] = char(c);
   }
   buffer[i] = '\0';
   s.assign(buffer, i);
   if (c!=EOF) {
      ungetc(c, stdin);
      if (!isspace(c)) {
         std::string remainder;
         read_string(remainder);
         s.append(remainder);
      }
      return true;
   } else
      return false;
};
```

To test these two functions, we can use the following tiny program:

```
#include <time.h>

int main() {
   std::string word;
   int n = 0;
   bool not_eof;
   clock_t start = clock();
#if defined(SIMPLE)
   while (cin)
      cin >> word;
#else
   do {
      not_eof = read_string(word);
      ++n;
   } while (not_eof);
#endif
   printf("Read %d words in %f seconds.\n",
          n, (clock()-start)/CLOCKS_PER_SEC);
   return 0;
}
```

I gave the three variants of this program,

1. simple "`cin >> word`"-based input (define the `SIMPLE` macro)
2. `<iostream>`-based `read_string`
3. `<stdio.h>`-based `read_string`

an early electronic draft of the C++ standard as input. Here are the respective outputs on a not-so-young workstation:

1. Read 185,225 words in 20.147000 seconds
2. Read 185,225 words in 5.452000 seconds
3. Read 185,225 words in 3.578000 seconds

The effort pays off! The `read_string` algorithm by itself speeds the program up almost fourfold, while switching to C-style input/output brought another 35 percent improvement. Similar observations can be made on many other platforms.

This is a rather unpleasant conclusion if you prefer the typesafe style of `<iostream>`-based input/output. Hopefully, library and compiler technology will improve to close the performance gap between the two styles.

Exercise 20.16

Write a function `itos(int)` that returns a `string` representing its `int` argument.

Hint: Compare this to the almost identical Exercise 6.17.

Here is one possible solution:

```
std::string itos(int value) {
   // Assume no more than 64 bits (about 20 digits):
   int const bufsize = 20;
   assert(sizeof(int)<=8);
   char buffer[bufsize];
   bool negative = value<0;
   if (negative)
      value = -value;
   int endpos = bufsize;
   do {
      buffer[--endpos] = char('0'+value%10);
   } while (value /= 10);
   if (negative)
      buffer[--endpos] = '-';
   return std::string(&buffer[endpos], bufsize-endpos);
}
```

Note how we construct the representation right to left by repeatedly dividing by 10. Various low-level optimizations apply to this algorithm, but the complexity of constructing and copying the `std:string` may dominate the cost of this function.

Supplementary Exercise

(See Exercise 7.16.) Generalize `itos` for representation in any base between 2 and 16 (not just decimal, as above). Adapt the function signature as needed.

Chapter 21

Streams

Every useful program performs at least some output. The C++ standard streams provide a basic set of tools that allow you to write portable programs that deal with a platform's file system, as well as with terminal I/O (through `cin`, `cout`, and `cerr`).

The exercises in this chapter discuss various idioms applicable to C++ streams and also show some more general techniques, such as proxy objects and "resource acquisition is initialization." The latter is particularly important when dealing with system resources, such as file streams and dynamically allocated memory.

Exercise 21.1

Read a file of floating-point numbers, make complex numbers out of pairs of numbers read, and write out the complex numbers.

Hint: A good introductory C++ exercise.

Here is one way of doing this:

```
#include <complex>
#include <fstream>
#include <iostream>
#include <stdlib.h>

void error(char const *prog, char const *entity,
           char const *msg) {
   std::cerr << prog << ": " << entity << msg << '\n';
   exit(1);
}
```

```
int main(int argc, char argv[]) {
   if (argc!=2)
      error(argv[0], "",
            ": unexpected number of arguments.");
   std::ifstream numbers(argv[1]);
   if (!numbers)
      error(argv[0], argv[1],
            ": cannot open file for reading.");
   double re, im;
   while (numbers >> re) {
      if (numbers >> im)
         std::cout << complex<double>(re, im) << '\n';
      else
         std::cout << complex<double>(re, 0.0)
                   << "\nWarning: odd number of values.\n";
   }
   return 0;
}
```

The C++ standard library provides a template `complex` which can be instantiated with the types `float`, `double`, and `long double`. Formatted input and output operations are also provided for this template, but the input operation follows conventions different from those implemented above. See §22.5 for a discussion of the complex arithmetic capabilities offered by the standard library.

Exercise 21.2

Define a type `Name_and_address`. Define << and >> for it. Copy a stream of `Name_and_address` objects.

Hint: A moderately simple combination of class design and operator overloading for I/O.

Let's use a first string for the family name and consolidate all the given names in a second string. For the address part, we will store the street address, the city, and the country in three separate strings. The ZIP code is included in the city string.

```
#include <string>
using std::string;

enum FieldType { kName = 0, kFamilyName = 0, kGivenNames = 1,
                 kStreet = 2, kStreetAddress = 2, kCity = 3,
                 kCityAndZip = 3, kCountry = 4 };
```

```
struct Name_and_address {
   Name_and_address(string const &name,
                    string const &given_names,
                    string const &street,
                    string const &city,
                    string const &country) {
      field_[kName] = name;
      field_[kGivenNames] = given_names;
      field_[kStreetAddress] = street;
      field_[kCityAndZip] = city;
      field_[kCountry] = country;
   }

   void set_field(FieldType f, string const &field_value) {
      field_[f] = field_value;
   }

   string const& field(FieldType f) const {
      return field_[f];
   }
private:
   string field_[5];
};
```

Some programmers feel that making structure/class fields private, only to provide read and write access to them using get and set functions, has no practical advantage over using a plain C-style `struct` with public fields. However, my experience differs. In all the nontrivial projects I participated in, there came a time in the development at which an obvious internal representation that had been selected during initial design turned out to be suboptimal. Those projects that religiously avoided making data members private changed the representation relatively painlessly behind the get and set functions. The projects that had exposed data members, on the other hand, could not afford to trace all the uses of the member throughout the hundreds of thousands of lines of C and C++ code and ended up sticking to the inefficient original design. Of course, this should not lead anyone to the conclusion that every data member needs get and set functions. On the contrary, get and set functions are perhaps the thinnest veil of abstraction available, and one will typically prefer an interface that is closer to the natural operations applicable to a type.

There are several ways of writing out a `Name_and_address` object to a stream. The one proposed here writes each string on a separate line:

```
#include <ostream>
using std::ostream;
ostream& operator<<(ostream &output,
                    Name_and_address const &obj) {
```

```
    for (int k = kName; k!=kCountry+1; ++k)
        output << obj.field(k) << '\n';
    return output;
};
```

A possible alternative would consist of prefixing each field by its length. Such a prefixing approach tends to lead to more efficient input operations but looks awkward if the output were intended to be sent to `std::cout`.

```
#include <istream>
#include <string>
using std::istream;

istream& operator>>(istream &input, Name_and_address &obj) {
    for (int k = kName; k!=kCountry+1; ++k) {
        std::string line;
        std::getline(input, line);
        obj.set_field(k, line);
    }
    return input;
}
```

Once all this machinery is in place, copying a stream of `Name_and_address` objects can be done concisely; most of the remaining work consists of verifying the command-line arguments:

```
#include <iostream>
#include <algorithms>
#include <stdlib.h>

void error(char const *prog, char const *entity,
           char const *msg) {
    std::cerr << prog << ": " << entity << msg << '\n';
    exit(1);
}

int main() {
    if (argc!=3)
        error(argv[0], "",
              ": unexpected number of arguments.");
    std::ifstream input(argv[1]);
    if (!input)
        error(argv[0], argv[1],
              ": cannot open file for reading.");
```

```
    std::ofstream output(argv[2]);
    if (!output)
        error(argv[0], argv[2],
              ": cannot open file for writing.");
    std::copy(istream_iterator<Name_and_address>(input),
              istream_iterator<Name_and_address>(),
              ostream_iterator<Name_and_address>(output));
    return 0;
}
```

Supplementary Exercise

(A similar alternative) Implement input and output routines for the `Name_and_address` structure such that each `string` is prefixed by its length. Handle a raw (binary) format using the `istream::read` and `ostream::write` members. Compare the resulting performance with the scheme presented above.

Exercise 21.12

Implement a class for which `[]` is overloaded to implement random reading of characters from a file.

Hint: `istream::seekg`. (§21.6.2).

We will discuss a minimal class to illustrate how this could be done. Clearly, such a class could be endowed with a much richer set of operations. Basically, we use an `fstream`'s `seekg` member to read from a stream at an arbitrary position:

```
#include <fstream>

struct MappedFile {
    MappedFile(std::ifstream &file): file_(file) {}
    char operator[](std::ifstream::pos_type i) const {
        file_.seekg(i);
        return char(file_.get());
    }
private:
    std::ifstream &file_;
};
```

This little program compares the behavior, but not the performance, of random access through the above class with sequential access through an `fstream`'s `get` member.

```
#include <fstream>
#include <iostream>
#include <stdlib.h>

void error(char const *prog, char const *entity,
           char const *msg) {
   std::cerr << prog << ": " << entity << msg << '\n';
   exit(1);
}

int main(int argc, char argv[]) {
   if (argc!=2)
      error(argv[0], "",
            ": unexpected number of arguments.");
   std::ifstream pass0(argv[1]);
   if (!pass0)
      error(argv[0], argv[1],
            ": cannot open file for reading.");
   std::ifstream::pos_type n = 0;
   int chksum0 = 0;
   while (pass0) {
      chksum0 += char(pass0.get());
      ++n;
   }
   std::cerr << "Reading " << n << " characters.\n";
   std::cerr << chksum0 << '\n';
   int chksum1 = 0;
   std::ifstream pass1(argv[1]);
   if (!pass1)
      error(argv[0], argv[1],
            ": cannot open file for reading.");
   MappedFile accessor(pass1);
   for (size_t k = 0; k!=n; ++k) {
      chksum1 += accessor[k];
   }
   std::cerr << chksum1 << '\n';
   return 0;
}
```

Perhaps surprisingly, I obtained the following output when applying this program to a small text file I had lying around.

```
Reading 1069 characters.
83111
80912
```

The last two values are the sum of all the character values read using both methods. In this case, the difference is attributable to the fact that on this platform, the new-line character is represented in the file by using two bytes. Although, ordinarily, the `fstream` members combine these two characters into a single '\n' when reading, they cannot do so after arbitrarily repositioning the input position using `seekg`. Instead, they treat the combination as two separate characters. (If `seekg` and `seekp` were required to treat the 2-byte combination as a single position, these operations would become unacceptably slow or the system would have to keep an unacceptable amount of bookkeeping information to remember where the various 2-byte combinations appear in a opened file.)

Exercise 21.13

Repeat Exercise 21.12 but make [] useful for both reading and writing. Hint: Make [] return an object of a "descriptor type" for which assignment means "assign through descriptor to file" and implicit conversion to `char` means "read from file through descriptor."

Hint: Try Exercise 21.12 first. Use the proxy techniques presented in §25.7.

This exercise nicely illustrates the idea of a proxy object. Such objects are also closely related to the *handle classes* discussed in §25.7.

```
#include <fstream>
using std::fstream;

struct MappedFile {
   struct CharProxy {
      CharProxy(fstream file, fstream::pos_type pos)
         : file_(file), pos_(pos) {}
      CharProxy& operator=(char c) {
         file_.seekp(pos_);
         file_.put(c);
         return *this;
      }
      operator char() {
         file_.seekg(pos_);
         return char(file_.get());
      }
   private:
      fstream &file_;
      fstream::pos_type pos_;
   };
```

```
    MappedFile(fstream &file): file_(file) {}
    char operator[](fstream::pos_type i) const {
       file_.seekg(i);
       return char(file_.peek());
    }
    CharProxy operator[](fstream::pos_type i) {
       return CharProxy(file_, i);
    }
private:
    fstream &file_;
};
```

Applying the subscript operator to a non-const `MappedFile` object returns a `MappedFile::CharProxy` object that "remembers" which file was mapped and which position was subscripted. If this `CharProxy` object is used in a context that expects a `char` value, the conversion operator will read a `char` value from the file at the position that the proxy object has stored away. If, however, one tries to assign a `char` value to the proxy object, this assignment is executed as a write (`put`) operation into the file.

This technique does not cover every situation. For example, the following is not possible:

```
void make_star(char& ch) {
    ch = '*';
}

void stars(fstream &file) {
   MappedFile map(file);
   for (int k = 0; k!=10; ++k)
      make_star(map[k]);
}
```

In other words, although a proxy type acts a little like a C++ reference, it is not equivalent to a `char&`.

Supplementary Exercise

(A straightforward experiment) Compare the performance of the two `MappedFile` implementations in Exercises 21.12 and 21.13 when reading a file sequentially with a more straightforward loop of calls to `get`.

Finally, it should be noted that many modern operating systems provide efficient memory-mapping services. When available, these services often deliver better performance than using the standard C++ I/O facilities.

Exercise 21.18

Define an output manipulator `based` that takes two arguments—a base and an `int` value—and outputs the integer in the representation specified by the base. For example, `based(2,9)` should print `1001`.

Hint: Manipulators are discussed §21.4.6. See Exercise 20.16 for code that represents integers using an arbitrary base.

User-defined manipulators bear some resemblance to the proxy types illustrated in Exercise 21.13 and are also somewhat related to function objects (§18.4).

In this exercise, the manipulator is a class `based` whose constructor stores away the value and the base in which to represent it. A formatted output operator is then provided for `based` such that the desired effect is implemented:

```
namespace { // unnamed namespace
   char ext_digits[] =
      "0123456789abcdefghijklmnopqrstuvwxyz";
} // end unnamed namespace

struct based {
   based(int b, int v): base_(b), value_(v) {
      if (b<2 or b>sizeof(ext_digits))
         throw std::invalid_argument();
   }
   base() const { return base_; }
   value() const { return value_; }
private:
   int base_, value_;
};

std::ostream& operator<<(std::ostream output, based v) {
   int const bufsize = 130; // Enough for 128 base-2 digits
   char buffer[bufsize];
   int value = v.value(), base = v.base();
   bool negative = value<0;
   if (negative)
      value = -value;
   int endpos = bufsize;
   buffer[--endpos] = '\0'; // C-string termination
   do {
      buffer[--endpos] = ext_digits[value%base];
      assert(endpos>0);
   } while (value /= base);
```

```
    if (negative)
       buffer[--endpos] = '-';
    return output << &buffer[endpos];
}
```

Note the strong resemblance between this `operator<<` for `based` objects and the `itos` function devised in Exercise 20.16.

The `based` manipulator is related to the `std::setbase` manipulator but is more general. It will, however, not output a prefix indicating the base used. For example, executing '`std::cout << based(16, 31)`' will output '`1f`,' not '`0x1f`.' (Remember that the standard prefix can also be turned off with the `showbase` manipulator, §21.4.6.2.)

Exercise 21.24

Modify `readints()` (§21.3.6) to handle all exceptions. Hint: Resource acquisition is initialization.

Hint: A simple, but fundamental, technique for handling exceptions.

Here is the `readints` function of §21.3.6:

```
void readints(vector<int> &s) { // Neither Stroustrup's,
                                // nor my favorite style!
   ios_base::iostate old_state = cin.exceptions();
   cin.exceptions(ios_base::eofbit);
   for (;;)
      try {
         int i;
         cin >> i;
         s.push_back(i);
      } catch (ios_base::eof) {
         // OK: end of file reached
      }
   cin.exceptions(old_state); // Reset exception state
}
```

§21.3.6 already suggests that the reliance on exceptions in this context may not be justified (§14.4.1). Reaching the end of the input is by no means exceptional; it is not even an error.

This exercise is meant to illustrate another shortcoming of the function as written above. Consider what happens if the `push_back` member of vector `s` throws an exception: The function aborts prematurely, and the last statement that was to restore the exception state is not executed! This is undesirable because a call to `readints` may now unpredictably change the state of the input stream. Instead, this state information should be treated as a resource to be restored no matter how the function call terminates. Here is how:

```
using std::ios_base;

struct ModifyIOXState {
   ModifyIOXState(ios_base &stream, ios_base::iostate state)
      : stream_(stream), old_state_(stream.exceptions()) {
      stream.exceptions(state);
   }
   ~ModifyIOXState() { stream_.exceptions(old_state_); }
private:
   ios_base &stream_;
   ios_base::iostate old_state_;
};

void readints(vector<int> &s) { // Still not our favorite
                                // style, but better hand-
                                // ling of exceptions
   ModifyIOXState throw_on_eof(std::cin, ios_base::eofbit);
   for (;;)
      try {
         int i;
         std::cin >> i;
         s.push_back(i);
      } catch (ios_base::eof) {
         // OK: end of file reached
      }
}
```

Remember that the "resource acquisition is initialization" idiom acquires a resource by initializing a specialized local object. In the solution above, this object has type `ModifyIOXState`. Because any path that terminates the function call is guaranteed to call the destructor of the object `throw_on_eof` so created, it is safe and convenient to relinquish the resource (the old state value) that was acquired in this destructor.

The fact that relying on exception handling for the detection of an end-of-file condition is less than ideal should not distract you from the merits of "resource acquisition is initialization." It is a fundamental tool to make the best of exception handling.

[blank page]

Chapter 22

Numerics

Not every application must perform sophisticated numerical computations. However, if yours do, you'll probably want the number crunching to be as fast as possible and require that the code implementing this does not add significantly to the complexity of the underlying mathematics. The ability to overload arithmetic operators takes care of the latter requirement, but achieving high performance with overloaded binary arithmetic operators is not trivial. It is possible however, as illustrated by the second half of Exercise 22.8, which builds on the relatively advanced material of §22.4.7 and assumes you're comfortable with C++ templates (Chapter 13).

Other exercises in this chapter study the use of function objects and seminumerical algorithms to efficiently and concisely express computations over large data sets.

Exercise 22.1

Write a function that behaves like `apply()` (§22.4.3) except that it is a nonmember function and accepts function objects.

Hint: A straightforward algorithm.

The member function `apply` of `valarray` takes a pointer to a function as an argument and "applies" that function on each element of the array. Such an interface is less general and often less efficient than the use of function objects, as in other areas of the C++ standard library (for example, the `std::sort` function template, §18.4, §18.7.1). Writing your own nonmember application function though takes relatively few lines of code:

```
#include <valarray>

template<typename T, typename F>
std::valarray<T> apply(std::valarray<T> const &a, F f) {
   std::valarray<T> result(a);
```

```
    for (std::size_t k = 0; k!=a.size(); ++k)
        result[k] = f(result[k]);
    return result;
}
```

Unfortunately, many implementations will not be able to eliminate all the temporary arrays created when calling this version of `apply`. The following exercise shows an alternative interface that is often more efficient in this respect.

Exercise 22.2

Write a function that behaves like `apply()` except that it is a nonmember function, accepts function objects, and modifies its `valarray` argument.

The advantage of this interface is that the transformation occurs "in place" and, therefore, no temporaries need to be created by `apply` itself. You may still need to create an explicit copy of your own because you wish to preserve the values of the array before the transformation, but this slight inconvenience is often preferable to the inefficiencies of the version in Exercise 22.1.

```
#include <valarray>

template<typename T, typename F>
void apply(std::valarray<T> &a, F f) {
    for (std::size_t k = 0; k!=a.size(); ++k)
        a[k] = f(a[k]);
}
```

I hear you saying, "Doesn't `apply` really do the same thing as the special invocation of `std::transform` that outputs its result onto its input sequence?" Well, *yes* it does, and you could write `apply` in terms of `std::transform` if you had a valid pair of iterators that delimit a `valarray`. What's more, if `a` is a `valarray`, then '`&a[0]`' and '`&a[0]+a.size()`' combine into such a pair (unless the element type overloaded the *address-of* `operator&`). So we could write `apply` as follows:

```
template<typename T, typename F> inline
void apply(std::valarray<T> &a, F f) {
    std::transform(&a[0], &a[0]+a.size(), &a[0], f);
}
```

The iterator type is `T*` and you can see that we can treat `valarray` somewhat like a built-in array when needed.

Exercise 22.4

Rewrite the program from §17.4.1.3 using `accumulate()`.

Hint: See §22.6.1 for a description of `accumulate`.

The program in §17.4.1.3 reads (`string`, `int`) pairs from `cin` into a `map` that treats the `string` as a key for the sum of all integers associated with that `string`. Here is the function that reads and accumulates these values:

```
#include <iostream>
#include <map>
#include <string>

void readitems(std::map<std::string, int> &m) {
   std::string word;
   int val = 0;
   while (std::cin >> word >> val)
      m[word] += val;
}
```

(You can find out more about maps provided by the C++ standard library in Chapter 17.)

The function `main` calls `readitems` and then proceeds to report the accumulated values and the grand total. Here is the adaptation of this main function to compute the grand total using `std::accumulate`:

```
#include <map>
#include <string>

int main() {
   using std::string;
   std::map<string, int> tbl;
   readitems(tbl);
   int total = 0;
   typedef std::map<string, int>::const_iterator const_iter;
   for (const_iter p = tbl.begin(); p!=tbl.end(); ++p)
      std::cout << p->first << '\t' << p->second << '\n';
   std::cout << "----------------\ntotal\t"
             << std::accumulate(tbl.begin(), tbl.end(), 0)
             << '\n';
   return !std::cin;
}
```

Exercise 22.8

Implement a `valarray`-like class and implement + and * for it. Compare its performance to the performance of your C++ implementation's `valarray`. Hint: Include x = 0.5*(x + y) + z among your test cases, and try it with a variety of sizes for the vectors x, y, and z.

Hint: The solution presented in the second half of this discussion is probably the most advanced code in this book. It requires thorough understanding of templates.

We will implement two `valarray`-like classes: one that uses a relatively straightforward approach to operator overloading, and another one that uses a more involved scheme to achieve better performance. In both cases, only a limited amount of functionality is implemented and the element type is chosen as `double` rather than left open as a template parameter. The objective is to show techniques and their impact on performance, not to provide a full-fledged library.

The first class is `SimpleArray`:

```
struct SimpleArray {
   SimpleArray(): size_(0) {}
   SimpleArray(ptrdiff_t s): a_(new double[s]), s_(s) {}
   SimpleArray(SimpleArray const &a)
      : a_(new double[a.s_]), s_(a.s_) {
      copy(a);
   }
   ~SimpleArray() { if (s_!=0) delete[] a_; }
   SimpleArray& operator=(SimpleArray const&a) {
      if (&a!=this)
         copy(a);
      return *this;
   }
   ptrdiff_t size() const { return s_; }
   void size(ptrdiff_t s) {
      assert(s_==0 and s>0);
      s_ = s;
      a_ = new double[s];
   }
   double const& operator[](ptrdiff_t k) const
      { return a_[k]; }
   double& operator[](ptrdiff_t k) { return a_[k]; }
   void copy(SimpleArray const &a) {
      memcpy(a_, a.a_, s_*sizeof(double));
   }
private:
   double *a_;
   ptrdiff_t s_;
};
```

Clearly, this is not the most flexible interface. For example, the array may be resized only once, and only if it has size zero, because it was constructed using the default constructor.

The subscript type was chosen to be `ptrdiff_t` because that is also the subscript type for built-in arrays. Occasionally, choosing another subscript type may lead to a surprising overload resolution outcome.

Here is how you could add a few overloaded operators:

```
SimpleArray operator+(SimpleArray const &a,
                      SimpleArray const &b) {
   SimpleArray result(a.size());
   for (ptrdiff_t k = 0; k!=a.size(); ++k)
      result[k] = a[k]+b[k];
   return result;
}

SimpleArray operator*(SimpleArray const &a,
                      SimpleArray const &b) {
   SimpleArray result(a.size());
   for (ptrdiff_t k = 0; k!=a.size(); ++k)
      result[k] = a[k]*b[k];
   return result;
}

SimpleArray operator*(double a, SimpleArray const &b) {
   SimpleArray result(b.size());
   for (ptrdiff_t k = 0; k!=b.size(); ++k)
      result[k] = a*b[k];
   return result;
}
```

This is sufficient to benchmark the performance of `SimpleArray` on the expression `0.5*(x+y)+z` (where x, y, and z are `SimpleArray` objects of equal sizes). I leave out the actual benchmarking, but §22.4.7 summarizes some ways in which the `SimpleArray` scheme implemented above is suboptimal:

- Every application of the operator creates a potentially large temporary object.
- Every application of the operator requires additional traversals of various arrays.

The need for temporaries could potentially be eliminated—at the expense of notational elegance—by using only *computed assignment* operations (+=, *=, and so forth). Even so, the remaining inefficiency due to the multiple array traversals is often unacceptable in serious scientific computing. In other situations, the exclusive use of computed assignments is not sufficient to avoid the need for additional arrays to hold intermediate results. For example, the expression 'x = 1.2*x+x*y' could be written as

```
SimpleArray tmp = x;
tmp *= y;
x *= 1.2;
x += tmp;
```

However, the same computation could be completed without the potentially large `tmp` array:

```
for (ptrdiff_t k = 0; k!=x.size(); ++k)
   x[k] = 1.2*x[k]+x[k]*y[k];
```

This latter loop is the code that we really want our expression to reduce to. A technique that is transparent enough for good compilers to achieve performance equal to that of the above loop for expressions that fit a small set of given patterns can be found in §22.4.7. Unfortunately, that technique does not scale to more general expressions likely to be encountered in scientific computing applications.

What follows may be beyond what you want to sink your teeth into on a first reading. Among other things, it requires a thorough understanding of C++ templates.

I will outline a technique exploiting the C++ template mechanism that is similar to, but more general than, §22.4.7. The `valarray`-like class developed in the discussion below will be called `Vector` and will offer an interface close to that of `SimpleArray`.

As with the technique presented in §22.4.7, the various overloaded operators return objects that encode the operations without actually computing the result. To make these objects template-friendlier, they will all be specializations of a common template `Vop`. Here is how that could work:

```
template<typename Op>
struct Vop {
   Vop(Op const &op): op_(op) {}
   double operator[](ptrdiff_t k) const { return op_[k]; }
   ptrdiff_t size() const { return op_.size(); }
private:
   Op const op_;
};

template<typename Va, typename Vb>
struct Vadd {
   Vadd(Va const &a, Vb const &b): a_(a), b_(b) {}
   double operator[](ptrdiff_t k) const
      { return a_[k]+b_[k]; }
   ptrdiff_t size() const { return a_.size(); }
private:
   Va const &a_;
   Vb const &b_;
};
```

```
inline
Vop<Vadd<Vector, Vector> > // <- return type
operator+(Vector const &a, Vector const &b) { // (1)
   return Vop<Vadd<Vector, Vector> >
             (Vadd<Vector, Vector>(a, b));
}

template<typename Va> inline
Vop<Vadd<Va, Vector> > // <- return type
operator+(Vop<Va> const &a, Vector const &b) { // (2)
   return Vop<Vadd<Va, Vector> >(Vadd<Va, Vector>(a, b));
}

template<typename Vb> inline
Vop<Vadd<Vector, Vb> > // <- return type
operator+(Vector const &a, Vop<Vb> const &b) { // (3)
   return Vop<Vadd<Vector, Vb> >(Vadd<Vector, Vb>(a, b));
}

template<typename Va, typename Vb> inline
Vop<Vadd<Va, Vb> > // <- return type
operator+(Vop<Va> const &a, Vop<Vb> const &b) { // (4)
   return Vop<Vadd<Va, Vb> >(Vadd<Va, Vb>(a, b));
}
```

Let's see what happens with the following expressions:

```
Vector a, b, c;
(a+b)[5];
(a+b+c)[0];
```

Consider '`(a+b)[5]`' first. Because `a` and `b` are both of type `Vector`, version (1) of `operator+` is used to produce a small object of type `Vop<Vadd<Vector, Vector> >`, which holds a `Vadd<Vector, Vector>` object. The subscript operator of this returned object is forwarded to the embedded `Vadd<Vector, Vector>` object, which then computes a sum of two `Vector` elements on the fly. Because all the functions and member functions involved are inline, a good C++ compiler is able to see through the small temporary variables and generate code that is essentially equivalent to '`a[5]+b[5]`'.

The second expression parses as '`((a+b)+c)[0]`' so that the first '+' in '`(a+b+c)[0]`' is treated similarly to the one in '`(a+b)[5]`,' but the second one has argument types `Vop<Vadd<Vector, Vector> >` and `Vector` so that, this time, version (3) of `operator+` is selected and the object returned has type

```
Vop<Vadd<Vadd<Vector, Vector>, Vector> >
```

Applying the subscript operator to such an object again results in the subscript operator being applied to the `Vadd` object that it contains. This time, however, this `Vadd` object itself

contains a `Vadd` object whose subscript operator returns '`a[0]+b[0]`' in our example, as explained above. When the first (or outer) `Vadd` object then computes its '`a_[k]+b_[k]`' expression in the subscript operation, '`a_[k]`' computes '`a[0]+b[0]`,' whereas '`b_[k]`' reduces to '`c[0]`.'

The problem that remains is to assign these *expression templates* to a `Vector` object. With a *member function template*, this can be done in a compact way:

```
struct Vector {
   Vector(): s_(0) {}

   Vector(ptrdiff_t s): a_(new double[s]), s_(s) {}

   Vector(Vector const &a): a_(new double[a.s_]), s_(a.s_) {
      copy(a);
   }

   template<typename Op>
      Vector(Vop<Op> const &expr) {
         s_ = expr.size();
         a_ = new double[s_];
         for (ptrdiff_t k = 0; k!=s_; ++k)
            a_[k] = expr[k];
      }

   ~Vector() { if (s_!=0) delete[] a_; }

   Vector& operator=(Vector const&a)
      { if (this==&a) copy(a); return *this; }

   template<typename Op>
      Vector& operator=(Vop<Op> const &expr) {
         for (ptrdiff_t k = 0; k!=s_; ++k)
            a_[k] = expr[k];
         return *this;
      }

   ptrdiff_t size() const { return s_; }

   void size(ptrdiff_t s) {
      assert(s_==0 and s>0);
      s_ = s;
      a_ = new double[s];
   }
```

```
    double const& operator[](ptrdiff_t k) const
       { return a_[k]; }

    double& operator[](ptrdiff_t k) { return a_[k]; }

    void copy(Vector const &a) {
       memcpy(a_, a.a_, s_*sizeof(double));
    }
private:
    double *a_;
    ptrdiff_t s_;
};
```

Note how the fact that `Vop` and `Vadd` objects provide a `size` member function is essential to allow constructing a `Vector` from an expression template object. For non-`Vector` types, `size()` calls are simply delegated to an embedded object; eventually, the call will be applied to a `Vector`.

Convince yourself that the above four versions of `operator+` cover all the vector-vector additions: a + a, a + (b + c), a + (b + c) + a, and so forth. Adding `operator*` then becomes a matter of rewriting the templates with + substituted by *. The beauty of the scheme is that combinations of * and + become possible without additional templates:

```
template<typename Va, typename Vb>
struct Vmul {
    Vmul(Va const &a, Vb const &b): a_(a), b_(b) {}
    double operator[](ptrdiff_t k) const
       { return a_[k]*b_[k]; }
    ptrdiff_t size() const { return a_.size(); }
private:
    Va const &a_;
    Vb const &b_;
};

inline
Vop<Vmul<Vector, Vector> >
operator*(Vector const &a, Vector const &b) {
    return Vop<Vmul<Vector, Vector> >
              (Vmul<Vector, Vector>(a, b));
}

template<typename Va> inline
Vop<Vmul<Va, Vector> >
operator*(Vop<Va> const &a, Vector const &b) {
    return Vop<Vmul<Va, Vector> >(Vmul<Va, Vector>(a, b));
}
```

```
template<typename Vb> inline
Vop<Vmul<Vector, Vb> >
operator*(Vector const &a, Vop<Vb> const &b) {
   return Vop<Vmul<Vector, Vb> >(Vmul<Vector, Vb>(a, b));
}

template<typename Va, typename Vb> inline
Vop<Vmul<Va, Vb> >
operator*(Vop<Va> const &a, Vop<Vb> const &b) {
   return Vop<Vmul<Va, Vb> >(Vmul<Va, Vb>(a, b));
}
```

One problem is left: mixed vector-scalar operations. There are a variety of options to integrate this in the expression templates framework. The one presented below requires relatively little additional code. To achieve the desired effect, we will "disguise" the scalar as an array using the following class:

```
struct VScalar {
   VScalar(double v, ptrdiff_t s): v_(v), size_(s) {}
   double operator[]() const { return v_; }
   ptrdiff_t size() const { return size_; }
private:
   double v_;
   ptrdiff_t size_;
};
```

With this, we can add the missing operators as follows:

```
inline
Vop<Vmul<VScalar, Vector> >
operator*(double a, Vector const &b) {
   return Vop<Vmul<VScalar, Vector> >
             (Vmul<VScalar, Vector>
                  (VScalar(a, b.size()), b));
}

inline
Vop<Vmul<Vector, VScalar> >
operator*(Vector const &a, double b) {
   return Vop<Vmul<Vector, VScalar> >
             (Vmul<Vector, VScalar>
                  (b, VScalar(b, a.size())));
}
```

```
template<typename Va> inline
Vop<Vmul<Va, VScalar> >
operator*(Vop<Va> const &a, double b) {
   return Vop<Vmul<Va, VScalar> >
             (Vmul<Va, VScalar>(a, VScalar(b, a.size())));
}

template<typename Vb> inline
Vop<Vmul<VScalar, Vb> >
operator*(VScalar a, Vop<Vb> const &b) {
   return Vop<Vmul<VScalar, Vb> >
             (Vmul<VScalar, Vb>(VScalar(a, b.size()), b));
}
```

Vector-scalar additions can be handled similarly.

The original designers of `valarray` were unaware of the preceding technique, and their specifications did not allow such techniques to be applied for the implementation of the `valarray` template. These specifications were later modified slightly so as not to exclude the use of these expression templates, but it remains a tedious task to implement `valarrays` using these techniques. The original idea of incorporating knowledge about `valarray` semantics directly in the compiler seems to have attracted little interest so far. Hence, it comes as no surprise that the `Vector` template outperforms the few commercial `valarray` templates I could find. Indeed, a few compilers manage to optimize the expression:

```
Vector x(1000), y(1000), z(1000);
// ...
x = 0.5*(x+y)+z;
```

to the point where it takes about the same time as:

```
double x[1000], y[1000], z[1000];
// ...
for (ptrdiff_t k = 0; k!=0; ++k)
   x[k] = 0.5*(x[k]+y[k])+z[k];
```

Supplementary Exercise

(More advanced template usage) The implementation of `Vector` requires many versions of each operator to account for the various combinations of `Vector`, `Vop`, and `VScalar` arguments. This could be improved on by merging the concepts of `Vector` and `Vop`—for example, a `Vop<void>` could be specialized to mean what `Vector` currently does. Do this, and rename `Vop` to `Vector`. How much coding does this save?

[blank page]

Chapter 23

Development and Design

This chapter contains no solutions because the corresponding chapter in *The C++ Programming Language, Third Edition* does not propose exercises.

[blank page]

Chapter 24

Design and Programming

This chapter contains no solutions because the corresponding chapter in *The C++ Programming Language, Third Edition* does not propose exercises.

[blank page]

Chapter 25

Roles of Classes

C++ started out as *C with Classes* (§1.4), and throughout more than a decade of development, the class concept has remained central to the power of this language. Whether a class is templatized, polymorphic, abstract, a base class of another class, or any combination of these things, reasonable uses for them are found all the time. This chapter shows a few possible scenarios and many more are to be found throughout this book. Chapters 10 and 15 complement this material in particular.

Note that the class concept can be expressed in C++ using both the keyword `class` and the keyword `struct`. These keywords differ only in the default accessibility of their members and bases; which one you use is purely a matter of taste.

Exercise 25.1

The Io template from §25.4.1 does not work for built-in types. Modify it so that it does.

Hint: An illustration of the ability to explicitly or partially specialize templates (§13.5).

Reading §25.4.1, you will find that the `Io` template is defined as

```
#include <iostream>

template<class T>
class Io: public T, public Io_obj {
public:
   Io(istream&); // initialize from input stream
   Io* clone() const { // overrides Io_obj::clone
      return new Io(*this);
   }
   static Io* new_io(istream &s) { return new Io(s); }
   // ...
};
```

where `Io_obj` is an abstract base class dictating that the `clone` member be provided by its derivations.

This template makes it convenient to add the interface of a simple object I/O interface to a class that did not know about this interface while maintaining the functionality of that class. Clearly, one cannot instantiate this class template for built-in types because it is not possible in C++ to derive from such types.

However, C++'s template mechanism is powerful and one of the facilities it provides to address problems of this kind is explicit specialization (§13.5). Using specialization, you can request that the compiler use a nongeneric implementation of `Io` for specific template arguments rather than attempt the instantiation of the generic version. For example, `Io<int>` could be written as

```
#include <iostream>

template<> // announces an explicit specialization
class Io<int>: public Io_obj {
   Io<int>(int v): value_(v) {}
   Io<int>(istream&);
   Io<int>& operator=(int v) { value_ = v; return *this; }
   operator int() const { return value_; }
   Io<int>* clone() const { return new Io<int>(*this); }
   static Io<int>* new_io(istream &s)
      { return new Io<int>(s); }
   // ...
private:
   int value_;
};
```

Here we are exploiting the ability to provide an implicit conversion operator (§11.4) to make class `Io<int>` behave as an `int`. As usual, one should be cautious when dealing with such implicit conversions.

`Io` is less likely to be instantiated for pointer types because pointers refer to addresses in memory that may not be available when they are loaded from an I/O system (assuming that the program was terminated and restarted since they were stored away). Nevertheless, you may find yourself in a situation in which you would like to specialize a template implementation for every pointer type. Again, C++ allows us to do just that through partial specialization (§13.5):

```
template<typename T>
class Io<T*>: public Io_obj { // Partial spec. for pointers
   Io(T *p): ptr_(p) {}
   Io(istream&);
   Io& operator=(T *p) { ptr_ = p; return *this; }
   T* operator->() const { return ptr_; }
   T& operator*() const { return *ptr_; }
   operator T*() const { return ptr_; }
```

```
    Io* clone() const { return new Io(*this); }
    static Io* new_io(istream &s) { return new Io(s); }
    // ...
private:
    T *ptr_;
};
```

Through this definition, we state that if `Io` is instantiated for a pointer type, this more specialized template ought to be used instead of the generic one (also called the primary template). Note that what distinguishes the declaration of a partial specialization from that of its *primary template* is the fact that the partial specialization names a *template-id*—a name with angle brackets, like '`Io<T*>`.'

Exercise 25.5

Use the `Filter` framework to implement a program that removes adjacent duplicate words from an input stream but otherwise copies the input to output.

Hint: An oversimplified demonstration of frameworks in action.

The `Filter` framework is presented as a minimal example of a framework in §25.8. It consists of an abstract interface definition:

```
#include <iostream>

struct Filter {
    struct Retry {
        virtual char const* message() { return 0; }
    };
    virtual ~Filter() {}
    virtual void start() {}
    virtual int read() = 0;
    virtual void write() {}
    virtual void compute() {}
    virtual int result() = 0;
    virtual int retry(Retry &m) {
        std::cerr << m.message() << '\n';
        return 2;
    }
};
```

and a driving function:

```
int main_loop(Filter *p) {
   for (;;) {
      try {
         p->start();
         while (p->read()) {
            p->compute();
            p->write();
         }
         return p->result();
      }
      catch(Filter::Retry &m) {
         if (int i = p->retry(m))
            return i;
      }
      catch (...) {
         std::cerr << "Fatal filter error.\n";
         return 1;
      }
   }
}
```

The user of the framework "plugs in" his or her own components by deriving them from the `Filter` class and passing them to the `main_loop` function. In our case, we must create a component that will eliminate duplicate consecutive words:

```
struct UniqueWords: Filter {
   UniqueWords(std::istream &in, std::ostream &out)
      : in_(in), out_(out), n_eliminated_(0) {}
   virtual int read()
      { in_ >> next_word_; return in_.good(); }
   virtual void compute() {
      if (last_word_==next_word_) {
         next_word_ = ""; // Kill next word
         ++n_eliminated_;
      } else {
         last_word_ = next_word_;
   }
   virtual void write() { out_ << next_word_; }
   virtual int result() { return n_eliminated_; }
private:
   std::istream &in_;
   std::ostream &out_;
   std::string last_word_, next_word_;
   int n_eliminated_;
};
```

The `main` function can be as simple as

```
int main() {
   UniqueWords uniq(cin, cout);
   main_loop(&uniq);
   return 0;
}
```

Exercise 25.7

Write a `Range` template that takes both the range and the element type as template parameters.

Hint: Try this to familiarize yourself with non-type template arguments.

The `Range` template of §25.6.1 is an example of a *restricted interface*. `Range` objects behave like an `int`, but check that their value remains between the bounds defined by the template's arguments. This can be generalized to scalar types other than `int` as follows:

```
template<typename T, T low, T high>
struct Range {
   struct Error {};
   Range(T value): value_(value) {
      if (value>=high || value<low)
         throw Error();
   }
   Range& operator=(T value) { return *this = Range(value); }
   operator T() { return value_; }
private:
   T value_;
};
```

Choosing to define the boundaries of the allowable range through non-type template parameters imposes a number of limitations on the type `T`. It must be one of the following:

- An integral or enumeration type initialized with an integral *constant-expression*
- A reference type initialized with an lvalue designating a function or object with external linkage
- A pointer type initialized with the address of a function or named object with external linkage
- A pointer-to-member type initialized with the name of a member of a class

That means that the following:

```
Range<double, -1.0, 1.0> val(0.0);
```

is not a valid instantiation of the `Range` template because non-type template parameters cannot have floating type. On the other hand, the slightly more wordy

```
extern double const low = -1.0, high = 1.0;
Range<double&, low, high> val(0.0);
```

should work just fine since the non-type parameters now have a reference type with external linkage.

Exercise 25.19

Write a class for composing operations represented as function objects. Given two function objects, `f` and `g`, `Compose(f, g)` should make an object that can be invoked with an argument `x` suitable for `g` and returns `f(g(x))`, provided the return value of `g()` is an acceptable argument type for `f()`.

Hint: Another exercise illustrating the combination of function template argument deduction and function object templates (see Exercises 18.2 and 18.10).

The C++ standard library provides a variety of function-object class templates, including some—like binders and adapters—that can be combined with other function objects to further increase the variety of readily available function object types. Function objects are described in more detail in §18.4.

Consider the following function-object class template that implements composition:

```
template<typename F, typename G>
struct FunctionComposition {
   typedef typename F::result_type result_type;
   typedef typename G::argument_type argument_type;
   FunctionComposition(F const &f, G const &g)
      : f_(f), g_(g) {}
   result_type operator(argument_type x)
      { return f_(g_(x)); }
private:
   F f_;
   G g_;
};
```

This sort of class is often more conveniently usable when the template arguments are *deduced* (§C.13.4). This calls for a function template:

```
template<typename F, typename G>
FunctionComposition<F, G> Compose(F const &f, G const &g) {
   return FunctionComposition<F, G>(f, g);
}
```

Compare this to the `std::make_pair` template presented in §17.4.1.2.

Note that we rely on the fact that the function objects have member types `result_type` and `argument_type`, as is the case for the standard `<function>` types. If you have a function object type that you need to use, but which does not define these types, you can introduce a sort of *interface class* (§25.6). For example, say you bought the following magical numerical function class that you would like to use with the above `FunctionComposition` template:

```
struct Incantation {
   double operator()(double);
private:
   double elixir_, wand_;
};
```

For that to be possible, the types `argument_type` and `return_type` must be added to `Incantation`. Here is one way of achieving this:

```
struct Spell: Incantation {
   typedef double result_type;
   typedef double argument_type;
};
```

[blank page]

Index

D

E

F

G

H

I

N

Q

R

S

T

U

V

W

X

Y

Z