

Notes for Data Structures and Algorithms in C++

CHAPTER 1: A C++ Primer

1.1. Basic C++ Programming Elements

C++ is a powerful and flexible programming language, which was designed to build upon the constructs of the C programming language. Thus, with minor exceptions, C++ is a superset of the C programming language. C++ shares C's ability to deal efficiently with hardware at the level of bits, bytes, words, addresses, etc. In addition, C++ adds several enhancements over C (which motivates the name "C++"), with the principal enhancement being the object-oriented concept of a **class**.

A class is a user-defined type that encapsulates many **important** mechanisms such as guaranteed initialization, implicit type conversion, control of memory management, operator overloading, and polymorphism.

● A Simple C++ Program

Like many programming languages, creating and running a C++ program requires several steps. First, we create a C++ source file into which we enter the lines of our program. After we save this file, we then run a program, called a **compiler**, which creates a machine-code interpretation of this program. Another program, called a **linker** (which is typically invoked automatically by the compiler), includes any required library code functions needed and produces the final machine-executable file. In order to run our program, the user requests that the system execute this file.

Let us consider a very simple program to illustrate some of the language's basic elements. (The line numbers are not part of the program; they are just for our reference.)

```
1 #include <cstdlib>
2 #include <iostream>
3 /* This program inputs two numbers x and y and outputs their sum */
4 int main( ) {
5     int x, y;
6     std::cout << "Please enter two numbers: ";
7     std::cin >> x >> y;           // input x and y
8     int sum = x + y;             // compute their sum
9     std::cout << "Their sum is " << sum << std::endl;
10    return EXIT_SUCCESS;        // terminate successfully
11 }
```

Program Elements

Let us consider the elements of the above program in greater detail. Lines 1 and 2 input the two **header files**, "cstdlib" and "iostream." Header files are used to **provide special declarations and definitions**, which are of use to the program. The **first** provides some standard system definitions, and the **second** provides definitions needed for input and output.

The **function body** is given within curly braces (`{...}`), which start on line 4 and end on line 11. The program terminates when the return statement on line 10 is executed.

By **convention**, the function `main` returns the value zero to indicate success and returns a nonzero value to indicate failure. The include file "cstdlib" defines the constant **EXIT_SUCCESS** to be 0.

The statement on line 6 prints a string using the **output** operator ("`<<`"). The statement on line 7 inputs the values of the variables `x` and `y` using the **input** operator ("`>>`"). The name **std::cout** indicates that output is to be sent to the **standard output stream**. There are two other important I/O streams in C++: **standard input** is where input is typically read, and **standard error** is where

error output is written. These are denoted **std::cin** and **std::cerr**, respectively.

The prefix "std::" indicates that these objects are from the system's **standard library**. We should include this prefix when referring to objects from the standard library. **Nonetheless, it is possible** to inform the compiler that we wish to use objects from the standard library—and so omit this prefix—by utilizing the "**using**" statement as shown below.

```
#include <iostream>
using namespace std; // makes std:: available
// ...
cout << "Please enter two numbers: "; // (std:: is not needed)
cin >> x >> y;
```

Returning to our simple example C++ program, we note that the statement on line 9 outputs the value of the variable sum, which in this case stores the computed sum of x and y. By default, the output statement does not produce an end of line. The special object **std::endl** generates a special end-of-line character. **Another way** to generate an end of line is to output the newline character, '\n'.

● Fundamental Types

bool	Boolean value, either true or false
char	character
short	short integer
int	integer
long	long integer
float	single-precision floating-point number
double	double-precision floating-point number

There is also an enumeration, or **enum**, type to represent a set of discrete values. Together, enumerations and the types **bool**, **char**, and **int** are called **integral types**. Finally, there is a special type **void**, which *explicitly indicates the absence of any type information*.

Characters

A **char** variable holds a single character. A **char** in C++ is typically 8-bits, but the exact number of bits used for a **char** variable is dependent on the particular implementation.

A **literal** is a **constant value** appearing in a program. **Character literals** are enclosed in **single quotes**, as in 'a', 'Q', and '+'. A backslash (\) is used to specify a number of **special character literals** as shown below.

'\n'	newline	'\t'	tab
'\b'	backspace	'\0'	null
'\'	single quote	'\"'	double quote
'\\'	backslash		

The **null** character, '\0', is sometimes used to **indicate the end of a string of characters**. Every character is associated with an integer code. The function **int(ch)** **returns the integer value associated with a character variable ch**.

Integers

An **int** variable holds an integer. Integers come in three sizes: **short int**, (plain) **int**, and **long int**. The terms "**short**" and "**long**" are synonyms for "**short int**" and "**long int**," respectively. The **suffix** "L" or "L" can be added to indicate a long integer, as in 123456789L. **Octal** (base 8) **constants** are specified by prefixing the number with the zero digit, and **hexadecimal** (base 16) **constants** can be specified by prefixing the number with "0x." For example, the literals 256, 0400, and 0x100 all represent the integer value 256 (in decimal).

When declaring a variable, we have the option of providing a definition, or initial value. If no definition is given, the initial value is unpredictable, so it is important that each variable be assigned a value before being used. **Variable names** may consist of any combination of letters,

digits, or the underscore (_) character, **but** the first character cannot be a digit. Here are some examples of declarations of integral variables.

```
short n;           // n's value is undefined
int octalNumber = 0400; // 400 (base 8) = 256 (base 10)
char newline_character = '\n';
long BIGnumber = 314159265L;
short _aSTRANGE__1234_variABIE_NaMe;
```

Although it is legal to start a variable name with an underscore, it is **best to avoid this practice**, since some C++ compilers use this convention for **defining their own internal identifiers**.

C++ does **not** specify the exact number of bits in each type, but a **short** is at least 16 bits, and a **long** is at least 32 bits. In fact, there is no requirement that **long** be strictly longer than **short** (but it cannot be shorter!). Given a type T, the expression `sizeof(T)` returns the size of type T, *expressed as some number of multiples of the size of char*.

Enumerations

An enumeration is a user-defined type that can hold any of a set of discrete values. Once defined, enumerations behave much like an integer type. A common use of enumerations is to provide **meaningful names** to a set of related values. Each element of an enumeration is associated with *an integer value*. By default, these values count up from 0, but it is also possible to define explicit constant values as shown below.

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
enum Mood { HAPPY = 3, SAD = 1, ANXIOUS = 4, SLEEPY = 2 };

Day today = THU;           // today may be any of MON ... SAT
Mood myMood = SLEEPY;     // myMood may be HAPPY, ..., SLEEPY
```

As a **hint** to the reader, we write *enumeration names and other constants* with **all capital letters**.

Floating Point

A variable of type **float** holds a single-precision floating-point number, and a variable of type **double** holds a double-precision floating-point number. **By default**, floating point literals, such as 3.14159 and -1234.567 are of type **double**. Scientific or exponential notation may be specified using either "e" or "E" to separate the mantissa from the exponent, as in 3.14E5, which means 3.14×10^5 . **To force a literal to be a float**, add the suffix "f" or "F," as in 2.0f or 1.234e - 3F.

● Pointers, Arrays, and Structures

We next discuss how to combine fundamental types to form more complex ones.

Pointers

Each program variable is stored in the computer's memory at some location, or *address*. **A pointer** is a variable that holds the value of such an address. Given a type T, the type **T*** denotes a pointer to a variable of type T. For example, **int*** denotes a pointer to an integer.

Two essential operators are used to manipulate pointers. The first returns the address of an object in memory, and the second returns the contents of a given address. In C++ the first task is performed by the *address-of operator*, **&**. Accessing an object's value from its address is called *dereferencing*. This is done using the ***** operator.

```
char ch = 'Q';
char* p = &ch;           // p holds the address of ch
cout << *p;             // outputs the character 'Q'
ch = 'Z';                // ch now holds 'Z'
cout << *p;             // outputs the character 'Z'
*p = 'X';                // ch now holds 'X'
cout << ch;              // outputs the character 'X'
```

Pointers need not point only to *fundamental types*, such as **char** and **int**—they may also point to *complex types and even to functions*.

It is useful to have a pointer value that points to nothing, that is, a **null pointer**. By convention, such a pointer is assigned the value zero. An attempt to dereference a null pointer results in a run-time error. All C++ implementations define a special symbol **NULL**, which is equal to zero. This definition is activated by inserting the statement "#include <cstdlib>" in the beginning of a program file.

We mentioned earlier that the special type **void** is used to indicate no type information at all. Although we cannot declare a variable to be of type **void**, we can declare a pointer to be of type **void***. Such a pointer can point to a variable of *any type*. Since the compiler is unable to check the correctness of such references, the use of **void*** pointers is *strongly discouraged*, except in *unusual cases where direct access to the computer's memory is needed*.

Caution: Beware when declaring two or more pointers on the same line. The * operator binds with the *variable name, not with the type name*. Consider the following misleading declaration.

```
int* x, y, z; // same as: int* x; int y; int z;
```

This declares one pointer variable *x*, but the other two variables are plain integers.

Arrays

An *array* is a collection of elements of the same type. Given any type *T* and a constant *N*, a variable of type *T[N]* holds an array of *N* elements, each of type *T*. Each element of the array is referenced by its *index*, that is, a number from 0 to *N* – 1.

```
double f[5]; // array of 5 doubles: f[0], ..., f[4]
int m[10]; // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]]; // outputs f[4], which is 2.5
```

A *two-dimensional array* is implemented as an "array of arrays." For example "int A[15][30]" declares *A* to be an array of 30 objects, each of which is an array of 15 integers. An element in such an array is indexed as *A[i][j]*, where *i* is in the range 0 to 14 and *j* is in the range 0 to 29.

When declaring an array, we can initialize its values by enclosing the elements in curly braces ({...}). When doing so, we do not have to specify the size of the array, since the compiler can figure this out.

```
int a[] = {10, 11, 12, 13}; // declares and initializes a[4]
bool b[] = {false, true}; // declares and initializes b[2]
char c[] = {'c', 'a', 't'}; // declares and initializes c[3]
```

Just as it is possible to declare an array of integers, it is possible to declare an array of pointers to integers. For example, *int* r[17]*.

Pointers and Arrays

There is an interesting connection between arrays and pointers, which C++ inherited from the C programming language—the name of an array is equivalent to a pointer to the array's initial element and vice versa.

```
char c[] = {'c', 'a', 't'};
char* p = c; // p points to c[0]
char* q = &c[0]; // q also points to c[0]
cout << c[2] << p[2] << q[2]; // outputs "ttt"
```

Caution: This equivalence between array names and pointers can be confusing.

Strings

A string literal, such as "Hello World", is represented as a fixed-length array of characters that ends with the **null character**. Character strings represented in this way are called **C-style strings**, since they were inherited from C.

C++ provides a **string type** as part of its Standard Template Library (**STL**). When we need to distinguish, we call these **STL strings**. In order to use STL strings it is necessary to include the header file `<string>`. Since STL strings are part of the standard namespace (see Section 1.1.4), their full name is `std::string`. STL strings may be concatenated using the **+ operator**, they may be compared with each other using lexicographic (or dictionary) order, and they may be input and output using the `>>` and `<<` operators, respectively. For example:

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;           // t = "not to be"
string u = s + " or " + t;      // u = "to be or not to be"
if (s > t)                     // true: "to be" > "not to be"
    cout << u;                 // outputs "to be or not to be"
string s = "John";             // s = "John"
int i = s.size();              // i = 4
char c = s[3];                // c = 'n'
s += " Smith";                // now s = "John Smith"
```

C-Style Structures

A **structure** is useful for storing an aggregation of elements. Unlike an array, the elements of a structure may be of different types. Each **member**, or **field**, of a structure is referred to by a given name.

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string     name;           // passenger name
    MealType   mealPref;       // meal preference
    bool       isFreqFlyer;    // in the frequent flyer program?
    string     freqFlyerNo;    // the passenger's freq. flyer number
};

Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

The individual members of the structure are accessed using the **member selection operator**, which has the form `struct.name.member`.

```
pass.name = "Pocahontas";      // change name
pass.mealPref = REGULAR;        // change meal preference
```

Structures of the same type may be assigned to one another. For example, if `p1` and `p2` are of type `Passenger`, then `p2=p1` copies the elements of `p1` to `p2`.

Pointers, Dynamic Memory, and the "new" Operator

We often find it useful in data structures to create objects dynamically as the need arises. The C++ run-time system reserves a large block of memory called the **free store**, for this reason. (This memory is also sometimes called **heap memory**, but this should not be confused with the heap data structure, which is discussed in Chapter 8.) The operator **new** dynamically allocates the correct amount of storage for an object of a given type from the free store and returns a pointer to this object.

Because **complex objects like structures are often allocated dynamically**, C++ provides a shorter way to access members using the `->` operator.

`pointer_name->member` is equivalent to `(*pointer_name).member`

For example, we could allocate a new passenger object and initialize its members as follows.

```

Passenger *p;
// ...
p = new Passenger;
p->name = "Pocahontas";
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";

```

This new passenger object continues to exist in the free store until it is explicitly deleted—a process that is done using the **delete** operator, which destroys the object and returns its space to the free store.

```

delete p; // destroy the object p points to

```

The **delete** operator should only be applied to objects that have been allocated through **new**. **Arrays can also be allocated with new.** A dynamically allocated array with elements of type T would be declared being of type ***T**. **Arrays allocated in this manner cannot be deallocated** using the standard **delete** operator. Instead, the **operator delete[]** is used.

```

char* buffer = new char[500]; // allocate a buffer of 500 chars
buffer[3] = 'a'; // elements are still accessed using []
delete [] buffer; // delete the buffer

```

Memory Leaks

Failure to delete dynamically allocated objects can cause problems. If we were to change the (address) value of p without first deleting the structure to which it points, there would be no way for us to access this object. It would continue to exist for the lifetime of the program, using up space that could otherwise be used for other allocated objects. Having such inaccessible objects in dynamic memory is called a **memory leak**. An important rule for a disciplined C++ programmer is the following:

If an object is allocated with **new**, it should eventually be deallocated with **delete**.

References

Pointers provide one way to refer indirectly to an object. Another way is through references. A **reference** is simply an alternative name for an object. Given a type T, the notation T& indicates a reference to an object of type T. Unlike pointers, which can be NULL, a reference in C++ **must** refer to an actual variable. **When a reference is declared, its value must be initialized.** Afterwards, any access to the reference is treated exactly as if it is an access to the underlying object.

```

string author = "Samuel Clemens";
string& penName = author; // penName is an alias for author
penName = "Mark Twain"; // now author = "Mark Twain"
cout << author; // outputs "Mark Twain"

```

References are **most often used for passing function arguments** and are also often used for **returning results from functions**.

- Named Constants, Scope, and Namespaces

We can easily name variables without concern for naming conflicts in small problems. It is much **harder for us to avoid conflicts in large software systems**, which may consist of hundreds of files written by many different programmers.

Constants and Typedef

Good programmers commonly like to associate names with constant quantities. By adding the keyword **const** to a declaration, we indicate that the value of the associated object cannot be changed. As a **hint** to the reader, we will use all capital letters when naming constants.

```

const double PI      = 3.14159265;
const int   CUT_OFF[] = {90, 80, 70, 60};
const int   N_DAYS    = 7;
const int   N_HOURS   = 24*N_DAYS; // using a constant expression
int counter[N_HOURS];           // an array of 168 ints

```

Note that **enumerations** (see Section 1.1.2) provide another convenient way to define integer-valued constants, especially **within structures and classes**.

In addition to associating names with constants, it is often **useful to associate a name with a type**. This association can be done with a **typedef** declaration. Rather than declaring a variable, a **typedef** defines a new type name.

```

typedef char* BufferPtr;           // type BufferPtr is a pointer to char
typedef double Coordinate;        // type Coordinate is a double

BufferPtr p;                      // p is a pointer to char
Coordinate x, y;                 // x and y are of type double

```

We will follow the **convention** of indicating user-defined types by **capitalizing the first character of their names**.

Local and Global Scopes

When a group of C++ statements are enclosed in curly braces (`{...}`), they define a **block**. Variables and types that are declared within a block are **only accessible from within the block**. **They are said to be local to the block**. Blocks can be nested within other blocks. In C++, a variable may be declared outside of any block. **Such a variable is global**, in the sense that it is accessible from everywhere in the program. *The portions of a program from which a given name is accessible* are called its **scope**.

A local variable "hides" any global variables of the same name as shown in the following example.

```

const int Cat = 1;                // global Cat

int main() {
    const int Cat = 2;            // this Cat is local to main
    cout << Cat;                // outputs 2 (local Cat)
    return EXIT_SUCCESS;
}

int dog = Cat;                   // dog = 1 (from the global Cat)

```

Namespaces

Global variables present many problems in large software systems because they can be accessed and possibly modified anywhere in the program. As a result, **it is best to avoid global variables**.

A **namespace** is a mechanism that allows a group of related names to be defined in one place. This helps organize global objects into natural groups and minimizes the problems of globals.

```

namespace myglobals {
    int cat;
    string dog = "bow wow";
}

```

Namespaces may generally contain definitions of more complex objects, including types, classes, and functions. We can access an object `x` in namespace group, using the **notation group::x**, which is called its **fully qualified name**.

The Using Statement

If we are repeatedly using variables from the same namespace, it is possible to avoid entering

namespace specifiers by telling the system that we want to "use" a particular specifier. We communicate this desire by utilizing the **using** statement, which *makes some or all of the names from the namespace accessible, without explicitly providing the specifier*. This statement has **two forms that allow us to list individual names or to make every name in the namespace accessible** as shown below.

```
using std::string;           // makes just std::string accessible
using std::cout;             // makes just std::cout accessible
using namespace myglobals;   // makes all of myglobals accessible
```

1.2. Expressions

An **expression** combines *variables and literals with operators to create new values*. Throughout, we use **var** to denote a variable or anything to which a value may be assigned. We use **exp** to denote an expression and **type** to denote a type.

Member Selection and Indexing

class_name . member	class/structure member selection
pointer -> member	class/structure member selection
array [exp]	array subscripting

Arithmetic Operators

The following are the **binary** arithmetic operators:

exp + exp	addition
exp - exp	subtraction
exp * exp	multiplication
exp / exp	division
exp % exp	modulo (remainder)

There are also **unary** minus ($-x$) and unary plus ($+x$) operations. Division between two integer operands results in an integer result by truncation, even if the result is being assigned to a floating-point variable. The modulo operator $n \% m$ yields the remainder that would result from the integer division n / m .

Increment and Decrement Operators

The **post-increment** operator returns a variable's value and then increments it by 1. The post-decrement operator is analogous but decreases the value by 1. The **pre-increment** operator first increments the variables and then returns the value.

var ++	post increment
var --	post decrement
++ var	pre increment
-- var	pre decrement

```
int a[] = {0, 1, 2, 3};
int i = 2;
int j = i++;
int k = --i;
cout << a[k++];
```

// j = 2 and now i = 3
// now i = 2 and k = 2
// a[2] (= 2) is output; now k = 3

Relational and Logical Operators

C++ provides the usual comparison operators.

exp < exp	less than
exp > exp	greater than
exp <= exp	less than or equal
exp >= exp	greater than or equal
exp == exp	equal to
exp != exp	not equal to

These return a Boolean result—either **true** or **false**.

The following logical operators are also provided.

<code>! exp</code>	logical not
<code>exp && exp</code>	logical and
<code>exp exp</code>	logical or

The operators `&&` and `||` evaluate sequentially from left to right. If the left operand of `&&` is false, the entire result is false, and the right operand is not evaluated. The `||` operator is analogous, but evaluation stops if the left operand is true.

Bitwise Operators

The following operators act on the representations of numbers as binary bit strings. They can be applied to any integer type, and the result is an integer type.

<code>~ exp</code>	bitwise complement
<code>exp & exp</code>	bitwise and
<code>exp ^ exp</code>	bitwise exclusive-or
<code>exp exp</code>	bitwise or
<code>exp1 << exp2</code>	shift <code>exp1</code> left by <code>exp2</code> bits
<code>exp1 >> exp2</code>	shift <code>exp1</code> right by <code>exp2</code> bits

The left shift operator always fills with zeros. How the right shift fills depends on a variable's type.

Assignment Operators

In addition to the familiar assignment operator (`=`), C++ includes a special form for each of the arithmetic binary operators (`+`, `-`, `*`, `/`, `%`) and each of the bitwise binary operators (`&`, `|`, `^`, `<<`, `>>`), that combines a binary operation with assignment.

```
int    i = 10;
int    j = 5;
string s = "yes";
i    -= 4;           // i = i - 4 = 6
j    *= -2;          // j = j * (-2) = -10
s    += " or no";    // s = s + " or no" = "yes or no"
```

Other Operators

Here are some other useful operators.

<code>class_name :: member</code>	class scope resolution
<code>namespace_name :: member</code>	namespace resolution
<code>bool_exp ? true_exp : false_exp</code>	conditional expression
<code>smaller = (x < y ? x : y);</code>	// smaller = min(x,y)

We also have the following operations on input/output streams.

<code>stream >> var</code>	stream input
<code>stream << exp</code>	stream output

Although they look like the bitwise shift operators, the input (`>>`) and output (`<<`) stream operators are quite different. They are examples of C++'s powerful capability, called **operator overloading**.

Operator Precedence

Operators in C++ are assigned a **precedence** that determines the order in which operations are performed in the absence of parentheses.

Operator Precedences

Type	Operators
scope resolution	namespace_name :: member
selection/subscripting	class_name.member pointer->member array[exp]
function call	function(args)
postfix operators	var++ var--
prefix operators	++var --var +exp -exp ~exp !exp
dereference/address	*pointer &var
multiplication/division	* / %
addition/subtraction	+ -
shift	<< >>
comparison	< <= > >=
equality	== !=
bitwise and	&
bitwise exclusive-or	^
bitwise or	
logical and	&&
logical or	
conditional	bool_exp ? true_exp : false_exp
assignment	= += -= *= /= %= >= <<= &= ^= =

Figure 1.1 The C++ precedence rules. The notation "**exp**" denotes any expression.

Since these rules are complex, it is **a good idea** to *add parentheses to complex expressions to make your intent clear to someone reading your program*.

- Changing Types through Casting

Casting is an operation that allows us to *change the type of a variable*. In essence, we can take a variable of one type and **cast it into an equivalent variable of another type**. Casting is useful in many situations. There are **two fundamental types of casting** that can be done in C++. We can either *cast with respect to the fundamental types* or we can *cast with respect to class objects and pointers*.

Let *exp* be some expression, and let *T* be a type. To cast the value of the expression to type *T* we can use the notation "(*T*)*exp*." We call this a **C-style cast**. If the desired type is a type name (as opposed to a type expression), there is an alternate **functional-style cast**. This has the form "*T(exp)*."

```
int    cat = 14;
double dog = (double) cat;           // traditional C-style cast
double pig = double(cat);           // C++ functional cast
```

Both forms of casting are legal, **but some authors prefer the functional-style cast**.

Casting to a type of higher precision or size is often needed in forming expressions.

```
int    i1 = 18;
int    i2 = 16;
double dv1 = i1 / i2;               // dv1 has value 1.0
double dv2 = double(i1) / double(i2); // dv2 has value 1.125
double dv3 = double( i1 / i2 );     // dv3 has value 1.0
```

Explicit Cast Operators

Casting operations can vary from *harmless to dangerous*, depending on how similar the two types are and whether information is lost. **One important element of good software design is that programs be portable**, meaning that they behave the same on different machines.

For this reason, C++ provides a number of casting operators that make the **safety of the cast** much more explicit. These are called the `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`.

Static Casting

Static casting is used when a conversion is made between two related types,
`static_cast<desired_type>(expression)`

The most common use is for conversions between numeric types.

```
double d1 = 3.2;
double d2 = 3.9999;
int    i1 = static_cast<int>(d1);           // i1 has value 3
int    i2 = static_cast<int>(d2);           // i2 has value 3
```

This type of casting is more verbose than the C-style and functional-style casts shown earlier. But this form is appropriate, because it serves as a visible warning to the programmer that a potentially unsafe operation is taking place.

Implicit Casting

There are many instances where the programmer has not requested an **explicit cast**, but a change of types is required. In many of these cases, C++ performs an **implicit cast**. That is, the compiler automatically inserts a cast into the machine-generated code. For example, when numbers of different types are involved in an operation, the compiler automatically casts to the stronger type.

```
int    i   = 3;
double d   = 4.8;
double d3 = i / d;           // d3 = 0.625 = double(i)/d
int    i3 = d3;              // i3 = 0 = int(d3)
                           // Warning! Assignment may lose information
```

A general rule with casting is to "play it safe." If a compiler's behavior regarding the implicit casting of a value is uncertain, then we are safest in using an explicit cast. Doing so makes our intentions clear.

1.3. Control Flow

If Statement

The most common method of making choices in a C++ program is through the use of an **if statement**.

```
if ( condition )
    true_statement
else if ( condition )
    else_if_statement
else
    else_statement
```

Each of the conditions should return a Boolean result. The conditions are tested one by one, and the statement associated with the first true condition is executed. All the other statements are skipped.

```
if ( snowLevel < 2 ) {
    goToClass();           // do this if snow level is less than 2
    comeHome();
}
else if ( snowLevel < 5 )
    haveSnowballFight(); // if level is at least 2 but less than 5
else if ( snowLevel < 10 )
    goSkiing();          // if level is at least 5 but less than 10
else
    stayAtHome();        // if snow level is 10 or more
```

Switch Statement

A **switch statement** provides an efficient way to distinguish between many different options according to the value of an integral type.

```

char command;
cin >> command;           // input command character
switch (command) {          // switch based on command value
    case 'I' :              // if (command == 'I')
        editInsert();
        break;
    case 'D' :              // else if (command == 'D')
        editDelete();
        break;
    case 'R' :              // else if (command == 'R')
        editReplace();
        break;
    default :                // else
        cout << "Unrecognized command\n";
        break;
}

```

The argument of the switch can be any integral type or enumeration. The "**default**" case is executed if none of the cases equals the switch argument.

Each case in a switch statement should be terminated with a **break** statement, which, when executed, exits the switch statement. Otherwise, the flow of control "**falls through**" to the next case.

While and Do-While Loops

C++ has two kinds of conditional loops for iterating over a set of statements as long as some specified condition holds. These two loops are the standard **while loop** and the **do-while loop**. One loop tests a Boolean condition before performing an iteration of the loop body and the other tests a condition after. (*The definition of iteration*: a single execution of a set of instructions that are to be repeated)

while (*condition*)
 loop_body_statement

```

int a[100];
// ...
int i = 0;
int sum = 0;
while (i < 100 && a[i] >= 0) {
    sum += a[i++];
}

```

do
 loop_body_statement
while (*condition*)

For Loop

Many loops involve three common elements: an initialization, a condition under which to continue execution, and an increment to be performed after each execution of the loop's body.

A **for loop** conveniently encapsulates these three elements.

for (*initialization* ; *condition* ; *increment*)
 loop_body_statement

```

const int NUM_ELEMENTS = 100;
double b[NUM_ELEMENTS];
// ...
for (int i = 0; i < NUM_ELEMENTS; i++) {
    if (b[i] > 0)
        cout << b[i] << '\n';
}

```

Break and Continue Statements

C++ provides statements to change control flow, including the **break**, **continue**, and **return** statements. A break statement is used to "break" out of a loop or switch statement.

```
int a[100];
// ...
int sum = 0;
for (int i = 0; i < 100; i++) {
    if (a[i] < 0) break;
    sum += a[i];
}
```

The other statement that is often useful for *altering loop behavior* is the **continue** statement. The continue statement can **only be used inside loops** (**for**, **while**, and **do-while**). The continue statement causes *the execution to skip to the end of the loop, ready to start a new iteration*.

1.4. Functions

A **function** is a chunk of code that can be called to perform some well-defined task. In order to define a function, we need to provide the following information to the compiler:

Return type. This specifies the type of value or object that is returned by the function. A function that returns no value (the return type is **void**) is sometimes called a **procedure**.

Function name. Ideally, the function's name should provide a hint to the reader as to what the function does.

Argument list. The argument list is given as a comma-separated list enclosed in parentheses, where each entry consists of the name of the argument and its type.

Function body. If the function returns a value, the body will typically end with a **return** statement, which specifies the final function value.

Function specifications in C++ typically involve **two steps, declaration and definition**. A function is **declared**, by specifying three things: *the function's return type, its name, and its argument list*. **The declaration makes the compiler aware of the function's existence**, and allows the compiler to verify that the function is being used correctly. This three-part combination of return type, function name, and argument types is called the **function's signature or prototype**.

```
bool evenSum(int a[], int n); // function declaration
```

Second, the function is **defined**. The definition consists *both of the function's signature and the function body*. The function declaration **must appear in every file that invokes the function**, but the definition **must appear only once**.

```
bool evenSum(int a[], int n) { // function definition
    int sum = 0;
    for (int i = 0; i < n; i++) // sum the array elements
        sum += a[i];
    return (sum % 2) == 0; // returns true if sum is even
}
```

The expression in the **return** statement **may** take a minute to understand (return **true** or **false**).

```
bool evenSum(int a[], int n); // function declaration
```

```
int main() {
    int list[] = {4, 2, 7, 8, 5, 1};
    bool result = evenSum(list, 6); // invoke the function
    if (result) cout << "the sum is even\n";
    else cout << "the sum is odd\n";
    return EXIT_SUCCESS;
}
```

Let us consider this example in greater detail. The names "a" and "n" in the function definition are called **formal arguments** since they serve merely as placeholders. The variable "list" and

literal "6" in the function call in the main program are the **actual arguments**. Exact type agreement is not always necessary, however, for the compiler may perform implicit type conversions in some cases, such as casting a short actual argument to match an int formal argument.

- Argument Passing

By default, arguments in C++ programs are passed by value.

Sometimes it is useful for the function to modify one of its arguments. To do so, we can explicitly define a formal argument to be a **reference type** (as introduced in Section 1.1.3). When we do this, any modifications made to an argument in the function modifies the corresponding actual argument. This is called **passing the argument by reference**.

```
void f(int value, int& ref) {           // one value and one reference
    value++;
    ref++;
    cout << value << endl;               // no effect on the actual argument
    cout << ref << endl;                 // modifies the actual argument
}                                         // outputs 2
                                           // outputs 6

int main() {
    int cat = 1;
    int dog = 5;
    f(cat, dog);                      // pass cat by value, dog by ref
    cout << cat << endl;               // outputs 1
    cout << dog << endl;               // outputs 6
    return EXIT_SUCCESS;
}
```

Another way to modify an argument is to **pass the address of the argument**, rather than the argument itself. Reference arguments achieve essentially the same result with less notational burden.

Constant References as Arguments

There is a good reason for choosing to pass structure and class arguments by reference. Passing such an argument by reference is **much more efficient**, since *only the address of the structure need be passed*.

Since most function arguments are not modified, an even better practice is to pass an argument as a "constant reference." Such a declaration informs the compiler that, even though the argument is being passed by reference, the function cannot alter its value. Furthermore, the function is not allowed to pass the argument to another function that might modify its value.

```
void someFunction(const Passenger& pass) {
    pass.name = "new name";           // ILLEGAL! pass is declared const
}
```

Array Arguments

When an array is passed to a function, it is converted to a pointer to its initial element. That is, an object of type `T[]` is converted to type `T*`. Thus, an assignment to an element of an array within a function does modify the actual array contents. In short, **arrays are not passed by value**.

By the same token, it is not meaningful to pass an array back as the result of a function call. Essentially, an attempt to do so will only pass a pointer to the array's initial element.

- Overloading and Inlining

Overloading means defining two or more functions or operators that have the same name, but whose effect depends on the types of their actual arguments.

Function Overloading

Function overloading occurs when two or more functions are defined with the same name but with different argument lists. Such definitions are useful in situations where we desire two functions that achieve essentially the same purpose, but do it with different types of arguments.

```
void print(int x) // print an integer
{ cout << x; }

void print(const Passenger& pass) { // print a Passenger
    cout << pass.name << " " << pass.mealPref;
    if (pass.isFreqFlyer)
        cout << " " << pass.freqFlyerNo;
}
```

When the print function is used, the compiler considers the types of the actual argument and invokes the appropriate function, that is, the one with signature closest to the actual arguments.

Operator Overloading

C++ also allows overloading of operators, such as `+,*,+=,` and `<<`. Not surprisingly, such a definition is called **operator overloading**.

```
bool operator==(const Passenger& x, const Passenger& y) {
    return x.name == y.name
        && x.mealPref == y.mealPref
        && x.isFreqFlyer == y.isFreqFlyer
        && x.freqFlyerNo == y.freqFlyerNo;
}
```

This definition is similar to a function definition, but in place of a function name we use "operator`==`". In general, the `==` is replaced by whatever operator is being defined. For binary operators we have two arguments, and for unary operators we have just one.

Another useful application of operator overloading is for defining input and output operators for classes and structures.

```
ostream& operator<<(ostream& out, const Passenger& pass) {
    out << pass.name << " " << pass.mealPref;
    if (pass.isFreqFlyer) {
        out << " " << pass.freqFlyerNo;
    }
    return out;
}
```

The type `ostream` is the system's output stream type.

Operator overloading is a powerful mechanism, but it is easily abused. Good programmers usually restrict operator overloading to certain general purpose operators such as "`<<`" (output), "`=`" (assignment), "`==`" (equality), "`[]`" (indexing, for sequences).

In-line Functions

Very short functions may be defined to be **inline**. This is a hint to the compiler it should simply expand the function code in place, rather than using the system's call-return mechanism. As a rule of thumb, *in-line functions should be very short (at most a few lines) and should not involve any loops or conditionals.*

```
inline int min(int x, int y) { return (x < y ? x : y); }
```

1.5. Classes

The concept of a *class* is fundamental to C++, since it provides a way to define new user-defined types, complete with associated functions and operators.

● Class Structure

A class consists of *members*. Members that are variables or constants are *data members* (also called *member variables*) and members that are functions are called *member functions* (also called *methods*).

```
class Counter { // a simple counter
public:
    Counter(); // initialization
    int getCount(); // get the current count
    void increaseBy(int x); // add x to the count
private:
    int count; // the counter's value
};
```

Observe that the class definition is separated into two parts by the keywords **public** and **private**. The public section defines the class's *public interface*. These are the entities that users of the class are allowed to access. In contrast, the private section declares entities that **cannot be accessed by users of the class**.

Next, we present the definitions of these member functions.

```
Counter::Counter() // constructor
{
    count = 0;
}
int Counter::getCount() // get current count
{
    return count;
}
void Counter::increaseBy(int x) // add x to the count
{
    count += x;
}
```

The first of these functions has the same name as the class itself. This is a **special member function** called a *constructor*. A constructor's job is to initialize the values of the class's member variables.

We declare a new object of type Counter, called ctr. This **implicitly invokes the class's constructor**, and thus initializes the counter's value to 0.

```
Counter ctr; // an instance of Counter
cout << ctr.getCount() << endl; // prints the initial value (0)
ctr.increaseBy(3); // increase by 3
cout << ctr.getCount() << endl; // prints 3
ctr.increaseBy(5); // increase by 5
cout << ctr.getCount() << endl; // prints 8
```

Access Control

One important feature of classes is the notion of *access control*. Members may be declared to be *public*, which means that **they are accessible from outside the class**, or *private*, which means that **they are accessible only from within the class**. (We discuss two **exceptions** to this later: protected access and friend functions.)

```
Counter ctr; // ctr is an instance of Counter
// ...
cout << ctr.count << endl; // ILLEGAL - count is private
```

The syntax for a class is as follows.

```
class < class_name > {
public:
    public_members
private:
    private_members
};
```

Note that if no *access specifier* is given, **the default is private for classes and public for structures**.

Member Functions

Notice that the constructor does **not** have a return type.

```
class Passenger {           // Passenger (as a class)
public:
    Passenger();          // constructor
    bool isFrequentFlyer() const; // is this a frequent flyer?
                                // make this a frequent flyer
    void makeFrequentFlyer(const string& newFreqFlyerNo);
    // ... other member functions
private:
    string     name;        // passenger name
    MealType   mealPref;    // meal preference
    bool       isFreqFlyer;  // is a frequent flyer?
    string     freqFlyerNo; // frequent flyer number
};
```

Class **member functions** can be placed in **two major categories**: *accessor functions*, which only **read class data**, and *update functions*, which may **alter class data**. The keyword "**const**" indicates that the member function `isFrequentFlyer` is an accessor. This **informs the user of the class that this function will not change the object contents**. It also allows the compiler to catch a potential error should we inadvertently attempt to modify any class member variables. (Note this concept is similar to the **command query separation (CQS) principle** which states that every method should be either a command or a query, but not both)

Member functions may either be defined inside or outside the class body. Most C++ style manuals *recommend defining all member functions outside the class*, in order to present a clean public interface in the class's definition.

```
bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}
void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}
```

Notice that when we are within the body of a member function, the member variables (such as `isFreqFlyer` and `freqFlyerNo`) are *given without reference to a particular object*. These functions will be invoked on **a particular Passenger object**.

```
Passenger pass;           // pass is a Passenger
// ...
if ( !pass.isFrequentFlyer() ) { // not already a frequent flyer?
    pass.makeFrequentFlyer("392953"); // set pass's freq flyer number
}
pass.name = "Joe Blow";    // ILLEGAL! name is private
```

In-Class Function Definitions

We can **also define members within the class body**. When a member function is defined within a class it is **compiled in line** (recall Section 1.4.2). As with in-line functions, in-class function definitions should be reserved for **short functions** that do not involve loops or conditionals.

```
class Passenger {
public:
    // ...
    bool isFrequentFlyer() const { return isFreqFlyer; }
    // ...
};
```

● Constructors and Destructors

A **constructor** is a special member function whose task is to perform such an initialization. It is

invoked when a new class object comes into existence. There is an analogous *destructor* member function that is called when a class object goes out of existence.

Constructors

A constructor member function's name is the *same as the class*, and it has *no return type*. Because objects may be initialized in different ways, it is natural to define different constructors and *rely on function overloading* to determine which one is to be called.

The first constructor has no arguments. Such a constructor is called a **default constructor**, since it is used in the absence of any initialization information. The second constructor is given the values of the member variables to initialize. The third constructor is given a Passenger reference from which to copy information. This is called a **copy constructor**.

```
class Passenger {  
private:  
    // ...  
public:  
    Passenger(); // default constructor  
    Passenger(const string& nm, MealType mp, const string& ffn = "NONE");  
    Passenger(const Passenger& pass); // copy constructor  
    // ...  
};
```

Look *carefully* at the second constructor. The notation `ffn="NONE"` indicates that the argument for `ffn` is a **default argument**. That is, an actual argument need not be given, and if so, the value "`NONE`" is used instead. Default arguments can be assigned any legal value and can be used for more than one argument. It is often *useful to define default values for all the arguments of a constructor*. Such a constructor is the **default constructor** because it is called if no arguments are given (Such constructor *cannot coexist with* the `class_name()` function like `Passenger()`). Default arguments can be used with any function (not just constructors). *Note that the default argument is given in the declaration, but not in the definition.*

```
Passenger::Passenger() { // default constructor  
    name = "--NO NAME--";  
    mealPref = NO_PREF;  
    isFreqFlyer = false;  
    freqFlyerNo = "NONE";  
}  
// constructor given member values  
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {  
    name = nm;  
    mealPref = mp;  
    isFreqFlyer = (ffn != "NONE"); // true only if ffn given  
    freqFlyerNo = ffn;  
}  
// copy constructor  
Passenger::Passenger(const Passenger& pass) {  
    name = pass.name;  
    mealPref = pass.mealPref;  
    isFreqFlyer = pass.isFreqFlyer;  
    freqFlyerNo = pass.freqFlyerNo;  
}
```

Here are some examples of how the constructors above can be invoked to define Passenger objects.

```
Passenger p1; // default constructor  
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor  
Passenger p3("Pocahontas", REGULAR); // not a frequent flyer  
Passenger p4(p3); // copied from p3  
Passenger p5 = p2; // copied from p2  
Passenger* pp1 = new Passenger(); // default constructor  
Passenger* pp2 = new Passenger("Joe Blow", NO_PREF); // 2nd constr.  
Passenger pa[20]; // uses the default constructor
```

Although they look different, the declarations for p4 and p5 both call the copy constructor.

Initializing Class Members with Initializer Lists

If the type of name is a class without an assignment operator, this type of initialization ("name=nm") might not be possible. In order to deal with the issue of initializing member variables that are themselves classes, C++ provides an alternate method of initialization called an **initializer list**. This list is placed between the constructor's argument list and its body. It consists of a colon (:) followed by a comma-separated list of the form `member_name(initial_value)`. The initializer list is executed before the body of the constructor.

```
// constructor using an initializer list
Passenger::Passenger(const string& nm, MealType mp, string ffn)
: name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE")
{ freqFlyerNo = ffn; }
```

Destructors

A constructor is called when a class object comes into existence. A **destructor** is a member function that is automatically called when a class object ceases to exist. If a class object comes into existence dynamically using the **new** operator, the destructor will be called when this object is destroyed using the **delete** operator. If a class object comes into existence because it is a local variable in a function that has been called, the destructor will be called when the function returns. The destructor for a class T is denoted `~T`. It takes no arguments and has no return type. Destructors are needed when classes allocate resources, such as memory, from the system.

```
class Vect {                                // a vector class
public:
    Vect(int n);                         // constructor, given size
    ~Vect();                             // destructor
    // ... other public members omitted
private:
    int*      data;                      // an array holding the vector
    int       size;                      // number of array entries
};

Vect::Vect(int n) {                        // constructor
    size = n;
    data = new int[n];                  // allocate array
}

Vect::~Vect() {                           // destructor
    delete [] data;                   // free the allocated array
}
```

We are not strictly required by C++ to provide our own destructor. Nonetheless, if our class allocates memory, we should write a destructor to free this memory.

● Classes and Memory Allocation

When a class performs memory allocation using **new**, care must be taken to avoid a number of common programming errors. We have shown above that failure to deallocate storage in a class's destructor can result in memory leaks. A somewhat more insidious problem occurs when classes that allocate memory fail to provide a copy constructor or an assignment operator. Consider the following example, using our Vect class.

```
Vect a(100);                            // a is a vector of size 100
Vect b = a;                             // initialize b from a (DANGER!)
Vect c;                               // c is a vector (default size 10)
c = a;                                // assign a to c (DANGER!)
```

In reality all three of these vectors share the same 100-element array.

The declaration "Vect b=a" initializes b from a. Since we provided **no copy constructor** in Vect, the system uses its default, which simply copies each member of a to b. In particular it sets "b.data=a.data." Notice that this **does not copy the contents of the array**; rather it copies the pointer to the array's initial element. This default action is sometimes called a **shallow copy**.

The statement "c=a," **also does a shallow copy of a to c**. Only pointers are copied, not array contents. Worse yet, we have lost the pointer to c's original 10-element array, *thus creating a memory leak*.

Fortunately, **there is a simple fix for all of these problems**. The problems arose *because we allocated memory and we used the system's default copy constructor and assignment operator*. If a class allocates memory, you **should provide a copy constructor and assignment operator to allocate new memory for making copies**. A **copy constructor** for a class T is typically declared to take a single argument,

```
Vect::Vect(const Vect& a) {  
    size = a.size; // copy sizes  
    data = new int[size]; // allocate new array  
    for (int i = 0; i < size; i++) { // copy the vector contents  
        data[i] = a.data[i];  
    }  
}
```

The **assignment operator** is handled by overloading the = operator as shown below. The argument "a" plays the role of the object on the **right side of the assignment operator**. The assignment operator *deletes the existing array storage, allocates a new array of the proper size, and copies elements into this new array*. The if statement checks against the possibility of **self-assignment**. (This can sometimes happen when different variables reference the same object.) We perform this check using the keyword **this**. For any instance of a class object, "**this**" is **defined to be the address of this instance**.

```
Vect& Vect::operator=(const Vect& a) { // assignment operator from a  
    if (this != &a) { // avoid self-assignment  
        delete [] data; // delete old array  
        size = a.size; // set new size  
        data = new int[size]; // allocate new array  
        for (int i=0; i < size; i++) { // copy the vector contents  
            data[i] = a.data[i];  
        }  
    }  
    return *this;  
}
```

Notice that in the last line of the assignment operator we *return a reference to the current object with the statement "return *this."* Such **an approach is useful for assignment operators**, since it allows us to chain together assignments, as in "a=b=c." The assignment "b=c" invokes the assignment operator, copying variable c to b and then returns a reference to b. This result is then assigned to variable a.

Remember

Every class that allocates its own objects using **new** should:

- Define a **destructor** to free any allocated objects.
- Define a **copy constructor**, which allocates its **own new member storage** and **copies the contents of member variables**.
- Define an **assignment operator**, which **deallocates old storage, allocates new storage**, and **copies all member variables**.

Some programmers recommend that these functions be included for every class, even if memory is not allocated, but we are not so fastidious. In rare instances, we may want to forbid users from using one or more of these operations. For example, we may not want a huge data structure to be copied inadvertently. In this case, we can define **empty copy constructors** and **assignment functions** and make them **private** members of the class.

● Class Friends and Class Members

Complex data structures typically involve the interaction of many different classes. In such cases, there are often issues coordinating the actions of these classes to allow sharing of information. We said *private members of a class may only be accessed from within the class* (Note here the range is the class, so different objects of the same class can access private members with each other), but there is **an exception to this**. Specifically, we can declare a function as a **friend**, which means that **this function may access the class's private data**. There are a number of reasons for defining friend functions. One is that *syntax requirements may forbid us from defining a member function*.

```
class SomeClass {  
private:  
    int secret;  
public:  
    // ... // give << operator access to secret  
    friend ostream& operator<<(ostream& out, const SomeClass& x);  
};  
  
ostream& operator<<(ostream& out, const SomeClass& x)  
{ cout << x.secret; }
```

Another time when it is appropriate to use friends is *when two different classes are closely related*.

```
class Vector { // a 3-element vector  
public: // ... public members omitted  
private:  
    double coord[3]; // storage for coordinates  
    friend class Matrix; // give Matrix access to coord  
};  
  
class Matrix { // a 3x3 matrix  
public:  
    Vector multiply(const Vector& v); // multiply by vector v  
    // ... other public members omitted  
private:  
    double a[3][3]; // matrix entries  
};  
  
Vector Matrix::multiply(const Vector& v) { // multiply by vector v  
    Vector w;  
    for (int i = 0; i < 3; i++)  
        for (int j = 0; j < 3; j++)  
            w.coord[i] += a[i][j] * v.coord[j]; // access to coord allowed  
    return w;  
}
```

The ability to **declare friendship relationships between classes is useful**, but the **extensive use**

of friends often indicates a poor class structure design. For example, a better solution would be to have class Vector define a public subscripting operator. Then the multiply function could use this public member to access the vector class, rather than access private member data. Note that "friendship" is not transitive. For example, if a new class Tensor was made a friend of Matrix, Tensor would not be a friend of Vector, unless class Vector were to explicitly declare it to be so.

Nesting Classes and Types within Classes

We know that classes may define member variables and member functions. **Classes may also define their own types as well.** In particular, we can nest a class definition within another class. Such a **nested class** is often **convenient in the design of data structures.**

```
class Book {
public:
    class Bookmark {
        // ... (Bookmark definition here)
    };
    // ... (Remainder of Book definition)
}
```

We might define a member function that returns a bookmark within the book, say, to the start of some chapter. **Outside the class Book**, we use the **scope-resolution operator**, Book::Bookmark, in order to refer to this nested class.

- The Standard Template Library

The **Standard Template Library (STL)** is a collection of useful classes for common data structures. In addition to the **string** class, which we have seen many times, it also provides data structures for the following **standard containers**.

stack	Container with last-in, first-out access
queue	Container with first-in, first-out access
deque	Double-ended queue
vector	Resizeable array
list	Doubly linked list
priority_queue	Queue ordered by value
set	Set
map	Associative array (dictionary)

Templates and the STL Vector Class

One of the **important features of the STL** is that each such object can *store objects of any one type*. Such a class whose definition depends on a user-specified type is called **a template**.

We **specify the type of the object being stored in the container** in angle brackets (`<...>`).

```
#include <vector>
using namespace std;           // make std accessible

vector<int> scores(100);      // 100 integer scores
vector<char> buffer(500);      // buffer of 500 characters
vector<Passenger> passenList(20); // list of 20 Passengers
```

As usual, the include statement provides the necessary declarations for using the vector class. Each **instance of an STL vector** can only hold objects of one type.

STL vectors are **superior to standard C++ arrays** in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([]). They can also be accessed by the at member function. The advantage of the latter is that it performs range checking and generates an error exception if the index is out of bounds. Recall that standard arrays in C++ do not even know their size, and hence range checking is not even possible. In contrast, a vector

object's size is given by its `size` member function. Unlike standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the `resize` member function.

```
int i = // ...
cout << scores[i];           // index (range unchecked)
buffer.at(i) = buffer.at(2 * i); // index (range checked)
vector<int> newScores = scores; // copy scores to newScores
scores.resize(scores.size() + 10); // add room for 10 more elements
```

More on STL Strings

Earlier, we discussed the use of the addition operator ("`+`") for concatenating strings, the operator "`+=`" for appending a string to the end of an existing string, the function `size` for determining the length of a string, and the indexing operator ("`[]`") for accessing individual characters of a string.

Let us present a few more string functions. In the table below, let `s` be an STL string, and let `p` be either an STL string or a standard C++ string. Let `i` and `m` be nonnegative integers.

<code>s.find(p)</code>	Return the index of first occurrence of string <code>p</code> in <code>s</code>
<code>s.find(p, i)</code>	Return the index of first occurrence of string <code>p</code> in <code>s</code> on or after position <code>i</code>
<code>s.substr(i,m)</code>	Return the substring starting at position <code>i</code> of <code>s</code> and consisting of <code>m</code> characters
<code>s.insert(i, p)</code>	Insert string <code>p</code> just prior to index <code>i</code> in <code>s</code>
<code>s.erase(i, m)</code>	Remove the substring of length <code>m</code> starting at index <code>i</code>
<code>s.replace(i, m, p)</code>	Replace the substring of length <code>m</code> starting at index <code>i</code> with <code>p</code>
<code>getline(is, s)</code>	Read a single line from the input stream <code>is</code> and store the result in <code>s</code>

In order to indicate that a pattern string `p` is not found, the `find` function returns the **special value `string::npos`**. Strings can also be compared lexicographically, using the C++ comparison operators: `<`, `<=`, `>`, `>=`, `==`, and `!=`.

```
string s = "a dog";           // "a dog"
s += " is a dog";             // "a dog is a dog"
cout << s.find("dog");        // 2
cout << s.find("dog", 3);     // 11
if (s.find("doug") == string::npos) { } // true
cout << s.substr(7, 5);       // "s a d"
s.replace(2, 3, "frog");      // "a frog is a dog"
s.erase(6, 3);                // "a frog a dog"
s.insert(0, "is ");            // "is a frog a dog"
if (s == "is a frog a dog") { } // true
if (s < "is a frog a toad") { } // true
if (s < "is a frog a cat") { } // false
```

1.6. C++ Program and File Organization

A typical large C++ program consists of many files, with *related pieces of code residing within each file*. For example, C++ programmers **commonly place each major class in its own file**.

Source Files

There are **two common file types**, source files and header files. **Source files** typically contain *most of the executable statements and data definitions*. This includes the **bodies of functions** and

definitions of any global variables.

Different **compilers** use different file naming conventions. Source file names typically have distinctive suffixes, such as ".cc", ".cpp", and ".C". Source files may be compiled separately by the compiler, and then these files are combined into one program by a system program called a **linker**.

Each nonconstant global variable and function may be defined only once. Other source files may share such a global variable or function provided they have a matching declaration. To indicate that a global variable is defined in another file, the type specifier "extern" is added. This keyword is not needed for functions.

```
File: Source1.cpp
    int cat = 1;                      // definition of cat
    int foo(int x) { return x+1; }      // definition of foo
File: Source2.cpp
    extern int cat;                   // cat is defined elsewhere
    int foo(int x);                  // foo is defined elsewhere
```

Header Files

Since **source files using shared objects** must provide identical declarations, we commonly store these shared declarations in a **header file**, which is then read into each such source file using an #include statement. Statements beginning with # are handled by a special program, called the **preprocessor**, which is invoked automatically by the compiler. A header file typically contains many declarations, including classes, structures, constants, enumerations, and **typedefs**. Header files generally do not contain the definition (body) of a function. In-line functions are an exception, however, as their bodies are given in a header file.

Except for some standard library headers, the convention is that header file names end with a ".h" suffix. Standard library header files are indicated with angle brackets, as in <iostream>, while **other local header files are indicated using quotes**, as in "myIncludes.h".

```
#include <iostream>                 // system include file
#include "myIncludes.h"              // user-defined include file
```

As a general rule, we should avoid including namespace using directives in header files, because any source file that includes such a header file has its namespace expanded as a result.

● An Example Program

The CreditCard Class

(Page 48)

1.7. Writing a C++ Program

As with any programming language, writing a program in C++ involves three fundamental steps:

1. Design
2. Coding
3. Testing and Debugging.

● Design

In this step, we decide how to divide the workings of our program into classes, we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning C++ programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some general rules of thumb that we can apply when determining how to

define our classes.

Responsibilities: Divide the work into different **actors**, each with a different responsibility. Try to describe responsibilities using action verbs. These actors form the classes for the program.

Independence: Define the work for each class to be as independent from other classes as possible. **Subdivide responsibilities between classes** so that each class has *autonomy over some aspect of the program*. Give data (as member variables) to the class that has jurisdiction over the actions that require access to this data.

Behaviors: **Define the behaviors for each class** carefully and precisely, so that the consequences of each action performed by a class are well understood by other classes with which it interacts. These behaviors define the member functions that this class performs. The set of behaviors for a class is sometimes referred to as a **protocol**, since we expect the behaviors for a class to hold together as a cohesive unit.

Defining the classes, together with their member variables and member functions, determines the design of a C++ program.

Pseudo-Code

Programmers are often asked to describe algorithms in a way that is intended for human eyes only, prior to writing actual code. Such descriptions are called **pseudo-code**. Pseudo-code is not a computer program, but is more structured than usual prose. Pseudo-code is a mixture of natural language and high-level programming constructs that **describe the main ideas behind a generic implementation of a data structure or algorithm**. There really is no precise definition of the pseudo-code language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudo-code mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as C, C++, and Java.

These constructs include the following:

Expressions: We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign (\leftarrow) as the assignment operator in assignment statements (equivalent to the `=` operator in C++) and we use the equal sign ($=$) as the equality relation in Boolean expressions (equivalent to the `==` relation in C++).

Function declarations: **Algorithm** `name(arg1,arg2, ...)` declares a new function "name" and its arguments.

Decision structures: **if** *condition* **then** *true-actions* [**else** *false-actions*]. We use indentation to indicate what actions should be included in the true-actions and false-actions.

While-loops: **while** *condition* **do** *actions*. We use indentation to indicate what actions should be included in the loop actions.

Repeat-loops: **repeat** *actions* **until** *condition*. We use indentation to indicate what actions should be included in the loop actions.

For-loops: **for** *variable-increment-definition* **do** *actions*. We use indentation to indicate what actions should be included among the loop actions.

Array indexing: $A[i]$ represents the *i*th cell in the array *A*. The cells of an *n*-celled array *A*

are indexed from $A[0]$ to $A[n - 1]$ (consistent with C++).

Member function calls: `object.method(args)` (object is optional if it is understood).

Function returns: `return value`. This operation returns the value specified to the method that called this one.

Comments: `{ Comment goes here. }`. We enclose comments in braces.

- Coding

As mentioned above, one of the key steps in coding up an object-oriented program is *coding up the descriptions of classes and their respective data and member functions*.

Many programmers do their initial coding not on a computer, but by using **CRC cards**. **Class-Responsibility-Collaborator (CRC)** cards are simple index cards that subdivide the work required of a program. The **main idea** behind this tool is to have each card represent a component, which will ultimately become a class in our program.

An alternative to CRC cards is the use of **UML (Unified Modeling Language)** diagrams to express the organization of a program, and the use of pseudo-code to describe the algorithms. UML diagrams are a standard visual notation to express object-oriented software designs.

Once we have decided on the classes and their respective responsibilities for our programs, we are ready to begin coding. We create the actual code for the classes in our program by using either an independent text editor (such as emacs, notepad, or vi), or the editor embedded in an **integrated development environment (IDE)**, such as Microsoft's Visual Studio and Eclipse.

Once we have completed coding for a program (or file), **we then compile** this file into working code by invoking a compiler. If our program contains syntax errors, they will be identified, and we will have to go back into our editor to fix the offending lines of code. Once we have **eliminated all syntax errors and created the appropriate compiled code, we then run our program**.

Readability and Style

Programs should be made **easy to read and understand**. Good programmers should therefore be mindful of their coding style and develop a style that communicates the important aspects of a program's design for both humans and computers. Much has been written about good coding style. Here are some of the main principles:

Use meaningful names for identifiers. Try to choose names that can be read aloud and reflect the action, responsibility, or data each identifier is naming. **The tradition in most C++ circles is to capitalize the first letter of each word in an identifier, except for the first word in an identifier for a variable or method.** So, in this tradition, "Date," "Vector," and "DeviceManager" would identify classes, and "isFull," "insertItem," "studentName," and "studentHeight" would respectively identify member functions and variables.

Use named constants and enumerations instead of embedded values. Readability, robustness, and modifiability are enhanced if we include a series of definitions of named constant values in a class definition. These can then be used within this class and others to refer to special values for this class. Our convention is to **fully capitalize such constants** as shown below.

```

const int MIN_CREDITS = 12;      // min. credits in a term
const int MAX_CREDITS = 24;      // max. credits in a term
                                // enumeration for year
enum Year { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };

```

Indent statement blocks. Typically programmers indent each statement block **by four spaces**. (In this book, we typically use two spaces to avoid having our code overrun the book's margins.)

Organize each class in a consistent order. In the examples in this book, we usually use the following order:

1. Public types and nested classes
2. Public member functions
3. Protected member functions (internal utilities)
4. Private member data

Use comments that add meaning to a program and explain ambiguous or confusing constructs. *In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.*

- Testing and Debugging

Testing is the process of verifying the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the errors in it. Testing and debugging are often the most time-consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually not feasible, we should aim at executing the program on a representative subset of inputs. At the very minimum, we should make sure that every method in the program is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on special cases of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts an array of integers (that is, arranges them in ascending order), we should consider the following inputs:

- The array has zero length (no elements)
- The array has one element
- All the elements of the array are the same
- The array is already sorted
- The array is reverse sorted

In addition to special inputs to the program, we should also consider special conditions for the structures used by the program. For example, if we use an array to store data, we should make sure that boundary cases, such as inserting/removing at the beginning or end of the subarray holding data, are properly handled. While it is essential to use hand-crafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs.

There is a hierarchy among the classes and functions of a program induced by the "caller-callee" relationship. Namely, a function A is above a function B in the hierarchy if A calls B. There are two main testing strategies, **top-down** and **bottom-up**, which differ in the order in which functions are tested.

Bottom-up testing proceeds from lower-level functions to higher-level functions. Namely, bottom-level functions, which do not invoke other functions, are tested first, followed by functions that call only bottom-level functions, and so on. This strategy ensures that errors found in a method are not likely to be caused by lower-level functions nested within it.

Top-down testing proceeds from the top to the bottom of the method hierarchy. It is typically used in conjunction with **stubs**, a boot-strapping technique that replaces a lower-level method with a stub, a replacement for the method that simulates the output of the original method (+Mock). For example, if function A calls function B to get the first line of a file, we can replace B with a stub that returns a fixed string when testing A.

Debugging

The simplest debugging technique consists of using **print statements** (typically using the stream output operator, "<<") to track the values of variables during the execution of the program. *The problem with this approach is that the print statements need to be removed or commented out before the program can be executed as part of a "production" software system.*

A better approach is to run the program within a debugger, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is **the insertion of breakpoints within the code**. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected. In addition to fixed breakpoints, advanced debuggers allow for **specification of conditional breakpoints**, which are triggered only if a given expression is satisfied.

Many IDEs, such as Microsoft Visual Studio and Eclipse provide built-in debuggers.

CHAPTER 2: Object-Oriented Design

2.1. Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented design paradigm are called **objects**. An object comes from a **class**, which is *a specification of the data members that the object contains*, as well as the **member functions** (also called **methods** or **operations**) that the *object can execute*. Each class presents to the outside world a concise and consistent view of the **objects that are instances of this class**, without going into too much unnecessary detail or giving others access to the inner workings of the objects. This view of computing is intended to **fulfill several goals and incorporate several design principles**, as explained below.

- Object-Oriented Design Goals

Software implementations should achieve **robustness**, **adaptability**, and **reusability**.

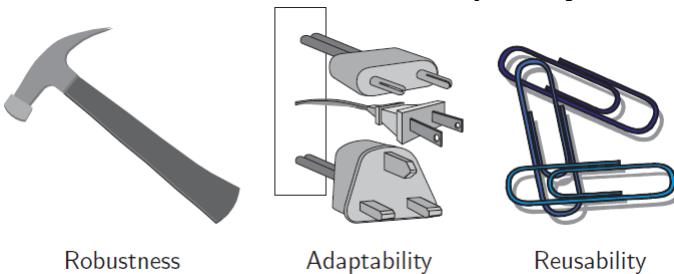


Figure 2.1 Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, which means that *a program produces the right output for all the anticipated inputs in the program's application*. In addition, we want software to be **robust**, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (for example, representing the price of an item) and instead is given a negative integer, then **the program should be able to recover gracefully from this error**. More importantly, in **life-critical applications**, where a software error can lead to injury or loss of life, software that is not robust could be deadly.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software therefore needs to be able to *evolve over time in response to changing conditions in its environment*. Thus, another important goal of quality software is that it achieves **adaptability** (also called **evolvability**). Related to this concept is **portability**, which is the *ability of software to run with minimal change on different hardware and operating system platforms*. An advantage of writing software in C++ is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, **the same code should be usable as a component of different systems in various applications**. Developing quality software can be *an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications*. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

- Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are *intended to facilitate the goals outlined above*, are the following:

- Abstraction
- Encapsulation
- Modularity.

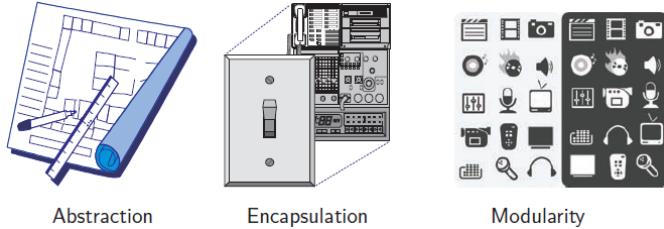


Figure 2.2 Principles of object-oriented design.

Abstraction

The notion of **abstraction** is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies **what** each operation does, but not **how** it does it. In C++, the functionality of a data structure is expressed through the public interface of the associated class or classes that define the data structure. By **public interface**, we mean *the signatures (names, return types, and argument types) of a class's public member functions*. This is the **only part of the class that can be accessed by a user of the class**.

An ADT is realized by a concrete data structure, which is modeled in C++ by a **class**. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify **how** the operations are performed in the body of each function. A C++ class is said to **implement an interface** if its functions include all the functions declared in the interface, thus providing a body for them. However, a class can have more functions than those of the interface.

Encapsulation

Another important principle of object-oriented design is the concept of **encapsulation**, which states that **different components of a software system should not reveal the internal details of their respective implementations (or implementation details)**. One of the **main advantages of encapsulation** is that *it gives the programmer freedom in implementing the details of a system*. The **only constraint** on the programmer is to maintain the abstract interface that outsiders see.

Modularity

In addition to abstraction and encapsulation, a fundamental principle of object-oriented design is **modularity**. Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. In object-oriented design, this code structuring approach centers around the concept of **modularity**. Modularity refers to an organizing principle for code in which **different components of a software system are divided into separate functional units**.

Hierarchical Organization

The structure imposed by modularity **helps to enable software reusability**. If software modules are written in an abstract way to solve general problems, then **modules can be reused when**

instances of these same general problems arise in other contexts.

For example, the structural definition of a wall is the same from house to house, typically being defined in terms of vertical studs, spaced at fixed-distance intervals, etc. Thus, an organized architect can reuse his or her wall definitions from one house to another. In reusing such a definition, some parts may require redefinition, for example, a wall in a commercial building may be similar to that of a house, but the electrical system and stud material might be different.

A natural way to organize various structural components of a software package is in a **hierarchical fashion**, which groups similar abstract definitions together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. A common use of such hierarchies is in an organizational chart where each link going up can be read as "is a," as in "a ranch is a house is a building." This kind of hierarchy is useful in software design, for it groups together common functionality at the most general level, and views specialized behavior as an extension of the general one.

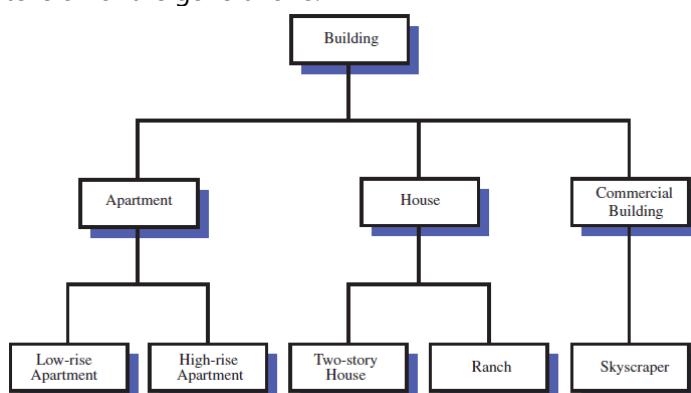


Figure 2.3 An example of an "is a" hierarchy involving architectural buildings.

- Design Patterns

One of the advantages of object-oriented design is that it facilitates robust, adaptable, and reusable software. **Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.**

Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. Of special relevance to this book is the concept of a **design pattern**, which describes a solution to a "typical" software design problem. A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of a name, which identifies the pattern, a context, which describes the scenarios for which this pattern can be applied, a template, which describes how the pattern is applied, and a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be consistently applied to implementations of data structures and algorithms. These design patterns fall into two groups—patterns for solving algorithm design problems and patterns for solving software engineering problems.

2.2. Inheritance and Polymorphism

To take advantage of hierarchical relationships, which are common in software projects, the

object-oriented design approach provides ways of reusing code (for reusability).

- Inheritance in C++

The object-oriented paradigm provides a modular and hierarchical organizing structure for reusing code through a technique called **inheritance**. This technique allows the design of generic classes that can be specialized to more particular classes, with the specialized classes reusing the code from the generic class (**And supports the polymorphism**).

A generic class is also known as a **base class**, **parent class**, or **superclass**. It defines "generic" members that apply in a multitude of situations. Any class that **specializes** or **extends** a base class need not give new implementations for the general functions, for it **inherits** them. It should only define those functions that are specialized for this particular class. Such a class is called a **derived class**, **child class**, or **subclass**.

Let us consider an example to illustrate these concepts. Suppose that we are writing a program to deal with people at a university. Below we show a partial implementation of a generic class for a person. We use "/* ... */" to indicate code that is irrelevant to the example and so has been omitted.

```
class Person {                                // Person (base class)
private:
    string      name;                      // name
    string      idNum;                     // university ID number
public:
    /* ...
     * void print();                         // print information
     * string getName();                    // retrieve name
    */;
}
```

Suppose we next wish to define a student object. We can derive our class **Student** from class **Person** as shown below (i.e., inheritance).

```
class Student : public Person {              // Student (derived from Person)
private:
    string      major;                     // major subject
    int         gradYear;                  // graduation year
public:
    /* ...
     * void print();                         // print information
     * void changeMajor(const string& newMajor); // change major
    */;
}
```

The "public Person" phrase indicates that the Student is derived from the Person class. (The keyword "public" specifies **public inheritance**.) When we derive classes in this way, there is an implied "is a" relationship between them. In this case, a Student "is a" Person. In particular, a *Student object inherits all the member data and member functions of class Person* in addition to providing its own members. The relationship between these two classes is shown graphically in a **class inheritance diagram** in *Figure 2.4*.

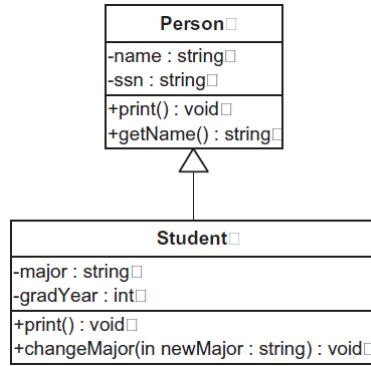


Figure 2.4 A class inheritance diagram, showing a base class Person and derived class Student.

Entries tagged with “-” are private and entries tagged with “+” are public. Each block of the diagram consists of three parts: the class name, the class member variables, and the class member functions. The type (or return type) of each member is indicated after the colon (“:”).

The arrow indicates that Student is derived from Person.

Member Functions

An object of type Person can access the public members of Person. An object of type Student can access the public members of **both classes**. If a Student object invokes the shared print function, it will **use its own version by default**. We use the **class scope operator (::)** to specify which class's function is used, as in Person::print and Student::print. Note that an object of type Person cannot access members of the base type, and thus it is not possible for a Person object to invoke the changeMajor function of class Student.

```

Person person("Mary", "12-345"); // declare a Person
Student student("Bob", "98-764", "Math", 2012); // declare a Student

cout << student.getName() << endl; // invokes Person::getName()
person.print(); // invokes Person::print()
student.print(); // invokes Student::print()
person.changeMajor("Physics"); // ERROR!
student.changeMajor("English"); // okay

```

C++ programmers often find it useful for a derived class to explicitly invoke a member function of a base class. For example, in the process of printing information for a student, it is natural to first print the information of the Person base class, and then print information particular to the student. Performing this task is done using the class scope operator.

```

void Person::print() { // definition of Person print
    cout << "Name " << name << endl;
    cout << "IDnum " << idNum << endl;
}

void Student::print() { // definition of Student print
    Person::print(); // first print Person information
    cout << "Major " << major << endl;
    cout << "Year " << gradYear << endl;
}

```

Without the “Person::” specifier used above, the Student::print function would call itself recursively, **which is not what we want**.

Protected Members

Even though class Student is inherited from class Person, **member functions of Student do not have access to private members of Person**. For example, **the following is illegal**.

```

void Student::printName() {
    cout << name << '\n'; // ERROR! name is private to Person
}

```

Special access privileges for derived classes can be provided by declaring members to be "protected." A protected member is "public" to all classes derived from this one, but "private" to all other functions. From a syntactic perspective, the keyword **protected** behaves in the same way as the keyword **private** and **public**. In the class example above, had we declared *name* to be protected rather than private, the above function *printName* would work fine.

Illustrating Class Protection

Consider for example, three classes: a base class *Base*, a derived class *Derived*, and an unrelated class *Unrelated*. The base class defines three integer members, one of each access type.

```
class Base {
    private:    int priv;
    protected: int prot;
    public:     int publ;
};

class Derived: public Base {
    void someMemberFunction() {
        cout << priv;           // ERROR: private member
        cout << prot;          // okay
        cout << publ;          // okay
    }
};

class Unrelated {
    Base X;

    void anotherMemberFunction() {
        cout << X.priv;        // ERROR: private member
        cout << X.prot;        // ERROR: protected member
        cout << X.publ;        // okay
    }
};
```

When designing a class, **we should give careful thought to the access privileges we give each member variable or function**. Member variables are almost always declared to be **private** or at least **protected**, since they determine the details of the class's implementation. A user of the class can access only the public class members, which **consist of the principal member functions for accessing and manipulating class objects**. Finally, **protected members** are commonly used for utility functions, which may be useful to derived classes.

Constructors and Destructors

When a derived class is constructed, it is the responsibility of this class's constructor to take care that the appropriate constructor is called for its base class. Class hierarchies in C++ are constructed bottom-up: base class first, then its members, then the derived class itself. For this reason, **the constructor for a base class needs to be called in the initializer list of the derived class**. The example below shows how constructors might be implemented for the *Person* and *Student* classes.

```
Person::Person(const string& nm, const string& id)
: name(nm),                      // initialize name
  idNum(id) {                     // initialize ID number

Student::Student(const string& nm, const string& id,
                 const string& maj, int year)
: Person(nm, id),                // initialize Person members
  major(maj),                     // initialize major
  gradYear(year) { }              // initialize graduation year
```

Only the *Person(nm, id)* call has to be in the initializer list. The other initializations could be placed in the constructor function body ({...}), **but putting class initializations in the**

initialization list is generally more efficient (**Note:** base class's data can't be initialized in this list separately, but can be in the body of the constructor of the derived class). Suppose that we create a new student object.

```
Student* s = new Student("Carol", "34-927", "Physics", 2014);
```

Note that the constructor for the Student class first makes a function call to Person("Carol", "34-927") to initialize the Person base class, and then it initializes the major to "Physics" and the year to 2014.

Classes are destroyed in the reverse order from their construction, with derived classes destroyed before base classes. For example, suppose that we declared destructors for these two classes. (**Note that destructors are not really needed in this case, because neither class allocates storage or other resources.**)

```
Person::~Person() { ... }           // Person destructor  
Student::~Student() { ... }        // Student destructor
```

If we were to destroy our student object, the Student destructor would be called first, followed by the Person destructor. Unlike constructors, the Student destructor does not need to (**and is not allowed to**) call the Person destructor. This happens automatically.

```
delete s;                          // calls ~Student() then ~Person()
```

Static Binding

When a class is derived from a base class, as with Student and Person, *the derived class becomes a subtype of the base class*, which means that we can use the derived class wherever the base class is acceptable.

```
Person* pp[100];                  // array of 100 Person pointers  
pp[0] = new Person(...);          // add a Person (details omitted)  
pp[1] = new Student(...);         // add a Student (details omitted)
```

Since getName is common to both classes, it can be invoked on either element of the array. A more interesting issue arises if we attempt to invoke print. Since pp[1] holds the address of a Student object, we might think that the function Student::print would be called. Surprisingly, the function Person::print is called in both cases, in spite of the apparent difference in the two objects. **Furthermore, pp[i] is not even allowed to access Student member functions.**

```
cout << pp[1] ->getName() << '\n'; // okay  
pp[0] ->print();                  // calls Person::print()  
pp[1] ->print();                  // also calls Person::print() (!)  
pp[1] ->changeMajor("English");    // ERROR!
```

The reason for this apparently anomalous behavior is called **static binding**—when determining which member function to call, C++'s default action is to consider an object's **declared type**, not its actual type. Since pp[1] is declared to be a pointer to a Person, the members for that class are used. Nonetheless, C++ provides a way to achieve the desired dynamic effect using the technique we describe next.

Dynamic Binding and Virtual Functions

As we saw above, C++ uses **static binding** by default to determine which member function to call for a derived class. Alternatively, in **dynamic binding**, an object's contents determine which member function is called. To specify that a member function should use dynamic binding, **the keyword "virtual" is added to the function's declaration**. Let us redefine our Person and Student, but this time we will declare the print function to be virtual.

```

class Person {
    virtual void print() { ... }
    // ...
};

class Student : public Person {
    virtual void print() { ... }
    // ...
};

```

Let us consider the effect of this change on our array example, thereby **illustrating the usefulness of dynamic binding**.

```

Person* pp[100];
pp[0] = new Person(...);
pp[1] = new Student(...);
pp[0]->print();           // calls Person::print()
pp[1]->print();           // calls Student::print()

```

In this case, `pp[1]` contains a pointer to an object of type `Student`, and by the power of dynamic binding with virtual functions, the function `Student::print` will be called. **The decision as to which function to call is made at run-time, hence the name **dynamic binding**.**

Virtual Destructors

There is no such thing as a virtual constructor. Such a concept does not make any sense. **Virtual destructors, however, are very important.** In our array example, since we store objects of both types `Person` and `Student` in the array, it is important that the appropriate destructor be called for each object. *However, if the destructor is nonvirtual, then only the Person destructor will be called in each case.* If the `Student` class had allocated memory dynamically, **the fact that the wrong destructor is called would result in a memory leak.**

When writing a base class, we cannot know, in general, whether a derived class may need to implement a destructor. So, to be safe, **when defining any virtual functions, it is recommended that a virtual destructor be defined as well.** This destructor may do nothing at all, and that is fine. **It is provided just in case a derived class needs to define its own destructor.** This principle is encapsulated in the following rule of thumb.

If a base class defines any virtual functions, it should define a **virtual destructor**, even if it is empty.

Dynamic binding is a powerful technique, since it allows us to create an object, such as the array `pp` above, whose behavior varies depending on its contents. **This technique is fundamental to the concept of polymorphism**, which we discuss in the next section.

- Polymorphism

Literally, "polymorphism" means "many forms." In the context of object-oriented design, it refers to the ability of a variable to take different types. **Polymorphism is typically applied in C++ using pointer variables.** In particular, a variable `p` declared to be a pointer to some class `S` implies that `p` can point to any object belonging to any derived class `T` of `S`.

Now consider what happens if both of these classes define a virtual member function `a`, and let us consider which of these functions is called when we invoke `p->a()`. Since dynamic binding is used, if `p` points to an object of type `T`, then it invokes the function `T::a`. In this case, `T` is said to **override** function `a` from `S` (**Note here we use the "virtual" keyword to override the member functions**). Alternatively, if `p` points to an object of type `S`, it will invoke `S::a`.

Polymorphism such as this is useful because the caller of `p->a()` does not have to know whether the pointer `p` refers to an instance of `T` or `S` in order to get the `a` function to execute correctly. A pointer variable `p` that points to a class object that has at least one virtual function is said to be **polymorphic**. That is, `p` can take many forms, depending on the specific class of the object it is referring to. **This kind of functionality allows a specialized class `T` to extend a class `S`, inherit the "generic" functions from class `S`, and redefine other functions from class `S` to account for specific properties of objects of class `T`.**

Inheritance, polymorphism, and function overloading support reusable software. We can define classes that inherit generic member variables and functions and can then define new, **more** specific variables and functions that deal with special aspects of objects of the new class. For example, suppose that we defined a generic class `Person` and then derived three classes `Student`, `Administrator`, and `Instructor`. We could store pointers to all these objects in a list of type `Person*`. When we invoke a virtual member function, such as `print`, to any element of the list, it will call the function appropriate to the individual element's type.

Specialization

There are two primary ways of using inheritance, one of which is **specialization**. *In using specialization, we are specializing a general class to a particular derived class* (such as override member functions). Such derived classes typically possess an "is a" relationship to their base class. The derived classes inherit all the members of the base class. For each inherited function, if that function operates correctly, independent of whether it is operating for a specialization, no additional work is needed. **If, on the other hand, a general function of the base class would not work correctly on the derived class, then we should override the function to have the correct functionality for the derived class.**

For example, we could have a general class, `Dog`, which has a function `drink` and a function `sniff`. Specializing this class to a `Bloodhound` class would probably not require that we override the `drink` function, as all dogs drink pretty much the same way. **But it could require that we override the `sniff` function**, as a `Bloodhound` has a much more sensitive sense of smell than a "generic" dog. In this way, the `Bloodhound` class specializes the functions of its base class, `Dog`.

Extension

Another way of using inheritance is **extension**. In using extension, **we reuse the code written for functions of the base class, but we then add new functions that are not present in the base class, so as to extend its functionality**. For example, returning to our `Dog` class, we might wish to create a derived class, `BorderCollie`, which inherits all the generic functions of the `Dog` class, but then adds a new function, `herd`, since Border Collies have a herding instinct that is not present in generic dogs, thereby extending the functionality of a generic dog.

- Examples of Inheritance in C++

To make the concepts of inheritance and polymorphism more concrete, let us consider a simple example in C++. We consider an example of several classes that print numeric progressions. A **numeric progression** is a sequence of numbers, where the value of each number depends on one or more of the previous values. For example, an **arithmetic progression** determines a next number by addition of a fixed increment. A **geometric progression** determines a next number by multiplication by a fixed base value. In any case, a **progression requires a way of defining its first value and it needs a way of identifying the current value as well** (This observation is used to utilize the inheritance).

Arithmetic progression (increment 1)	0, 1, 2, 3, 4, 5, ...
Arithmetic progression (increment 3)	0, 3, 6, 9, 12, ...
Geometric progression (base 2)	1, 2, 4, 8, 16, 32, ...
Geometric progression (base 3)	1, 3, 9, 27, 81, ...

We begin by defining a class, `Progression`, which is declared in the code fragment below. **It defines the "generic" members and functions of a numeric progression.** Specifically, it defines the following two long-integer variable members:

- `first`: first value of the progression
- `cur`: current value of the progression

Because we want these variables to be accessible from derived classes, **we declare them to be protected.**

We define a constructor, `Progression`, a destructor, `~Progression`, and the following three member functions.

```

firstValue(): Reset the progression to the first value and return it.
nextValue(): Step the progression to the next value and return it.
printProgression(n): Reset the progression and print its first n values.

class Progression {                                // a generic progression
public:
    Progression(long f = 0)                      // constructor
        : first(f), cur(f) { }
    virtual ~Progression() { };                  // destructor
    void printProgression(int n);                // print the first n values
protected:
    virtual long firstValue();                 // reset
    virtual long nextValue();                  // advance
protected:
    long first;                            // first value
    long cur;                             // current value
};

```

The member function `printProgression` is public and is defined below.

```

void Progression::printProgression(int n) {      // print n values
    cout << firstValue();                         // print the first
    for (int i = 2; i <= n; i++)
        cout << ', ' << nextValue();             // print 2 through n
    cout << endl;
}

```

In contrast, **the member functions `firstValue` and `nextValue` are intended as utilities** that will only be invoked from within this class or its derived classes. For this reason, we declare them to be protected. They are defined below.

```

long Progression::firstValue() {                  // reset
    cur = first;
    return cur;
}
long Progression::nextValue() {                   // advance
    return ++cur;
}

```

It is our intention that, in order to generate different progressions, derived classes will override one or both of these functions. For this reason, we have declared both to be virtual. Because there are virtual functions in our class, we have also provided a virtual destructor in order to be safe. At this point the destructor does nothing, but this might be overridden by derived classes.

Arithmetic Progression Class

Let us consider a class `ArithProgression`, shown below. We add a new member variable `inc`, which provides the value to be added to each new element of the progression. We also override the member function `nextValue` to produce the desired new behavior.

```

class ArithProgression : public Progression { // arithmetic progression
public:
    ArithProgression(long i = 1);           // constructor
protected:
    virtual long nextValue();              // advance
protected:
    long inc;                            // increment
};

```

The constructor and the new member function `nextValue` are defined below. **Observe that the constructor invokes the base class constructor `Progression` to initialize the base object in addition to initializing the value of `inc`.**

```

ArithProgression::ArithProgression(long i) // constructor
: Progression(), inc(i) { }

long ArithProgression::nextValue() {        // advance by adding
    cur += inc;
    return cur;
}

```

Polymorphism is at work here. When a `Progression` pointer is pointing to an `ArithProgression` object, it will use the `ArithProgression` functions `firstValue` and `nextValue`. **Even though the function `printProgression` is not virtual, it makes use of this polymorphism.** Its calls to the `firstValue` and `nextValue` functions are implicitly for the “current” object, which will be of the `ArithProgression` class.

A Geometric Progression Class

Let us next define `GeomProgression` that implements a geometric progression. As with the `ArithProgression` class, this new class inherits the member variables `first` and `cur`, and the member functions `firstValue` and `printProgression` from `Progression`. We add a new member variable `base`, which holds the base value to be multiplied to form each new element of the progression. The constructor initializes the base class with a starting value of 1 rather than 0. The function `nextValue` applies multiplication to obtain the next value.

```

class GeomProgression : public Progression { // geometric progression
public:
    GeomProgression(long b = 2);           // constructor
protected:
    virtual long nextValue();              // advance
protected:
    long base;                           // base value
};

GeomProgression::GeomProgression(long b) // constructor
: Progression(1), base(b) { }

long GeomProgression::nextValue() {        // advance by multiplying
    cur *= base;
    return cur;
}

```

A Fibonacci Progression Class

As a further example, we define a `FibonacciProgression` class that represents another kind of progression, the **Fibonacci progression**, where **the next value is defined as the sum of the current and previous values**. We show the `FibonacciProgression` class below. Recall that each

element of a Fibonacci series is the sum of the previous two elements.

Fibonacci progression (first = 0, second = 1): 0,1,1,2,3,5,8,...

In addition to the current value *cur* in the Progression base class, we also store here the value of the previous element, denoted *prev*. The constructor is given the first two elements of the sequence. The member variable *first* is inherited from the base class. We add a new member variable *second*, to store this second element. The default values for the first and second elements are 0 and 1, respectively.

```
class FibonacciProgression : public Progression { // Fibonacci progression
public:
    FibonacciProgression(long f = 0, long s = 1); // constructor
protected:
    virtual long firstValue(); // reset
    virtual long nextValue(); // advance
protected:
    long second; // second value
    long prev; // previous value
};
```

The initialization process is a bit tricky because we need to create a "fictitious" element that precedes the first element. Note that setting this element to the value *second - first* achieves the desired result. This change is reflected both in the constructor and the overridden member function *firstValue*. The overridden member function *nextValue* copies the current value to the previous value. We need to store the old previous value in a temporary variable.

```
FibonacciProgression::FibonacciProgression(long f, long s)
    : Progression(f), second(s), prev(second - first) {}

long FibonacciProgression::firstValue() { // reset
    cur = first;
    prev = second - first; // create fictitious prev
    return cur;
}

long FibonacciProgression::nextValue() { // advance
    long temp = prev;
    prev = cur;
    cur += temp;
    return cur;
}
```

Combining the Progression Classes

In order to visualize how the three different progression classes are derived from the generic Progression class, we give their **inheritance diagram** in Figure 2.5.

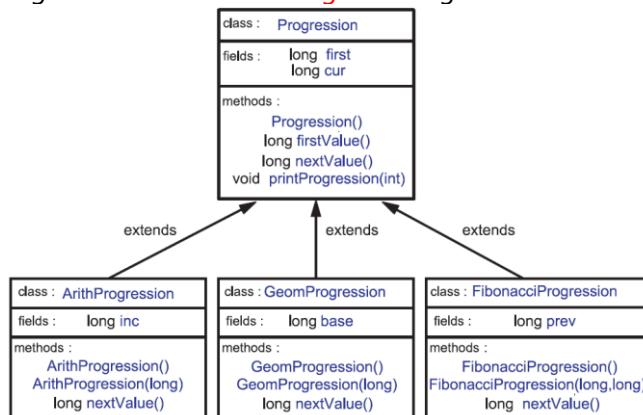


Figure 2.5 Inheritance diagram for class Progression and its subclasses.

● Multiple Inheritance and Class Casting

In C++, we are allowed to derive a class from a number of base classes, that is, C++ allows **multiple inheritance**. Although multiple inheritance can be useful, especially in defining

interfaces, it introduces a number of complexities. For example, if both base classes provide a member variable with the same name or a member function with the same declaration, the derived class must specify from which base class the member should be used (which is complicated).

We have been using public inheritance in our previous examples, indicated by the keyword **public** in specifying the base class. Remember that private base class members are not accessible in a derived class. Protected and public members of the base class become protected and public members of the derived class, respectively. C++ supports two other types of inheritance. These different types of inheritance diminish the access rights for base class members. In **protected inheritance**, fields declared to be public in the base class become protected in the child class. In **private inheritance**, fields declared to be public and protected in the base class become private in the derived class (In all cases, the derived class can't access the private members of base class).

```
class Base {                                // base class
    protected: int foo;
    public:    int bar;
};

class Derive1 : public Base {                // public inheritance
    // foo is protected and bar is public
};

class Derive2 : protected Base {             // protected inheritance
    // both foo and bar are protected
};

class Derive3 : private Base {              // public inheritance
    // both foo and bar are private
};
```

Protected and private inheritance are not used as often as public inheritance.

Casting in an Inheritance Hierarchy

An object variable can be viewed as being of various types, but it can be declared as only one type. Thus, a variable's declared type determines how it is used, and even determines how certain functions will act on it. Enforcing that all variables be typed and that operations declare the types they expect is called **strong typing**, which helps prevent bugs. Nonetheless, we sometimes need to explicitly change, or **cast**, a variable from one type to another.

To illustrate an example where we may want to perform a cast, recall our class hierarchy consisting of a base class Person and derived class Student. Suppose that we are storing pointers to objects of both types in an array *pp*. The following attempt to change a student's major would be flagged as an error by the compiler.

```
Person* pp[100];                      // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);              // add a Student (details omitted)
// ...
pp[1]->changeMajor("English");       // ERROR!
```

The problem is that the base class Person does not have a function **changeMajor**. Notice that this is different from the case of the function **print** because the **print** function was provided in both classes. Nonetheless, we "know" that *pp[1]* points to an object of class **Student**, so this operation should be legal.

To access the **changeMajor** function, we need to cast the *pp[1]* pointer from type **Person*** to

type `Student*`. Because the contents of a variable are dynamic, we need to use the C++ run-time system to determine whether this cast is legal, which is what a **dynamic cast** does. The syntax of a dynamic cast is shown below.

```
dynamic_cast<desired_type>( expression )
```

Dynamic casting can only be applied to polymorphic objects, that is, objects that come from a class with at least one virtual function (Because the polymorphic base class contains run-time type information). Below we show how to use dynamic casting to change the major of `pp[1]`.

```
Student* sp = dynamic_cast<Student*>(pp[1]); // cast pp[1] to Student*
sp->changeMajor("Chemistry"); // now changeMajor is legal
```

Dynamic casting is most often applied for casting pointers within the class hierarchy. **If an illegal pointer cast is attempted, then the result is a null pointer.** For example, we would get a NULL pointer from an attempt to cast `pp[0]` as above, since it points to a `Person` object.

To illustrate the use of dynamic cast, we access all the elements of the `pp` array and, for objects of (actual) type `Student`, change the major to "Undecided"

```
for (int i = 0; i < 100; i++) {
    Student *sp = dynamic_cast<Student*>(pp[i]);
    if (sp != NULL) // cast succeeded?
        sp->changeMajor("Undecided"); // change major
}
```

The casting we have discussed here could also have been done using the traditional C-style cast or through a static cast. **Unfortunately, no error checking would be performed in that case.** An attempt to cast a `Person` object pointer to a `Student` pointer would succeed "silently," but any attempt to use such a pointer would have disastrous consequences.

● Interfaces and Abstract Classes

For two objects to interact, they must "know" about each other's member functions. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the **application programming interface** (API), or simply **interface**, that their objects present to other objects. In the **ADT-based** approach to data structures followed in this book, an interface defining an ADT is specified as a type definition and a collection of member functions for this type, with the arguments for each function being of specified types.

Some programming languages provide a mechanism for defining ADTs. One example is Java's interface. **An interface is a collection of function declarations with no data and no bodies.** That is, the member functions of an interface are always empty. **When a class implements an interface, it must implement all of the member functions declared in the interface.**

C++ does not provide a direct mechanism for specifying interfaces. Nonetheless, throughout this book we often provide **informal interfaces**, even though they are not legal C++ structures. For example, a **stack** data structure is a container that supports various operations such as inserting (or **pushing**) an element onto the top of the stack, removing (or **popping**) an element from the top of the stack, and testing whether the stack is empty. Below we provide an example of a minimal interface for a stack of integers.

```
class Stack { // informal interface – not a class
public:
    bool isEmpty() const; // is the stack empty?
    void push(int x); // push x onto the stack
    int pop(); // pop the stack and return result
};
```

Abstract Classes

The above informal interface is **not** a valid construct in C++; it is just a documentation aid. In particular, it does not contain any data members or definitions of member functions. **Nonetheless, it is useful**, since it provides important information about a stack's public member functions and how they are called.

An **abstract class** in C++ is a class that is used only as a base class for inheritance; it cannot be used to create instances directly. **At first the idea of creating a class that cannot be instantiated seems to be nonsense, but it is often very important.** For example, suppose that we want to define a set of geometric shape classes, say, Circle, Rectangle, and Triangle. It is natural to derive these related classes from a single generic base class, say, Shape. Each of the derived classes will have a virtual member function draw, which draws the associated object. **The rules of inheritance require that we define such a function for the base class, but it is unclear what such a function means for a generic shape.**

One way to handle this would be to define empty function body ({ }), which would be a rather unnatural solution. What is really desired here is some way to inform the compiler that the class Shape is **abstract**; it is not possible to create objects of type Shape, only its subclasses. **In C++, we define a class as being abstract by specifying that one or more members of its functions are abstract, or pure virtual.** A function is declared pure virtual by giving "=0" in place of its body. **C++ does not allow the creation of an object that has one or more pure virtual functions.** Thus, any derived class must provide concrete definitions for all pure virtual functions of the base class.

As an example, recall our Progression class and consider the member function nextValue, which computes the next value in the progression. The meaning of this function is clear for each of the derived classes: ArithProgression, GeomProgression, and FibonacciProgression. However, in the base class Progression we invented a rather arbitrary default for the nextValue function. It would be more natural to leave this function undefined. We show below how to make it a **pure virtual** member function.

```
class Progression {           // abstract base class
// ...
virtual long nextValue() = 0; // pure virtual function
// ...
};
```

As a result, the compiler will not allow the creation of objects of type Progression, since the function nextValue is "pure virtual." However, its derived classes, ArithProgression for example, can be defined because they provide a definition for this member function.

Interfaces and Abstract Base Classes

We said above that C++ does not provide a direct mechanism for defining interfaces for abstract data types (ADTs). Nevertheless, **we can use abstract base classes to achieve much of the same purpose.**

In particular, we may construct a class for an interface in which all the functions are pure virtual as shown below for the example of a simple stack ADT.

```
class Stack {           // stack interface as an abstract class
public:
    virtual bool isEmpty() const = 0; // is the stack empty?
    virtual void push(int x) = 0;     // push x onto the stack
    virtual int pop() = 0;           // pop the stack and return result
};
```

A class that implements this stack interface can be derived from this abstract base class, and

then provide concrete definitions for all of these virtual functions as shown below.

```
class ConcreteStack : public Stack { // implements Stack
public:
    virtual bool isEmpty() { ... }      // implementation of members
    virtual void push(int x) { ... }    // ... (details omitted)
    virtual int pop() { ... }
private:
    // ...
};
```

There are practical limitations to this method of defining interfaces, so we only use informal interfaces for the purpose of illustrating ADTs.

2.3. Templates

Inheritance is only one mechanism that C++ provides in support of polymorphism. In this section, we consider another way—using **templates**.

- Function Templates

Let us consider the following function, which returns the minimum of two integers.

```
int integerMin(int a, int b)           // returns the minimum of a and b
{ return (a < b ? a : b); }
```

Such a function is very handy, so we might like to define a similar function for computing the minimum of two variables of other types, such as long, short, float, and double. **Each such function would require a different declaration and definition, however, and making many copies of the same function is an error-prone solution, especially for longer functions.**

C++ provides an automatic mechanism, called the **function template**, to produce a generic function for an arbitrary type T. A function template provides a well-defined pattern from which a concrete function may later be formally defined or **instantiated**. The example below defines a genericMin function template.

```
template <typename T>
T genericMin(T a, T b) {           // returns the minimum of a and b
    return (a < b ? a : b);
}
```

The declaration takes the form of the keyword "**template**" followed by the notation `<typename T>`, which is the parameter list for the template. In this case, there is just one parameter T. The keyword "typename" indicates that T is the name of some type. (Older versions of C++ do not support this keyword and instead the keyword "**class**" must be used.) We can have other types of template parameters, integers for example, but type names are the most common. Observe that the type parameter T takes the place of "int" in the original definition of the genericMin function.

We can now invoke our templated function to compute the minimum of objects of many different types. *The compiler looks at the argument types and determines which form of the function to instantiate.*

```
cout << genericMin(3, 4) << ', ' // = genericMin<int>(3,4)
<< genericMin(1.1, 3.1) << ', ' // = genericMin<double>(1.1, 3.1)
<< genericMin('t', 'g') << endl; // = genericMin<char>('t','g')
```

The template type does not need to be a fundamental type. We could use any type in this example, provided that the less than operator (<) is defined for this type.

- Class Templates

In addition to function templates, C++ allows classes to be templated, which is a powerful mechanism because it allows us to provide one data structure declaration that can be applied to many different types. In fact, the Standard Template Library uses class templates extensively.

Let us consider an example of a template for a restricted class BasicVector that stores a vector of elements, which is a simplified version of a structure discussed in greater detail in Chapter 6. This class has a constructor that is given the size of the array to allocate. In order to access elements of the array, we overload the indexing operator "[]".

We present a partial implementation of a class template for class BasicVector below. We have omitted many of the other member functions, such as the copy constructor, assignment operator, and destructor. The template parameter T takes the place of the actual type that will be stored in the array.

```
template <typename T>
class BasicVector {                      // a simple vector class
public:
    BasicVector(int capac = 10);          // constructor
    T& operator[](int i)                 // access element at index i
    { return a[i]; }
    // ... other public members omitted
private:
    T* a;                                // array storing the elements
    int capacity;                         // length of array a
};
```

We have defined one member function (the indexing operator) within the class body, and below we show how the other member function (the constructor) can be defined outside the class body. The constructor initializes the capacity value and allocates the array storage.

```
template <typename T>                  // constructor
BasicVector<T>::BasicVector(int capac) {
    capacity = capac;
    a = new T[capacity];                // allocate array storage
}
```

To **instantiate** a concrete instance of the class BasicVector, we provide the class name followed by the actual type parameter enclosed in angled brackets (< ... >). The code fragment below shows how we would define three vectors, one of type int, one of type double, and one of type string.

```
BasicVector<int>      iv(5);           // vector of 5 integers
BasicVector<double>    dv(20);          // vector of 20 doubles
BasicVector<string>    sv(10);          // vector of 10 strings
```

Since we have overloaded the indexing operator, we can access elements of each array in the same manner as we would for any C++ array.

```
iv[3] = 8;
dv[14] = 2.5;
sv[7] = "hello";
```

Templated Arguments

The actual argument in the instantiation of a class template can itself be a templated type. For example, we could create a BasicVector whose individual elements are themselves of type BasicVector<int>.

```
BasicVector<BasicVector<int>> xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

In this case, because no capacity argument could be provided to the constructor, each element of the vector is constructed using the default capacity of 10. Thus the above definition declares a BasicVector consisting of five elements, each of which is a BasicVector consisting of 10 integers. **Such a structure therefore behaves much like a two-dimensional array of integers.**

Note that in the declaration of `xv` above, we intentionally left a space after "`<int>`." The reason is that without the space, the character combination "`>>`" would be interpreted as a bitwise right-shift operator by the compiler.

2.4. Exceptions

Exceptions are unexpected events that occur during the execution of a program. An exception can be the result of an error condition or simply an unanticipated input. **In C++, exceptions can be thought of as being objects themselves.**

- Exception Objects

In C++, an exception is "thrown" by code that encounters some unexpected condition. Exceptions can also be thrown by the C++ run-time environment should it encounter an unexpected condition like running out of memory. A thrown exception is "caught" by other code that "handles" the exception somehow, or the program is terminated unexpectedly.

Exceptions are a relatively recent addition to C++. Prior to having exceptions, errors were typically handled by having the program abort at the source of the error or by having the involved function return some special value. **Exceptions provide a much cleaner mechanism for handling errors.** Nevertheless, for historical reasons, many of the functions in the C++ standard library do not throw exceptions. Typically they return some sort of special error status, or set an error flag, which can be tested.

Exceptions are thrown when a piece of code finds some sort of problem during execution. Since there are many types of possible errors, when an exception is thrown, it is identified by a type. Typically this type is a class whose members provide information as to the exact nature of the error, for example a string containing a descriptive error message.

Exception types often form hierarchies. For example, let's imagine a hypothetical mathematics library, which may generate many different types of errors. The library might begin by defining one generic exception, MathException, representing all types of mathematical errors, and then derive more specific exceptions for particular error conditions. The errMsg member holds a message string with an informative message. Here is a possible definition of this generic class.

```
class MathException {                                // generic math exception
public:
    MathException(const string& err)                // constructor
        : errMsg(err) { }
    string getError() { return errMsg; }             // access error message
private:
    string errMsg;                                // error message
};
```

Using Inheritance to Define New Exception Types

The above MathException class would likely have other member functions, for example, for accessing the error message. We may then add more specific exceptions, such as ZeroDivide, to handle division by zero, and NegativeRoot, to handle attempts to compute the square root of a negative number. We could use class inheritance to represent this hierarchical relationship, as follows.

```

class ZeroDivide : public MathException {
public:
    ZeroDivide(const string& err)           // divide by zero
    : MathException(err) { }
};

class NegativeRoot : public MathException {
public:
    NegativeRoot(const string& err)         // negative square root
    : MathException(err) { }
};

```

- Throwing and Catching Exceptions

Exceptions are typically processed in the context of "try" and "catch" blocks. A **try block** is a block of statements proceeded by the keyword **try**. After a try block, there are one or more **catch blocks**. Each catch block specifies the type of exception that it catches. Execution begins with the statements of the try block. If all goes smoothly, then execution leaves the try block and skips over its associated catch blocks. If an exception is thrown, then the control immediately jumps into the appropriate catch block for this exception.

For example, suppose that we were to use our mathematical library as part of the implementation of a numerical application. We would enclose the computations of the application within a **try block**. After the try block, we would catch and deal with any exceptions that arose in the computation.

```

try {
    // ... application computations
    if (divisor == 0)                      // attempt to divide by 0?
        throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
    // handle division by zero
}
catch (MathException& me) {
    // handle any math exception other than division by zero
}

```

Processing the above try block is done as follows. The computations of the try block are executed. When an attempt is discovered to divide by zero, **ZeroDivide** is thrown, and execution jumps immediately to the associated catch statement where corrective recovery and clean up should be performed.

Let us study the entire process in somewhat greater detail. The **throw** statement is typically written as follows:

throw exception-name(arg1,arg2,...)

where the arguments are passed to the exception's constructor.

Exceptions may also be thrown by the C++ run-time system itself. For example, if an attempt to allocate space in the free store using the **new** operator fails due to lack of space, then a **bad_alloc** exception is thrown by the system.

When an exception is thrown, it must be **caught** or the program will abort. In any particular function, an exception in that function can be passed through to the calling function or it can be caught in that function. When an exception is caught, it can be analyzed and dealt with. The general syntax for a **try-catch block** in C++ is as follows:

```

try
    try_statements
catch ( exception_type_1 identifier_1 )
    catch_statements_1
...
catch ( exception_type_n identifier_n )
    catch_statements_n

```

Execution begins in the "try statements." If this execution generates no exceptions, then the flow of control continues with the first statement after the last line of the entire try-catch block. If, on the other hand, an exception is generated, execution in the try block terminates at that point and execution jumps to the first catch block matching the exception thrown. Thus, **an exception thrown for a derived class will be caught by its base class**. For example, if we had thrown NegativeRoot in the example above, it would be caught by catch block for MathException. Note that because the system executes the first matching catch block, exceptions should be listed in order of most specific to least specific. **The special form "catch(...)" catches all exceptions.**

The "identifier" for the catch statement identifies the exception object itself. As is common in passing class arguments, the exception is typically passed as a reference or a constant reference. Once execution of the catch block completes, control flow continues with the first statement after the last catch block.

The recovery action taken in a catch block depends very much on the particular application. It may be as simple as printing an error message and terminating the program. It may require complex clean-up operations, such as deallocating dynamically allocated storage and restoring the program's internal state. There are also some interesting cases in which the best way to handle an exception is to ignore it (which can be specified by having an empty catch block). Ignoring an exception is usually done, for example, when the programmer does not care whether there was an exception or not. Another legitimate way of handling exceptions is to throw another exception, possibly one that specifies the exceptional condition more precisely.

- Exception Specification

When we declare a function, we should also specify the exceptions it might throw (Dynamic exception specification (until C++17) lists the exceptions that a function might directly or indirectly throw). This convention has both a functional and courteous purpose. For one, it lets users know what to expect. It also lets the compiler know which exceptions to prepare for. The following is an example of such a function definition.

```

void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}

```

This definition indicates that the function calculator (and any other functions it calls) can throw these two exceptions or exceptions derived from these types, but no others.

By specifying all the exceptions that might be thrown by a function, we prepare others to be able to handle all of the exceptional cases that might arise from using this function. Another benefit of declaring exceptions is that we do not need to catch those exceptions in our function, which is appropriate, for example, in the case where other code is responsible for causing the circumstances leading up to the exception.

The following illustrates an exception that is "passed through."

```

void getReadyForClass() throw(ShoppingListTooSmallException,
                           OutOfMoneyException) {
    goShopping(); // I don't have to try or catch the exceptions
                  // which goShopping() might throw because
                  // getReadyForClass() will just pass these along.
    makeCookiesForTA();
}

```

A function can declare that it throws as many exceptions as it likes. Such a listing can be simplified somewhat if all exceptions that can be thrown are derived classes of the same exception. In this case, we only have to declare that a function throws the appropriate base class.

Suppose that a function does not contain a **throw** specification. It would be natural to assume that such a function does not throw any exceptions. In fact, it has quite a different meaning. If a function does not provide a **throw** specification, then it may throw any exception. Although this is confusing, it is necessary to maintain compatibility with older versions of C++. To indicate that a function throws no exceptions, provide the **throw** specifier with an empty list of exceptions.

<code>void func1();</code>	<code>// can throw any exception</code>
<code>void func2() throw();</code>	<code>// can throw no exceptions</code>

Generic Exception Class

We declare many different exceptions in this book. In order to structure these exceptions hierarchically, we need to have one generic exception class that serves as the "mother of all exceptions." C++ does not provide such a generic exception, so we created one of our own. This class, called `RuntimeException`, is shown below. It has an error message as its only member. It provides a constructor that is given an informative error message as its argument. It also provides a member function `getMessage` that allows us to access this message.

```

class RuntimeException {           // generic run-time exception
private:
    string errorMsg;
public:
    RuntimeException(const string& err) { errorMsg = err; }
    string getMessage() const { return errorMsg; }
};

```

By deriving all of our exceptions from this base class, for any exception *e*, we can output *e*'s error message by invoking the inherited `getMessage` function.

CHAPTER 3: Arrays, Linked Lists, and Recursion

3.1 Using Arrays

In this section, we explore a few applications of arrays—the concrete data structures introduced in previous section (1.1) that access their entries using integer indices.

● Storing Game Entries in an Array

The first application we study is for storing entries in an array; in particular, high score entries for a video game. *Storing objects in arrays is a common use for arrays*, and we could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data structuring concepts.

Let us begin by thinking about what we want to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we call *score*. Another useful thing to include is the name of the person earning this score, which we simply call *name*. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. Let us keep our example simple, however, and just have two fields, *score* and *name*:

```
class GameEntry {                                // a game score entry
public:
    GameEntry(const string& n="", int s=0); // constructor
    string getName() const;                // get player name
    int getScore() const;                  // get score
private:
    string name;                         // player's name
    int score;                           // player's score
};
```

Also, we provide the definitions of the class constructor and two accessor member functions:

```
GameEntry::GameEntry(const string& n, int s) // constructor
: name(n), score(s) {}                      // accessors
string GameEntry::getName() const { return name; }
int GameEntry::getScore() const { return score; }
```

A Class for High Scores

Let's now design a class, called *Scores*, to store our game-score information. We store the highest scores in an array *entries*. The maximum number of scores may vary from instance to instance, so we create a member variable, *maxEntries*, storing the desired maximum. Its value is specified when a *Scores* object is first constructed. **In order to keep track of the actual number of entries**, we define a member variable *numEntries*. It is initialized to zero, and it is updated as entries are added or removed. We provide a constructor, a destructor, a member function for adding a new score, and one for removing a score at a given index:

```
class Scores {                                // stores game high scores
public:
    Scores(int maxEnt = 10);                 // constructor
    ~Scores();                               // destructor
    void add(const GameEntry& e);           // add a game entry
    GameEntry remove(int i);                 // remove the ith entry
    throw(IndexOutOfBoundsException);
private:
    int maxEntries;                        // maximum number of entries
    int numEntries;                         // actual number of entries
    GameEntry* entries;                    // array of game entries
};
```

Also, we present the class constructor, which allocates the desired amount of storage for the array using the "new" operator. Recall from the previous section (1.1) that C++ represents a dynamic array as a pointer to its first element, and this command returns such a pointer. The class destructor, `~Scores`, deletes this array.

```

Scores::Scores(int maxEnt) {
    maxEntries = maxEnt;
    entries = new GameEntry[maxEntries];
    numEntries = 0;
}

Scores::~Scores() {
    delete[] entries;
}

```

The entries that have been added to the array are stored in indices 0 through $numEntries - 1$. As more users play our video game, additional `GameEntry` objects are copied into the array. This is done using the class's `add` member function, which we describe below. **Only the highest `maxEntries` scores are retained**. We also provide a member function, `remove(i)`, which removes the entry at index i from the array. We assume that $0 \leq i \leq numEntries - 1$. If not, the `remove` function, throws an `IndexOutOfBoundsException` exception. We do not define this exception here, but it is derived from the class `RuntimeException` from the previous section (2.4).

In our design, we have chosen to order the `GameEntry` objects by their score values, from highest to lowest. We illustrate an example of the data structure in Figure 3.1.

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

Figure 3.1 The `entries` array of length eight storing six `GameEntry` objects in the cells from index 0 to 5. Here `maxEntries` is 10 and `numEntries` is 6.

Insertion

Next, let us consider how to add a new `GameEntry` e to the array of high scores. In particular, let us consider how we might perform the following update operation on an instance of the `Scores` class.

`add(e)`: Insert game entry e into the collection of high scores. If this causes the number of entries to exceed `maxEntries`, the smallest is removed.

The approach is to shift all the entries of the array whose scores are smaller than e 's score to the right, in order to make space for the new entry. (See Figure 3.2.)

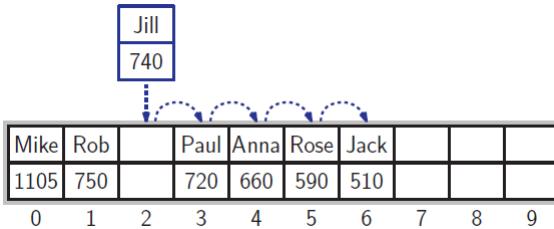


Figure 3.2 Preparing to add a new `GameEntry` object ("Jill", 740) to the `entries` array. In order to make room for the new entry, we shift all the entries with smaller scores to the right by one position.

Once we have identified the position in the `entries` array where the new game entry, e , belongs, we copy e into this position. (See Figure 3.3.)

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			

Figure 3.3 After adding the new entry at index 2.

The details of our algorithm for adding the new game entry e to the entries array are similar to this informal description and are given below. First, we consider whether the array is already full. If so, we check whether the score of the last entry in the array (which is at $\text{entries}[\maxEntries - 1]$) is at least as large as e 's score. If so, we can return immediately since e is not high enough to replace any of the existing highest scores. If the array is not yet full, we know that one new entry will be added, so we increment the value of numEntries . Next, we identify all the entries whose scores are smaller than e 's and shift them one entry to the right. To avoid overwriting existing array entries, we start from the right end of the array and work to the left. The loop continues until we encounter an entry whose score is not smaller than e 's, or we fall off the front end of the array. In either case, the new entry is added at index $i + 1$.

```

void Scores::add(const GameEntry& e) { // add a game entry
    int newScore = e.getScore(); // score to add
    if (numEntries == maxEntries) { // the array is full
        if (newScore <= entries[maxEntries-1].getScore())
            return; // not high enough - ignore
    }
    else numEntries++; // if not full, one more entry

    int i = numEntries-2; // start with the next to last
    while ( i >= 0 && newScore > entries[i].getScore() ) {
        entries[i+1] = entries[i]; // shift right if smaller
        i--;
    }
    entries[i+1] = e; // put e in the empty spot
}

```

Check the code carefully to see that all the limiting cases have been handled correctly by the add function (for example, largest score, smallest score, empty array, full array). **The number of times we perform the loop in this function depends on the number of entries that we need to shift. This is pretty fast if the number of entries is small. But if there are a lot to move, then this method could be fairly slow.**

Object Removal

Suppose some hot shot plays our video game and gets his or her name on our high score list. In this case, we might want to have a function that lets us remove a game entry from the list of high scores. Therefore, let us consider how we might remove a GameEntry object from the entries array. That is, let us consider how we might implement the following operation:

remove(i): Remove and return the game entry e at index i in the entries array. If index i is outside the bounds of the entries array, then this function throws an exception; otherwise, the entries array is updated to remove the object at index i and all objects previously stored at indices higher than i are “shifted left” to fill in for the removed object.

Our implementation of remove is similar to that of add, but in reverse. To remove the entry at index i , we start at index i and move all the entries at indices higher than i one position to the left. (See *Figure 3.4*.)

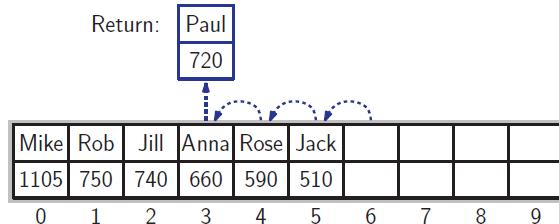


Figure 3.4 Removal of the entry ("Paul", 720) at index 3.

The code for performing the removal is presented below:

```
GameEntry Scores::remove(int i) throw(IndexOutOfBoundsException) {
    if ((i < 0) || (i >= numEntries))           // invalid index
        throw IndexOutOfBoundsException("Invalid index");
    GameEntry e = entries[i];                      // save the removed object
    for (int j = i+1; j < numEntries; j++)
        entries[j-1] = entries[j];                // shift entries left
    numEntries--;                                // one fewer entry
    return e;                                     // return the removed object
}
```

The removal operation involves a few subtle points. In order to return the value of the removed game entry (let's call it *e*), we must first save *e* in a temporary variable. When we are done, the function will return this value. The shifting process starts at the position just following the removal, *j* = *i* + 1. We repeatedly copy the entry at index *j* to index *j* – 1, and then increment *j*, until coming to the last element of the set. **Similar to the case of insertion, this left-to-right order is essential to avoid overwriting existing entries.** To complete the function, we return a copy of the removed entry that was saved in *e*.

These functions for adding and removing objects in an array of high scores are simple.

Nevertheless, they form the basis of techniques that are used repeatedly to build more sophisticated data structures.

These other structures may be more general than our simple array-based solution, and they may support many more operations. But studying the concrete array data structure, as we are doing now, is a great starting point for understanding these more sophisticated structures, **since every data structure has to be implemented using concrete means.**

● Sorting an Array

In the previous subsection, we worked hard to show how we can add or remove objects at a certain index *i* in an array while keeping the previous order of the objects intact. In this section, we consider how to rearrange objects of an array that are ordered arbitrarily in ascending order. This is known as **sorting**.

We study several sorting algorithms in this book, most of which appear in Chapter 11. As a warmup, we describe a simple sorting algorithm called **insertion-sort**. In this case, we describe a specific version of the algorithm where the input is an array of comparable elements. We consider more general kinds of sorting algorithms later in this book.

We begin with a high-level outline of the insertion-sort algorithm. We start with the first element in the array. One element by itself is already sorted. Then we consider the next element in the array. If it is smaller than the first, we swap them. Next we consider the third element in the array. We swap it leftward until it is in its proper order with the first two elements. We continue in this manner with each element of the array, swapping it leftward until it is in its proper position.

It is easy to see why this algorithm is called "insertion-sort"—**each iteration of the algorithm inserts the next element into the current sorted part of the array, which was previously the subarray in front of that element**. We may implement the above outline using two nested loops. The outer loop considers each element in the array in turn, and the inner loop moves that element to its proper location with the (sorted) subarray of elements that are to its left. We illustrate the resulting algorithm below:

```
Algorithm InsertionSort( $A$ ):
Input: An array  $A$  of  $n$  comparable elements
Output: The array  $A$  with elements rearranged in nondecreasing order
for  $i \leftarrow 1$  to  $n - 1$  do
    {Insert  $A[i]$  at its proper location in  $A[0], A[1], \dots, A[i - 1]$ }
     $cur \leftarrow A[i]$ 
     $j \leftarrow i - 1$ 
    while  $j \geq 0$  and  $A[j] > cur$  do
         $A[j + 1] \leftarrow A[j]$ 
         $j \leftarrow j - 1$ 
     $A[j + 1] \leftarrow cur$  { $cur$  is now in the right place}
void insertionSort( $\text{char}^* A$ ,  $\text{int } n$ ) {
    for ( $\text{int } i = 1$ ;  $i < n$ ;  $i++$ ) {
         $\text{char } cur = A[i];$  // sort an array of  $n$  characters
        int  $j = i - 1;$  // insertion loop
        while (( $j \geq 0$ ) && ( $A[j] > cur$ )) { // current character to insert
             $A[j + 1] = A[j];$  // start at previous character
             $j--;$  // while  $A[j]$  is out of order
             $A[j + 1] = cur;$  // move  $A[j]$  right
        }
    }
}
```

This description is already quite close to actual C++ code. It indicates which temporary variables are needed, how the loops are structured, and what decisions need to be made. We illustrate an example run in Figure 3.5.

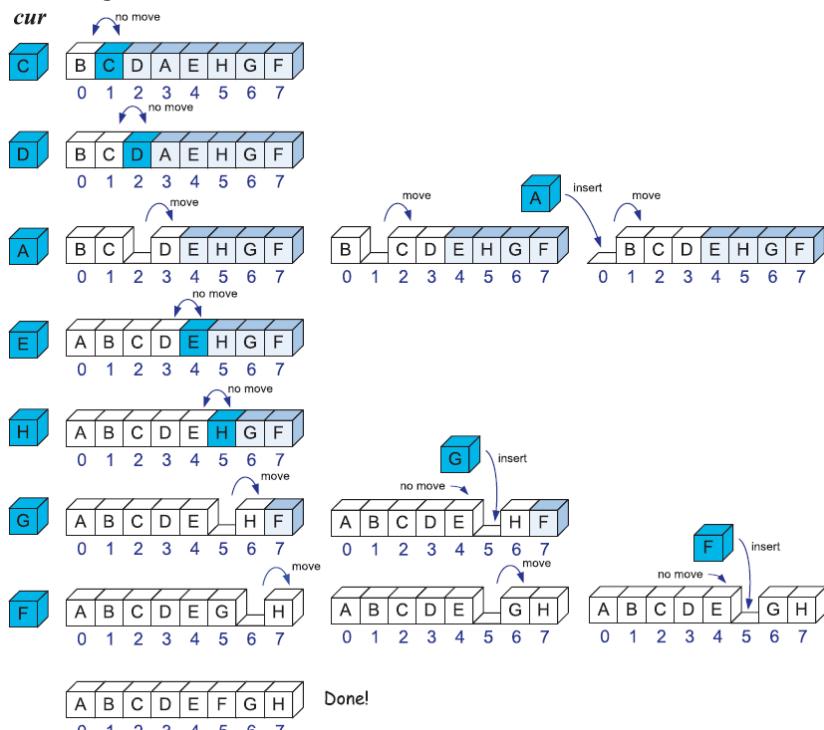


Figure 3.5 Execution of the insertion-sort algorithm on an array of eight characters. We show the completed (sorted) part of the array in white, and we color the next element that is being

inserted into the sorted part of the array with light blue. We also highlight the character on the left, since it is stored in the *cur* variable. Each row corresponds to an iteration of the outer loop, and each copy of the array in a row corresponds to an iteration of the inner loop.

Each comparison is shown with an arc. In addition, we indicate whether that comparison resulted in a move or not.

We present C++ code for our insertion-sort algorithm below. We assume that the array to be sorted consists of elements of type **char**, but it is easy to generalize this to other data types. The array *A* in the algorithm is implemented as a **char** array. Recall that each array in C++ is represented as a pointer to its first element, so the parameter *A* is declared to be of type **char***. We also pass the size of the array in an integer parameter *n*. The rest is a straightforward translation of the description given in code fragment above into C++ syntax.

```
void insertionSort(char* A, int n) {           // sort an array of n characters
    for (int i = 1; i < n; i++) {                // insertion loop
        char cur = A[i];                         // current character to insert
        int j = i - 1;                            // start at previous character
        while ((j >= 0) && (A[j] > cur)) {        // while A[j] is out of order
            A[j + 1] = A[j];                      // move A[j] right
            j--;
        }
        A[j + 1] = cur;                          // this is the proper place for cur
    }
}
```

An interesting thing happens in the insertion-sort algorithm if the array is already sorted. In this case, the inner loop does only one comparison, determines that there is no swap needed, and returns back to the outer loop. Of course, we might have to do a lot more work than this if the input array is extremely out of order. Indeed, **the worst case arises if the initial array is given in descending order.**

● Two-Dimensional Arrays and Positional Games

Many computer games, be they strategy games, simulation games, or first-person conflict games, use a two-dimensional "board." **Programs that deal with such positional games need a way of representing objects in a two-dimensional space.** A natural way to do this is with a **two-dimensional array**, where we use two indices, say *i* and *j*, to refer to the cells in the array. *The first index usually refers to a row number and the second to a column number.* Given such an array we can then maintain two-dimensional game boards, as well as perform other kinds of computations involving data that is stored in rows and columns.

Arrays in C++ are **one-dimensional**; we use a single index to access each cell of an array. **Nevertheless, there is a way we can define two-dimensional arrays in C++—we can create a two-dimensional array as an array of arrays.** That is, we can define a two-dimensional array to be an array with each of its cells being another array. Such a two-dimensional array is sometimes also called a **matrix**. In C++, we declare a two-dimensional array as follows:

```
int M[8][10];           // matrix with 8 rows and 10 columns
```

This statement creates a two-dimensional "array of arrays," *M*, which is 8×10 , having 8 rows and 10 columns. That is, *M* is an array of length 8 such that each element of *M* is an array of length 10 of integers. (See Figure 3.6.)

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 3.6 A two-dimensional integer array that has 8 rows and 10 columns. The value of $M[3][5]$ is 100 and the value of $M[6][2]$ is 632

Given integer variables i and j , we could output the element of row i and column j (or equivalently, the j th element of the i th array) as follows:

```
cout << M[i][j]; // output element in row i column j
```

It is often a good idea to use symbolic constants to define the dimensions in order to make your intentions clearer to someone reading your program.

```
const int N_DAYS = 7;
const int N_HOURS = 24;
int schedule[N_DAYS][N_HOURS];
```

Dynamic Allocation of Matrices

If the dimensions of a two-dimensional array are not known in advance, it is necessary to allocate the array dynamically. This can be done by applying the method that we discussed earlier for allocating arrays in previous section (1.1), but instead, we need to apply it to each individual row of the matrix.

For example, suppose that we wish to allocate an integer matrix with n rows and m columns. Each row of the matrix is an array of integers of length m . Recall that a dynamic array is represented as a pointer to its first element, so each row would be declared to be of type **int***. How do we group the individual rows together to form the matrix? The matrix is an array of row pointers. Since each row pointer is of type **int***, the matrix is of type **int****, that is, a pointer to a pointer of integers.

To generate our matrix, we first declare M to be of this type and allocate the n row pointers with the command "M = **new int*[n]**." The i th row of the matrix is allocated with the statement " $M[i] = **new int[m]**$." Below, we show how to do this given two integer variables n and m :

```
int** M = new int*[n]; // allocate an array of row pointers
for (int i = 0; i < n; i++)
    M[i] = new int[m]; // allocate the i-th row
```

Once allocated, we can access its elements just as before, for example, as " $M[i][j]$." As shown below, deallocating the matrix involves reversing these steps. First, we deallocate each of the rows, one by one. We then deallocate the array of row pointers. Since we are deleting an array, we use the command "**delete[]**".

```
for (int i = 0; i < n; i++)
    delete[] M[i]; // delete the i-th row
delete[] M; // delete the array of row pointers
```

Using STL Vectors to Implement Matrices

As we can see from the previous section, dynamic allocation of matrices is rather cumbersome. The STL vector class provides a much more elegant way to process matrices. We adapt the same approach as above by implementing a matrix as a vector of vectors. Each row of our matrix is

declared as "vector<int>." Thus, the entire matrix is declared to be a vector of rows, that is, "vector<vector<int>>." Let us declare M to be of this type.

Letting n denote the desired number of rows in the matrix, the constructor call $M(n)$ allocates storage for the rows. **However, this does not allocate the desired number of columns.** The reason is that the default constructor is called for each row, and the default is to construct an empty array.

To fix this, we make use of a nice feature of the vector class constructor. **There is an optional second argument, which indicates the value to use when initializing each element of the vector.** In our case, each element of M is a vector of m integers, that is, "vector<int>(m)." Thus, given integer variables n and m , the following code fragment generates an $n \times m$ matrix as a vector of vectors.

```
vector< vector<int> > M(n, vector<int>(m));
cout << M[i][j] << endl;
```

The space between vector<int> and the following ">" has been added to prevent ambiguity with the C++ input operator ">>." Because the STL vector class automatically takes care of deleting its members, we do not need to write a loop to explicitly delete the rows, as we needed with dynamic arrays.

Two-dimensional arrays have many applications. Next, we explore a simple application of two-dimensional arrays for implementing a positional game.

Tic-Tac-Toe

As most school children know, **Tic-Tac-Toe** is a game played on a three-by-three board. Two players, X and O, alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.

This is admittedly not a sophisticated positional game, and it's not even that much fun to play, since a good player O can always force a tie. Tic-Tac-Toe's saving grace is that it is a nice, simple example showing how two-dimensional arrays can be used for positional games. Software for more sophisticated positional games, such as checkers, chess, or the popular simulation games, are all based on the same approach we illustrate here for using a two-dimensional array for Tic-Tac-Toe.

The basic idea is to use a two-dimensional array, *board*, to maintain the game board. Cells in this array store values that indicate if that cell is empty or stores an X or O. That is, *board* is a three-by-three matrix. For example, its middle row consists of the cells *board*[1][0], *board*[1][1], and *board*[1][2]. In our case, we choose to make the cells in the *board* array be integers, with a 0 indicating an empty cell, a 1 indicating an X, and a -1 indicating O. **This encoding allows us to have a simple way of testing whether a given board configuration is a win for X or O,** namely, if the values of a row, column, or diagonal add up to -3 or 3, respectively.

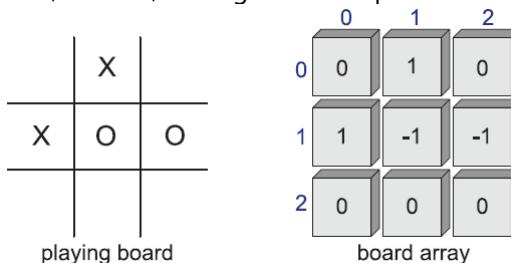


Figure 3.7 A Tic-Tac-Toe board and the array representing it

We give a complete C++ program for maintaining a Tic-Tac-Toe board for two players below. We show the resulting output in Figure 3.8. Note that this code is just for maintaining the Tic-Tac-Toe board and registering moves; it doesn't perform any strategy or allow someone to play Tic-Tac-Toe against the computer. The details of such a program are beyond the scope of this chapter, but it might nonetheless make a good project

```

#include <cstddef>           // system definitions
#include <iostream>          // I/O definitions
using namespace std;         // make std:: accessible

const int X = 1, O = -1, EMPTY = 0; // possible marks
int board[3][3];                // playing board
int currentPlayer;              // current player (X or O)

void clearBoard() {             // clear the board
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            board[i][j] = EMPTY;
    currentPlayer = X;           // every cell is empty
}                                // player X starts

void putMark(int i, int j) {      // mark row i column j
    board[i][j] = currentPlayer; // mark with current player
    currentPlayer = -currentPlayer; // switch players
}

bool isWin(int mark) {           // is mark the winner?
    int win = 3*mark;           // +3 for X and -3 for O
    return ((board[0][0] + board[0][1] + board[0][2] == win) // row 0
        || (board[1][0] + board[1][1] + board[1][2] == win) // row 1
        || (board[2][0] + board[2][1] + board[2][2] == win) // row 2
        || (board[0][0] + board[1][0] + board[2][0] == win) // column 0
        || (board[0][1] + board[1][1] + board[2][1] == win) // column 1
        || (board[0][2] + board[1][2] + board[2][2] == win) // column 2
        || (board[0][0] + board[1][1] + board[2][2] == win) // diagonal
        || (board[2][0] + board[1][1] + board[0][2] == win)); // diagonal
}

int getWinner() {                // who wins? (EMPTY means tie)
    if (isWin(X)) return X;
    else if (isWin(O)) return O;
    else return EMPTY;
}

void printBoard() {              // print the board
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            switch (board[i][j]) {
                case X: cout << "X"; break;
                case O: cout << "O"; break;
                case EMPTY: cout << " "; break;
            }
            if (j < 2) cout << "|";           // column boundary
        }
        if (i < 2) cout << "\n---\n";       // row boundary
    }
}

int main() {                     // main program
    clearBoard();               // clear the board
    putMark(0,0);               // add the marks
    putMark(0,1);
    putMark(0,2);
    putMark(2,0);
    putMark(2,1);
    putMark(2,2);
    putMark(1,0);
    printBoard();               // print the final board
    int winner = getWinner();   // print the winner
    if (winner != EMPTY)
        cout << " " << (winner == X ? 'X' : 'O') << " wins" << endl;
    else
        cout << " Tie" << endl;
    return EXIT_SUCCESS;
}

```

```

X|X|O
+---+
X|O|O
+---+
X|O|X  X wins

```

Figure 3.8 Output of the Tic-Tac-Toe program.

3.2 Singly Linked Lists

In the previous section, we presented the array data structure and discussed some of its applications. **Arrays are nice and simple for storing things in a certain order, but they have drawbacks. They are not very adaptable.** For instance, we have to fix the size n of an array in advance, which makes resizing an array difficult. (This drawback is remedied in STL vectors.) Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion. In this section, **we explore an important alternate implementation of sequence, known as the singly linked list.**

A **linked list**, in its simplest form, is a collection of **nodes** that together form a linear ordering. As in the children's game "Follow the Leader," **each node stores a pointer, called *next*, to the next node of the list.** In addition, **each node stores its associated element.** (See Figure 3.9.)



Figure 3.9 Example of a singly linked list of airport codes. The next pointers are shown as arrows. The null pointer is denoted by \emptyset .

The *next* pointer inside a node is a **link** or **pointer** to the next node of the list. Moving from one node to another by following a *next* reference is known as **link hopping** or **pointer hopping**. The first and last nodes of a linked list are called the **head** and **tail** of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the tail. We can identify the tail as the node having a null *next* reference. The structure is called a **singly linked list** because each node stores a single link.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of next links. **Unlike an array, a singly linked list does not have a predetermined fixed size.** It can be resized by adding or removing nodes.

● Implementing a Singly Linked List

Let us implement a singly linked list of strings. **We first define a class `StringNode` shown below.** The node stores two values, the member *elem* stores the element stored in this node, which in this case is a character string. The member *next* stores a pointer to the next node of the list. **We make the linked list class a friend**, so that it can access the node's private members.

```

class StringNode {
private:
    string elem; // element value
    StringNode* next; // next item in the list
    friend class StringLinkedList; // provide StringLinkedList access
};

```

Below, we define a class `StringLinkedList` for the actual linked list. It supports a number of member functions, including a constructor and destructor and functions for insertion and deletion. Their implementations are presented later. Its private data consists of a pointer to the head node of the list.

```

class StringLinkedList {
public:
    StringLinkedList(); // a linked list of strings
    ~StringLinkedList(); // empty list constructor
    bool empty() const; // destructor
    const string& front() const; // is list empty?
    void addFront(const string& e); // get front element
    void removeFront(); // add to front of list
private:
    StringNode* head; // remove front item list
};

```

A number of simple member functions are shown below. The list constructor creates an empty list by setting the head pointer to NULL. The destructor repeatedly removes elements from the list. It exploits the fact that the function `remove` (presented below) destroys the node that it removes. To test whether the list is empty, we simply test whether the head pointer is NULL.

```

StringLinkedList::StringLinkedList() // constructor
: head(NULL) { }

StringLinkedList::~StringLinkedList() // destructor
{ while (!empty()) removeFront(); }

bool StringLinkedList::empty() const // is list empty?
{ return head == NULL; }

const string& StringLinkedList::front() const // get front element
{ return head->elem; }

```

● Insertion to the Front of a Singly Linked List

We can easily insert an element at the head of a singly linked list. We first create a new node, and set its *elem* value to the desired string and set its *next* link to point to the current head of the list. We then set *head* to point to the new node. The process is illustrated in Figure 3.10.

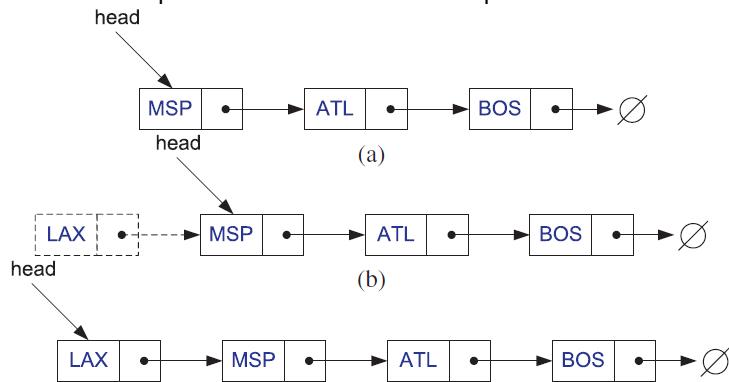


Figure 3.10 Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) creation of a new node; (c) after the insertion.

An implementation is shown below. Note that access to the private members *elem* and *next* of the `StringNode` class would normally be prohibited, but it is allowed here because `StringLinkedList` was declared to be a friend of `StringNode`.

```

void StringLinkedList::addFront(const string& e) { // add to front of list
    StringNode* v = new StringNode; // create new node
    v->elem = e; // store data
    v->next = head; // head now follows v
    head = v; // v is now the head
}

```

● Removal from the Front of a Singly Linked List

Next, we consider how to remove an element from the front of a singly linked list. We essentially

undo the operations performed for insertion. We first save a pointer to the old head node and advance the head pointer to the next node in the list. We then delete the old head node. This operation is illustrated in Figure 3.11.

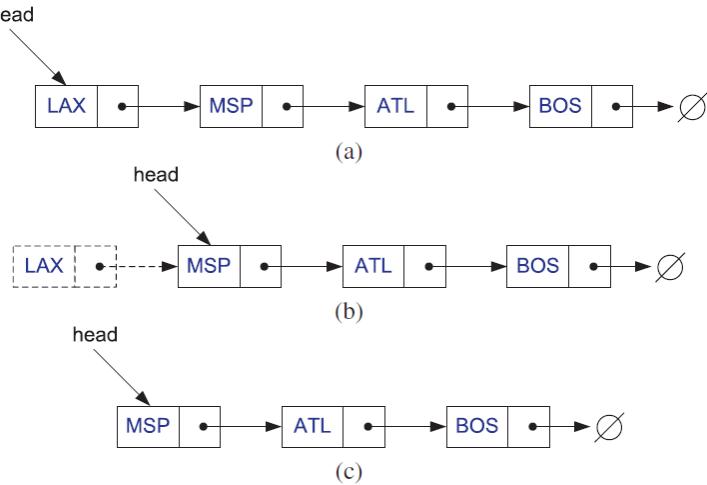


Figure 3.11 Removal of an element at the head of a singly linked list: (a) before the removal; (b) "linking out" the old new node; (c) after the removal.

An implementation of this operation is provided below. **We assume that the user has checked that the list is nonempty before applying this operation.** (A more careful implementation would throw an exception if the list were empty.) The function deletes the node in order to avoid any memory leaks. We do not return the value of the deleted node. If its value is desired, we can call the front function prior to the removal.

```

void StringLinkedList::removeFront() {
    StringNode* old = head;
    head = old->next;
    delete old;
} // remove front item
// save current head
// skip over old head
// delete the old head

```

It is noteworthy that we cannot as easily delete the last node of a singly linked list, even if we had a pointer to it. In order to delete a node, we need to update the *next* link of the node immediately **preceding** the deleted node. Locating this node involves traversing the entire list and could take a long time.

● Implementing a Generic Singly Linked List

The implementation of the singly linked list given above assumes that the element type is a character string. It is easy to convert the implementation so that it works for an arbitrary element type through the use of C++'s template mechanism. The resulting generic singly linked list class is called SLinkedList.

We begin by presenting the node class below, called SNode. **The element type associated with each node is parameterized by the type variable E.** In contrast to our earlier version above, references to the data type "string" have been replaced by "E." **When referring to our templated node and list class, we need to include the suffix "<E>."** For example, the class is SLinkedList<E> and the associated node is SNode<E>.

```

template <typename E>
class SNode { // singly linked list node
private:
    E elem; // linked list element value
    SNode<E>* next; // next item in the list
    friend class SLinkedList<E>; // provide SLinkedList access
};

```

The generic list class is presented below. As above, references to the specific element type "string" have been replaced by references to the generic type parameter "E." To keep things simple, we have omitted housekeeping functions such as a copy constructor.

```
template <typename E>
class SLinkedList { // a singly linked list
public:
    SLinkedList(); // empty list constructor
    ~SLinkedList(); // destructor
    bool empty() const; // is list empty?
    const E& front() const; // return front element
    void addFront(const E& e); // add to front of list
    void removeFront(); // remove front item list
private:
    SNode<E>* head; // head of the list
};
```

Below, we present the class member functions. Note the similarity with above code fragments.

*Observe that each definition is prefaced by the template specifier **template <typename E>**.*

```
template <typename E>
SLinkedList<E>::SLinkedList() // constructor
: head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const // is list empty?
{ return head == NULL; }

template <typename E>
const E& SLinkedList<E>::front() const // return front element
{ return head->elem; }

template <typename E>
SLinkedList<E>::~SLinkedList() // destructor
{ while (!empty()) removeFront(); }

template <typename E>
void SLinkedList<E>::addFront(const E& e) { // add to front of list
    SNode<E>* v = new SNode<E>; // create new node
    v->elem = e; // store data
    v->next = head; // head now follows v
    head = v; // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() { // remove front item
    SNode<E>* old = head; // save current head
    head = old->next; // skip over old head
    delete old; // delete the old head
}
```

We can generate singly linked lists of various types by simply setting the template parameter as desired as shown in the following code fragment.

```
SLinkedList<string> a; // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b; // list of integers
b.addFront(13);
```

Because templated classes carry a relatively high notational burden, we often sacrifice generality for simplicity, and avoid the use of templated classes in some of our examples.

3.3 Doubly Linked Lists

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked

list, **since we do not have a quick way of accessing the node immediately preceding the one we want to remove**. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the **doubly linked list**. In addition to its element member, **a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list, respectively**. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

Header and Trailer Sentinels

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a **header** node just before the head of the list, and a **trailer** node just after the tail of the list. These "dummy" or **sentinel** nodes do not store any elements. *They provide quick access to the first and last nodes of the list*. In particular, the header's *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.

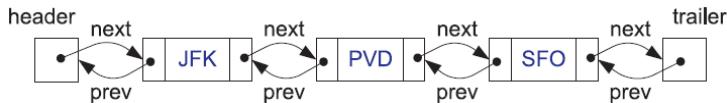


Figure 3.12 A doubly linked list with sentinels, *header* and *trailer*, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null *prev* pointer for the *header* nor do we show the null *next* pointer for the *trailer*.

- Insertion into a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. Given a node v of a doubly linked list (which could possibly be the header, but not the trailer), let z be a new node that we wish to insert immediately after v . Let w be the node following v , that is, w is the node pointed to by v 's next link. (This node exists, since we have sentinels.) To insert z after v , we link it into the current list, by performing the following operations:

- Make z 's *prev* link point to v
- Make z 's *next* link point to w
- Make w 's *prev* link point to z
- Make v 's *next* link point to z

This process is illustrated in Figure 3.13, where v points to the node JFK, w points to PVD, and z points to the new node BWI. Observe that this process works if v is any node ranging from the header to the node just prior to the trailer.

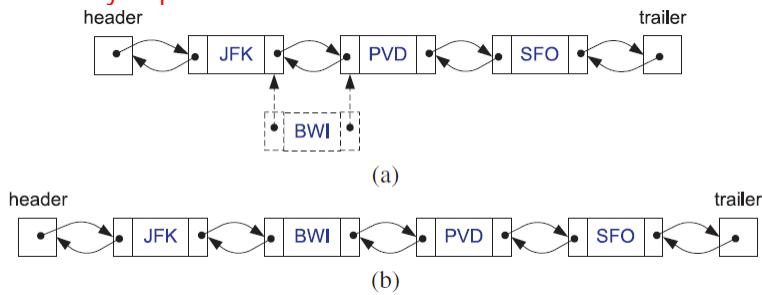


Figure 3.13 Adding a new node after the node storing JFK: (a) creating a new node with element BWI and linking it in; (b) after the insertion.

- Removal from a Doubly Linked List

Likewise, it is easy to remove a node v from a doubly linked list. Let u be the node just prior to v , and w be the node just following v . (These nodes exist, since we have sentinels.) To remove node v , we simply have u and w point to each other instead of to v . We refer to this operation as the **linking out** of v . We perform the following operations.

- Make w 's *prev* link point to u
- Make u 's *next* link point to w
- Delete node v

This process is illustrated in Figure 3.14, where v is the node PVD, u is the node JFK, and w is the node SFO. Observe that this process works if v is any node from the header to the tail node (the node just prior to the trailer).

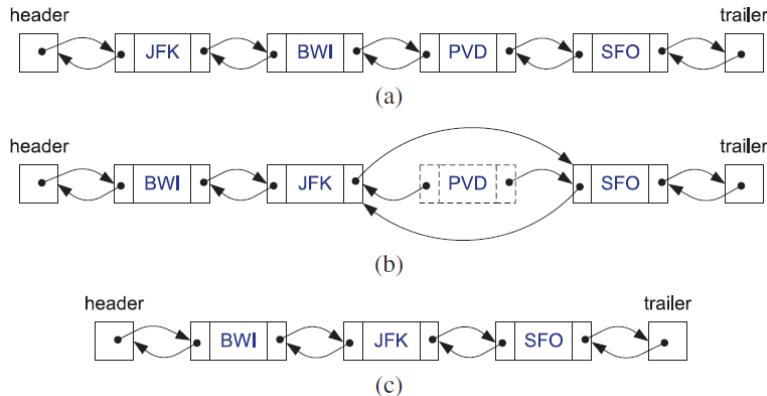


Figure 3.14 Removing the node storing PVD: (a) before the removal (some obvious errors); (b) linking out the old node; (c) after node deletion.

● A C++ Implementation

Let us consider how to implement a doubly linked list in C++. First, we present a C++ class for a node of the list below. To keep the code simple, we have chosen not to derive a templated class as we did for singly linked lists. Instead, we provide a **typedef** statement that defines the element type, called *Elem*. We define it to be a string, but any other type could be used instead. Each node stores an element. It also contains pointers to both the previous and next nodes of the list. We declare *DLinkedList* to be a friend, so it can access the node's private members.

```
typedef string Elem;                                // list element type
class DNode {                                         // doubly linked list node
private:
    Elem elem;                                       // node element value
    DNode* prev;                                      // previous node in list
    DNode* next;                                      // next node in list
    friend class DLinkedList;                         // allow DLinkedList access
};
```

Next, we present the definition of the doubly linked list class below, *DLinkedList*. In addition to a constructor and destructor, the public members consist of a function that indicates whether the list is currently empty (**meaning that it has no nodes other than the sentinels**) and accessors to retrieve the front and back elements. We also provide methods for inserting and removing elements from the front and back of the list. There are two private data members, *header* and *trailer*, which point to the sentinels. Finally, we provide two protected utility member functions, *add* and *remove*. They are used internally by the class and by its subclasses, but they cannot be invoked from outside the class.

```

class DLinkedList {                                // doubly linked list
public:
    DLinkedList();                                // constructor
    ~DLinkedList();                               // destructor
    bool empty() const;                          // is list empty?
    const Elem& front() const;                  // get front element
    const Elem& back() const;                   // get back element
    void addFront(const Elem& e);              // add to front of list
    void addBack(const Elem& e);                // add to back of list
    void removeFront();                         // remove from front
    void removeBack();                          // remove from back
private:
    DNode* header;                             // local type definitions
    DNode* trailer;                            // list sentinels
protected:
    void add(DNode* v, const Elem& e);        // local utilities
    void remove(DNode* v);                     // insert new node before v
};

```

Let us begin by presenting the class constructor and destructor as shown below. **The constructor creates the sentinel nodes and sets each to point to the other, and the destructor removes all but the sentinel nodes.**

```

DLinkedList::DLinkedList() {
    header = new DNode;                        // constructor
    trailer = new DNode;                       // create sentinels
    header->next = trailer;                  // have them point to each other
    trailer->prev = header;
}

DLinkedList::~DLinkedList() {
    while (!empty()) removeFront();           // destructor
    delete header;                           // remove all but sentinels
    delete trailer;                          // remove the sentinels
}

```

Next, below we show the basic class accessors. To determine whether the list is empty, we check that there is no node between the two sentinels. **We do this by testing whether the trailer follows immediately after the header.** To access the front element of the list, we return the element associated with the node that follows the list header. To access the back element, we return the element associated with node that precedes the trailer. Both operations assume that the list is nonempty. We could have enhanced these functions by throwing an exception if an attempt is made to access the front or back of an empty list.

```

bool DLinkedList::empty() const          // is list empty?
{ return (header->next == trailer); }

const Elem& DLinkedList::front() const   // get front element
{ return header->next->elem; }

const Elem& DLinkedList::back() const     // get back element
{ return trailer->prev->elem; }

```

In Section 3.3, we discussed how to insert a node into a doubly linked list. The local utility function `add`, which is shown below, implements this operation. In order to add a node to the front of the list, we create a new node, and insert it immediately after the header, or equivalently, immediately before the node that follows the header. In order to add a new node to the back of the list, we create a new node, and insert it immediately before the trailer.

```

        // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode; u->elem = e; // create a new node for e
    u->next = v;                      // link u in between v
    u->prev = v->prev;                // ...and v->prev
    v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
{ add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
{ add(trailer, e); }

```

Observe that the above code works even if the list is empty (meaning that the only nodes are the header and trailer). For example, if addBack is invoked on an empty list, then the value of *trailer->prev* is a pointer to the list header. Thus, the node is added between the header and trailer as desired. One of the major advantages of providing sentinel nodes is to avoid handling of special cases, which would otherwise be needed.

Finally, let us discuss deletion. In Section 3.3, we showed how to remove an arbitrary node from a doubly linked list. Below we present local utility function remove, which performs the operation. In addition to linking out the node, it also deletes the node. The public member functions removeFront and removeBack are implemented by deleting the nodes immediately following the header and immediately preceding the trailer, respectively.

```

void DLinkedList::remove(DNode* v) {           // remove node v
    DNode* u = v->prev;                      // predecessor
    DNode* w = v->next;                      // successor
    u->next = w;                            // unlink v from list
    w->prev = u;
    delete v;
}

void DLinkedList::removeFront()           // remove from front
{ remove(header->next); }

void DLinkedList::removeBack()            // remove from back
{ remove(trailer->prev); }

```

There are many more features that we could have added to our simple implementation of a doubly linked list. Although we have provided access to the ends of the list, we have not provided any mechanism for accessing or modifying elements in the middle of the list. Later, in Chapter 6, we discuss the concept of iterators, which provides a mechanism for accessing arbitrary elements of a list.

We have also performed no error checking in our implementation. It is the user's responsibility not to attempt to access or remove elements from an empty list. In a more robust implementation of a doubly linked list, we would design the member functions front, back, removeFront, and removeBack to throw an exception when an attempt is made to perform one of these functions on an empty list. Nonetheless, this simple implementation illustrates how easy it is to manipulate this useful data structure.

3.4 Circularly Linked Lists and List Reversal

In this section, we study some applications and extensions of linked lists. (@todo mark)

- Circularly Linked Lists

A **circularly linked list** has the **same** kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head

or tail, the nodes of a circularly linked list are linked into a cycle.

Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the **cursor**. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.

There are **two** positions of particular interest in a circular list. The first is the element that is referenced by the cursor, which is called the **back**, and the element immediately following this in the circular order, which is called the **front**. Although it may seem odd to think of a circular list as having a front and a back, observe that, if we were to cut the link between the node referenced by the cursor and this node's immediate successor, the result would be a singly linked list from the front node to the back node.

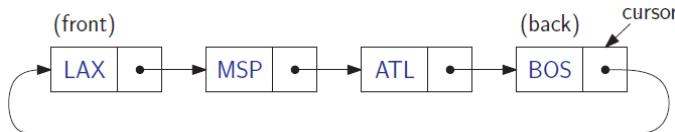


Figure 3.15 A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

We define the following functions for a circularly linked list:

- `front()`: Return the element referenced by the cursor; an error results if the list is empty.
- `back()`: Return the element immediately after the cursor; an error results if the list is empty.
- `advance()`: Advance the cursor to the next node in the list.
- `add(e)`: Insert a new node with element *e* immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.
- `remove()`: Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.

```

typedef string Elem;           // element type
class CNode {                 // circularly linked list node
private:
    Elem elem;                // linked list element value
    CNode* next;               // next item in the list

    friend class CircleList;   // provide CircleList access
};

class CircleList {            // a circularly linked list
public:
    CircleList();              // constructor
    ~CircleList();             // destructor
    bool empty() const;       // is list empty?
    const Elem& front() const; // element at cursor
    const Elem& back() const; // element following cursor
    void advance();           // advance cursor
    void add(const Elem& e); // add after cursor
    void remove();            // remove node after cursor
private:
    CNode* cursor;             // the cursor
};

CircleList::CircleList()          // constructor
: cursor(NULL) { }
CircleList::~CircleList()         // destructor
{ while (!empty()) remove(); }
  
```

```

bool CircleList::empty() const           // is list empty?
{ return cursor == NULL; }
const Elem& CircleList::back() const    // element at cursor
{ return cursor->elem; }
const Elem& CircleList::front() const   // element following cursor
{ return cursor->next->elem; }
void CircleList::advance()
{ cursor = cursor->next; }

```

Next, let us consider insertion. Recall that insertions to the circularly linked list occur **after** the cursor. We begin by creating a new node and initializing its data member. If the list is empty, we create a new node that points to itself. We then direct the cursor to point to this element. Otherwise, we link the new node just after the cursor.

```

void CircleList::add(const Elem& e) {           // add after cursor
    CNode* v = new CNode;                      // create a new node
    v->elem = e;
    if (cursor == NULL) {                      // list is empty?
        v->next = v;                          // v points to itself
        cursor = v;                           // cursor points to v
    }
    else {                                     // list is nonempty?
        v->next = cursor->next;              // link in v after cursor
        cursor->next = v;
    }
}

```

Finally, we consider removal. We assume that the user has checked that the list is nonempty before invoking this function. There are **two** cases. If this is the last node of the list (which can be tested by checking that the node to be removed points to itself) we set the cursor to NULL. Otherwise, we link the cursor's next pointer to skip over the removed node.

```

void CircleList::remove() {                   // remove node after cursor
    CNode* old = cursor->next;               // the node being removed
    if (old == cursor) {                     // removing the only node?
        cursor = NULL;                        // list is now empty
    }
    else {                                     // link out the old node
        cursor->next = old->next;            // delete the old node
        delete old;
    }
}

```

In front, back, and advance, we should first test whether the list is empty, since otherwise the cursor pointer will be NULL. In the first two cases, we should throw some sort of exception. In the case of advance, if the list is empty, we can **simply** return.

Maintaining a Playlist for a Digital Audio Player

To help illustrate the use of our CircleList implementation of the circularly linked list, let us consider how to build a simple interface for maintaining a playlist for a digital audio player, also known as an MP3 player. The songs of the player are stored in a circular list. **The cursor points to the current song.** By advancing the cursor, we can move from one song to the next. We can also add new songs and remove songs by invoking the member functions `insert` and `remove`, respectively. Of course, a complete implementation would need to provide a method for playing the current song, but our purpose is to illustrate how the circularly linked list can be applied to this task.

We declare an object `playList` to be a `CircleList`. The constructor creates an empty playlist. We proceed to add three songs, "Stayin Alive," "Le Freak," and "Jive Talkin." The comments on the right show the current contents of the list in square brackets. **The first entry** of the list is the element immediately following the cursor (which is where insertion and removal occur), and **the last** entry in the list is `cursor` (which is indicated with an asterisk).

```

int main() {
    CircleList playList;           // []
    playList.add("Stayin Alive");   // [Stayin Alive]
    playList.add("Le Freak");      // [Le Freak, Stayin Alive]
    playList.add("Jive Talkin");    // [Jive Talkin, Le Freak, Stayin Alive]

    playList.advance();            // [Le Freak, Stayin Alive, Jive Talkin]
    playList.advance();            // [Stayin Alive, Jive Talkin, Le Freak]
    playList.remove();             // [Jive Talkin, Le Freak]
    playList.add("Disco Inferno");  // [Disco Inferno, Jive Talkin, Le Freak]
    return EXIT_SUCCESS;
}

```

● Reversing a Linked List

As another example of the manipulation of linked lists, we present a simple function for reversing the elements of a doubly linked list. Given a list L , our approach involves first copying the contents of L in reverse order into a temporary list T , and then copying the contents of T back into L (but without reversing).

To achieve the initial reversed copy, we repeatedly extract the first element of L and copy it to the front of T . (To see why this works, observe that the later an element appears in L , the earlier it will appear in T .) To copy the contents of T back to L , we repeatedly extract elements from the front of T , but this time we copy each one to the back of list L :

```

void listReverse(DLinkedList& L) {           // reverse a list
    DLinkedList T;                          // temporary list
    while (!L.empty()) {                   // reverse L into T
        string s = L.front();  L.removeFront();
        T.addFront(s);
    }
    while (!T.empty()) {                   // copy T back to L
        string s = T.front();  T.removeFront();
        L.addBack(s);
    }
}

```

3.5 Recursion

We have seen that repetition can be achieved by writing loops, such as **for** loops and **while** loops. **Another way to achieve repetition** is through **recursion**, which occurs when a function refers to itself in its own definition (*The definition of recursion in mathematics*: an expression such that each term is generated by repeating a particular mathematical operation). We have seen examples of functions calling other functions, so it should come as no surprise that most modern programming languages, including C++, allow a function to call itself. In this section, we see why this capability provides an elegant and powerful alternative for performing repetitive tasks.

The Factorial Function

To illustrate recursion, let us begin with a **simple** example of computing the value of the **factorial function**. The factorial of a positive integer n , denoted $n!$, is defined as the product of the integers from 1 to n . If $n = 0$, then $n!$ is defined as 1 by convention. More formally, for any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. To make the connection with functions clearer, we use the **notation** $\text{factorial}(n)$ to denote $n!$.

The factorial function can be defined in a manner that suggests a recursive formulation. To see this, observe that

$$\text{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \text{factorial}(4)$$

Thus, we can define $\text{factorial}(5)$ in terms of $\text{factorial}(4)$. In general, for a positive integer n , we can define $\text{factorial}(n)$ to be $n \cdot \text{factorial}(n - 1)$. This leads to the following **recursive definition**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n - 1) & \text{if } n \geq 1 \end{cases}$$

This definition is typical of many recursive definitions. First, it contains one or more **base cases**, which are defined nonrecursively in terms of fixed quantities. In this case, $n = 0$ is the base case. It also contains **one or more recursive cases**, which are defined by appealing to the definition of the function being defined. Observe that there is **no circularity** in this definition because each time the function is invoked, its argument is smaller by one.

A Recursive Implementation of the Factorial Function

Let us consider a C++ implementation of the factorial function shown below under the name `recursiveFactorial`. Notice that no looping was needed here. The repeated recursive invocations of the function take the place of looping.

```
int recursiveFactorial(int n) { // recursive factorial function
    if (n == 0) return 1; // basis case
    else return n * recursiveFactorial(n-1); // recursive case
}
```

We can illustrate the execution of a recursive function definition by means of a **recursion trace**. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by an arrow to the newly called function. When the function returns, an arrow showing this return is drawn, and the return value may be indicated with this arrow. An example of a trace is shown in Figure 3.16.

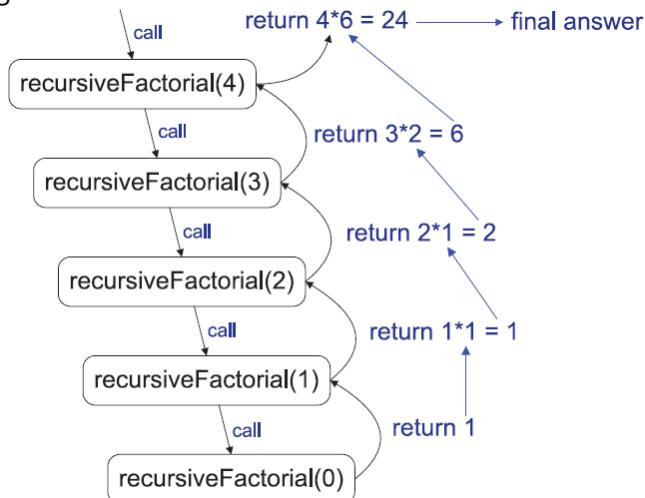


Figure 3.16 A recursion trace for the call `recursiveFactorial(4)`.

What is the advantage of using recursion? Although the recursive implementation of the factorial function is somewhat simpler than the iterative version, in this case there is no compelling reason for preferring recursion over iteration. For some problems, however, a **recursive implementation can be significantly simpler and easier to understand than an iterative implementation**. Such an example follows.

Drawing an English Ruler

As a more complex example of the use of recursion, consider how to draw the markings of a typical English ruler. Such a ruler is broken into intervals, and each interval consists of a set of **ticks**, placed at intervals of $1/2$ inch, $1/4$ inch, and so on. As the size of the interval decreases

by half, the tick length decreases by one.

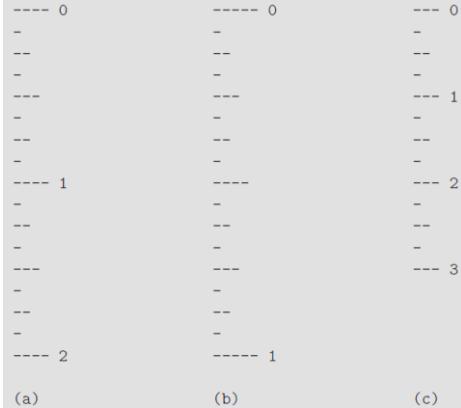


Figure 3.17 Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

Each fraction of an inch also has a numeric label. The longest tick length is called the **major tick length**. We won't worry about actual distances, however, and just print one tick per line.

A Recursive Approach to Ruler Drawing

Our approach to drawing such a ruler consists of three functions. The main function `drawRuler` draws the entire ruler. Its arguments are the total number of inches in the ruler, `nInches`, and the major tick length, `majorLength`. The utility function `drawOneTick` draws a single tick of the given length. It can also be given an optional integer label, which is printed if it is nonnegative.

The interesting work is done by the recursive function `drawTicks`, which draws the sequence of ticks within some interval. Its only argument is the tick length associated with the interval's central tick. Consider the English ruler with major tick length 5 shown in Figure 3.17 (b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3. In general, an interval with a central tick length $L \geq 1$ is composed of the following:

- An interval with a central tick length $L - 1$
- A single tick of length L
- An interval with a central tick length $L - 1$

With each recursive call, the length decreases by one. When the length drops to zero, we simply return. As a result, this recursive process always terminates. This suggests a recursive process in which the first and last steps are performed by calling the `drawTicks($L - 1$)` recursively. The middle step is performed by calling the function `drawOneTick(L)`. This recursive formulation is shown below. As in the factorial example, the code has a base case (when $L = 0$). In this instance we make two recursive calls to the function.

```

    // one tick with optional label
void drawOneTick(int tickLength, int tickLabel = -1) {
    for (int i = 0; i < tickLength; i++)
        cout << "-";
    if (tickLabel >= 0) cout << " " << tickLabel;
    cout << "\n";
}
void drawTicks(int tickLength) {           // draw ticks of given length
    if (tickLength > 0) {                 // stop when length drops to 0
        drawTicks(tickLength-1);          // recursively draw left ticks
        drawOneTick(tickLength);          // draw center tick
        drawTicks(tickLength-1);          // recursively draw right ticks
    }
}
void drawRuler(int nInches, int majorLength) { // draw the entire ruler
    drawOneTick(majorLength, 0);             // draw tick 0 and its label
    for (int i = 1; i <= nInches; i++) {      // draw ticks for this inch
        drawTicks(majorLength-1);            // draw ticks for this inch
        drawOneTick(majorLength, i);          // draw tick i and its label
    }
}

```

Illustrating Ruler Drawing using a Recursion Trace

The recursive execution of the recursive drawTicks function, defined above, can be visualized using a **recursion trace**.

The trace for drawTicks is more complicated than in the factorial example, however, because **each instance makes two recursive calls**. To illustrate this, we show the recursion trace in a form that is reminiscent of an outline for a document.

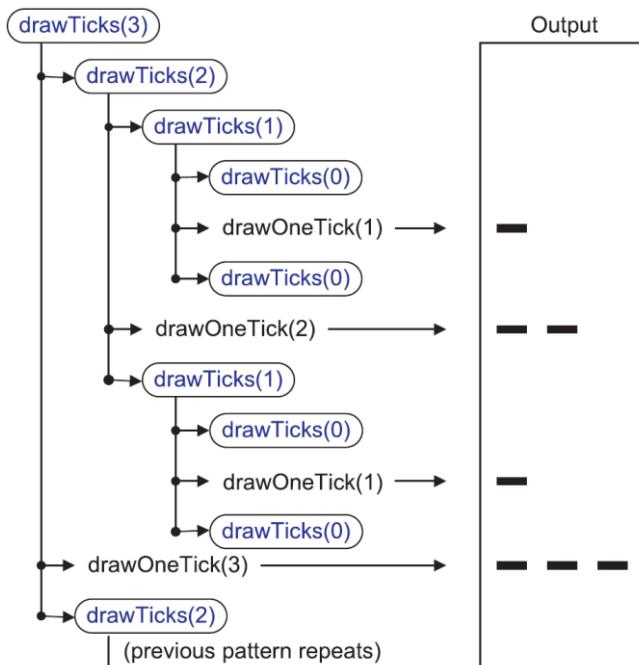


Figure 3.18 A partial recursion trace for the call `drawTicks(3)`. The second pattern of calls for `drawTicks(2)` is not shown, but it is identical to the first.

Further Illustrations of Recursion

As we discussed above, **recursion** is the concept of defining a function that makes a call to itself. When a function calls itself, we refer to this as a **recursive** call. We also consider a function M to be recursive if it calls another function that ultimately leads to a call back to M .

The main benefit of a recursive approach to algorithm design is that it allows us to take advantage of the repetitive structure present in many problems. By making our algorithm description exploit this repetitive structure in a recursive way, we can often avoid complex case

analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.

In addition, recursion is a useful way for defining objects that have a repeated similar structural form, such as in the following examples.

Example 3.1: Modern operating systems define file-system directories (which are also sometimes called "folders") in a recursive way. Namely, a file system consists of a top-level directory, and the contents of this directory consists of files and other directories, which in turn can contain files and other directories, and so on. The base directories in the file system contain only files, but by using this recursive definition, the operating system allows for directories to be nested arbitrarily deep (as long as there is enough space in memory).

Example 3.2: Much of the syntax in modern programming languages is defined in a recursive way. For example, we can define an argument list in C++ using the following notation:

```
argument-list:  
    argument  
    argument-list, argument
```

In other words, an argument list consists of either (i) an argument or (ii) an argument list followed by a comma and an argument. That is, an argument list consists of a comma-separated list of arguments. Similarly, arithmetic expressions can be defined recursively in terms of primitives (like variables and constants) and arithmetic expressions.

Example 3.3: There are many examples of recursion in art and nature. One of the most classic examples of recursion used in art is in the Russian Matryoshka dolls. Each doll is made of solid wood or is hollow and contains another Matryoshka doll inside it.

● Linear Recursion

The simplest form of recursion is **linear recursion**, where a function is defined so that it makes at most one recursive call each time it is invoked. This type of recursion is useful when we view **an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set**.

Summing the Elements of an Array Recursively

Suppose, for example, we are given an array, A , of n integers that we want to sum together. We can solve this summation problem using linear recursion by observing that the sum of all n integers in A is equal to $A[0]$, if $n = 1$, or the sum of the first $n - 1$ integers in A plus the last element in A . In particular, we can solve this summation problem using the recursive algorithm described below:

```
Algorithm LinearSum( $A, n$ ):  
    Input: A integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements  
    Output: The sum of the first  $n$  integers in  $A$   
    if  $n = 1$  then  
        return  $A[0]$   
    else  
        return LinearSum( $A, n - 1$ ) +  $A[n - 1]$ 
```

This example also illustrates an important property that a recursive function should always possess—the **function terminates**. We ensure this by writing a non-recursive statement for the case $n = 1$. In addition, we always perform the recursive call on a smaller value of the parameter ($n - 1$) than that which we are given (n), so that, at some point (at the "bottom" of the recursion),

we will perform the non-recursive part of the computation (returning $A[0]$). In general, an algorithm that uses linear recursion typically has the following form:

- **Test for base cases.** We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls eventually reaches a base case, and the handling of each base case should not use recursion.
- **Recur.** After testing for base cases, we then perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but *it should ultimately choose to make just one of these calls each time we perform this step*. Moreover, we should define each possible recursive call so that it makes progress towards a base case.

Analyzing Recursive Algorithms using Recursion Traces

We can analyze a recursive algorithm by using a visual tool known as a **recursion trace**. We used recursion traces, for example, to analyze and visualize the recursive factorial function of Section 3.5, and we similarly use recursion traces for the recursive sorting algorithms of Sections 11.1 and 11.2.

To draw a recursion trace, we create a box for each instance of the function and label it with the parameters of the function. Also, we visualize a recursive call by drawing an arrow from the box of the calling function to the box of the called function. For example, we illustrate the recursion trace of the LinearSum algorithm in Figure 3.19. We label each box in this trace with the parameters used to make this call. Each time we make a recursive call, we draw a line to the box representing the recursive call. We can also use this diagram to visualize stepping through the algorithm, since it proceeds by going from the call for n to the call for $n - 1$, to the call for $n - 2$, and so on, all the way down to the call for 1. When the final call finishes, it returns its value back to the call for 2, which adds in its value, and returns this partial sum to the call for 3, and so on, until the call for $n - 1$ returns its partial sum to the call for n .

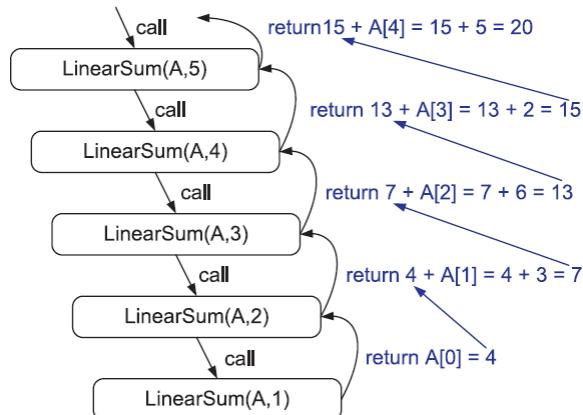


Figure 3.19 Recursion trace for an execution of $\text{LinearSum}(A, n)$ with input parameters $A = \{4,3,6,2,5\}$ and $n = 5$

From Figure 3.19, it should be clear that for an input array of size n , Algorithm LinearSum makes n calls. Hence, **it takes an amount of time that is roughly proportional to n** , since it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that **the memory space used by the algorithm (in addition to the array A) is also roughly proportional to n** , since we need a constant amount of memory space for each of the n boxes in the trace at the time we make the final recursive call (for $n = 1$).

Reversing an Array by Recursion

Next, let us consider the problem of reversing the n elements of an array, A , so that the first element becomes the last, the second element becomes second to the last, and so on. We can solve this problem using linear recursion, by observing that the reversal of an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array. We describe the details of this algorithm below, using the convention that the first time we call this algorithm we do so as $\text{ReverseArray}(A, 0, n - 1)$.

Algorithm $\text{ReverseArray}(A, i, j)$:

Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j

```
if  $i < j$  then
    Swap  $A[i]$  and  $A[j]$ 
     $\text{ReverseArray}(A, i + 1, j - 1)$ 
return
```

Note that, in this algorithm, we actually have **two base cases**, namely, when $i = j$ and when $i > j$. Moreover, in either case, we simply terminate the algorithm, since a sequence with zero elements or one element is trivially equal to its reversal. Furthermore, note that in the recursive step we are guaranteed to make progress towards one of these two base cases. If n is odd, we eventually reach the $i = j$ case, and if n is even, we eventually reach the $i > j$ case. The above argument immediately implies that the recursive algorithm is guaranteed to terminate.

Defining Problems in Ways that Facilitate Recursion

To design a recursive algorithm for a given problem, **it is useful to think of the different ways we can subdivide this problem to define problems that have the same general structure as the original problem**. This process sometimes means we need to redefine the original problem to facilitate similar-looking subproblems. For example, with the ReverseArray algorithm, we added the parameters i and j so that a recursive call to reverse the inner part of the array A would have **the same structure (and same syntax)** as the call to reverse all of A . Then, rather than initially calling the algorithm as $\text{ReverseArray}(A)$, we call it initially as $\text{ReverseArray}(A, 0, n - 1)$. **In general, if one has difficulty finding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be defined.**

Tail Recursion

Using recursion can often be a useful tool for designing algorithms that have elegant, short definitions. But this usefulness does come at a modest cost. When we use a recursive algorithm to solve a problem, **we have to use some of the memory locations in our computer to keep track of the state of each active recursive call**. When computer memory is at a premium, then it is **useful in some cases to be able to derive non-recursive algorithms from recursive ones**.

We can use the stack data structure, discussed in Section 5.1, to convert a recursive algorithm into a non-recursive algorithm, but there are some instances when we can do this conversion more easily and efficiently. Specifically, we can easily convert algorithms that use **tail recursion**. **An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation**. For example, the algorithm of ReverseArray uses tail recursion to reverse the elements of an array.

It is not enough that the last statement in the function definition includes a recursive call, however. In order for a function to use tail recursion, the recursive call must be absolutely the last thing the function does (unless we are in a base case, of course). **For example, the algorithm of LinearSum does not use tail recursion, even though its last statement includes a recursive call**. This recursive call is not actually the last thing the function does. After

it receives the value returned from the recursive call, it adds this value to $A[n - 1]$ and returns this sum. That is, the last thing this algorithm does is an add, not a recursive call.

When an algorithm uses tail recursion, we can convert the recursive algorithm into a non-recursive one, by iterating through the recursive calls rather than calling them explicitly. We illustrate this type of conversion by revisiting the problem of reversing the elements of an array. Below, we give a non-recursive algorithm that performs this task by iterating through the recursive calls of the algorithm of ReverseArray. We initially call this algorithm as IterativeReverseArray($A, 0, n - 1$).

```
Algorithm IterativeReverseArray( $A, i, j$ ):
    Input: An array  $A$  and nonnegative integer indices  $i$  and  $j$ 
    Output: The reversal of the elements of  $A$  starting at index  $i$  and ending at  $j$ 
    while  $i < j$  do
        Swap  $A[i]$  and  $A[j]$ 
         $i \leftarrow i + 1$ 
         $j \leftarrow j - 1$ 
    return
```

● Binary Recursion

When an algorithm makes two recursive calls, we say that it uses **binary recursion**. These calls can, for example, **be used to solve two similar halves of some problem**, as we did in Section 3.5 for drawing an English ruler. As another application of binary recursion, let us revisit the problem of summing the n elements of an integer array A . In this case, we can sum the elements in A by: (i) recursively summing the elements in the first half of A ; (ii) recursively summing the elements in the second half of A ; and (iii) adding these two values together. We give the details in the algorithm below, which we initially call as BinarySum($A, 0, n$).

```
Algorithm BinarySum( $A, i, n$ ):
    Input: An array  $A$  and integers  $i$  and  $n$ 
    Output: The sum of the  $n$  integers in  $A$  starting at index  $i$ 
    if  $n = 1$  then
        return  $A[i]$ 
    return BinarySum( $A, i, \lceil n/2 \rceil$ ) + BinarySum( $A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor$ )
```

To analyze Algorithm BinarySum, we consider, for simplicity, the case where n is a power of two. Figure 3.20 shows the recursion trace of an execution of function BinarySum(0,8). We label each box with the values of parameters i and n , which represent the starting index and length of the sequence of elements to be summed, respectively. Notice that the arrows in the trace go from a box labeled (i, n) to another box labeled $(i, n/2)$ or $(i + n/2, n/2)$. That is, the value of parameter n is halved at each recursive call. Thus, **the depth of the recursion**, that is, the maximum number of function instances that are active at the same time, is $1 + \log_2 n$. The running time of Algorithm BinarySum is still roughly proportional to n , however, since each box is visited in constant time when stepping through our algorithm and there are $2n - 1$ ($n + n - 1$) boxes.

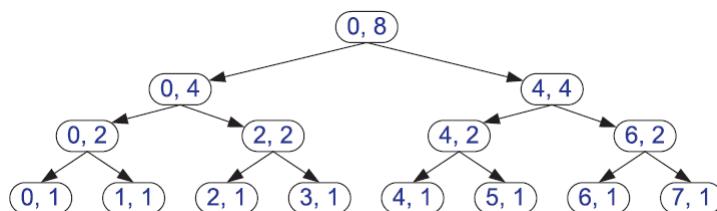


Figure 3.20 Recursion trace for the execution of BinarySum(0,8).

Computing Fibonacci Numbers via Binary Recursion

Let us consider the problem of computing the k th Fibonacci number. Recall from Section 2.2,

that the Fibonacci numbers are recursively defined as follows:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1\end{aligned}$$

By directly applying this definition, Algorithm BinaryFib, shown below, computes the sequence of Fibonacci numbers using binary recursion.

Algorithm BinaryFib(k):

Input: Nonnegative integer k
Output: The k th Fibonacci number F_k

```

if  $k \leq 1$  then
    return  $k$ 
else
    return BinaryFib( $k - 1$ ) + BinaryFib( $k - 2$ )

```

Unfortunately, in spite of the Fibonacci definition looking like a binary recursion, using this technique is inefficient in this case. In fact, it takes an exponential number of calls to compute the k th Fibonacci number in this way. Specifically, let n_k denote the number of calls performed in the execution of BinaryFib(k). Then, we have the following values for the n_k 's:

$$\begin{aligned}n_0 &= 1 \\n_1 &= 1 \\n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67\end{aligned}$$

If we follow the pattern forward, we see that the number of calls more than doubles for each two consecutive indices. That is, n_4 is more than twice n_2 , n_5 is more than twice n_3 , n_6 is more than twice n_4 , and so on. Thus, $n_k > 2^{k/2}$, which means that BinaryFib(k) makes a number of calls that are exponential in k . In other words, using binary recursion to compute Fibonacci numbers is very inefficient.

Computing Fibonacci Numbers via Linear Recursion

The main problem with the approach above, based on binary recursion, is that the computation of Fibonacci numbers is **really** a linearly recursive problem. It is not a good candidate for using binary recursion. We simply got tempted into using binary recursion because of the way the k th Fibonacci number, F_k , depends on the two previous values, F_{k-1} and F_{k-2} . But we can compute F_k much more efficiently using linear recursion.

In order to use linear recursion, however, **we need to slightly redefine the problem**. One way to accomplish this conversion is to define a recursive function that computes a pair of consecutive Fibonacci numbers (F_k, F_{k-1}) using the convention $F_{-1} = 0$. Then we can use the linearly recursive algorithm shown below.

```

Algorithm LinearFibonacci( $k$ ):
    Input: A nonnegative integer  $k$ 
    Output: Pair of Fibonacci numbers ( $F_k, F_{k-1}$ )
    if  $k \leq 1$  then
        return ( $k, 0$ )
    else
         $(i, j) \leftarrow \text{LinearFibonacci}(k - 1)$ 
        return ( $i + j, i$ )

```

The algorithm given in above shows that using linear recursion to compute Fibonacci numbers is much more efficient than using binary recursion. Since each recursive call to LinearFibonacci decreases the argument k by 1, the original call LinearFibonacci(k) results in a series of $k - 1$ additional calls. That is, computing the k th Fibonacci number via linear recursion requires k function calls. This performance is significantly faster than the exponential time needed by the algorithm based on binary recursion. Therefore, **when using binary recursion, we should first try to fully partition the problem in two** (as we did for summing the elements of an array) or we should be sure that overlapping recursive calls are really necessary.

Usually, we can eliminate overlapping recursive calls by using more memory to keep track of previous values. In fact, this approach is a central part of a technique called **dynamic programming**, which is related to recursion and is discussed in Section 12.2. (@todo mark)

- Multiple Recursion

Generalizing from binary recursion, we use **multiple recursion** when a function may make multiple recursive calls, with that number potentially being more than two. One of the most common applications of this type of recursion is used when we want to enumerate various configurations in order to **solve a combinatorial puzzle**. For example, the following are all instances of **summation puzzles**.

$$\begin{aligned}
 pot + pan &= bib \\
 dog + cat &= pig \\
 boy + girl &= baby
 \end{aligned}$$

To solve such a puzzle, **we need to assign a unique digit (that is, 0, 1, . . . , 9) to each letter in the equation, in order to make the equation true**. Typically, we solve such a puzzle by using our human observations of the particular puzzle we are trying to solve to eliminate configurations (that is, possible partial assignments of digits to letters) until we can work through the feasible configurations left, testing for the correctness of each one.

If the number of possible configurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one, without employing any human observations. In addition, such **an algorithm can use multiple recursion to work through the configurations in a systematic way**. We show pseudocode for such an algorithm below. To keep the description general enough to be used with other puzzles, the algorithm enumerates and tests all k -length sequences without repetitions of the elements of a given set U . We build the sequences of k elements by the following steps:

1. Recursively generating the sequences of $k - 1$ elements
2. Appending to each such sequence an element not already contained in it.

Throughout the execution of the algorithm, we use the set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U .

Another way to look at the algorithm below is that **it enumerates every possible size- k ordered subset of U , and tests each subset for being a possible solution to our puzzle.**

For summation puzzles, $U = \{0,1,2,3,4,5,6,7,8,9\}$ and each position in the sequence corresponds to a given letter. For example, the first position could stand for b , the second for o , the third for y , and so on.

Algorithm `PuzzleSolve(k, S, U):`

Input: An integer k , sequence S , and set U

Output: An enumeration of all k -length extensions to S using elements in U without repetitions

for each e in U **do**

 Remove e from U { e is now being used}

 Add e to the end of S

if $k = 1$ **then**

 Test whether S is a configuration that solves the puzzle

if S solves the puzzle **then**

return "Solution found: " S

else

 PuzzleSolve($k - 1, S, U$)

 Add e back to U { e is now unused}

 Remove e from the end of S

In Figure 3.21, we show a recursion trace of a call to `PuzzleSolve(3, S, U)`, where S is empty and $U = \{a, b, c\}$. During the execution, all the permutations of the three characters are generated and tested. Note that the initial call makes three recursive calls, each of which in turn makes two more. If we had executed `PuzzleSolve(3, S, U)` on a set U consisting of four elements, the initial call would have made four recursive calls, each of which would have a trace looking like the one in Figure 3.21.

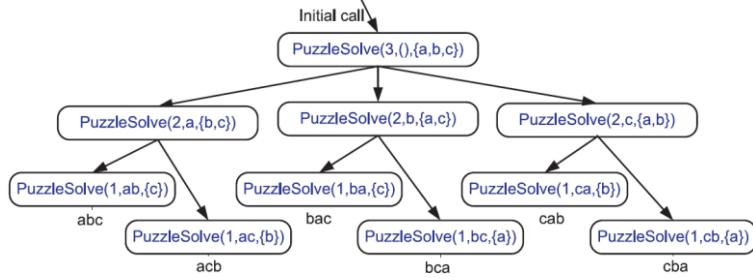


Figure 3.21 Recursion trace for an execution of `PuzzleSolve(3, S, U)`, where S is empty and $U = \{a, b, c\}$. This execution generates and tests all permutations of a , b , and c . We show the permutations generated directly below their respective boxes.

CHAPTER 4: Analysis Tools

4.1 The Seven Functions Used in This Book

In this section, we briefly discuss the seven most important functions used in the analysis of algorithms. We use only these seven simple functions for almost all the analysis we do in this book. In fact, sections that use a function other than one of these seven are marked with a star (*) to indicate that they are optional.

● The Constant Function

The simplest function we can think of is the **constant function**. This is the function,

$$f(n) = c,$$

for some fixed constant c , such as $c = 5, c = 27$, or $c = 2^{10}$. That is, for any argument n , the constant function $f(n)$ assigns the value c . In other words, it doesn't matter what the value of n is, $f(n)$ is always be equal to the constant value c .

Since we are most interested in integer functions, **the most fundamental constant function is $g(n) = 1$** , and this is the typical constant function we use in this book. Note that any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$. That is, $f(n) = cg(n)$ in this case.

As simple as it is, the constant function is useful in algorithm analysis because **it characterizes the number of steps needed to do a basic operation on a computer**, like adding two numbers, assigning a value to some variable, or comparing two numbers.

● The Logarithm Function

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the **logarithm function**, $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows:

$$x = \log_b n \quad \text{if and only if} \quad b^x = n.$$

By definition, $\log_b 1 = 0$. The value b is known as the **base** of the logarithm.

Computing the logarithm function exactly for any integer n involves the use of calculus, but we can use **an approximation that is good enough for our purposes without calculus**. In particular, we can easily compute the smallest integer greater than or equal $\log_a n$, since this number is equal to **the number of times we can divide n by a until we get a number less than or equal to 1**. For example, this evaluation of $\log_3 27$ is 3, since $27/3/3/3 = 1$. Likewise, this evaluation of $\log_4 64$ is 3, since $64/4/4/4 = 1$, and this approximation to $\log_2 12$ is 4, since $12/2/2/2/2 = 0.75 \leq 1$. This base-2 approximation arises in algorithm analysis, since **a common operation in many algorithms is to repeatedly divide an input in half**.

Indeed, since computers store integers in binary, **the most common base for the logarithm function in computer science is 2**. In fact, this base is so common that we typically leave it off when it is 2. That is, for us,

$$\log n = \log_2 n.$$

We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base 10, not base 2.

Proposition 4.1 (Logarithm Rules): Given real numbers $a > 0, b > 1, c > 0$ and $d > 1$, we have:

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$

$$4. \log_b a = (\log_d a) / \log_d b$$

$$5. b^{\log_d a} = a^{\log_d b}$$

● The Linear Function

Another simple yet important function is the **linear function**,

$$f(n) = n.$$

That is, given an input value n , the linear function f assigns the value n itself.

This function arises in algorithm analysis any time we have to do **a single basic operation for each of n elements**. For example, comparing a number x to each element of an array of size n requires n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects that are not already in the computer's memory, since reading in the n objects itself requires n operations.

● The N-Log-N Function

The next function we discuss in this section is the **n -log- n function**,

$$f(n) = n \log n.$$

That is, the function that assigns to an input n **the value of n times the logarithm base 2 of n** . This function grows a little faster than the linear function and a lot slower than the quadratic function. Thus, as we show on several occasions, if we can improve the running time of solving some problem from quadratic to n -log- n , **we have an algorithm that runs much faster in general**.

● The Quadratic Function

Another function that appears quite often in algorithm analysis is the **quadratic function**,

$$f(n) = n^2.$$

That is, given an input value n , the function f assigns the product of n with itself (in other words, " n squared").

The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have **nested loops**, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

Nested Loops and the Quadratic Function

The quadratic function can also arise in the context of **nested loops** where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, **the number of operations** is

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n.$$

In other words, this is the total number of operations that are performed by **the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop**. This quantity also has an interesting history.

In 1787, a German schoolteacher decided to keep his 9- and 10-year-old pupils occupied by adding up the integers from 1 to 100. But almost immediately one of the children claimed to have the answer! The teacher was suspicious, for the student had only the answer on his slate. But the answer was correct—5,050—and the student, Carl Gauss, grew up to be one of the greatest mathematicians of his time. It is widely suspected that young Gauss used the following identity.

Proposition 4.3: For any integer $n \geq 1$, we have:

$$1 + 2 + 3 + \dots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}.$$

The lesson to be learned from Proposition 4.3 is that if we perform *an algorithm with nested loops such that the operations in the inner loop increase by one each time*, then the total number of operations is quadratic in the number of times, n , we perform the outer loop. In particular, the number of operations is $n^2/2 + n/2$, in this case, which is a little more than a constant factor ($1/2$) times the quadratic function n^2 . In other words, such an algorithm is only slightly better than an algorithm that uses n operations each time the inner loop is performed. This observation might at first seem nonintuitive, but it is nevertheless true as shown in *Figure 4.1*.

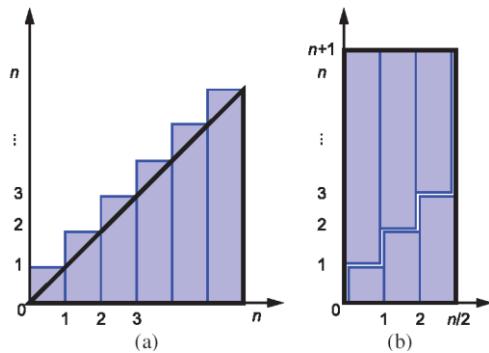


Figure 4.1 Visual justifications of Proposition 4.3. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \dots, n$. In (a), the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$ (along the y-axis, $1+n = n+1, 2+n-1 = n+1$, and so on).

● The Cubic Function and Other Polynomials

Continuing our discussion of functions that are *powers of the input*, we consider the **cubic function**,

$$f(n) = n^3,$$

which **assigns to an input value n the product of n with itself three times**. This function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Polynomials

Interestingly, the functions we have listed so far can be viewed as all being part of a larger class of functions, the **polynomials**.

A **polynomial** function is a function of the form,

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots + a_dn^d,$$

where a_0, a_1, \dots, a_d are constants, called the **coefficients** of the polynomial, and $a_d \neq 0$. Integer d , which indicates the highest power in the polynomial, is called the **degree** of the polynomial.

For example, the following functions are all polynomials:

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Therefore, we could argue that this book presents **just four important functions used in**

algorithm analysis, but we stick to saying that there are seven, since the constant, linear, and quadratic functions are too important to be lumped in with other polynomials. **Running times that are polynomials with degree, d , are generally better than polynomial running times of larger degree.**

Summations

A notation that appears again and again in the analysis of data structures and algorithms is the **summation**, which is defined as

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \dots + f(b),$$

where a and b are integers and $a \leq b$. Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations.

Using a summation, we can rewrite the formula of Proposition 4.3 as

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

Likewise, we can write a polynomial $f(n)$ of degree d with coefficients a_0, \dots, a_d as

$$f(n) = \sum_{i=0}^d a_i n^i.$$

Thus, **the summation notation gives us a shorthand way of expressing sums of increasing terms that have a regular structure.**

● The Exponential Function

Another function used in the analysis of algorithms is the **exponential function**,

$$f(n) = b^n,$$

where b is a positive constant, called the **base**, and the argument n is the **exponent**. That is, function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the n th iteration is 2^n . In addition, **an integer word containing n bits can represent all the nonnegative integers less than 2^n .** Thus, the exponential function with base 2 is quite common. The exponential function is also referred to as **exponent function**.

We sometimes have other exponents besides n , however; hence, it is useful for us to know a few handy rules for working with exponents. In particular, the following **exponent rules** are quite helpful.

Proposition 4.4 (Exponent Rules): Given positive integers a , b , and c , we have:

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

For example, we have the following:

- $256 = 16^2 = (2^4)^2 = 2^{4 \cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

We can extend the exponential function to exponents that are fractions or real numbers and to negative exponents, as follows. Given a positive integer k , we define $b^{1/k}$ to be k th root of b ,

that is, the number r such that $r^k = b$. For example, $25^{1/2} = 5$, since $5^2 = 25$. Likewise, $27^{1/3} = 3$ and $16^{1/4} = 2$. This approach allows us to define any power whose exponent can be expressed as a fraction, since $b^{a/c} = (b^a)^{1/c}$, by Exponent Rule 1. For example, $9^{3/2} = (9^3)^{1/2} = 729^{1/2} = 27$. Thus, $b^{a/c}$ is really just the c th root of the integral exponent b^a .

We can further extend the exponential function to define b^x for any real number x , by computing a series of numbers of the form $b^{a/c}$ for fractions a/c that get progressively closer and closer to x . Any real number x can be approximated arbitrarily close by a fraction a/c ; hence, we can use the fraction a/c as the exponent of b to get arbitrarily close to b^x . So, for example, the number 2^π is well defined. Finally, given a negative exponent d , we define $b^d = 1/b^{-d}$, which corresponds to applying Exponent Rule 3 with $a = 0$ and $c = -d$.

Geometric Sums

Suppose we have a loop where each iteration takes a multiplicative factor longer than the previous one. This loop can be analyzed using the following proposition.

Proposition 4.5: For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

(remembering that $a^0 = 1$ if $a > 0$). This summation is equal to

$$\frac{a^{n+1} - 1}{a - 1}.$$

Summations as shown in Proposition 4.5 are called **geometric** summations, because each term is geometrically larger than the previous one if $a > 1$. For example, **everyone working in computing should know** that

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

since this is the **largest integer** that can be represented in binary notation using n bits.

• Comparing Growth Rates

To sum up, *Table 4.1* shows each of the seven common functions used in algorithm analysis in order.

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ideally, we would like **data structure operations to run in times proportional to the constant or logarithm function**, and we would like our **algorithms to run in linear or n-log-n time**. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 4.2.

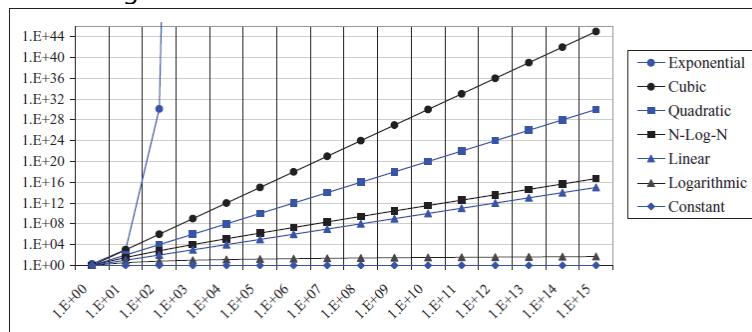


Figure 4.2 Growth rates for the seven fundamental functions used in algorithm analysis. We

use base $a = 2$ for the exponential function. The functions are plotted in a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart. Also, we use the scientific notation for numbers, where $aE + b$ denotes $a10^b$.

The Ceiling and Floor Functions

One additional comment concerning the functions above is in order. The value of a logarithm is typically not an integer, yet the running time of an algorithm is usually expressed by means of an integer quantity, such as the number of operations performed. Thus, the analysis of an algorithm may sometimes involve the use of the **floor function** and **ceiling function**, which are defined respectively as follows:

- $\lfloor x \rfloor$ = the largest integer less than or equal to x
- $\lceil x \rceil$ = the smallest integer greater than or equal to x

4.2 Analysis of Algorithms

In a classic story, the famous mathematician Archimedes was asked to **determine if a golden crown commissioned by the king was indeed pure gold, and not part silver**, as an informant had claimed. Archimedes discovered a way to perform this analysis while stepping into a (Greek) bath. He noted that water spilled out of the bath in proportion to the amount of him that went in. Realizing the implications of this fact, he immediately got out of the bath and ran naked through the city shouting, "Eureka, eureka!" for he had discovered an analysis tool (displacement), which, when combined with a simple scale, could determine if the king's new crown was good or not. That is, Archimedes could dip the crown and an equal-weight amount of gold into a bowl of water to see if they both displaced the same amount. This discovery was unfortunate for the goldsmith, however, for when Archimedes did his analysis, the crown displaced more water than an equal-weight lump of pure gold, indicating that the crown was not, in fact, pure gold.

In this book, we are interested in the design of "good" data structures and algorithms. Simply put, **a data structure** is *a systematic way of organizing and accessing data*, and **an algorithm** is *a step-by-step procedure for performing some tasks in a finite amount of time*. These concepts are central to computing, but to be able to classify some data structures and algorithms as "good," we must have precise ways of analyzing them.

The primary analysis tool we use in this book involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of "goodness," since time is a precious resource—computer solutions should run as fast as possible.

In general, the running time of an algorithm or data structure method increases with the input size, although it may also vary for different inputs of the same size. Also, **the running time is affected by the hardware environment (as reflected in the processor, clock rate, memory, disk, etc.) and software environment (as reflected in the operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed**. All other factors being equal, the running time of the same algorithm on the same input data is smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation run on a virtual machine. Nevertheless, in spite of the possible variations that come from different environmental factors, **we would like to focus on the relationship between the running time of an algorithm and the size of its input**.

We are interested in characterizing an algorithm's running time as a function of the input size.

But what is the proper way of measuring it?

- Experimental Studies

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution. Fortunately, such measurements can be taken in an accurate manner by using system calls that are built into the language or operating system (for example, by using the `clock()` function or calling the run-time environment with profiling enabled). Such tests assign a specific running time to a specific input size, but we are interested in determining **the general dependence of running time on the size of the input**. In order to determine this dependence, we should perform several experiments on many different test inputs of various sizes. Then we can visualize the results of such experiments by plotting the performance of each run of the algorithm as a point with x -coordinate equal to the input size, n , and y -coordinate equal to the running time, t . (See Figure 4.3.) From this visualization and the data that supports it, we can perform a statistical analysis that seeks to fit the best function of the input size to the experimental data. **To be meaningful, this analysis requires that we choose good sample inputs and test enough of them** to be able to make sound statistical claims about the algorithm's running time.

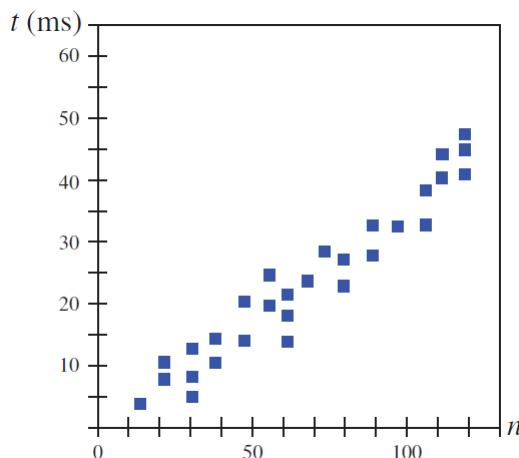


Figure 4.3 Results of an experimental study on the running time of an algorithm. A dot with coordinates (n, t) indicates that on an input of size n , the running time of the algorithm is t milliseconds (ms).

While experimental studies of running times are useful, they have three major limitations:

- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- We have difficulty comparing the experimental running times of two algorithms unless the experiments were performed in the same hardware and software environments.
- We have to fully implement and execute an algorithm in order to study its running time experimentally.

This last requirement is obvious, but it is probably the most time consuming aspect of performing an experimental analysis of an algorithm. The other limitations impose serious hurdles too, of course. **Thus, we would ideally like to have an analysis tool that allows us to avoid performing experiments.**

In the rest of this chapter, we develop **a general way of analyzing the running times of algorithms** that:

- Takes into account all possible inputs.
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment.
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

This methodology aims at associating, with each algorithm, a function $f(n)$ that characterizes the running time of the algorithm as a function of the input size n . Typical functions that are encountered include the seven functions mentioned earlier in this chapter.

● Primitive Operations

As noted above, experimental analysis is valuable, but it has its limitations. If we wish to analyze a particular algorithm without performing experiments on its running time, we can perform **an analysis directly on the high-level pseudo-code** instead. We define a set of **primitive operations** such as the following:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a function

Counting Primitive Operations

Specifically, **a primitive operation corresponds to a low-level instruction with an execution time that is constant**. Instead of trying to determine the specific execution time of each primitive operation, we simply **count how many primitive operations are executed**, and use this number t as a measure of the running time of the algorithm.

This operation count correlates to an actual running time in a specific computer, since each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations is fairly similar. Thus, the number, t , of primitive operations an algorithm performs is proportional to the actual running time of that algorithm.

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Unfortunately, such an **average-case analysis is typically quite challenging**. It requires us to define a probability distribution on the set of inputs, which is often a difficult task. Figure 4.4 schematically shows how, depending on the input distribution, the running time of an algorithm can be anywhere between the worst-case time and the best-case time. For example, what if inputs are really only of types "A" or "D"?

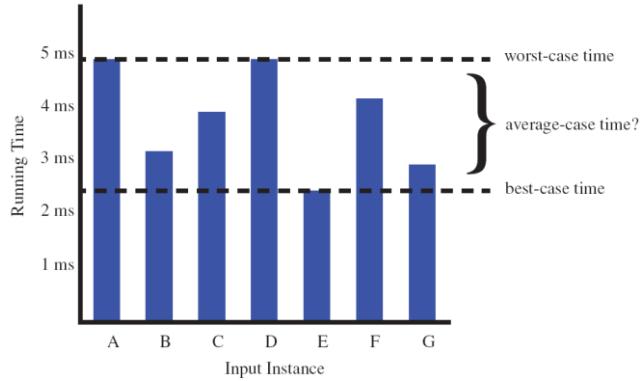


Figure 4.4 The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Focusing on the Worst Case

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. Therefore, for the remainder of this book, unless we specify otherwise, we characterize running times in terms of the **worst case**, as a function of the input size, n , of the algorithm.

Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple. Also, this approach typically leads to better algorithms. *Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it does well on every input.* That is, designing for the worst case leads to stronger algorithmic "muscles," much like a track star who always practices by running up an incline.

● Asymptotic Notation

In general, each basic step in a pseudo-code description or a high-level language implementation corresponds to a small number of primitive operations (except for function calls, of course). Thus, we can perform a simple analysis of an algorithm written in pseudo-code that estimates the number of primitive operations executed up to a constant factor, by pseudo-code steps (but we must be careful, since a single line of pseudo-code may denote a number of steps in some cases).

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n , taking a "big-picture" approach. It is often enough just to know that the running time of an algorithm such as arrayMax, shown in below, **grows proportionally to n** , with its true running time being n times a constant factor that depends on the specific computer.

```

Algorithm arrayMax( $A, n$ ):
  Input: An array  $A$  storing  $n \geq 1$  integers.
  Output: The maximum element in  $A$ .
  currMax  $\leftarrow A[0]$ 
  for  $i \leftarrow 1$  to  $n - 1$  do
    if currMax <  $A[i]$  then
      currMax  $\leftarrow A[i]$ 
  return currMax

```

We analyze algorithms using a mathematical notation for functions that disregards constant factors. **Namely, we characterize the running times of algorithms by using functions that map the size of the input, n , to values that correspond to the main factor that determines the growth rate in terms of n .** This approach allows us to focus on the "big-picture" aspects of an algorithm's running time.

The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$.

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as " $f(n)$ is **big-Oh** of $g(n)$." Alternatively, we can also say " $f(n)$ is **order of** $g(n)$."

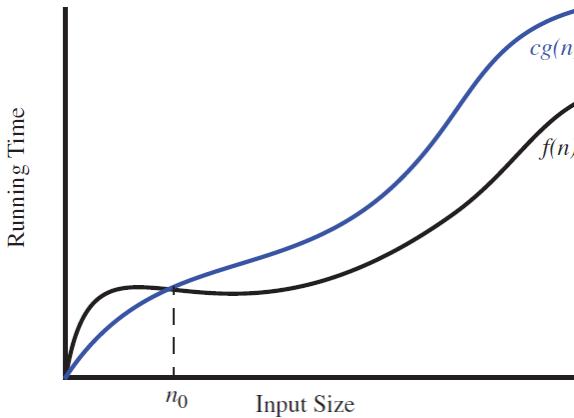


Figure 4.5 The "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Example 4.6: The function $8n - 2$ is $O(n)$.

Justification: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $8n - 2 \leq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 8$ and $n_0 = 1$. Indeed, this is one of infinitely many choices available because any real number greater than or equal to 8 works for c , and any integer greater than or equal to 1 works for n_0 .

The big-Oh notation allows us to say that a function $f(n)$ is "less than or equal to" another function $g(n)$ up to a constant factor and in the **asymptotic** sense as n grows toward infinity. This ability comes from the fact that the definition uses " \leq " to compare $f(n)$ to a $g(n)$ times a constant, c , for the asymptotic cases when $n \geq n_0$.

Characterizing Running Times using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n , which varies from problem to problem, but is always defined as a chosen measure of the "size" of the problem. For example, if we are interested in finding the largest element in an array of integers, as in the `arrayMax` algorithm, we should let n denote the number of elements of the array. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of algorithm `arrayMax` for **any** computer.

Proposition 4.7: The Algorithm `arrayMax`, for computing the *maximum* element in an array of n integers, runs in $O(n)$ time.

Justification: The number of primitive operations executed by algorithm `arrayMax` in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm `arrayMax` on an input of size n is at most a constant times n , that is, we may conclude that the running time of algorithm `arrayMax` is $O(n)$.

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth.

Example 4.8: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Justification: Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$, for $c = 15$, when $n \geq n_0 = 1$.

In fact, we can characterize the growth rate of any polynomial function.

Proposition 4.9: If $f(n)$ is a polynomial of degree d , that is,

$$f(n) = a_0 + a_1n + \dots + a_dn^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \dots \leq n^d$; hence,

$$a_0 + a_1n + a_2n^2 + \dots + a_dn^d \leq (a_0 + a_1 + a_2 + \dots + a_d)n^d.$$

Therefore, we can show $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 + \dots + a_d$ and $n_0 = 1$.

Thus, **the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial**. We consider some additional properties of the big-Oh notation in the exercises. Let us consider some further examples here, however, focusing on combinations of the seven fundamental functions used in algorithm design.

Example 4.10: $5n^2 + 3n \log n + 2n + 5$ is $O(n^2)$.

Justification: $5n^2 + 3n \log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 2$ (note that $n \log n$ is zero for $n = 1$).

Example 4.11: $20n^3 + 10n \log n + 5$ is $O(n^3)$.

Justification: $20n^3 + 10n \log n + 5 \leq 35n^3$, for $n \geq 1$.

Example 4.12: $3 \log n + 2$ is $O(\log n)$.

Justification: $3 \log n + 2 \leq 5 \log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case.

Example 4.13: 2^{n+2} is $O(2^n)$.

Justification: $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case.

Example 4.14: $2n + 100 \log n$ is $O(n)$.

Justification: $2n + 100 \log n \leq 102n$, for $n \geq n_0 = 2$; hence, we can take $c = 102$ in this case.

Characterizing Functions in Simplest Terms

In general, **we should use the big-Oh notation to characterize a function as closely as possible**. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$. Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can find some food, it may be truthful for the farmer to say, "certainly no longer than 12 hours," but it is much more accurate (and helpful) for him to say, "you can find a market just a few minutes drive up this road." Thus, even with the big-Oh notation, we should strive as much as possible to tell the whole truth.

It is also considered poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n \log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in **simplest terms**.

The seven functions listed in Section 4.1 are the most common functions used in conjunction

with the big-Oh notation to characterize the running times and space usage of algorithms. Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time $4n^2 + n \log n$ is a **quadratic-time** algorithm, since it runs in $O(n^2)$ time. Likewise, an algorithm running in time at most $5n + 20 \log n + 4$ would be called a **linear-time** algorithm.

Big-Omega

Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, the following notations provide **an asymptotic way of saying that a function grows at a rate that is "greater than or equal to" that of another.**

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ (pronounced " $f(n)$ is big-Omega of $g(n)$ ") if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq cg(n), \text{ for } n \geq n_0.$$

This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Example 4.15: $3n \log n + 2n$ is $\Omega(n \log n)$.

Justification: $3n \log n + 2n \geq 3n \log n$, for $n \geq 2$.

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that $f(n)$ is $\Theta(g(n))$ (pronounced " $f(n)$ is big-Theta of $g(n)$ ") if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \text{ for } n \geq n_0.$$

Example 4.16: $3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$.

Justification: $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5)n \log n$, for $n \geq 2$.

● Asymptotic Analysis

Suppose two algorithms solving the same problem are available: an algorithm A , which has a running time of $O(n)$, and an algorithm B , which has a running time of $O(n^2)$. Which algorithm is better? We know that n is $O(n^2)$, which implies that algorithm A is **asymptotically better** than algorithm B , although for a small value of n , B may have a lower running time than A .

We can use the big-Oh notation to order classes of functions by asymptotic growth rate. Our seven functions are ordered by increasing growth rate in the sequence below, that is, if a function $f(n)$ precedes a function $g(n)$ in the sequence, then $f(n)$ is $O(g(n))$:

$$1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n.$$

We illustrate the growth rates of some important functions in Table 4.2.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.2: Selected values of fundamental functions in algorithm analysis.

We further illustrate the importance of the asymptotic viewpoint in Table 4.3. This table explores **the maximum size allowed for an input instance that is processed by an algorithm in 1 second, 1 minute, and 1 hour**. It shows the importance of good algorithm design, because an asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse.

Running Time (μs)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 4.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

The importance of good algorithm design goes beyond just what can be solved effectively on a given computer, however. As shown in Table 4.4, even if we achieve a dramatic speed-up in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm. This table shows the new maximum problem size achievable for any fixed amount of time, assuming algorithms with the given running times are now run on a computer 256 times faster than the previous one.

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

Table 4.4: Increase in the maximum size of a problem that can be solved in a fixed amount of time by using a computer that is 256 times faster than the previous one. Each entry is a function of m , the previous maximum problem size.

● Using the Big-Oh Notation

Having made the case of using the big-Oh notation for analyzing algorithms, let us briefly discuss a few issues concerning its use. **It is considered poor taste, in general, to say " $f(n) \leq O(g(n))$," since the big-Oh already denotes the "less-than-or-equal-to" concept.** Likewise, although common, it is not fully correct to say " $f(n) = O(g(n))$ " (with the usual understanding of the " $=$ " relation), since there is no way to make sense of the statement " $O(g(n)) = f(n)$." In addition, it is completely wrong to say " $f(n) \geq O(g(n))$ " or " $f(n) > O(g(n))$," since the $g(n)$ in the big-Oh expresses an upper bound on $f(n)$. It is best to say,

" $f(n)$ is $O(g(n))$."

For the more mathematically inclined, it is also correct to say,

" $f(n) \in O(g(n))$,"

for the big-Oh notation is, technically speaking, denoting a whole collection of functions. In this book, we stick to presenting big-Oh statements as " $f(n)$ is $O(g(n))$." Even with this interpretation, there is considerable freedom in how we can use arithmetic operations with the big-Oh notation, and with this freedom comes a certain amount of responsibility.

Some Words of Caution

A few words of caution about asymptotic notation are in order at this point. **First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large.** For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we prefer the $O(n \log n)$ time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100} , which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding."

The observation above raises the issue of what constitutes a "fast" algorithm. Generally speaking, any algorithm running in $O(n \log n)$ time (with a reasonable constant factor) should be considered efficient. **Even an $O(n^2)$ time method may be fast enough in some contexts, that is, when n is small.** But an algorithm running in $O(2^n)$ time should almost never be considered efficient.

Exponential Running Times

There is a famous story about the inventor of the game of chess. He asked only that his king pay him 1 grain of rice for the first square on the board, 2 grains for the second, 4 grains for the third, 8 for the fourth, and so on. It is an interesting test of programming skills to write a program to compute exactly the number of grains of rice the king would have to pay. In fact, any C++ program written to compute this number in a single integer value causes an integer overflow to occur (although the run-time machine probably won't complain).

If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinction between algorithms with a running time that is $O(n^c)$, for some constant $c > 1$, and those with a running time that is $O(b^n)$, for some constant $b > 1$. Like so many notions we have discussed in this section, this too should be taken with a "grain of salt," for an algorithm running in $O(n^{100})$ time should probably not be considered "efficient." Even so, **the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.**

To summarize, the asymptotic notations of big-Oh, big-Omega, and big-Theta provide a convenient language for us to analyze data structures and algorithms. **As mentioned earlier, these notations provide convenience because they let us concentrate on the "big picture" rather than low-level details.**

Two Examples of Asymptotic Algorithm Analysis

We conclude this section by analyzing **two algorithms that solve the same problem but have rather different running times**. The problem we are interested in is the one of computing the so-called **prefix averages** of a sequence of numbers. Namely, given an array X storing n numbers, we want to compute an array A such that $A[i]$ is the average of elements $X[0], \dots, X[i]$, for $i = 0, \dots, n - 1$, that is,

$$A[i] = \frac{\sum_{j=0}^i X[j]}{i + 1}.$$

Computing prefix averages has many applications in economics and statistics. For example, given the year-by-year returns of a mutual fund, an investor typically wants to see the fund's average annual returns for the last year, the last three years, the last five years, and the last ten years. Likewise, given a stream of daily Web usage logs, a Web site manager may wish to track

average usage trends over various time periods.

A Quadratic-Time Algorithm

Our first algorithm for the **prefix averages problem**, called `prefixAverages1`, is shown below. It computes every element of A separately, following the definition.

Algorithm `prefixAverages1(X)`:

Input: An n -element array X of numbers.
Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

Let A be an array of n numbers.

```
for  $i \leftarrow 0$  to  $n - 1$  do
     $a \leftarrow 0$ 
    for  $j \leftarrow 0$  to  $i$  do
         $a \leftarrow a + X[j]$ 
         $A[i] \leftarrow a/(i + 1)$ 
return array  $A$ 
```

Let us analyze the `prefixAverages1` algorithm.

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element and takes $O(n)$ time.
- There are two nested **for** loops that are controlled by counters i and j , respectively. The body of the outer loop, controlled by counter i , is executed n times for $i = 0, \dots, n - 1$. Thus, statements $a = 0$ and $A[i] = a/(i + 1)$ are executed n times each. This implies that these two statements, plus the incrementing and testing of counter i , contribute a number of primitive operations proportional to n , that is, $O(n)$ time.
- The body of the inner loop, which is controlled by counter j , is executed $i + 1$ times, depending on the current value of the outer loop counter i . Thus, statement $a = a + X[j]$ in the inner loop is executed $1 + 2 + 3 + \dots + n$ times. By recalling Proposition 4.3, we know that $1 + 2 + 3 + \dots + n = n(n + 1)/2$, which implies that the statement in the inner loop contributes $O(n^2)$ time. A similar argument can be done for the primitive operations associated with the incrementing and testing counter j , which also take $O(n^2)$ time.

The running time of algorithm `prefixAverages1` is given by the sum of three terms. The first and the second term are $O(n)$, and the third term is $O(n^2)$. By a simple application of Proposition 4.9, the running time of `prefixAverages1` is $O(n^2)$.

A Linear-Time Algorithm

In order to compute prefix averages more efficiently, we can observe that two consecutive averages $A[i - 1]$ and $A[i]$ are similar:

$$\begin{aligned} A[i - 1] &= (X[0] + X[1] + \dots + X[i - 1])/i \\ A[i] &= (X[0] + X[1] + \dots + X[i - 1] + X[i])/(i + 1). \end{aligned}$$

If we denote with S_i the **prefix sum** $X[0] + X[1] + \dots + X[i]$, we can compute the prefix averages as $A[i] = S_i/(i + 1)$. It is easy to keep track of the current prefix sum while scanning array X with a loop. We are now ready to present Algorithm `prefixAverages2` below.

Algorithm prefixAverages2(X):

Input: An n -element array X of numbers.
Output: An n -element array A of numbers such that $A[i]$ is
the average of elements $X[0], \dots, X[i]$.
Let A be an array of n numbers.
 $s \leftarrow 0$
for $i \leftarrow 0$ **to** $n - 1$ **do**
 $s \leftarrow s + X[i]$
 $A[i] \leftarrow s/(i + 1)$
return array A

The analysis of the running time of algorithm prefixAverages2 follows:

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element, and takes $O(n)$ time.
- Initializing variable s at the beginning takes $O(1)$ time.

There is a single **for** loop, which is controlled by counter i . The body of the loop is executed n times, for $i = 0, \dots, n - 1$. Thus, statements $s = s + X[i]$ and $A[i] = s/(i + 1)$ are executed n times each. This implies that these two statements plus the incrementing and testing of counter i contribute a number of primitive operations proportional to n , that is, $O(n)$ time.

The running time of algorithm prefixAverages2 is given by the sum of three terms. The first and the third term are $O(n)$, and the second term is $O(1)$. By a simple application of Proposition 4.9, the running time of prefixAverages2 is $O(n)$, which is much better than the quadratic-time algorithm prefixAverages1.

● A Recursive Algorithm for Computing Powers

As a more interesting example of algorithm analysis, let us consider the problem of raising a number x to an arbitrary nonnegative integer, n . That is, we wish to compute the **power function** $p(x, n)$, defined as $p(x, n) = x^n$. This function has an immediate recursive definition based on linear recursion:

$$p(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n - 1) & \text{otherwise} \end{cases}$$

This definition leads immediately to a recursive algorithm that uses $O(n)$ function calls to compute $p(x, n)$. **We can compute the power function much faster than this**, however, by using the following alternative definition, also based on linear recursion, which employs a **squaring technique**:

$$p(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, (n - 1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

To illustrate how this definition works, consider the following examples:

$$\begin{aligned} 2^4 &= 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\ 2^5 &= 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\ 2^6 &= 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\ 2^7 &= 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128 \end{aligned}$$

```

Algorithm Power( $x, n$ ):
  Input: A number  $x$  and integer  $n \geq 0$ 
  Output: The value  $x^n$ 

  if  $n = 0$  then
    return 1
  if  $n$  is odd then
     $y \leftarrow \text{Power}(x, (n - 1)/2)$ 
    return  $x \cdot y \cdot y$ 
  else
     $y \leftarrow \text{Power}(x, n/2)$ 
    return  $y \cdot y$ 

```

To analyze the running time of the algorithm, we observe that each recursive call of function $\text{Power}(x, n)$ divides the exponent, n , by two. Thus, there are $O(\log n)$ recursive calls, not $O(n)$. That is, by using linear recursion and the squaring technique, **we reduce the running time for the computation of the power function from $O(n)$ to $O(\log n)$, which is a big improvement.**

- Some More Examples of Algorithm Analysis

Now that we have the big-Oh notation for doing algorithm analysis, let us give some more examples of simple algorithms that can have their running times characterized using this notation. Moreover, in keeping with our earlier promise, we illustrate below how each of the seven functions given earlier in this chapter can be used to characterize the running time of an example algorithm.

A Constant-Time Method

To illustrate a constant-time algorithm, consider the following C++ function, which returns the size of an STL vector, that is, the current number of cells in the array:

```

int capacity(const vector<int>& arr) {
  return arr.size();
}

```

This is a very simple algorithm, because the size of a vector is stored as a member variable in the vector object, so it takes only a constant-time lookup to return this value. **Thus, the capacity function runs in $O(1)$ time**; that is, the running time of this function is independent of the value of n , the size of the array.

Revisiting the Method for Finding the Maximum in an Array

For our next example, let us reconsider a simple problem studied earlier, finding the largest value in an array of integers. We assume that the array is stored as an STL vector. This can be done in C++ as follows:

```

int findMax(const vector<int>& arr) {
  int max = arr[0];
  for (int i = 1; i < arr.size(); i++) {
    if (max < arr[i]) max = arr[i];
  }
  return max;
}

```

This function, which amounts to a C++ implementation of the `arrayMax` algorithm, compares each of the n elements in the input array to a current maximum, and each time it finds an element larger than the current maximum, it updates the current maximum to be this value. Thus, it spends a constant amount of time for each of the n elements in the array; **hence, as with the pseudocode version of the `arrayMax` algorithm, the running time of this**

algorithm is $O(n)$.

Further Analysis of the Maximum-Finding Algorithm

A more interesting question, with respect to the above maximum-finding algorithm, is to ask **how many times we update the current maximum value**. Note that this statement is executed only if we encounter a value of the array that is larger than our current maximum. In the worst case, this condition could be true each time we perform the test. For instance, this situation would occur if the input array is given to us in sorted order. Thus, in the worst-case, the statement $max = arr[i]$ is performed $n - 1$ times, hence $O(n)$ times.

But what if the input array is given to us in random order, with all orders equally likely; what would be the expected number of times we updated the maximum value in this case? To answer this question, note that we update the current maximum in the i th iteration only if the i th element in the array is bigger than all the elements that precede it. But if the array is given to us in random order, the probability that the i th element is larger than all elements that precede it is $1/i$; hence, the expected number of times we update the maximum in this case is $H_n = \sum_{i=1}^n 1/i$, which is known as the **n th Harmonic number**. It turns out (see Proposition A.16) that H_n is $O(\log n)$. Therefore, the expected number of times the maximum is updated when the above maximum-finding algorithm is run on a random array is $O(\log n)$.

Three-Way Set Disjointness

Suppose we are given three sets, A , B , and C , with these sets stored in three different integer arrays, a , b , and c , respectively. The **three-way set disjointness problem** is to determine if these three sets are disjoint, that is, whether there is no element x such that $x \in A$, $x \in B$, and $x \in C$. A simple C++ function to determine this property is given below:

```
bool areDisjoint(const vector<int>& a, const vector<int>& b,
                  const vector<int>& c) {
    for (int i = 0; i < a.size(); i++)
        for (int j = 0; j < b.size(); j++)
            for (int k = 0; k < c.size(); k++)
                if ((a[i] == b[j]) && (b[j] == c[k])) return false;
    return true;
}
```

This simple algorithm loops through each possible triple of indices i , j , and k to check if the respective elements indexed in a , b , and c are equal. Thus, if each of these arrays is of size n , then the worst-case running time of this function is $O(n^3)$. Moreover, the worst case is achieved when the sets are disjoint, since in this case we go through all n^3 triples of valid indices, i , j , and k . Such a running time would generally not be considered very efficient, but, fortunately, there is a better way to solve this problem, which we explore in Exercise C-4.3.

Supplement materials

(Exercise C-4.3) Assuming it is possible to sort n numbers in $O(n \log n)$ time, show that it is possible to solve the three-way set disjointness problem in $O(n \log n)$ time.

Recursion Run Amok

The next few example algorithms we study are for solving the **element uniqueness problem**, in which we are given a range, $i, i + 1, \dots, j$, of indices for an array, A , which we assume is given as an STL vector. We want to determine if the elements of this range, $A[i], A[i + 1], \dots, A[j]$, are all unique, that is, there is no repeated element in this group of array entries. The first algorithm we give for solving the element uniqueness problem is a recursive one. But it uses recursion in a very inefficient manner, as shown in the following C++ implementation.

```

bool isUnique(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    if (!isUnique(arr, start, end-1))
        return false;
    if (!isUnique(arr, start+1, end))
        return false;
    return (arr[start] != arr[end]);
}

```

You should first convince yourself that the function is correct. To analyze this recursive algorithm's running time, let us first determine how much time we spend outside of recursive calls in any invocation of this function. Note, in particular, that there are no loops—just comparisons, arithmetic operations, array element references, and function returns. Thus, the nonrecursive part of each function invocation runs in constant time, that is, $O(1)$ time; hence, to determine the worst-case running time of this function we only need to determine the worst-case total number of calls we make to the `isUnique` function.

Let n denote the number of entries under consideration, that is, let

$$n = end - start + 1.$$

If $n = 1$, then the running time of the `isUnique` is $O(1)$, since there are no recursive calls for this case. To characterize the running time of the general case, **the important observation to make is that in order to solve a problem of size n , the `isUnique` function makes two recursive calls on problems of size $n - 1$** . Thus, in the worst case, a call for a range of size n makes two calls on ranges of size $n - 1$, which each make two calls on ranges of size $n - 2$, which each make two calls on ranges of size $n - 3$, and so on. Thus, in the worst case, the total number of function calls is given by the geometric summation

$$1 + 2 + 4 + \dots + 2^{n-1},$$

which is equal to $2^n - 1$ by Proposition 4.5. Thus, the worst-case running time of function `isUnique` is $O(2^n)$. **This is an incredibly inefficient method for solving the element uniqueness problem.** Its inefficiency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly.

An Iterative Method for Solving the Element Uniqueness Problem

We can do much better than the above exponential-time method by using the following iterative algorithm:

```

bool isUniqueLoop(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    for (int i = start; i < end; i++)
        for (int j = i+1; j <= end; j++)
            if (arr[i] == arr[j]) return false;
    return true;
}

```

This function solves the element uniqueness problem by looping through all distinct pairs of indices, i and j , and checking if any of them indexes a pair of elements that are equal to each other. It does this using two nested **for** loops, such that the first iteration of the outer loop causes $n - 1$ iterations of the inner loop, the second iteration of the outer loop causes $n - 2$ iterations of the inner loop, the third iteration of the outer loop causes $n - 3$ iterations of the inner loop, and so on. Thus, the worst-case running time of this function is proportional to

$$1 + 2 + 3 + \dots + (n - 1),$$

which is $O(n^2)$ as we saw earlier in this chapter (Proposition 4.3).

Using Sorting as a Problem-Solving Tool

An even better algorithm for the element uniqueness problem is based on **using sorting as a problem-solving tool**. In this case, **by sorting an array of elements, we are guaranteed that**

any duplicate elements will be placed next to each other. Thus, it suffices to sort the array and look for duplicates among consecutive elements. A C++ implementation of this algorithm follows.

```
bool isUniqueSort(const vector<int>& arr, int start, int end) {
    if (start >= end) return true;
    vector<int> buf(arr);
    sort(buf.begin() + start, buf.begin() + end); // sort the subarray
    for (int i = start; i < end; i++) // check for duplicates
        if (buf[i] == buf[i+1]) return false;
    return true;
}
```

The function `sort` is provided by the STL. On most systems, it runs in $O(n \log n)$ time. Since the other steps run in $O(n)$ time, the entire algorithm runs in $O(n \log n)$ time. Incidentally, we can solve the element uniqueness problem even faster, at least in terms of its average-case running time, by using the hash table data structure we explore in Section 9.2.

4.3 Simple Justification Techniques

Sometimes, we want to make claims about an algorithm, such as showing that it is correct or that it runs fast. **In order to rigorously make such claims**, we must use *mathematical language*, and in order to back up such claims, we must justify or **prove** our statements. Fortunately, there are several simple ways to do this.

- By Example

Some claims are of the generic form, "**There is an element x in a set S that has property P .**" To justify such a claim, we only need to produce a particular x in S that has property P . Likewise, some hard-to-believe claims are of the generic form, "**Every element x in a set S has property P .**" To justify that such a claim is false, we only need to produce a particular x from S that does not have property P . Such an instance is called a **counterexample**.

Example 4.17: Professor Amongus claims that every number of the form $2^i - 1$ is a prime, when i is an integer greater than 1. Professor Amongus is wrong.

Justification: To prove Professor Amongus is wrong, we find a counter-example. Fortunately, we need not look too far, for $2^4 - 1 = 15 = 3 \cdot 5$.

- The "Contra" Attack

Another set of **justification techniques** involves the use of the negative. **The two primary such methods** are the use of the **contrapositive** and the **contradiction**. The use of the contrapositive method is like looking through a negative mirror. To justify the statement "if p is true, then q is true" we establish that "if q is not true, then p is not true" instead. Logically, these two statements are the same, but the latter, which is called **the contrapositive of the first, may be easier to think about**.

Supplement materials

In predicate logic, the contrapositive (or transposition) of the statement " p implies q " is "not- q implies not- p ." A statement and its contrapositive are always logically equivalent, unlike a statement's inverse or its converse.

Contradiction is a statement that is necessarily false.

Example 4.18: Let a and b be integers. If ab is even, then a is even or b is even.

Justification: To justify this claim, consider the contrapositive, "If a is odd and b is odd, then

ab is odd." So, suppose $a = 2i + 1$ and $b = 2j + 1$, for some integers i and j . Then $ab = 4ij + 2i + 2j + 1 = 2(2i + i + j) + 1$; hence, ab is odd.

Besides showing a use of the contrapositive justification technique, the previous example also contains an application of **DeMorgan's Law**. This law helps us deal with negations, for it states that the negation of a statement of the form " p or q " is "not p and not q ." Likewise, it states that the negation of a statement of the form " p and q " is "not p or not q ."

Contradiction

Another **negative justification technique** is justification by **contradiction**, which also often involves using DeMorgan's Law. In applying the justification by contradiction technique, we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as $2 \neq 2$ or $1 > 3$). By reaching such a contradiction, we show that no consistent situation exists with q being false, so q must be true. Of course, in order to reach this conclusion, we must be sure our situation is consistent before we assume q is false.

Example 4.19: Let a and b be integers. If ab is odd, then a is odd and b is odd.

Justification: Let ab be odd. We wish to show that a is odd and b is odd. So, with the hope of leading to a contradiction, let us assume the opposite, namely, **suppose a is even or b is even**. In fact, without loss of generality, we can assume that a is even (since the case for b is symmetric). Then $a = 2i$ for some integer i . Hence, $ab = (2i)b = 2(ib)$, that is, ab is even. But this is a contradiction: ab cannot simultaneously be odd and even. Therefore a is odd and b is odd.

● Induction and Loop Invariants

Most of the claims we make about a running time or a space bound involve an integer parameter n (usually denoting an intuitive notion of the "size" of the problem). Moreover, most of these claims are equivalent to saying **some statement $q(n)$ is true "for all $n \geq 1$.** Since this is making a claim about an infinite set of numbers, we cannot justify this exhaustively in a direct fashion.

Induction

We can often justify claims such as those above as true, however, by using the technique of **induction**. This technique amounts to showing that, for any particular $n \geq 1$, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that $q(n)$ is true. Specifically, we begin a justification by induction by showing that $q(n)$ is true for $n = 1$ (and possibly some other values $n = 2, 3, \dots, k$, for some constant k). Then we justify that the inductive "step" is true for $n > k$, namely, we show "if $q(i)$ is true for $i < n$, then $q(n)$ is true." The combination of these two pieces completes the justification by induction.

Proposition 4.20: Consider the Fibonacci function $F(n)$, where we define $F(1) = 1$, $F(2) = 2$, and $F(n) = F(n - 1) + F(n - 2)$ for $n > 2$. (See Section 2.2.3.) We claim that $F(n) < 2^n$.

Justification: We show our claim is right by induction.

Base cases: ($n \leq 2$). $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$.

Induction step: ($n > 2$). Suppose our claim is true for $n' < n$. Consider $F(n)$. Since $n > 2$, $F(n) = F(n - 1) + F(n - 2)$. Moreover, since $n - 1 < n$ and $n - 2 < n$, we can apply the inductive assumption (sometimes called the "inductive hypothesis") to imply that $F(n) < 2^{n-1} + 2^{n-2}$, since

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

Let us do another inductive argument, this time for a fact we have seen before.

Proposition 4.21: (which is the same as Proposition 4.3)

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

Justification: We justify this equality by induction.

Base case: $n = 1$. Trivial, for $1 = n(n+1)/2$, if $n = 1$.

Induction step: $n \geq 2$. Assume the claim is true for $n' < n$. Consider n .

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i.$$

By the induction hypothesis, then

$$\sum_{i=1}^n i = n + \frac{(n-1)n}{2},$$

which we can simplify as

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}.$$

We may sometimes feel overwhelmed by the task of justifying something true for all $n \geq 1$. **We should remember, however, the concreteness of the inductive technique.** It shows that, for any particular n , there is a finite step-by-step sequence of implications that starts with something true and leads to the truth about n . In short, **the inductive argument is a formula for building a sequence of direct justifications.**

Loop Invariants

The final justification technique we discuss in this section is the **loop invariant**. To prove some statement S about a loop is correct, define S in terms of a series of smaller statements S_0, S_1, \dots, S_k , where:

1. The **initial** claim, S_0 , is true before the loop begins.
2. If S_{i-1} is true before iteration i , then S_i is true after iteration i .
3. The final statement, S_k , implies the statement S that we wish to be true.

Let us give a simple example of using **a loop-invariant argument to justify the correctness of an algorithm**. In particular, let us consider using a loop invariant to justify the correctness of arrayFind, shown below, for finding an element x in an array A .

Algorithm arrayFind(x, A):

Input: An element x and an n -element array, A .

Output: The index i such that $x = A[i]$ or -1 if no element of A is equal to x .

```

i ← 0
while i < n do
    if x = A[i] then
        return i
    else
        i ← i + 1
return −1

```

To show that arrayFind is correct, we inductively define a series of statements, S_i , that lead to the correctness of our algorithm. Specifically, we claim the following is true at the beginning of iteration i of the **while** loop:

S_i : x is not equal to any of the first i elements of A .

This claim is true at the beginning of the first iteration of the loop, since there are no elements among the first 0 in A (this kind of a trivially true claim is said to hold **vacuously**). In iteration i , we compare element x to element $A[i]$ and return the index i if these two elements are equal, which is clearly correct and completes the algorithm in this case. If the two elements x and $A[i]$ are not equal, then we have found one more element not equal to x and we increment the index i . Thus, the claim S_i is true for this new value of i ; hence, it is true at the beginning of the next iteration. If the while-loop terminates without ever returning an index in A , then we have $i = n$. That is, S_n is true—there are no elements of A equal to x . Therefore, the algorithm correctly returns -1 to indicate that x is not in A .

CHAPTER 5: Stacks, Queues, and Deques

5.1 Stacks

A **stack** is a container of objects that are inserted and removed according to the **last-in first-out (LIFO) principle**. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time. The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate. Perhaps an even more amusing metaphor would be a PEZ candy dispenser, which stores mint candies in a spring-loaded container that "pops" out the top-most candy in the stack when the top of the dispenser is lifted. (See Figure 5.1.) Stacks are a fundamental data structure. They are used in many applications, including the following.

Example 5.1: Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "**back**" button.

Example 5.2: Text editors usually provide an "**undo**" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

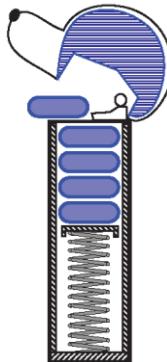


Figure 5.1 A schematic drawing of a PEZ dispenser; a physical implementation of the stack ADT.

● The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important, since they are used in a host of different applications that include many more sophisticated data structures. Formally, a stack is an abstract data type (ADT) that supports the following operations:

- `push(e)`: Insert element *e* at the top of the stack.
- `pop()`: Remove the top element from the stack; an error occurs if the stack is empty.
- `top()`: Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

`size()`: Return the number of elements in the stack.

`empty()`: Return true if the stack is empty and false otherwise.

Example 5.3: The following table shows a series of stack operations and their effects on an initially empty stack of integers.

Operation	Output	Stack Contents
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	5	(5)
pop()	–	()
pop()	“error”	()
top()	“error”	()
empty()	true	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	4	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	3	(9,7,3)

- The STL Stack

The Standard Template Library provides an implementation of a stack. The underlying implementation is based on the STL vector class, which is presented in Sections 1.5 and 6.1. In order to declare an object of type stack, **it is necessary to first include the definition file, which is called "stack." As with the STL vector, the class stack is part of the std namespace, so it is necessary either to use "std::stack" or to provide a "using" statement.** The stack class is templated with the class of the individual elements. For example, the code fragment below declares a stack of integers.

```
#include <stack>
using std::stack;           // make stack accessible
stack<int> myStack;        // a stack of integers
```

We refer to the type of individual elements as the stack's **base type**. As with STL vectors, **an STL stack dynamically resizes itself as new elements are pushed on.**

The STL stack class supports the same operators as our interface. Below, we list the principal member functions. Let *s* be declared to be an STL vector, and let *e* denote a single object whose type is the same as the base type of the stack. (For example, *s* is a vector of integers, and *e* is an integer.)

- size()*: Return the number of elements in the stack.
- empty()*: Return true if the stack is empty and false otherwise.
- push(*e*)*: Push *e* onto the top of the stack.
- pop()*: Pop the element at the top of the stack.
- top()*: Return a reference to the element at the top of the stack.

There is one significant difference between the STL implementation and our own definitions of the stack operations. **In the STL implementation, the result of applying either of the operations top or pop to an empty stack is undefined.** In particular, no exception is thrown. Even though no exception is thrown, it may very likely result in your program aborting. Thus, it is up to the programmer to be sure that no such illegal accesses are attempted.

- A C++ Stack Interface

Before discussing specific implementations of the stack, let us first consider how to define an abstract data type for a stack. When defining an abstract data type, our principal concern is specifying the **Application Programming Interface (API), or simply interface**, which **describes the names of the public members that the ADT must support and how they are to be declared and used**. An interface is not a complete description of all the public members. For example, it does not include the private data members. **Rather, it is a list of members that any implementation must provide**. The C++ programming language does not provide a simple method for defining interfaces, and therefore, the interface defined here is not an official C++ class. It is offered principally for the purpose of illustration.

The informal interface for the stack ADT is given below. This interface defines a class template. Recall from Section 2.3 that such a definition implies that the base type of element being stored in the stack will be provided by the user. In below code fragment, this element type is indicated by *E*. For example, *E* may be any fundamental type (such as **int**, **char**, **bool**, and **double**), any built-in or user-defined class (such as **string**), or a pointer to any of these.

```
template <typename E>
class Stack { // an interface for a stack
public:
    int size() const; // number of items in stack
    bool empty() const; // is the stack empty?
    const E& top() const throw(StackEmpty); // the top element
    void push(const E& e); // push x onto the stack
    void pop() throw(StackEmpty); // remove the top element
};
```

Observe that the member functions `size`, `empty`, and `top` are all declared to be **const**, which informs the compiler that they do not alter the contents of the stack. The member function `top` returns a constant reference to the top of the stack, which means that its value may be read but not written.

Note that `pop` does not return the element that was popped. If the user wants to know this value, it is necessary to perform a `top` operation first, and save the value. The member function `push` takes a constant reference to an object of type *E* as its argument. **Recall from Section 1.4 that this is the most efficient way of passing objects to a function**.

An error condition occurs when calling either of the functions `pop` or `top` on an empty stack. This is signaled by throwing an exception of type `StackEmpty`, which is defined below.

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
    StackEmpty(const string& err) : RuntimeException(err) {}
};
```

- A Simple Array-Based Stack Implementation

We can implement a stack by storing its elements in an array. Specifically, the stack in this implementation consists of an *N*-element array *S* plus an integer variable *t* that gives the index of the top element in array *S*. (See Figure 5.2.)

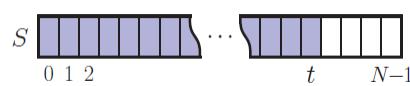


Figure 5.2 Realization of a stack by means of an array *S*. The top element in the stack is stored in the cell *S[t]*.

Recalling that arrays in C++ start at index 0, we initialize t to -1 , and use this value for t to identify when the stack is empty. Likewise, we can use this variable to determine the number of elements in a stack ($t + 1$). We also introduce a new type of exception, called `StackFull`, to signal the error condition that arises if we try to insert a new element and the array S is full. Exception `StackFull` is specific to our implementation of a stack and is not defined in the stack ADT. Given this new exception, we can then implement the stack ADT functions as described below.

```

Algorithm size():
    return  $t + 1$ 
Algorithm empty():
    return ( $t < 0$ )
Algorithm top():
    if empty() then
        throw StackEmpty exception
    return  $S[t]$ 
Algorithm push( $e$ ):
    if size() =  $N$  then
        throw StackFull exception
     $t \leftarrow t + 1$ 
     $S[t] \leftarrow e$ 
Algorithm pop():
    if empty() then
        throw StackEmpty exception
     $t \leftarrow t - 1$ 
```

The correctness of the functions in the array-based implementation follows immediately from the definition of the functions themselves. Table 5.1 shows the running times for member functions in a realization of a stack by an array. Each of the stack functions in the array realization executes a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, in this implementation of the Stack ADT, each function runs in constant time, that is, they each run in $O(1)$ time.

Operation	Time
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Table 5.1 Performance of an array-based stack. The space usage is $O(N)$, where N is the array's size. Note that the space usage is independent from the number $n \leq N$ of elements that are actually in the stack.

A C++ Implementation of a Stack

In this section, we present a concrete C++ implementation of the above pseudocode specification by means of a class, called `ArrayStack`. Our approach is to store the elements of a stack in an array. To keep the code simple, we have omitted the standard housekeeping utilities, such as a destructor, an assignment operator, and a copy constructor. We leave their implementations as an exercise.

We begin by providing the `ArrayStack` class definition below.

```

template <typename E>
class ArrayStack {
    enum { DEF_CAPACITY = 100 };           // default stack capacity
public:
    ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
    int size() const;                    // number of items in the stack
    bool empty() const;                 // is the stack empty?
    const E& top() const throw(StackEmpty); // get the top element
    void push(const E& e) throw(StackFull); // push element onto stack
    void pop() throw(StackEmpty);        // pop the stack
    // ...housekeeping functions omitted
private:
    E* S;                                // member data
    int capacity;                         // array of stack elements
    int t;                                // stack capacity
    // index of the top of the stack
};

```

In addition to the member functions required by the interface, we also provide a constructor, that is given the desired capacity of the stack as its only argument. If no argument is given, the default value given by `DEF_CAPACITY` is used. This is an example of using default arguments in function calls. We use an enumeration to define this default capacity value. This is the simplest way of defining symbolic integer constants within a C++ class. Our class is templated with the element type, denoted by `E`. The stack's storage, denoted `S`, is a dynamically allocated array of type `E`, that is, a pointer to `E`.

Next, we present the implementations of the `ArrayStack` member functions below. The constructor allocates the array storage, whose size is set to the default capacity. The members `capacity` and `t` are also set to their initial values. In spite of the syntactical complexities of defining templated member functions in C++, the remaining member functions are straightforward implementations of their definitions above. Observe that functions `top` and `pop` first check that the stack is not empty, and otherwise, they throw an exception. Similarly, `push` first checks that the stack is not full, and otherwise, it throws an exception.

```

template <typename E> ArrayStack<E>::ArrayStack(int cap)
: S(new E[cap]), capacity(cap), t(-1) {} // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
{ return (t + 1); }                      // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
{ return (t < 0); }                      // is the stack empty?

template <typename E>                   // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
    if (empty()) throw StackEmpty("Top of empty stack");
    return S[t];
}

template <typename E>                   // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
    if (size() == capacity) throw StackFull("Push to full stack");
    S[++t] = e;
}

template <typename E>                   // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
    if (empty()) throw StackEmpty("Pop from empty stack");
    --t;
}

```

Example Output

Below, we present an example of the use of our `ArrayStack` class. To demonstrate the flexibility

of our implementation, we show two stacks of different base types. The instance *A* is a stack of integers of the default capacity (100). The instance *B* is a stack of character strings of capacity 10.

```
ArrayStack<int> A;
A.push(7);
A.push(13);
cout << A.top() << endl; A.pop();
A.push(9);
cout << A.top() << endl;
cout << A.top() << endl; A.pop();
ArrayStack<string> B(10);
B.push("Bob");
B.push("Alice");
cout << B.top() << endl; B.pop();
B.push("Eve");
```

// A = [], size = 0
// A = [7*], size = 1
// A = [7, 13*], size = 2
// A = [7*], outputs: 13
// A = [7, 9*], size = 2
// A = [7, 9*], outputs: 9
// A = [7*], outputs: 9
// B = [], size = 0
// B = [Bob*], size = 1
// B = [Bob, Alice*], size = 2
// B = [Bob*], outputs: Alice
// B = [Bob, Eve*], size = 2

(The contents of the stack are shown in the comment following the operation. The top of the stack is indicated by an asterisk ("*").)

Note that our implementation, while simple and efficient, could be enhanced in a number of ways. For example, it assumes a fixed upper bound *N* on the ultimate size of the stack. In previous code fragment, we chose the default capacity value *N* = 100 more or less arbitrarily (although the user can set the capacity in the constructor). An application may actually need much less space than the given initial size, and this would be wasteful of memory. Alternatively, an application may need more space than this, in which case our stack implementation might "crash" if too many elements are pushed onto the stack.

Fortunately, there are other implementations that do not impose an arbitrary size limitation. One such method is to use the STL stack class, which was introduced earlier in this chapter. **The STL stack is also based on the STL vector class, and it offers the advantage that it is automatically expanded when the stack overflows its current storage limits.** In practice, the STL stack would be the easiest and most practical way to implement an array-based stack. Later in this chapter, we see other methods that use space proportional to the actual size of the stack.

In instances **where we have a good estimate on the number of items needing to go in the stack**, the array-based implementation is hard to beat from the perspective of speed and simplicity. Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack ADT implementation, such as the simple array-based implementation.

● Implementing a Stack with a Generic Linked List

In this section, we show how to implement the stack ADT using a singly linked list. Our approach is to use the generic singly linked list, called *SLinkedList*, which was presented earlier in Section 3.2. The definition of our stack, called *LinkedStack*, is presented below.

To avoid **the syntactic messiness inherent in C++ templated classes**, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the stack's elements, called *Elem*. In this example, we define *Elem* to be of type *string*. We leave the task of producing a truly generic implementation as an exercise.

```

typedef string Elem;           // stack element type
class LinkedStack {           // stack as a linked list
public:
    LinkedStack();             // constructor
    int size() const;        // number of items in the stack
    bool empty() const;      // is the stack empty?
    const Elem& top() const throw(StackEmpty); // the top element
    void push(const Elem& e); // push element onto stack
    void pop() throw(StackEmpty); // pop the stack
private:
    SLinkedList<Elem> S;     // linked list of elements
    int n;                  // number of elements
};

```

The principal data member of the class is the generic linked list of type *Elem*, called *S*. Since the *SLinkedList* class does not provide a member function *size*, we store the current size in a member variable, *n*.

In below code fragment, we present the implementations of the constructor and the *size* and *empty* functions. Our constructor creates the initial stack and initializes *n* to zero. We do not provide an explicit destructor, relying instead on the *SLinkedList* destructor to deallocate the linked list *S*.

```

LinkedStack::LinkedStack()
: S(), n(0) { }                                // constructor

int LinkedStack::size() const
{ return n; }                                    // number of items in the stack

bool LinkedStack::empty() const
{ return n == 0; }                                // is the stack empty?

```

The definitions of the stack operations, *top*, *push*, and *pop*, are presented below. Which side of the list, head or tail, should we chose for the top of the stack? Since *SLinkedList* can insert and delete elements in constant time only at the head, **the head is clearly the better choice**. Therefore, the member function *top* returns *S.front()*. The functions *push* and *pop* invoke the functions *addFront* and *removeFront*, respectively, and update the number of elements.

```

// get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
    if (empty()) throw StackEmpty("Top of empty stack");
    return S.front();
}
void LinkedStack::push(const Elem& e) { // push element onto stack
    ++n;
    S.addFront(e);
}
// pop the stack
void LinkedStack::pop() throw(StackEmpty) {
    if (empty()) throw StackEmpty("Pop from empty stack");
    --n;
    S.removeFront();
}

```

● Reversing a Vector Using a Stack

The **basic** idea is to push all the elements of the vector in order into a stack and then fill the vector back up again by popping the elements off of the stack.

```

template <typename E>
void reverse(vector<E>& V) {           // reverse a vector
    ArrayStack<E> S(V.size());
    for (int i = 0; i < V.size(); i++)      // push elements onto stack
        S.push(V[i]);
    for (int i = 0; i < V.size(); i++) {     // pop them in reverse order
        V[i] = S.top(); S.pop();
    }
}

```

● Matching Parentheses and HTML Tags

In this section, we explore two related applications of stacks. The **first** is **matching** parentheses and **grouping** symbols in arithmetic expressions. Arithmetic expressions can contain various pairs of grouping symbols, such as

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"
- Floor function symbols: "[" and "]"
- Ceiling function symbols: "[" and "]"

and each opening symbol **must** match with its corresponding closing symbol. For example, a left bracket symbol ("[") must match with a corresponding right bracket ("]") as in the following expression:

- Correct: ()(()){([()])}
- Correct: ((()(()){([()])}))
- Incorrect:)(()){([()])}
- Incorrect: ({[]})
- Incorrect: (

An Algorithm for Parentheses Matching

An important problem in processing arithmetic expressions is to make sure their grouping symbols match up correctly. We can use a **stack** S to perform the matching of grouping symbols in an arithmetic expression with a single **left-to-right** scan. The algorithm tests that left and right symbols match up and also that the left and right symbols are both of the same type. Suppose we are given a sequence $X = x_0x_1x_2 \dots x_{n-1}$, where each x_i is a **token** that can be a grouping symbol, a variable name, an arithmetic operator, or a number. The **basic** idea behind checking that the grouping symbols in S match correctly, is to process the tokens in X in order. Each time we encounter an **opening** symbol, we push that symbol onto S , and each time we encounter a **closing** symbol, we pop the top symbol from the stack S (assuming S is not empty) and we check that these two symbols are of corresponding types. (For example, if the symbol "(" was pushed, the symbol ")" should be its match.) If the stack is **empty** after we have processed the **whole** sequence, then the symbols in X match.

Algorithm ParenMatch(X, n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: true if and only if all the grouping symbols in X match

Let S be an empty stack

for $i \leftarrow 0$ to $n - 1$ **do**

- if** $X[i]$ is an opening grouping symbol **then**
- $S.push(X[i])$
- else if** $X[i]$ is a closing grouping symbol **then**
- if** $S.empty()$ **then**
- return** false {nothing to match with}
- if** $S.top()$ does not match the type of $X[i]$ **then**
- return** false {wrong type}
- $S.pop()$
- if** $S.empty()$ **then**
- return** true {every symbol matched}
- else**
- return** false {some symbols were never matched}

Matching Tags in an HTML Document

(Page 205)

5.2 Queues

Another fundamental data structure is the **queue**, which is a close relative of the stack. A queue is a container of elements that are inserted and removed according to the **first-in first-out (FIFO)** principle. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the **rear** and are removed from the **front**.

● The Queue Abstract Data Type

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are **restricted** to the first element in the sequence, which is called the **front** of the queue, and element insertion is restricted to the end of the sequence, which is called the **rear** of the queue.

enqueue(e): Insert element e at the rear of the queue.**dequeue()**: Remove element at the front of the queue; an error occurs if the queue is empty.**front()**: Return, but do not remove, a reference to the front element in the queue; an error occurs if the queue is empty.**size()**: Return the number of elements in the queue.**empty()**: Return true if the queue is empty and false otherwise.

Operation	Output	$front \leftarrow Q \leftarrow rear$
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
front()	5	(5, 3)
size()	2	(5, 3)
dequeue()	–	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	7	(7)
dequeue()	–	()
dequeue()	“error”	()
empty()	true	()

● The STL Queue

```
#include <queue>
using std::queue;
queue<float> myQueue; // make queue accessible
// a queue of floats
```

size(): Return the number of elements in the queue.
empty(): Return true if the queue is empty and false otherwise.
push(*e*): Enqueue *e* at the rear of the queue.
pop(): Dequeue the element at the front of the queue.
front(): Return a reference to the element at the queue's front.
back(): Return a reference to the element at the queue's rear.

- A C++ Queue Interface

```

template <typename E>
class Queue {                                     // an interface for a queue
public:
    int size() const;                           // number of items in queue
    bool empty() const;                         // is the queue empty?
    const E& front() const throw(QueueEmpty);   // the front element
    void enqueue (const E& e);                  // enqueue element at rear
    void dequeue() throw(QueueEmpty);           // dequeue element at front
};
  
```

- A Simple Array-Based Implementation

The **main issue** with this implementation is deciding how to keep track of the front and rear of the queue.

One possibility is to adapt the approach we used for the stack implementation. In particular, let $Q[0]$ be the front of the queue and have the queue grow from there. This is **not** an efficient solution, however, for it requires that we move all the elements forward one array cell each time we perform a dequeue operation.

Using an Array in a Circular Way

To **avoid** moving objects once they are placed in Q , we define three variables, f , r , n , which have the following meanings:

- f is the index of the cell of Q storing the front of the queue. If the queue is nonempty, this is the index of the element to be removed by dequeue.
- r is an index of the cell of Q following the rear of the queue. If the queue is not full, this is the index where the element is inserted by enqueue.
- n is the current number of elements in the queue.

Initially, we set $n = 0$ and $f = r = 0$, indicating an empty queue. When we dequeue an element from the front of the queue, we decrement n and increment f to the next cell in Q . Likewise, when we enqueue an element, we increment r and increment n . This allows us to implement the enqueue and dequeue functions in constant time.

Nonetheless, there is still a problem with this approach: we would get an array-out-of-bounds error. To avoid this problem and be able to utilize all of the array Q , we let the f and r indices “wrap around” the end of Q . That is, we now view Q as a “circular array” that goes from $Q[0]$ to $Q[N - 1]$ and then immediately back to $Q[0]$ again.



Figure 5. 3 Using array Q in a circular fashion: (a) the “normal” configuration with $f \leq r$; (b) the “wrapped around” configuration with $r < f$. The cells storing queue elements are shaded.

Using the Modulo Operator to Implement a Circular Array

Implementing this circular view of Q is actually pretty easy. Each time we increment f or r , we simply need to compute this increment as “ $(f + 1) \bmod N$ ” or “ $(r + 1) \bmod N$,” respectively, where the operator “ \bmod ” is the **modulo** operator.

```

Algorithm size():
    return n
Algorithm empty():
    return (n == 0)
Algorithm front():
    if empty() then
        throw QueueEmpty exception
    return Q[f]
Algorithm dequeue():
    if empty() then
        throw QueueEmpty exception
    f  $\leftarrow$  (f + 1) mod N
    n = n - 1
Algorithm enqueue(e):
    if size() = N then
        throw QueueFull exception
    Q[r]  $\leftarrow$  e
    r  $\leftarrow$  (r + 1) mod N
    n = n + 1

```

- Implementing a Queue with a Circularly Linked List

We **cannot** use our singly linked list class, since it provides efficient access only to one side of the list.

Recall that CircleList maintains a pointer, called the **cursor**, which points to one node of the list. In order to implement a queue, the element referenced by back will be the rear of the queue and the element referenced by front will be the front.

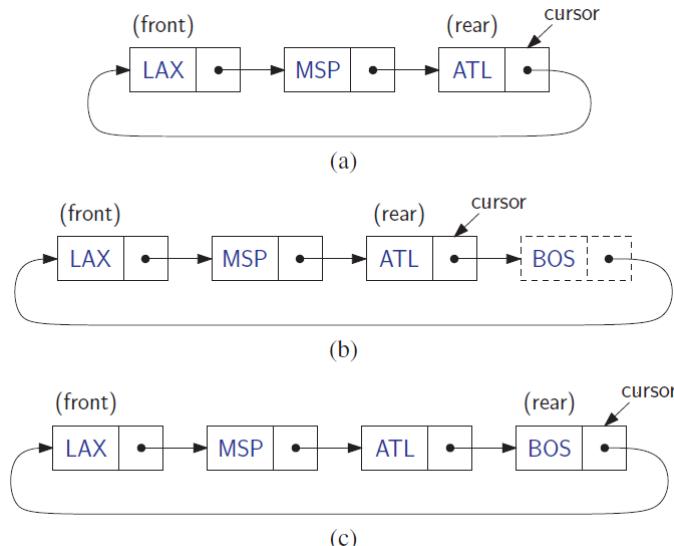


Figure 5.4 Enqueueing “BOS” into a queue represented as a circularly linked list: (a) before the operation; (b) after adding the new node; (c) after advancing the cursor.

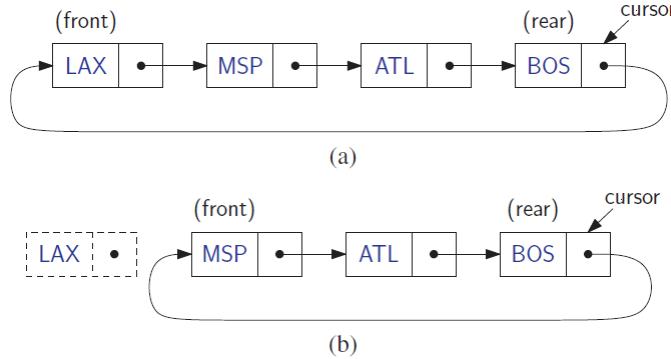


Figure 5.5 Dequeueing an element (in this case "LAX") from the front queue represented as a circularly linked list: (a) before the operation; (b) after removing the node immediately following the cursor.

```

typedef string Elem;                                // queue element type
class LinkedQueue {                                // queue as doubly linked list
public:
    LinkedQueue();                                  // constructor
    int size() const;                            // number of items in the queue
    bool empty() const;                          // is the queue empty?
    const Elem& front() const throw(QueueEmpty); // the front element
    void enqueue(const Elem& e);                // enqueue element at rear
    void dequeue() throw(QueueEmpty);            // dequeue element at front
private:
    CircleList C;                                 // circular list of elements
    int n;                                      // number of elements
};
LinkedQueue::LinkedQueue()                         // constructor
: C(), n(0) { }

int LinkedQueue::size() const                  // number of items in the queue
{ return n; }

bool LinkedQueue::empty() const                // is the queue empty?
{ return n == 0; }

const Elem& LinkedQueue::front() const throw(QueueEmpty) {
    if (empty())
        throw QueueEmpty("front of empty queue");
    return C.front();                           // list front is queue front
}

void LinkedQueue::enqueue(const Elem& e) {      // enqueue element at rear
    C.add(e);                                    // insert after cursor
    C.advance();                                 // ...and advance
    n++;
}
void LinkedQueue::dequeue() throw(QueueEmpty) { // dequeue element at front
    if (empty())
        throw QueueEmpty("dequeue of empty queue");
    C.remove();                                 // remove from list front
    n--;
}

```

5.3 Double-Ended Queues

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Such an extension of a queue is called a **double-ended queue**, or **deque**.

- The Deque Abstract Data Type

The functions of the deque ADT are as follows, where D denotes the deque:

- `insertFront(e)`: Insert a new element *e* at the beginning of the deque.
- `insertBack(e)`: Insert a new element *e* at the end of the deque.
- `eraseFront()`: Remove the first element of the deque; an error occurs if the deque is empty.
- `eraseBack()`: Remove the last element of the deque; an error occurs if the deque is empty.
- `front()`: Return the first element of the deque; an error occurs if the deque is empty.
- `back()`: Return the last element of the deque; an error occurs if the deque is empty.
- `size()`: Return the number of elements of the deque.
- `empty()`: Return true if the deque is empty and false otherwise.

<i>Operation</i>	<i>Output</i>	<i>D</i>
<code>insertFront(3)</code>	–	(3)
<code>insertFront(5)</code>	–	(5, 3)
<code>front()</code>	5	(5, 3)
<code>eraseFront()</code>	–	(3)
<code>insertBack(7)</code>	–	(3, 7)
<code>back()</code>	7	(3, 7)
<code>eraseFront()</code>	–	(7)
<code>eraseBack()</code>	–	()

Figure 5. 6 The following example shows a series of operations and their effects on an initially empty deque, *D*, of integers.

● The STL Deque

```
#include <deque>
using std::deque; // make deque accessible
deque<string> myDeque; // a deque of strings
    size(): Return the number of elements in the deque.
    empty(): Return true if the deque is empty and false otherwise.
    push_front(e): Insert e at the beginning the deque.
    push_back(e): Insert e at the end of the deque.
    pop_front(): Remove the first element of the deque.
    pop_back(): Remove the last element of the deque.
    front(): Return a reference to the deque's first element.
    back(): Return a reference to the deque's last element.
```

● Implementing a Deque with a Doubly Linked List

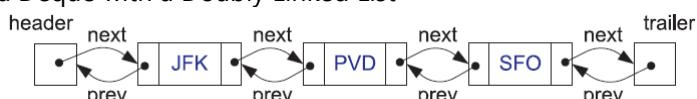


Figure 5. 7 A doubly linked list with sentinels, header and trailer. The front of our deque is stored just after the header ("JFK"), and the back of our deque is stored just before the trailer ("SFO").

```

typedef string Elem; // deque element type
class LinkedDeque { // deque as doubly linked list
public:
    LinkedDeque(); // constructor
    int size() const; // number of items in the deque
    bool empty() const; // is the deque empty?
    const Elem& front() const throw(DequeEmpty); // the first element
    const Elem& back() const throw(DequeEmpty); // the last element
    void insertFront(const Elem& e); // insert new first element
    void insertBack(const Elem& e); // insert new last element
    void removeFront() throw(DequeEmpty); // remove first element
    void removeBack() throw(DequeEmpty); // remove last element
private: // member data
    DLinkedList D; // linked list of elements
    int n; // number of elements
};

// insert new first element
void LinkedDeque::insertFront(const Elem& e) {
    D.addFront(e);
    n++;
}

// insert new last element
void LinkedDeque::insertBack(const Elem& e) {
    D.addBack(e);
    n++;
}

// remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
    if (empty())
        throw DequeEmpty("removeFront of empty deque");
    D.removeFront();
    n--;
}

// remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
    if (empty())
        throw DequeEmpty("removeBack of empty deque");
    D.removeBack();
    n--;
}

```

<i>Operation</i>	<i>Time</i>
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

Figure 5. 8 Performance of a deque realized by a doubly linked list. The space usage is $O(n)$, where n is number of elements in the deque.

- Adapters and the Adapter Design Pattern

CHAPTER 6: List and Iterator ADTs

6.1 Vectors

Suppose we have a collection S of n elements stored in a certain linear order, so that we can refer to the elements in S as first, second, third, and so on. Such a collection is generically referred to as a **list** or **sequence**. We can uniquely refer to each element e in S using an **integer** in the range $[0, n - 1]$ that is equal to the number of elements of S that precede e in S . The **index** of an element e in S is the number of elements that are **before** e in S . Hence, the first element in S has index 0 and the last element has index $n - 1$. Also, if an element of S has index i , its previous element (if it exists) has index $i - 1$, and its next element (if it exists) has index $i + 1$. This concept of index is **related** to that of the **rank** of an element in a list, which is usually defined to be **one more** than its index; so the first element is at rank 1, the second is at rank 2, and so on.

A **sequence** that supports access to its elements by their indices is called a **vector**.

- The Vector Abstract Data Type

A **vector**, also called an **array list**, is an ADT that supports the following fundamental functions.

In all cases, the index parameter i is assumed to be in the **range** $0 \leq i \leq \text{size}() - 1$.

at(i): Return the element of V with index i ; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e ; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i ; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i ; an error condition occurs if i is out of range.

We do **not** insist that an array be used to implement a vector.

The index of an element may **change** when the sequence is updated.

Operation	Output	V
insert(0, 7)	—	(7)
insert(0, 4)	—	(4, 7)
at(1)	7	(4, 7)
insert(2, 2)	—	(4, 7, 2)
at(3)	“error”	(4, 7, 2)
erase(1)	—	(4, 2)
insert(1, 5)	—	(4, 5, 2)
insert(1, 3)	—	(4, 3, 5, 2)
insert(4, 9)	—	(4, 3, 5, 2, 9)
at(2)	5	(4, 3, 5, 2, 9)
set(3, 8)	—	(4, 3, 5, 8, 9)

Figure 5. 9 We show below some operations on an initially empty vector V .

- A Simple Array-Based Implementation

An obvious choice for implementing the vector ADT is to use a fixed size array A , where $A[i]$ stores the element at index i . We choose the size N of array A to be sufficiently large, and we maintain the number $n < N$ of elements in the vector in a member variable.

To implement the **at(i)** operation, for example, we just return $A[i]$,

Algorithm **insert(i, e)**:

```

for  $j = n - 1, n - 2, \dots, i$  do
     $A[j + 1] \leftarrow A[j]$            {make room for the new element}
     $A[i] \leftarrow e$ 
     $n \leftarrow n + 1$ 

```

Algorithm **erase(i)**:

```

for  $j = i + 1, i + 2, \dots, n - 1$  do
     $A[j - 1] \leftarrow A[j]$            {fill in for the removed element}
     $n \leftarrow n - 1$ 

```

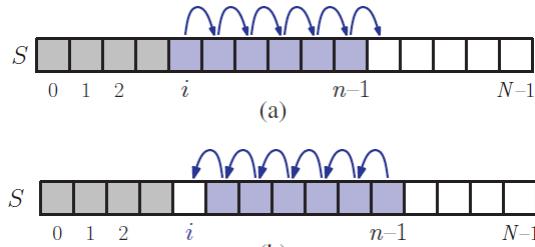


Figure 5.10 Array-based implementation of a vector V that is storing n elements: (a) shifting up for an insertion at index i ; (b) shifting down for a removal at index i .

The Performance of a Simple Array-Based Implementation

Operation	Time
<code>size()</code>	$O(1)$
<code>empty()</code>	$O(1)$
<code>at(i)</code>	$O(1)$
<code>set(i, e)</code>	$O(1)$
<code>insert(i, e)</code>	$O(n)$
<code>erase(i)</code>	$O(n)$

Figure 5.11 Performance of a vector with n elements realized by an array. The space usage is $O(N)$, where N is the size of the array.

- An Extendable Array Implementation

A major **weakness** of the simple array implementation for the vector ADT given in Section 6.1.2 is that it requires advance specification of a fixed capacity, N , for the total number of elements that may be stored in the vector.

Let us provide a **means** to grow the array A that stores the elements of a vector V . When an **overflow** occurs, that is, when $n = N$ and function `insert` is called, we perform the following steps:

1. Allocate a new array B of capacity N
2. Copy $A[i]$ to $B[i]$, for $i = 0, \dots, N - 1$
3. Deallocate A and reassign A to point to the new array B

This array replacement strategy is known as an **extendable array**, for it can be viewed as extending the end of the underlying array to make room for more elements.

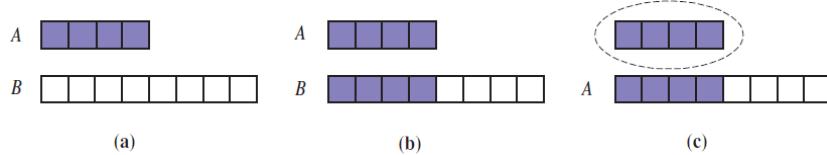


Figure 6.1 The three steps for “growing” an extendable array: (a) create new array B ; (b) copy elements from A to B ; (c) reassign A to refer to the new array and delete the old array.

```

typedef int Elem; // base element type
class ArrayVector {
public:
    ArrayVector(); // constructor
    int size() const; // number of elements
    bool empty() const; // is vector empty?
    Elem& operator[](int i); // element at index
    Elem& at(int i) throw(IndexOutOfBoundsException); // element at index
    void erase(int i); // remove element at index
    void insert(int i, const Elem& e); // insert element at index
    void reserve(int N); // reserve at least N spots
    // ... (housekeeping functions omitted)
private:
    int capacity; // current array size
    int n; // number of elements in vector
    Elem* A; // array storing the elements
};

ArrayVector::ArrayVector() // constructor
: capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const // number of elements
{ return n; }

bool ArrayVector::empty() const // is vector empty?
{ return size() == 0; }

Elem& ArrayVector::operator[](int i) // element at index
{ return A[i]; }

Elem& ArrayVector::at(int i) throw(IndexOutOfBoundsException) {
    if (i < 0 || i >= n)
        throw IndexOutOfBoundsException("illegal index in function at()");
    return A[i];
}

void ArrayVector::erase(int i) { // remove element at index
    for (int j = i+1; j < n; j++) // shift elements down
        A[j - 1] = A[j];
    n--; // one fewer element
}

void ArrayVector::reserve(int N) { // reserve at least N spots
    if (capacity >= N) return; // already big enough
    Elem* B = new Elem[N]; // allocate bigger array
    for (int j = 0; j < n; j++) // copy contents to new array
        B[j] = A[j];
    if (A != NULL) delete [] A; // discard old array
    A = B; // make B the new array
    capacity = N; // set new capacity
}

void ArrayVector::insert(int i, const Elem& e) {
    if (n >= capacity) // overflow?
        reserve(max(1, 2 * capacity)); // double array size
    for (int j = n - 1; j >= i; j--) // shift elements up
        A[j+1] = A[j];
    A[i] = e; // put in empty slot
    n++; // one more element
}

```

● STL Vectors

A **container** is a data structure that stores a collection of objects. Many of the data structures that we study later in this book, such as stacks, queues, and lists, are examples of STL containers. The class `vector` is perhaps the most basic example of an STL container class.

```

#include <vector> // provides definition of vector
using std::vector; // make vector accessible

vector<int> myVector(100); // a vector with 100 integers

```

We refer to the type of individual elements as the vector's **base type**. Each element is initialized

to the base type's default value, which for integers is zero.

STL vector objects behave in many respects like standard C++ arrays, but they provide **many** additional features.

- As with arrays, individual elements of a vector object can be indexed using the usual index operator ("[]"). Elements can also be accessed by a member function called `at`. The advantage of this member function over the index operator is that it performs range checking and generates an error exception if the index is out of bounds.
- Unlike C++ arrays, STL vectors can be dynamically resized, and new elements may be efficiently appended or removed from the end of an array.
- When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements. (With C++ arrays, it is the obligation of the programmer to do this explicitly.)
- STL vectors provide a number of useful functions that operate on entire vectors, not just on individual elements. This includes, for example, the ability to copy all or part of one vector to another, the ability to compare the contents of two arrays, and the ability to insert and erase multiple elements.

Let V be declared to be an STL vector of some base type, and let e denote a single object of this same base type.

`vector(n)`: Construct a vector with space for n elements; if no argument is given, create an empty vector.

`size()`: Return the number of elements in V .

`empty()`: Return true if V is empty and false otherwise.

`resize(n)`: Resize V , so that it has space for n elements.

`reserve(n)`: Request that the allocated storage space be large enough to hold n elements.

`operator[i]`: Return a reference to the i th element of V .

`at(i)`: Same as $V[i]$, but throw an `out_of_range` exception if i is out of bounds, that is, if $i < 0$ or $i \geq V.size()$.

`front()`: Return a reference to the first element of V .

`back()`: Return a reference to the last element of V .

`push_back(e)`: Append a copy of the element e to the end of V , thus increasing its size by one.

`pop_back()`: Remove the last element of V , thus reducing its size by one.

6.2 Lists

Using an index is **not** the only means of referring to the place where an element appears in a list. If we have a list L implemented with a (singly or doubly) linked list, then it could possibly be more natural and efficient to use a **node** instead of an index as a means of identifying where to access and update a list.

● Node-Based Operations and Iterators

Let L be a (singly or doubly) linked list. We would like to **define** functions for L that take nodes of the list as parameters and provide nodes as return types.

To abstract and unify the different ways of storing elements in the various implementations of a list, we **introduce** a data type that abstracts the notion of the relative position or place of an element within a list. Such an object might naturally be called a **position**. Because we **want** this object not only to access individual elements of a list, but also to move around in order to enumerate all the elements of a list, we adopt the convention used in the C++ Standard Template Library, and call it an **iterator**.

Containers and Positions

In order to **safely** expand the set of operations for lists, we abstract a notion of "position" that allows us to enjoy the efficiency of doubly or singly linked list implementations without violating

object-oriented design principles. In this framework, we think of a list as an instance of a more **general** class of objects, called a container. A **container** is a data structure that stores any collection of elements. We assume that the elements of a container can be arranged in a **linear** order. A **position** is defined to be an abstract data type that is associated with a particular container and which supports the following function.

`element():` Return a reference to the element stored at this position.

A position is always defined in a relative manner, that is, in terms of its neighbors. A position q , which is associated with some element e in a container, **does not change**, even if the index of e changes in the container, unless we explicitly remove e . If the associated node is removed, we say that q is **invalidated**.

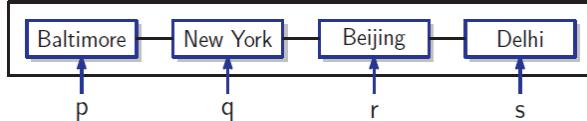


Figure 6. 2 A list container. The positions in the current order are p , q , r , and s .

Iterators

Although a position is a useful object, it would be more **useful** still to be able to navigate through the container, for example, by advancing to the next position in the container. Such an object is called an **iterator**. An iterator is an extension of a position. It supports the ability to access a node's element, but it also provides the ability to navigate forwards (and possibly backwards) through the container.

In addition to navigating through the container, we need some way of initializing an iterator to the first node of a container and determining whether it has gone beyond the end of the container. To do this, we assume that each container provides two special iterator values, **begin** and **end**. The beginning iterator refers to the first position of the container. We think of the ending iterator as referring to an imaginary position that lies **just after** the last node of the container.



Figure 6. 3 The special iterators $L.begin()$ and $L.end()$ for a list L .

In order to **enumerate** all the elements of a given container L , we define an iterator p whose value is initialized to $L.begin()$. The associated element is accessed using $*p$. We can enumerate all of the elements of the container by advancing p to the next node using the operation $++p$. We repeat this until p is equal to $L.end()$, which means that we have fallen off the end of the list.

● The List Abstract Data Type

Using the concept of an iterator to encapsulate the idea of "**node**" in a list, we can define another type of sequence ADT, called simply the **list** ADT.

`begin():` Return an iterator referring to the first element of L ; same as `end()` if L is empty.

`end():` Return an iterator referring to an imaginary element just after the last element of L .

`insertFront(e):` Insert a new element e into L as the first element.

`insertBack(e):` Insert a new element e into L as the last element.

`insert(p, e):` Insert a new element e into L before position p in L .

`eraseFront():` Remove the first element of L .

`eraseBack():` Remove the last element of L .

`erase(p):` Remove from L the element at position p ; invalidates p as a position.

An **error** condition occurs if an invalid position is passed as an argument to one of the list operations. Reasons for a position p to be invalid include:

- p was never initialized or was set to a position in a different list
- p was previously removed from the list
- p results from an illegal operation, such as attempting to perform $++p$, where $p = L.end()$, that is, attempting to access a position beyond the end position

<i>Operation</i>	<i>Output</i>	<i>L</i>
insertFront(8)	–	(8)
$p = begin()$	$p : (8)$	(8)
insertBack(5)	–	(8, 5)
$q = p; ++q$	$q : (5)$	(8, 5)
$p == begin()$	true	(8, 5)
insert($q, 3$)	–	(8, 3, 5)
$*q = 7$	–	(8, 3, 7)
insertFront(9)	–	(9, 8, 3, 7)
eraseBack()	–	(9, 8, 3)
erase(p)	–	(9, 3)
eraseFront()	–	(3)

Figure 6.4 We show a series of operations for an initially empty list L below. We use variables p and q to denote different positions, and we show the object currently stored at such a position in parentheses in the Output column.

● Doubly Linked List Implementation

Probably the most natural and efficient way is to use a doubly linked list. Recall that our doubly linked list structure is based on two **sentinel nodes**, called the **header** and **trailer**.

Before defining the class, which we call `NodeList`, we define **two important** structures. The first represents a **node** of the list and the other represents an **iterator** for the list. Both of these objects are defined as **nested classes** within `NodeList`. Since users of the class access nodes exclusively through iterators, the node is declared a private member of `NodeList`, and the iterator is a public member.

```
struct Node {                                // a node of the list
    Elem elem;                             // element value
    Node* prev;                            // previous in list
    Node* next;                            // next in list
};
```

Our **iterator** object is called `Iterator`. To users of class `NodeList`, it can be accessed by the qualified type name `NodeList::Iterator`. We declare `NodeList` to be a friend, so that it may access the private members of `Iterator`. We **also** provide a private constructor, which initializes the node pointer. (The constructor is private **so that only** `NodeList` is allowed to create new iterators.)

```
class Iterator {                           // an iterator for the list
public:
    Elem& operator*();                  // reference to the element
    bool operator==(const Iterator& p) const; // compare positions
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();            // move to next position
    Iterator& operator--();            // move to previous position
    friend class NodeList;             // give NodeList access
private:
    Node* v;                          // pointer to the node
    Iterator(Node* u);                // create from node
};
```

```

NodeList::Iterator::Iterator(Node* u)           // constructor from Node*
{ v = u; }

Elem& NodeList::Iterator::operator*()          // reference to the element
{ return v->elem; }

bool NodeList::Iterator::operator==(const Iterator& p) const
{ return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
{ return v != p.v; }

// move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
{ v = v->next; return *this; }

// move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
{ v = v->prev; return *this; }

typedef int Elem;                           // list base element type
class NodeList {                          // node-based list
private:
    // insert Node declaration here...
public:
    // insert Iterator declaration here...
public:
    NodeList();                         // default constructor
    int size() const;                  // list size
    bool empty() const;               // is the list empty?
    Iterator begin() const;          // beginning position
    Iterator end() const;            // (just beyond) last position
    void insertFront(const Elem& e); // insert at front
    void insertBack(const Elem& e);  // insert at rear
    void insert(const Iterator& p, const Elem& e); // insert e before p
    void eraseFront();                // remove first
    void eraseBack();                // remove last
    void erase(const Iterator& p);   // remove p
    // housekeeping functions omitted...
private:
    int n;                            // data members
    Node* header;                    // number of items
    Node* trailer;                  // head-of-list sentinel
    Node* trailer;                  // tail-of-list sentinel
};

NodeList::NodeList() {
    n = 0;
    header = new Node;              // initially empty
    trailer = new Node;             // create sentinels
    header->next = trailer;        // have them point to each other
    trailer->prev = header;
}

int NodeList::size() const               // list size
{ return n; }

bool NodeList::empty() const            // is the list empty?
{ return (n == 0); }

NodeList::Iterator NodeList::begin() const // begin position is first item
{ return Iterator(header->next); }

NodeList::Iterator NodeList::end() const  // end position is just beyond last
{ return Iterator(trailer); }

```

```

        // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
    Node* w = p.v;
    Node* u = w->prev;
    Node* v = new Node;
    v->elem = e;
    v->next = w; w->prev = v;
    v->prev = u; u->next = v;
    n++;
}

void NodeList::insertFront(const Elem& e) // insert at front
{ insert(begin(), e); }

void NodeList::insertBack(const Elem& e) // insert at rear
{ insert(end(), e); }

void NodeList::erase(const Iterator& p) { // remove p
    Node* v = p.v;
    Node* w = v->next;
    Node* u = v->prev;
    u->next = w; w->prev = u;
    delete v;
    n--;
}

void NodeList::eraseFront() // remove first
{ erase(begin()); }

void NodeList::eraseBack() // remove last
{ erase(--end()); }

```

● STL Lists

```

#include <list>
using std::list; // make list accessible
list<float> myList; // an empty list of floats
list(n): Construct a list with n elements; if no argument list is
given, an empty list is created.
size(): Return the number of elements in L.
empty(): Return true if L is empty and false otherwise.
front(): Return a reference to the first element of L.
back(): Return a reference to the last element of L.
push_front(e): Insert a copy of e at the beginning of L.
push_back(e): Insert a copy of e at the end of L.
pop_front(): Remove the fist element of L.
pop_back(): Remove the last element of L.

```

● STL Containers and Iterators

In order to develop a fuller understanding of STL vectors and lists, it is necessary to understand the concepts of STL **containers** and **iterators**.

STL Container	Description
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

STL Iterators

(Page 249)

Using Iterators

(Page 250)

STL Iterator-Based Container Functions

(Page 251)

STL Vectors and Algorithms

(Page 253)

An Illustrative Example

(Page 253)

6.3 Sequences

In this section, we define an abstract data type that generalizes the **vector** and **list** ADTs. This ADT therefore provides access to its elements using both indices and positions, and is a versatile data structure for a wide variety of applications.

● The Sequence Abstract Data Type

A **sequence** is an ADT that supports all the functions of the list ADT, but it also provides functions for accessing elements by their index, as we did in the vector ADT. The interface consists of the operations of the list ADT, plus the following two “bridging” functions, which provide connections between indices and positions.

atIndex(*i*): Return the position of the element at index *i*.

indexOf(*p*): Return the index of the element at position *p*.

● Implementing a Sequence with a Doubly Linked List

One possible implementation of a sequence, of course, is with a doubly linked list.

```
class NodeSequence : public NodeList {
public:
    Iterator atIndex(int i) const;           // get position from index
    int indexOf(const Iterator& p) const;     // get index from position
};

// get position from index
NodeSequence::Iterator NodeSequence::atIndex(int i) const {
    Iterator p = begin();
    for (int j = 0; j < i; j++) ++p;
    return p;
}

// get index from position
int NodeSequence::indexOf(const Iterator& p) const {
    Iterator q = begin();
    int j = 0;
    while (q != p) {                         // until finding p
        ++q; ++j;                            // advance and count hops
    }
    return j;
}
```

Both of these functions are quite fragile, and are likely to abort if their arguments are not in bounds. A more careful implementation of atIndex would first check that the argument *i* lies in the range from 0 to *n* – 1, where *n* is the size of the sequence.

● Implementing a Sequence with an Array

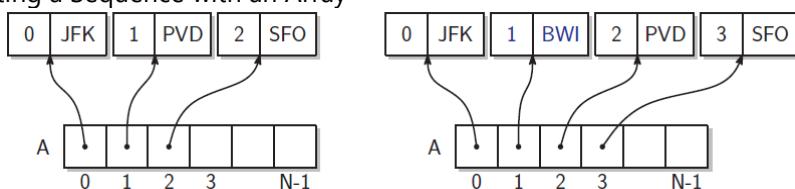


Figure 6.5 An array-based implementation of the sequence ADT.

<i>Operations</i>	<i>Circular Array</i>	<i>List</i>
size, empty	$O(1)$	$O(1)$
atIndex, indexOf	$O(1)$	$O(n)$
begin, end	$O(1)$	$O(1)$
$*p, ++p, --p$	$O(1)$	$O(1)$
insertFront, insertBack	$O(1)$	$O(1)$
insert, erase	$O(n)$	$O(1)$

Figure 6. 6 Comparison of the running times of the functions of a sequence implemented with either an array (used in a circular fashion) or a doubly linked list. We denote with n the number of elements in the sequence at the time the operation is performed. The space usage is $O(n)$ for the doubly linked list implementation, and $O(N)$ for the array implementation, where N is the size of the array.

6.4 Case Study: Bubble-Sort on a Sequence

- The Bubble-Sort Algorithm

The **sorting** problem is to reorder the sequence (n elements) so that the elements are in **nondecreasing** order. The **bubble-sort** algorithm (see Figure 6.7) solves this problem by performing a series of **passes** over the sequence. In each pass, the elements are scanned by increasing rank, from rank 0 to the end of the sequence. At each position in a pass, an element is compared with its neighbor, and if these two consecutive elements are found to be in the wrong relative order (that is, the preceding element is larger than the succeeding one), then the two elements are swapped. The sequence is sorted by completing n such passes.

pass	swaps	sequence
		(5, 7, 2, 6, 9, 3)
1st	$7 \leftrightarrow 2$ $7 \leftrightarrow 6$ $9 \leftrightarrow 3$	(5, 2, 6, 7, 3, 9)
2nd	$5 \leftrightarrow 2$ $7 \leftrightarrow 3$	(2, 5, 6, 3, 7, 9)
3rd	$6 \leftrightarrow 3$	(2, 5, 3, 6, 7, 9)
4th	$5 \leftrightarrow 3$	(2, 3, 5, 6, 7, 9)

Figure 6. 7 The bubble-sort algorithm on a sequence of integers. For each pass, the swaps performed and the sequence after the pass are shown.

The bubble-sort algorithm has the following properties:

- In the first pass, once the largest element is reached, it keeps on being swapped until it gets to the last position of the sequence.
- In the second pass, once the second largest element is reached, it keeps on being swapped until it gets to the second-to-last position of the sequence.
- In general, at the end of the i th pass, the right-most i elements of the sequence (that is, those at indices from $n - 1$ down to $n - i$) are in final position.

- A Sequence-Based Analysis of Bubble-Sort

Assume that the implementation of the sequence is such that the accesses to elements and the swaps of elements performed by bubble-sort take $O(1)$ time each. That is, the running time of the i th pass is $O(n - i + 1)$. We have that the overall running time of bubble-sort is

$$O\left(\sum_{i=1}^n (n - i + 1)\right).$$

We can rewrite the sum inside the big-Oh notation as

$$O\left(\sum_{i=1}^n (n - i + 1)\right) = O(n + (n - 1) + \dots + 2 + 1) = O\left(\sum_{i=1}^n i\right) = O\left(\frac{n(n + 1)}{2}\right)$$

Thus, bubble-sort runs in $O(n^2)$ time, provided that accesses and swaps can each be implemented in $O(1)$ time.

```

void bubbleSort1(Sequence& S) { // bubble-sort by indices
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        for (int j = 1; j < n-i; j++) {
            Sequence::Iterator prec = S.atIndex(j-1); // predecessor
            Sequence::Iterator succ = S.atIndex(j); // successor
            if (*prec > *succ) { // swap if out of order
                int tmp = *prec; *prec = *succ; *succ = tmp;
            }
        }
    }
}

void bubbleSort2(Sequence& S) { // bubble-sort by positions
    int n = S.size();
    for (int i = 0; i < n; i++) { // i-th pass
        Sequence::Iterator prec = S.begin(); // predecessor
        for (int j = 1; j < n-i; j++) {
            Sequence::Iterator succ = prec;
            ++succ; // successor
            if (*prec > *succ) { // swap if out of order
                int tmp = *prec; *prec = *succ; *succ = tmp;
            }
            ++prec; // advance predecessor
        }
    }
}

```

CHAPTER 7: Trees

7.1 General Trees

Productivity experts say that breakthroughs come by thinking “nonlinearly.” In this chapter, we discuss one of the **most important** nonlinear data structures in computing—**trees**.

The relationships in a tree are **hierarchical**, with some objects being “above” and some “below” others. **Actually**, the main terminology for tree data structures comes from family trees, with the terms “parent,” “child,” “ancestor,” and “descendant” being the most common words used to describe relationships.

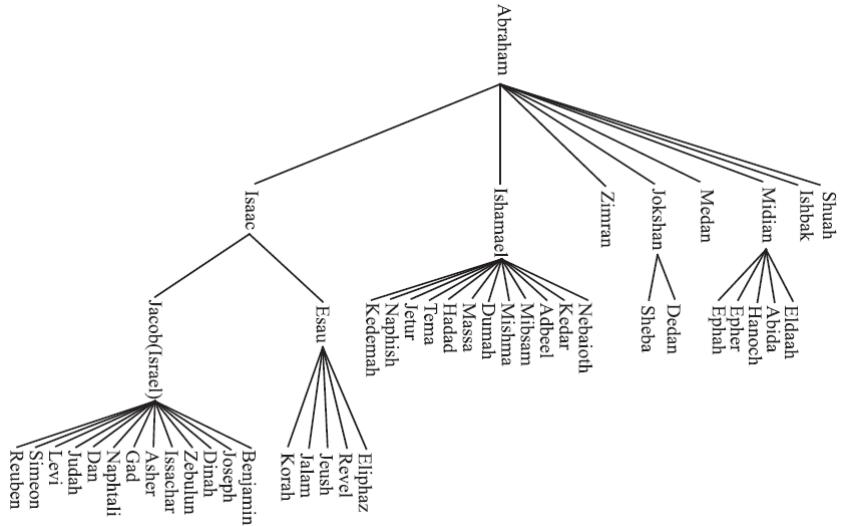


Figure 7. 1 A family tree showing some descendants of Abraham.

- Tree Definitions and Properties

A **tree** is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a **parent** element and zero or more **children** elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. (See Figure 7.2.) We typically call the top element the **root** of the tree, but it is drawn as the highest element, with the other elements being connected below (just the **opposite** of a botanical tree).

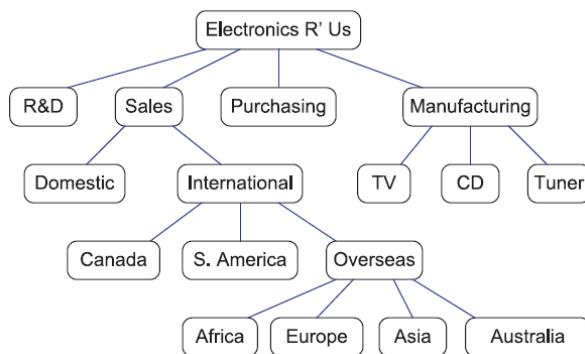


Figure 7. 2 A tree with 17 nodes representing the organizational structure of a fictitious corporation.

Formal Tree Definition

Formally, we define **tree** T to be a set of **nodes** storing elements in a **parent-child** relationship with the following properties:

- If T is nonempty, it has a **special** node, called the **root** of T , that has no parent.
- Each node v of T different from the root has a **unique parent** node w ; every node with parent w is a **child** of w .

Other Node Relationships

Two nodes that are children of the same parent are **siblings**. A node v is **external** if v has no children. A node v is **internal** if it has one or more children. External nodes are also known as **leaves**.

Example 7.1: In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. (See Figure 7.3.) More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX and Linux operating systems, the root of the tree is appropriately called the “root directory,” and is represented by the symbol “/.”

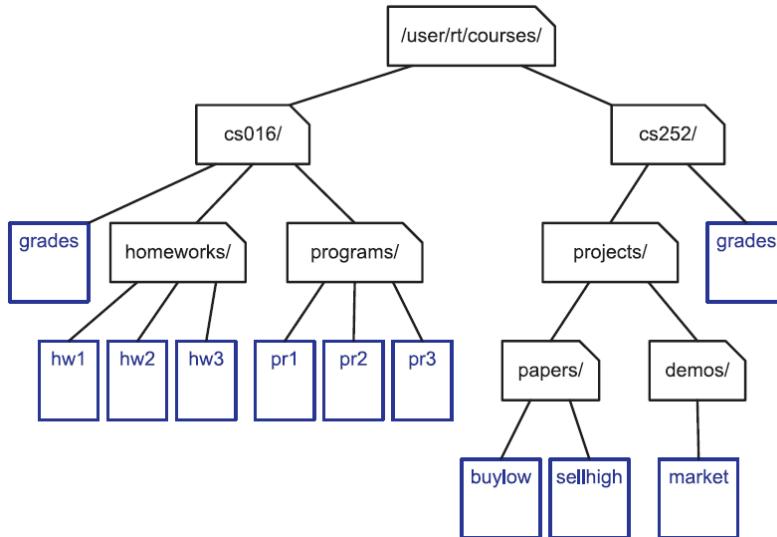


Figure 7.3 Tree representing a portion of a file system.

A node u is an **ancestor** of a node v if $u = v$ or u is an ancestor of the parent of v . Conversely, we say that a node v is a **descendent** of a node u if u is an ancestor of v . For example, in Figure 7.3, cs252/ is an ancestor of papers/, and pr3 is a descendent of cs016/. The **subtree** of T **rooted** at a node v is the tree consisting of all the descendants of v in T (including v **itself**). In Figure 7.3, the subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.

Edges and Paths in Trees

An **edge** of tree T is a pair of nodes (u, v) such that u is the parent of v , or vice versa. A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. For example, the tree in Figure 7.3 contains the path (cs252/, projects/, demos/, market).

Ordered Trees

A tree is **ordered** if there is a **linear** ordering defined for the children of each node;

Example 7.3: (Page 271)

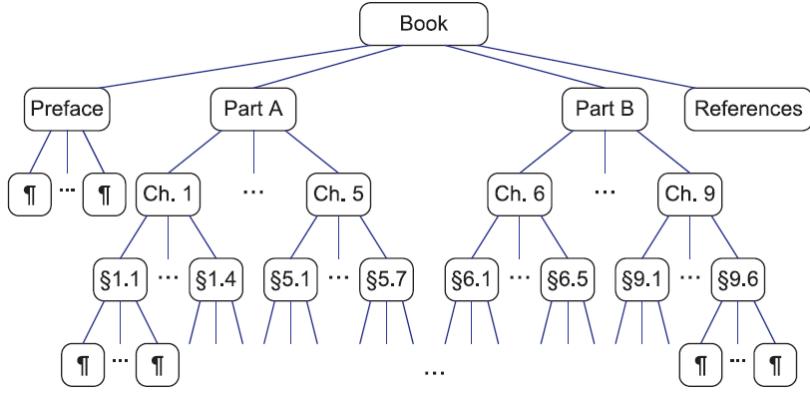


Figure 7.4 An ordered tree associated with a book.

● Tree Functions

The tree ADT stores elements at the nodes of the tree. Because nodes are internal aspects of our implementation, we **do not allow** access to them directly. Instead, each node of the tree is associated with a **position** object, which provides public access to nodes.

It is useful to store collections of positions. For example, the children of a node in a tree can be presented to the user as such a list. We define **position list**, to be a list whose elements are tree positions.

The real power of a tree position arises from its **ability** to access the neighboring elements of the tree. Given a **position** p of tree T , we define the following:

- $p.parent()$: Return the parent of p ; an error occurs if p is the root.
- $p.children()$: Return a position list containing the children of node p .
- $p.isRoot()$: Return true if p is the root and false otherwise.
- $p.isExternal()$: Return true if p is external and false otherwise.

The tree itself provides the following functions.

- $\text{size}()$: Return the number of nodes in the tree.
- $\text{empty}()$: Return true if the tree is empty and false otherwise.
- $\text{root}()$: Return a position for the tree's root; an error occurs if the tree is empty.
- $\text{positions}()$: Return a position list of all the nodes of the tree.

● A C++ Tree Interface

Let us present an **informal** C++ interface for the tree ADT.

```

template <typename E> // base element type
class Position<E> { // a node position
public:
    E& operator*(); // get element
    Position parent() const; // get parent
    PositionList children() const; // get node's children
    bool isRoot() const; // root node?
    bool isExternal() const; // external node?
};

template <typename E> // base element type
class Tree<E> { // public types
public:
    class Position; // a node position
    class PositionList; // a list of positions
public:
    int size() const; // number of nodes
    bool empty() const; // is tree empty?
    Position root() const; // get the root
    PositionList positions() const; // get positions of all nodes
};

```

● A Linked Structure for General Trees

A natural way to realize a tree T is to use a **linked structure**, where we represent each node of

T by a position object p with the following fields: a reference to the node's element, a link to the node's parent, and some kind of collection (for example, a list or array) to store links to the node's children.

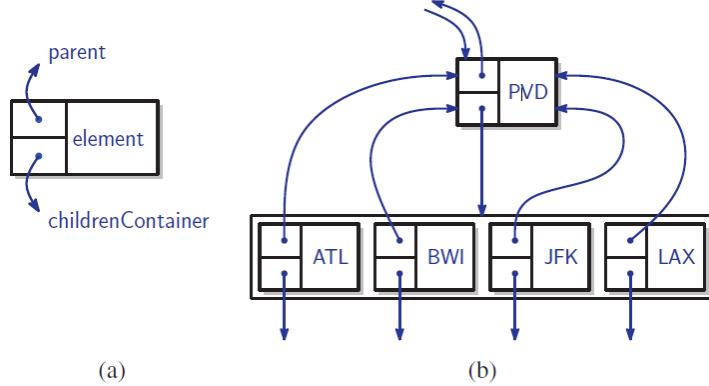


Figure 7.5 The linked structure for a general tree: (a) the node structure; (b) the portion of the data structure associated with a node and its children.

Operation	Time
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Figure 7.6 Running times of the functions of an n -node linked tree structure. Let c_p denote the number of children of a node p . The space usage is $O(n)$.

7.2 Tree Traversal Algorithms

• Depth and Height

Let p be a node of a tree T . The **depth** of p is the number of ancestors of p , excluding p itself. For example, in the tree of Figure 7.2, the node storing *International* has depth 2. Note that this definition implies that the depth of the root of T is 0. The depth of p 's node can also be **recursively** defined as follows:

- If p is the root, then the depth of p is 0
- Otherwise, the depth of p is one plus the depth of the **parent** of p

Algorithm $\text{depth}(T, p)$:

```

if  $p.\text{isRoot}()$  then
    return 0
else
    return 1 +  $\text{depth}(T, p.\text{parent}())$ 

int  $\text{depth}(\text{const Tree\&} T, \text{const Position\&} p)$  {
    if ( $p.\text{isRoot}()$ )
        return 0;                                // root has depth 0
    else
        return 1 +  $\text{depth}(T, p.\text{parent}())$ ;    // 1 + (depth of parent)
}

```

The running time of algorithm $\text{depth}(T, p)$ is $O(d_p)$, where d_p denotes the depth of the node p in the tree T , because the algorithm performs a constant-time recursive step for each ancestor of p .

The **height** of a node p in a tree T is also defined **recursively**.

- If p is external, then the height of p is 0
- Otherwise, the height of p is one plus the **maximum height** of a child of p

The **height** of a tree T is the height of the root of T . For example, the tree of Figure 7.2 has height 4.

Proposition 7.4: The **height** of a **tree** is equal to the **maximum depth** of its **external nodes**.

```

Algorithm height1(T):
    h = 0
    for each p ∈ T.positions() do
        if p.isExternal() then
            h = max(h, depth(T, p))
    return h

int height1(const Tree& T) {
    int h = 0;
    PositionList nodes = T.positions();           // list of all nodes
    for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
        if (q->isExternal())
            h = max(h, depth(T, *q));           // get max depth among leaves
    }
    return h;
}

```

Unfortunately, algorithm height1 is **not** very efficient. (Page 276)

```

Algorithm height2(T, p):
    if p.isExternal() then
        return 0
    else
        h = 0
        for each q ∈ p.children() do
            h = max(h, height2(T, q))
        return 1 + h

int height2(const Tree& T, const Position& p) {
    if (p.isExternal()) return 0;           // leaf has height 0
    int h = 0;
    PositionList ch = p.children();         // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
        h = max(h, height2(T, *q));
    return 1 + h;                         // 1 + max height of children
}

```

Algorithm height2 is more efficient than height1. (Page 277)

Proposition 7.5: Let *T* be a tree with *n* nodes, and let *c_p* denote the **number of children** of a node *p* of *T*. Then $\sum_p c_p = n - 1$.

- Preorder Traversal

A **traversal** of a tree *T* is a systematic way of accessing, or “visiting,” **all** the nodes of *T*.

In a **preorder** traversal of a tree *T*, the root of *T* is visited first and then the subtrees rooted at its children are traversed **recursively**.

We initially invoke this routine with the call `preorder(T, T.root())`.

```

Algorithm preorder(T, p):
    perform the “visit” action for node p
    for each child q of p do
        recursively traverse the subtree rooted at q by calling preorder(T, q)

```

The preorder traversal algorithm is **useful** for producing a **linear ordering** of the nodes of a tree where parents **must** always come before their children in the ordering.

Example 7.6: The preorder traversal of the tree associated with a document, as in Example 7.3, examines an entire document sequentially, from beginning to end. If the external nodes are removed before the traversal, then the traversal examines the table of contents of the document. (See Figure 7.7.)

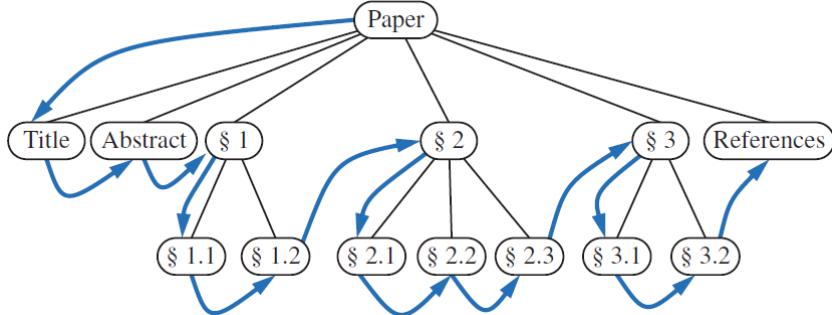


Figure 7.7 Preorder traversal of an ordered tree, where the children of each node are ordered from left to right.

The preorder traversal is also an efficient way to access all the nodes of a tree. (Page 279)

```
void preorderPrint(const Tree& T, const Position& p) {
    cout << *p;                                // print element
    PositionList ch = p.children();              // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
        cout << " ";
        preorderPrint(T, *q);
    }
}
```

There is an **interesting** variation of the `preorderPrint` function that outputs a different representation of an entire tree. The **parenthetic string representation** $P(T)$ of tree T is **recursively** defined as follows. If T consists of a single node referenced by a position p , then

$$P(T) = *p.$$

Otherwise,

$$P(T) = *p + "(" + P(T_1) + P(T_2) + \dots + P(T_k) + ")$$

where p is the root position of T and T_1, T_2, \dots, T_k are the subtrees rooted at the children of p , which are given in order if T is an ordered tree.

The parenthetic representation of the tree of Figure 7.2 is shown in Figure 7.8.

```
Electronics R'Us (
    R&D
    Sales (
        Domestic
        International (
            Canada
            S.America
            Overseas ( Africa  Europe  Asia  Australia ) )
        Purchasing
        Manufacturing ( TV  CD  Tuner ) )
```

Figure 7.8 Parenthetic representation of the tree of Figure 7.2. Indentation, line breaks, and spaces have been added for clarity.

```
void parenPrint(const Tree& T, const Position& p) {
    cout << *p;                                // print node's element
    if (!p.isExternal()) {
        PositionList ch = p.children();          // list of children
        cout << "(";
        for (Iterator q = ch.begin(); q != ch.end(); ++q) {
            if (q != ch.begin()) cout << " ";    // print separator
            parenPrint(T, *q);                    // visit the next child
        }
        cout << ")";
    }
}
```

● Postorder Traversal

Another important tree traversal algorithm is the **postorder traversal**. This algorithm can be viewed as the **opposite** of the preorder traversal, because it **recursively** traverses the subtrees

rooted at the children of the root first, and then visits the root.

Algorithm postorder(T, p):

for each child q of p **do**

 recursively traverse the subtree rooted at q by calling postorder(T, q)

 perform the “visit” action for node p

The name of the postorder traversal comes from the **fact** that this traversal method visits a node p **after** it has visited **all** the other nodes in the subtree rooted at p . (See Figure 7.9.)

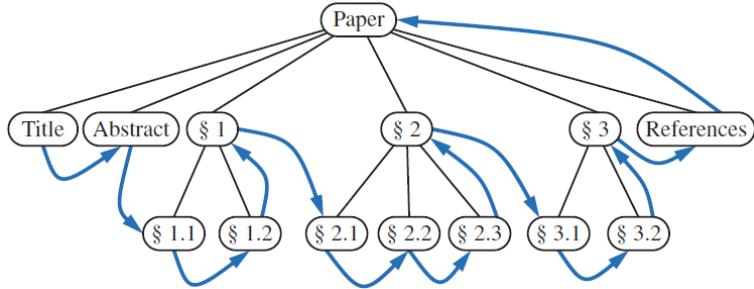


Figure 7.9 Postorder traversal of the ordered tree of Figure 7.7.

```
void postorderPrint(const Tree& T, const Position& p) {
    PositionList ch = p.children(); // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
        postorderPrint(T, *q);
        cout << " ";
    }
    cout << *p; // print element
}
```

The postorder traversal method is **useful** for solving problems where we wish to compute some property for each node p in a tree, but computing that property for p requires that we have already computed that same property for p 's children.

Example 7.7: Consider a file-system tree T , where external nodes represent files and internal nodes represent directories (Example 7.1). Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of the following (see Figure 7.10):

- The size of the directory itself
- The sizes of the files in the directory
- The space used by the children directories

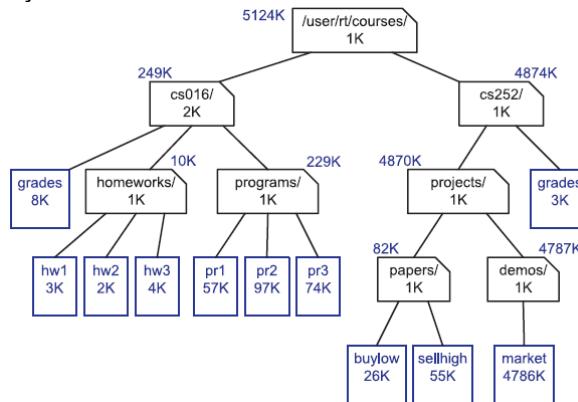


Figure 7.10 The tree of Figure 7.3 representing a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.

```

int diskSpace(const Tree& T, const Position& p) {
    int s = size(p);                                // start with size of p
    if (!p.isExternal()) {                          // if p is internal
        PositionList ch = p.children();           // list of p's children
        for (Iterator q = ch.begin(); q != ch.end(); ++q)
            s += diskSpace(T, *q);                // sum the space of subtrees
        cout << name(p) << ": " << s << endl; // print summary
    }
    return s;
}

```

Other Kinds of Traversals

Preorder traversal is **useful** when we want to perform an action for a node and then recursively perform that action for its children, and postorder traversal is **useful** when we want to first perform an action on the descendants of a node and then perform that action on the node.

We could traverse a tree so that we visit all the nodes at depth d before we visit the nodes at depth $d + 1$. Such a traversal, called a ***breadth-first traversal***, could be implemented using a queue, whereas the preorder and postorder traversals use a stack. In addition, binary trees, which we discuss next, support an **additional** traversal method known as the inorder traversal.

7.3 Binary Trees

A **binary tree** is an **ordered** tree in which every node has at **most two** children.

1. Every node has at most two children.
2. Each child node is labeled as being either a **left child** or a **right child**.
3. A left child precedes a right child in the ordering of children of a node.

The subtree rooted at a left or right child of an internal node is called the node's **left subtree** or **right subtree**, respectively. A binary tree is **proper** if each node has **either** zero or two children. Some people also refer to such trees as being **full** binary trees. Thus, in a proper binary tree, every internal node has **exactly** two children. A binary tree that is not proper is **improper**.

Example 7.8: (Page 284)

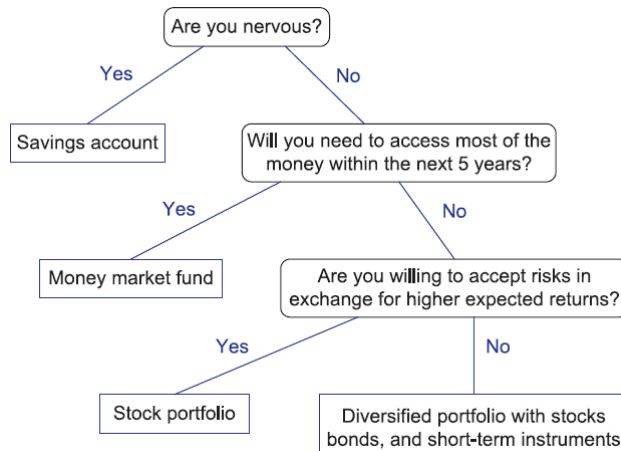


Figure 7. 11 A **decision tree** providing investment advice.

Example 7.9: An **arithmetic expression** can be represented by a tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators $+$, $-$, \times , and $/$. (See Figure 7.12.) Each node in such a tree has a value associated with it.

- If a node is external, then its value is that of its variable or constant.
- If a node is internal, then its value is defined by applying its operation to the values of its children.

Such an arithmetic-expression tree is a **proper** binary tree, since each of the operators $+$, $-$, \times , and $/$ take exactly two operands. **Of course**, if we were to allow for unary operators, like negation ($-$), as in " $-x$," then we could have an **improper** binary tree.

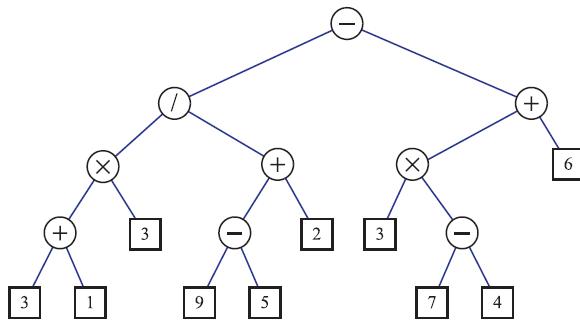


Figure 7.12 A binary tree representing an arithmetic expression. This tree represents the expression $\left(\left((3 + 1) \times 3 \right) / \left((9 - 5) + 2 \right) \right) - \left(\left(3 \times (7 - 4) \right) + 6 \right)$. The value associated with the internal node labeled "/" is 2.

A Recursive Binary Tree Definition

Incidentally, we can also define a binary tree in a **recursive** way such that a binary tree is either empty or consists of:

- A node r , called the **root** of T and storing an element
 - A binary tree, called the **left subtree** of T
 - A binary tree, called the **right subtree** of T

● The Binary Tree ADT

As with our earlier tree ADT, each node of the tree stores an element and is associated with a **position** object, which provides public access to nodes.

 - $p.\text{left}()$: Return the left child of p ; an error condition occurs if p is an external node.
 - $p.\text{right}()$: Return the right child of p ; an error condition occurs if p is an external node.
 - $p.\text{parent}()$: Return the parent of p ; an error occurs if p is the root.
 - $p.\text{isRoot}()$: Return true if p is the root and false otherwise.
 - $p.\text{isExternal}()$: Return true if p is external and false otherwise.
 - $\text{size}()$: Return the number of nodes in the tree.
 - $\text{empty}()$: Return true if the tree is empty and false otherwise.
 - $\text{root}()$: Return a position for the tree's root; an error occurs if the tree is empty.
 - $\text{positions}()$: Return a position list of all the nodes of the tree.

- A C++ Binary Tree Interface

```

tree interface
template <typename E>
class Position<E> {
public:
    E& operator*();
    Position left() const;
    Position right() const;
    Position parent() const;
    bool isRoot() const;
    bool isExternal() const;
};

template <typename E>
class BinaryTree<E> {
public:
    class Position;
    class PositionList;
public:
    int size() const;
    bool empty() const;
    Position root() const;
    PositionList positions() const;
};
// base element type
// a node position
// get element
// get left child
// get right child
// get parent
// root of tree?
// an external node?

// base element type
// binary tree
// public types
// a node position
// a list of positions
// member functions
// number of nodes
// is tree empty?
// get the root
// list of nodes

```

- Properties of Binary Trees

Binary trees have several interesting properties dealing with relationships between their heights and number of nodes. We denote the set of all nodes of a tree T , at the same depth d , as the **level** d of T . In a binary tree, level 0 has one node (the root), level 1 has, at most, two nodes (the children of the root), level 2 has, at most, four nodes, and so on. (See Figure 7.13.) In general, level d has, at most, 2^d nodes.

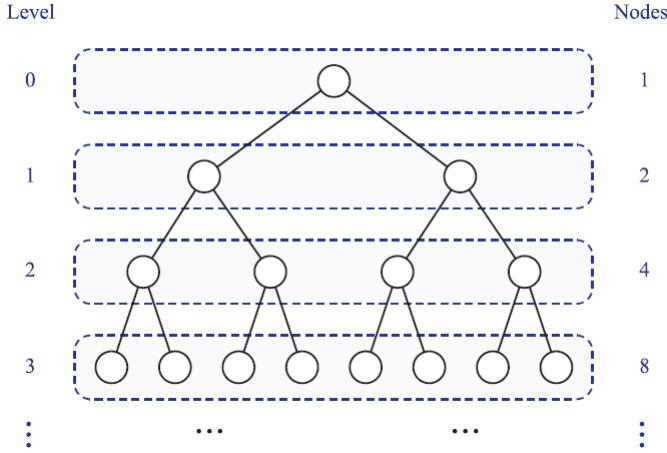


Figure 7.13 Maximum number of nodes in the levels of a binary tree.

Proposition 7.10: Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T , respectively. Then T has the following properties:

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq n - 1$

Also, if T is **proper**, then it has the following properties:

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

Proposition 7.11: In a nonempty proper binary tree T , the number of external nodes is one more than the number of internal nodes.

- A Linked Structure for Binary Trees

In this section, we present an implementation of a binary tree T as a **linked structure**, called `LinkedBinaryTree`. We represent each **node** v of T by a node object storing the associated element and pointers to its parent and two children. (See Figure 7.14.) For simplicity, we assume the tree is **proper**, meaning that each node has either zero or two children.

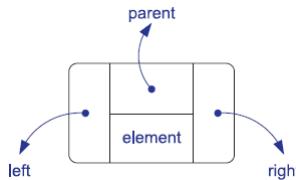


Figure 7.14 A **node** in a linked data structure for representing a binary tree.

In Figure 7.15, we show a linked structure representation of a **binary tree**. The structure stores the tree's size, that is, the number of nodes in the tree, and a pointer to the root of the tree. The rest of the structure consists of the nodes linked together appropriately.

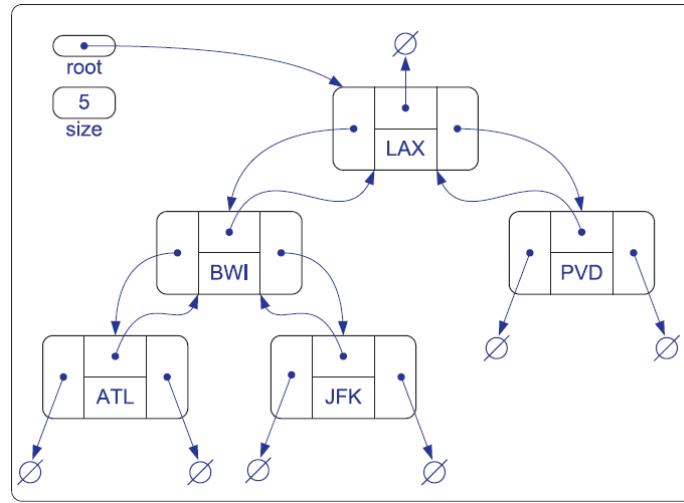


Figure 7. 15 An example of a linked data structure for representing a binary tree.

```

struct Node {
    Elem elt;                                // a node of the tree
    Node* par;                               // parent
    Node* left;                              // left child
    Node* right;                             // right child
    Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
};
class Position {                           // position in the tree
private:
    Node* v;                                // pointer to the node
public:
    Position(Node* _v = NULL) : v(_v) { }      // constructor
    Elem& operator*()                      // get element
    { return v->elt; }
    Position left() const                   // get left child
    { return Position(v->left); }
    Position right() const                  // get right child
    { return Position(v->right); }
    Position parent() const                 // get parent
    { return Position(v->par); }
    bool isRoot() const                  // root of the tree?
    { return v->par == NULL; }
    bool isExternal() const                // an external node?
    { return v->left == NULL && v->right == NULL; }
    friend class LinkedBinaryTree;          // give tree access
};
typedef std::list<Position> PositionList;   // list of positions

```

```

typedef int Elem; // base element type
class LinkedBinaryTree {
protected:
    // insert Node declaration here...
public:
    // insert Position declaration here...
public:
    LinkedBinaryTree(); // constructor
    int size() const; // number of nodes
    bool empty() const; // is tree empty?
    Position root() const; // get the root
    PositionList positions() const; // list of nodes
    void addRoot();
    void expandExternal(const Position& p); // expand external node
    Position removeAboveExternal(const Position& p); // remove p and parent
    // housekeeping functions omitted...
protected: // local utilities
    void preorder(Node* v, PositionList& pl) const; // preorder utility
private:
    Node* _root; // pointer to the root
    int n; // number of nodes
};

LinkedBinaryTree::LinkedBinaryTree() // constructor
: _root(NULL), n(0) { }
int LinkedBinaryTree::size() const // number of nodes
{ return n; }
bool LinkedBinaryTree::empty() const // is tree empty?
{ return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
{ return Position(_root); }
void LinkedBinaryTree::addRoot() // add root to empty tree
{ _root = new Node; n = 1; }

```

Binary Tree Update Functions

expandExternal(*p*): Transform *p* from an external node into an internal node by creating two new external nodes and making them the left and right children of *p*, respectively; an error condition occurs if *p* is an internal node.

removeAboveExternal(*p*): Remove the external node *p* together with its parent *q*, replacing *q* with the sibling of *p* (see Figure 7.15, where *p*'s node is *w* and *q*'s node is *v*); an error condition occurs if *p* is an internal node or *p* is the root.

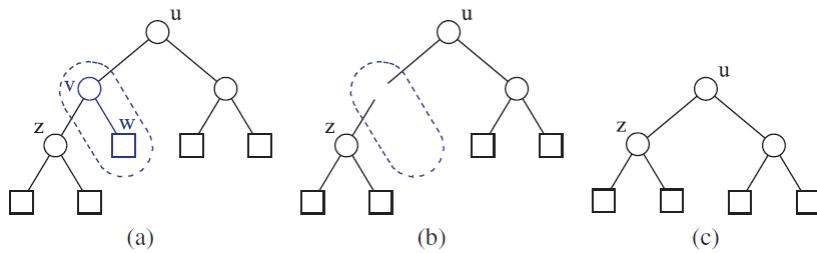


Figure 7.16 Operation `removeAboveExternal(p)`, which removes the external node *w* to which *p* refers and its parent node *v*.