Appendix A

C++ Overview

This appendix is an overview of the C++ programming language for those with little or no experience with it. It is not an introduction to programming principles, but assumes that you are experienced in programming with some other language such as Fortran, Ada, or Java. The coverage is not comprehensive, as C++ is a large and complex language. Rather, it describes those features of the C++ language that you will need for this book and relies mostly on examples, from which you should be able to determine the general rules of writing C++ programs.

A.1 Data Types

Procedural languages like C++ contain variables. Every variable has three characteristics: name, type, and value. The *name* is an identifier, defined by the syntax rules of the language. The *type* can be a built-in or primitive type supplied by the language, or a new type that is defined by the programmer that in some way extends a primitive type. Both the name and the type of a variable are determined when the software designer writes and compiles the program. The *value* of a variable, on the other hand, is stored in the main memory of the computer as the program is executing.

A characteristic that is determined at compile time is called a *static characteristic*, and one that is determined at program execution time is called a *dynamic characteristic*. C++ variables are statically typed because their types are determined at compile time.

Identifiers

A C++ identifier is composed of letters, digits, and the underscore character _. It must begin with either a letter or an underscore. Letters are case sensitive. For example, the identifier `cat` is not the same as `Cat`. Following is a list of the 63 C++ reserved words, which are lowercase. They cannot be used as identifiers.

```
asm             auto            bool
break           case            catch
char            class           const
const_cast      continue        default
delete          do              double
```

```
dynamic_cast        else                reinterpret_cast
explicit            export              signed
false               float               static_cast
friend              goto                template
inline              int                 true
mutable             namespace           typeid
operator            private             unsigned
public              register            void
return              short               while
sizeof              static
struct              switch
this                throw
try                 typedef
typename            union
using               virtual
volatile            wchar_t
```

The style standard for this book is to begin each programmer-defined identifier with a lowercase letter. Subsequent words within the identifier begin with an uppercase letter for legibility. For example, the identifier `insertinmid` is more difficult to read than `insertInMid`. This style is known as *camel case* because the identifier has the shape of a camel.

C++ has two kinds of comment—line oriented and line independent. Line oriented comments begin with a double slash anywhere on a line `//` and continue to the end of the line. They do not automatically continue on to the next line. Line independent comments always come in pairs. The comment begins with `/*` anywhere on a line and ends with `*/` anywhere after the `/*` on the same line or anywhere on a later line.

Primitive types

The built-in types are

`bool`	boolean with possible values true or false
`char`	integer whose range can hold a single character, usually ASCII
`double`	double-precision floating-point number
`float`	single-precision floating-point number
`int`	integer whose range is machine dependent

Type `float` is infrequently used, because all computations in C++ are carried out in double precision anyway. There is no computation time benefit to using `float`. Furthermore, literal numeric constants, such as `5.72` are interpreted as type `double` by default.

In C++, you declare a variable by stating its type followed by one or more spaces or tabs followed by its name. Multiple variables of the same type can be declared together by separating their names with a comma. For example,

```
int i, j;
double d;
char ch;
```


declares variables `i` and `j` to be integers, `d` to be a double precision floating point real, and `ch` to be a character.

A constant of type `char` is written enclosed in single quotes, for example, `'B'`. Special non printable characters are preceded by the backslash character in Unix style. Here are some common ones.

`\n`	newline
`\t`	horizontal tab
`\b`	backspace
`\\`	backslash
`\'`	single quote
`\"`	double quote

A curious feature of C++ is that char is an integer usually represented with a single machine byte (eight bits). Because it is an integer you can do arithmetic with it. For example, if you have determined that ch is an uppercase character, then the expression

```
ch + 'a' -'A'
```

is the corresponding lowercase character.

You can specify additional types with the modifiers `unsigned`, `long`, and `short`, which are usually applied to type `int`. When applied to integers, the `int` can be omitted. Here are some examples.

`short int`	integer with possibly smaller range of values than `int`
`short`	same as `short int`
`long int`	integer with possibly larger range of values than `int`
`long`	same as `long int`
`unsigned int`	integer restricted to nonnegative values

The `sizeof` operator gives the size of each type as a multiple of the size of a `char`. The more bytes used to store the value, the larger the range. The specific ranges of the types are not specified in the C++ language definition. Instead, the ranges satisfy

$$\begin{aligned} 1 &= \texttt{sizeof(char)} \\ &\leq \texttt{sizeof(short)} \\ &\leq \texttt{sizeof(int)} \\ &= \texttt{sizeof(unsigned)} \\ &\leq \texttt{sizeof(long)} \end{aligned}$$

The `unsigned` modifier restricts the range to nonnegative values, but does not change the number of values that are capable of being stored by the variable. For example, a `short` integer may be stored as a single byte. In two's complement representation, the most common representation in today's computers, the range of possible values would be -128 to 127, a total of 256 possible values. If the one-bye integer were unsigned, the range of possible values would be 0 to 255, a total of 256 possible values as well.

C++ evolved from C, which in the beginning had no primitive boolean type. Instead, boolean values were represented as integers. The interpretation rule was:

false – zero
true – any value not equal to zero

C++ continued this convention in its early years, and only later did the language provide the primitive `bool` type. Because of this history boolean variables are considered to be integers in C++ and can legally be used in arithmetic expressions. When combined in an integer expression, false is converted to 0 and true is converted to 1. For example, if `i` and `b` are declared as

```
int i;
bool b;
```

and `i` has the value 5 and `b` has the value true, then the expression

```
i + b
```

is perfectly legal and has the value 6.

A.2 Operations

Expressions are central in the C++ language. An expression is like a function because it returns a value. Some expressions not only return a value, but perform an operation as a side effect.

Operators in C++ are either unary, binary, or trinary. *Unary operators* have one operand with an operator symbol that sometimes follows the operand (postfix) and sometimes precedes it (prefix). *Binary operators* are infix, with the operator between two operands. The *trinary operator* has two operator symbols between three operands.

The assignment operator

The assignment operator is =. It evaluates the expression on the right hand side and assigns the value to the variable on the left hand side. For example, if `i` is an integer variable with value 7, and `j` is an integer variable with value 4, the assignment statement

```
i = 2 * j + 3;
```

gives the value of 2 times 4 plus 3, which is 11, to `i`. The original value of 7 for `i` is gone forever, as a variable can hold only one value.

Languages of the Algol lineage use `:=` for the assignment operator to distinguish it from the equals operator. If your prior programming experience was with one of those languages, the = symbol in C++ will undoubtedly be a source of headaches. The problem is that those languages use the = symbol for equality, to be consistent with the equality notation in mathematics. In C++, = does not mean equals. It means assignment. C++ has another symbol for equality. You should read the assignment operator as "gets". Read the above statement as "`i` *gets* two times `j` plus three." Do not read it as "`i` *equals* two times `j` plus three."

Unlike many programming languages, C++ treats an assignment as an expression that returns a value that can be used in another expression. The value returned is the value assigned to the variable, and multiple assignments are right associative. That is, they are evaluated from right to left. For example, the expression

```
i = j = 0;
```


is equivalent to

```
i = (j = 0);
```

and has the effect of first assigning 0 to `j` and then assigning the returned value, 0, to `i`.

You can also use the assignment symbol when you declare a variable to give it an initial value. For example, the statements

```
int i = 0;
double d = 3.14;
```

both declare the variables and set their initial values. The programming style in this book is to initialize variables so they will always have well-defined values.

Arithmetic operators

The infix binary arithmetic operations are

`+` addition
`-` subtraction
`*` multiplication
`/` integer or floating-point division, depending on operand types
`%` remainder of integer division, undefined for floating-point operands

Any arithmetic operator can be combined with the assignment operator to create a compound assignment. For example, the expression

```
i += 5;
```

which you should read as "i plus gets 5," is the same as

```
i = i + 5;
```

Similarly,

```
i *= 2;
```

is the same as

```
i = i * 2;
```

Compound assignments can also be constructed with the bitwise operators described below. The C++ compound assignment operators are

```
+=      -=      *=      /=      %=
<<=     >>=     &=      ^=      |=
```

whose meaning should be obvious from the examples.

C++ also has the famous unary increment and decrement operators, which can be used in either prefix form or postfix form.

`n++` return the current value of `n`, then increment `n` by 1
`++n` increment `n` by 1, then return the new value of `n`
`n--` return the current value of `n`, then decrement `n` by 1
`--n` decrement `n` by 1, then return the new value of `n`

Each of the operators returns a value and changes the value of `n` as a side effect. Here are some examples of the increment and compound assignment operators.

```
n += 1;     is equivalent to   n = n + 1;
n++;        is equivalent to   n = n + 1;
m = n++;    is equivalent to   m = n; n = n + 1;
m = ++n;    is equivalent to   n = n + 1; m = n;
```

The last two examples do not produce the same effect. If `n` has value 7, then

```
m = n++;
```

gives `m` the value 7, while

```
m = ++n;
```

gives `m` the value 8. In both cases the final value of `n` is 8. C++ is an extension of the C language, and its inventor used a little play on words with the postfix increment operator when he named his new language.

Relational and logic operators

The following relational operators return 0 for false and 1 for true.

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

These operators provide the novice programmer with a wonderful opportunity to write incorrect code that is difficult to debug. Here is a typical scenario. Suppose `i` is an integer variable and you write the statement

```
if (i = 0)
```

forgetting that the equals operator is ==. You think your code is testing if `i` equals zero. The `if` statement is perfectly legal in C++ and the compiler will generate code. The problem is that the code it generates is not what you intend. When the `if` statement executes, `i` gets 0, and that 0 is returned to the `if` statement, which always interprets it as false regardless of the original value of `i`. The original value of `i` is destroyed. The reason this bug is so insidious is that the language permits it, and the programmer must track down the bug during execution. This bug is so common that most compilers will issue a warning, because the probability is so low that a programmer intends to do an assignment where a test is normally used.

The following logical operators interpret zero as false, any nonzero quantity as true, and return 0 or 1.

`&&`	boolean AND (conjunction $\wedge$), binary
`||`	boolean OR (disjunction $\vee$), binary
`!`	boolean NOT (negation $\neg$), unary

The logical operators use short-circuit evaluation, in which the second of two operands is skipped if the first operand determines the truth value. For example, the boolean expression

```
(i >= 0) && (list[i] < limit)
```

first tests if `i` is greater than or equal to zero. If `i` is less than zero, the second comparison of an element of the array list is short-circuited, that is, it is not evaluated. The boolean expression will be false regardless of the value of the second comparison because of the logical properties of conjunction. Similarly, if the first operand of a disjunction is true then true is returned without evaluating the second operand.

Bitwise operators

There is a set of bitwise operators that manipulate the machine-dependent bit representations of integer operands.

`~`	one's complement, unary
`<<`	left shift, binary
`>>`	right shift, binary
`&`	bitwise AND, binary
`|`	bitwise OR, binary
`^`	bitwise exclusive OR, binary

Be careful to not write `&` when you mean to write `&&`.

An example of the left shift operator is

```
i << 3;
```

which shifts `i` three places to the left hence returning its value multiplied by 8. The left and right shift operators are rarely used this way in C++. Most programs use a standard input/output library that redefines `<<` as the output operator and `>>` as the input operator.

C++ also has a trinary conditional operator consisting of the two symbols `?` and `:` used in the form

```
expr1 ? expr2 : expr3
```

If `expr1` is true it returns `expr2`, otherwise it returns `expr3`. For example, the statement

```
myMax = left > right ? left : right;
```

sets `myMax` to the maximum of `left` and `right`.

The above examples all assume a rather involved precedence order as shown in Figure A.1. It lists the operators in order of decreasing strength. Operators are binary unless otherwise noted.

Type conversions

Many numeric type conversions are automatic. When you mix integer and real types in an arithmetic expression, the compiler converts integer types to real. The following examples assume `i` and `j` are integers and `d` is a double precision real.

To evaluate the expression

Operators	Associativity
`::`	left to right
`[]` `()` `->` `.` *po* `++` *po* `--`	left to right
`!` `~` *pr* `++` *pr* `--` *un* `+` *un* `-` *un* `*` *un* `&` `new` `delete` `sizeof`	right to left
`(`*type*`)`	right to left
`*` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
tr `?` `:`	right to left
`=` `+=` `-=` `*=` `/=` `=|` `%=` `&=` `^=` `<<=` `>>=`	right to left
`,`	left to right

Figure A.1 The precedence of the C++ operators. The operators are listed in order of decreasing precedence. An operator higher in the table has precedence over an operator lower in the table, and will evaluate first. The symbol *po* indicates the postfix version of the operator, *pr* the prefix version, and *un* the unary version. The symbol *tr* indicates the trinary operator. The symbol *type* indicates the `()` operator when it is used to type cast.

```
i * 3.14;
```

C++ first converts integer `i` to double, then performs the multiplication. It returns type double. The assignment

```
d = i * 3.14;
```

gives the double precision value to `d`. The assignment

```
j = i * 3.14;
```

however, performs yet another conversion. If you assign a real to an integer, C++ truncates the real value, that is, it discards the fractional part. `j` gets the truncated value.

Assigning a real to an integer is illegal in many other programming languages, and may produce a compiler warning with your C++ compiler.

You must take special care with the `/` operator, as it has a different meaning when applied to integer operands than when applied to real operands. The expression

```
d / 3.14
```

performs real division and returns a double. The expression

```
d / 3
```

converts integer 3 to double, performs the real division, and returns a double. However, if both the operands are integer, `/` performs integer division truncating the result and returning an integer. This operation is sometimes denoted `div` in other programming languages for the integer divide operation. The expression

```
i / 3
```

performs integer division. If `i` has value 17, the result is integer 5. The `%` operator is sometimes denoted `mod` in other programming languages for the modulus operation.

```
i % 3
```

is integer 2, the remainder when you divide `i` by 3.

C++ provides two notations for changing a value from one type to another type—the old-style cast from the original C language and the new-style casts with the operators `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`. This book sometimes has occasion to use casts and generally uses the new-style cast.

In case you ever run across the old-style cast, here is a description of it. To convert an expression to a different type precede it with the type enclosed in parentheses. For example, the expression

```
(int) (d + 45.0)
```

adds 45.0 to d producing a double precision value, which is then converted to an integer by truncation with the type cast. The fourth entry of the precedence table in Figure A.1, (type), refers to this old-style cast.

The new-style cast that corresponds to the above example is the `static_cast` operator. It converts between related types (e.g. between numeric types) and is used like a function with the type enclosed in angle brackets <>. The above expression written with the new-style cast is

```
static_cast<int>(d + 45.0)
```

The other new-style casts are described as the situations arise.

A.3 Input/Output

The input/output features of C++ are not defined in the language. Instead, they are contained in a library of input/output functions known as `iostream`. This library overloads the left shift operator << as the output operator and the right shift operator >> as the input operator. Overloading, a feature that Section A6 describes in more detail, in effect

changes the meaning of the operator depending on its operands. The operating system maintains a standard output stream, normally the screen, and a standard input stream, normally the keyboard. The library `iostream` associates the standard output stream with `cout` and the standard input stream with `cin`.

Prompting for input

For example, to send the value of double precision variable amount to the screen, preceded by some identifying text you could execute

```
cout << "amount = " << amount << endl;
```

The output operator sends three things to the standard output stream—the identifying text, the value of the variable, and a command, `endl`, for the cursor to position itself at the start of the next line.

To prompt for a value of `myData` from the keyboard you could execute something like

```
cout << "Enter a real value: ";
cin >> amount;
```

Formatting output

The format of the output depends on the type and the value of the variable whose value is to be sent to the screen. The default is to use the minimum number of characters needed to represent the value. Any surrounding white space must be supplied explicitly.

For example, suppose tax has value 4.56789. Then the output statement

```
cout << "Your tax is $" << tax << endl;
```

produces

```
Your tax is $4.56789
```

You can format the output for both integers and reals with the `width` function, which sets the field width in which the integer or real value is displayed. You must set the field width each time you use the << output operator, because each execution of the operator resets the field width to its original default value. For ease of aligning columns of numeric values, they are right justified in the field and padded with leading spaces.

If integer variable `num` has the value 123, the following C++ code

```
cout.width(5); cout << num << endl;
cout.width(4); cout << num << endl;
cout.width(3); cout << num << endl;
cout.width(2); cout << num << endl;
```

produces this output.

```
  123
 123
123
123
```


The last line above shows that if the field width is too small to contain the value, the field width expands to accommodate the minimum number of characters necessary to represent the value.

For floating point numbers the precision function controls the number of significant figures displayed. The following C++ code

```
cout.width(6); cout.precision(5); cout << tax << endl;
cout.width(6); cout.precision(4); cout << tax << endl;
cout.width(6); cout.precision(3); cout << tax << endl;
cout.width(6); cout.precision(2); cout << tax << endl;
cout.width(6); cout.precision(1); cout << tax << endl;
```

produces this output.

```
4.5679
 4.568
  4.57
   4.6
     5
```

when tax has the value 4.56789.

If you want to print a column of floating point numbers so that the decimal points are aligned, you can set a flag to change the meaning of precision to be the number of places past the decimal point instead of the number of significant figures displayed. The following C++ code

```
cout.width(7); cout.precision(2);
cout.setf(ios::fixed | ios::showpoint);
cout << 123.456 << endl;
cout.width(7); cout.precision(2);
cout.setf(ios::fixed | ios::showpoint);
cout << 123.45 << endl;
cout.width(7); cout.precision(2);
cout.setf(ios::fixed | ios::showpoint);
cout << 23.4 << endl;
cout.width(7); cout.precision(2);
cout.setf(ios::fixed | ios::showpoint);
cout << 3. << endl;
```

produces this output.

```
 123.46
 123.45
  23.40
   3.00
```

A.4 Control Structures

Each simple statement in C++ ends with a semicolon `;`. This syntax is different from the programming language Pascal where a semicolon separates two statements. Statements can be grouped together by enclosing them with braces `{}` to make a compound statement.

Selection

The condition of an `if` statement must be enclosed in parentheses `()`. If more than one statement is to be included in the true or the false part of an `if` statement they must be enclosed by braces `{}`. If the braces are omitted, C++ assumes that only one statement is in the true or false part. The style convention in this book is to use braces regardless of whether they are syntactically necessary.

This `if` has no `else` part. It executes *statement1* if *condition* is true. *statement2* executes regardless of *condition*.

```
if (condition) {  // Preferred style
     statement1;
}
statement2;
```

C++ syntax permits you to write the equivalent statement without braces.

```
if (condition)  // Not recommended
     statement1;
statement2;
```

If more than one statement is to be included in the true alternative, braces are mandatory.

```
if (condition) {
     statement1;
     statement2;
}
statement3;
```

If you forget the braces and write the statement as

```
if (condition)  // Misleading
     statement1;
     statement2;
statement3;
```

the compiler will not include *statement2* in the true part, but will compile your code as if you had written

```
if (condition) {
     statement1;
}
statement2;
statement3;
```

As with the true alternative of an `if` statement, the style convention for this book is to always include braces with the else part regardless of whether they are syntactically necessary.

```
if (condition) {  // Preferred style
     statement1;
     statement2;
```



```
} else {
    statement3;
}
statement4;
```

The braces around *statement3* are not syntactically necessary.

Unfortunately, C++ syntax based on the compound statement suffers from the famous "dangling else problem" of many old languages. The problem occurs with nested `if` statements having one `else` part. Suppose you write the following `if` statement without any indentation and not in the preferred style.

```
if (condition1) // Bad style
if (condition2) {
statement1;
} else {
statement3;
}
statement4;
```

The problem is that there are two `if` statements but only one `else` part. The question is, To which `if` statement does the `else` belong—the `if` with *condition1* or the `if` with *condition2*? The C++ compiler ignores indentation, but there are two ways to indent the code fragment depending on which `if` the `else` belongs to.

```
if (condition1)
    if (condition2) {
        statement1;
    } else {
    statement3;
}
statement4;
```

```
if (condition1)
    if (condition2) {
        statement1;
    } else {
        statement3;
    }
statement4;
```

The answer to the question is that an `else` part goes with the most recent `if`. So, the indentation of the second code fragment above is correct. Even if you indent as in the first code fragment, the compiler will produce code for the second fragment.

It sometimes happens that an `if` statement is nested inside the `else` part of another `if`, as in the following fragment on the left. When this code pattern occurs, you can dispense with a pair of braces and simplify the indentation as in the fragment on the right.

```
if (condition1) {
    statement1;
} else {
    if (condition2) {
        statement2;
    } else {
        statement3;
    }
}
statement4;
```

```
if (condition1) {
    statement1;
} else if (condition2) {
    statement2;
} else {
    statement3;
}
statement4;
```

The `switch` statement in C++ allows a direct branch to a `case` label. You can

insert an optional `default` label that executes if the switch expression is not equal to one of the case values. If the default clause is not present and none of the cases match, no action takes place. It is good programming practice to include a default case to test for errors even when all the expected cases are present in your code. A `break` statement exits the `switch`. If you do not include the `break` statement control will simply flow through to the next case, which is usually not the desired effect.

Here is an example that shows how to include more than one case in an alternative.

```
switch (expression) {
case value1:
     statement1;
     break;
case value2: case value3:
     statement2;
     break;
case value4:
     statement3;
     statement4;
     break;
default:
     statement5;
     break;
}
statement6;
```

Repetition

C++ has three loop statements—the `while`, the `do`, and the `for`.

The `while` and the `do` loops differ primarily in the location of the test for termination. The `while` loop tests for termination at the beginning of the loop body, and the `do` tests for termination at the end of the loop body. It is possible for the body of the `while` loop to never execute. The body of the `do` loop is guaranteed to execute at least one time. Both loops execute while the test condition is true. Here is an illustration of the syntax for both.

```
while (condition) {
     statement1;
     statement2;
}
statement3;
```

```
do {
     statement1;
     statement2;
} while (condition);
statement3;
```

The general form of the `for` statement is

```
for (expression1; expression2; expression3) {
     statement;
}
```

expression1 is an initialization expression, normally an assignment statement, *expression2* is the condition for loop termination, and *expression3* is executed at the end of each loop. The above `for` statement is exactly equivalent to the following `while` loop.


```
expression1;
while (expression2) {
    statement;
    expression3;
};
```

To loop through the elements of an array starting with the first one, you could write

```
for (int i = 0; i < CAPACITY; i++) {
    statement;
}
```

where *statement* would use the value of `i` as an index to the array. It is common C++ practice to initialize a new control variable `i` in a `for` statement. To go through the elements in reverse order, you could write

```
for (int i = CAPACITY - 1; 0 <= i; i--)
```

And, to process every other element of the array starting with the first one, you could write

```
for (int i = 0; i < CAPACITY; i += 2)
```

A `break` statement can occur in the body of a `while`, `do`, or `for` statement as well as in a `switch` statement. A `break` causes the innermost enclosing loop or switch to be exited immediately. Inside a loop, it is normally used in conjunction with an `if` statement to test for loop termination at some place neither at the beginning nor the end of the loop body.

```
while (true) {
    statement1;
    if (condition)
        break;
    statement2;
}
```

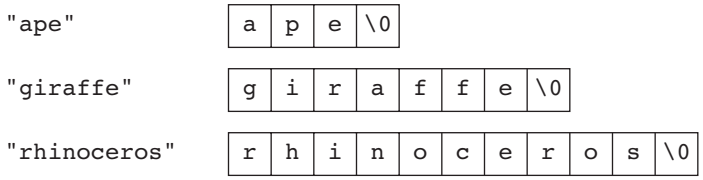
A.5 Arrays

An *array* is a finite collection of values, all of which have the same type. An array is a direct access data structure. That is, you can access any element of an array directly without sequentially accessing all its preceding elements.

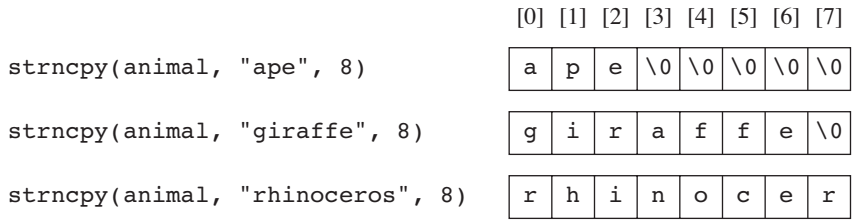
Index range

All arrays in C++ have a starting index of zero. To declare an array, you specify the number of elements in the array, not the value of the last index. The value of the last index will be one less than the number of elements that you specify. For example, the declaration

```
double data[16];
```

(a) Storage for three string literal constants. The number of cells required to store a string is one plus the length of the string.



(b) The effect of three executions of `strncpy`. In each case, the number of characters copied is equal to the capacity of the destination array.

Figure A.2 String literal constants and the `strncpy` function.

declares data to be an array of 16 values indexed from 0 to 15. The first is stored in `data[0]`, the second in `data[1]`, and the last in `data[15]`.

C++ does not automatically check for index values that are out of range. With the above declaration, you can execute

```
data[17] = 5.3;
```

and clobber memory in some unpredictable, probably disastrous way without receiving a "subscript range" error message. This unsafe ability of C and C++ to access memory outside the bounds of the array helps to make virus programming possible.

Character arrays

String literal constants are sequences of characters terminated with the null character, written `\0`. In C++, you indicate a string constant by enclosing it between double quotes as Figure A.2(a) shows. Even though the word "ape" has only three letters, it requires four cells for storage because of the requirement for storing the sentinel `\0`. When you write a string literal between double quote marks the compiler automatically supplies the terminating `\0`. The length of a string literal is the number of characters it contains, not including the sentinel. The length of "ape" is three. The number of cells required to store a string literal is one plus the length of the string because of the sentinel. "ape" requires four cells for storage because its length is three.

Be careful to distinguish double quotes for strings and single quotes for characters. String `"B"` contains two characters because of the sentinel, while character `'B'` contains only one character.

To store a string value in a variable requires that the variable be an array of characters. The following declaration accommodates a string of up to seven characters.


```
char animal[8];
```

Even though animal has eight cells, each capable of holding a single character, the array can only accommodate a string value of length seven because the last character must be `\0`.

To assign a string literal to an array of characters, you cannot use a simple assignment statement. The attempted assignment

```
animal = "giraffe"  // Compile error
```

is illegal, even though the number of characters in the literal (including the `\0` sentinel) exactly fits the array of characters.

C++ requires you to copy the characters one at a time into the array.

```
animal[0] = 'g';
animal[1] = 'i';
animal[2] = 'r';
animal[3] = 'a';
animal[4] = 'f';
animal[5] = 'f';
animal[6] = 'e';
animal[7] = '\0';
```

Fortunately, there is a library function in the `cstring` library that will do the character copy for you automatically. It is called `strncpy` and it requires you to supply the destination string, followed by the source string, followed by the number of characters to copy, including the sentinel. The function call

```
strncpy(animal, "giraffe", 8);
```

copies the eight characters, including the sentinel, to `animal` as desired.

If the string to be copied into animal has fewer than seven characters, you can still supply eight for the number of characters to copy. `strncpy` will pad the destination string with null characters as shown in Figure A.2(b). The precondition for `strncpy` to copy a sentinel from the source string is that the length of the source be less than the number of characters copied. If the source string is too long, the last character copied will be one of the characters in the source string and the destination array will not contain a sentinel `\0`.

An even worse situation arises if the number of characters to be copied exceeds the capacity of the destination array. For example, if you execute

```
strncpy(animal, "rhinoceros", 11);  // Serious error
```

the function will copy characters beyond the rightmost boundary of the animal array. Remember that C++ does not automatically check for index values that are out of range. This call will corrupt memory in unknown ways and could cause unpredictable errors at execution time. The precondition for `strncpy` to not write past the boundary of the destination of the array is that the number of characters to be copied is less than or equal to the capacity of the destination array. It is up to the programmer to supply a number of characters to copy that does not exceed the capacity of the destination array.

One common operation on string values is to compare for alphabetic order. For example, the string "berry" should be greater than the string "bear" because berry comes

after bear in alphabetic order. You cannot use any of the relational operators to compare strings. For example, if animal has the value "berry" and creature, also an array of eight characters, has the value "bear" the statement

```
if (animal < creature)  // Legal but meaningless
```

is a legal C++ statement, but it does not compare the string values in the two arrays. (It makes a comparison that is meaningless for our purposes, and will not be described here.) The capability of using the relational operators to compare string variables is common in other programming languages. So, beware if you are used to this notation for alphabetic comparisons.

Two-dimensional arrays

You declare a two-dimensional array by including a second subscript bracket. For example, the declaration

```
double matrix[4][8];
```

declares 32 values for `matrix` with elements ranging from `matrix[0][0]` to `matrix[3][7]`. Some programming languages allow you to reference the element of a two-dimensional matrix with a single bracket with the first and second subscript separated by a comma as

```
d = matrix[2, 5]; // Legal syntax but misleading
```

However, C++ requires the indices to be in separate brackets as

```
d = matrix[2][5];
```

You might wonder why the compiler does not complain about the comma notation. In C++, the comma is the sequencing operator, which, from Figure A.1, associates from left to right. The expression

```
2, 5
```

is a sequence of two expressions, 2 and 5, which returns 5. So, the expression `matrix[2,5]` is equivalent to `matrix[5]`. Furthermore, even though matrix is two-dimensional, this expression is legal for reasons not described here. Two-dimensional arrays in C++, especially when they are passed as parameters to functions, are notoriously tricky to program correctly.

Constants, typedefs, and enumerations

As with most other programming languages, C++ has a facility to name constants. You put the keyword `const` before the type in the declaration and complete the declaration with the assignment operator. For example, the declarations

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
const int CAPACITY = 1024;
const double SALES_TAX_RATE = 0.05;
```


create four integer constants and one double constant. The value of a constant cannot be changed. For example, it is an error to make the assignment

```
GREEN = 2;  // Illegal
```

or to use one of the increment or decrement operators, such as `GREEN++`. The style convention in this book is to spell the constant with all uppercase letters Words within an identifier are separated by an underscore character _.

You can use a named constant anywhere you can use a literal constant of the same type. One common situation is to prevent the use of *magic numbers* in your code. A magic number is a value, usually other than 0 or 1, that should be documented with a name. For example, the declaration

```
double data[1024];  // Bad style, magic number
```

is better written

```
double data[CAPACITY];
```

Magic numbers are bad for two reasons. First, they make your code difficult to read. If you see some number like `0.05` in an expression somewhere, how do you know what it represents? However, if you see the same expression with the constant `salesTaxRate` in place of the `0.05` the significance of the value is more apparent. Second, magic numbers make your code difficult to modify. If the sales tax rate ever changes from `0.05` to `0.06` and you have used a named constant you only need to change one line of code to change the tax rate everywhere it occurs in your program.

The `typedef` facility in C++ permits you to make an synonym for the name of a type. For example, suppose you have an algorithm to store and manipulate some values in an array. If the values are double precision reals, you might declare the variables

```
double vector[CAPACITY];
double temp;
```

with similar declarations throughout your program. Then, if you want to modify your program so that the values to be stored and manipulated are integers, you would need to change the word `double` in the above declarations and all the similar ones throughout your program to `int`.

However, if you use the `typedef` facility you can declare your variables with the synonym `MyType` as follows.

```
typedef double MyType;
MyType vector[CAPACITY];
MyType temp;
```

and use the name of the type `MyType` similarly throughout your program. Then, if you ever want to change your code to process integers instead of reals, you only need to change one line of code

```
typedef int MyType;
```

The advantage of naming a type is similar to the advantage of naming a constant. The style convention is to begin each programmer-defined type name with an uppercase letter.

Enumerations are another facility in C++ that have features similar to both named constants and named types. An enumeration defines a type and a set of constant values that are essentially integers. An enumeration begins with the keyword `enum`, followed by the name of the type, followed by a set of named integer constants enclosed in braces and separated by commas. By default, values of the constants increase sequentially starting with zero.

For example, the type definition

```
enum EColorType {E_RED, E_GREEN, E_BLUE};
```

declares `EColorType` to be an enumerated type with `E_RED` having value 0, `E_GREEN` having value 1, and `E_BLUE` having value 2. Enumerated values provide type checking and range checking for assignment statements that integer constants do not provide. If you want to limit the values that a variable can take to be only those enumerated in the declaration, you make the variable the enumerated type. With the declarations

```
int color;
EColorType myEColor;
```

and `GREEN` defined to be a `const int` as in the previous example, the assignments

```
color = GREEN;
color = E_GREEN;
myEColor = E_GREEN;
```

are all legal. But the assignments

```
myEColor = GREEN;  // Illegal
myEColor = 1;  // Illegal
```

are not. Enumerations force you to use the named values when making assignments to variables that have the enumeration type. Even making an assignment from the equivalent integer constant is not allowed.

A.6 Functions

A function is a subprogram that is defined in one place and then called or invoked from one or more other places. Functions have parameter lists, enclosed in parentheses, which may contain formal parameters or may be empty. In C++, all functions return a value. The special type void is used for return values that are ignored by the calling program. Such functions are sometimes called procedures in other programming languages.

Defining functions

You define a function by giving its type, its name, and its formal parameters in a parameter list enclosed in parentheses `()`. The body of the function follows enclosed in braces `{}`. A `return` statement within the function causes control to return to the calling function with the value specified.


```
#include <cstdlib> // EXIT_SUCCESS.
#include <iostream> // cin, cout.
using namespace std;

int promptAge(); // Function prototype.

int main() { // Main program.
    int age = promptAge();
    if (age < 18) {
        cout << "Tax rate is 0%" << endl;
    } else if (age < 65) {
        cout << "Tax rate is 10%" << endl;
    } else {
        cout << "Tax rate is 5%" << endl;
    }
    return EXIT_SUCCESS;
}

int promptAge() { // Function definition.
    int i = 0;
    cout << "What is your age? ";
    cin >> i;
    while (i <= 0) {
        cout << "Must be greater than 0. Age? ";
        cin >> i;
    }
    return i;
}
```

Figure A.3 `FigureA3Main.cpp`. A C++ main program and a function that it calls.

Figure A.3 shows a function `promptAge` that prompts the user for her age and continually prompts until she enters an integer greater than zero. A complete C++ application consists of a main program that usually invokes subprograms that in turn invoke other subprograms.

Most C++ programs rely on a library of functions to perform input/output and other operations. The `#include` statements in Figure A.3 are necessary to access the various library files. The library file `cstdlib` defines constant `EXIT_SUCCESS` to be 0. The library file `iostream` provides the `cin` and `cout` classes.

Because C++ programs have access to many libraries there is a danger that an identifier in one library might be the same as an identifier in another library. C++ provides a namespace facility for distinguishing between duplicate identifiers from different libraries. A namespace is a collection of identifiers that enables you to make the distinction. `std` is a standard C++ namespace that includes the identifiers `cin` and `cout`. For example, suppose you include two libraries that both provide the identifier `cout`. To

use the `cout` identifier from the standard library, you would prefix `cout` with `std::`, as in

```
std::cout << "What is your age? ";
```

To use the `cout` from the other library you would prefix it with its namespace identifier. Without the line

```
using namespace std;
```

in Figure A.3, all occurrences of `cin` and `cout` would need to be prefixed with `std::`.

The main program is itself a function that returns `int`. The convention with C++ is for a main program to return zero to the operating system when it terminates normally and a nonzero integer when it terminates abnormally. All the main programs in this book assume normal termination and return the `EXIT_SUCCESS` constant. The parameter list of the main function in Figure ref is empty. There are times when the main program should have parameters in its parameter list, but this book will never have occasion to do so. The main program calls the function on the right side of the initialization statement

```
int age = promptAge();
```

There is a rule in C++ that says you must declare or define a function before you use it. Because of that rule, the definition of function `promptAge` must be placed physically before function `main`. Function prototypes allow you to place the definition of the function after the function that calls it.

A function prototype consists of the first line of the function including type, name, and parameter list, terminated by a semicolon. The body of the function including the outer braces is omitted. The information in the function prototype is enough for the compiler to determine how the function is to be used in the main program. In Figure A.3, if the function prototype were not present in the listing, the function definition would have to appear before the main program.

Parameter passing mechanisms

C++ provides three parameter passing mechanisms.

- Pass by value—The formal parameter gets the value of the actual parameter. If the function changes the formal parameter, the actual parameter does *not* change.
- Pass by reference—The formal parameter gets a reference to the actual parameter. If the function changes the formal parameter, the actual parameter *does* change.
- Pass by constant reference—The formal parameter gets a reference to the actual parameter, but the function is prevented from changing the parameter.

Each of these mechanisms serves a specific purpose. It is important to know when to use each one.

Pass by value is the default. That is, if you do not specify any other parameter passing mechanism, the parameter will be passed by value. Here is a function definition whose parameter is passed by value.

```
bool isEven(int num) {
    return num % 2 == 0;
}
```


The purpose of the function is to determine if the parameter is even. A main program could call the function with the statement

```
if (isEven(number))
```

where `number` is declared in the main program. `num` is the *formal parameter*, and `number` is the *actual parameter*.

The purpose of pass by value is to pass information from the calling function to the called function. The purpose is not to change the value of the actual parameter in the calling function. If you change the value of the formal parameter in the above function with a statement like

```
num = 5;
```

the actual parameter `number` will not change. The change is made to `num`, which received a copy of the value of `number` when the function was called.

You indicate pass by reference by placing the `&` symbol after the type in the parameter list. Here is a function definition whose parameters are passed by reference. It puts the two actual parameters in numeric order if they are out of order.

```
void putInOrder(int &i, int &j) {
    int temp = 0;
    if (i > j) {
        temp = i;
        i = j;
        j = temp;
    }
}
```

It might be called by the following statement in the main program.

```
putInOrder(first, second);
```

Because the parameters are passed by reference, the statement in the function

```
i = j;
```

has the effect of

```
first = second;
```

Formal parameter `i` refers to actual parameter `first`. So, when `i` changes in the called function, `first` changes in the calling function. Use call by reference when you want to change the value of the actual parameter in the calling function.

The general rule is that a parameter is passed by value unless the `&` operator appears after the type of the formal parameter. For arrays, however, because of their relationship to pointers as described in Chapter 2, the effect is as if they are always passed by reference even without the `&` operator. The following function clears an array to all zeros. In effect, the array `arr` is passed by reference, and the integer `cap` is passed by value.

```
void clearArray(int arr[], int cap) {
    int i = 0;
    while (i < cap) {
```



```
        arr[i++] = 0;
    }
}
```

Note that the postfix meaning of `i++` in function `clearArray` returns the value of `i` for the subscript, then increments. Hence, the function clears the first `cap` elements of `arr` from `arr[0]` to `arr[cap - 1]`.

If you do not intend to change the value of an array in a function, you should pass it by constant reference with the keyword `const`. For example, suppose you have an array of integers `arr` as well as a particular integer `searchNum`. You want to write a function that determines whether `searchNum` is in the array, and, if it is, what is the index of `arr` where it is located. You write a function named `search` that returns the index of where `searchNum` is located, or −1 if it is not in the array. The function could be declared as follows.

```
int search(const int arr[], int cap, int searchNum);
```

Use pass by constant reference if you want to pass an array, and you do not intend for the content of the array to change. In the above example, function `search` assumes that `arr` already contains values. Its purpose is not to change the values, so pass by constant reference is appropriate.

Program libraries

There are two kinds of libraries—the system library that is provided as part of the C++ language and is the same for all C++ development systems, and a programmer's library that is constructed by individual programmers and differs from one software project to another. This book comes with a set of software called the dp4ds distribution software, to which you should have access for study and for working exercises. The distribution software is your programmer's library. Figure A.4 shows the specification of the functions in this library and Figures A.5 and A.6 show its implementation.

A programmer specifies her library in a header file, whose name typically ends with `.hpp` or something similar. The name of the corresponding implementation file typically ends with `.cpp` or something similar. The name of a system library usually begins with `c` for the older C system library. The implementation of the system library is not usually available to programmers.

The `#include` line is a preprocessor directive at the beginning of the program listing. The statement

```
#include <iostream>  // cin, cout, endl, istream, ...
```

in the listing of Figure A.5 includes the input/output routines from the system library so the program can use `cin` and `cout`. System library files are enclosed between brackets < and >. Programmer library files are enclosed between double quotes ".

`#include` is a preprocessor directive that helps construct a stream of text for the C++ compiler to scan. When you give the name of a file to the directive, it is as if the text from that file is placed physically at the point of the `#include` directive. In Figure A.3, the effect is as if the text from file `iostream` were inserted for the compiler to scan just before the function prototype for `promptAge()`.


```
#ifndef UTILITIES_HPP_
#define UTILITIES_HPP_

#include <fstream> // ifstream.
#include <string> // string.
using namespace std;

const double PI = 3.1415926535898;

double promptDoubleGE(const string &prompt, double limit);
// Prompts the user with message prompt, requesting value >= limit.
// Continually prompts with error message when value input is not >= limit.
// Post: Double precision real >= limit is returned.

int promptIntGE(const string &prompt, int limit);

int promptIntBetween(const string &prompt, int lo, int hi);
// Prompts the user with message prompt, requesting value in lo..hi.
// Continually prompts with error message when value input is not in lo..hi.
// Post: Integer value in lo..hi is returned.

void promptFileOpen(ifstream &ifs);
// Prompts the user for a file name and opens it.

int sgn(int i);
// Post: If 0 <= i then 1 is returned, else -1 is returned.

int gcd(int m, int n);
// Pre: 0 <= m, n.
// Post: The greatest common divisor of m and n is returned.

int abs(int i);
// Post: The absolute value of i is returned.

#endif
```

Figure A.4 `Utilities.hpp`. Specification of the utilities library that comes with the dp4ds distribution software for this book.

Figure A.4 shows the header file for the utilities in the dp4ds distribution software. You use the utilities by including the file name `Utilities.hpp` enclosed in double quote marks in your program. The utilities file contains constants—such as the value of pi for mathematical computations—and function prototypes for several functions. The three statements

```
#ifndef Utilities_hpp
#define Utilities_hpp
```



```
#include <iostream>  // cin, cout, endl, istream, ostream, ifstream.
#include "Utilities.hpp"
using namespace std;

double promptDoubleGE(const string &prompt, double limit) {
    double d;
    cout << prompt << " (>= " << limit << "): ";
    cin >> d;
    while (d < limit) {
        cout << "Must be greater than or equal to " << limit << "." << endl;
        cout << prompt << "(>= " << limit << "): ";
        cin >> d;
    }
    return d;
}

int promptIntGE(const string &prompt, int limit) {
    int i;
    cout << prompt << " (>= " << limit << "): ";
    cin >> i;
    while (i < limit) {
        cout << "Must be greater than or equal to " << limit << "." << endl;
        cout << prompt << "(>= " << limit << "): ";
        cin >> i;
    }
    return i;
}

int promptIntBetween(const string &prompt, int lo, int hi) {
    int i;
    cout << prompt << " (" << lo << ".." << hi << "): ";
    cin >> i;
    while (i < lo || i > hi) {
        cout << "Must be between " << lo << " and " << hi << "." << endl;
        cout << prompt << " (" << lo << ".." << hi << "): ";
        cin >> i;
    }
    return i;
}
```

Figure A.5 `Utilities.cpp`. Implementation of the utility functions `promptDoubleGE()`, `promptIntGE()` and `promptIntBetween()` in the distribution software. The program listing continues in the next figure.


```
void promptFileOpen(ifstream &ifs) {
    string inFileName;
    cout << "File Name? ";
    cin >> inFileName;
    ifs.open(inFileName.c_str(), ios::in);
    if (!ifs) {
        cout << "Cannot open " << inFileName << '.' << endl;
    }
}

int sgn(int i) {
    return (0 <= i ? 1 : -1);
}

int gcd(int m, int n) {
    if (0 == n) {
        return m;
    } else {
        return gcd(n, m % n);
    }
}

int abs(int i) {
    return (0 <= i ? i : -i);
}
```

Figure A.6 `Utilities.cpp` (continued). Implementation of the utility functions `promptFileOpen()`, `sgn()`, `gcd()`, and `abs()` in the distribution software. This concludes the program listing.

and

```
#endif
```

are preprocessor directives. When you write your own header files you should always place similar preprocessor directives in your code. They are necessary because one application may use many library programs, that in turn use other library programs.

Figure A.7 shows why you need these directives. Suppose the program in file `A.cpp` includes `B.hpp` and `C.hpp`, then `B.hpp` and `C.hpp` each include `Utilities.hpp`. The effect is that `Utilities.hpp` would be scanned twice by the compiler. It would be as if you had duplicate definitions of the constants and of the function prototypes of Figure A.4. The second scan would generate a compiler error complaining of multiple definitions of the same identifier.

In Figure A.4, the preprocessor directive

```
#define UTILITIES_HPP_
```

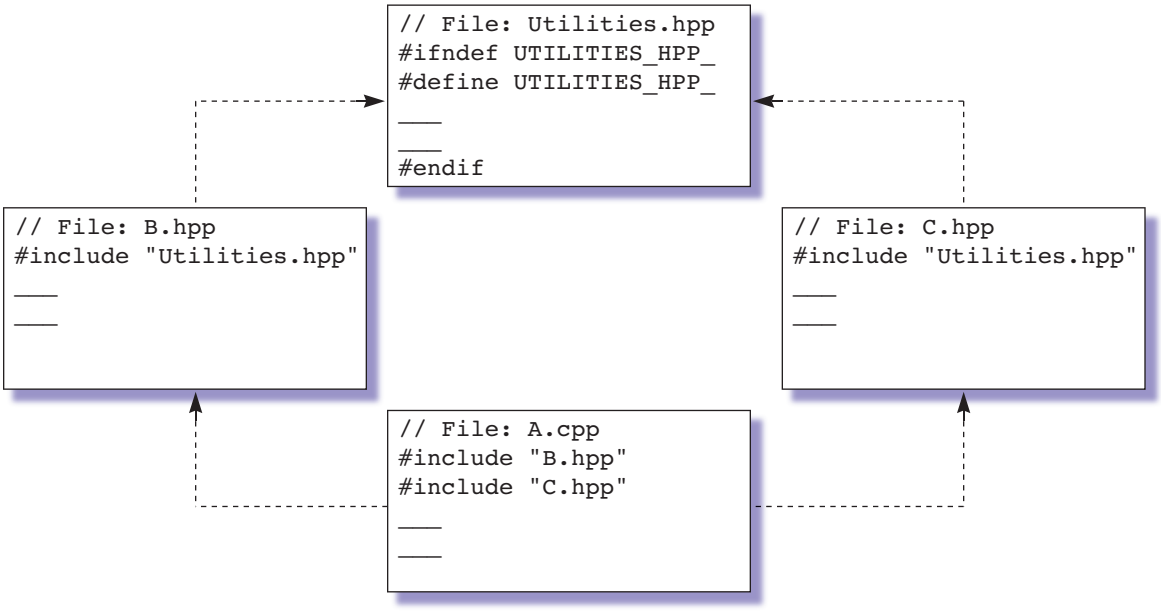



Figure A.7 A scenario to illustrate the necessity of compiler directives `#ifndef` and `#define`. The dashed arrows signify dependence by virtue of the `#include` statements.

defines the symbol `UTILITIES_HPP_`. Before it executes, the symbol is undefined. The compiler directive

```
#ifndef UTILITIES_HPP_
```

tests to see if the symbol has not been defined. If it has not been defined, the text up to `#endif` is scanned by the compiler. If it has been defined, the text up to the next `#endif` is skipped.

In Figure A.7, the compiler would begin by scanning `A.cpp`. The first `#include` in that file would send it to `B.hpp`, whose `#include` would send it to `Utilities.hpp`. The compiler would scan `#ifndef` and detect that the symbol `UTILITIES_HPP_` has not been defined. It would then scan `#define` and define the symbol. Then it would scan the constants and function prototypes, followed by the text in `B.hpp`. When it returns to scanning `A.cpp`, it hits the `#include` statement for `C.hpp`, which immediately sends it to scan `Utilities.hpp`. This time, when the compiler scans `#ifndef` in `Utilities.hpp` the symbol has been defined, so the text in the file is not rescanned, and the compiler error is avoided.

The identifier that you use for the symbol is somewhat arbitrary, but it must be unique among all the scans. This book follows a common convention of constructing the symbol from the name of the file in all uppercase letters with the period replaced by the underscore character _ and an additional underscore appended.

Figures A.5 and A.6 show the implementation of the utility functions. Functions `dPromptGE` and `promptIntGE` are general purpose prompt utilities that request the


```
#include <cstdlib> // EXIT_SUCCESS.
#include <iostream> // cin, cout.
#include "Utilities.hpp" // promptIntGE.
using namespace std;

int main() {
    int iAge = promptIntGE("What is your age? ", 0);
    if (iAge < 18) {
        cout << "Tax rate is 0%" << endl;
    } else if (iAge < 65) {
        cout << "Tax rate is 10%" << endl;
    } else {
        cout << "Tax rate is 5%" << endl;
    }
    return EXIT_SUCCESS;
}
```

Figure A.8 `FigureA8Main.cpp`. The program of Figure A.3 written with the use of the utility function `promptIntGE`.

user to enter values greater than some limit supplied by the calling function. Function `iPromptBetween` performs a similar service, but allows the calling function to supply both a lower and upper limit on the response. These functions use a prompt string supplied by the calling function.

Function `promptFileOpen` prompts the user for the name of a file. The type `ifstream` is an input file stream. The function prompts for the name of a file, then attempts to open the file so data values can be read from it. This book is not concerned with the details of file input/output. You can use the file routines supplied with the distribution software without necessarily understanding exactly how they work.

Figure A.8 shows a main program that uses the `promptIntGE` function from `Utilities.hpp`. You can see that it is much shorter than the program in Figure A.3 that does essentially the same thing.

The string library

The C++ `string` library alleviates most of the problems of character arrays described in the previous section. To use the `string` library, include `<string>` at the beginning of your program. The `string` library is not the same as the `cstring` library, which provides functions for manipulating C-style null-terminated arrays of characters. The `string` library provides the string type `string`, which is more than just a `typedef`. It is an example of a class, as described in the next section. You use it the same way you use a type, except for how you call functions.

You declare a `string` the same way you declare a primitive variable. The declarations

```
string myString;
string yourString("Hello world.");
```


create `myString`, which is empty, and `yourString`, which is initialized to a string value. Another way to give a value to a string is to input it from the input stream using the same input operator as for primitive types. The statement

```
cin >> myString;
```

inputs the next group of characters from the standard input stream into `myString`. Similarly

```
cout << yourString;
```

outputs the string to the standard output stream.

If you want to know how many characters your string contains, call the `size()` function. Assuming `yourString` is initialized as above, the statement

```
cout << yourString.size();
```

outputs the value 12, the number of characters in the string. You can access an individual character of type `char` by subscripting the string using the usual array notation with the first element at position 0. For example, if the value of integer `i` is six, the expression

```
yourString[i]
```

has the value `'w'`. Accessing a string element this way is not safe. The compiler does not prevent you from accessing a location outside the string's boundaries. A safe access of the same location is the expression

```
yourString.at(i)
```

The `at()` function checks `i` to make sure it is within the string's boundaries before accessing the location and throws an exception if it is not. Section 1.2 describes the concept of an exception in C++. The usual effect is for the program to emit an error message and terminate.

One major advantage of the `string` class over the `cstring` class is the ability to assign one string to another string with the = operator instead of the `strncpy` function. To assign the value of `yourString` to `myString` simply write

```
myString = yourString;
```

The assignment copies the string value from `yourString` character-by-character in a loop behind the scenes. It does this by overloading the assignment operator, a technique described in the next section.

Another major advantage is the ability to use the relation operators, `==`, `!=`, `<`, `<=`, `>`, and `>=` to compare two strings instead of using the awkward `strcmp` function. The statement

```
if (myString < yourString)
```

is a legal and meaningful alphabetic comparison.

The `Utilities` library in the dp4ds distribution uses the `string` library for the user prompts. Some other projects in the software distribution use the features of the `string` class described above. Many other facilities are provided in the `string` library including those for converting between the string type and the older C-style strings, for concatenating two strings, and for searching for a substring within a string.

A.7 Classes

C++ provides a programming construct called a *class*, which is a set of related data and a set of functions called *member functions* that directly manipulate the data. The effect of defining a class is to create a new, programmer-defined type that is not supplied as a primitive type by the programming language. A primitive type defines a set of values that a variable of that type can have. For example, if variable `i` has type `int`, the set of values that `i` can have include $2, 1, 0, 1, 2$, and so on. These values define the type `int`. Analogously, a class serves as a blueprint for a set of values that an object can have. An object is to a class as a variable is to a primitive type.

Suppose you need a program to process rational numbers. A rational number is a number of the form n/d, where n and d are integers. For example, $3/7$ is a rational number, but $\sqrt{2}$ is not because it cannot be written as the ratio of two integers. You would like to manipulate two rational numbers using the rules of algebra. For example, your program should be able to add $3/4$ and $2/3$ to get $17/12$, and it should be able to multiply them to get $1/2$. The problem is that C++ provides primitive numeric types such as `int` for integers and `double` for reals, but it does not provide something like `rat` for rational numbers. You can create a programmer-defined type called `Rational` by encapsulating the pair of integers `_numerator` and `_denominator` and the functions that manipulate them into a class.

Figure A.9 shows the input and output of a program that processes rational numbers. There is a main prompt that asks the user which action is to be performed. The program uses a class named `Rational` that is a programmer-defined type. It declares an array of five rationals, which the user accesses via the main prompt. The main program provides the user with the option of creating and manipulating up to five rational numbers. The session shows how a user would enter rational numbers $3/4$ and $2/3$, add them, multiply them, and output them.

The remainder of this section describes the C++ program that produces the user interaction of Figure A.9. Four files contain the source code:

- `Rational.hpp` A header file that specifies a rational number class
- `Rational.cpp` The implementation of the class
- `RationalMain.hpp` A header file that specifies the main program
- `RationalMain.cpp` The implementation of the main program

Class specification

Figure A.10 shows the specification of class `Rational`. In general, a class contains *attributes*, which are the data stored by the class, and *operations*, which are the functions that operate on the data. In class `Rational`, the attributes are the integers `_numerator` and `_denominator`, and the operations are the two functions named `Rational()`, the function named `toDouble()`, and the two functions that return an integer, `numerator()` and `denominator()`. The attributes are in the private part of the class and the operations are in the public part. Items that are private cannot be accessed directly by the user of the class, but items that are public can be accessed directly.

Specification of function numerator

```
int numerator() const;
```



```
There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: m
Which rational? (0..4): 0
Enter a rational (a or a/b): 3/4

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: m
Which rational? (0..4): 1
Enter a rational (a or a/b): 2/3

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: a
Left operand? (0..4): 0
Right operand? (0..4): 1
Result operand? (0..4): 2

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: u
Left operand? (0..4): 0
Right operand? (0..4): 1
Result operand? (0..4): 3

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: p
Which rational? (0..4): 2
17/12

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: p
Which rational? (0..4): 3
1/2

There are [0..4] rationals.
(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   (d)ivide
(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  (q)uit: q
```

Figure A.9 Input/output of a program to process rational numbers. User input is bold.


```
#ifndef Rational_hpp
#define Rational_hpp

#include <iostream> // istream, ostream.
using namespace std;

class Rational {
private:
    int _numerator;
    int _denominator;
    // Invariant: 0 < _denominator.
    // Invariant: gcd(_numerator, _denominator) == 1.

public:
    Rational(int numerator = 0, int denominator = 1);
    // Post: This Rational is numerator/denominator.

    Rational(const Rational &rhs);
    // Post: This Rational is a copy of rhs.

    double toDouble() const;
    // Post: The double value of this Rational is returned.

    int numerator() const;
    int denominator() const;
};

// Overloaded binary algebraic operators.
Rational operator+(const Rational &lhs, const Rational &rhs);
Rational operator-(const Rational &lhs, const Rational &rhs);
Rational operator*(const Rational &lhs, const Rational &rhs);
Rational operator/(const Rational &lhs, const Rational &rhs);
// Pre: rhs != 0.

// Overloaded unary negation operator.
Rational operator-(const Rational &rhs);

// Overloaded comparison operators.
bool operator==(const Rational &lhs, const Rational &rhs);
bool operator!=(const Rational &lhs, const Rational &rhs);
bool operator<(const Rational &lhs, const Rational &rhs);
bool operator>(const Rational &lhs, const Rational &rhs);

// Overloaded input and output operators.
ostream &operator<<(ostream &os, const Rational &rhs);
istream &operator>>(istream &is, Rational &rhs);
// Pre: The next input in is has the form a or a/b, where a, b are int.

#endif
```

Figure A.10 `Rational.hpp`. The specification for class `Rational`.

contains `const`, which makes it a *constant member function*. Constant member functions cannot modify the attributes of the class. The compiler prevents such modification, and the word `const` serves as documentation to that effect. If you ever declare an object to be a constant, it can only call constant member functions. For example, if `MyClass` is a class and you declare

```
const MyClass myObject
```

then `myObject` can execute only the constant member functions of `MyClass`.

The terminology "member function" for an operation that is part of a class is specific to C++. In general object-oriented terminology, methods are known as *operations*. This book frequently uses general object-oriented terminology, but you should be aware that some C++ documentation uses terminology specific only to C++.

Overloaded function names

When you write a program and the functions it will call, it is usually a good idea to give each function a unique name that indicates the task that it does for the calling program. For example, you may wish to write a function that prints the value of a dollar amount with two places past the decimal point to indicate dollars and cents. To do this you could write a function named `printReal()` with three parameters, the first a double precision real for the value to be printed, the second an integer for the field width, and the third an integer for the number of places to print past the decimal point.

For example, suppose you have the following variable declarations in your main program.

```
double accountBalance;
int numberOfEntries;
```

To print the account balance you would make the function call

```
printReal(accountBalance, 12, 2);
```

But now suppose you want to print the number of entries with a field width of eight. Because `numberOfEntries` is integer, you would need to give your second function another name, like `printInteger()`, with two parameters, the first an integer for the value to be printed and the second the field width. To print the number of entries you would make the function call

```
printInteger(numberOfEntries, 8);
```

Rather than making up a different name for each function, you can use overloading and name both functions `print()`. Then, to print the account balance you would call

```
print(accountBalance, 12, 2);
```

and to print the number of entries you would call

```
print(numberOfEntries, 8);
```


How does C++ know which print function to call? By looking at the number and type of the actual parameters. Because the first function call has one real parameter followed by two integer parameters, the compiler looks for a function definition named `print()` that has three formal parameters, a real followed by two integers. Similarly, for the second call it looks for a function named `print()` that has two formal parameters, both integers.

The signature of a function consists of

- the number of its formal parameters,
- the types of its formal parameters, and
- whether the function is a const function.

The signature does not include the type returned by the function. You can define two functions with the same name as long as the functions have different signatures. It is illegal to define two functions with the same name and the same signatures.

You can carry function name overloading one step further by applying it to operators. Consider the addition operator `+`. To add two integers you write an expression like `3+4` with `+` used as a binary infix operator. That is, the `+` symbol is placed between the operands `3` and `4`. However, you can consider addition to be a function of two integer variables that returns their sum. If it were legal to have symbols other than letters, digits and _ characters in identifiers, such a function could be declared as

```
int operator+(int lhs, int rhs)
```

where `lhs` stands for the left hand side of the operator and `rhs` stands for the right hand side. To add 3 and 4 you would call `operator+(3,4)`.

C++ allows you to overload not only function names, but operator names as well. The program that processes rational numbers with the Rational class overloads the binary arithmetic operators `+`, `-`, `*`, `/`, the unary negation operator `-`, the boolean comparison operators `==` and `<`, and the shift operators `<<` and `>>` that are already overloaded for input and output. Operators have names that begin with the reserved word `operator` followed by the operator symbol. For example, the `+` operator has the equivalent function name `operator+`, which is the name that is overloaded.

This overloading makes it convenient to process rational numbers. Suppose you declare

```
Rational myRational, yourRational, herRational;
```

and you want to give `herRational` the sum of `myRational` and `yourRational`. You can do it with the assignment statement

```
herRational = myRational + yourRational;
```

Because `+` is a binary operator, you can continue to use it as an infix operator as you would when you add integers or reals.

Constructors

A *constructor* is a special function whose purpose is to initialize the attributes of an object. A constructor is special because

- its name is the same as the name of the class, and
- it has no return type.

Because the name of this class is `Rational`, the two functions also named `Rational` are constructors for the class. Their names are overloaded. The first one has two parameters in its parameter list while the second one has only one parameter. The difference in the signatures is how the compiler distinguishes between the two constructors.

Constructors behave like ordinary functions except in the way that they are called. Ordinary functions must be called explicitly, while constructors can be called implicitly or explicitly when storage is allocated for the object. For example, in Figure A.3 the function `promptAge` is called explicitly by the statement

```
int age = promptAge();
```

On the other hand, suppose you have the declarations

```
int myInteger;
Rational myRational(3, 4);
Rational yourRational(3);
Rational herRational;
```

when using the `Rational` class of Figure A.10. The declaration of the integer does not cause a function to be called, but the declarations of the rational numbers cause the first constructor

```
Rational(int numerator = 0, int denominator = 1);
```

to be called. The assignments of the formal parameters indicate *default parameter* values that are given when the corresponding actual parameter is missing.

The declaration for `myRational` has 3 and 4 as its actual parameters. This declaration calls the constructor and gives 3 to formal parameter `numerator` and 4 to formal parameter `denominator`. The declaration for `yourRational` has 3 as its first actual parameter and is missing its second parameter. This declaration calls the constructor and gives 3 to formal parameter `numerator` and 1 to `denominator`. Because the declaration for `herRational` is missing both actual parameters, the constructor is called with 0 assigned to `numerator` and 1 to `denominator`.

This constructor is also called implicitly to do a type conversion when you assign an integer to a rational. When the statement

```
herRational = 5;
```

executes, the compiler recognizes `herRational` from the rational class and 5 as an integer. It generates a call to the constructor giving 5 to numerator and the default value of 1 to denominator.

The constructor

```
Rational(const Rational& rhs);
```

is a special kind of constructor known as a *copy constructor*. A copy constructor is a constructor that has one formal parameter of the same type as the class and is called by constant reference. The name of the formal parameter for this copy constructor is `rhs`, which stands for right hand side. The purpose of the copy constructor is to initialize an

object whenever a copy of the object is required. The copy operation is necessary in two situations—when an object is passed by value as a parameter in a function's parameter list, and when an object is used as the returned value of a function. An example of when a copy constructor is called under the second condition is execution of the statement

```
myRational = yourRational + herRational;
```

Using + as an infix operator is equivalent to calling the function `operator+` shown in Figure A.10. That function returns a value of type `Rational`. After the function computes the sum of the two rational numbers it makes a copy of the result using the copy constructor, which `myRational` gets from the assignment.

Class implementation

Figures A.11 and A.12 show the implementation of class `Rational`. Many of the methods are incomplete with the actual implementation left as an exercise for the student. Execution of the statement

```
throw -1;
```

in the constructor for `Rational` causes the program to abort. Section 1.2 describes the `throw` statement in more detail. The constructor uses it here to implement the precondition that the denominator cannot be zero. Some functions are left as exercises for the student to complete. Remove the error message and `throw` statement when you implement a function.

The function heading of each method contains the `::` operator, which separates the method name from the name of the class of which it is a member. For example, the function heading

```
int Rational::numerator() const
```

declares that function `numerator()` is a member of class `Rational`. The corresponding declaration in the header file of Figure A.10

```
int numerator() const;
```

does not require the class name with the `::` separator, because the function declaration is placed within the class as follows.

```
class Rational {
    ...
    int numerator() const; ...
};
```

The rational numbers are all stored in lowest form with a positive denominator. For example, if you declare

```
Rational myRational(10, -15);
```

then `myRational` should be stored with a value of -2 for `_numerator` and 3 for `_denominator`, because the rational number $10/-15$ in lowest form is $-2/3$. To ensure that rational numbers are in lowest form, the constructor


```
#include "Rational.hpp"
#include "Utilities.hpp" // abs, sgn, gcd.

Rational::Rational(int numerator, int denominator) {
    if (0 == denominator) {
        cerr << "Rational precondition violated: 0 == denominator" << endl;
        throw -1;
    }
    _numerator = abs(numerator);
    _denominator = abs(denominator);
    int gcdTemp = gcd(_numerator, _denominator);
    _numerator = sgn(numerator) * sgn(denominator) * _numerator / gcdTemp;
    _denominator = _denominator / gcdTemp;
}

Rational::Rational(const Rational &rhs) {
    _numerator = rhs._numerator;
    _denominator = rhs._denominator;
}

double Rational::toDouble() const {
    return static_cast<double> (_numerator) / static_cast<double> (_denominator);
}

int Rational::numerator() const {
    return _numerator;
}

int Rational::denominator() const {
    return _denominator;
}

Rational operator+(const Rational &lhs, const Rational &rhs) {
    int gcdTemp = gcd(lhs.denominator(), rhs.denominator());
    int lhsMul = rhs.denominator() / gcdTemp;
    int rhsMul = lhs.denominator() / gcdTemp;
    return Rational(lhs.numerator() * lhsMul + rhs.numerator() * rhsMul,
        lhs.denominator() * lhsMul);
}

Rational operator-(const Rational &lhs, const Rational &rhs) {
    cerr << "operator- (binary): Exercise for the student." << endl;
    throw -1;
}
```

Figure A.11 `Rational.cpp`. Implementation of class `Rational`. The listing continues in the next figure.


```
Rational operator*(const Rational &lhs, const Rational &rhs) {
    cerr << "operator*: Exercise for the student." << endl;
    throw -1;
}

Rational operator/(const Rational &lhs, const Rational &rhs) {
    cerr << "operator/: Exercise for the student." << endl;
    throw -1;
}

Rational operator-(const Rational &rhs) {
    cerr << "operator- (unary): Exercise for the student." << endl;
    throw -1;
}

bool operator==(const Rational &lhs, const Rational &rhs) {
    cerr << "operator==: Exercise for the student." << endl;
    throw -1;
}

bool operator!=(const Rational &lhs, const Rational &rhs) {
    return !(lhs == rhs);
}

bool operator<(const Rational &lhs, const Rational &rhs) {
    cerr << "operator<: Exercise for the student." << endl;
    throw -1;
}

bool operator>(const Rational &lhs, const Rational &rhs) {
    return !(lhs == rhs) && !(lhs < rhs);
}

ostream &operator<<(ostream &os, const Rational &rhs) {
    cerr << "operator<<: Exercise for the student." << endl;
    throw -1;
}

istream &operator>>(istream &is, Rational &rhs) {
    int numerator, denominator = 1;
    is >> numerator;
    char c = (char) is.peek();
    if (c == '/') {
        is >> c >> denominator;
    }
    rhs = Rational(numerator, denominator);
    return is;
}
```

Figure A.12 `Rational.cpp` (continued). Implementation of class `Rational`. This completes the listing.


```
Rational::Rational(int numerator, int denominator)
```

first stores the absolute values of the formal parameters into the attributes of `myRational`. Then it calls a utility function `gcd()` that computes the greatest common divisor with the statement

```
int gcdTemp = gcd(_numerator, _denominator);
```

In this example, local variable `gcdTemp` gets the greatest common divisor of 10 and 15, which is 5. The constructor divides the numerator and denominator by the greatest common divisor and adjusts the signs to get the rational number in lowest form.

Figure A.6 on page 475 shows the implementation of the greatest common divisor function `gcd()`. It is based on the recursive definition

$$\gcd(m,n) = \begin{cases} m, & \text{for } n = 0; \\ \gcd(n, m \bmod n), & \text{for } n > 0. \end{cases}$$

For example, the greatest common divisor of 10 and 15 is computed recursively as

$$\begin{aligned} \gcd(10,15) &= \gcd(15, 10 \bmod 15) &&= \gcd(15,10) \\ &= \gcd(10, 15 \bmod 10) &&= \gcd(10,5) \\ &= \gcd(5, 10 \bmod 5) &&= \gcd(5,0) \\ &= 5 \end{aligned}$$

The second constructor assumes that all rational numbers are in lowest form. For example, if you declare

```
Rational yourRational(myRational);
```

the constructor

```
Rational::Rational(const Rational& rhs)
```

which is called implicitly, simply copies the numerator and denominator attributes from `myRational` to `yourRational` without verifying that their greatest common divisor is 1.

The implementation of the + operator illustrates a common problem that algorithms processing integer values must address. The problem is that a computation may require intermediate integer values that overflow the range of allowed values. For example, suppose you want to add $27/20$ and $35/12$. A straightforward algorithm would compute the common denominator as the product of the denominators.

$$\frac{27}{20} + \frac{35}{12} = \frac{27 \cdot 12}{20 \cdot 12} + \frac{35 \cdot 20}{12 \cdot 20} = \frac{27 \cdot 12 + 35 \cdot 20}{20 \cdot 12} = \frac{324 + 700}{240} = \frac{1024}{240} = \frac{64}{15}$$

However, a better algorithm would compute the least common denominator as follows. The greatest common divisor of 20 and 12 is 4. So, the multiplier for $27/20$ is $12/4 = 3$, and the multiplier for $35/12$ is $20/4 = 5$. The least common denominator is 60, and the sum is computed as

$$\frac{27}{20} + \frac{35}{12} = \frac{27 \cdot 3}{20 \cdot 3} + \frac{35 \cdot 5}{12 \cdot 5} = \frac{27 \cdot 3 + 35 \cdot 5}{20 \cdot 3} = \frac{81 + 175}{60} = \frac{256}{60} = \frac{64}{15}$$

The largest intermediate value for the first algorithm is 1024, but the largest intermediate value for the second algorithm is only 256. With these small numbers, even the larger intermediate value would not be outside the range of a C++ `int`. But for many rational numbers, the first algorithm produces an intermediate computation that overflows, while the second algorithm does not.

The + operator in Figure A.11 implements the second algorithm. Local variable `gcdTemp` corresponds to the greatest common divisor of 20 and 12 in the above scenario. The implementation uses the first constructor to get the sum in lowest form. For example, suppose `yourRational` has value 27/20 and `herRational` has value 35/12. If you execute the statement

```
myRational = yourRational + herRational;
```

then the function

```
Rational operator+(const Rational &lhs, const Rational &rhs)
```

executes. It returns a value that it declares with the first constructor. It computes the numerator as 256 and the denominator as 60. Rather than declaring a local variable to store the sum and giving it a name, the function simply declares the returned value on the fly with the C++ return statement.

The `toDouble()` method uses the new-style type cast described earlier. The expression

```
static_cast<double>(_numerator)
```

converts integer `_numerator` to a double precision real so that the `/` operator will perform real division instead of integer division with truncation.

The input operator >> is overloaded so you can input rational values into variables the same way you input values into variables of primitive types. For example, you can prompt the user for a value then execute

```
cin >> myRational;
```

If the user enters `10/-15`, then `myRational` gets −2 for the numerator and 3 for the denominator. `cin` is an object whose class is `istream`, which stands for input stream. When the above statement executes, formal parameter `is` corresponds to actual parameter `cin`. The function peeks ahead in the input stream to see if the user entered the `/` character. If `/` is entered, the function inputs the denominator. Otherwise the denominator maintains its initial value of 1.

Using classes

Figure A.13 shows the specification of the main program and Figures A.14, A.15, and A.16 show its implementation. The main program declares an array of five rational numbers with

```
const int CAP = 5;
Rational rats[CAP];
```

then calls the loop with


```
#ifndef RationalMain_hpp
#define RationalMain_hpp

class Rational;

void promptLoop(Rational r[], int cap);
// Loop to prompt the user with the top-level main prompt.
// Post: User has selected the quit option.

void makeRational(Rational &r);
// Prompts user for a rational in the form a or a/b.
// Post: r is assigned the prompted rational.

void clearRational(Rational &r);
// Post: r is the rational 0.

void addRational(Rational r[], int cap);
void subtractRational(Rational r[], int cap);
void multiplyRational(Rational r[], int cap);
void divideRational(Rational r[], int cap);
void negateRational(Rational &r);
void displayRational(Rational &r);
void convertToDecimal(Rational &r);
void equalRational(Rational r[], int cap);
void lessThanRational(Rational r[], int cap);

#endif
```

Figure A.13 `RationalMain.hpp`. The header file for the main program that produces the input/output of Figure A.9.

```
promptLoop(rats, CAP);
```

to prompt the user for the options. Each option has a corresponding function that is executed depending on the option chosen by the user.

Constructors are used throughout for returning results. For example, to clear a rational number to zero, the function

```
void clearRational(Rational &r) {
    r = 0;
}
```

simply sets `r` to 0. The compiler knows that `r` has type `Rational`, and zero has type `int`. Because of the type mismatch, it looks for a constructor for the `Rational` class and uses

```
Rational(int numerator = 0, int denominator = 1)
```

to do the conversion. That is, the statement


```
#include <iostream>
#include <cstdlib> // EXIT_SUCCESS
#include <cctype> // toupper.
#include "Rational.hpp"
#include "Utilities.hpp" // promptIntBetween, promptDoubleGE.
#include "RationalMain.hpp"
using namespace std;

int main() {
    const int CAP = 5;
    Rational rats[CAP];
    promptLoop(rats, CAP);
    return EXIT_SUCCESS;
}

void promptLoop(Rational r[], int cap) {
    char response;
    do {
        cout << "\nThere are [0.." << cap - 1 << "] rationals." << endl;
        cout << "(m)ake    (c)lear    (a)dd      (s)ubtract  m(u)ltiply   "
             << "(d)ivide\n"
             << "(n)egate  dis(p)lay  dec(i)mal  (e)quals    (l)ess than  "
             << "(q)uit: ";
        cin >> response;
        switch (toupper(response)) {
        case 'M':
            makeRational(r[promptIntBetween("Which rational?", 0, cap - 1)]);
            break;
        case 'C':
            clearRational(r[promptIntBetween("Which rational?", 0, cap - 1)]);
            break;
        case 'A':
            addRational(r, cap);
            break;
        case 'S':
            subtractRational(r, cap);
            break;
        case 'U':
            multiplyRational(r, cap);
            break;
        case 'D':
            divideRational(r, cap);
            break;
        case 'N':
            negateRational(r[promptIntBetween("Which rational?", 0, cap - 1)]);
            break;
```

Figure A.14 `RationalMain.cpp`. Implementation of the main program for the header file of Figure A.13. The program listing continues in the next figure.


```
        case 'P':
            cout << "\n";
            displayRational(r[promptIntBetween("Which rational?", 0, cap - 1)]);
            break;
        case 'I':
            convertToDecimal(r[promptIntBetween("Which rational?", 0, cap - 1)]);
            break;
        case 'E':
            equalRational(r, cap);
            break;
        case 'L':
            lessThanRational(r, cap);
            break;
        case 'Q':
            break;
        default:
            cout << "\nIllegal command." << endl;
            break;
        }
    } while (toupper(response) != 'Q');
}

void makeRational(Rational &r) {
    cout << "Enter a rational(a or a/b): ";
    cin >> r;
}

void clearRational(Rational &r) {
    r = 0;
}

void addRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    int result = promptIntBetween("Result operand?", 0, cap - 1);
    r[result] = r[left] + r[right];
}

void subtractRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    int result = promptIntBetween("Result operand?", 0, cap - 1);
    r[result] = r[left] - r[right];
}
```

Figure A.15 `RationalMain.cpp` (continued). Implementation of the main program for the header file of Figure A.13. The program listing continues in the next figure.


```
void multiplyRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    int result = promptIntBetween("Result operand?", 0, cap - 1);
    r[result] = r[left] * r[right];
}

void divideRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    int result = promptIntBetween("Result operand?", 0, cap - 1);
    r[result] = r[left] / r[right];
}

void negateRational(Rational &r) {
    r = -r;
}

void displayRational(Rational &r) {
    cout << "\n" << r << endl;
}

void convertToDecimal(Rational &r) {
    cout << "\n" << r.toDouble() << endl;
}

void equalRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    cout << "\n"<< r[left] << " == " << r[right] << ": "
         << (r[left] == r[right] ? "true" : "false") << endl;
}

void lessThanRational(Rational r[], int cap) {
    int left = promptIntBetween("Left operand?", 0, cap - 1);
    int right = promptIntBetween("Right operand?", 0, cap - 1);
    cout << "\n" << r[left] << " < " << r[right] << ": "
         << (r[left] < r[right] ? "true" : "false") << endl;
}
```

Figure A.16 `RationalMain.cpp` (continued). Implementation of the main program for the header file of Figure A.13. This concludes the program listing.


```
r = 0;
```

means

```
r = Rational(0);
```

which has the effect of calling the constructor with the default value of 1 for denominator.

Exercises

This book comes with a set of software called the "dp4ds distribution," to which you should have access for study and for working exercises. Exercises at the end of the chapter ask you to implement or modify parts of the distribution software. The distribution software is available at

```
http://www.cslab.pepperdine.edu/warford/cosc320/
```

1–1 Suppose you have determined that `char` variable `ch` has a value between `'0'` and `'9'`. If `i` is a variable of type `int`, write a C++ statement that assigns the numeric value of the corresponding character value of `ch` to `i`. For example, if `ch` has the character value `'3'` then `i` should get the integer value 3. Do not use any magic numbers.

1–2 Suppose `animal` is declared as in Section A.4 and you execute the following C++ statements.

```
strncpy(sAnimal, "abcdefg", 8);
strncpy(sAnimal, "wxyz", 4);
```

Draw the content of the eight cells of `animal` as they are drawn in Figure A.2(b). What would be the result of executing

```
cout << sAnimal << endl;
```

1–3 In function `clearArray` in Section A.6, explain what would happen if the last statement were changed to use the prefix version of the increment operator `arr[++i]=0]`.

1–4 Write a program that asks the user to input two numbers between one and 90 using `promptIntBetween` from `Utilities.hpp`. Then, print a six by six multiplication table starting with the numbers input. Format the table exactly as shown below with the numbers aligned properly in straight columns for all possible values input by the user. The table is for an input of 3 and 25.

```
   |    3    4    5    6    7    8
---|-----------------------------
25 |   75  100  125  150  175  200
26 |   78  104  130  156  182  208
27 |   81  108  135  162  189  216
28 |   84  112  140  168  196  224
29 |   87  116  145  174  203  232
30 |   90  120  150  180  210  240
```


1–5 Write a program that prompts the user for two strings with a maximum length of 63 characters. Output them in alphabetic order using `strcmp` from the `cstring` library.

1–6 Complete the implementation of the following member functions and operators for the `Rational` class in Figures A.11 and A.12.

(a) `operator<<`. Display the value $6/1$ as `6`. Do this one first so you can see the results of the rest of this exercise.

(b) Unary `operator-`.

(c) Binary `operator-`. To subtract, simply add the negation of the right hand side to the left hand side. Negate the right hand side with the unary operator of part (b).

(d) `operator*`. Do not implement the multiplication operator with a simple cross multiplication, as the intermediate products can be so large as to overflow unnecessarily. Instead, compute the gcd of the cross product terms and divide before multiplying. For example, to multiply $20/21$ by $35/8$, do not simply multiply the numerators as $20 \cdot 35 = 700$ and the denominators as $21 \cdot 8 = 168$ both of which are unnecessarily large. The gcd of 20 and 8 is 4, and the gcd of 35 and 21 is 7. Therefore, the numerator of the product should be computed as $5 \cdot 5 = 25$ and the denominator as $3 \cdot 2 = 6$. Be sure to take into account the possibility of negative rational numbers.

(e) `operator/`. Figure A.10 shows that the operator has a precondition, which you should implement. Return the product of the left hand side with a rational number that is the inverse of the right hand side, using the binary operator of part (d) for the product.

(f) `operator==`. Do not use method `toDouble()` or any equivalent idea. In other words, do all computations with integers.

(g) `operator<`. Use method `toDouble()`.

