Thinking in C++, 2nd ed. Volume 1

3: The C in C++

Since C++ is based on C, you must be familiar with the syntax of C in order to program in C++, just as you
must be reasonably fluent in algebra in order to tackle calculus.

If you've never seen C before, this chapter will give you a decent background in the style of C used in C++. If you are familiar with the style of C described in the first edition of Kernighan & Ritchie (often called K&R C), you will find some new and different features in C++ as well as in Standard C. If you are familiar with Standard C, you should skim through this chapter looking for features that are particular to C++. Note that there are some fundamental C++ features introduced here, which are basic ideas that are akin to the features in C or often modifications to the way that C does things. The more sophisticated C++ features will not be introduced until later chapters.

This chapter is a fairly fast coverage of C constructs and introduction to some basic C++ constructs, with the understanding that you've had some experience programming in another language. A more gentle introduction to C is found in the CD ROM packaged in the back of this book, titled *Thinking in C: Foundations for Java & C++* by Chuck Allison (published by MindView, Inc., and also available at www.MindView.net). This is a seminar on a CD ROM with the goal of taking you carefully through the fundamentals of the C language. It focuses on the knowledge necessary for you to be able to move on to the C++ or Java languages rather than trying to make you an expert in all the dark corners of C (one of the reasons for using a higher-level language like C++ or Java is precisely so we can avoid many of these dark corners). It also contains exercises and guided solutions. Keep in mind that because this chapter goes beyond the *Thinking in C* CD, the CD is not a replacement for this chapter, but should be used instead as a preparation for this chapter and for the book.

Creating functions

In old (pre-Standard) C, you could call a function with any number or type of arguments and the compiler wouldn't complain. Everything seemed fine until you ran the program. You got mysterious results (or worse, the program crashed) with no hints as to why. The lack of help with argument passing and the enigmatic bugs that resulted is probably one reason why C was dubbed a "high-level assembly language." Pre-Standard C programmers just adapted to it.

Standard C and C++ use a feature called *function prototyping*. With function prototyping, you must use a description of the types of arguments when declaring and defining a function. This description is the "prototype." When the function is called, the compiler uses the prototype to ensure that the proper arguments are passed in and that the return value is treated correctly. If the programmer makes a mistake when calling the function, the compiler catches the mistake.

Essentially, you learned about function prototyping (without naming it as such) in the previous chapter, since the form of function declaration in C++ requires proper prototyping. In a function prototype, the argument list contains the types of arguments that must be passed to the function and (optionally for the declaration) identifiers for the arguments. The order and type of the arguments must match in the declaration, definition, and function call. Here's an example of a function prototype in a declaration:

```
int translate(float x, float y, float z);
```

You do not use the same form when declaring variables in function prototypes as you do in ordinary variable definitions. That is, you cannot say: **float x, y, z**. You must indicate the type of *each* argument. In a function declaration, the following form is also acceptable:

```
int translate(float, float, float);
```

Since the compiler doesn't do anything but check for types when the function is called, the identifiers are only included for clarity when someone is reading the code.

In the function definition, names are required because the arguments are referenced inside the function:

```
int translate(float x, float y, float z) {
  x = y = z;
  // ...
}
```

It turns out this rule applies only to C. In C++, an argument may be unnamed in the argument list of the function definition. Since it is unnamed, you cannot use it in the function body, of course. Unnamed arguments are allowed to give the programmer a way to "reserve space in the argument list." Whoever uses the function must still call the function with the proper arguments. However, the person creating the function can then use the argument in the future without forcing modification of code that calls the function. This option of ignoring an argument in the list is also possible if you leave the name in, but you will get an annoying warning message about the value being unused every time you compile the function. The warning is eliminated if you remove the name.

C and C++ have two other ways to declare an argument list. If you have an empty argument list, you can declare it as **func( )** in C++, which tells the compiler there are exactly zero arguments. You should be aware that this only means an empty argument list in C++. In C it means "an indeterminate number of arguments (which is a "hole" in C since it disables type checking in that case). In both C and C++, the declaration **func(void);** means an empty argument list. The **void** keyword means "nothing" in this case (it can also mean "no type" in the case of pointers, as you'll see later in this chapter).

The other option for argument lists occurs when you don't know how many arguments or what type of arguments you will have; this is called a *variable argument list*. This "uncertain argument list" is represented by ellipses (**...**). Defining a function with a variable argument list is significantly more complicated than defining a regular function. You can use a variable argument list for a function that has a fixed set of arguments if (for some reason) you want to disable the error checks of function prototyping. Because of this, you should restrict your use of variable argument lists to C and avoid them in C++ (in which, as you'll learn, there are much better alternatives). Handling variable argument lists is described in the library section of your local C guide.

Function return values

A C++ function prototype must specify the return value type of the function (in C, if you leave off the return value type it defaults to **int**). The return type specification precedes the function name. To specify that no value is returned, use the **void** keyword. This will generate an error if you try to return a value from the function. Here are some complete function prototypes:

```
int f1(void); // Returns an int, takes no arguments
int f2(); // Like f1() in C++ but not in Standard C!
float f3(float, int, char, double); // Returns a float
void f4(void); // Takes no arguments, returns nothing
```

To return a value from a function, you use the **return** statement. **return** exits the function back to the point right after the function call. If **return** has an argument, that argument becomes the return value of the function. If a function says that it will return a particular type, then each **return** statement must return that type. You can have more than one **return** statement in a function definition:

```
//: C03:Return.cpp
// Use of "return"
#include <iostream>
using namespace std;

char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

In **cfunc( )**, the first **if** that evaluates to **true** exits the function via the **return** statement. Notice that a function declaration is not necessary because the function definition appears before it is used in **main( )**, so the compiler knows about it from that function definition.

Using the C function library

All the functions in your local C function library are available while you are programming in C++. You should look hard at the function library before defining your own function – there's a good chance that someone has already solved your problem for you, and probably given it a lot more thought and debugging.

A word of caution, though: many compilers include a lot of extra functions that make life even easier and are tempting to use, but are not part of the Standard C library. If you are certain you will never want to move the application to another platform (and who is certain of that?), go ahead –use those functions and make your life easier. If you want your application to be portable, you should restrict yourself to Standard library functions. If you must perform platform-specific activities, try to isolate that code in one spot so it can be changed easily when porting to another platform. In C++, platform-specific activities are often encapsulated in a class, which is the ideal solution.

The formula for using a library function is as follows: first, find the function in your programming reference (many programming references will index the function by category as well as alphabetically). The description of the function should include a section that demonstrates the syntax of the code. The top of this section usually has at least one **#include** line, showing you the header file containing the function prototype. Duplicate this **#include** line in your file so the function is properly declared. Now you can call the function in the same way it appears in the syntax section. If you make a mistake, the compiler will discover it by comparing your function call to the function prototype in the header and tell you about your error. The linker searches the Standard library by default, so that's all you need to do: include the header file and call the function.

Creating your own libraries with the librarian

You can collect your own functions together into a library. Most programming packages come with a librarian that manages groups of object modules. Each librarian has its own commands, but the general idea is this: if you want to create a library, make a header file containing the function prototypes for all the functions in your library. Put this header file somewhere in the preprocessor's search path, either in the local directory (so it can be found by **#include "header"**) or in the include directory (so it can be found by **#include <header>**). Now take all the object modules and hand them to the librarian along with a name for the finished library (most librarians require a common extension, such as **.lib** or **.a**). Place the finished library where the other libraries reside so the linker can find it. When you use your library, you will have to add something to the command line so the linker knows to search the library for the functions you call. You must find all the details in your local manual, since they vary from system to system.

Controlling execution

This section covers the execution control statements in C++. You must be familiar with these statements before you can read and write C or C++ code.

C++ uses all of C's execution control statements. These include **if-else**, **while**, **do-while**, **for**, and a selection statement called **switch**. C++ also allows the infamous **goto**, which will be avoided in this book.

True and false

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **A == B**. This uses the conditional operator == to see if the variable **A** is equivalent to the variable **B.** The expression produces a Boolean **true** or **false** (these are keywords only in C++; in C an expression is "true" if it evaluates to a nonzero value). Other conditional operators are >, <, >=, etc. Conditional statements are covered more fully later in this chapter.

if-else

The **if-else** statement can exist in two forms: with or without the **else**. The two forms are:

```
if(expression)
    statement
```

or

```
if(expression)
    statement
else
    statement
```

The "expression" evaluates to **true** or **false**. The "statement" means either a simple statement terminated by a semicolon or a compound statement, which is a group of simple statements enclosed in braces. Any time the word "statement" is used, it always implies that the statement is simple or compound. Note that this statement can also be another **if**, so they can be strung together.

```
//: C03:Ifthen.cpp
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "It's greater than 5" << endl;
  else
    if(i < 5)
      cout << "It's less than 5 " << endl;
    else
      cout << "It's equal to 5 " << endl;

  cout << "type a number and 'Enter'" << endl;
```

```
  cin >> i;
  if(i < 10)
    if(i > 5)  // "if" is just another statement
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Matches "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

It is conventional to indent the body of a control flow statement so the reader may easily determine where it begins and ends[30].

while

while, **do-while,** and **for** control looping. A statement repeats until the controlling expression evaluates to **false**. The form of a **while** loop is

```
while(expression)
    statement
```

The expression is evaluated once at the beginning of the loop and again before each further iteration of the statement.

This example stays in the body of the **while** loop until you type the secret number or press control-C.

```
//: C03:Guess.cpp
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" is the "not-equal" conditional:
  while(guess != secret) { // Compound statement
    cout << "guess the number: ";
    cin >> guess;
  }
  cout << "You guessed it!" << endl;
} ///:~
```

The **while**'s conditional expression is not restricted to a simple test as in the example above; it can be as complicated as you like as long as it produces a **true** or **false** result. You will even see code where the loop has no body, just a bare semicolon:

```
while(/* Do a lot here */)
 ;
```

In these cases, the programmer has written the conditional expression not only to perform the test but also to do the work.

do-while

The form of **do-while** is

```
do
    statement
while(expression);
```

The **do-while** is different from the while because the statement always executes at least once, even if the expression evaluates to false the first time. In a regular **while**, if the conditional is false the first time the statement never executes.

If a **do-while** is used in **Guess.cpp**, the variable **guess** does not need an initial dummy value, since it is initialized by the **cin** statement before it is tested:

```
//: C03:Guess2.cpp
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // No initialization needed here
  do {
    cout << "guess the number: ";
    cin >> guess; // Initialization happens
  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

For some reason, most programmers tend to avoid **do-while** and just work with **while**.

for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the **for** loop is:

```
for(initialization; conditional; step)
 statement
```

Any of the expressions *initialization*, *conditional,* or *step* may be empty. The *initialization* code executes once at the very beginning. The *conditional* is tested before each iteration (if it evaluates to false at the beginning, the statement never executes). At the end of each loop, the *step* executes.

for loops are usually used for "counting" tasks:

```
//: C03:Charlist.cpp
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;
```

```
int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // ANSI Terminal Clear screen
      cout << " value: " << i
           << " character: "
           << char(i) // Type conversion
           << endl;
} ///:~
```

You may notice that the variable **i** is defined at the point where it is used, instead of at the beginning of the block denoted by the open curly brace '**{**'. This is in contrast to traditional procedural languages (including C), which require that all variables be defined at the beginning of the block. This will be discussed later in this chapter.

The break and continue keywords

Inside the body of any of the looping constructs **while**, **do-while,** or **for**, you can control the flow of the loop using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin a new iteration.

As an example of **break** and **continue**, this program is a very simple menu system:

```
//: C03:Menu.cpp
// Simple menu program demonstrating
// the use of "break" and "continue"
#include <iostream>
using namespace std;

int main() {
  char c; // To hold response
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: left, r: right, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // Out of "while(1)"
    if(c == 'l') {
      cout << "LEFT MENU:" << endl;
      cout << "select a or b: ";
      cin >> c;
      if(c == 'a') {
        cout << "you chose 'a'" << endl;
        continue; // Back to main menu
      }
      if(c == 'b') {
        cout << "you chose 'b'" << endl;
        continue; // Back to main menu
      }
      else {
        cout << "you didn't choose a or b!"
             << endl;
        continue; // Back to main menu
      }
    }
    if(c == 'r') {
```

```
      cout << "RIGHT MENU:" << endl;
      cout << "select c or d: ";
      cin >> c;
      if(c == 'c') {
        cout << "you chose 'c'" << endl;
        continue; // Back to main menu
      }
      if(c == 'd') {
        cout << "you chose 'd'" << endl;
        continue; // Back to main menu
      }
      else {
        cout << "you didn't choose c or d!"
             << endl;
        continue; // Back to main menu
      }
    }
    cout << "you must type l or r or q!" << endl;
  }
  cout << "quitting menu..." << endl;
} ///:~
```

If the user selects 'q' in the main menu, the **break** keyword is used to quit, otherwise the program just continues to execute indefinitely. After each of the sub-menu selections, the **continue** keyword is used to pop back up to the beginning of the while loop.

The **while(true)** statement is the equivalent of saying "do this loop forever." The **break** statement allows you to break out of this infinite while loop when the user types a 'q.'

switch

A **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    case integral-value4 : statement; break;
    case integral-value5 : statement; break;
    (...)
    default: statement;
}
```

Selector is an expression that produces an integral value. The **switch** compares the result of *selector* to each *integral value*. If it finds a match, the corresponding statement (simple or compound) executes. If no match occurs, the **default** statement executes.

You will notice in the definition above that each **case** ends with a **break**, which causes execution to jump to the end of the **switch** body (the closing brace that completes the **switch**). This is the conventional way to build a **switch** statement, but the **break** is optional. If it is missing, your **case** "drops through" to the one after it. That is, the code for the following **case** statements execute until a **break** is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer.

The **switch** statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value at compile-time. If you want to use, for example, a **string** object as a selector, it won't work in a **switch** statement. For a **string** selector, you must instead use a series of **if** statements and compare the **string** inside the conditional.

The menu example shown above provides a particularly nice example of a **switch**:

```
//: C03:Menu2.cpp
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag for quitting
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

The **quit** flag is a **bool**, short for "Boolean," which is a type you'll find only in C++. It can have only the keyword values **true** or **false**. Selecting 'q' sets the **quit** flag to **true**. The next time the selector is evaluated, **quit == false** returns **false** so the body of the **while** does not execute.

Using and misusing goto

The **goto** keyword is supported in C++, since it exists in C. Using **goto** is often dismissed as poor programming style, and most of the time it is. Anytime you use **goto**, look at your code and see if there's another way to do it. On rare occasions, you may discover **goto** can solve a problem that can't be solved otherwise, but still, consider it carefully. Here's an example that might make a plausible candidate:

```
//: C03:gotoKeyword.cpp
// The infamous goto is supported in C++
#include <iostream>
using namespace std;

int main() {
```

```
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bottom;
        // Break would only go to the outer 'for'
    }
  }
  bottom: // A label
  cout << val << endl;
} ///:~
```

The alternative would be to set a Boolean that is tested in the outer **for** loop, and then do a **break** from the inner for loop. However, if you have several levels of **for** or **while** this could get awkward.

Recursion

Recursion is an interesting and sometimes useful programming technique whereby you call the function that you're in. Of course, if this is all you do, you'll keep calling the function you're in until you run out of memory, so there must be some way to "bottom out" the recursive call. In the following example, this "bottoming out" is accomplished by simply saying that the recursion will go only until the **cat** exceeds 'Z':[31]

```
//: C03:CatsInHats.cpp
// Simple demonstration of recursion
#include <iostream>
using namespace std;

void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    removeHat(cat + 1); // Recursive call
  } else
    cout << "VOOM!!!" << endl;
}

int main() {
  removeHat('A');
} ///:~
```

In **removeHat( )**, you can see that as long as **cat** is less than 'Z', **removeHat( )** will be called from *within* **removeHat( )**, thus effecting the recursion. Each time **removeHat( )** is called, its argument is one greater than the current **cat** so the argument keeps increasing.

Recursion is often used when evaluating some sort of arbitrarily complex problem, since you aren't restricted to a particular "size" for the solution – the function can just keep recursing until it's reached the end of the problem.

Introduction to operators

You can think of operators as a special type of function (you'll learn that C++ operator overloading treats operators precisely that way). An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary function calls, but the effect is the same.

From your previous programming experience, you should be reasonably comfortable with the operators that have been used so far. The concepts of addition (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment(=) all have essentially the same meaning in any programming language. The full set of operators is enumerated later in this chapter.

Precedence

Operator precedence defines the order in which an expression evaluates when several different operators are present. C and C++ have specific rules to determine the order of evaluation. The easiest to remember is that multiplication and division happen before addition and subtraction. After that, if an expression isn't transparent to you it probably won't be for anyone reading the code, so you should use parentheses to make the order of evaluation explicit. For example:

```
A = X + Y - 2/2 + Z;
```

has a very different meaning from the same statement with a particular grouping of parentheses:

```
A = X + (Y - 2)/(2 + Z);
```

(Try evaluating the result with X = 1, Y = 2, and Z = 3.)

Auto increment and decrement

C, and therefore C++, is full of shortcuts. Shortcuts can make code much easier to type, and sometimes much harder to read. Perhaps the C language designers thought it would be easier to understand a tricky piece of code if your eyes didn't have to scan as large an area of print.

One of the nicer shortcuts is the auto-increment and auto-decrement operators. You often use these to change loop variables, which control the number of times a loop executes.

The auto-decrement operator is '**--**' and means "decrease by one unit." The auto-increment operator is '++' and means "increase by one unit." If **A** is an **int**, for example, the expression **++A** is equivalent to (**A = A + 1**). Auto-increment and auto-decrement operators produce the value of the variable as a result. If the operator appears before the variable, (i.e., **++A**), the operation is first performed and the resulting value is produced. If the operator appears after

the variable (i.e. **A++**), the current value is produced, and then the operation is performed. For example:

```
//: C03:AutoIncrement.cpp
// Shows use of auto-increment
// and auto-decrement operators.
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
} ///:~
```

If you've been wondering about the name "C++," now you understand. It implies "one step beyond C."

Introduction to data types

Data types define the way you use storage (memory) in the programs you write. By specifying a data type, you tell the compiler how to create a particular piece of storage, and also how to manipulate that storage.

Data types can be built-in or abstract. A built-in data type is one that the compiler intrinsically understands, one that is wired directly into the compiler. The types of built-in data are almost identical in C and C++. In contrast, a user-defined data type is one that you or another programmer create as a class. These are commonly referred to as abstract data types. The compiler knows how to handle built-in types when it starts up; it "learns" how to handle abstract data types by reading header files containing class declarations (you'll learn about this in later chapters).

Basic built-in types

The Standard C specification for built-in types (which C++ inherits) doesn't say how many bits each of the built-in types must contain. Instead, it stipulates the minimum and maximum values that the built-in type must be able to hold. When a machine is based on binary, this maximum value can be directly translated into a minimum number of bits necessary to hold that value. However, if a machine uses, for example, binary-coded decimal (BCD) to represent numbers, then the amount of space in the machine required to hold the maximum numbers for each data type will be different. The minimum and maximum values that can be stored in the various data types are defined in the system header files **limits.h** and **float.h** (in C++ you will generally **#include <climits>** and **<cfloat>** instead).

C and C++ have four basic built-in data types, described here for binary-based machines. A **char** is for character storage and uses a minimum of 8 bits (one byte) of storage, although it may be larger. An **int** stores an integral number and uses a minimum of two bytes of storage. The **float** and **double** types store floating-point numbers, usually in IEEE floating-point

format. **float** is for single-precision floating point and **double** is for double-precision floating point.

As mentioned previously, you can define variables anywhere in a scope, and you can define and initialize them at the same time. Here's how to define variables using the four basic data types:

```
//: C03:Basic.cpp
// Defining the four basic data
// types in C and C++

int main() {
  // Definition without initialization:
  char protein;
  int carbohydrates;
  float fiber;
  double fat;
  // Simultaneous definition & initialization:
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
  // Exponential notation:
  double fudge_ripple = 6e-4;
} ///:~
```

The first part of the program defines variables of the four basic data types without initializing them. If you don't initialize a variable, the Standard says that its contents are undefined (usually, this means they contain garbage). The second part of the program defines and initializes variables at the same time (it's always best, if possible, to provide an initialization value at the point of definition). Notice the use of exponential notation in the constant 6e-4, meaning "6 times 10 to the minus fourth power."

bool, true, & false

Before **bool** became part of Standard C++, everyone tended to use different techniques in order to produce Boolean-like behavior. These produced portability problems and could introduce subtle errors.

The Standard C++ **bool** type can have two states expressed by the built-in constants **true** (which converts to an integral one) and **false** (which converts to an integral zero). All three names are keywords. In addition, some language elements have been adapted:

Element	Usage with bool
&& || !	Take bool arguments and produce **bool** results.
< > <= >= == !=	Produce **bool** results.
if, **for**,	Conditional expressions

while, **do**	convert to **bool** values.
? :	First operand converts to **bool** value.

Because there's a lot of existing code that uses an **int** to represent a flag, the compiler will implicitly convert from an **int** to a **bool** (nonzero values will produce **true** while zero values produce **false**). Ideally, the compiler will give you a warning as a suggestion to correct the situation.

An idiom that falls under "poor programming style" is the use of ++ to set a flag to true. This is still allowed, but *deprecated*, which means that at some time in the future it will be made illegal. The problem is that you're making an implicit type conversion from **bool** to **int**, incrementing the value (perhaps beyond the range of the normal **bool** values of zero and one), and then implicitly converting it back again.

Pointers (which will be introduced later in this chapter) will also be automatically converted to **bool** when necessary.

Specifiers

Specifiers modify the meanings of the basic built-in types and expand them to a much larger set. There are four specifiers: **long**, **short**, **signed**, and **unsigned**.

long and **short** modify the maximum and minimum values that a data type will hold. A plain **int** must be at least the size of a **short**. The size hierarchy for integral types is: **short int**, **int**, **long int**. All the sizes could conceivably be the same, as long as they satisfy the minimum/maximum value requirements. On a machine with a 64-bit word, for instance, all the data types might be 64 bits.

The size hierarchy for floating point numbers is: **float**, **double**, and **long double**. "long float" is not a legal type. There are no **short** floating-point numbers.

The **signed** and **unsigned** specifiers tell the compiler how to use the sign bit with integral types and characters (floating-point numbers always contain a sign). An **unsigned** number does not keep track of the sign and thus has an extra bit available, so it can store positive numbers twice as large as the positive numbers that can be stored in a **signed** number. **signed** is the default and is only necessary with **char**; **char** may or may not default to **signed**. By specifying **signed char**, you force the sign bit to be used.

The following example shows the size of the data types in bytes by using the **sizeof** operator, introduced later in this chapter:

```
//: C03:Specify.cpp
// Demonstrates the use of specifiers
#include <iostream>
using namespace std;

int main() {
  char c;
```

```
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Same as short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Same as long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

Be aware that the results you get by running this program will probably be different from one machine/operating system/compiler to the next, since (as mentioned previously) the only thing that must be consistent is that each different type hold the minimum and maximum values specified in the Standard.

When you are modifying an **int** with **short** or **long**, the keyword **int** is optional, as shown above.

Introduction to pointers

Whenever you run a program, it is first loaded (typically from disk) into the computer's memory. Thus, all elements of your program are located somewhere in memory. Memory is typically laid out as a sequential series of memory locations; we usually refer to these locations as eight-bit *bytes* but actually the size of each space depends on the architecture of the particular machine and is usually called that machine's *word size*. Each space can be uniquely distinguished from all other spaces by its *address*. For the purposes of this discussion, we'll just say that all machines use bytes that have sequential addresses starting at zero and going up to however much memory you have in your computer.

Since your program lives in memory while it's being run, every element of your program has an address. Suppose we start with a simple program:

```
//: C03:YourPets1.cpp
```

```
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
} ///:~
```

Each of the elements in this program has a location in storage when the program is running. Even the function occupies storage. As you'll see, it turns out that what an element is and the way you define it usually determines the area of memory where that element is placed.

There is an operator in C and C++ that will tell you the address of an element. This is the '**&**' operator. All you do is precede the identifier name with '**&**' and it will produce the address of that identifier. **YourPets1.cpp** can be modified to print out the addresses of all its elements, like this:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

The **(long)** is a *cast*. It says "Don't treat this as if it's normal type, instead treat it as a **long**." The cast isn't essential, but if it wasn't there, the addresses would have been printed out in hexadecimal instead, so casting to a **long** makes things a little more readable.

The results of this program will vary depending on your computer, OS, and all sorts of other factors, but it will always give you some interesting insights. For a single run on my computer, the results looked like this:

```
f(): 4198736
dog: 4323632
cat: 4323636
```

```
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

You can see how the variables that are defined inside **main( )** are in a different area than the variables defined outside of **main( )**; you'll understand why as you learn more about the language. Also, **f( )** appears to be in its own area; code is typically separated from data in memory.

Another interesting thing to note is that variables defined one right after the other appear to be placed contiguously in memory. They are separated by the number of bytes that are required by their data type. Here, the only data type used is **int**, and **cat** is four bytes away from **dog**, **bird** is four bytes away from **cat**, etc. So it would appear that, on this machine, an **int** is four bytes long.

Other than this interesting experiment showing how memory is mapped out, what can you do with an address? The most important thing you can do is store it inside another variable for later use. C and C++ have a special type of variable that holds an address. This variable is called a *pointer*.

The operator that defines a pointer is the same as the one used for multiplication: '*'. The compiler knows that it isn't multiplication because of the context in which it is used, as you will see.

When you define a pointer, you must specify the type of variable it points to. You start out by giving the type name, then instead of immediately giving an identifier for the variable, you say "Wait, it's a pointer" by inserting a star between the type and the identifier. So a pointer to an **int** looks like this:

```
int* ip; // ip points to an int variable
```

The association of the '*' with the type looks sensible and reads easily, but it can actually be a bit deceiving. Your inclination might be to say "intpointer" as if it is a single discrete type. However, with an **int** or other basic data type, it's possible to say:

```
int a, b, c;
```

whereas with a pointer, you'd *like* to say:

```
int* ipa, ipb, ipc;
```

C syntax (and by inheritance, C++ syntax) does not allow such sensible expressions. In the definitions above, only **ipa** is a pointer, but **ipb** and **ipc** are ordinary **int**s (you can say that "* binds more tightly to the identifier"). Consequently, the best results can be achieved by using only one definition per line; you still get the sensible syntax without the confusion:

```
int* ipa;
int* ipb;
int* ipc;
```

Since a general guideline for C++ programming is that you should always initialize a variable at the point of definition, this form actually works better. For example, the variables above are not initialized to any particular value; they hold garbage. It's much better to say something like:

```
int a = 47;
int* ipa = &a;
```

Now both **a** and **ipa** have been initialized, and **ipa** holds the address of **a**.

Once you have an initialized pointer, the most basic thing you can do with it is to use it to modify the value it points to. To access a variable through a pointer, you *dereference* the pointer using the same operator that you used to define it, like this:

```
*ipa = 100;
```

Now **a** contains the value 100 instead of 47.

These are the basics of pointers: you can hold an address, and you can use that address to modify the original variable. But the question still remains: why do you want to modify one variable using another variable as a proxy?

For this introductory view of pointers, we can put the answer into two broad categories:

1. To change "outside objects" from within a function. This is perhaps the most basic use of pointers, and it will be examined here.
2. To achieve many other clever programming techniques, which you'll learn about in portions of the rest of the book.

Modifying the outside object

Ordinarily, when you pass an argument to a function, a copy of that argument is made inside the function. This is referred to as *pass-by-value*. You can see the effect of pass-by-value in the following program:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

In **f( )**, **a** is a *local variable*, so it exists only for the duration of the function call to **f( )**. Because it's a function argument, the value of **a** is initialized by the arguments that are passed when the function is called; in **main( )** the argument is **x,** which has a value of 47, so this value is copied into **a** when **f( )** is called.

When you run this program you'll see:

```
x = 47
a = 47
a = 5
x = 47
```

Initially, of course, **x** is 47. When **f( )** is called, temporary space is created to hold the variable **a** for the duration of the function call, and **a** is initialized by copying the value of **x**, which is verified by printing it out. Of course, you can change the value of **a** and show that it is changed. But when **f( )** is completed, the temporary space that was created for **a** disappears, and we see that the only connection that ever existed between **a** and **x** happened when the value of **x** was copied into **a**.

When you're inside **f( )**, **x** is the *outside object* (my terminology), and changing the local variable does not affect the outside object, naturally enough, since they are two separate locations in storage. But what if you *do* want to modify the outside object? This is where pointers come in handy. In a sense, a pointer is an alias for another variable. So if we pass a *pointer* into a function instead of an ordinary value, we are actually passing an alias to the outside object, enabling the function to modify that outside object, like this:

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

Now **f( )** takes a pointer as an argument and dereferences the pointer during assignment, and this causes the outside object **x** to be modified. The output is:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Notice that the value contained in **p** is the same as the address of **x** – the pointer **p** does indeed point to **x**. If that isn't convincing enough, when **p** is dereferenced to assign the value 5, we see that the value of **x** is now changed to 5 as well.

Thus, passing a pointer into a function will allow that function to modify the outside object. You'll see plenty of other uses for pointers later, but this is arguably the most basic and possibly the most common use.

Introduction to C++ references

Pointers work roughly the same in C and in C++, but C++ adds an additional way to pass an address into a function. This is *pass-by-reference* and it exists in several other programming languages so it was not a C++ invention.

Your initial perception of references may be that they are unnecessary, that you could write all your programs without references. In general, this is true, with the exception of a few important places that you'll learn about later in the book. You'll also learn more about references later, but the basic idea is the same as the demonstration of pointer use above: you can pass the address of an argument using a reference. The difference between references and pointers is that *calling* a function that takes references is cleaner, syntactically, than calling a function that takes pointers (and it is exactly this syntactic difference that makes references essential in certain situations). If **PassAddress.cpp** is modified to use references, you can see the difference in the function call in **main( )**:

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

In **f( )**'s argument list, instead of saying **int*** to pass a pointer, you say **int&** to pass a reference. Inside **f( )**, if you just say '**r**' (which would produce the address if **r** were a pointer) you get *the value in the variable that **r** references*. If you assign to **r**, you actually assign to the variable that **r** references. In fact, the only way to get the address that's held inside **r** is with the '**&**' operator.

In **main( )**, you can see the key effect of references in the syntax of the call to **f( )**, which is just **f(x)**. Even though this looks like an ordinary pass-by-value, the effect of the reference is

that it actually takes the address and passes it in, rather than making a copy of the value. The output is:

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

So you can see that pass-by-reference allows a function to modify the outside object, just like passing a pointer does (you can also observe that the reference obscures the fact that an address is being passed; this will be examined later in the book). Thus, for this simple introduction you can assume that references are just a syntactically different way (sometimes referred to as "syntactic sugar") to accomplish the same thing that pointers do: allow functions to change outside objects.

Pointers and references as modifiers

So far, you've seen the basic data types **char**, **int**, **float,** and **double**, along with the specifiers **signed**, **unsigned**, **short,** and **long**, which can be used with the basic data types in almost any combination. Now we've added pointers and references that are orthogonal to the basic data types and specifiers, so the possible combinations have just tripled:

```
//: C03:AllDefinitions.cpp
// All possible combinations of basic data types,
// specifiers, pointers and references
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);

int main() {} ///:~
```

Pointers and references also work when passing objects into and out of functions; you'll learn about this in a later chapter.

There's one other type that works with pointers: **void**. If you state that a pointer is a **void***, it means that any type of address at all can be assigned to that pointer (whereas if you have an **int***, you can assign only the address of an **int** variable to that pointer). For example:

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
  double d;
  // The address of ANY type can be
  // assigned to a void pointer:
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

Once you assign to a **void*** you lose any information about what type it is. This means that before you can use the pointer, you must cast it to the correct type:

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
  // Can't dereference a void pointer:
  // *vp = 3; // Compile-time error
  // Must cast back to int before dereferencing:
  *((int*)vp) = 3;
} ///:~
```

The cast **(int*)vp** takes the **void*** and tells the compiler to treat it as an **int***, and thus it can be successfully dereferenced. You might observe that this syntax is ugly, and it is, but it's worse than that – the **void*** introduces a hole in the language's type system. That is, it allows, or even promotes, the treatment of one type as another type. In the example above, I treat an **int** as an **int** by casting **vp** to an **int***, but there's nothing that says I can't cast it to a **char*** or **double***, which would modify a different amount of storage that had been allocated for the **int**, possibly crashing the program. In general, **void** pointers should be avoided, and used only in rare special cases, the likes of which you won't be ready to consider until significantly later in the book.

You cannot have a **void** reference, for reasons that will be explained in Chapter 11.

Scoping

Scoping rules tell you where a variable is valid, where it is created, and where it gets destroyed (i.e., goes out of scope). The scope of a variable extends from the point where it is defined to the first closing brace that matches the closest opening brace before the variable was defined. That is, a scope is defined by its "nearest" set of braces. To illustrate:

```
//: C03:Scope.cpp
// How variables are scoped
```

```
int main() {
  int scp1;
  // scp1 visible here
  {
    // scp1 still visible here
    //.....
    int scp2;
    // scp2 visible here
    //.....
    {
      // scp1 & scp2 still visible here
      //..
      int scp3;
      // scp1, scp2 & scp3 visible here
      // ...
    } // <-- scp3 destroyed here
    // scp3 not available here
    // scp1 & scp2 still visible here
    // ...
  } // <-- scp2 destroyed here
  // scp3 & scp2 not available here
  // scp1 still visible here
  //..
} // <-- scp1 destroyed here
///:~
```

The example above shows when variables are visible and when they are unavailable (that is, when they *go out of scope*). A variable can be used only when inside its scope. Scopes can be nested, indicated by matched pairs of braces inside other matched pairs of braces. Nesting means that you can access a variable in a scope that encloses the scope you are in. In the example above, the variable **scp1** is available inside all of the other scopes, while **scp3** is available only in the innermost scope.

Defining variables on the fly

As noted earlier in this chapter, there is a significant difference between C and C++ when defining variables. Both languages require that variables be defined before they are used, but C (and many other traditional procedural languages) forces you to define all the variables at the beginning of a scope, so that when the compiler creates a block it can allocate space for those variables.

While reading C code, a block of variable definitions is usually the first thing you see when entering a scope. Declaring all variables at the beginning of the block requires the programmer to write in a particular way because of the implementation details of the language. Most people don't know all the variables they are going to use before they write the code, so they must keep jumping back to the beginning of the block to insert new variables, which is awkward and causes errors. These variable definitions don't usually mean much to the reader, and they actually tend to be confusing because they appear apart from the context in which they are used.

C++ (not C) allows you to define variables anywhere in a scope, so you can define a variable right before you use it. In addition, you can initialize the variable at the point you define it, which prevents a certain class of errors. Defining variables this way makes the code much

easier to write and reduces the errors you get from being forced to jump back and forth within a scope. It makes the code easier to understand because you see a variable defined in the context of its use. This is especially important when you are defining and initializing a variable at the same time – you can see the meaning of the initialization value by the way the variable is used.

You can also define variables inside the control expressions of **for** loops and **while** loops, inside the conditional of an **if** statement, and inside the selector statement of a **switch**. Here's an example showing on-the-fly variable definitions:

```
//: C03:OnTheFly.cpp
// On-the-fly variable definitions
#include <iostream>
using namespace std;

int main() {
  //..
  { // Begin a new scope
    int q = 0; // C requires definitions here
    //..
    // Define at point of use:
    for(int i = 0; i < 100; i++) {
      q++; // q comes from a larger scope
      // Definition at the end of the scope:
      int p = 12;
    }
    int p = 1;  // A different p
  } // End scope containing q & outer p
  cout << "Type characters:" << endl;
  while(char c = cin.get() != 'q') {
    cout << c << " wasn't it" << endl;
    if(char x = c == 'a' || c == 'b')
      cout << "You typed a or b" << endl;
    else
      cout << "You typed " << x << endl;
  }
  cout << "Type A, B, or C" << endl;
  switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
  }
} ///:~
```

In the innermost scope, **p** is defined right before the scope ends, so it is really a useless gesture (but it shows you can define a variable anywhere). The **p** in the outer scope is in the same situation.

The definition of **i** in the control expression of the **for** loop is an example of being able to define a variable *exactly* at the point you need it (you can do this only in C++). The scope of **i** is the scope of the expression controlled by the **for** loop, so you can turn around and re-use **i** in the next **for** loop. This is a convenient and commonly-used idiom in C++; **i** is the classic name for a loop counter and you don't have to keep inventing new names.

Although the example also shows variables defined within **while**, **if,** and **switch** statements, this kind of definition is much less common than those in **for** expressions, possibly because the syntax is so constrained. For example, you cannot have any parentheses. That is, you cannot say:

```
while((char c = cin.get()) != 'q')
```

The addition of the extra parentheses would seem like an innocent and useful thing to do, and because you cannot use them, the results are not what you might like. The problem occurs because '**!=**' has a higher precedence than '**=**', so the **char c** ends up containing a **bool** converted to **char**. When that's printed, on many terminals you'll see a smiley-face character.

In general, you can consider the ability to define variables within **while**, **if,** and **switch** statements as being there for completeness, but the only place you're likely to use this kind of variable definition is in a **for** loop (where you'll use it quite often).

Specifying storage allocation

When creating a variable, you have a number of options to specify the lifetime of the variable, how the storage is allocated for that variable, and how the variable is treated by the compiler.

Global variables

Global variables are defined outside all function bodies and are available to all parts of the program (even code in other files). Global variables are unaffected by scopes and are always available (i.e., the lifetime of a global variable lasts until the program ends). If the existence of a global variable in one file is declared using the **extern** keyword in another file, the data is available for use by the second file. Here's an example of the use of global variables:

```
//: C03:Global.cpp
//{L} Global2
// Demonstration of global variables
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
  globe = 12;
  cout << globe << endl;
  func(); // Modifies globe
  cout << globe << endl;
} ///:~
```

Here's a file that accesses **globe** as an **extern**:

```
//: C03:Global2.cpp {O}
// Accessing external global variables
extern int globe;
// (The linker resolves the reference)
void func() {
  globe = 47;
```

```
} ///:~
```

Storage for the variable **globe** is created by the definition in **Global.cpp**, and that same variable is accessed by the code in **Global2.cpp**. Since the code in **Global2.cpp** is compiled separately from the code in **Global.cpp**, the compiler must be informed that the variable exists elsewhere by the declaration

```
extern int globe;
```

When you run the program, you'll see that the call to **func( )** does indeed affect the single global instance of **globe**.

In **Global.cpp**, you can see the special comment tag (which is my own design):

```
//{L} Global2
```

This says that to create the final program, the object file with the name **Global2** must be linked in (there is no extension because the extension names of object files differ from one system to the next). In **Global2.cpp**, the first line has another special comment tag **{O},** which says "Don't try to create an executable out of this file, it's being compiled so that it can be linked into some other executable." The **ExtractCode.cpp** program in Volume 2 of this book (downloadable at *www.BruceEckel.com*) reads these tags and creates the appropriate **makefile** so everything compiles properly (you'll learn about **makefile**s at the end of this chapter).

Local variables

Local variables occur within a scope; they are "local" to a function. They are often called *automatic* variables because they automatically come into being when the scope is entered and automatically go away when the scope closes. The keyword **auto** makes this explicit, but local variables default to **auto** so it is never necessary to declare something as an **auto**.

Register variables

A register variable is a type of local variable. The **register** keyword tells the compiler "Make accesses to this variable as fast as possible." Increasing the access speed is implementation dependent, but, as the name suggests, it is often done by placing the variable in a register. There is no guarantee that the variable will be placed in a register or even that the access speed will increase. It is a hint to the compiler.

There are restrictions to the use of **register** variables. You cannot take or compute the address of a **register** variable. A **register** variable can be declared only within a block (you cannot have global or **static register** variables). You can, however, use a **register** variable as a formal argument in a function (i.e., in the argument list).

In general, you shouldn't try to second-guess the compiler's optimizer, since it will probably do a better job than you can. Thus, the **register** keyword is best avoided.

static

The **static** keyword has several distinct meanings. Normally, variables defined local to a function disappear at the end of the function scope. When you call the function again, storage for the variables is created anew and the values are re-initialized. If you want a value to be extant throughout the life of a program, you can define a function's local variable to be **static** and give it an initial value. The initialization is performed only the first time the function is called, and the data retains its value between function calls. This way, a function can "remember" some piece of information between function calls.

You may wonder why a global variable isn't used instead. The beauty of a **static** variable is that it is unavailable outside the scope of the function, so it can't be inadvertently changed. This localizes errors.

Here's an example of the use of **static** variables:

```
//: C03:Static.cpp
// Using a static variable in a function
#include <iostream>
using namespace std;

void func() {
  static int i = 0;
  cout << "i = " << ++i << endl;
}

int main() {
  for(int x = 0; x < 10; x++)
    func();
} ///:~
```

Each time **func**( ) is called in the for loop, it prints a different value. If the keyword **static** is not used, the value printed will always be '1'.

The second meaning of **static** is related to the first in the "unavailable outside a certain scope" sense. When **static** is applied to a function name or to a variable that is outside of all functions, it means "This name is unavailable outside of this file." The function name or variable is local to the file; we say it has *file scope*. As a demonstration, compiling and linking the following two files will cause a linker error:

```
//: C03:FileStatic.cpp
// File scope demonstration. Compiling and
// linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;

int main() {
  fs = 1;
} ///:~
```

Even though the variable **fs** is claimed to exist as an **extern** in the following file, the linker won't find it because it has been declared **static** in **FileStatic.cpp**.

```
//: C03:FileStatic2.cpp {O}
// Trying to reference fs
extern int fs;
void func() {
  fs = 100;
} ///:~
```

The **static** specifier may also be used inside a **class**. This explanation will be delayed until you learn to create classes, later in the book.

extern

The **extern** keyword has already been briefly described and demonstrated. It tells the compiler that a variable or a function exists, even if the compiler hasn't yet seen it in the file currently being compiled. This variable or function may be defined in another file or further down in the current file. As an example of the latter:

```
//: C03:Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // The data definition
void func() {
  i++;
  cout << i;
} ///:~
```

When the compiler encounters the declaration '**extern int i**', it knows that the definition for **i** must exist somewhere as a global variable. When the compiler reaches the definition of **i**, no other declaration is visible, so it knows it has found the same **i** declared earlier in the file. If you were to define **i** as **static**, you would be telling the compiler that **i** is defined globally (via the **extern**), but it also has file scope (via the **static**), so the compiler will generate an error.

Linkage

To understand the behavior of C and C++ programs, you need to know about *linkage*. In an executing program, an identifier is represented by storage in memory that holds a variable or a compiled function body. Linkage describes this storage as it is seen by the linker. There are two types of linkage: *internal linkage* and *external linkage*.

Internal linkage means that storage is created to represent the identifier only for the file being compiled. Other files may use the same identifier name with internal linkage, or for a global variable, and no conflicts will be found by the linker – separate storage is created for each identifier. Internal linkage is specified by the keyword **static** in C and C++.

External linkage means that a single piece of storage is created to represent the identifier for all files being compiled. The storage is created once, and the linker must resolve all other references to that storage. Global variables and function names have external linkage. These are accessed from other files by declaring them with the keyword **extern**. Variables defined outside all functions (with the exception of **const** in C++) and function definitions default to external linkage. You can specifically force them to have internal linkage using the **static** keyword. You can explicitly state that an identifier has external linkage by defining it with the **extern** keyword. Defining a variable or function with **extern** is not necessary in C, but it is sometimes necessary for **const** in C++.

Automatic (local) variables exist only temporarily, on the stack, while a function is being called. The linker doesn't know about automatic variables, and so these have *no linkage*.

Constants

In old (pre-Standard) C, if you wanted to make a constant, you had to use the preprocessor:

```
#define PI 3.14159
```

Everywhere you used **PI**, the value 3.14159 was substituted by the preprocessor (you can still use this method in C and C++).

When you use the preprocessor to create constants, you place control of those constants outside the scope of the compiler. No type checking is performed on the name **PI** and you can't take the address of **PI** (so you can't pass a pointer or a reference to **PI**). **PI** cannot be a variable of a user-defined type. The meaning of **PI** lasts from the point it is defined to the end of the file; the preprocessor doesn't recognize scoping.

C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed. The modifier **const** tells the compiler that a name represents a constant. Any data type, built-in or user-defined, may be defined as **const**. If you define something as **const** and then attempt to modify it, the compiler will generate an error.

You must specify the type of a **const**, like this:

```
const int x = 10;
```

In Standard C and C++, you can use a named constant in an argument list, even if the argument it fills is a pointer or a reference (i.e., you can take the address of a **const**). A **const** has a scope, just like a regular variable, so you can "hide" a **const** inside a function and be sure that the name will not affect the rest of the program.

The **const** was taken from C++ and incorporated into Standard C, albeit quite differently. In C, the compiler treats a **const** just like a variable that has a special tag attached that says

“Don’t change me.” When you define a **const** in C, the compiler creates storage for it, so if you define more than one **const** with the same name in two different files (or put the definition in a header file), the linker will generate error messages about conflicts. The intended use of **const** in C is quite different from its intended use in C++ (in short, it’s nicer in C++).

Constant values

In C++, a **const** must always have an initialization value (in C, this is not true). Constant values for built-in types are expressed as decimal, octal, hexadecimal, or floating-point numbers (sadly, binary numbers were not considered important), or as characters.

In the absence of any other clues, the compiler assumes a constant value is a decimal number. The numbers 47, 0, and 1101 are all treated as decimal numbers.

A constant value with a leading 0 is treated as an octal number (base 8). Base 8 numbers can contain only digits 0-7; the compiler flags other digits as an error. A legitimate octal number is 017 (15 in base 10).

A constant value with a leading 0x is treated as a hexadecimal number (base 16). Base 16 numbers contain the digits 0-9 and a-f or A-F. A legitimate hexadecimal number is 0x1fe (510 in base 10).

Floating point numbers can contain decimal points and exponential powers (represented by e, which means “10 to the power of”). Both the decimal point and the **e** are optional. If you assign a constant to a floating-point variable, the compiler will take the constant value and convert it to a floating-point number (this process is one form of what’s called *implicit type conversion*). However, it is a good idea to use either a decimal point or an **e** to remind the reader that you are using a floating-point number; some older compilers also need the hint.

Legitimate floating-point constant values are: 1e4, 1.0001, 47.0, 0.0, and -1.159e-77. You can add suffixes to force the type of floating-point number: **f** or **F** forces a **float**, **L** or **l** forces a **long double**; otherwise the number will be a **double**.

Character constants are characters surrounded by single quotes, as: ‘**A**’, ‘**0**’, ‘ ‘. Notice there is a big difference between the character ‘**0**’ (ASCII 96) and the value **0**. Special characters are represented with the “backslash escape”: ‘**n**’ (newline), ‘**t**’ (tab), ‘\\’ (backslash), ‘**r**’ (carriage return), ‘**"**’ (double quotes), ‘**'**’ (single quote), etc. You can also express char constants in octal: ‘**17**’ or hexadecimal: ‘**xff**’.

volatile

Whereas the qualifier **const** tells the compiler “This never changes” (which allows the compiler to perform extra optimizations), the qualifier **volatile** tells the compiler “You never know when this will change,” and prevents the compiler from performing any optimizations based on the stability of that variable. Use this keyword when you read some value outside the control of your code, such as a register in a piece of communication hardware. A **volatile** variable is always read whenever its value is required, even if it was just read the line before.

A special case of some storage being "outside the control of your code" is in a multithreaded program. If you're watching a particular flag that is modified by another thread or process, that flag should be **volatile** so the compiler doesn't make the assumption that it can optimize away multiple reads of the flag.

Note that **volatile** may have no effect when a compiler is not optimizing, but may prevent critical bugs when you start optimizing the code (which is when the compiler will begin looking for redundant reads).

The **const** and **volatile** keywords will be further illuminated in a later chapter.

Operators and their use

This section covers all the operators in C and C++.

All operators produce a value from their operands. This value is produced without modifying the operands, except with the assignment, increment, and decrement operators. Modifying an operand is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value produced is available for your use just as in operators without side effects.

Assignment

Assignment is performed with the operator =. It means "Take the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)." An rvalue is any constant, variable, or expression that can produce a value, but an lvalue must be a distinct, named variable (that is, there must be a physical space in which to store data). For instance, you can assign a constant value to a variable (**A = 4;**), but you cannot assign anything to constant value – it cannot be an lvalue (you can't say **4 = A;**).

Mathematical operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (+), subtraction (-), division (/), multiplication (*), and modulus (**%**; this produces the remainder from integer division). Integer division truncates the result (it doesn't round). The modulus operator cannot be used with floating-point numbers.

C and C++ also use a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable **x** and assign **x** to the result, you say: **x += 4;**.

This example shows the use of the mathematical operators:

```
//: C03:Mathops.cpp
// Mathematical operators
#include <iostream>
using namespace std;
```

```
// A macro to display a string and a value.
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl

int main() {
  int i, j, k;
  float u, v, w;  // Applies to doubles, too
  cout << "enter an integer: ";
  cin >> j;
  cout << "enter another integer: ";
  cin >> k;
  PRINT("j",j);  PRINT("k",k);
  i = j + k; PRINT("j + k",i);
  i = j - k; PRINT("j - k",i);
  i = k / j; PRINT("k / j",i);
  i = k * j; PRINT("k * j",i);
  i = k % j; PRINT("k % j",i);
  // The following only works with integers:
  j %= k; PRINT("j %= k", j);
  cout << "Enter a floating-point number: ";
  cin >> v;
  cout << "Enter another floating-point number:";
  cin >> w;
  PRINT("v",v); PRINT("w",w);
  u = v + w; PRINT("v + w", u);
  u = v - w; PRINT("v - w", u);
  u = v * w; PRINT("v * w", u);
  u = v / w; PRINT("v / w", u);
  // The following works for ints, chars,
  // and doubles too:
  PRINT("u", u); PRINT("v", v);
  u += v; PRINT("u += v", u);
  u -= v; PRINT("u -= v", u);
  u *= v; PRINT("u *= v", u);
  u /= v; PRINT("u /= v", u);
} ///:~
```

The rvalues of all the assignments can, of course, be much more complex.

Introduction to preprocessor macros

Notice the use of the macro **PRINT( )** to save typing (and typing errors!). Preprocessor macros are traditionally named with all uppercase letters so they stand out – you'll learn later that macros can quickly become dangerous (and they can also be very useful).

The arguments in the parenthesized list following the macro name are substituted in all the code following the closing parenthesis. The preprocessor removes the name **PRINT** and substitutes the code wherever the macro is called, so the compiler cannot generate any error messages using the macro name, and it doesn't do any type checking on the arguments (the latter can be beneficial, as shown in the debugging macros at the end of the chapter).

Relational operators

Relational operators establish a relationship between the values of the operands. They produce a Boolean (specified with the **bool** keyword in C++) **true** if the relationship is true,

and **false** if the relationship is false. The relational operators are: less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equivalent (==), and not equivalent (**!**=). They may be used with all built-in data types in C and C++. They may be given special definitions for user-defined data types in C++ (you'll learn about this in Chapter 12, which covers operator overloading).

Logical operators

The logical operators *and* (**&&**) and *or* (||) produce a **true** or **false** based on the logical relationship of its arguments. Remember that in C and C++, a statement is **true** if it has a non-zero value, and **false** if it has a value of zero. If you print a **bool**, you'll typically see a '**1**' for **true** and '**0**' for **false**.

This example uses the relational and logical operators:

```
//: C03:Boolean.cpp
// Relational and logical operators.
#include <iostream>
using namespace std;

int main() {
  int i,j;
  cout << "Enter an integer: ";
  cin >> i;
  cout << "Enter another integer: ";
  cin >> j;
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
       << ((i < 10) && (j < 10))  << endl;
} ///:~
```

You can replace the definition for **int** with **float** or **double** in the program above. Be aware, however, that the comparison of a floating-point number with the value of zero is strict; a number that is the tiniest fraction different from another number is still "not equal." A floating-point number that is the tiniest bit above zero is still true.

Bitwise operators

The bitwise operators allow you to manipulate individual bits in a number (since floating point values use a special internal format, the bitwise operators work only with integral types: **char**, **int** and **long**). Bitwise operators perform Boolean algebra on the corresponding bits in the arguments to produce the result.

The bitwise *and* operator (**&**) produces a one in the output bit if both input bits are one; otherwise it produces a zero. The bitwise or operator (|) produces a one in the output bit if

either input bit is a one and produces a zero only if both input bits are zero. The bitwise *exclusive or*, or *xor* (^) produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise *not* (~, also called the *ones complement* operator) is a unary operator – it only takes one argument (all other bitwise operators are binary operators). Bitwise *not* produces the opposite of the input bit – a one if the input bit is zero, a zero if the input bit is one.

Bitwise operators can be combined with the = sign to unite the operation and assignment: **&=**, **|=**, and ^= are all legitimate operations (since ~ is a unary operator it cannot be combined with the = sign).

Shift operators

The shift operators also manipulate bits. The left-shift operator (<<) produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator. The right-shift operator (>>) produces the operand to the left of the operator shifted to the right by the number of bits specified after the operator. If the value after the shift operator is greater than the number of bits in the left-hand operand, the result is undefined. If the left-hand operand is unsigned, the right shift is a logical shift so the upper bits will be filled with zeros. If the left-hand operand is signed, the right shift may or may not be a logical shift (that is, the behavior is undefined).

Shifts can be combined with the equal sign (<<= and >>=). The lvalue is replaced by the lvalue shifted by the rvalue.

What follows is an example that demonstrates the use of all the operators involving bits. First, here's a general-purpose function that prints a byte in binary format, created separately so that it may be easily reused. The header file declares the function:

```
//: C03:printBinary.h
// Display a byte in binary
void printBinary(const unsigned char val);
///:~
```

Here's the implementation of the function:

```
//: C03:printBinary.cpp {O}
#include <iostream>
void printBinary(const unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
} ///:~
```

The **printBinary( )** function takes a single byte and displays it bit-by-bit. The expression

```
(1 << i)
```

produces a one in each successive bit position; in binary: 00000001, 00000010, etc. If this bit is bitwise *and*ed with **val** and the result is nonzero, it means there was a one in that position in **val**.

Finally, the function is used in the example that shows the bit-manipulation operators:

```
//: C03:Bitwise.cpp
//{L} printBinary
// Demonstration of bit manipulation
#include "printBinary.h"
#include <iostream>
using yle='color:blue'>namespace std;

// A macro to save typing:
#define PR(STR, EXPR) \
  cout << STR; printBinary(EXPR); cout << endl;

int main() {
  unsigned int getval;
  unsigned char a, b;
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; a = getval;
  PR("a in binary: ", a);
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; b = getval;
  PR("b in binary: ", b);
  PR("a | b = ", a | b);
  PR("a & b = ", a & b);
  PR("a ^ b = ", a ^ b);
  PR("~a = ", ~a);
  PR("~b = ", ~b);
  // An interesting bit pattern:
  unsigned char c = 0x5A;
  PR("c in binary: ", c);
  a |= c;
  PR("a |= c; a = ", a);
  b &= c;
  PR("b &= c; b = ", b);
  b ^= a;
  PR("b ^= a; b = ", b);
} ///:~
```

Once again, a preprocessor macro is used to save typing. It prints the string of your choice, then the binary representation of an expression, then a newline.

In **main( )**, the variables are **unsigned**. This is because, in general, you don't want signs when you are working with bytes. An **int** must be used instead of a **char** for **getval** because the "**cin >>**" statement will otherwise treat the first digit as a character. By assigning **getval** to **a** and **b**, the value is converted to a single byte (by truncating it).

The << and >> provide bit-shifting behavior, but when they shift bits off the end of the number, those bits are lost (it's commonly said that they fall into the mythical *bit bucket*, a place where discarded bits end up, presumably so they can be reused...). When manipulating bits you can also perform *rotation*, which means that the bits that fall off one end are inserted back at the other end, as if they're being rotated around a loop. Even though most computer

processors provide a machine-level rotate command (so you'll see it in the assembly language for that processor), there is no direct support for "rotate" in C or C++. Presumably the designers of C felt justified in leaving "rotate" off (aiming, as they said, for a minimal language) because you can build your own rotate command. For example, here are functions to perform left and right rotations:

```
//: C03:Rotation.cpp {O}
// Perform left and right rotations

unsigned char rol(unsigned char val) {
  int highbit;
  if(val & 0x80) // 0x80 is the high bit only
    highbit = 1;
  else
    highbit = 0;
  // Left shift (bottom bit becomes 0):
  val <<= 1;
  // Rotate the high bit onto the bottom:
  val |= highbit;
  return val;
}

unsigned char ror(unsigned char val) {
  int lowbit;
  if(val & 1) // Check the low bit
    lowbit = 1;
  else
    lowbit = 0;
  val >>= 1; // Right shift by one position
  // Rotate the low bit onto the top:
  val |= (lowbit << 7);
  return val;
} ///:~
```

Try using these functions in **Bitwise.cpp**. Notice the definitions (or at least declarations) of **rol( )** and **ror( )** must be seen by the compiler in **Bitwise.cpp** before the functions are used.

The bitwise functions are generally extremely efficient to use because they translate directly into assembly language statements. Sometimes a single C or C++ statement will generate a single line of assembly code.

Unary operators

Bitwise *not* isn't the only operator that takes a single argument. Its companion, the *logical not* (**!**), will take a **true** value and produce a **false** value. The unary minus (**-**) and unary plus (**+**) are the same operators as binary minus and plus; the compiler figures out which usage is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler can figure out:

```
x = a * -b;
```

but the reader might get confused, so it is safer to say:

```
x = a * (-b);
```

The unary minus produces the negative of the value. Unary plus provides symmetry with unary minus, although it doesn't actually do anything.

The increment and decrement operators (**++** and **--**) were introduced earlier in this chapter. These are the only operators other than those involving assignment that have side effects. These operators increase or decrease the variable by one unit, although "unit" can have different meanings according to the data type – this is especially true with pointers.

The last unary operators are the address-of (**&**), dereference (* and **->**), and cast operators in C and C++, and **new** and **delete** in C++. Address-of and dereference are used with pointers, described in this chapter. Casting is described later in this chapter, and **new** and **delete** are introduced in Chapter 4.

The ternary operator

The ternary **if-else** is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary **if-else** statement. It consists of three expressions: if the first expression (followed by a **?**) evaluates to **true**, the expression following the **?** is evaluated and its result becomes the value produced by the operator. If the first expression is **false**, the third expression (following a **:**) is executed and its result becomes the value produced by the operator.

The conditional operator can be used for its side effects or for the value it produces. Here's a code fragment that demonstrates both:

```
a = --b ? b : (b = -99);
```

Here, the conditional produces the rvalue. **a** is assigned to the value of **b** if the result of decrementing **b** is nonzero. If **b** became zero, **a** and **b** are both assigned to -99. **b** is always decremented, but it is assigned to -99 only if the decrement causes **b** to become 0. A similar statement can be used without the "**a =**" just for its side effects:

```
--b ? b : (b = -99);
```

Here the second B is superfluous, since the value produced by the operator is unused. An expression is required between the **?** and **:**. In this case, the expression could simply be a constant that might make the code run a bit faster.

The comma operator

The comma is not restricted to separating variable names in multiple definitions, such as

```
int i, j, k;
```

Of course, it's also used in function argument lists. However, it can also be used as an operator to separate expressions – in this case it produces only the value of the last expression. All the rest of the expressions in the comma-separated list are evaluated only for their side effects. This example increments a list of variables and uses the last one as the rvalue:

```
//: C03:CommaOperator.cpp
#include <iostream>
using namespace std;
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl;
  // The parentheses are critical here. Without
  // them, the statement will evaluate to:
  (a = b++), c++, d++, e++;
  cout << "a = " << a << endl;
} ///:~
```

In general, it's best to avoid using the comma as anything other than a separator, since people are not used to seeing it as an operator.

Common pitfalls when using operators

As illustrated above, one of the pitfalls when using operators is trying to get away without parentheses when you are even the least bit uncertain about how an expression will evaluate (consult your local C manual for the order of expression evaluation).

Another extremely common error looks like this:

```
//: C03:Pitfall.cpp
// Operator mistakes

int main() {
  int a = 1, b = 1;
  while(a = b) {
    // ....
  }
} ///:~
```

The statement **a = b** will always evaluate to true when **b** is non-zero. The variable **a** is assigned to the value of **b**, and the value of **b** is also produced by the operator =. In general, you want to use the equivalence operator == inside a conditional statement, not assignment. This one bites a lot of programmers (however, some compilers will point out the problem to you, which is helpful).

A similar problem is using bitwise *and* and *or* instead of their logical counterparts. Bitwise *and* and *or* use one of the characters (**&** or |), while logical *and* and *or* use two (**&&** and ||). Just as with = and ==, it's easy to just type one character instead of two. A useful mnemonic device is to observe that "Bits are smaller, so they don't need as many characters in their operators."

Casting operators

The word *cast* is used in the sense of "casting into a mold." The compiler will automatically change one type of data into another if it makes sense. For instance, if you assign an integral value to a floating-point variable, the compiler will secretly call a function (or more probably, insert code) to convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

To perform a cast, put the desired data type (including all modifiers) inside parentheses to the left of the value. This value can be a variable, a constant, the value produced by an expression, or the return value of a function. Here's an example:

```
//: C03:SimpleCast.cpp
int main() {
  int b = 200;
  unsigned long a = (unsigned long int)b;
} ///:~
```

Casting is powerful, but it can cause headaches because in some situations it forces the compiler to treat data as if it were (for instance) larger than it really is, so it will occupy more space in memory; this can trample over other data. This usually occurs when casting pointers, not when making simple casts like the one shown above.

C++ has an additional casting syntax, which follows the function call syntax. This syntax puts the parentheses around the argument, like a function call, rather than around the data type:

```
//: C03:FunctionCallCast.cpp
int main() {
  float a = float(200);
  // This is equivalent to:
  float b = (float)200;
} ///:~
```

Of course in the case above you wouldn't really need a cast; you could just say **200f** (in effect, that's typically what the compiler will do for the above expression). Casts are generally used instead with variables, rather than constants.

C++ explicit casts

Casts should be used carefully, because what you are actually doing is saying to the compiler "Forget type checking – treat it as this other type instead." That is, you're introducing a hole in the C++ type system and preventing the compiler from telling you that you're doing something wrong with a type. What's worse, the compiler believes you implicitly and doesn't perform any other checking to catch errors. Once you start casting, you open yourself up for all kinds of problems. In fact, any program that uses a lot of casts should be viewed with suspicion, no matter how much you are told it simply "must" be done that way. In general, casts should be few and isolated to the solution of very specific problems.

Once you understand this and are presented with a buggy program, your first inclination may be to look for casts as culprits. But how do you locate C-style casts? They are simply type names inside of parentheses, and if you start hunting for such things you'll discover that it's often hard to distinguish them from the rest of your code.

Standard C++ includes an explicit cast syntax that can be used to completely replace the old C-style casts (of course, C-style casts cannot be outlawed without breaking code, but compiler writers could easily flag old-style casts for you). The explicit cast syntax is such that you can easily find them, as you can see by their names:

static_cast	For "well-behaved" and "reasonably well-behaved" casts, including things you might now do without a cast (such as an automatic type conversion).
const_cast	To cast away **const** and/or **volatile**.
reinterpret_cast	To cast to a completely different meaning. The key is that you'll need to cast back to the original type to use it safely. The type you cast to is typically used only for bit twiddling or some other mysterious purpose. This is the most dangerous of all the casts.
dynamic_cast	For type-safe downcasting (this cast will be described in Chapter 15).

The first three explicit casts will be described more completely in the following sections, while the last one can be demonstrated only after you've learned more, in Chapter 15.

static_cast

A **static_cast** is used for all conversions that are well-defined. These include "safe" conversions that the compiler would allow you to do without a cast and less-safe conversions that are nonetheless well-defined. The types of conversions covered by **static_cast** include typical castless conversions, narrowing (information-losing) conversions, forcing a conversion from a **void***, implicit type conversions, and static navigation of class hierarchies (since you haven't seen classes and inheritance yet, this last topic will be delayed until Chapter 15):

```
//: C03:static_cast.cpp
void func(int) {}
```

```
int main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  // (1) Typical castless conversions:
  l = i;
  f = i;
  // Also works:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  // (2) Narrowing conversions:
  i = l; // May lose digits
  i = f; // May lose info
  // Says "I know," eliminates warnings:
  i = static_cast<int>(l);
  i = static_cast<int>(f);
  char c = static_cast<char>(i);

  // (3) Forcing a conversion from void* :
  void* vp = &i;
  // Old way produces a dangerous conversion:
  float* fp = (float*)vp;
  // The new way is equally dangerous:
  fp = static_cast<float*>(vp);

  // (4) Implicit type conversions, normally
  // performed by the compiler:
  double d = 0.0;
  int x = d; // Automatic type conversion
  x = static_cast<int>(d); // More explicit
  func(d); // Automatic type conversion
  func(static_cast<int>(d)); // More explicit
} ///:~
```

In Section (1), you see the kinds of conversions you're used to doing in C, with or without a cast. Promoting from an **int** to a **long** or **float** is not a problem because the latter can always hold every value that an **int** can contain. Although it's unnecessary, you can use **static_cast** to highlight these promotions.

Converting back the other way is shown in (2). Here, you can lose data because an **int** is not as "wide" as a **long** or a **float**; it won't hold numbers of the same size. Thus these are called *narrowing conversions*. The compiler will still perform these, but will often give you a warning. You can eliminate this warning and indicate that you really did mean it using a cast.

Assigning from a **void*** is not allowed without a cast in C++ (unlike C), as seen in (3). This is dangerous and requires that programmers know what they're doing. The **static_cast**, at least, is easier to locate than the old standard cast when you're hunting for bugs.

Section (4) of the program shows the kinds of implicit type conversions that are normally performed automatically by the compiler. These are automatic and require no casting, but again **static_cast** highlights the action in case you want to make it clear what's happening or hunt for it later.

const_cast

If you want to convert from a **const** to a non**const** or from a **volatile** to a non**volatile**, you use **const_cast**. This is the *only* conversion allowed with **const_cast**; if any other conversion is involved it must be done using a separate expression or you'll get a compile-time error.

```
//: C03:const_cast.cpp
int main() {
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
} ///:~
```

If you take the address of a **const** object, you produce a pointer to a **const**, and this cannot be assigned to a non**const** pointer without a cast. The old-style cast will accomplish this, but the **const_cast** is the appropriate one to use. The same holds true for **volatile**.

reinterpret_cast

This is the least safe of the casting mechanisms, and the one most likely to produce bugs. A **reinterpret_cast** pretends that an object is just a bit pattern that can be treated (for some dark purpose) as if it were an entirely different type of object. This is the low-level bit twiddling that C is notorious for. You'll virtually always need to **reinterpret_cast** back to the original type (or otherwise treat the variable as its original type) before doing anything else with it.

```
//: C03:reinterpret_cast.cpp
#include <iostream>
using namespace std;
const int sz = 100;

struct X { int a[sz]; };

void print(X* x) {
  for(int i = 0; i < sz; i++)
    cout << x->a[i] << ' ';
  cout << endl << "--------------------" << endl;
}

int main() {
  X x;
  print(&x);
  int* xp = reinterpret_cast<int*>(&x);
  for(int* i = xp; i < xp + sz; i++)
    *i = 0;
  // Can't use xp as an X* at this point
  // unless you cast it back:
  print(reinterpret_cast<X*>(xp));
  // In this example, you can also just use
  // the original identifier:
```

```
  print(&x);
} ///:~
```

In this simple example, **struct X** just contains an array of **int**, but when you create one on the stack as in **X x**, the values of each of the **int**s are garbage (this is shown using the **print( )** function to display the contents of the **struct**). To initialize them, the address of the **X** is taken and cast to an **int** pointer, which is then walked through the array to set each **int** to zero. Notice how the upper bound for **i** is calculated by "adding" **sz** to **xp**; the compiler knows that you actually want **sz** pointer locations greater than **xp** and it does the correct pointer arithmetic for you.

The idea of **reinterpret_cast** is that when you use it, what you get is so foreign that it cannot be used for the type's original purpose unless you cast it back. Here, we see the cast back to an **X*** in the call to print, but of course since you still have the original identifier you can also use that. But the **xp** is only useful as an **int***, which is truly a "reinterpretation" of the original **X**.

A **reinterpret_cast** often indicates inadvisable and/or nonportable programming, but it's available when you decide you have to use it.

sizeof – an operator by itself

The **sizeof** operator stands alone because it satisfies an unusual need. **sizeof** gives you information about the amount of memory allocated for data items. As described earlier in this chapter, **sizeof** tells you the number of bytes used by any particular variable. It can also give the size of a data type (with no variable name):

```
//: C03:sizeof.cpp
#include <iostream>
using namespace std;
int main() {
  cout << "sizeof(double) = " << sizeof(double);
  cout << ", sizeof(char) = " << sizeof(char);
} ///:~
```

By definition, the **sizeof** any type of **char** (**signed**, **unsigned** or plain) is always one, regardless of whether the underlying storage for a **char** is actually one byte. For all other types, the result is the size in bytes.

Note that **sizeof** is an operator, not a function. If you apply it to a type, it must be used with the parenthesized form shown above, but if you apply it to a variable you can use it without parentheses:

```
//: C03:sizeofOperator.cpp
int main() {
  int x;
  int i = sizeof x;
} ///:~
```

sizeof can also give you the sizes of user-defined data types. This is used later in the book.

The asm keyword

This is an escape mechanism that allows you to write assembly code for your hardware within a C++ program. Often you're able to reference C++ variables within the assembly code, which means you can easily communicate with your C++ code and limit the assembly code to that necessary for efficiency tuning or to use special processor instructions. The exact syntax that you must use when writing the assembly language is compiler-dependent and can be discovered in your compiler's documentation.

Explicit operators

These are keywords for bitwise and logical operators. Non-U.S. programmers without keyboard characters like **&**, **|**, **^**, and so on, were forced to use C's horrible *trigraphs*, which were not only annoying to type, but obscure when reading. This is repaired in C++ with additional keywords:

Keyword	Meaning
and	**&&** (logical *and*)
or	**||** (logical *or*)
not	**!** (logical NOT)
not_eq	**!=** (logical not-equivalent)
bitand	**&** (bitwise *and*)
and_eq	**&=** (bitwise *and*-assignment)
bitor	**|** (bitwise *or*)
or_eq	**|=** (bitwise or-assignment)
xor	**^** (bitwise exclusive-or)
xor_eq	**^=** (bitwise exclusive-or-assignment)
compl	**~** (ones complement)

If your compiler complies with Standard C++, it will support these keywords.

Composite type creation

The fundamental data types and their variations are essential, but rather primitive. C and C++ provide tools that allow you to compose more sophisticated data types from the fundamental data types. As you'll see, the most important of these is **struct**, which is the foundation for **class** in C++. However, the simplest way to create more sophisticated types is simply to alias a name to another name via **typedef**.

Aliasing names with typedef

This keyword promises more than it delivers: **typedef** suggests "type definition" when "alias" would probably have been a more accurate description, since that's what it really does. The syntax is:

typedef existing-type-description alias-name

People often use **typedef** when data types get slightly complicated, just to prevent extra keystrokes. Here is a commonly-used **typedef**:

```
typedef unsigned long ulong;
```

Now if you say **ulong** the compiler knows that you mean **unsigned long**. You might think that this could as easily be accomplished using preprocessor substitution, but there are key situations in which the compiler must be aware that you're treating a name as if it were a type, so **typedef** is essential.

One place where **typedef** comes in handy is for pointer types. As previously mentioned, if you say:

```
int* x, y;
```

This actually produces an **int*** which is **x** and an **int** (not an **int***) which is **y**. That is, the '*' binds to the right, not the left. However, if you use a **typedef**:

```
typedef int* IntPtr;
IntPtr x, y;
```

Then both **x** and **y** are of type **int***.

You can argue that it's more explicit and therefore more readable to avoid **typedef**s for primitive types, and indeed programs rapidly become difficult to read when many **typedef**s are used. However, **typedef**s become especially important in C when used with **struct**.

Combining variables with struct

A **struct** is a way to collect a group of variables into a structure. Once you create a **struct**, then you can make many instances of this "new" type of variable you've invented. For example:

```
//: C03:SimpleStruct.cpp
struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1, s2;
```

```
  s1.c = 'a'; // Select an element using a '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
style="mso-spacerun: yes">  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

The **struct** declaration must end with a semicolon. In **main( )**, two instances of **Structure1** are created: **s1** and **s2**. Each of these has their own separate versions of **c**, **i**, **f**, and **d**. So **s1** and **s2** represent clumps of completely independent variables. To select one of the elements within **s1** or **s2**, you use a '**.**', syntax you've seen in the previous chapter when using C++ **class** objects – since **class**es evolved from **struct**s, this is where that syntax arose from.

One thing you'll notice is the awkwardness of the use of **Structure1** (as it turns out, this is only required by C, not C++). In C, you can't just say **Structure1** when you're defining variables, you must say **struct Structure1**. This is where **typedef** becomes especially handy in C:

```
//: C03:SimpleStruct2.cpp
// Using typedef with struct
typedef struct {
  char c;
  int i;
  float f;
  double d;
} Structure2;

int main() {
  Structure2 s1, s2;
  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

By using **typedef** in this way, you can pretend (in C; try removing the **typedef** for C++) that **Structure2** is a built-in type, like **int** or **float**, when you define **s1** and **s2** (but notice it only has data – characteristics – and does not include behavior, which is what we get with real objects in C++). You'll notice that the **struct** identifier has been left off at the beginning, because the goal is to create the **typedef**. However, there are times when you might need to refer to the **struct** during its definition. In those cases, you can actually repeat the name of the **struct** as the **struct** name and as the **typedef**:

```
//: C03:SelfReferential.cpp
// Allowing a struct to refer to itself

typedef struct SelfReferential {
  int i;
```

```
  SelfReferential* sr; // Head spinning yet?
} SelfReferential;

int main() {
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
} ///:~
```

If you look at this for awhile, you'll see that **sr1** and **sr2** point to each other, as well as each holding a piece of data.

Actually, the **struct** name does not have to be the same as the **typedef** name, but it is usually done this way as it tends to keep things simpler.

Pointers and structs

In the examples above, all the **struct**s are manipulated as objects. However, like any piece of storage, you can take the address of a **struct** object (as seen in **SelfReferential.cpp** above). To select the elements of a particular **struct** object, you use a '**.**', as seen above. However, if you have a pointer to a **struct** object, you must select an element of that object using a different operator: the '**->**'. Here's an example:

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Point to a different struct object
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
} ///:~
```

In **main( )**, the **struct** pointer **sp** is initially pointing to **s1**, and the members of **s1** are initialized by selecting them with the '**->**' (and you use this same operator in order to read those members). But then **sp** is pointed to **s2**, and those variables are initialized the same way. So you can see that another benefit of pointers is that they can be dynamically redirected to point to different objects; this provides more flexibility in your programming, as you will learn.

For now, that's all you need to know about **struct**s, but you'll become much more comfortable with them (and especially their more potent successors, **class**es) as the book progresses.

Clarifying programs with enum

An enumerated data type is a way of attaching names to numbers, thereby giving more meaning to anyone reading the code. The **enum** keyword (from C) automatically enumerates any list of identifiers you give it by assigning them values of 0, 1, 2, etc. You can declare **enum** variables (which are always represented as integral values). The declaration of an **enum** looks similar to a **struct** declaration.

An enumerated data type is useful when you want to keep track of some sort of feature:

```
//: C03:Enum.cpp
// Keeping track of shapes

enum ShapeType {
  circle,
  square,
  rectangle
};  // Must end with a semicolon like a struct

int main() {
  ShapeType shape = circle;
  // Activities here....
  // Now do something based on what the shape is:
  switch(shape) {
    case circle:  /* circle stuff */ break;
    case square:  /* square stuff */ break;
    case rectangle:  /* rectangle stuff */ break;
  }
} ///:~
```

shape is a variable of the **ShapeType** enumerated data type, and its value is compared with the value in the enumeration. Since **shape** is really just an **int**, however, it can be any value an **int** can hold (including a negative number). You can also compare an **int** variable with a value in the enumeration.

You should be aware that the example above of switching on type turns out to be a problematic way to program. C++ has a much better way to code this sort of thing, the explanation of which must be delayed until much later in the book.

If you don't like the way the compiler assigns values, you can do it yourself, like this:

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

If you give values to some names and not to others, the compiler will use the next integral value. For example,

```
enum snap { crackle = 25, pop };
```

The compiler gives **pop** the value 26.

You can see how much more readable the code is when you use enumerated data types. However, to some degree this is still an attempt (in C) to accomplish the things that we can do with a **class** in C++, so you'll see **enum** used less in C++.

Type checking for enumerations

C's enumerations are fairly primitive, simply associating integral values with names, but they provide no type checking. In C++, as you may have come to expect by now, the concept of type is fundamental, and this is true with enumerations. When you create a named enumeration, you effectively create a new type just as you do with a class: The name of your enumeration becomes a reserved word for the duration of that translation unit.

In addition, there's stricter type checking for enumerations in C++ than in C. You'll notice this in particular if you have an instance of an enumeration **color** called **a**. In C you can say **a++,** but in C++ you can't. This is because incrementing an enumeration is performing two type conversions, one of them legal in C++ and one of them illegal. First, the value of the enumeration is implicitly cast from a **color** to an **int**, then the value is incremented, then the **int** is cast back into a **color**. In C++ this isn't allowed, because **color** is a distinct type and not equivalent to an **int**. This makes sense, because how do you know the increment of **blue** will even be in the list of colors? If you want to increment a **color**, then it should be a class (with an increment operation) and not an **enum**, because the class can be made to be much safer. Any time you write code that assumes an implicit conversion to an **enum** type, the compiler will flag this inherently dangerous activity.

Unions (described next) have similar additional type checking in C++.

Saving memory with union

Sometimes a program will handle different types of data using the same variable. In this situation, you have two choices: you can create a **struct** containing all the possible different types you might need to store, or you can use a **union**. A **union** piles all the data into a single space; it figures out the amount of space necessary for the largest item you've put in the **union**, and makes that the size of the **union**. Use a **union** to save memory.

Anytime you place a value in a **union**, the value always starts in the same place at the beginning of the **union**, but only uses as much space as is necessary. Thus, you create a "super-variable" capable of holding any of the **union** variables. All the addresses of the **union** variables are the same (in a class or **struct**, the addresses are different).

Here's a simple use of a **union**. Try removing various elements and see what effect it has on the size of the **union**. Notice that it makes no sense to declare more than one instance of a single data type in a **union** (unless you're just doing it to use a different name).

```
//: C03:Union.cpp
// The size and simple use of a union
#include <iostream>
using namespace std;
```

```
union Packed { // Declaration similar to a class
  char i;
  short j;
  int k;
  long l;
  float f;
  double d;
  // The union will be the size of a
  // double, since that's the largest element
};  // Semicolon ends a union, like a struct

int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

The compiler performs the proper assignment according to the union member you select.

Once you perform an assignment, the compiler doesn't care what you do with the union. In the example above, you could assign a floating-point value to **x**:

```
x.f = 2.222;
```

and then send it to the output as if it were an **int**:

```
cout << x.i;
```

This would produce garbage.

Arrays

Arrays are a kind of composite type because they allow you to clump a lot of variables together, one right after the other, under a single identifier name. If you say:

```
int a[10];
```

You create storage for 10 **int** variables stacked on top of each other, but without unique identifier names for each variable. Instead, they are all lumped under the name **a**.

To access one of these *array elements*, you use the same square-bracket syntax that you use to define an array:

```
a[5] = 47;
```

However, you must remember that even though the *size* of **a** is 10, you select array elements starting at zero (this is sometimes called *zero indexing*), so you can select only the array elements 0-9, like this:

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

Array access is extremely fast. However, if you index past the end of the array, there is no safety net – you'll step on other variables. The other drawback is that you must define the size of the array at compile time; if you want to change the size at runtime you can't do it with the syntax above (C does have a way to create an array dynamically, but it's significantly messier). The C++ **vector**, introduced in the previous chapter, provides an array-like object that automatically resizes itself, so it is usually a much better solution if your array size cannot be known at compile time.

You can make an array of any type, even of **struct**s:

```
//: C03:StructArray.cpp
// An array of struct

typedef struct {
  int i, j, k;
} ThreeDpoint;

int main() {
  ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

Notice how the **struct** identifier **i** is independent of the **for** loop's **i**.

To see that each element of an array is contiguous with the next, you can print out the addresses like this:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = "<< sizeof(int) << endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
         << (long)&a[i] << endl;
} ///:~
```

When you run this program, you'll see that each element is one **int** size away from the previous one. That is, they are stacked one on top of the other.

Pointers and arrays

The identifier of an array is unlike the identifiers for ordinary variables. For one thing, an array identifier is not an lvalue; you cannot assign to it. It's really just a hook into the square-bracket syntax, and when you give the name of an array, without square brackets, what you get is the starting address of the array:

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

When you run this program you'll see that the two addresses (which will be printed in hexadecimal, since there is no cast to **long**) are the same.

So one way to look at the array identifier is as a read-only pointer to the beginning of an array. And although we can't change the array identifier to point somewhere else, we *can* create another pointer and use that to move around in the array. In fact, the square-bracket syntax works with regular pointers as well:

```
//: C03:PointersAndBrackets.cpp
int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

The fact that naming an array produces its starting address turns out to be quite important when you want to pass an array to a function. If you declare an array as a function argument, what you're really declaring is a pointer. So in the following example, **func1( )** and **func2( )** effectively have the same argument lists:

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}

void func2(int* a, int size) {
```

```
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}

void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
    cout << name << "[" << i << "] = "
         << a[i] << endl;
}

int main() {
  int a[5], b[5];
  // Probably garbage values:
  print(a, "a", 5);
  print(b, "b", 5);
  // Initialize the arrays:
  func1(a, 5);
  func1(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
  // Notice the arrays are always modified:
  func2(a, 5);
  func2(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
} ///:~
```

Even though **func1( )** and **func2( )** declare their arguments differently, the usage is the same inside the function. There are some other issues that this example reveals: arrays cannot be passed by value[32], that is, you never automatically get a local copy of the array that you pass into a function. Thus, when you modify an array, you're always modifying the outside object. This can be a bit confusing at first, if you're expecting the pass-by-value provided with ordinary arguments.

You'll notice that **print( )** uses the square-bracket syntax for array arguments. Even though the pointer syntax and the square-bracket syntax are effectively the same when passing arrays as arguments, the square-bracket syntax makes it clearer to the reader that you mean for this argument to be an array.

Also note that the **size** argument is passed in each case. Just passing the address of an array isn't enough information; you must always be able to know how big the array is inside your function, so you don't run off the end of that array.

Arrays can be of any type, including arrays of pointers. In fact, when you want to pass command-line arguments into your program, C and C++ have a special argument list for **main( )**, which looks like this:

```
int main(int argc, char* argv[]) { // ...
```

The first argument is the number of elements in the array, which is the second argument. The second argument is always an array of **char***, because the arguments are passed from the command line as character arrays (and remember, an array can be passed only as a pointer). Each whitespace-delimited cluster of characters on the command line is turned into a

separate array argument. The following program prints out all its command-line arguments by stepping through the array:

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
         << argv[i] << endl;
} ///:~
```

You'll notice that **argv[0]** is the path and name of the program itself. This allows the program to discover information about itself. It also adds one more to the array of program arguments, so a common error when fetching command-line arguments is to grab **argv[0]** when you want **argv[1]**.

You are not forced to use **argc** and **argv** as identifiers in **main( )**; those identifiers are only conventions (but it will confuse people if you don't use them). Also, there is an alternate way to declare **argv**:

```
int main(int argc, char** argv) { // ...
```

Both forms are equivalent, but I find the version used in this book to be the most intuitive when reading the code, since it says, directly, "This is an array of character pointers."

All you get from the command-line is character arrays; if you want to treat an argument as some other type, you are responsible for converting it inside your program. To facilitate the conversion to numbers, there are some helper functions in the Standard C library, declared in **<cstdlib>**. The simplest ones to use are **atoi( )**, **atol( ),** and **atof( )** to convert an ASCII character array to an **int**, **long,** and **double** floating-point value, respectively. Here's an example using **atoi( )** (the other two functions are called the same way):

```
//: C03:ArgsToInts.cpp
// Converting command-line arguments to ints
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

In this program, you can put any number of arguments on the command line. You'll notice that the **for** loop starts at the value **1** to skip over the program name at **argv[0]**. Also, if you put a floating-point number containing a decimal point on the command line, **atoi( )** takes only the digits up to the decimal point. If you put non-numbers on the command line, these come back from **atoi( )** as zero.

Exploring floating-point format

The **printBinary( )** function introduced earlier in this chapter is handy for delving into the internal structure of various data types. The most interesting of these is the floating-point format that allows C and C++ to store numbers representing very large and very small values in a limited amount of space. Although the details can't be completely exposed here, the bits inside of **float**s and **double**s are divided into three regions: the exponent, the mantissa, and the sign bit; thus it stores the values using scientific notation. The following program allows you to play around by printing out the binary patterns of various floating point numbers so you can deduce for yourself the scheme used in your compiler's floating-point format (usually this is the IEEE standard for floating point numbers, but your compiler may not follow that):

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cout << "Must provide a number" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  unsigned char* cp =
    reinterpret_cast<unsigned char*>(&d);
  for(int i = sizeof(double); i > 0 ; i -= 2) {
    printBinary(cp[i-1]);
    printBinary(cp[i]);
  }
} ///:~
```

First, the program guarantees that you've given it an argument by checking the value of **argc**, which is two if there's a single argument (it's one if there are no arguments, since the program name is always the first element of **argv**). If this fails, a message is printed and the Standard C Library function **exit( )** is called to terminate the program.

The program grabs the argument from the command line and converts the characters to a **double** using **atof( )**. Then the double is treated as an array of bytes by taking the address and casting it to an **unsigned char***. Each of these bytes is passed to **printBinary( )** for display.

This example has been set up to print the bytes in an order such that the sign bit appears first – on my machine. Yours may be different, so you might want to re-arrange the way things are printed. You should also be aware that floating-point formats are not trivial to understand; for example, the exponent and mantissa are not generally arranged on byte boundaries, but instead a number of bits is reserved for each one and they are packed into the memory as tightly as possible. To truly see what's going on, you'd need to find out the size of each part of the number (sign bits are always one bit, but exponents and mantissas are of differing sizes) and print out the bits in each part separately.

Pointer arithmetic

If all you could do with a pointer that points at an array is treat it as if it were an alias for that array, pointers into arrays wouldn't be very interesting. However, pointers are more flexible than this, since they can be modified to point somewhere else (but remember, the array identifier cannot be modified to point somewhere else).

Pointer arithmetic refers to the application of some of the arithmetic operators to pointers. The reason pointer arithmetic is a separate subject from ordinary arithmetic is that pointers must conform to special constraints in order to make them behave properly. For example, a common operator to use with pointers is ++, which "adds one to the pointer." What this actually means is that the pointer is changed to move to "the next value," whatever that means. Here's an example:

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
  cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
  cout << "dp = " << (long)dp << endl;
} ///:~
```

For one run on my machine, the output is:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

What's interesting here is that even though the operation ++ appears to be the same operation for both the **int*** and the **double***, you can see that the pointer has been changed only 4 bytes for the **int*** but 8 bytes for the **double***. Not coincidentally, these are the sizes of **int** and **double** on my machine. And that's the trick of pointer arithmetic: the compiler figures out the right amount to change the pointer so that it's pointing to the next element in the array (pointer arithmetic is only meaningful within arrays). This even works with arrays of **struct**s:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
```

```
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;

int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
       << sizeof(Primitives) << endl;
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

The output for one run on my machine was:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

So you can see the compiler also does the right thing for pointers to **struct**s (and **class**es and **union**s).

Pointer arithmetic also works with the operators **--**, **+**, and **-**, but the latter two operators are limited: you cannot add two pointers, and if you subtract pointers the result is the number of elements between the two pointers. However, you can add or subtract an integral value and a pointer. Here's an example demonstrating the use of pointer arithmetic:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EX) cout << #EX << ": " << EX << endl;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++)
    a[i] = i; // Give it index values
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
  P(ip2 - ip); // Yields number of elements
} ///:~
```

It begins with another macro, but this one uses a preprocessor feature called *stringizing* (implemented with the '**#**' sign before an expression) that takes any expression and turns it

into a character array. This is quite convenient, since it allows the expression to be printed, followed by a colon, followed by the value of the expression. In **main( )** you can see the useful shorthand that is produced.

Although pre- and postfix versions of **++** and **--** are valid with pointers, only the prefix versions are used in this example because they are applied before the pointers are dereferenced in the expressions above, so they allow us to see the effects of the operations. Note that only integral values are being added and subtracted; if two pointers were combined this way the compiler would not allow it.

Here is the output of the program above:

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

In all cases, the pointer arithmetic results in the pointer being adjusted to point to the "right place," based on the size of the elements being pointed to.

If pointer arithmetic seems a bit overwhelming at first, don't worry. Most of the time you'll only need to create arrays and index into them with **[ ]**, and the most sophisticated pointer arithmetic you'll usually need is **++** and **--**. Pointer arithmetic is generally reserved for more clever and complex programs, and many of the containers in the Standard C++ library hide most of these clever details so you don't have to worry about them.

Debugging hints

In an ideal environment, you have an excellent debugger available that easily makes the behavior of your program transparent so you can quickly discover errors. However, most debuggers have blind spots, and these will require you to embed code snippets in your program to help you understand what's going on. In addition, you may be developing in an environment (such as an embedded system, which is where I spent my formative years) that has no debugger available, and perhaps very limited feedback (i.e. a one-line LED display). In these cases you become creative in the ways you discover and display information about the execution of your program. This section suggests some techniques for doing this.

Debugging flags

If you hard-wire debugging code into a program, you can run into problems. You start to get too much information, which makes the bugs difficult to isolate. When you think you've found the bug you start tearing out debugging code, only to find you need to put it back in again. You can solve these problems with two types of flags: preprocessor debugging flags and runtime debugging flags.

Preprocessor debugging flags

By using the preprocessor to **#define** one or more debugging flags (preferably in a header file), you can test a flag using an **#ifdef** statement and conditionally include debugging code. When you think your debugging is finished, you can simply **#undef** the flag(s) and the code will automatically be removed (and you'll reduce the size and runtime overhead of your executable file).

It is best to decide on names for debugging flags before you begin building your project so the names will be consistent. Preprocessor flags are traditionally distinguished from variables by writing them in all upper case. A common flag name is simply **DEBUG** (but be careful you don't use **NDEBUG**, which is reserved in C). The sequence of statements might be:

```
#define DEBUG // Probably in a header file
//...
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```

Most C and C++ implementations will also let you **#define** and **#undef** flags from the compiler command line, so you can re-compile code and insert debugging information with a single command (preferably via the makefile, a tool that will be described shortly). Check your local documentation for details.

Runtime debugging flags

In some situations it is more convenient to turn debugging flags on and off during program execution, especially by setting them when the program starts up using the command line. Large programs are tedious to recompile just to insert debugging code.

To turn debugging code on and off dynamically, create **bool** flags:

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Debug flags aren't necessarily global:
bool debug = false;

int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {
      // Debugging code here
      cout << "Debugger is now on!" << endl;
    } else {
      cout << "Debugger is now off." << endl;
    }
    cout << "Turn debugger [on/off/quit]: ";
    string reply;
```

```
    cin >> reply;
    if(reply == "on") debug = true; // Turn it on
    if(reply == "off") debug = false; // Off
    if(reply == "quit") break; // Out of 'while'
  }
} ///:~
```

This program continues to allow you to turn the debugging flag on and off until you type "quit" to tell it you want to exit. Notice it requires that full words are typed in, not just letters (you can shorten it to letter if you wish). Also, a command-line argument can optionally be used to turn debugging on at startup – this argument can appear anyplace in the command line, since the startup code in **main( )** looks at all the arguments. The testing is quite simple because of the expression:

```
string(argv[i])
```

This takes the **argv[i]** character array and creates a **string**, which then can be easily compared to the right-hand side of the **==**. The program above searches for the entire string **--debug=on**. You can also look for **--debug=** and then see what's after that, to provide more options. Volume 2 (available from *www.BruceEckel.com*) devotes a chapter to the Standard C++ **string** class.

Although a debugging flag is one of the relatively few areas where it makes a lot of sense to use a global variable, there's nothing that says it must be that way. Notice that the variable is in lower case letters to remind the reader it isn't a preprocessor flag.

Turning variables and expressions into strings

When writing debugging code, it is tedious to write print expressions consisting of a character array containing the variable name, followed by the variable. Fortunately, Standard C includes the *stringize* operator '#', which was used earlier in this chapter. When you put a # before an argument in a preprocessor macro, the preprocessor turns that argument into a character array. This, combined with the fact that character arrays with no intervening punctuation are concatenated into a single character array, allows you to make a very convenient macro for printing the values of variables during debugging:

```
#define PR(x) cout << #x " = " << x << "\n";
```

If you print the variable **a** by calling the macro **PR(a)**, it will have the same effect as the code:

```
cout << "a = " << a << "\n";
```

This same process works with entire expressions. The following program uses a macro to create a shorthand that prints the stringized expression and then evaluates the expression and prints the result:

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;
```

```
int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

You can see how a technique like this can quickly become indispensable, especially if you have no debugger (or must use multiple development environments). You can also insert an **#ifdef** to cause **P(A)** to be defined as "nothing" when you want to strip out debugging.

The C assert( ) macro

In the standard header file **<cassert>** you'll find **assert( )**, which is a convenient debugging macro. When you use **assert( )**, you give it an argument that is an expression you are "asserting to be true." The preprocessor generates code that will test the assertion. If the assertion isn't true, the program will stop after issuing an error message telling you what the assertion was and that it failed. Here's a trivial example:

```
//: C03:Assert.cpp
// Use of the assert() debugging macro
#include <cassert>  // Contains the macro
using namespace std;

int main() {
  int i = 100;
  assert(i != 100); // Fails
} ///:~
```

The macro originated in Standard C, so it's also available in the header file **assert.h**.

When you are finished debugging, you can remove the code generated by the macro by placing the line:

```
#define NDEBUG
```

in the program before the inclusion of **<cassert>**, or by defining NDEBUG on the compiler command line. NDEBUG is a flag used in **<cassert>** to change the way code is generated by the macros.

Later in this book, you'll see some more sophisticated alternatives to **assert( )**.

Function addresses

Once a function is compiled and loaded into the computer to be executed, it occupies a chunk of memory. That memory, and thus the function, has an address.

C has never been a language to bar entry where others fear to tread. You can use function addresses with pointers just as you can use variable addresses. The declaration and use of function pointers looks a bit opaque at first, but it follows the format of the rest of the language.

Defining a function pointer

To define a pointer to a function that has no arguments and no return value, you say:

```
void (*funcPtr)();
```

When you are looking at a complex definition like this, the best way to attack it is to start in the middle and work your way out. “Starting in the middle” means starting at the variable name, which is **funcPtr**. “Working your way out” means looking to the right for the nearest item (nothing in this case; the right parenthesis stops you short), then looking to the left (a pointer denoted by the asterisk), then looking to the right (an empty argument list indicating a function that takes no arguments), then looking to the left (**void,** which indicates the function has no return value). This right-left-right motion works with most declarations.

To review, “start in the middle” (“**funcPtr** is a ...”), go to the right (nothing there – you're stopped by the right parenthesis), go to the left and find the ‘*’ (“... pointer to a ...”), go to the right and find the empty argument list (“... function that takes no arguments ... ”), go to the left and find the **void** (“**funcPtr** is a pointer to a function that takes no arguments and returns **void**”).

You may wonder why ***funcPtr** requires parentheses. If you didn't use them, the compiler would see:

```
void *funcPtr();
```

You would be declaring a function (that returns a **void***) rather than defining a variable. You can think of the compiler as going through the same process you do when it figures out what a declaration or definition is supposed to be. It needs those parentheses to “bump up against” so it goes back to the left and finds the ‘*’, instead of continuing to the right and finding the empty argument list.

Complicated declarations & definitions

As an aside, once you figure out how the C and C++ declaration syntax works you can create much more complicated items. For instance:

```
//: C03:ComplicatedDefinitions.cpp

/* 1. */     void * (*(*fp1)(int))[10];

/* 2. */     float (*(*fp2)(int,int,float))(int);
```

```
/* 3. */      typedef double (*(*(*fp3)())[10])();
              fp3 a;

/* 4. */      int (*(*f4())[10])();

int main() {} ///:~
```

Walk through each one and use the right-left guideline to figure it out. Number 1 says "**fp1** is a pointer to a function that takes an integer argument and returns a pointer to an array of 10 **void** pointers."

Number 2 says "**fp2** is a pointer to a function that takes three arguments (**int**, **int,** and **float**) and returns a pointer to a function that takes an integer argument and returns a **float**."

If you are creating a lot of complicated definitions, you might want to use a **typedef**. Number 3 shows how a **typedef** saves typing the complicated description every time. It says "An **fp3** is a pointer to a function that takes no arguments and returns a pointer to an array of 10 pointers to functions that take no arguments and return doubles." Then it says "**a** is one of these **fp3** types." **typedef** is generally useful for building complicated descriptions from simple ones.

Number 4 is a function declaration instead of a variable definition. It says "**f4** is a function that returns a pointer to an array of 10 pointers to functions that return integers."

You will rarely if ever need such complicated declarations and definitions as these. However, if you go through the exercise of figuring them out you will not even be mildly disturbed with the slightly complicated ones you may encounter in real life.

Using a function pointer

Once you define a pointer to a function, you must assign it to a function address before you can use it. Just as the address of an array **arr[10]** is produced by the array name without the brackets (**arr**), the address of a function **func()** is produced by the function name without the argument list (**func**). You can also use the more explicit syntax **&func()**. To call the function, you dereference the pointer in the same way that you declared it (remember that C and C++ always try to make definitions look the same as the way they are used). The following example shows how a pointer to a function is defined and used:

```
//: C03:PointerToFunction.cpp
// Defining and using a pointer to a function
#include <iostream>
using namespace std;

void func() {
  cout << "func() called..." << endl;
}

int main() {
  void (*fp)();  // Define a function pointer
```

```
  fp = func;  // Initialize it
  (*fp)();    // Dereferencing calls the function
  void (*fp2)() = func;  // Define and initialize
  (*fp2)();
} ///:~
```

After the pointer to function **fp** is defined, it is assigned to the address of a function **func()** using **fp = func** (notice the argument list is missing on the function name). The second case shows simultaneous definition and initialization.

Arrays of pointers to functions

One of the more interesting constructs you can create is an array of pointers to functions. To select a function, you just index into the array and dereference the pointer. This supports the concept of *table-driven code*; instead of using conditionals or case statements, you select functions to execute based on a state variable (or a combination of state variables). This kind of design can be useful if you often add or delete functions from the table (or if you want to create or change such a table dynamically).

The following example creates some dummy functions using a preprocessor macro, then creates an array of pointers to those functions using automatic aggregate initialization. As you can see, it is easy to add or remove functions from the table (and thus, functionality from the program) by changing a small amount of code:

```
//: C03:FunctionTable.cpp
// Using an array of pointers to functions
#include <iostream>
using namespace std;

// A macro to define dummy functions:
#define DF(N) void N() { \
   cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };

int main() {
  while(1) {
    cout << "press a key from 'a' to 'g' "
      "or q to quit" << endl;
    char c, cr;
    cin.get(c); cin.get(cr); // second one for CR
    if ( c == 'q' )
      break; // ... out of while(1)
    if ( c < 'a' || c > 'g' )
      continue;
    (*func_table[c - 'a'])();
  }
} ///:~
```

At this point, you might be able to imagine how this technique could be useful when creating some sort of interpreter or list processing program.

Make: managing separate compilation

When using *separate compilation* (breaking code into a number of translation units), you need some way to automatically compile each file and to tell the linker to build all the pieces – along with the appropriate libraries and startup code – into an executable file. Most compilers allow you to do this with a single command-line statement. For the GNU C++ compiler, for example, you might say

```
g++ SourceFile1.cpp SourceFile2.cpp
```

The problem with this approach is that the compiler will first compile each individual file, regardless of whether that file *needs* to be rebuilt or not. With many files in a project, it can become prohibitive to recompile everything if you've changed only a single file.

The solution to this problem, developed on Unix but available everywhere in some form, is a program called **make**. The **make** utility manages all the individual files in a project by following the instructions in a text file called a **makefile**. When you edit some of the files in a project and type **make**, the **make** program follows the guidelines in the **makefile** to compare the dates on the source code files to the dates on the corresponding target files, and if a source code file date is more recent than its target file, **make** invokes the compiler on the source code file. **make** only recompiles the source code files that were changed, and any other source-code files that are affected by the modified files. By using **make**, you don't have to re-compile all the files in your project every time you make a change, nor do you have to check to see that everything was built properly. The **makefile** contains all the commands to put your project together. Learning to use **make** will save you a lot of time and frustration. You'll also discover that **make** is the typical way that you install new software on a Linux/Unix machine (although those **makefile**s tend to be far more complicated than the ones presented in this book, and you'll often automatically generate a **makefile** for your particular machine as part of the installation process).

Because **make** is available in some form for virtually all C++ compilers (and even if it isn't, you can use freely-available **make**s with any compiler), it will be the tool used throughout this book. However, compiler vendors have also created their own project building tools. These tools ask you which files are in your project and determine all the relationships themselves. These tools use something similar to a **makefile**, generally called a *project file*, but the programming environment maintains this file so you don't have to worry about it. The configuration and use of project files varies from one development environment to another, so you must find the appropriate documentation on how to use them (although project file tools provided by compiler vendors are usually so simple to use that you can learn them by playing around – my favorite form of education).

The **makefile**s used within this book should work even if you are also using a specific vendor's project-building tool.

Make activities

When you type **make** (or whatever the name of your "make" program happens to be), the **make** program looks in the current directory for a file named **makefile**, which you've created if it's your project. This file lists dependencies between source code files. **make** looks at the dates on files. If a dependent file has an older date than a file it depends on, **make** executes the *rule* given after the dependency.

All comments in **makefile**s start with a **#** and continue to the end of the line.

As a simple example, the **makefile** for a program called "hello" might contain:

```
# A comment
hello.exe: hello.cpp
        mycompiler hello.cpp
```

This says that **hello.exe** (the target) depends on **hello.cpp**. When **hello.cpp** has a newer date than **hello.exe**, **make** executes the "rule" **mycompiler hello.cpp**. There may be multiple dependencies and multiple rules. Many **make** programs require that all the rules begin with a tab. Other than that, whitespace is generally ignored so you can format for readability.

The rules are not restricted to being calls to the compiler; you can call any program you want from within **make**. By creating groups of interdependent dependency-rule sets, you can modify your source code files, type **make** and be certain that all the affected files will be rebuilt correctly.

Macros

A **makefile** may contain *macros* (note that these are completely different from C/C++ preprocessor macros). Macros allow convenient string replacement. The **makefile**s in this book use a macro to invoke the C++ compiler. For example,

```
CPP = mycompiler
hello.exe: hello.cpp
        $(CPP) hello.cpp
```

The **=** is used to identify **CPP** as a macro, and the **$** and parentheses expand the macro. In this case, the expansion means that the macro call **$(CPP)** will be replaced with the string **mycompiler**. With the macro above, if you want to change to a different compiler called **cpp**, you just change the macro to:

```
CPP = cpp
```

You can also add compiler flags, etc., to the macro, or use separate macros to add compiler flags.

Suffix Rules

It becomes tedious to tell **make** how to invoke the compiler for every single **cpp** file in your project, when you know it's the same basic process each time. Since **make** is designed to be a time-saver, it also has a way to abbreviate actions, as long as they depend on file name suffixes. These abbreviations are called *suffix rules*. A suffix rule is the way to teach **make** how to convert a file with one type of extension (**.cpp**, for example) into a file with another type of extension (**.obj** or **.exe**). Once you teach **make** the rules for producing one kind of file from another, all you have to do is tell **make** which files depend on which other files. When **make** finds a file with a date earlier than the file it depends on, it uses the rule to create a new file.

The suffix rule tells **make** that it doesn't need explicit rules to build everything, but instead it can figure out how to build things based on their file extension. In this case it says "To build a file that ends in **exe** from one that ends in **cpp**, invoke the following command." Here's what it looks like for the example above:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
```

The **.SUFFIXES** directive tells **make** that it should watch out for any of the following file-name extensions because they have special meaning for this particular makefile. Next you see the suffix rule **.cpp.exe,** which says "Here's how to convert any file with an extension of **cpp** to one with an extension of **exe**" (when the **cpp** file is more recent than the **exe** file). As before, the **$(CPP)** macro is used, but then you see something new: **$<**. Because this begins with a '**$**' it's a macro, but this is one of **make**'s special built-in macros. The **$<** can be used only in suffix rules, and it means "whatever prerequisite triggered the rule" (sometimes called the *dependent*), which in this case translates to "the **cpp** file that needs to be compiled."

Once the suffix rules have been set up, you can simply say, for example, "**make Union.exe**," and the suffix rule will kick in, even though there's no mention of "Union" anywhere in the **makefile**.

Default targets

After the macros and suffix rules, **make** looks for the first "target" in a file, and builds that, unless you specify differently. So for the following **makefile**:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
target1.exe:
target2.exe:
```

If you just type '**make**', then **target1.exe** will be built (using the default suffix rule) because that's the first target that **make** encounters. To build **target2.exe** you'd have to explicitly say '**make target2.exe**'. This becomes tedious, so you normally create a default "dummy" target that depends on all the rest of the targets, like this:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
all: target1.exe target2.exe
```

Here, '**all**' does not exist and there's no file called '**all**', so every time you type **make**, the program sees '**all**' as the first target in the list (and thus the default target), then it sees that '**all**' does not exist so it had better make it by checking all the dependencies. So it looks at **target1.exe** and (using the suffix rule) sees whether (1) **target1.exe** exists and (2) whether **target1.cpp** is more recent than **target1.exe**, and if so runs the suffix rule (if you provide an explicit rule for a particular target, that rule is used instead). Then it moves on to the next file in the default target list. Thus, by creating a default target list (typically called '**all**' by convention, but you can call it anything) you can cause every executable in your project to be made simply by typing '**make**'. In addition, you can have other non-default target lists that do other things – for example, you could set it up so that typing '**make debug**' rebuilds all your files with debugging wired in.

Makefiles in this book

Using the program **ExtractCode.cpp** from Volume 2 of this book, all the code listings in this book are automatically extracted from the ASCII text version of this book and placed in subdirectories according to their chapters. In addition, **ExtractCode.cpp** creates several **makefile**s in each subdirectory (with different names) so you can simply move into that subdirectory and type **make -f mycompiler.makefile** (substituting the name of your compiler for '**mycompiler**', the '**-f**' flag says "use what follows as the **makefile**"). Finally, **ExtractCode.cpp** creates a "master" **makefile** in the root directory where the book's files have been expanded, and this **makefile** descends into each subdirectory and calls **make** with the appropriate **makefile**. This way you can compile all the code in the book by invoking a single **make** command, and the process will stop whenever your compiler is unable to handle a particular file (note that a Standard C++ conforming compiler should be able to compile all the files in this book). Because implementations of **make** vary from system to system, only the most basic, common features are used in the generated **makefile**s.

An example makefile

As mentioned, the code-extraction tool **ExtractCode.cpp** automatically generates **makefile**s for each chapter. Because of this, the **makefile**s for each chapter will not be placed in the book (all the makefiles are packaged with the source code, which you can download from *www.BruceEckel.com*). However, it's useful to see an example of a **makefile**. What follows is a shortened version of the one that was automatically generated for this chapter by the book's extraction tool. You'll find more than one **makefile** in each subdirectory (they have different names; you invoke a specific one with '**make -f**'). This one is for GNU C++:

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
  $(CPP) $(CPPFLAGS) -c $<
.c.o :
```

```
  $(CPP) $(CPPFLAGS) -c $<

all: \
  Return \
  Declare \
  Ifthen \
  Guess \
  Guess2
# Rest of the files for this chapter not shown

Return: Return.o
  $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
  $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
  $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
  $(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
  $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

The macro CPP is set to the name of the compiler. To use a different compiler, you can either edit the **makefile** or change the value of the macro on the command line, like this:

```
make CPP=cpp
```

Note, however, that **ExtractCode.cpp** has an automatic scheme to automatically build **makefile**s for additional compilers.

The second macro **OFLAG** is the flag that's used to indicate the name of the output file. Although many compilers automatically assume the output file has the same base name as the input file, others don't (such as Linux/Unix compilers, which default to creating a file called **a.out**).

You can see that there are two suffix rules here, one for **cpp** files and one for **.c** files (in case any C source code needs to be compiled). The default target is **all**, and each line for this target is "continued" by using the backslash, up until **Guess2**, which is the last one in the list and thus has no backslash. There are many more files in this chapter, but only these are shown here for the sake of brevity.

The suffix rules take care of creating object files (with a **.o** extension) from **cpp** files, but in general you need to explicitly state rules for creating the executable, because normally an executable is created by linking many different object files and **make** cannot guess what those are. Also, in this case (Linux/Unix) there is no standard extension for executables so a suffix

rule won't work for these simple situations. Thus, you see all the rules for building the final executables explicitly stated.

This **makefile** takes the absolute safest route of using as few **make** features as possible; it only uses the basic **make** concepts of targets and dependencies, as well as macros. This way it is virtually assured of working with as many **make** programs as possible. It tends to produce a larger **makefile**, but that's not so bad since it's automatically generated by **ExtractCode.cpp**.

There are lots of other **make** features that this book will not use, as well as newer and cleverer versions and variations of **make** with advanced shortcuts that can save a lot of time. Your local documentation may describe the further features of your particular **make**, and you can learn more about **make** from *Managing Projects with Make* by Oram and Talbott (O'Reilly, 1993). Also, if your compiler vendor does not supply a **make** or it uses a non-standard **make**, you can find GNU make for virtually any platform in existence by searching the Internet for GNU archives (of which there are many).

Summary

This chapter was a fairly intense tour through all the fundamental features of C++ syntax, most of which are inherited from and in common with C (and result in C++'s vaunted backwards compatibility with C). Although some C++ features were introduced here, this tour is primarily intended for people who are conversant in programming, and simply need to be given an introduction to the syntax basics of C and C++. If you're already a C programmer, you may have even seen one or two things about C here that were unfamiliar, aside from the C++ features that were most likely new to you. However, if this chapter has still seemed a bit overwhelming, you should go through the CD ROM course *Thinking in C: Foundations for C++ and Java* (which contains lectures, exercises, and guided solutions), which is bound into this book, and also available at *www.BruceEckel.com*.

Exercises

Solutions to selected exercises can be found in the electronic document *The Thinking in C++ Annotated Solution Guide*, available for a small fee from www.BruceEckel.com.

1. Create a header file (with an extension of '**.h**'). In this file, declare a group of functions by varying the argument lists and return values from among the following: **void**, **char**, **int**, and **float**. Now create a **.cpp** file that includes your header file and creates definitions for all of these functions. Each definition should simply print out the function name, argument list, and return type so you know it's been called. Create a second **.cpp** file that includes your header file and defines **int main( )**, containing calls to all of your functions. Compile and run your program.
2. Write a program that uses two nested **for** loops and the modulus operator (**%**) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1).
3. Write a program that uses a **while** loop to read words from standard input (**cin**) into a **string**. This is an "infinite" **while** loop, which you break out of (and exit the program) using a **break** statement. For each word that is read, evaluate it by first using a

sequence of **if** statements to "map" an integral value to the word, and then use a **switch** statement that uses that integral value as its selector (this sequence of events is not meant to be good programming style; it's just supposed to give you exercise with control flow). Inside each **case**, print something meaningful. You must decide what the "interesting" words are and what the meaning is. You must also decide what word will signal the end of the program. Test the program by redirecting a file into the program's standard input (if you want to save typing, this file can be your program's source file).

4. Modify **Menu.cpp** to use **switch** statements instead of **if** statements.
5. Write a program that evaluates the two expressions in the section labeled "precedence."
6. Modify **YourPets2.cpp** so that it uses various different data types (**char**, **int**, **float**, **double,** and their variants). Run the program and create a map of the resulting memory layout. If you have access to more than one kind of machine, operating system, or compiler, try this experiment with as many variations as you can manage.
7. Create two functions, one that takes a **string*** and one that takes a **string&**. Each of these functions should modify the outside **string** object in its own unique way. In **main( )**, create and initialize a **string** object, print it, then pass it to each of the two functions, printing the results.
8. Write a program that uses all the trigraphs to see if your compiler supports them.
9. Compile and run **Static.cpp**. Remove the **static** keyword from the code, compile and run it again, and explain what happens.
10. Try to compile and link **FileStatic.cpp** with **FileStatic2.cpp**. What does the resulting error message mean?
11. Modify **Boolean.cpp** so that it works with **double** values instead of **int**s.
12. Modify **Boolean.cpp** and **Bitwise.cpp** so they use the explicit operators (if your compiler is conformant to the C++ Standard it will support these).
13. Modify **Bitwise.cpp** to use the functions from **Rotation.cpp**. Make sure you display the results in such a way that it's clear what's happening during rotations.
14. Modify **Ifthen.cpp** to use the ternary **if-else** operator (**?:**).
15. Create a **struct** that holds two **string** objects and one **int**. Use a **typedef** for the **struct** name. Create an instance of the **struct**, initialize all three values in your instance, and print them out. Take the address of your instance and assign it to a pointer to your **struct** type. Change the three values in your instance and print them out, all using the pointer.
16. Create a program that uses an enumeration of colors. Create a variable of this **enum** type and print out all the numbers that correspond with the color names, using a **for** loop.
17. Experiment with **Union.cpp** by removing various **union** elements to see the effects on the size of the resulting **union**. Try assigning to one element (thus one type) of the **union** and printing out a via a different element (thus a different type) to see what happens.
18. Create a program that defines two **int** arrays, one right after the other. Index off the end of the first array into the second, and make an assignment. Print out the second array to see the changes cause by this. Now try defining a **char** variable between the first array definition and the second, and repeat the experiment. You may want to create an array printing function to simplify your coding.
19. Modify **ArrayAddresses.cpp** to work with the data types **char**, **long int**, **float,** and **double**.
20. Apply the technique in **ArrayAddresses.cpp** to print out the size of the **struct** and the addresses of the array elements in **StructArray.cpp**.

21. Create an array of **string** objects and assign a string to each element. Print out the array using a **for** loop.
22. Create two new programs starting from **ArgsToInts.cpp** so they use **atol( )** and **atof( )**, respectively.
23. Modify **PointerIncrement2.cpp** so it uses a **union** instead of a **struct**.
24. Modify **PointerArithmetic.cpp** to work with **long** and **long double**.
25. Define a **float** variable. Take its address, cast that address to an **unsigned char**, and assign it to an **unsigned char** pointer. Using this pointer and **[ ]**, index into the **float** variable and use the **printBinary( )** function defined in this chapter to print out a map of the **float** (go from 0 to **sizeof(float)**). Change the value of the **float** and see if you can figure out what's going on (the **float** contains encoded data).
26. Define an array of **int**. Take the starting address of that array and use **static_cast** to convert it into an **void***. Write a function that takes a **void***, a number (indicating a number of bytes), and a value (indicating the value to which each byte should be set) as arguments. The function should set each byte in the specified range to the specified value. Try out the function on your array of **int**.
27. Create a **const** array of **double** and a **volatile** array of **double**. Index through each array and use **const_cast** to cast each element to non-**const** and non-**volatile**, respectively, and assign a value to each element.
28. Create a function that takes a pointer to an array of **double** and a value indicating the size of that array. The function should print each element in the array. Now create an array of **double** and initialize each element to zero, then use your function to print the array. Next use **reinterpret_cast** to cast the starting address of your array to an **unsigned char***, and set each byte of the array to 1 (hint: you'll need to use **sizeof** to calculate the number of bytes in a **double**). Now use your array-printing function to print the results. Why do you think each element was not set to the value 1.0?
29. (Challenging) Modify **FloatingAsBinary.cpp** so that it prints out each part of the **double** as a separate group of bits. You'll have to replace the calls to **printBinary( )** with your own specialized code (which you can derive from **printBinary( )**) in order to do this, and you'll also have to look up and understand the floating-point format along with the byte ordering for your compiler (this is the challenging part).
30. Create a makefile that not only compiles **YourPets1.cpp** and **YourPets2.cpp** (for your particular compiler) but also executes both programs as part of the default target behavior. Make sure you use suffix rules.
31. Modify **StringizingExpressions.cpp** so that **P(A)** is conditionally **#ifdef**ed to allow the debugging code to be automatically stripped out by setting a command-line flag. You will need to consult your compiler's documentation to see how to define and undefine preprocessor values on the compiler command line.
32. Define a function that takes a **double** argument and returns an **int**. Create and initialize a pointer to this function, and call the function through your pointer.
33. Declare a pointer to a function taking an **int** argument and returning a pointer to a function that takes a **char** argument and returns a **float**.
34. Modify **FunctionTable.cpp** so that each function returns a **string** (instead of printing out a message) and so that this value is printed inside of **main( )**.
35. Create a **makefile** for one of the previous exercises (of your choice) that allows you to type **make** for a production build of the program, and **make debug** for a build of the program including debugging information.

[30] Note that all conventions seem to end after the agreement that some sort of indentation take place. The feud between styles of code formatting is unending. See Appendix A for the description of this book's coding style.

[31] Thanks to Kris C. Matson for suggesting this exercise topic.

[32] Unless you take the very strict approach that "all argument passing in C/C++ is by value, and the 'value' of an array is what is produced by the array identifier: it's address." This can be seen as true from the assembly-language standpoint, but I don't think it helps when trying to work with higher-level concepts. The addition of references in C++ makes the "all passing is by value" argument more confusing, to the point where I feel it's more helpful to think in terms of "passing by value" vs. "passing addresses."