C and C++
For Java Programmers

Scott D. Anderson
Wellesley College
Scott.Anderson@acm.org

Fall 2002, version 5

This document is intended to evolve into an introduction to C and C++ for programmers whose primary experience is Java. C is a much simpler and older language than Java,™ but that simplicity does not necessarily make it easy for Java programmers. Similarly, because C++ evolved from C, it shares many of the design criteria of C, and so it can also be confusing for Java programmers.

Unfortunately, this document is still in the "draft" stage, particularly the chapters on C++, so I hope you'll bear with me. Please let me know of any suggestions you have to improve it.

Contents

Acknowledgments

I am grateful to my students and colleagues for reading early drafts of this and giving me feedback. They include Jessica Linker, Erin Stadler, Leah Kaplan, Erika Symmonds, Allen Downey and others who didn't give their names. A later draft was aided by Rachel Tornheim, Megan Heenahan and Franklyn Turbak. The remaining errors and contorted sentences are my own.

Chapter 1

Beginning

1.1 Introduction

C is a fairly old programming language, dating to the early 1970s. It was developed for smaller computers, unlike the "giant" mainframes that now seem so puny and underpowered (except for their electricity usage). It was developed for the first writing of the Unix operating system—the first operating system to be written in an "high-level" language (not assembly language), though nowadays C is thought of as a fairly low-level language, certainly compared to Java.

Because of the amazing success of Unix, in the academic and business world (until the ascendancy of Microsoft in the last decade), C became the most widely used programming language, one that every programmer was expected to at least be familiar with. A *lingua franca*, if you will. Consequently, many languages developed since the 80s have adopted the C syntax, the most notable being C++ and Java. C continues to be an important language, not only because of the large amount of existing C code, but because of Linux, which is a growing share of the server market (IBM has put a billion dollars behind developing Linux for its computers), and Linux (like Unix before it) is coded mostly in C.

New hardware can be like new land pushed up out of the sea by volcanic activity. At first, nothing grows at all. Then lichens gain a foothold, and by their action, produce soil that can eventually support grasses, trees, and the whole panoply of life. It all starts with that first hardy, simple pioneer. C is like the lichens: it's often the first "high-level" language (meaning not assembly language) to colonize new computer hardware. This is partly because, like lichens, it is a simple language. Indeed, it's been called "structured assembly language." However, once C has gained a foothold on the new machine, other languages can be ported to it, since those other languages are often implemented in C. The core of Emacs is written in C, including a lisp interpreter, and the rest of Emacs is written in Emacs-lisp running on that lisp interpreter. Without C—no Emacs. Similarly, the compilers for other languages (possibly C++) are written in C. I don't know, but I would be willing to wager that the Java Virtual Machine (the JVM) and the Java compiler (javac) are written in C. For this reason, C is still an important language and is likely to remain so.

Sometimes, the hardware is one that no other "higher-level" languages will be used on. Despite all the millions of PCs in the world, there are probably more "embedded processors." These are the processors that run our fuel-injection cars, our digital cameras, our Game Boys,^{TM} our microwave ovens and our toasters. While some of these might run Java, most will never run any thing more than machine code, and that machine code will probably be created by a cross-compiler for C.

Both C++ and Java are modern, object-oriented languages, but they have different design philosophies that are reflected in differences in their syntax and semantics. Thus, although they seem to fill the same niche, it's not as easy to learn the other as one might hope. This document hopes to help Java programmers cross the bridge to C and C++ (though you may want to return once you get there). Since the C++ language includes all of C, it's useful to read everything if you're learning C++. On the other hand, if you're just interested in learning C, you can skip all the parts on C++.

1.2 Other Sources

The classic reference for C is by Kernighan and Richie, the designers of C, and is affectionately known as "the white book." It's short, with lots of examples. Check the library. However, you should know that it's an old book, and some of its style is no longer allowed. There is a newer edition, with the word "ANSI" stamped in red letters on the cover, which has been updated.

I like Steve Oualine's O'Reilly book on C++, called *Practical C++ Programming*, so I'm sure his C book, called *Practical C Programming* is good, too.

A good reference for the C library as well as the language is *C: A Reference Manual*, by Samuel P. Harbison and Guy L. Steele jr. This covers many of the standard C library functions as well as covering the language itself. It is, however, a reference manual, not a tutorial.

Appendix D of Hennessy and Patterson's undergraduate computer architecture book, *Computer Organization and Design: the Hardware/Software Interface*, has a short introduction to C for Pascal programmers. However, the syntax it covers is mostly the C syntax that is the same as Java syntax, so this isn't so useful for Java programmers.

I recommend Tom Anderson's introduction to "C++ for C programmers," which is just a 20-page handout. Even if you're not a C programmer yet, but many of the things he'll say will be quite understandable anyhow. He wrote his for a collection of educational software called "Nachos," which is used in many operating systems classes. I like a lot of his advice, though I don't agree with all of it.

There are a number of online tutorials as well, including one on the Wellesley CWIS: `http://www.wellesley.edu/CS/courses/CS331/CTutorial.html`

1.3 C Design Philosophy

Because C was intended as a low-level language for small computers, essentially a replacement for assembly language programming, its designers and compiler-writer always chose speed and efficiency over safety. For example, there are no run-time checks to ensure that array indices are within bounds. This is fast, because the object code doesn't have to do any checking, but if the programmer makes a mistake, there is no `ArrayIndexOutOfBounds` exception to indicate the error. This puts C in stark contrast with Java, whose designers knew that safety would make Java programs more reliable and easier and quicker to code. They were willing to sacrifice code speed to get coding speed.

By and large, C programmers won't admit that the language is hard to code in, and, if they do, it's with a kind of macho bravado. Some of them are the kind of people who would race a motorcycle without a helmet because "the helmet adds weight and drag." Most, however, understand that these are design tradeoffs. They prefer to turn on the error checking when necessary, but to leave it off by default.

Often, that's a very reasonable position. If a quick-and-dirty program gets an error because an array index is out of bounds, well, you fix it, recompile and continue. Unfortunately, that kind of thinking can infect programs that aren't (or shouldn't be) quick-and-dirty. Web servers that can crash when some unexpected input is received often have programming errors that are no more complex than an array index out of bounds. Of course, throwing an exception in Java isn't any better if the program still crashes. Safe programs can be written in C just as in Java. The coding goal must be to anticipate all situations and make sure that none of them crash the program. This may be a little harder in C than in other languages, but the biggest difference is in the habits of the people using the language. As you code in C (and C++), consider the goals of the program and whether you are checking for all of the situations that you ought to.

1.4 Errors

To a first, cynical, approximation, C and C++ have only one run-time error message:

```
segmentation fault: core dumped
```

You can also sometimes get a bus error, but they all mean the same thing: the program has finally done something to go outside its memory allocation (its "segment"). The underlying cause may or may not have have been recent; you'll have to trace back and use some debugging techniques to figure out what went

wrong. If all else fails, put "print" statements in your program until you can determine what line is failing (but see the advice about buffered output, below).

There are debuggers and debugging environments for C and C++. Almost every experienced C programmer has used `gdb`, the GNU debugger, at some point. There are even some with graphical user interfaces that can show you linked structures in memory and update their contents as necessary. These are all worth learning, but this document doesn't cover them at all.

Instead, I urge you to rely on logic. For example, given the following code:

```
if( ... ) {
   ...
   printf("The value of x is %d\n",x);
   ...
} else {
   ...
   printf("The value of y is %d\n",y);
   ...
}
while( ... ) {
   printf("The current value of z is %d\n",z);
   ...
}
```

If neither x nor y is printed, the error, whatever it is, cannot be with the `while` loop. A student I knew once spent an hour trying to find the bug in the `while` loop, convinced that the bug was there, and I was unable to persuade her otherwise. I know it was very frustrating for her, as it would be for anyone.

It's inevitable that you will make coding mistakes as you learn C and C++; everyone does. Sometimes you can find the error easily by just scanning over your code, but if that doesn't work, you'll have to debug it. Don't get stuck reading and re-reading your code. It's better to have a plan of attack for debugging it.

Frustrating as debugging is, it is a good opportunity to practice some logical detective work. We will all try to emulate Sherlock Holmes, and remember that

> eliminate the impossible and whatever remains, however implausible, must be the truth.

Here are some rules for effective debugging. These are true for any language, and are useful with or without a debugger.

Cut the problem down to size.

Finding an error in a 1000 line C program is a lot harder than finding an error in a 100 line program. So the first thing to do is to cut out parts of the file. (This advice is often called "divide and conquer.") Anything that you think is irrelevant, cut out. Do this in several steps: first cut out all the stuff that you're really sure is irrelevant, then test the file. If the error remains, cut out the stuff that you're more dubious about, and test again. Keep cutting back until the error disappears.

Aside: emotionally, it's very difficult to cut out code that you've worked on and possibly debugged, because you don't want to have to re-do it. Furthermore, it feels like it's a waste of time, compared to just re-reading the code. This is an illusion, though. You will actually save time by debugging in an effective way. First, there are two ways to avoid having to re-do all the code you cut out. One is to copy the entire program to a temporary file called, say `bug1.c`. You can then cut parts out of `bug1.c` and experiment with it until you determine what caused the error in the original file, then you can discard `bug1.c` and go back to the original file and just fix the error. Alternatively, you can copy/paste the pieces you remove from the original to an `outtakes.c` file. Once you've found and fixed the bug, it's easy to copy/paste them back. Cutting out code makes your debugging more efficient even if you're just re-reading the code, since you have less to re-read and your reading is focussed on the relevant portion of the file.

Of course, you're probably thinking that it's easier just to comment out sections, and indeed that is true. However, I'm advocating being very ruthless with the cutting, and when large chunks of the code are commented out, it can become difficult to read the relevant code. Therefore, if you're only having to cut out

a small part of the code, by all means go ahead and comment out. I do. However, when you feel confusion set in and frustration rise, take a deep breath and start cutting.

How do you decide which parts are irrelevant? This is a very difficult decision: it depends on the nature of the problem you're having, and you will get better at it as you learn more. Programs are heavily contextual, so a bug later in the program is often dependent on much that went before. However, if you can determine the set of values prevailing at the time a bug occurs, you may be able to cut out anything that is unrelated to setting those values.

Cut at function boundaries

If you can determine that the bug is in a function or method, you can isolate it because you know that the function or method should depend only on its inputs (and maybe some global variables, hopefully only a few).

Confirm what you believe.

As you go through a chain of reasoning, confirm things that you can:

1. At this point, x is positive
2. So, that means, here, this file will be read
3. That means that we'll skip this alternative calculation
4. Therefore, this other function is called
5. And so this while loop will run

And so forth. It's up to you whether to put a "print" statement for each of these steps. On clever technique is to try to use "binary search." Put a print statement at the third inference above to determine if it's true. If it is, you can worry about inferences 4 and 5. If it's not, you have to worry about inferences 1 and 2.

All of these steps may seem like a waste of time: editing the file, adding the print statements, re-compiling and re-running. Nevertheless, they are more productive than re-reading the code for the umpteenth time, hoping to catch the problem. They also exercise your logical faculties and help you understand your program much better, which increases the chance that the error will stand out.

A common problem that occurs when debugging is getting confused about the error and your hypotheses. For example, when we inferred, above, that the alternative calculation will be skipped, and we test that and it's not true, we now have a new bug: why isn't the alternative calculation skipped? (This is at least a more interesting bug than "segmentation fault!") Still, it's easy to become confused.

Take notes; make versions

It's perfectly reasonable, when tracking down a difficult bug, to end up with five or ten versions of your program, with different debugging statements in them. Try to name them or put a note in them about what you're working on, particularly if you have to leave the computer before you find the bug.

Finally,

Take a break

Sometimes I do my best debugging in the shower or while walking home. At the very least, you can clear your head and come at it again with fresh thoughts and attitude. Then you'll find it quickly.

1.5 Warnings and Advice

Here are some quick general warnings. Some of these will make more sense after you've read through the text, but they're up front so you can easily remind yourself.

- C and C++ do not do bounds-checking for arrays, so you can take the 15th or 150th element of an array of 10 elements. What happens? All bets are off. Often, your program just crashes (segmentation fault). Other times, it computes mysterious things. Just be really careful.
- C and C++ do not check pointers before following them. So, all the same warnings from the previous item apply. (In fact, in C/C++ an array is just a pointer to its first element, so these are actually the same thing.) You won't get a `NullPointerException`; you'll get a segmentation fault.
- C and C++ do not automatically initialize variables. If you say:

  ```
  int x;
  int* y;              /* pointer to an integer */
  ```

  `x` could have any value at all, and `y` could point anywhere in memory. Pointers aren't initialized to NULL, and integers aren't initialized to zero. You should *always* initialize your variables and you should *always* check a pointer before following it; the cost in efficiency is simply not worth admitting this source of error.
- Don't name a program `test`, even though it's a test program. That's because there's a built-in command in Unix called "test," which will take precedence over your program name. You can call it pretty much anything else. If you're ever worried that the program that is running is not the one you wrote, just add a print statement in the first line of `main`.
- Print statements in C and C++ use buffered I/O, which means they aren't printed to the screen the moment they are executed, but instead are printed to an internal buffer to be printed later. This can be confusing for debugging statements. Therefore, your debugging statements should *always* end with a newline character, like this:

  ```
  printf("Got to this line.\n");
  ```
- There is no "new" operator in C, as there is in Java, so you have to think about variables in the C way.
- Until you feel more comfortable with the language, avoid dynamic memory allocation in C if you can. It won't hurt to pre-allocate strings and arrays while you're still a novice.
- Error messages from the C or C++ compiler can be cryptic. One that often trips people up is "bad lvalue." This means that the compiler came across something that it thinks is an assignment, but the thing on the *left* (the *l*value) isn't a storage location. For example, you might get that error message with the following code:

  ```
  if( x+y = z ) { ...
  ```

  You probably meant to write `x+y==z`, but accidentally left out an equals sign — a common error for people using C, C++ or Java. The error message just means that you can't assign a value to `x+y`, since that's not a storage location.

1.6 Conventions

All code will be in a fixed-width `typewriter` font. When an interaction with the command line is given, I'll use the percent sign as the Unix prompt and user input will be in bold; the computer's response will be in ordinary typewriter font. For example:

```
% date
Tue Oct  8 16:56:40 EDT 2002
```

To the extent that it matters, I will be assuming a Unix/Linux programming environment. If you're programming in Windows or on a Mac, you'll have to make the appropriate translations.

Chapter 2

Overview of C

In this opening chapter, I'll use a small example to introduce some general mechanics of programming in C. The next chapter will survey the C language, comparing it to Java when that's helpful. I'll defer all the details of the code until those sections. For now, let's just dive in and get our hands dirty.

As with most programming, we start in an editor. Start with your favorite editor, which should be Emacs.[1] Write the following code. This is the nauseatingly famous "hello world" program.

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

Save that as `hello.c`, and then switch to a "shell." (I'm assuming you're programing in Unix/Linux). We now need to compile and run the program.

The usual C compiler on a Linux machine is the GNU C Compiler, called gcc. You can run it on your program, putting the output in `hello` as follows:

```
% gcc hello.c -o hello
%
```

Notice that if it succeeds, the command just finishes without any noise or warning. Don't look for any congratulations from Unix.

You could put your output in any filename, but it's traditional to name it the same as the main source file, but without an "extension" (anything after the dot).

You now need to run the program. This is done by giving its name on the command line, just as if it were a command.

```
% hello
Hello World!
%
```

(On some computers, you need to precede the command name with `./`, because you don't have "." (dot) on your PATH. Consult a local guru or system administrator if you don't know what that last sentence means.)

Returning to the code, you'll see that there is a function named `main`, and it calls a function named `printf` to print a string (the stuff in quotation marks). Don't worry about the `#include` line for now.

Let's do something slightly more interesting. Let's write a program that will tell us the successor of any number we give it. Type the following code into an editor:

[1] Forgive me this burst of bigotry; I'll try to be more even-handed in the language comparisons.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int n = atoi(argv[1]);
    printf("The number after %d is %d\n", n, n+1);
}
```

Then compile it (put the output in a file named `next`) and run the program as follows:

```
% next 3
The number after 3 is 4
% next 4
The number after 4 is 5
% next five
The number after 0 is 1
% next
Segmentation fault
```

The first two examples show the program working normally. The third shows the program behavior when we give it an erroneous command line argument. The last shows the program behavior when we forget to give it the command line argument. When something really goes wrong with a C program, we get a segmentation fault. That's usually the most informative error message, and it's pretty un-informative.

Let's go back and look at the code. The `argv` is a vector of the command-line arguments. Each is a string (we'll get into how strings are represented and handled in the next chapter). Element 1 is the first command-line argument, and you can see that we hand that to `atoi` to yield an integer. The `atoi` function converts from a string (alphabetic) to integer; it returns zero if it doesn't understand the string. Finally, we print out the value using `printf` and strange `%d` markers where the numbers are supposed to go.

Take `atoi` and `printf` on faith for now; we'll learn more about them later. Let's go on to make the program a bit more useful by computing something we can't necessarily do in our heads. We'll use it to compute powers. Type the following into an editor:

```
#include <stdio.h>              /* for printf */
#include <math.h>               /* for pow */

/* Note that there is an error in this program! */

int main(int argc, char* argv[]) {
    float x, y;

    x = atof(argv[1]);
    y = atof(argv[2]);
    printf("%f to the %f is %f\n", x, y, pow(x,y));
}
```

The `atof` function is like `atoi`, except that it converts a string to a floating point number. Otherwise, the program is a simple variant of what we've seen. However, as the comment says, there's a subtle error in this program that I think it's useful to experience firsthand. Let's compile the program:

```
% gcc pow1.c -o pow1
/tmp/ccrVusQp.o: In function `main':
/tmp/ccrVusQp.o(.text+0x57): undefined reference to `pow'
collect2: ld returned 1 exit status
```

The error message says that the `pow` function is undefined. That's correct; we have to get it from a library. The math functions aren't automatically included in the compiled file, so that the file can be as small as

possible. If you use them, you have to notify the compiler, or rather the *linker*, which is the program named `ld` in the error message above. To tell the linker to link to a library, you have to add a `-l`*libraryName* to the command line. For the sake of brevity, the math library is just known as "m." So, we change our compilation command to the following:

```
% gcc pow1.c -lm -o pow1
```

Ah, sweet success! Now let's run it, just to see:

```
% pow1 2.0 8.0
1073741824.000000 to the 1075838976.000000 is inf
```

What? This doesn't make any sense! Why isn't `atof` working properly? The answer is that the compiler didn't see a declaration of `atof` anywhere, so it *assumed* that it took integer arguments and returned integer values. You could argue that this is a stupid assumption, and I would agree with you, and we might rant together that the compiler should just give an error message instead. Nevertheless, this is what C does. So, we modify the program to add the correct declaration by including the correct header file:

```
#include <stdio.h>              /* for printf */
#include <stdlib.h>             /* for atof */
#include <math.h>               /* for pow */

int main(int argc, char* argv[]) {
    float x, y;

    x = atof(argv[1]);
    y = atof(argv[2]);
    printf("%f to the %f is %f\n", x, y, pow(x,y));
}
```

Now, we re-compile, carefully remembering to link in the math library, and we get the following pleasant behavior:

```
% gcc pow2.c -lm -o pow2
% pow2 2.0 8.0
2.000000 to the 8.000000 is 256.000000
dudley cpp/ pow2 2.0 0.5
2.000000 to the 0.500000 is 1.414214
```

So, the message from this example is that using library functions requires *both* including the correct header files (the ".h" files) and linking with the correct libraries at compile time. Fortunately, for most beginner programs, the only library you will ever have to worry about is the math library.

For our next example, let's see how to define and use our own functions. Let's write a program that computes factorials:

```
#include <stdio.h>              /* for printf */
#include <stdlib.h>             /* for atoi */

void indent(n) {
    int i;
    for( i=0; i<n; i++) printf("  ");
}

int factorial( int );           /* declaration */

int factorial( int n ) {        /* definition */
```

```
    int result;

    indent(n);
    printf("factorial of %d ... \n",n);
    if( 0 == n ) {
        result= 1;
    } else {
        result = n*factorial(n-1);
    }
    indent(n);
    printf("factorial of %d is %d\n",n,result);
    return result;
}

int main( int argc, char* args[] ) {
    factorial(atoi(args[1]));
}
```

Before we discuss the code, let's see it in action. Compile the file and run it as follows:

```
% factorial 6
              factorial of 6 ...
          factorial of 5 ...
        factorial of 4 ...
      factorial of 3 ...
    factorial of 2 ...
  factorial of 1 ...
factorial of 0 ...
factorial of 0 is 1
  factorial of 1 is 1
    factorial of 2 is 2
      factorial of 3 is 6
        factorial of 4 is 24
          factorial of 5 is 120
            factorial of 6 is 720
%
```

So, we expect to see something that indents based on the number that we are computing the factorial of, and we expect to see the computation of the factorial. Let's look at the code.

The first thing we see is the definition of the `indent` function. It just uses a `for` loop to print some number of spaces, depending on a numeric argument. Nothing too challenging for a Java programmer, even if the syntax is a little new. Note that the function returns no value, so it's declared to return `void`.

Next, we see a *declaration* of the `factorial` function. A declaration just gives the name, return type and argument types for a function. This notifies the compiler about the function so that calls to that function can be properly compiled.[2] These are the kinds of things that header files, like `math.h` and `stdio.h` are filled with. Finally, we see the *definition* of the `factorial` function, complete with excessive print statements. The result of compiling a definition is what lives in a library that we link to using the `-l` option to `gcc`.

In these few brief examples, we've seen the basic mechanics of writing, compiling and running C programs, and we've seen some samples of what C code looks like. We're now ready to survey the C language. You're encouraged to write little programs to test features as they are presented, as well as trying the sample programs presented here. When it comes to learning a language, there's no substitute for actual experience.

[2] The declaration isn't strictly necessary for this example, but it was worth introducing the syntax.

Chapter 3

The C Language

3.1 C Syntax

Fortunately, Java borrows most of its syntax from C, so learning C syntax will be easy. All the syntax of `for`, `while` and `do` is unchanged. Expression syntax is unchanged. Java even adopted the maligned `?` ternary operator.[1]

Thus, a Java program consists of a bunch of classes, each comprising instance variables and methods. A C program consists of a bunch of function definitions, along with the occasional structure definition and global variable. We'll look at the syntax of a file later (see section 3.7 on page 22).

The syntax of much of the executable part of the language, particularly control structures, is the same in C, C++ and Java:

- assignment statements
- arithmetic expressions
- operator precedence
- `if` statements
- `switch` statements
- `while` loops
- `for` loops
- `break` and `continue` statements

Unfortunately, there's still a lot more to cover, even before we get to complex things like structs and classes.

3.2 Comments

Comments in C are like this:

```
/* this is
a long
paragraph of
comments
*/
```

They begin with `/*` and end with `*/`. They don't nest. There is no "end of line" comment, like the `//` of Java and C++, although the GCC compiler will often let you use the `//` comment, so feel free to try it. Just remember that it's not strictly legal.

[1] If you don't know what this is, your professors have hidden this horror from you. See section 3.21, if you really want to know.

3.3 Constants

In Java, you can declare a constant by declaring a variable and marking it "final," as if it were an immutable variable. For example:

```
class Math {
    public final float PI = 3.1416;
    ...
    }
```

In C, that's not possible. You can either declare it to be like an ordinary variable or you can use the *pre-processor*. The pre-processor is covered in more detail in section 3.22 but for now suffice it to say that the preprocessor is a not particularly smart program that scans over the input file looking for "directives" marked by a pound sign. It then makes textual substitutions using the directives. So, in C, the way we define pi as a constant is:

```
#define PI 3.1416
  ...
float circum = PI*diameter;
```

You can use `PI` anywhere you could use the number 3.1416, but not anywhere you could use a variable. You also have to be careful not to use the letters PI in other contexts, because the pre-processor is dumb enough to replace all occurences of the characters `PI` with the characters `3.1416`.

For this reason, C programs always use all caps for constants and nothing else gets all caps.

3.4 Function Syntax

Historically, functions are the antecedents of methods, so they share a lot both syntactically and semantically. C function syntax is actually a bit simpler than that of Java methods.

3.4.1 Calling Functions

C predates the widespread adoption of object-oriented programming (OOP). Thus, it does not have classes and methods. Of course, Java is a thoroughly object-oriented language, so it *only* has classes and methods. C, on the other hand, *only* has functions. When Java wants a plain function, such as the $\sin(\theta)$ function that you learned in trigonometry, Java makes it belong to a class, requiring you to write

```
y = Math.sin(theta);
```

In C, you simply write:

```
y = sin(theta);
```

So, calling a function in C is really quite simple if you know how to do it in Java.[2]

3.4.2 Defining Functions

C functions don't belong to any class, so you just define them at "top level" in your file of code. ("Top level" means that it is not surrounded by any other syntax, though it may be preceded or followed by other top level items.) To define the "hypotenuse" function, you need only write:

```
float hypotenuse( float a, float b ) {
   return sqrt(a*a+b*b);
}
```

[2]It's not quite as simple as I've implied, because to call the `sin` function, you have to have included the math library headers, which we'll discuss later.

The function is *not* part of a class definition. Contrast that with a Java implementation:

```
public class Hypotenuse {
   public float hypotenuse( float a, float b ) {
      return Math.sqrt(a*a+b*b);
   }
}
```

As we saw earlier, to *call* a function, you need only give its name, without worrying about classes. The above C function can be called like this:

```
float c=hypotenuse(3,4);
```

The Java method, on the other hand, would be:

```
float c=Hypotenuse.hypotenuse(3,4);
```

Secondly, as you noticed, you don't need to worry about keywords like "public" or "static." By default, all your functions are *public*, and so are usable at any point after (in the file) they are defined. If you want to use them in a different file, we'll have to make use of the extra machinery described in section 6.10 on Separate Compilation (page 62).

3.4.3 Local Variables

In C, all variables must be declared at the beginning of the function, regardless of when they're used. That is, all the (non-executable) variable definitions have to precede the first executable line of code.

In C++ and Java, on the other hand, you don't have to declare a variable until you need it, which is much more convenient. In Java and C++, we also have locutions like this:

```
for(int i=0; i<N, i++) {
   ...
}
```

In C, the variable `i` must be declared at the top of the function, not in the `for` loop.

Because the first few lines of a function are variable declarations, preceding the executable code, it's common, although not required, to put a blank line between the variable declarations and the first line of real code. For example:

```
int foo(int a, int b) {
   int i;
   float x, y;

   for( i=0; i<a*b; i++ ) {
      x=bar(a,i);
      y=baz(i,b);
   }
}
```

Notice that this style makes it easy to forget to initialize a variable. In this case, the variables `x`, `y` and `i` didn't need to be initialized. However, this is a bad habit, because the one time that they do need to be initialized and you forget, you could spend hours tracking down a bug. It's always safer to code like this:

```
int foo(int a, int b) {
   int i=0;
   float x=0, y=0;

   for( i=0; i<a*b; i++ ) {
      x=bar(a,i);
      y=baz(i,b);
   }
}
```

C gives you speed over safety by default, but that doesn't prevent you from adding your own safety measures.

You can (and should) assign a value to a variable when you define it, but as we get to more complex variables, we'll see cases where it's difficult to initialize the variable with its correct value. When that happens, Java would initialize the variable to zero or null; you should do that explicitly in C or C++.

3.5 Printing

Printing is done wildly differently in C than in Java (or in C++). Personally, I like the way C does printing. I think printing has not found a natural, intuitive expression in object-oriented thinking, though maybe it will someday. Meanwhile, it's good to learn different ways of thinking about how to accomplish something.

In C, as in most languages, it's easy to print a simple, fixed string. Here we will use the standard C function called `printf`; you'll remember this example from our "hello world" program in the beginning. (Anytime you use `printf`, which will be pretty much every program, remember to include the correct header file, which is `<stdio.h>`. See section 3.8.)

```
printf("Hello world!\n");
```

As in Java, we can use `\n` to denote a newline character. Unlike Java, which has two different print methods depending on whether you want a newline at the end (`System.out.print` and `System.out.println`), in C, we only use `printf`, ending the string with `\n` if we want a newline character.

3.5.1 Printing Variables

Of course, we also want to print variable values. In C, we think of printing as a "fill in the blank" template. You first state the template, which is all the plain text you want printed, and then you "fill in" parts of the template from the values of variables. Here's how to print two integers, the values of variables `x` and `y`:

```
printf("At this point, x = %d and y = %d\n", x, y);
```

The percent sign followed by a letter marks a place to fill in the template. However, just to be tricky, we have to choose the correct letter based on the datatype of the variable we want to print. C is perfectly happy to print a floating point number as if it were an integer or vice versa, if you use the wrong letter. The result is usually gibberish, so if your numbers seem to be printing strangely, double-check the format string. Here are the basic set of letters:

i,d	integer
c	character
s	string (AKA pointer to character)
f	float

As we will learn in the section on library functions, there are online manual pages that contain a lot of information. You can find out about the huge number of other options, including formatted I/O (such as specifying the number of decimal places, leading zeros and so forth) by looking at the online manual page for `printf`.

3.5.2 Examples

The following program shows some normal usages of the `printf` function:

```
#include <stdio.h>              /* for printf */
#include <math.h>               /* for pow and M_PI */

int main() {
    int n;
    printf("A string %s walks into a bar....\n", "hello");
    printf("That %c is a real character.\n", 'q');
```

```
    n = (int) pow(2,8);
    printf("Every programmer knows that 2 to the %d is %d\n", 8, n);
    printf("Many geeks memorize PI to 3 (%.3f), 5 (%.5f) or more (%f) places.\n",
           M_PI, M_PI, M_PI);
}
```

The compilation and output of the program is as follows:

```
% gcc printf.c -lm -o printf
% printf
A string hello walks into a bar....
That q is a real character.
Every programmer knows that 2 to the 8 is 256
Many geeks memorize PI to 3 (3.142), 5 (3.14159) or more (3.141593) places.
%
```

Warning: the C compiler doesn't check that you have supplied the correct number of arguments to `printf`. For example, in the following print statement, we have supplied too many arguments; `z` is just silently ignored:

```
printf("At this point, x = %d and y = %d\n", x, y, z);
```

On the other hand, if we supply too few arguments, C will just print whatever random junk is in memory, (or it will produce a segmentation fault):

```
printf("At this point, x = %d and y = %d\n", x);
```

3.5.3 Buffered Printing

One of the things that can be strange about printing is *buffering*. Consider the following example:

```
#include <stdio.h>

int main() {
    int x,y;
    printf("1. about to set x\n");
    x=0;
    printf("2. doing the division:");
    y=1/x;
    printf("3. result is: %i\n",y);
}
```

Now, you would expect to get the following output:

```
1. about to set x
2. doing the division:
Floating point exception
```

However, what you actually get probably is:

```
1. about to set x
Floating point exception
```

Looking back at the code, this seems to indicate that the error happened between print statements 1 and 2, but how could this be? How could setting `x` to zero go wrong?

The answer is that nothing went wrong with setting `x`, it's just that the second print statement didn't get printed to your screen. For efficiency reasons, it got printed to an internal "buffer." Later, when the buffer is full, the buffer will all get printed. (This is also called "flushing" the buffer.) Usually, you're not

aware of that delay, except when a program crashes (as in this case), and so the contents of the buffer never get printed.

How can you deal with this? The best way is to take advantage of the fact that the buffer gets flushed when the program prints a "newline" character. In fact, print statement 1 gets printed only because there was a newline character at the end of it. If we hadn't done that, the program's output would have been:

```
Floating point exception
```

We would think that we got a floating point exception before the program even started!

> Make sure that your debugging print statements always end with a newline (`\n`).

3.6 Types

The primitive datatypes of Java and C++ are mostly adopted from C, so `int` and `float` and `double` and so forth all work as you expect, except that Java added the "wrapper" classes, like `Integer`. Don't use wrapper classes in either C or C++: they don't exist.

The following subsections discuss the basic primitive types in C/C++ and Java.

3.6.1 Integers

C has integer variables just as Java does, and they are declared and used the same way.

```
int x = 7;
```

The only tricky thing you may run into is that C defines an integer to be a "word-size" quantity[3] so that you can be sure that ints are fast but you can't be sure how big they are. However, most modern machines have 32-bit words (or larger), so this is no longer a serious concern. However, there are still machines and compilers in which the following code will not work as expected, because an int is not large enough to hold a value that big:

```
int y = 10000000;
```

In Java, on the other hand, ints are guaranteed to be 32 bits.

3.6.2 Booleans

C doesn't have boolean as a separate type. Instead, boolean is a subtype of `int`: zero is false, and every non-zero integer is true. Thus, both of the following print "true":

```
x = 7;
if( x != 0 ) print("True\n");
if( x ) print("True\n");
```

The former is clearer and is therefore considered better style, but occasionally, you'll find tricky code that takes advantage of the boolean nature of integers. You may even find it convenient yourself sometimes, once you're comfortable with this.

---

[3] A "word" of computer memory is a chunk that is of the correct size and address to be brought to the processor from memory in a single access. Most current processors, such as the Pentium, are 32-bit processors and the word-size is 32 bits. Newer processors such as the Itanium has a 64-bit word.

3.6.3 Characters

Literal characters in C are notated with single quote marks. This is a way to create a character constant:

```
char ok = 'Y';
```

A few characters that are hard to specify in this way have special "escaped" notation:

```
char newline = '\n';
char carriage_return = '\r';
char tab = '\t';
```

Characters in C are all ASCII (since Unicode wasn't even a gleam in anyone's eye when C was invented). Thus, they are just a single byte, not two as in Java. Also, the `char` datatype is not distinct from `int`; it's considered a one-byte integer instead of a word-sized integer. Therefore, a character can also be specified by its ASCII code:

```
char capitalA = 65;
char space = 32;
char null = 0;
```

The last character is also called the "null byte," since characters take up just one byte.

Furthermore, C will coerce characters to numbers and vice versa, using the ASCII table. Thus, the following is possible, and produces the number (integer) 25 (since 'Z' has ASCII code 90 and 'A' has ASCII code 65):

```
int diff = 'Z' - 'A';
```

What this statement does is find the "distance" (in number of characters) between the two given characters.

You can also do things like this, where `next` would have the character `'B'` as its value:

```
char next = 'A' + 1;
```

In fact, the datatype "byte" doesn't exist in C: it's called "char."

What is ASCII? ASCII stands for "American Standard Code for Information Interchange" and is pronounced "ask-key." It's just a set of characters that have been numbered, but the code was agreed upon by a number of computer manufacturers long ago (back in the 1950s). There was a competing code from IBM called EBCDIC, but C was invented for non-IBM computers, so it used ASCII. The ASCII code has 128 characters in it, the first 32 or so were "control characters" that were used to control machinery. For example, there was a character to tell the printer (back in the days when printers were like typewriters and printed each character individually by bashing a metal key onto a ribbon and then onto the paper) to go back to the first column (carriage return). This happened to be "control M". Another control character told the printer to move the paper up one line (line feed) and this was "control J". Control G rang the bell. Control H backed up by one character. Control L advanced the paper to the top of the next page.

Here's a table of the visible characters, plus space (SPC) and delete (DEL). Each row has 8 characters in it, and the number at the left of the row is the ASCII code of the leftmost character. You can determine the ASCII code of other characters by counting over, using the numbers at the top of the column. For example, the ASCII code for "n" is 104+6 or 110.

	0	1	2	3	4	5	6	7
32	SPC	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL

You can see that ASCII is very simple. As far as C is concerned, a character is just a small integer (less that 128) that you happen to refer to in the context of strings and characters. That's why we can assign numbers to character variables and so forth.

Of course, ASCII has some limitations. For example, there are lots (*lots*) of characters that are not in it. Not just characters like ∅ from Norwegian or α from Greek, but ¿ from Spanish and anything at all from Chinese or Arabic. Java uses Unicode, which is a much larger character set (and, incidentally, a Java character takes up two bytes, as opposed to one byte for ASCII), designed to handle all the complexity of the world's languages. The result is that characters in Java are complicated. Unicode is clearly the way of the future, but ASCII is still enormously common and convenient. (The international standards organizations are just now approving non-ASCII characters in domain names, so that a Greek web site on literacy can be "$\alpha\beta\gamma$.com" (actually, it would probably be "$\alpha\beta\gamma$.co.gr").

3.6.4 Typecasting

You can convert from one type to another using a "type cast," just as you would in Java. The syntax is to put the new type in parentheses to the left of the expression. For example, to convert from an integer to a float, you could say

```
int y = 3;
float x = (float) y;
```

Of course, in practice, the compiler will usually do the right thing, so until you start getting fancy, you won't need typecast. However, we will see some examples later, so it's best to be prepared.

3.6.5 Typedef

So far in this section, we've looked at the primitive types of C and seen that they are very similar to the primitive types of Java. C does have one feature that doesn't seem to have a direct analog in Java, and that is the ability to define a new type as a synonym for another type.

Suppose we want to define "number" as a synonym for `int`. We might do this to allow ourselve the freedom to later specify that "number" is a synonym for `float` or even `double` instead. We can define "number" as a type using `typedef`. The syntax of typedef is surprisingly straightforward: just declare the new type as if it were a variable of the old type, and precede the declaration with the keyword `typedef`. For example:

```
int x;                /* x is an integer variable */
typedef int number;   /* number is a type, synonymous with int */
number y;             /* y is also an integer variable */
```

This allows us to define any variable using the new type:

```
number y;

number square( number x ) {
   return x*x;
}
```

If we later change our mind and decide that `number` should be a `float`, we can just change the `typedef` in one place and all the definitions will change automatically.

3.6.6 Wrapper Classes

As you recall, C doesn't have wrapper classes. The lack of wrapper classes doesn't weaken the language substantially, but it is slightly less convenient. In Java, you probably use the wrapper classes, say `Integer`, for one of the following purposes:

- Determining limits, such as the constant `Integer.MAX_VALUE`. In C, there are constants (defined in header files, see section 3.8), such as `MAXINT`.
- Parsing string values, such as the method `Intger.parseInt`. In C, there are library functions, such as `atoi`.[4]
- Packaging a primitive integer to put in one of the generic containers, such as a hashtable. In C, you would probably do this "by hand" (creating a structure to hold the integer) and using untyped variables in the container. This is a relatively advanced and rare programming technique; most C programs (like most Java programs) have collections of fairly homogeneous items. In C, the programmer would probably go to some effort to determine the set of types to be handled and would use a *union* to generalize over that set of types.

The following tiny program is an example of the first two purposes. The program reads a number from the command line and reports if it's bigger than half the largest possible integer. An example of the third purpose is in the section on unions (section 3.12, page 27). Note that this example uses some stuff about library functions and header files (page 22) and command-line arguments (page 38) and printing (page 16) that we haven't gotten to yet.

```
#include <stdio.h>              /* for printf */
#include <limits.h>             /* for INT_MAX */
#include <stdlib.h>             /* for atoi */

int main(int argc, char** argv) {
    int x = atoi(argv[1]);
    if( x > INT_MAX/2 )
        printf("That's a mighty big number!\n");
}
```

Here it is in action. Each line beginning with a percent sign is an execution of the program with a different large number. (The percent sign just stands in for the unix prompt; yours is probably different.)

```
% readnum 987654321
% readnum 9876543210
That's a mighty big number!
```

Packaging a primitive type so that it can be put in a generic container means essentially making it a subtype of `Object`, the most general Java class and the ancestor of all classes (including Integer and ones that are user defined). In C, there is no equivalent to Java's `Object` class. If you want a function that

[4] That name seems bizarre until you realize that it's an abbreviation of "ASCII to integer," where "ASCII" means a string of characters. We'll learn about ASCII soon.

operates on anything at all, you declare the arguments to be of type `void*`: a pointer to void is a C idiom that means a pointer to some unknown chunk of memory. The called function typically casts the pointer to some known type before doing anything with it. This paragraph will make more sense when we've learned about pointers and type casting, but I think it's best to just forget about `Object` in C. The `void*` trick is ugly and best avoided at all costs.

3.7 File Syntax

In Java, a program usually comprises several, perhaps many, files, each defining a single class. The Java run-time (the `java` program) links all these classes together into a complete program. C takes a very different approach, where often everything about a program is jumbled into one file. This isn't as clean as the Java approach, but can sometimes be easier during editing.

A file of C code consists of the following kinds of "top-level" things:

- constants (defined using the pre-processor, see section 3.22)
- type definitions
- global variable definitions
- function definitions

These can be specified in any order, with the rule that something must be defined before it is used. Therefore, a function definition must appear before the definition of the function that will call it. Thus, the following won't work:

```
float area(float radius) {
    return PI*square(radius)
}

#define PI=3.14

float square(float x) {
   return x*x;
}
```

When you compile this, the compiler will complain while it's compiling `area` that it doesn't know what PI is and what the `square` function is. Even worse, it may go on and assume that the types of all the arguments to the unknown function are integers and that the return type is an integer.

The consequence of this "define before use" rule is that files tend to be written with things in the order we saw earlier: constants first, then types, then global variables, then functions, with the last function being `main`. Strange: the beginning of the program is at the end of the file.

3.8 Library Functions

C's minimalist philosophy has a design with a tiny language—essentially just the control structures and data definitions that we've just seen—and everything else is relegated to libraries. This makes it easy to write a C compiler for a new machine, and, if the libraries are also written in C, the libraries can then be quickly "ported" (transformed to work on the new computer). Thus, C is often a "pioneer" language: the first language to be ported to new hardware.

Java similarly relegates many functions to libraries, which you must "import" at the top of the file. Java differs from C in that every Java program implicitly starts with

```
import java.lang.*
```

That is, a number of Java functions are available automatically, without asking. In C, you have to ask. You do so with the `include` syntax; to include the "Standard I/O" library, you put the following at the top of your file:

```
#include <stdio.h>
```

The disadvantage of C's minimalist approach is that there are many library functions out there, suitable for use (indeed, I'm mentioning this early so that I can refer to these functions in the examples below), but you have to (a) know that they exist, (b) know how to use them, and (c) include the correct "header files" in order to use them (more on these in a minute).

The hardest problem to address is knowing that they exist and what their names are. The best source I've found is Harbison and Steele (referred to in section 1.2), but many C books will, inevitably, cover the standard library routines.

Once you know the name of a function you'd like to use, it's much easier to learn how to use it. The online manuals on a Unix system include manual pages for the C library functions, so, for example, if you want to learn how to use `atoi`, you can just type `man atoi` to your Unix prompt and read the manual page. The quality of the writing varies, and they are usually not tutorials, but with a little practice you can use these quite effectively.

Secondly, the manual page will tell you what "header file" to include. A header file is file that the compiler can read to determine the syntax associated with a library of functions. It includes the names of the functions, the number and types of their arguments, and their return types. It also includes declarations of important types and variables and anything else necessary for the compiler to read for you to be able to use a library function. For example, the synopsis at the top of the manual page for `atoi` is:

```
SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);
       long atol(const char *nptr);
       long long atoll(const char *nptr);
       long long atoq(const char *nptr);
```

This tells you that to use `atoi`, you need to put the statement

```
#include <stdlib.h>
```

at the top of your file. The rest of the synopsis gives the signature for the function and its relatives, so you can read that `atoi` takes a string (`char*`) as its argument and it returns an `int`. (We'll discuss strings in section 3.10 on page 25). The rest of the man page describes what the function does; in this case, `atoi` converts a string representation of a number to its binary representation.

Also, you may have to change the way to compile and link the file. For example, when you use the math library, you'll have to add the `-lm` switch to the command line of your compilation command, so that the compiler will link your program with the math library. For example, instead of typing the following at your unix prompt to compile the `hypotenuse` program:

```
% gcc -o hypotenuse hypotenuse.c
```

you might instead have to type:

```
% gcc -o hypotenuse -lm hypotenuse.c
```

Many libraries, such as `stdlib` and `stdio` require no change at all to the compilation, but you should suspect a link error if you get an error message like this:

```
% cc -g hypotenuse.c -o hypotenuse
/tmp/ccGiy7ah.o: In function `hypotenuse':
/home/anderson/writing/cpp/hypotenuse.c:5: undefined reference to `sqrt'
collect2: ld returned 1 exit status
make: *** [hypotenuse] Error 1
```

The giveaway is the "ld returned 1 exit status" line: "ld" is the linker, so if it gets an error, you'll need to add a library to the compilation line. Consult a guru if this happens.

There is more on compiling C in section 6.1 on page 53.

3.9 Arrays

In Java, an array is an object, so it has properties and methods. For example, in Java there is a `length` property, so you can pass around an array and know its length using the property. Here's a Java method that finds the number of zeros in an array of integers:

```
public int zeros( int[] anArray ) {
   int count=0;
   for(int i=0; i<anArray.length; i++) {
      if( anArray[i] == 0 ) count++;
   }
   return count;
}
```

Here's how you would have to do that in C:

```
int zeros( int anArray[], int len ) {
   int count=0;
   int i=0;
   for(i=0; i<len; i++) {
      if( anArray[i] == 0 ) count++;
   }
   return count;
}
```

We have to add another argument to pass in the length. This all stems from the fact that in C, an array is simply a sequence of consecutive memory locations, with no marker of the beginning or the end. The array's name (like `anArray`, above) is just the address of its first element. The locations of other elements are computed from that base, with no bounds checking at all, because the program simply doesn't know what the bounds are.[5]

You also noticed the difference in how the array is declared. The C style is possible in Java as well, though uncommon, but in C, you must put the brackets, `[]`, *after* the variable name.

You can initialize an array when it is declared, if you like:

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

If the number of initializations is smaller than the number of elements of the array, the remaining elements are not initialized, just as you'd expect.

Many languages have multi-dimensional arrays, which can be quite useful. For example, a spreadsheet might be created using a two-dimensional array, or a linear algebra program might represent matrices with two-dimensional arrays. Internally, these multi-dimensional arrays are represented by a single array that holds all the elements, with a simple mathematical function mapping from multi-dimensional indices such as (i, j) or (i, j, k) to the correct one-dimensional index n. (You could try to derive the function yourself; it's not hard.) C, however, doesn't represent multi-dimensional arrays this way. Instead, C has arrays of arrays. To represent a two dimensional array, for example, there would be a one-dimensional "backbone" array, each of whose elements is a one-dimensional array for a row.

In practice, you will barely see the difference in internal representation. Here is an example of a program that creates and prints a multiplication table up to 10 by 10:

[5] Of course, under the covers, Java's arrays are also sequences of consecutive memory locations, but it has wrapped them up in a class that records the array's length and checks indices before accessing the array. In our `zeros` example, if the array has 100 elements, the Java code will probably check all 100 indices for correctness, which seems excessive, while the C implementation will check none. Speed versus safety.

```
#include <stdio.h>

int main() {
    int a[10][10];
    int i, j;

    /* Create multiplication table */
    for( i=0; i<10; i++ )
        for( j=0; j<10; j++ )
            a[i][j] = i*j;
    /* Now, print it out */
    for( i=0; i<10; i++ ) {
        for( j=0; j<9; j++ )
            printf("%d\t", a[i][j]);
        printf("%d\n", a[i][9]);
    }
}
```

You will see the effects of the array-of-arrays representation if you want to dynamically allocate a multi-dimensional array. We will cover this later. (See section 3.14.)

In C and C++, you can't resize an array, as you can in Java. However, in C++, the Standard Template Library (see part 7.2) adds the `Vector` class, which is like the Java Array class, and has a `length` property, allows resizing and so forth.

3.10 Strings

C does not have "strings" as a built-in datatype. Instead, it has arrays of characters, and in C terminology, a string is simply an array of characters. Thus, we can create a string as follows:

```
char s1[100];
```

This string has enough room to store 100 characters. Of course, since C doesn't check array bounds, you still have to be careful not to store any characters after the 100th.

Using the approach for initializing arrays, you could initialize a string:

```
char s2[10] = { 'f', 'r', 'i', 'e', 'n', 'd', 'l', 'y' };
```

This is very tedious, though. We'll see a better approach a bit later.

Suppose you have a string stored in one array (of length `len`), and you want to copy its contents to another. You could write code like this:

```
void copy_string( char dest[], char source[], int len ) {
   for( i=0; i<len; i++) {
      dest[i] = source[i];
      }
   }
```

Of course, you would have to know how long the arrays were, so that you could supply a value for `len`.

However, rather than pass around string lengths, C almost always uses a different approach, which is to have an explicit marker for the end of the string. Since a string is an array of characters, this marker must be a character. C might have used the newline character or the period or some other, but it chose to use the character whose ASCII value is zero, also called the null character or the null byte. We saw this character earlier, in the section on characters. For this reason, C-style strings are often called null-terminated strings.[6] The copy function above might be re-written as:

[6] You'll sometimes hear these referred to as ASCIIZ strings, the Z for the zero byte at the end. That terminology seems to be dying out, though.

```
void copy_string2( char dest[], char source[]) {
   int i = 0;
   while( source[i] != '\0' ) {
      dest[i] = source[i];
      i++;
   }
   dest[i] = source[i];
}
```

Note that both of these functions will copy characters until they reach their stopping condition, regardless of the amount of space allocated to the destination string. This is a common source of error.

It's not necessary to implement either of these string copying functions yourself. They have already been implemented in the `<string.h>` library. (See section 3.8, above.) They include:

```
strcpy(dest,source);      /* copy a string to null byte*/
strncpy(dest,source,len); /* copy a string given length */
strlen(source);           /* return the length of a null-terminated string */
strcmp(x,y);              /* compare null-terminated strings */
```

3.11 Structs

One of the things you do with classes in Java is to collect related data together as a set of instance variables. You also provide methods to operate on them, but we will put that aside for now and return to it later. Suppose we wanted to define a "point" object in Java. We might start like this:

```
public class Point {
    public int x;
    public int y;

    public static void main( String[] args ) {
        Point origin = new Point();
        Point P1     = new Point();
        origin.x = 0;
        origin.y = 0;
        P1.x = 42;
        P1.y = 13;
    }
}
```

Notice that this also creates a "type" that we can use for declaring variables, as we did in `main`.

The rough equivalent in C is to create a "struct" (short for "structure"). The following example comes close to mimicking Java's syntax:

```
struct Point {
    int x;
    int y;
};

int main() {
    struct Point origin;
    struct Point P1;
    origin.x = 0;
    origin.y = 0;
    P1.x = 42;
    P1.y = 13;
}
```

However, there is a better way of handling this idea in C, which we will defer until after we've discussed pointers.

3.12 Unions

write this

3.13 Pointers

Pointers are an important datatype in C, while in Java, pointers seem to be nearly non-existent. Java does, in fact, have pointers, but the language has tried to make them fairly invisible. Also, Java controls its pointers much more than C does, while C is perfectly willing to let you shoot yourself in the foot. Thus, it's difficult to even compare the languages with respect to pointers. Instead, I will just describe pointers as they are used in C, and we can instead compare how certain pointer-based data structures are created.

Pointers are crucial in all languages, even though they are handled differently in different languages. Why are they so important? One major reason is that pointers are lightweight things, while many of the things we use in programs are big, heavy things. For example, an object in a graphics program could well have instance variables for the (x, y, z) location of the object, the sets of vertices and edges that it has, the color of each line, the texture of each face, and possibly bitmaps and pictures that will be mapped onto different faces. The total size in memory could easily be tens of thousands of bytes. You don't want to have multiple copies of those in memory, you don't want to have to move them around (copy them from place to place), and you don't want to be constantly creating and destroying them. Pointers, on the other hand, are just a single word of memory (4 bytes on most modern machines), which is literally the easiest and most efficient quantity to copy around.

If you have a datatype that you want to point to, you denote this syntactically by putting an asterisk (star) after the datatype, like this:

```
int*  px;   /* px points to an integer */
char* pc;   /* pc points to a character */
```

Note that neither of these pointers has any value. They *can* point to an integer or a character, but they don't at the moment. Recall that C does not initialize variables. Later, we'll see how to make these variables point to things. As a slight variation on the syntax, you will very often see the asterisk next to the variable name instead of the datatype. Thus, the example above will often be written as:

```
int  *px;   /* px points to an integer */
char *pc;   /* pc points to a character */
```

Personally, I consider the "star on the variable" syntax to be old-style and needlessly confusing, but it is in common use, so I feel obliged to tell you about it, in case you're reading someone else's code, or are asked to use that style by your employer. It also has some advantages, as with a list of variables:

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

Here, `x` and `z` are both integers, and `y` is a pointer to an integer.

3.13.1 Pointers and Addresses

Suppose we have some data and a pointer to that data. To keep it simple, lets use integers:

```
int x;
int* px;
```

We all know how to make `x` have a value; what about `px`? A pointer is an address, so we want to give `px` the address of an integer. We have one right there, so let's take the address of it. For that, we need the C "address of" operator, which is an ampersand. Think about the following example:

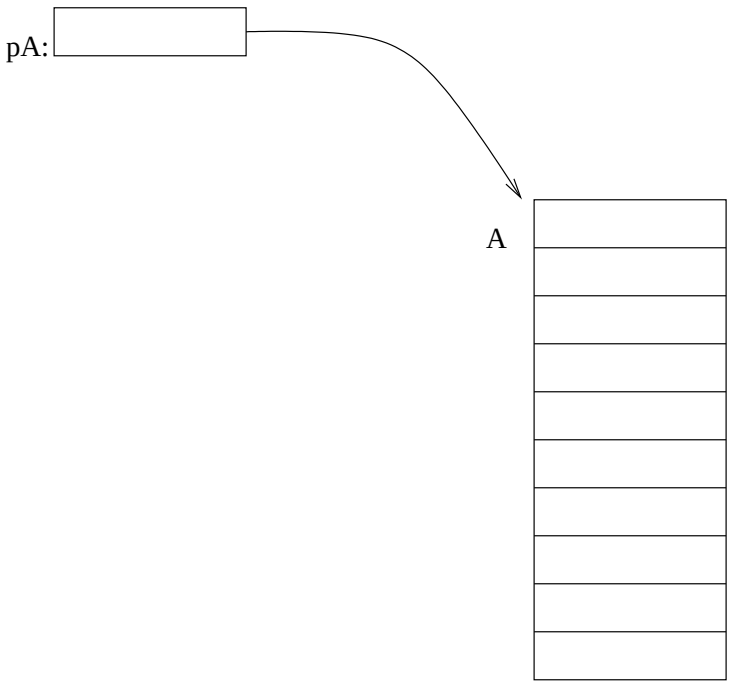

Figure 3.1: An array and a pointer to its first element

```
px = &x;
*px = 5;                    // x=5
*px = *px + 1;              // x=x+1
if( x == 6 ) return true;
```

- The first line sets `px` to be the address of `x`, so it points to the same memory location that `x` corresponds to. (Think back to your assembly language class: it's code like this that makes C be described as "high level assembly language.")
- The second line "dereferences" the pointer: it says to make the thing pointed to by `px` equal to 5. In other words, it's entirely equivalent to the code in the comment.
- The third line just reinforces the point about dereferencing: the `*px` on the RHS of the statement means "the thing that `px` points to," and since `px` is declared to point to an integer, that starred expression is an integer (in this case, 5).
- The fourth line will be true in any implementation of C/C++.

Rarely will we want to point to integers; that was just for warm-up. Integers, after all, are small like pointers, so there's no problem with copying them around. The fact that the preceding code is possible does not mean anyone advocates it. Blot it from your memory. Done? Good.

3.13.2 Pointers and Arrays

Suppose that we have an array of integers. C makes the fundamental commitment that an array is just a sequence of consecutive memory locations (addresses) and that the name of an array is just the first of those addresses. Since an address is a pointer, we find that the following surprising assignment can be made, which is illustrated in figure 3.1.

```
int A[10];
int* pA = A;
```

Furthermore, the pointer `pA` and the array `A` are, in C, fundamentally *the same*, so you can do the following:

```
pA[1] = 5;
```

Indeed, `pA` is an array just as much as `A` is. C programs often take advantage of this "pun" or *double entendre*: that a pointer is the same as an array. Indeed, you can declare a variable as an array of no elements:

```
int  B[];
int* C;
```

The variables `B` and `C` have the same datatype!

3.13.3 Pointers and Multi-dimensional Arrays

Since a pointer is the same as an array, you can have an array of arrays by putting some one-dimensional arrays into a "container" array. Here's a brief example, where we actually create a "triangular" array, where the first row has 3 elements, the second two and the third one:

```
int row1[3] = { 1, 2, 3 };
int row2[2] = { 4, 5 };
int row3[1] = { 6 };
int container[][] = { row1, row2, row3 };
```

You will rarely create an array this way. Typically, the array is rectangular. If it's a static array, you will initialize it like this:

```
int matrix[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} };
```

Alternatively, the array's elements will be dynamically computed by your code, in which case you'll just declare the array in a simple way and let the code do the rest:

```
int matrix[3][3];
for(i=0;i<3;i++) {
   for(j=0;j<3;j++) {
      matrix[i][j] = i*3+j+1;
   }
}
```

In either case, you can treat the array as a two-dimensional structure and not think about how it is represented. While you're still learning the language, don't worry about the details of array representation. Later, when it becomes an issue, as when you are dynamically allocating two-dimensional arrays, you can review this.

3.13.4 Pointers and Characters

Since arrays and pointers are the same, a string can be viewed or treated as a pointer to its first character. Therefore, the following is the standard way of declaring a string pointer in C:

```
char* myString;
```

A string literal has the datatype of `char*`—a pointer to a character, so you can initialize a string variable like this:

```
char* myString = "This is a string literal.";
```

This is entirely analogous to the earlier discussion of arrays and pointers, so figure 3.1 also illustrates this concept.

3.13.5 Pointers and Structs

Earlier, we briefly mentioned structs. Structs are similar to arrays in that they are composite data structures, comprising multiple memory locations. Thus, arrays and structs are typically "big" things: too big to pass to a function. It is far more efficient to pass a pointer to the array or structure. A pointer to an array is, we've learned, a trivial thing, since the name of the array is equivalent to a pointer.

A pointer to a structure is not quite so trivial, but is still not too hard, and it is one of the primary ways that pointers are used in C and C++. Let's imagine that we want to define an object called "student" that comprises several disparate pieces of information. Here is how we might do it in Java:

```
public class Student {
    public String name;
    public int shoeSize;
    public float gpa;

    public static void main( String[] args ) {
        Student bart = new Student();
        bart.name = "Bart Simpson";
        bart.gpa = 0.5f;             // "f" because float, not double
        Student lisa = new Student();
        lisa.name = "Lisa Simpson";
        lisa.gpa = 4.0f;
    }
}
```

Here is how we might do it in C, using pointers and a library function called `malloc` that we will discuss soon. (Section 3.14, page 33). For now, you can think of it as analogous to Java's "new" operator, which doesn't exist in C.

```
struct Student {
    char* name;
    int shoeSize;
    float gpa;
};

int main() {
    struct Student* bart = (struct Student*) malloc(sizeof(struct Student));
    struct Student* lisa = (struct Student*) malloc(sizeof(struct Student));
    bart->name = "Bart Simpson";
    bart->gpa = 0.5;
    lisa->name = "Lisa Simpson";
    lisa->gpa = 4.0;
}
```

Contrast this to the earlier example of the `point` struct. Notice that the members of the structure are accessed using the C "arrow operator," which is just a hyphen followed by a greater-than:

```
bart->gpa = 0.5;
```

We use the arrow syntax in C any time we have a pointer to a structure and we are accessing an element. You'll notice that on the preceding line, we declared `bart` to be a pointer to a student, not a student:

```
struct Student* bart = (struct Student*) malloc(sizeof(struct Student));
```

The example code gives us pointers to structures, which are then easier to pass around if you should want to do so. (However, if you're not going to pass structures around, and you prefer the syntax of the previous "point" example, feel free.)

There are two main differences between the "Point" example (back in section 3.11 on page 26) and the "Student" example from this section: how the variables are declared and initialized, and how the elements are accessed. Not surprisingly, these are tightly related.

- In the "Point" example, the variables are declared to be structs, so we don't need to do "new" any more than we have to use "new" with ints or floats. Elements of such structs are accessed through the variable by just using a *dot*.
- In the "Student" example, the variables are declared to be *pointers to* structs. This means we have to use `malloc` (the C analog of Java's `new` operator—see section 3.14) to initialize the variable. Elements of such structs are accessed through the variable by the *arrow* notation. For more information, see section 3.13 (page 27).

If you're comparing the syntax and semantics of programming languages, there's an interesting disconnection here. Java borrowed the "dot" syntax for accessing members of a class (instance variables and methods), but the semantics of its variables are more like the "Student" example, which uses the "arrow" syntax. The Student example actually mimicks Java's semantics better. See section 3.14 (page 33) and section 5.4 (page 51).

You also noticed that the Java code declared the instance variables to be public, though such usage is frowned upon in OOP because clients should go through your method interface. You probably were taught to define the class thus:

```
public class Student2 {
    private String name;
    private int shoeSize;
    private float gpa;

    public void setName( String name ) {
        this.name = name;
    }

    public void setGPA( float gpa ) {
        this.gpa = gpa;
    }

    public static void main( String[] args ) {
        Student2 bart = new Student2();
        bart.setName("Bart Simpson");
        bart.setGPA(0.5f);
        Student2 lisa = new Student2();
        lisa.setName("Lisa Simpson");
        lisa.setGPA(4.0f);
    }
}
```

We can, of course, define and use such an interface in C. Notice the use of pointers in passing a struct to a function. Of course, we will name our interface functions a bit differently, but those differences are primarily cosmetic.

```
struct Student {
    char* name;
    int shoeSize;
    float gpa;
};

void Student_setName( struct Student* it, char* name ) {
```

```
    it->name = name;
}

void Student_setGPA( struct Student* it, float gpa ) {
    it->gpa = gpa;
}

int main() {
    struct Student* bart = (struct Student*) malloc(sizeof(struct Student));
    struct Student* lisa = (struct Student*) malloc(sizeof(struct Student));
    Student_setName(bart, "Bart Simpson");
    Student_setGPA(bart, 0.5);
    Student_setName(lisa, "Lisa Simpson");
    Student_setGPA(lisa, 4.0);
}
```

However, in C, we can't protect the members of a struct by declaring them to be private. Members of a struct are (implicitly) public, and we can't change that. (However, C++ allows struct members to be declared private.) Thus, it's possible for a client to bypass your interface and mess with the representation of your structs. C was invented back when programmers were considered trustworthy `:-)`.

3.13.6 Using Typedef

The syntax in the previous "Student" and "Point" examples was very cumbersome: there was constant repetition of `struct Student` or `struct Point`. This can be streamlined by using `typedef` to define a new *type*, parallel to built-in types like `int` or `char`.

Note that this is another place where Java (and C++) have improved upon C. Those languages create a new type whenever a class is defined. So, a Java definition like:

```
class MySillyClass {
   ...
}
```

immediately and automatically allows the use of the name as the type of a variable:

```
MySillyClass rodents;
```

This isn't the case in C. However, it's possible to use `typedef` to make it nearly as easy. We can use typedef to improve the "Student" example, as follows:

```
typedef struct {
    char* name;
    int shoeSize;
    float gpa;
} Student;

typedef Student* pStudent;

void Student_setName( pStudent it, char* name ) {
    it->name = name;
}

void Student_setGPA( pStudent it, float gpa ) {
    it->gpa = gpa;
}
```

```
int main() {
    pStudent bart = (pStudent) malloc(sizeof(Student));
    pStudent lisa = (pStudent) malloc(sizeof(Student));
    Student_setName(bart, "Bart Simpson");
    Student_setGPA(bart, 0.5);
    Student_setName(lisa, "Lisa Simpson");
    Student_setGPA(lisa, 4.0);
}
```

3.14 Allocating Memory

In Java, we create new objects using the `new` operator. Among other things, what `new` does is allocate memory from a place called "the heap." In C, memory is allocated using a library function called `malloc`. In Java, we don't have to worry about returning storage when we're done with an object, because Java is has garbage collection, but in C, we have to manage our own storage, recycling objects when they're no longer in use.

The `malloc` function returns a *pointer* to the allocated memory, so the result must always be stored in a pointer variable. (See section 3.13, above.) Also, because `malloc` doesn't know what kind of object is being created, it just returns an object of type `void*`—a pointer to void is a C idiom that means a pointer to some unknown chunk of memory. Thus, you should always *cast* (see section 3.6.4 on page 20) the result of a call to `malloc` to the correct pointer type before storing it in the pointer variable.

Unlike `new`, which takes the name of a type as its argument, `malloc` wants a *number*—how many bytes to allocate. Fortunately, there is a built-in function called `sizeof` that takes a type name and returns the size of the object in bytes. Suppose we wanted to allocate space to store an integer. (This would be relatively rare, but could happen.) We would do the following:

```
int* px = (int*) malloc( sizeof(int) ) ;
```

What this bizarre-looking statement says is that `px` is a pointer to an integer and it will point to the result of the call to `malloc`, with the type properly cast. The `malloc` function has been asked to return enough space to store one integer.

3.14.1 Allocating Structs

In practice, you typically use dynamic memory allocation for structures and arrays. To create a `Student` structure (see the examples in section 3.11, above), we do the following:

```
Student* poindexter = (Student*) malloc(sizeof(Student));
```

From now on, you can use the memory pointed to by `poindexter` just as you would any pointer to a structure:

```
poindexter->name = "Muffy Worthington";
```

3.14.2 Freeing Space

Unlike Java, C is not a garbage-collected language, so an ideal program must always return to the heap all the memory it allocates. This is not important for programs that will run briefly and exit before memory is exhausted, but for long-running programs (such as server demons), memory must be carefully managed. Memory is returned to the heap using the `free` function, handing it the same pointer you got from `malloc`:

```
free(poindexter);
```

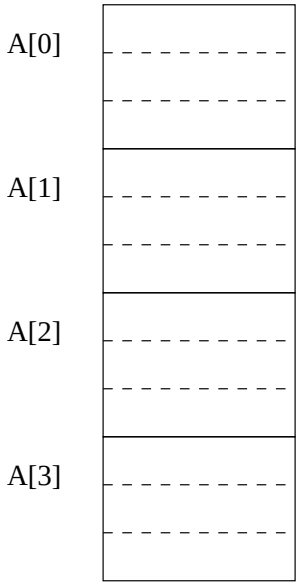

Figure 3.2: An array of "Student" objects

3.14.3 Allocating Arrays

Since arrays are just consecutive memory locations, allocating an array at run-time is fairly straightforward. Suppose that an int takes n bytes of storage; then an array of 10 ints would take $10n$ bytes of storage. Thus, to allocate an array of things, we just ask for the proper number of bytes. Here's an example that allocates an array of ints:

```
int* A = (int*) malloc(n*sizeof(int));
```

Note that we must use `int*` in the type casting; you can't cast something to be an array, even though arrays and pointers are the same.

Similarly, to allocate an array of "students," we can do the following. Figure 3.2 shows what the data structure looks like.

```
Student* B = (Student*) malloc(n*sizeof(Student));
```

Note that an array of structures is different from an array of pointers to structures. To do that, we'd use the following code; figure 3.3 shows the result.

```
Student** A = (Student**) malloc(n*sizeof(Student*));
for(i=0;i<n;i++) A[i] = (Student*) malloc(\sizeof(Student));
```

Here's a complete example, including a fancy use of `main` (see section 3.18, below).

```
#include <stdio.h>
#include <stdlib.h>

/* Creates and fills in a triangular array, with elements consecutively
   numbered.  The size is given on the command line.  Primarily
   demonstrates dynamic array allocation. */

int main( int argc, char* argv[] ) {
    int i,j;
    int elt = 0;                /* The element number */
```

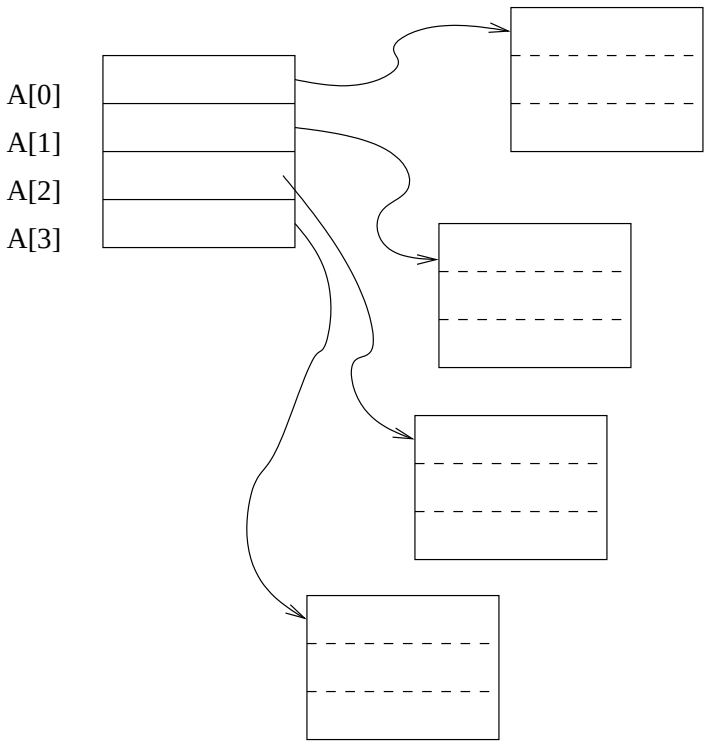

Figure 3.3: An array of pointers to “Student” objects

```
int len;                    /* the size of the arrays */
int** A;                    /* A is an array of arrays */

if(argc < 2) {
    printf("Usage: %s size\n", argv[0]);
    exit(1);
}
len = atoi(argv[1]);       /* decode size */

/* assign A to be an array of pointers  */
A = (int**) malloc(len*sizeof(int*));

/* fill in the rows of the array */
for(i=0; i<len; i++) {
    /* assign each element to be an array of ints */
    A[i] = (int*) malloc((len-i)*sizeof(int));
    for(j=0; j< (len-i) ; j++) {
        A[i][j] = ++elt;
    }
}

/* Now, print the result. */
for(i=0; i<len; i++) {
    for(j=0; j< (len-i) ; j++)
        printf("%d\t", A[i][j] );
    printf("\n");
```

```
    }
}
```

That last program is short but fairly complicated, so don't worry if you can't read it easily. Re-visit it on occasion as you become more comfortable with C and want to learn more from the example.

3.15 Input

Input in C is done similarly to output, strangely enough. While there are low-level functions (one is called `read`,which we'll discuss later), a lot of I/O is done using `scanf`. Like `printf`, `scanf` takes a string as its first argument, indicating the format that the input will be in, with placeholders indicating data that will be read and stored into the variables that are supplied as additional arguments. More precisely, `scanf` is given the *addresses* of the variables, requiring the use of the `&` operator.[7]

Here's an example, a little program that computes the means of two columns of integers. For simplicity, the program assumes exactly ten numbers.

```
#include <stdio.h>

int main() {
    int sumx = 0;
    int sumy = 0;
    int i, x, y;

    for( i=0; i<10; i++ ) {
        scanf("%d %d\n", &x, &y);
        sumx += x;
        sumy += y;
    }
    printf("Sums are %d and %d\n", sumx, sumy);
    printf("Means are %f and %f\n", sumx/10.0, sumy/10.0);
    return 0;
}
```

Notice the use of the ampersand on the input variables! Also, the same letters are used to denote different types of variables as are used in `printf`.

The other characters in the `scanf` string are matched against the input. In the previous example, we specified that the inputs would be separated with a single space, but we could have specified that they were separated by a comma, or three colons or whatever. See the manual page on `scanf` for more information.

It ought to be that easy. Unfortunately, there is a bug in `scanf` so that sometimes it hangs (never returns) when reading from the keyboard. It turns out to be better to read a line of input from the keyboard into a "buffer" (a temporary storage location) and then scan that buffer. Thus, we use

```
char buf[80];
...
fgets(buf,sizeof(buf),stdin);
sscanf(buf,format, args...);
```

Here's our example, modified to use `fgets` and `sscanf`:

```
#include <stdio.h>
```

[7]This C operator is omitted in Java, which does its best to hide pointers from us. If you put an ampersand (`&`) in front of a variable, you get the address of the variable. Until you get to much more sophisticated programs, the only place you'll see an ampersand is in a `scanf` function and some other call-by-reference functions. See section 3.17.

```
int main() {
    char buf[80];
    int sumx = 0;
    int sumy = 0;
    int i, x, y;

    for( i=0; i<10; i++ ) {
        fgets(buf,sizeof(buf),stdin);
        sscanf(buf,"%d %d\n", &x, &y);
        sumx += x;
        sumy += y;
    }
    printf("Sums are %d and %d\n", sumx, sumy);
    printf("Means are %f and %f\n", sumx/10.0, sumy/10.0);
    return 0;
}
```

3.16 File I/O

write this

3.17 Call by Value

C functions are all call-by-value. This means you have to pass a *pointer* to an object if you want the function to be able to modify the contents of the object. Generally, a function cannot modify an argument. For example, the following program prints 3, not 7.

```
#include <stdio.h>

void set_to_7(int x) {
    x = 7;
}

int main() {
    int a = 3;
    set_to_7(a);
    printf("%d\n", a);
}
```

Passing a pointer works because it's not the pointer that is being changed, but the object that is pointed to. Thus, the `Student_setName` function from the earlier example works fine.

If you want to do call-by-reference, you have to do it by passing the address of a variable, which has the datatype of a pointer to the variable's type. Here's an example where the program does print 7. Notice that we pass the address of the variable `a` by prefixing it with the address-of operator, which is the ampersand (`&`). Then, we store into the variable by using the pointer operator, which is the asterisk (`*`).

```
#include <stdio.h>

void set_to_7(int* x) {
    *x = 7;
}

int main() {
    int a = 3;
```

```
    set_to_7(&a);
    printf("%d\n", a);
}
```

However, if you have a pointer to a structure, the call-by-value versus call-by-reference issue doesn't cause any difficulty. The following example shows both, and the print statement prints 7 and 7.

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void set_to_7(struct Point* pt) {
    pt->x = 7;
}

int main() {
    struct Point p1;
    struct Point* p2 = (struct Point*) malloc(sizeof(struct Point));
    set_to_7(&p1);
    set_to_7(p2);
    printf("%d and %d \n", p1.x, p2->x);
}
```

The simplicity of passing around pointers to structures is a reason for using the `malloc` form.

3.18 Main

Execution of a C/C++ program begins with `main`. Obviously, something like this was adopted by Java, but there are a few differences. In C/C++, there must be only one `main` in the entire program. This is true of all functions, of course, but you probably wouldn't be tempted to have more than one of other kinds of functions. In Java, it's common to define a `main` method for each class, which then gets invoked if that class is named on the `java` command line and is otherwise ignored. It's a clever idea, but it's not available in C/C++, where a program must have a unique starting point specified at compile time.

Note that there are two ways to declare `main`:

```
int main() {
...
```

and, if you want to use command-line arguments:

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

The semantics are the same as they are in Java: `main` is supplied with an array of strings, one for each command-line argument. The differences are because in C we have to be told how long an array is, and because we use `char*` instead of `String`. The variable `argc` tells us the argument count—how many arguments we got. The variable `argv` is the argument vector—an array of strings. (Recall that `char*` is the datatype of a string.

Because of the duality of pointers and arrays, you will often see `main` declared as follows:

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

Here's an example that just prints its arguments back out, similar to a program called `echo` in the Unix world.

```
#include <stdio.h>

int main( int argc, char* argv[] ) {
    int i=0;

    printf("There were %d arguments:\n",argc);
    for( i=1; i<argc; i++ ) {
        printf("%s ",argv[i]);
    }
    printf("\n");
    return 0;
}
```

If you want to try this example, be sure to save it as `myecho` or something like that, not `echo` or `test`, since there are already commands with those names, built into the shell, which will take precedence over yours.

3.19 Short-Circuit Operators

C's logical operators like `&&` and `||` stop evaluating if the left operand determines the value, so they are "short-circuiting" operators. This is true in Java as well, so it should be familiar to you. Here's an example that is useful:

```
#include <stdio.h>

int main() {
    int A[10];
    int n;

    if( n < 10 && n >= 0 && A[n] > 15 ) {
        printf("A[n] is %d\n", A[n] );
    }
}
```

Here, we're able to test that `n` is a valid array index and `A[n]` has some property all in one test. If we didn't have short circuiting, we'd have to use the slightly more cumbersome:

```
#include <stdio.h>

int main() {
    int A[10];
    int n;

    if( n < 10 && n >= 0 ) {
        if( A[n] > 15 ) {
            printf("A[n] is %d\n", A[n] );
        }
    }
}
```

3.20 Bitwise Operators

C has operators to read and modify particular bits within a word. Java has the same operators, but they are less commonly used, just because the kind of applications written in Java tend to be more high-level, while C is commonly used for low-level "bit-twiddling" applications.

operation	char
bitwise and	`&`
bitwise or	`|`
bitwise xor	`^`

To understand the use of these operators, you have to remember how integers are represented. An assignment like this:

```
x = 5;
y = 9;
```

puts particular bit patterns into memory. The binary representation of 5 is 101. Of course, on a 32-bit computer, the actual representation has 29 leading zeros. For brevity, let's assume we have an old 8-bit computer, so the bit-pattern is 00000101. The bit pattern for 9 is 00001001.

If we have an assignment like this:

```
z = x & y;
```

The computation is the *bitwise* AND of the bits of `x` and the bits of `y`. That means that the ith bit of each integer is matched up and combined using AND to produce a new number, which is then stored in `z`. In this case, `z` will get the bit-pattern 00000001, because the only place where both `x` and `y` contain a 1 is in the rightmost bit. Similarly:

```
z = x | y ;
z = x ^ y ;
```

The first is inclusive OR, so the result is 00001101; the second is exclusive OR, with 00001100 as the result.

Say more here

3.21 The Ternary Operator

Every computer language has unary operators (such as `!`, which produces the logical negative of a single boolean value) and binary operators (such as `&&` which produces the logical AND of two boolean values). As far as I know, only C and its successors have an infix *ternary* operator: an operator that operates on *three* values.

The semantics of the operator is like that of `if...else`, so let's explain the ternary operator with an example. Here's one way to define the `max` function:

```
int max( int x, int y ) {
   if( x > y ) {
      return x;
   } else {
      return y;
   }
}
```

If we use the ternary operator, denoted `?`, we can re-write that function as:

```
int max( int x, int y ) {
    return x > y ? x : y ;
}
```

The question mark operator follows its first value. If that first value is true (non-zero), it returns the value preceding the colon; otherwise, it returns the value following the colon. (In C, comparisons return boolean (integer) values, which can be stored, printed and so forth.)

The important difference between `if...else` and `?` is that the operator is an *expression* and so it can be embedded in other expressions. Above, we see it as the argument of the `return` keyword.

The ternary operator is maligned because it would be easy to write completely unreadable code using it, but for purposes like `max`, it can be a wonderful shorthand. For that reason, it was adopted in C's successor languages.

3.22 The Pre-processor

There are a few common issues of programming languages that C solves by use of an external program, called the pre-processor. This is a program that reads the source file before the compiler does, making various textual substitutions. The Unix name for the program is `cpp`, and it is sometimes used with other programming and configuration languages as well.

Definitions and other information that is meant to be read by the pre-processor and not by the C compiler are called *directives*. The CPP directives are begun at the left margin with a pound sign (`#`). The following sections briefly describe the issues the C assigns to the pre-processor.

3.22.1 Constants

The C language doesn't have constants ("read-only" variables) as many other languages do. Instead, it uses the pre-processor to substitute a string of digits wherever a string of characters is found. For example:

```
#define PI 3.1415926
#define ARRAY_MAX 100
```

The two letters `PI` will have the number substituted wherever they are found, so you can use this as a constant wherever a floating point number is allowed. In many cases this is just what you want. Similarly for constant `ARRAY_MAX`, so you can do things like this:

```
int A[ARRAY_MAX];
float cospi = cos(PI);
```

Conventionally, constants are all upper-case, so that they stand out and so that you don't use them accidentally.

Warning: the pre-processor is a purely textual substitution, so using such a constant where a constant is not allowed will cause weird and incomprehensible error messages. For example, given the definition of `ARRAY_MAX` above, the following line of code:

```
ARRAY_MAX = 150;
```

will result in the cryptic "bad lvalue" error message, because what your program really says (what the compiler saw) was:

```
100 = 150;
```

and you can't assign a new value to 100.

3.22.2 Aliases

Sometimes we want to allow an alias for say, a type. This would allow us to re-define it later. (We saw this same issue above in section 3.13.6 on `typedef`.) Using `typedef` is probably a superior way of accomplishing aliases, but the text substitution of the pre-processor is very powerful, so it is often employed. Here is a simple example:

```
#define number float
```

Because of the textual substitution, we can now use `number` wherever the word `float` would be allowed. If we later decided that numbers should be `double`, we could change just one line of our program.

3.22.3 Imports

You won't be surprised to learn that the pre-processor is used for what Java does with `import`: providing another file of information to the compiler. Thus, the following is actually the work of the pre-processor:

```
#include <stdio.h>
```

What this literally means is that the pre-processor should read this other file and include its contents in place of the directive. The C compiler will have no idea that more than one file is being read.

If you make up your own `include` files, which you will when your programs increase in complexity, you should remember that because the compiler doesn't know about the work of the pre-processor, error messages can sometimes be obscure. For example, a missing semi-colon at the end of an included file could trigger an error message in the main file (or even another included file)!

3.22.4 Conditional Compilation

(This topic is somewhat advanced, so feel free to skip it on a first reading.)

The pre-processor can conditionally include lines of code in its output, depending on various "switches," which can be specified at the top of the source code or even on the command line. Thus, to optionally include some debugging code, the common solution is the following:

```
#ifdef DEBUG
printf("The value of x is %f\n", x);
#endif
```

If the symbol `DEBUG` is defined (as in `#define`), the line of code will be passed on to the compiler, otherwise it won't.

Here's an complete example, using our factorial example from the very beginning:

```
#include <stdio.h>              /* for printf */
#include <stdlib.h>             /* for atoi */

#define TRACE

#ifdef TRACE
void indent(n) {
    int i;
    for( i=0; i<n; i++) printf("  ");
}
#endif

int factorial( int );           /* declaration */

int factorial( int n ) {        /* definition */
    int result;

#ifdef TRACE
    indent(n);
    printf("factorial of %d ... \n",n);
#endif
    if( 0 == n ) {
        result= 1;
    } else {
        result = n*factorial(n-1);
    }
#ifdef TRACE
```

```
    indent(n);
    printf("factorial of %d is %d\n",n,result);
#endif
    return result;
}

int main( int argc, char* argv[] ) {
    int answer = factorial(atoi(argv[1]));
    printf("%s! = %d\n", argv[1],answer);
}
```

The lines surrounded by the pre-processor directives `#ifdef` and `#endif` will only be compiled if that symbol is defined. Compiling and running the program as it is, yields the following result:

```
% gcc -o factorial-trace factorial-trace.c
% factorial-trace 4
4! = 24
```

Notice how there is no "tracing" output. If we define that symbol, we get the tracing behavior as we did before.

How can we define the symbol? One way is to insert a line like the following at the top of the file:

```
#define TRACE
```

Another way to define the symbol is on the compilation command line, with the `-D` switch to gcc. For example:

```
% gcc -o factorial-trace -DTRACE factorial-trace.c
% factorial-trace 4
        factorial of 4 ...
      factorial of 3 ...
    factorial of 2 ...
  factorial of 1 ...
factorial of 0 ...
factorial of 0 is 1
  factorial of 1 is 1
    factorial of 2 is 2
      factorial of 3 is 6
        factorial of 4 is 24
4! = 24
```

3.23 Static Variables

Static variables are obscure, but occasionally useful. As a novice C programmer, you will probably never use them, but you may see them in other people's code. Essentially, they are variables that, like global variables, are initialized when the program begins, not when the function does. Thus, their values last between function calls. Here's an example that acts like a one-element cache:

```
function foo(int x) {
   static int last_x = 0;
   static int last_y = 0;
   if( last_x == x ) return last_y;
   ... long expensive computation of y
   last_x = x;
   last_y = y;
}
```

The idea is that the function remembers the last argument it was called with and the result, so that if it's called again with the same argument, it doesn't have to re-compute it (which is expensive) but can immediately return the previously computed value.

3.24 Examples

In this section, we just compile some examples of code in Java and C, to contrast the coding styles.

3.24.1 Hypotenuse

The first example is a very simple program to read two numbers from the user, compute the hypotenuse of a right triangle with legs of those lengths, and print the result. The computation is easy, so we can concentrate on syntax and such.

First, in Java:

```
class Hypotenuse {
    public static double hypotenuse( double a, double b ) {
return Math.sqrt(a*a+b*b);
    }
}

class TestHypo {
    public static void main( String[] args ) {
double c = Hypotenuse.hypotenuse(3,4);
System.out.println("Hypotenuse of a 3,4 right triangle is "+c);
    }
}
```

Now, in C:

```
#include <stdio.h>              /* for printf and scanf */
#include <math.h>               /* for sqrt  */

float hypotenuse(float a, float b) {
    return sqrt(a*a+b*b);
}

int main() {
    float x, y, z;
    printf("What is the first leg? ");
    scanf("%f", &x);
    printf("What is the second leg? ");
    scanf("%f", &y);
    z = hypotenuse(x,y);
    printf("The result is %f\n", z);
}
```

3.24.2 Factorial

The next example is the classic factorial program that every student who has ever faced recursion knows well. Just to make it slightly more interesting, we'll compute and print the factorial of numbers given on the command line.

First, in Java:

```
class Factorial {

    public static int factorial( int n ) {
        if( 0 == n ) {
            return 1;
        } else {
            return n*factorial(n-1);
        }
    }

    public static void main( String[] args ) {
        for( int i=0 ; i < args.length ; i++ ) {
            System.out.println(args[i]+"! = "+
                               factorial(Integer.parseInt(args[i])));
        }
    }
}
```

Now in C:

```
#include <stdio.h>              /* for printf */
#include <stdlib.h>             /* for atoi */

int factorial( int n ) {
    if( 0 == n ) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}

int main( int argc, char* args[] ) {
    int i;
    for( i=0 ; i < argc ; i++ ) {
        printf("%s! = %d\n", args[i], factorial(atoi(args[i])));
    }
}
```

Of course, this example is entirely not object-oriented, so it doesn't show any of the nice OO features of Java. The "students" example from section 3.11, above, shows data management in both Java and C.

3.24.3 String Modification

The following program reads in a string, then computes a new string by adding one to each charater and prints the result. Notice how we measure the length of the original string and allocate space for the new one, and manage the null byte at the end.

```
#include <stdio.h>              /* for printf, scanf */

char* strmod(char* original) {
    int len = strlen(original);
    char* copy = (char*) malloc(len+1);
```

```
    int i=0;

    for(i=0;i<len;i++) {
        printf("%c\n",original[i]);
        copy[i] = original[i] + 1;
    }
    copy[len] = '\0';           /* null byte */
    return copy;
}

int main() {
    char* s;
    printf("What is the string? ");
    /* This only reads to the first space. */
    scanf("%s", s);
    printf("Result is %s\n", strmod(s));
}
```

3.24.4 Others

I'm open to suggestions.

Chapter 4

C++ Programming

4.1 Advice

Before we start, some advice.

- For any class you define, go to the effort of writing a print method for it, right away, even if you won't use it in the final program. At some point in your debugging, you'll certainly need to print out an object of that type, and if you have to defined a method then, you might not. It really will be more efficient in the long run.

  Note that if you want to be a fancy C++ hacker, you should overload the << operator, but that's not necessary.

  See the demo2 and demo3 examples.

- Initially, err on the side of making everything public in your classes. There's no sense wrestling with "permission" along with other bugs. While I believe that proper design is important and you should certainly think about what should and shouldn't be private (put your thoughts in a comment), there's no great purpose in turning your program into a "police state."

- Initially, don't worry about using "delete" to free up unused objects. You will eventually need to do that, and it is worth thinking about this beforehand and putting a comment in your code to that effect:

  ```
  Block* b1 = new Block;
  ...
  // delete b1;
  ```

  Of course, if you've defined a destructor (say, for closing files or whatever), you'll need to do this. Otherwise, omitting the `delete` just causes a memory leak, which is not immediately fatal, although you should eventually fix it.

  An example of using "delete" is in demo3.

- To use ASSERT, do the following somewhere at the top of your program.

  ```
  #include <assert.h>
  ```

4.2 Concepts and Exercises

4.2.1 Structure of C++ programs

includes, globals, classes, methods, main. Free functions versus methods.

- `demo1`. Copy it and use "make" to compile it. Notice the command line. Read the code.

- demo2/demo2. Copy the directory and use "make" to compile the main program. Read the code files.

4.2.2 Standard I/O

cout, cin, cerr and endl. Flushing streams.
formatted I/O
overloading output operator.

4.2.3 Strings

strcpy, strncpy, the use of man.

4.2.4 Object Representation

sizeof
    alignment
    pre-allocating space and using up array spaces successively

4.2.5 Page Sizes

We'll do random file I/O in chunks of 512 bytes.

4.2.6 Files

creating, opening, closing, randomly reading from them, deleting (unlinking) them

4.2.7 Linking and Makefiles

.cc (.C), .cpp, .h (.H), .o, .exe, .bin

4.2.8 Pointers, References

Exercise with one copy of an object and two pointers to it. Returning a pointer to an object from a function.
    difference between an object and a pointer to an object. Using new and delete.

4.2.9 Error Detection and Handling

error detection from unix system calls.

We're not going to implement the sophisticated queuing mechanism that minibase did, which allows errors to be traced back. Instead, just try to detect every error and return an appropriate error code to the caller. Ideally, we will propagate the error code up the call tree until it gets to the user, to then be displayed (possibly via a web page).

Put in print statements along the way, so that you will be able to debug where things are coming from.

Chapter 5

Semantic Comparison

5.1 Address Arithmetic

5.2 First Class Functions

Comparison of function pointers (and the fact that they aren't closures) with inner classes.

5.3 Pointers versus References

Arrows versus dots.

5.4 Stack Allocation

5.5 Calling Protocols

5.6 Function Overloading

5.7 Operator Overloading

5.8 The Preprocessor

Chapter 6

Practical Matters

6.1 Compiling C

Once your source code (`.c` file) is saved to disk, you'll want to be able to compile and run it. Here is the general procedure.

For simple programs that are all contained in one file, you need only run the compiler. The compiler will automatically run the linker, producing an executable program. For example, if I write the following classic program:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

and I save it as `hello.c`, I can compile it as follows:[1]

```
% gcc hello.c
% a.out
Hello World!
```

Unfortunately, the standard behavior of the C compiler is to put the output in a file called `a.out`. We almost never want this. In fact, the standard Unix practice is to put the executable program in a file that has no extension. To name an output file for the C compiler, we use the `-o` switch:

```
% gcc hello.c -o hello
% hello
Hello World!
```

Warning: The C compiler doesn't check that the input file is a real source file before it opens and destroys the output file. So, if you get these two backwards, you will destroy your program:

```
% gcc hello -o hello.c        # WRONG!
```

For that reason among others, I recommend compiling using `make`. To use `make`, you give the name of the thing you are trying to create (not the thing you already have) and `make` tries to figure out how to create it. It will tell you the steps it uses. Here's an example:

```
% make hello
gcc -g    hello.c   -o hello
% hello
Hello World!
```

[1] The percent sign is meant to stand for your unix prompt, so these are *commands*, not C or C++ code. Your actual prompt is probably different. Don't type the percent sign when trying these commands.

Here, `make` has figured out the correct `gcc` command. It has also added the `-g` switch to add debugging information, which is a good idea.

If compiling your program is more complicated, you (or someone) can create a `Makefile` that includes the additional rules. When you run `make`, it will read that file and use those rules to compile your program. This is a tremendous convenience, since you don't have to remember the complexity, you just have to write it down once.

6.2 Compiling C++

This short section covers the same ground as the preceding section of compiling C, except it's C++, so if you're reading this document straight through, you can skip it or skim it.

Once your source code (`.cc` or `.cpp` file) is saved to disk, you'll want to be able to compile and run it. Here is the general procedure.

For simple programs that are all contained in one file, you need only run the compiler. The compiler will automatically run the linker, producing an executable program. For example, if I write the following variation of a classic program:

```
#include <iostream>

int main() {
    cout << "Y'all come back now, y'hear?" << endl;
    return 0;
}
```

and I save it as `goodbye.cc`, I can compile it as follows:

```
% g++ goodbye.cc
% a.out
Y'all come back now, y'hear?
```

Unfortunately, the standard behavior of the C++ compiler is to put the output in a file called `a.out`. We almost never want this. In fact, the standard Unix practice is to put the executable program in a file that has no extension. To name an output file for the C++ compiler, we use the `-o` switch:

```
% g++ goodbye.cc -o goodbye
% goodbye
Y'all come back now, y'hear?
```

For that reason among others, I recommend compiling using `make`. To use `make`, you give the name of the thing you are trying to create (not the thing you already have) and `make` tries to figure out how to create it. It will tell you the steps it uses. Here's an example:

```
% make goodbye
g++     goodbye.cc   -o goodbye
% goodbye
Y'all come back now, y'hear?
```

If compiling your program is more complicated, you (or someone) can create a `Makefile` that includes the additional rules. When you run `make`, it will read that file and use those rules to compile your program. This is a tremendous convenience, since you don't have to remember the complexity, you just have to write it down once.

6.3 The GNU C Compiler

The GNU project has build a C compiler for almost any hardware, so the GNU C compiler (`gcc`), is nearly ubiquitous. So is its companion C++ compiler, `g++`. You should first know that gcc/g++ will execute the

linker for you, so you don't have to learn to use a separate program. There are, however, a few command-line switches that are important:

-o This means to put the output on the following filename. Warning: gcc/g++ don't check that you're not overwriting something important, so be careful. You'll follow this switch with the name of your object file (`something.o`) or the name of your final executable.

-c This means to compile only, don't link. As I mentioned, the compiler will invoke the linker, and it does so automatically, unless you stop it. The `-c` option is how you stop the compiler from linking.

-g This means to compile/link in some debugging information. Useful if you plan to use the GNU debugger (`gdb`).

-Wall This means to print all warning messages. Some warnings are not deemed important enough to mention unless you use this switch.

6.4 Running

Once you have an executable program, you can run it just by typing its name to the Unix prompt. There is no equivalent to the "java" command that comes first on the command line. Examples of this appear in the preceding subsections on compiling C and C++ programs.

The way that Unix does this is by looking through all the directories on your "path" until it finds one that has the same name as the name you typed. It then runs that file. This is the same procedure it uses for running programs like `ls` or `gcc` or `emacs`.

Sometimes, this procedure fails: Unix either doesn't find the program (it's not on your "path") or it finds something else that has the same name instead (try writing and running a program called "ls"). If this seems to be happening to you, consult your unix documentation or local guru.

6.5 Separate Compilation

Java is a modern language that drew on the best ideas of existing languages—including C++—but did not have to support older models of compilation and linking. Bjarne Stroustrup, the designer of C++, was building his language on the much older C language, and so it retains many of the older models of compilation and linking that dominated in the past.

(What this means is that what you'll be learning will seem weird, coming from a Java perspective, but historically speaking, it's Java that is new and different. Better, in many ways, but not at all "standard.")

Historically, few programming languages have syntactic elements to break up programs into separate files. In Scheme and Common Lisp, for example, a program is just a set of functions and variables. In either language, the elements can be put into separate files and separately compiled and loaded, but that's entirely up to the programmer; it's not required by the language.

Java, unlike these older languages, encourages a model where each class is stored in a different file, each file is separately compiled, and the compiled files are linked together at run-time by the Java Virtual Machine (JVM). Hopefully, this long section will give you a new perspective on what those terms mean. I would like to just tell you how to do Java-style compiling and linking in C++, but in order to explain what is going on, I need to give you more of the history. Please bear with me.

6.5.1 Older Models

Most languages do not run on a virtual machine like the JVM. In Java, the compiler (`javac`) translates the Java source code into Java byte code, which is the machine-code of the JVM. Instead, most compilers translate the source code into the machine-code of the particular hardware that the program is running on. That's why the program has to be re-compiled for different hardware.

Because there is no JVM, the result of the compilation must be a "complete" executable, saved in a file. To run this program, you give its name on the command line, just as you would `java`, `javac` or `emacs`. Keep

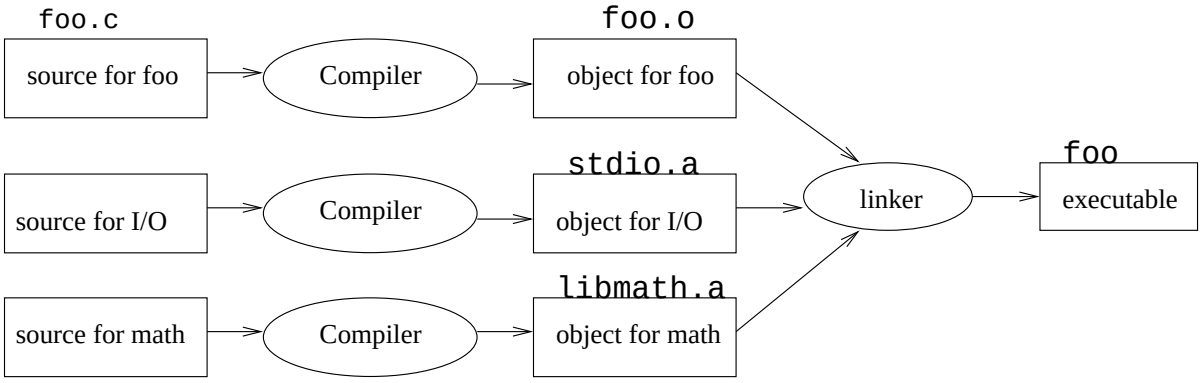

Figure 6.1: The structure of separate compilation and linking. The programmer writes the source code for a program that will be called "foo." Supposing the programming language is C, the source code would be called `foo.c` and corresponds to the box at the upper left. The programmer runs the compiler to compile `foo.c` and produce `foo.o`, which is the object code. Then the the programmer runs the linker to link `foo.o` with other object modules to produce a complete executable, called `foo`.

the word *executable* in mind: the word looks like an adjective, but most computer scientists use it as a noun for a complete binary program.

A traditional model of compilation is that the compiler reads the *entire* source code of the program and translates it into a complete executable. In practice, this almost never happened, because from the very beginning people realized it was silly to continually re-compile infrastructure such as the input/output routine, math routines (such as square root or sine and cosine), and things like that. Thus, the construction of an executable consisted of *compilation* and *linking*. Compilation is, of course, the translation of source code into "object" code[2]. The new object code is then *linked* with the object code of the infrastructure routines, which was previously saved in files. That means that the compilation of the infrastructure routines only has to be done once, and the object code is linked to the object code of whatever program we're creating.

Figure 6.1 illustrates the basic idea. There are three compilations shown, but the bottom two happen only once, when the I/O and math libraries are created for this machine. Subsequently, programmers can develop programs using those libraries by compiling their files and linking them to the libraries to produce a complete executable.

There are some common naming conventions that people use:

- source: source files usually end with some abbreviation of the source language, like `.java` or `.c`, `.pas` (for pascal), `.for` (fortran) and so on. C++ files are usually named either `.cc` or `.C`. The latter (uppercase C) doesn't work so well on case-insensitive file systems, because it becomes impossible to tell the difference between C and C++, so `.cc` is preferable.

- object: Object files in Java are the `.class` files, but remember that the "machine code" in them is for the Java virtual machine, not for the actual hardware. Object files on Unix usually end in `.o`, regardless of the source language. (Library files often end in `.a` instead of `.o`, but they're essentially the same.) (It's even possible to link together object files that came from different source languages!) On Windows, one kind of object file ends in `.DLL`; you'll sometimes hear about someone's Windows system crashing because it has a missing or corrupted DLL.

- executable: executable files on Unix usually have no extension, so that they don't look any different

[2]The standard term is "object," but this has *nothing* to do with objects in the sense of object-oriented programming. This is the word "object" in the sense of target or goal, as in "the object of the game is to throw the ball through the hoop." Object code is the same as machine code or binary code, but it has the implication of being incomplete, not executable, because it is in need of *linking*.

from commands like `ls` or `emacs`. In fact, it is a design goal of Unix that there be no distinction between user-developed programs and "system" programs, so, for example, `ls` is an ordinary program that someone wrote and compiled just like anything else. Executables on Windows usually end in `.EXE`.

FYI: in Unix, the C compiler is usually called either `cc` or `gcc` (if it's the GNU c compiler) and the C++ compiler from GNU is called `g++`. The Unix linker is called `ld`: you'll sometimes see that as the source of an error, so you know that the error happened during linking, not compiling. Here's an example caused by forgetting to link the math library:

```
% cc -g use-math.c -o use-math
/tmp/ccPRdjpp.o: In function 'main':
/home/anderson/writing/cpp/use-math.c:5: undefined reference to 'cos'
collect2: ld returned 1 exit status
make: *** [use-math] Error 1
%
```

The give-away is the `ldreturned1exitstatus` part.

Okay, that's enough of history. I think you can see where Java comes from: programs are in separate files that are separately compiled, but the linking step is done implicitly by the `java` program when you run your program. You never explicitly create an executable file in Java, since it's really `java` that is the executable.

6.6 Separate Compilation in C and C++

We have seen a program broken up into pieces that are compiled separately. We can informally call such pieces "modules." A good program design will have each module be a coherent, logically connected set of data structures and functions. In Java, modules are always classes.

This idea of separately compiled "modules" is great for several reasons: first, it saves time, because we don't have to continually re-compile the library functions; second, it allows incremental building of programs; and third, it sharpens the separation of a program into modules. We'd like to be able to do it with our own programs: divide them into separately compiled modules.

6.6.1 Header Files

Earlier, we said that one example of a function that we would put in a library would be a math function, like `sqrt` or `cos`. Our source code (in `foo.c`) will presumably have something like this in it:

```
float a = 1.5;
float pi = 3.1415926;
float z = cos(a*pi);
```

How does the compiler know that `cos` is a function that takes one argument, which is a float, and returns a float? After all, the definition of `cos` is not in this file—that's the whole point! Is the compiler just supposed to trust the programmer? Programmers make mistakes!

The solution is that instead of giving the compiler the complete *definition* of `cos`, we will just give it the *declaration*: just enough information to do the necessary compatibility testing, but not enough to slow down the compilation. The declaration looks like this:

```
float cos(float x);
```

Who writes this declaration? The author of the `cos` function. Where is it written? It's put in a "header" file, that contains other declarations like this of functions from the math library. Conventionally, header files end with `.h`.

Now that we have a header file filled with declarations, how can we get the compiler to read the header file when it compiles our program? We do that with a special directive (which actually controls something called the "pre-processor") like this:

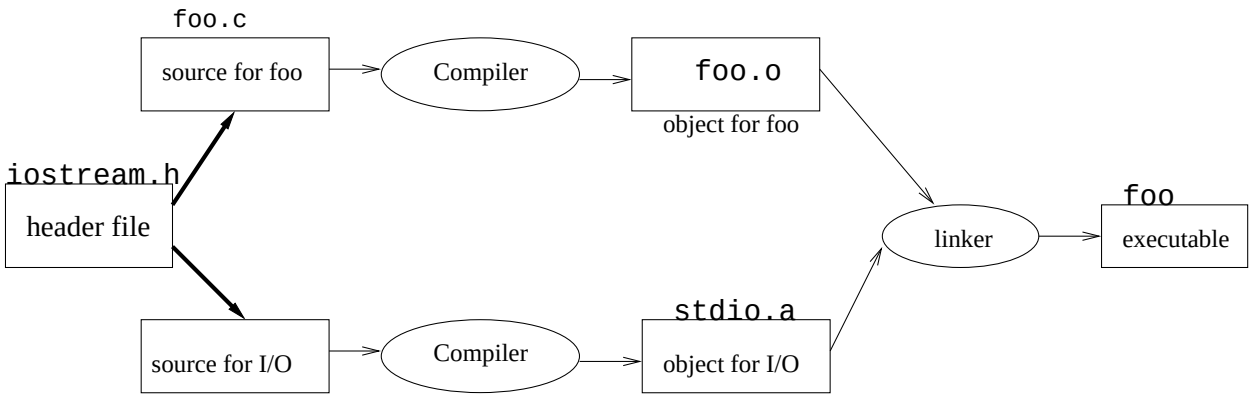

Figure 6.2: The header file is read by the compiler when compiling either the client code (`foo.c`) or the implementation. Thus, we ensure that both sides have the correct arguments and types for all functions and methods. The bold arrow indicates that the file at the tail is `#include`'d in the file at the head of the arrow.

```
#include <stdio.h>     // declarations for the I/O library
#include <math.h>      // declarations for the math library
```

The `#include` directive is very much like Java's `import` statement in effect, but it is purely textual: it says to go off and read this other file, as if it were right here. Thus, the pre-processor gives the compiler the illusion that everything (all the code) is in one file.

Note that the separation of declarations from definitions is not used in Java: the `import` statement allows the compiler to examine the `.class` to check that the methods are used correctly. In C++, that information is not in the object file, so it must be gotten from some source file.

6.6.2 Interfaces and Recompilation

C/C++ uses the technique of header files to allow separate compilation and type-checking across boundaries, so the header file essentially defines an interface between the *user* or *client* of some code (another programmer who is creating objects, calling functions and invoking methods) and that *implementor* of the code (who defines the classes, functions and methods). Computer scientists call this an Application Programmer Interface, or API. In the C/C++ way of doing things, a header file often defines an API. In Java, the set of public methods of a class define its API (but just the *signatures*[3] of the methods, not their implementation).

APIs are good things, because they allow freedom on both sides of the interface: the application programmers (the clients) are able to use the features of the API as they see fit, as long as they invoke things properly given the rules of the API. The implementors are able to implement the functionality any way they see fit, as long as they follow the rules of the API.

We have to re-draw our figure to reflect the existence of header files; the new version is in Figure 6.2. I simplified it by having only one extra module, but the same idea can be repeated: there would be a `math.h` that defines the interface (API) to the functions and constants in the math library.

What if we want to use this idea in modularizing our C++ program, with each class in a different file (as we do in Java)? If that's the case, we end up with a dependency structure like that in figure 6.3—notice that it's very similar to figure 6.2. The difference is only that we the application programmers create all the files, instead of getting `stdio.a` from a library.

I have created a simple example that exhibits all these features; it's a push-only stack. The idea is not that the code is particularly useful, merely that it is sufficient to see how C++ programs are usually structured. The code for these files is as follows:

demo2.cc This is the client code. It creates a stack and does some operations on it. The code looks like:

[3]The signature of a function or method is its name and argument list

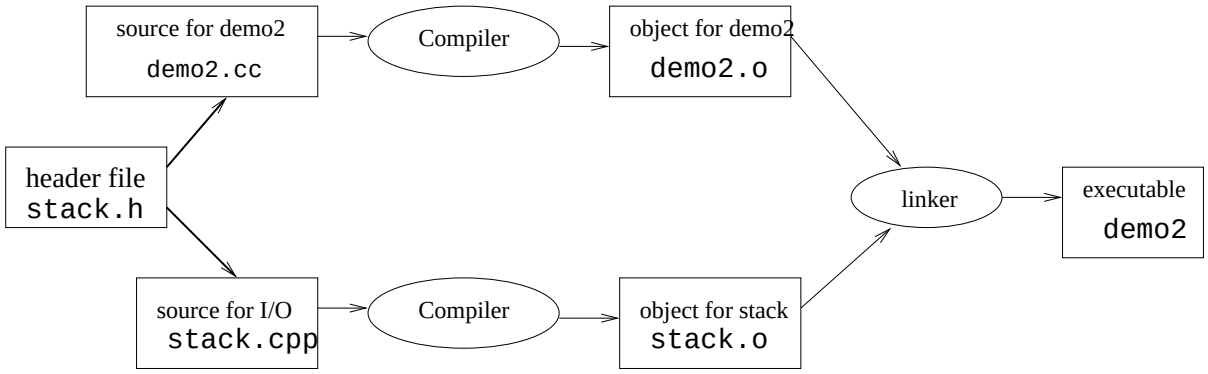

Figure 6.3: The header file for the class `Stack` is `stack.h` is read by the compiler when compiling either the client code (`demo2.cc`) or the implementation file for the stack class, `stack.cpp`.

```
#include <iostream.h>
#include "stack.h"

int main() {
    Stack* stk3 = new Stack; // a pointer to a stack
    Stack* stk4 = new Stack(100); // a pointer to another stack

    stk3->push(4);
    stk3->push(3);
    cout << "Pushed " << stk3->size() << endl;
    cout << "elements are:" << endl;
    stk3->print();

    delete stk3;
    delete stk4;
    return 0;
}
```

stack.h This is the header/API file. It defines the stack class and declares all the member functions (methods). Private data members and methods can also be defined and declared here. The example file looks like:

```
class Stack {
private:
    int* elts;
    int top;
    int cap;
public:
    Stack();

    Stack(int capacity);

    void push(int);

    int size();
```

```
    void print();
};
```

stack.cpp This file implements all the methods that were declared in `stack.h`

```
#include <iostream.h>
#include "stack.h"

const int DEFAULT_CAPACITY = 10;

// This is a no-argument constructor
Stack::Stack() {
    cap = DEFAULT_CAPACITY;
    elts = new int[DEFAULT_CAPACITY];
    top = 0;
}

// This is a unary (one-argument) constructor, specifying the capacity
Stack::Stack(int capacity) {
    cap = capacity;
    elts = new int[capacity];
    top = 0;
}

// This method pushes on the stack
void Stack::push(int new_elt) {
    if( top == cap ) return;
    elts[top] = new_elt;
    top++;
}

// This method returns the current size of the stack.
int Stack::size() {
    return top;
}

// This method prints out the elements of the stack
void Stack::print() {
    for( int i=0; i<top; i++ ) {
cout << elts[i] << " " ;
    }
}
```

You'll note a standard naming convention: class files end in `.cpp`, and the file that contains `main` ends in `.cc`. It's important to remember that C/C++ executables must contain exactly one `main` function, since that's where the program starts. In Java, each class can have a `main` method because you can specify which one runs by giving its name on the the Java virtual machine (the `java` program). Java's way is cool, but it's only possible because of the JVM.

Notice the difference in syntax between the declaration of a method (in `stack.h`) and its definition (in `stack.cpp`). In the declaration, we just give its return type, name, and argument list, ending with a semicolon. In the definition, we give the same ending with the code in braces. We also *must* qualify the method name with the class name using the double colon syntax, such as `Stack::push`.

Look at the first two lines of `demo2.cc` and note that there are two ways that `#include` is used. With angle brackets, like `<iostream.h>`, it means to look in the "standard places" so that is like your `CLASSPATH` in Java. With quotation marks, like `"stack.h"`, it means to look in the current directory.

We introduced all this complication so that we could not only clarify the API for our stack class, even putting it in a separate header file, but also to allow us to do separate compilation. Looking back at figure 6.3, we can see that if the client changes (we edit `demo2.cc`), we only need to recompile it and re-link to `stack.o`. If the implementation changes (we edit `stack.cpp`), we only need to recompile it and re-link to `demo2.o`. What if `stack.h` changes? If that's the case, *everything* needs to be recompiled. We've fundamentally changed the relationship between the client and the class, and so everything needs to be updated.

In professional software development, design teams first determine the interfaces between classes and these decisions are nailed down, often in the form of header files. Then, implementation teams can start working, *independently,* on the different sides of the interface. That's partly why I put `stack.h` leftmost—it happens first, and everything depends on it. This strategy is just our old friend "divide and conquer" reappearing in a new guise.

6.7 Testing

The Java idea of each class having a `main` method makes it easy to use the main method of "infrastructure" classes (like, say, stacks) as test functions: the main method calls the methods of the class to test various scenarios (can we pop from an empty stack, does a push followed by a pop yield the original element, and so forth). Unfortunately, we can't do that in C++, so we typically implement testing in a different file. For example, the `demo2.cc` is essentially a test file for the `Stack` class—it stands in for the real clients who have real needs for this kind of stack.

6.8 Including

One problem that can sometimes come up is when a header file gets included more than once. For example, lots of different modules may need to include the `iostream.h` header file. The compiler may worry that objects like `cout` (which is defined in the `iostream` module) are being defined more than once, which is illegal. As an example of what might occur when you're programming, the `Stack` class might be used by another class, say `Parser` and both classes might be used by the main program. In this situation, `parser.cpp` will have a `#include` of `stack.h` and so will the main program. This causes the compiler to complain that the `Stack` class is being redefined.

The solution is a trick of the pre-processor. (There may be other tricks out there, or smarter compilers, but this is a standard solution to a common problem.) The pre-processor has a conditional statement in it that will skip some lines of code if a "symbol" is defined or not defined. Don't think too much about what these "symbols" are—they are not a part of your program, they are just a part of the pre-processor. The pre-processor also has a directive to make a symbol defined. Thus, the following trick will ensure that the enclosed lines of code are processed only once:

```
#ifndef SYMBOL
#define SYMBOL
lines of
code
#endif
```

It works because the first line says to process the lines up to the `#endif` only if the `SYMBOL` is not defined. The second line defines the symbol. Therefore, it's common to enclose the contents of a header file with that idiom, to ensure that it can be `#include`'d as many times as necessary, without confusing the compiler.

This conditional compilation ability is also commonly used for debugging statements, since you can easily turn them on and off, often with just a command-line option to the compiler.

6.9 The GNU C Compiler

Given a dependency graph like figure 6.3, how do we compile the program? This depends, of course, on the compiler; I'll describe how to compile it using the nearly ubiquitous compilers from GNU: gcc and g++.

Let's first discuss how to compile a program that is *not* divided up into separate modules. Thus, it's complete in one file and the compiler and compile and link it in one step. This is the easiest case. If the file is `foo.cc`, we execute one of the following commands:

```
g++ foo.cc -o foo
g++ -g foo.cc -o foo
g++ -Wall -g foo.cc -o foo
```

The second version links in the debugger information, and the last adds the warnings. For simplicity, I'm going to omit these options from the following examples, but you can always add either or both.

To compile a program like `demo2`, we have to first compile each side, then link the two together. I'll number these steps:

```
g++ -c demo2.cc -o demo2.o    // 1
g++ -c stack.cpp -o stack.o   // 2
g++ demo2.o stack.o -o demo2  // 3 = link
```

Notice that the step 3, linking, has no source files; only object files. If we change the source code in `stack.cpp`, we only have to redo steps 2 and 3. If we change the source code in `demo2.cc`, we only have to redo steps 1 and 3.

6.10 Makefiles

As the dependency graphs that we draw for complex programs get bigger and bigger, it can become confusing to determine what needs to be recompiled when some changes are made. Our list of three commands, as in the last section, will grow to a list of a dozen or more. Keeping track of what commands need to be done is the problem solved by the Unix program called `make`. The `make` program requires you to write down the dependency graph and the commands necessary to create targets (object files, executable files, or whatever). You can then compile a program just by running `make`, which reads the dependency graph and does whatever is necessary.

6.10.1 Defining Makefiles

The dependency graph and the commands are stored in a file called `Makefile` or `makefile`. The structure of such a file is like:

```
target:  dependencies
       commands
```

In terms of our dependency graph, a target is something towards the right that is the output or result of some step. The commands are the steps that create that file. The dependencies are the files that the target depends on. The basic rule of a makefile is: if a target is older than any of its dependency files, the target is out of date and so the commands must be run in order to bring the target up to date.

In figure 6.4, we give the steps for creating the targets of our dependency graph. The makefile for this program is subsumed by the following makefile:

```
CC = g++ -g -Wall

demo2: stack.o demo2.o
$(CC) -o demo2 stack.o demo2.o

stack.o: stack.h stack.cpp
```

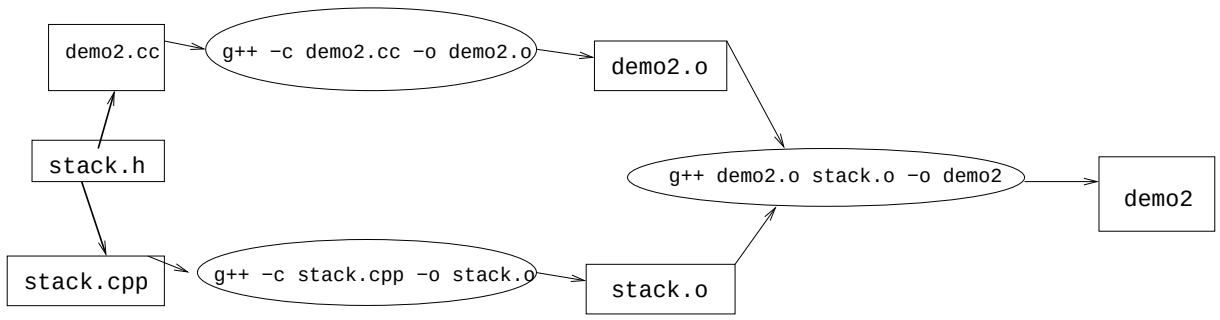

Figure 6.4: Steps for Separate Compilation

```
$(CC) -c -o stack.o stack.cpp

demo2.o: stack.h demo2.cc
$(CC) -c -o demo2.o demo2.cc

clean:
rm *.o demo2
```

Things to notice about that makefile: first, the `stack.h` file is mentioned in the dependency list for each of the `.o` files, since they must be recompiled if the `stack.h` file changes. The `make` program doesn't look inside your source files to see what other files they depend on, so you have to do that yourself.

Secondly, you can define "variable" to stand for programs, files, or whatever. Here, we defined the `CC` variable to mean the `g++` compiler with the `-g` and `-Wall` options.

6.10.2 Running "make"

Once you've written a Makefile, you can create a target by just saying

`make` *`target`*

The make program will recursively make all the dependencies of this *target* up to date, and then make the target up to date. If you don't specify a target, make will use the first one mentioned in the makefile.

In the example makefile above, you may have noticed a target called "clean." We can have targets like this one that don't depend on anything, so they're *always* out of date. That means the commands will always be executed. Thus, a makefile can store arbitrary sets of commands that can easily be executed by making a particular target.

If you ever download Unix software from the Internet, you'll find that it often comes with a Makefile, and the instructions for compiling and installing it are to do:

```
make
make install
```

Convenient, no?

You could even use "make" with java programs, so that the dependency for each `.class` file is the corresponding `.java` file, and the command is to run `javac` to re-compile.

6.10.3 Writing Makefiles

There is one awful, horrible secret to writing makefiles and that is that the command lines *must* begin with a "tab" character. Spaces will not do. I cannot imagine an adequate defense for this stupid rule, but if you ever intend to write a makefile, you have to know this.

6.11 Linking: Static vs. Dynamic

There is another difference between C/C++ linking and Java linking that deserves a brief mention. Java linking is dynamic: if a module or class that is imported isn't used, it won't be linked with the rest of the program or loaded into memory. For example:

```
if( prime(x) ) {
   Stack s = new Stack();
   ...
}
```

If this is the only use of the Stack object in this program, and the value of `x` never turns out to be prime, the Stack module is never linked in. This is cool, but it means that the run-time system has to have a linker in it, which can be big and slow.

C/C++ often insists on static linking, which means that when the program is built, we have to decide what modules to link with it. If they aren't used when the program executes, the work was wasted, but that wasn't at run-time, so we're willing to pay that price. It also means that the size of the executable is bigger than is necessary.

However, modern compilers for C/C++ (including the ones from GNU), can allow dynamically loaded modules, linking them in at run-time. On Unix machines, these files usually end in `.so` and are found in `/lib` directories. On Windows machines, these files are called `DLL`s, for dynamically loaded libraries.

6.12 External Variables

Another issue that comes up that I didn't mention earlier is when several modules want to make use of a global variable. They all need to see the declaration of the variable, in order to know its name and type, so the declaration will go into a header file. However, only one module can actually create (define) the variable, and the trouble is, variable declarations and definitions are combined into one syntax (unlike functions and methods).

The solution is to declare the variable as "external," which means that someone else will actually define it. Thus, the header file could have:

```
extern int global_counter;
```

Then in the main program (a good place to put anything that must be defined once), we can put:

```
int global_counter = 0;
```

The uses of extern are, fortunately, rare, but they do arise in real, complex programs.

Chapter 7

Standard Template Library

7.1 Templates

A template is a "generic" chunk of code that abstracts over one or more types. It's as if you took a normal piece of code and replaced all the types for some things (variables, functions) with a parameter. Let's look at an example. Here's a function that returns the larger of two arguments:

```
int bigger( int x, int y) { return x > y ? x : y ; }
```

However, this only works for integers. If we want to write one for floats and doubles, we have to duplicate the code:

```
int    bigger( int    x, int    y) { return x > y ? x : y ; }
float  bigger( float  x, float  y) { return x > y ? x : y ; }
double bigger( double x, double y) { return x > y ? x : y ; }
```

Coding this way is boring, painful and error prone. What we want is to introduce some abstraction:

```
template <class numType>
numType bigger( numType x, numType y) {
  return x > y ? x : y ;
  }
```

Now, whenever we use the function `bigger`, the C++ compiler will create a function of the appropriate types and include it in the compiled code.

Templates are still fairly new, and it's not unusual for a compiler to make a mistake with compiling code that uses templates. Fortunately, such problems are becoming rare.

7.2 Introducing the Standard Template Library

The Standard Template Library (STL) is a set of templates to implement some standard data structures, such as stacks, queues, hashtables and the like. It is delivered with most modern C++ implementations.

write this section

Appendix A

TO DO

- Make the arrows in figure 6.2 bolder.
- Organize the C sections better.
- write section on unions
- write section on file I/O in C. talk about stdout, stderr, stdin
- Write up C++ syntax
- Make existing C++ sections more generic
- Write up templates
- fix TOC formatting
- Make use of John Lewis's comparison
- talk about overloading toString in Java and C++
- talk about gdb
- distinguish better between advanced stuff and beginner stuff