A Survival Guide to COS 126/128 at Princeton

Greg Humphreys '97
Head Lab TA, Fall'95—Spring'96

This is guide is meant to get you started on using the C programming language in your assignments. The sketchy introduction to C given here should suffice for the first two programming assignments. After that, use the the course reader and the text. Your class instructors and lab TAs are also excellent sources for more information on C.

The chapters in this guide are organized by *weeks*. The one titled *Week 0* discusses the basics of using the computers to program: typing your program using a text editor (here **emacs**), compiling it and running it. The chapter titled *Week 1* discusses the basics of C: how to define and use **variables**, and write simple programs using **if-else**, **for**, and **while**. The section titled *Week 2* discusses **functions** and **arrays**.

Week 0

Let's see what a C program looks like and how you run it on a computer.

0.1 The simplest C program

When learning any programming language, the first thing everyone does (simply out of convention), is to write a program that interacts with the user. Let's write such a program. The code to do this is shown in figure 0.1. Let's take a look

```
1: #include <stdio.h>
2:
3: void main(void)
4: {
5:      int i;
6:      printf ("Hello, world\n");
7:      printf ("Type a number:\n");
8:      scanf ("%d",&i);
9:      printf ("You typed: %d",i);
10: }
```

Figure 0.1: Your first C program

at those 5 very important lines of C code. Line 1 is an **include statement**. It simply takes the contents of the file **stdio.h** and inserts them into your C file. It's not important to understand what's in stdio.h right now.

Line 3 (or some variant) must exist in every C program. It is called the **main** function. This is where execution will begin when your program is run.

The "curly braces" on lines 4 and 10 define a **block** of code. This particular block is the **body** of the main function. When you run a C program, only the body of the main function is executed. This body may invoke code outside of the main body, but for now, it is all that is run.

Lines 5 and 6 are the actual statements that produce simple output. **printf** is the C command meaning, "Print this to the screen". Printf is possibly the most commonly used function in C, and is very powerful. It is discussed more in depth in section 1.1.2.

Line 8 may look strange — it's the way you read input from the user. Since you just printed out a message asking the user to type a number, you now have to prepare the computer to receive it. The %d tells `scanf` that you're expecting an integer, and the `&i` tells the computer to put the result into the variable named `i`. If none of this means anything to you, don't worry, just remember that the ampersand is necessary, and all will become clear later.

Line 9 repeats the integer you just typed with a message. Again, the %d means that you want an integer there, and the `i` tells the computer that you want to print the value stored in `i` (remember, we just read it into `i`).

But this still doesn't explain how you actually get your computer to execute this program.

0.1.1 Entering and running your program

In order to make the computer understand your C program, you must first enter it into the computer's memory. To do this, you will use any text editor you want. The standard one used by Princeton CS students is called **emacs**. This editor has several special features for programmers, one of which is that it automatically formats your C program in a pleasing and informative way (i.e., with indentations that emphasize the program's logical structure). Creating and editing files in **emacs** is easy using the mouse and pull-down menus; the lab TAs can also help you. Here I will point out several mistakes I see people making constantly:

1. Don't run more than one emacs at a time. Emacs has the ability to edit more than one file at a time. In fact, running more than one emacs is a surefire way to use up all of your computer's memory and slow your system down to an unbearable speed. You should run emacs once when you log in, and quit emacs when you log out. Ask your TA about editing multiple files.

2. Don't forget the ampersand! When you start up a program that is designed for the X windows environment, it is possible to get your prompt back immediately by typing an ampersand (&) after the command. Therefore, to run emacs, you should type: `emacs &`. If you forget the ampersand, the window in which you ran emacs will appear to "hang", that is, you won't get a prompt back. This isn't a problem; just go to that window, hold down the Control key, and type a 'z'. This will "suspend" emacs and give you a prompt back. Now, to start up emacs again and keep your prompt, you need to type "bg" at the prompt. **bg** tells the computer to run the suspended task in the background. Note that this method of regaining your prompt works for any X application, not just emacs.

3. Be careful with versions. Sometimes, people will get into a state where they are editing an older version of their program without knowing it. This is usually apparent if, instead of your filename (say, hello.c) at the bottom of the window, you see something like hello.c<2>. This usually happens when you are using a debugger from within emacs (if this doesn't mean anything to you, that means you won't encounter this problem, so don't worry about it). Make sure you are editing what you think you are.

Once your file is written and saved, you must change it from a human readable language (C) to something the computer can understand (machine code). To do this, you use a program called a **compiler**. The standard compiler at Princeton is **lcc**, written by Prof. David Hanson at Princeton. To compile your program, simply type: **lcc hello.c** (if hello.c is the name of your C file. Otherwise, replace hello.c with the name of your C file).

If you made any syntactic mistakes when entering your program, lcc will tell you when it is compiling. You MUST fix all compiler errors before you can run the program. Unlike english, the computer can't figure out what you mean if you make a tiny grammatical mistake (like forgetting the semicolon at the end of every statement).

Being able to understand what the compiler tells you is a very important skill in getting your program to compile. Many of the questions asked me in the lab are because students don't understand the error messages the compiler is issuing simply because they don't read them. Don't panic if lcc spits out a screenful of errors, just start at the topmost one, figure out what it's talking about, and fix it. It's usually just some little thing. Let's say you forgot the semicolon after the printf statement in the simple program in section 0.1. lcc would produce the error shown in figure 0.2.

```
hello.c:6: syntax error; found `}´ expecting `;´
```

Figure 0.2: The most common compiler error

Now, while this error may not mean anything at first glance, remember that the compiler reads programs just as you do, top to bottom, left to right. So let's look at this error. First, it says it's a syntax error. Those are the easiest kind to fix, it just means that your grammar was a little off. Next, it says that it FOUND a right curly brace, but it was expecting a semicolon. This is very useful information. As the compiler was scanning the program, it got to the end of the printf statement, and it thought it was going to be ended with a semicolon like it should. Instead, it saw a right curly brace and got confused. This error message is pretty self explanatory. I'll look at more C error messages as we get into more complicated stuff.

Once your C program compiles successfully, lcc produces an "executable" version of your program. This is what you run to make the computer do what you told it to. By default, this executable is named **a.out**. This is for historical reasons way too convoluted to get into here. In any case, simply type a.out in a shell window, and your program will run.

At this time I would like to point out two things. First, the compiler can issue warnings as well as errors. These are a little more tricky. If lcc issues a warning, it means that it saw something that it thinks is suspicious, but maybe it's what you meant. For example, if you had forgotten to include the file stdio.h, lcc would issue a warning telling you it hasn't heard of the function printf. The file would still compile and run just as expected, though. In general, it is a good idea to identify the source of all warnings and eliminate them.

Second, just because a program compiles without warnings or errors doesn't mean it's going to work. In fact, having a program work the first time is rare (I think it has happened to me only three or four times in my life). There are really two classifications of errors, or "bugs", in a program. The first kind are called "compile-time" bugs. These are errors that are issued by the compiler, and they are easy to find, because the compiler points you at the actual line in your source code where they occur. The other kind are "run-time" bugs. Most of the time, run-time bugs will not be as easy to find. Finding bugs in programs – debugging – is a painful process, and your TAs will help you with it.