Chapter 1: Your Basic C Program

In This Chapter

- **Finding out some C language history**
- **Building a C program one piece at a time**
- **Understanding the C program skeleton**

Learning a programming language is like trying to eat an entire banquet in one bite: You have so much to swallow at once, even to understand the most basic stuff, that it isn't a question of where to start, but rather what not to eat so that you don't get too sick too quickly.

This chapter provides a quick and dirty introduction to a single C language program. The chapter doesn't even explain why things are necessary because, honestly, at this point in the game you're probably more interested in accomplishing something than truly figuring out why something works. Don't worry: That comes later. For now, this chapter offers a small taste of the feast to come.

Remember to use the `basic` folder or directory for the source code and program files in this book.

The Section Where the Author Cannot Resist Describing the History of C

In the beginning was Charles Babbage and his Analytical Engine, a machine he built in 1822 that could be *programmed* to carry out different computations. Move forward more than 100 years, where the U.S. government in 1942 used concepts from Babbage's engine to create the ENIAC, the first modern computer.

To program Babbage's computer, you had to literally replace stacks of gears. To make the ENIAC carry out different tasks, it had to be rewired by hand.

By the early 1950s, computer programming had evolved from rewiring to entering instructions using rows of switches. The process began with Professor John von Neumann, who in 1945 developed the concept of the *function* (or *subroutine*), the *IF-THEN* choice statement, and the repeating *FOR loop*. By 1949, Von Neumann had come up with a binary programming

language he called Short Code. Then, in 1951, Grace Hopper developed the first *compiler*, which allowed the computer to be programmed using words and symbols rather than binary ones and zeroes. Computers could then be programmed with written instructions rather than by rewiring or throwing switches.

In the mid-1950s, the first major programming language appeared. Named FORTRAN, for *for*mula *tran*slating system, it incorporated *variables* and introduced logical conditions to the art of programming. Then, in 1959, the COBOL programming language was developed as businesses began to adopt computers. COBOL was the first programming language with a truly English-like grammar.

The Algol language was developed at about the same time as COBOL. From Algol, many programming techniques were introduced that are still used today.

In 1968, Zurich professor Niklaus Wirth used Algol as a basis for the Pascal programming language. Pascal was designed as a teaching tool because it didn't allow for sloppy programming and forced users to follow the rules of structured programming.

Meanwhile, over at the AT&T Bell Labs, in 1972 Dennis Ritchie was working with two languages: B (for Bell) and BCPL (Basic Combined Programming Language). Inspired by Pascal, Mr. Ritchie developed the C programming language.

The C language was used to write the Unix operating system. Ever since that first version of Unix in the early 1970s, a C compiler has always been a part of the operating system — even with Unix variations like Linux and Mac OS. It also explains why Unix comes with so many programming utilities. (Indeed, Unix is often called the "programmer's operating system.")

In 1983, C programmer Bjarne Stoustroup developed object oriented programming (OOP) extensions to the C language and created the C++ programming language. Even though it's often taught as a separate subject, C++ is really about 95 percent original C. Even so, some purists stick with the original C language and don't bother to discover that extra 5 percent of C++ — despite the power and flexibility it offers.

Branching off from C in the 1990s was another programming language: Java, from Sun Microsystems. Originally developed for interactive TV, the Java language truly found a home on the Web in 1994 and has been a popular Web programming language ever since.

Time to Program!

The C language has a certain structure to it — a form, a visual look, a feeling. Unlike in more freeform languages, you must obey traditions and rules to put together the most basic of C programs. That's what gives the C language its look and feel.

The following sections introduce you to the basic structure of a simple C program — the skeleton. Each section builds on the next, so read them in order.

Ensure that you have read through Appendix A, which discusses how to set up the C language compiler on your computer and the basic steps for editing, compiling, and running a program.

One suggestion: Please save all the source code files for this book in the `basic` folder on your hard drive (`prog/c/basic`).

The basic, simplest C program

When you program a computer, you tell it exactly what to do. The instructions are given in a particular language — in this case, the C language. Those instructions are then compiled into object code. The object code is then linked with a C language library, and the result is an executable file or a program you can run on your computer. It's just like magic!

I save the boring specifics for the next chapter. For now, consider the most basic of all programs:

Yes, that's a blank line. Open your editor and type a blank line. Just press the Enter key: *Thunk*! That's it.

Save the file to disk with the name DUMB.C. Remember to add to the end of the filename that .C part, which is what tells most modern compilers that you have created a C language file and not a C++ file.

Compile the source code for DUMB.C. Here's the command for most C compilers:

```
gcc dumb.c -o dumb
```

No, it doesn't compile. You get some sort of error message. That's for two reasons.

The first obvious reason is that your source code file is *blank!* It contains no instructions for the computer! With some programming languages, that would be acceptable, and the resulting program would also do nothing. But C is more structured than that.

The second reason is that the program needs a place to start. That place in all C language programs is the `main()` function.

The `main()` function

All C language programs must have a `main()` function. It's the core of every program. It's required.

A function is like a machine that does something. In C, built-in functions do things like compute the sine of an angle, display text on the screen, or return values from the computer's internal clock. You can also create your own functions that do wondrous things.

The `main()` function doesn't really have to do anything, other than be present inside your C source code. Eventually, it contains instructions that tell the computer to carry out whatever task your program is designed to do. But it's not officially *required* to do anything.

When the operating system runs a program, it passes control of the computer over to that program. This is like the captain of a huge ocean liner handing you the wheel. Aside from any fears that may induce, the key point is that the operating system needs to know *where* inside your program the control needs to be passed. In the case of a C language program, again, it's the `main()` function that the operating system is looking for.

At a minimum, the `main()` function looks like this:

```
main() {}
```

First comes the function's name, `main`. Then comes a set of parentheses. Finally comes a set of braces, also called curly braces.

Use your text editor to reedit the DUMB.C source code. In that file, on the first line, copy the `main()` function I just showed you. Copy it exactly as shown. Press the Enter key to end that line. (Many compilers require a blank line after the last bit of course code.)

Compile the program again:

```
gcc dumb.c -o dumb
```

This time, it compiles.

Run the program. Type **dumb** or **./dumb** at the prompt to run the resulting program.

Nothing happens.

That's right. And that's perfect because the program doesn't tell the computer to do anything. Even so, the operating system found the `main()` function and was able to pass control to that function — which did nothing and then immediately returned control right back to the operating system. It's a perfect, flawless program.

Inside the `main()` function

All C language functions have the same decorations. First comes the function name, `main`, and then comes a set of parentheses, `()`, and, finally, a set of braces, `{}`.

The name is used to refer to, or *call,* the function. In this case, `main()` is called by the operating system when the program runs.

The set of parentheses is used to contain any *arguments* for the function — stuff for the function to digest. For example, in the `sqrt()` function, the parentheses hug a value; the function then discovers the square root of that value.

The `main()` function uses its parentheses to contain any information typed after the program name at the command prompt. This topic is covered in a later chapter. For now, you can leave the parentheses blank.

The braces are used for organization. They contain programming instructions that belong to the function. Those programming instructions are how the function carries out its task or "does its thing."

By not specifying any contents, as was done for the `main()` function in the DUMB.C source code, you have created what the C Lords call a *dummy function* — which is kind of appropriate, given this book's title.

"Am I done?"

Note that the basic, simple dummy function `main()` doesn't require a specific keyword or procedure for ending the program. In some programming languages, an END or EXIT command is required, but not in C.

In the C language, the program ends when it encounters the last brace in the `main()` function. That's the sign that the program is done, after which control returns to the operating system.

Now you have various ways to end a program before the last brace is encountered. You can use the `return` keyword in addition to the `abort()` and `exit()` functions. You find out about the functions later; the `return` keyword is a key part of the `main()` function, as described in the next section.

Declaring `main()` as an `int`

When a program starts, it can optionally accept information from the operating system. Likewise, when the program is done, it can return information to the operating system. These return codes, called *errorlevels* in DOS and Windows and *exit codes* in Unix, can then be evaluated by the operating system or some shell script or batch file language.

To accommodate that evaluation, and to further define `main()` as a function, you need to declare the function as a specific type. The type is based on the kind of information the function produces. There are five basic variable types, and please don't memorize this list:

- `void`: A function that doesn't return anything
- `int`: A function that returns a whole number or *integer* value
- `char`: A function that returns text
- `float`: A function that returns a non-whole number or value with a fractional part
- `double`: The same as float, but more precise

For example, a function that reads a character typed at the keyboard is a `char` function; it returns a character. The function to calculate the arctangent of an angle would return a `float` or `double` value — some obnoxious number with a decimal part.

In the case of the `main()` function, what it can return to the operating system is a value, usually in the range of zero on up to some huge number (depending on the operating system). That makes `main()` qualify as an integer function, or `int`. Here's the proper way to define the `main()` function:

```
int main() {}
```

This line means that the `main()` function returns an integer (a whole number) value when it's done. How? By using the `return` keyword. `return` is responsible for sending a value back from a function. Here's how it looks:

```
return(0);
```

`return` is a C language keyword. It's followed by an optional set of parentheses and then the value to `return`, which is shown as `0` in the preceding line. Because this is a C language statement, it ends in a semicolon, which serves the same purpose as a period does in English.

Because the `return(0);` statement belongs to the `main()` function, it must be placed in the braces; thus:

```
int main() {return(0);}
```

Now you have a function that's properly defined to return an integer value and does in fact return the value `0` to the operating system.

Summon your editor again and bring up the source code for DUMB.C. Edit the first line so that it looks like the one I just showed you.

Save the change to disk, and then compile. Run.

The output is no different; the program runs and basically does nothing other than cough up the value `0` for the operating system to evaluate. (Because you're not evaluating the program, you have no way to tell, but the `0` is still produced.)

A little sprucing up

The C language compiler doesn't really care how pretty your source code is. There is value, of course, in presenting your source code in a readable format. This subject is directly addressed in another chapter. For now, return to your editor and add some space by changing your source code to look more like this:

```
int main()
{
    return(0);
}
```

The `int main()` function declaration appears on a line by itself, sort of "announcing" the function to the world.

The braces are kept on a line by themselves, as are the statements belonging to the function; however, the statements inside the braces are indented one tab stop. That keeps the statements visually together and helps align the braces so that you know that they are a pair.

By putting the `return(0);` statement on a line by itself, you can confirm that it ends in a semicolon.

Save back to disk the changes you have made to your source code. You can recompile the program if you like, but nothing has really changed as far as the compiler is concerned.

Finally, making `main()` do something

To breathe some life into the DUMB program, you need to employ the C language to direct the computer to *do* something. That's what programming is all about. Although the program already returns a value to the operating system, that just isn't enough.

The things that a program can do are limitless. The problem with that vast scope is that it takes a lot of "talking" in a program language to make the computer do impressive things. On the other hand, a simple thing you can direct the computer to do is display text on the screen. To do that, you can use the simple `puts()` function.

I suppose that *puts* stands for put string, where a string is a bit of text you put to the screen. Something like that. Here's how it works:

```
puts("Greetings, human!");
```

The text to display — the string — is enclosed in the function's parentheses. Furthermore, it's enclosed in double quotes, which is how you officially make text inside the C language, and how the compiler tells the difference between text and programming statements. Finally, the statement ends in a semicolon.

Here's how `puts()` fits into the DUMB.C source code:

```
int main()
{
    puts("Greetings, human!");
    return(0);
}
```

The `puts()` function works inside the `main()` function. It's run first, displaying the text `Greetings, human!` on the screen. Then the `return(0);` statement is run next, which quits the program and returns control to the operating system.

Save the changes to DUMB.C. Compile the program and then run the result. Unlike the previous renditions of DUMB.C, this one displays output on the screen:

```
Greetings, human!
```

One more thing!

As with the `main` function, you should also declare the `puts()` function, formally introducing it to the compiler. In fact, not doing this in some compilers generates an error.

Fortunately, the job of formal introductions has already been done for you. That information is contained inside a *header file* that was installed along with your compiler.

You find out more about header files in Book I, Chapter 2, but for now, to complete the task of formal introduction to the `puts()` function, add the following line at the top of your DUMB.C source code:

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

`#include` isn't a word in the C programming language. Instead, it's an instruction for the compiler to *include* another file on disk, inserting the text from that file into your source code at that specific point in the program. In this case, the text is grabbed from the file named STDIO.H, which is the standard input/output header file for the C language. It's also where the `puts()` function is formally introduced.

With the inclusion of that line, the final complete source code for DUMB.C should now look like this:

```
#include <stdio.h>

int main()
{
    puts("Greetings, human!");
    return(0);
}
```

Spruce up the source code in your compiler so that it resembles what you see here. Save it to disk. Compile and run:

```
Greetings, human!
```

Just about any time you use a function in C, you have to specify a header file to be included with your source code. It needs to be done only once; so, if you're using two functions that are both defined in the STDIO.H header file, you need only one `#include <stdio.h>`. But other functions use other header files, such as MATH.H, STDLIB.H, and numerous others. Again, it's all covered in Book I, Chapter 2.

The C Skeleton

Most C language source code listings start with a basic skeleton that looks like this:

```
#include <something.h>

int main()
{
    statement;
    statement;
    return(0);
}
```

Obviously, the skeleton will have more "meat" on it eventually — including various other braces and interesting whatnot. That's what comes with later chapters of the book, but at its most basic, C language source code looks like what you see here.