

- The guy who created the C programming language at Bell Labs is Dennis Ritchie. I mention him in case you're ever walking on the street and you happen to bump into Mr. Ritchie. In that case, you can say "Hey, aren't you Dennis Ritchie, the guy who invented C?" And he'll say "Why — why, yes I am." And you can say "Cool."

The C Development Cycle

Here is how you create a C program in seven steps — in what's known as the *development cycle:*

1. Come up with an idea for a program.
2. Use an editor to write the source code.
3. Compile the source code and link the program by using the C compiler.
4. Weep bitterly over errors (optional).
5. Run the program and test it.
6. Pull out hair over bugs (optional).
7. Start over (required).

No need to memorize this list. It's like the instructions on a shampoo bottle, though you don't have to be naked and wet to program a computer. Eventually, just like shampooing, you start following these steps without thinking about it. No need to memorize anything.

- The C development cycle is not an exercise device. In fact, programming does more to make your butt fit more snugly into your chair than anything.
- Step 1 is the hardest. The rest fall naturally into place.
- Step 3 consists of two steps: compiling and linking. For most of this book, however, they are done together, in one step. Only later — if you're still interested — do I go into the specific differences of a compiler and a linker.

From Text File to Program

When you create a program, you become a programmer. Your friends or relatives may refer to you as a "computer wizard" or "guru," but trust me when I say that *programmer* is a far better title.

TIP

Extra help in typing the GOODBYE.C source code

The first line looks like this:

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

Type a pound sign (press Shift+#) and then **include** and a space. Type a left angle bracket (it's above the comma key) and then **stdio**, a period, **h**, and a right angle bracket. Everything must be in lowercase — no capitals! Press Enter to end this line and start the second line.

Press the Enter key alone on the second line to make it blank. Blank lines are common in programming code; they add space that separates pieces of the code and makes it more readable. And, trust me, anything that makes programming code more readable is okay by me!

Type the word **int**, a space, **main**, and then two parentheses hugging nothing:

```
int main()
```

There is no space between `main` and the parentheses and no space inside the parentheses. Press Enter to start the fourth line.

Type a left curly brace:

```
{
```

This character is on a line by itself, right at the start of the line. Press Enter to start the fifth line.

```
printf("Goodbye, cruel
    world!\n");
```

If your editor was smart enough to automatically indent this line, great. If not, press the Tab key to indent. Then type **printf**, the word *print* with a little *f* at the end. (It's pronounced "print-eff.") Type a left parenthesis. Type a double quote. Type **Goodbye, cruel world**, followed by an exclamation point. Then type a backslash, a little **n**, double quotes, a right parenthesis, and, finally, a semicolon. Press Enter to start the sixth line.

```
return(0);
```

If the editor doesn't automatically indent the sixth line, press the Tab key to start the line with an indent. Then type **return**, a paren, **0** (zero), a paren, and a semicolon. Press Enter.

On the seventh line, type the right curly brace:

```
}
```

Some editors automatically unindent this brace for you. If not, use your editor to back up the brace so that it's in the first column. Press the Enter key to end this line.

Leave the eighth line blank.

The compiler and the linker

After the source code is created and saved to disk, it must be translated into a language the computer can understand. This job is tackled by the compiler.

The *compiler* is a special program that reads the instructions stored in the source code file, examines each instruction, and then translates the information into the machine code understood only by the computer's microprocessor.

- The linker's job is to pull together different pieces of a program. If it spots something it doesn't recognize, such as `retrun`, it assumes, "Hey, maybe it's something from another part of the program." So the error slides by. But, when the linker tries to look for the unrecognized word, it hoists its error flags high in the full breeze.

TECHNICAL STUFF

All about errors!

A common programming axiom is that you don't write computer programs as much as you remove errors from them. Errors are everywhere, and removing them is why it can take years to write good software.

Compiler errors: The most common error, initially discovered by the compiler as it tries to churn the text you write into instructions the computer can understand. These errors are the friendly ones, generally self-explanatory with line numbers and all the trimmings. The errors are caught before the program is built.

Linker errors: Primarily involve misspelled commands. In advanced C programming, when you're working with several source files, or modules, to create a larger program, linker errors may involve missing modules. Also, if your linker requires some "library" file and it can't be found, another type of error message is displayed. Pieces of the program are built, but errors prevent it from them being glued together.

Run-time errors: Generated by the program when it runs. They aren't bugs; instead, they're things that look totally acceptable to the compiler and linker but just don't do quite what you intended. (This happens often in C.) The most common run-time error is a *null pointer assignment*. You aggravate over this one later. The program is built, but usually gets shut down by the operating system when it's run.

Bugs: The final type of error you encounter. The compiler diligently creates the program you wrote, but whether that program does what you intended is up to the test. If it doesn't, you must work on the source code some more. Bugs include everything from things that work slowly to ones that work unintentionally or not at all. These are the hardest things to figure out and are usually your highest source of frustration. The program is built and runs, but it doesn't behave the way you think it would.

TECHNICAL STUFF

Even more keyword madness!

Keywords are worth noting because their use is restricted or reserved. For example, you cannot think up your own function and name it `short`. That's because `short` is a keyword, reserved only for its specific purpose in the core C language. That's one way the keywords are special.

In addition to the 32 keywords shown in Table 3-1 are these two depreciated C language keywords:

fortran

entry

C once had these keywords, but no longer. Still, I would avoid using them in your programs. (That's what "depreciated" means.)

Also, the C++ language has a hoard of reserved words. If you plan to study C++, include these words in your do-not-use, reserved C language vocabulary:

asm	false	private	throw
bool	friend	protected	true
catch	inline	public	try
class	mutable	reinterpret_cast	typeid
const_cast	namespace	static_cast	using
delete	new	template	virtual
dynamic_cast	operator	this	

It's better to know these words now and not use them than to use one (such as `new` or `friend`) and run into trouble later when you eventually find out how to use C++.

In addition to grammar, languages require rules, exceptions, jots and tittles, and all sorts of fun and havoc. Programming languages are similar to spoken language in that they have various parts and lots of rules.

- The keywords can also be referred to as *reserved words*.
- Note that all keywords are lowercase. This sentence is always true for C: Keywords, as well as the names of functions, are lowercase. C is case sensitive, so there is a difference between `return`, `Return`, and `RETURN`.
- You are never required to memorize the 32 keywords.
- In fact, of the 32 keywords, you may end up using only half on a regular basis.
- Some keywords are real words! Others are abbreviations or combinations of two or more words. Still others are cryptograms of the programmers' girlfriends' names.
- Each of the keywords has its own set of problems. You don't just use the keyword `else`, for example; you must use it *in context*.

Run the program! The output looks something like this:

```
What is your name?dan
What is your favorite color?brown
dan's favorite color is brown
```

In Windows XP, you have to run the command by using the following line:

```
.\color
```

The reason is that COLOR is a valid console command in Windows XP, used to change the foreground and background color of the console window.

Experimentation time!

Which is more important: the order of the `%s` doodads or the order of the variables — the arguments — in a `printf` statement? Give up? I'm not going to tell you the answer. You have to figure it out for yourself.

Make the following modification to Line 12 in the COLOR.C program:

```
printf("%s's favorite color is %s\n",color,name);
```

The order of the variables here is reversed: `color` comes first and then `name`. Save this change to disk and recompile. The program still runs, but the output is different because you changed the variable order. You may see something like this:

```
brown's favorite color is Dan.
```

See? Computers *are* stupid! The point here is that you must remember the order of the variables when you have more than one listed in a `printf()` function. The `%s` thingies? They're just fill-in-the-blanks.

How about making this change:

```
printf("%s's favorite color is %s\n",name,name);
```

This modification uses the `name` variable twice — perfectly allowable. All `printf()` needs are two string variables to match the two `%s` signs in its formatting string. Save this change and recompile. Run the program and examine the output:

```
Dan's favorite color is Dan
```

Why are comments necessary?

Comments aren't necessary for the C compiler. It ignores them. Instead, comments are for you, the programmer. They offer bits of advice, suggestions for what you're trying to do, or hints on how the program works. You can put anything in the comments, though the more useful the information, the better it helps you later on.

Most C programs begin with a few lines of comments. All my C programs start with information such as the following:

```
/* COOKIES.C
Dan Gookin, 1/20/05 @ 2:45 a.m.
Scan Internet cookie files for expired
dates and delete.
*/
```

These lines tell me what the program is about and when I started working on it.

In the source code itself, you can use comments as notes to yourself, such as

```
/* Find out why this doesn't work */
```

or this:

```
save=itemv;    /* Save old value here */
```

or even reminders to yourself in the future:

```
/*
Someday you will write the code here that makes
the computer remember what it did last time this
program ran.
*/
```

The point is that comments are notes *for yourself.* If you were studying C programming in school, you would write the comments to satiate the fixations of your professor. If you work on a large programming project, the comments placate your team leader. For programs you write, the comments are for you.

Comment Styles of the Nerdy and Not-Quite-Yet-Nerdy

The MADLIB1.C program contains five comments and uses three different commenting styles. Though you can comment your programs in many more ways, these are the most common:

Like `scanf()` reading in text, `gets()` requires a `char` variable to store what's entered. It reads everything typed at the keyboard until the Enter key is pressed. Here's the format:

```
gets(var);
```

`gets()`, like all functions, is followed by a set of parentheses. Because `gets()` is a complete statement, it always ends in a semicolon. Inside the parentheses is *var,* the name of the string variable text in which it is stored.

Another completely rude program example

The following is the INSULT1.C program. This program is almost identical to the WHORU.C program, introduced in Chapter 4, except that `gets()` is used rather than `scanf()`.

```
#include <stdio.h>

int main()
{
    char jerk[20];

    printf("Name some jerk you know:");
    gets(jerk);
    printf("Yeah, I think %s is a jerk, too.\n",jerk);
    return(0);
}
```


Enter this source code into your editor. Save the file to disk and name it INSULT1.C.

Compile the program. Reedit the text if you find any errors. Remember your semicolons and watch how the double quotes are used in the `printf()` functions.

Run the resulting program. The output looks something like this:

```
Name some jerk you know:Bill
Yeah, I think Bill is a jerk, too.
```

- `gets()` reads a variable just like `scanf()` does. Yet no matter what reads it, the `printf()` statement can display it.
- `gets(var)` is the same as `scanf("%s",var)`.
- If you get a warning error when compiling, see the next section.

Another silly command-prompt program

To see how `puts()` works, create the following program, STOP.C. Yeah, this program is really silly, but you're just starting out, so bear with me:

```
#include <stdio.h>

int main()
{
    puts("Unable to stop: Bad mood error.");
    return(0);
}
```

Save this source code to disk as STOP.C. Compile it, link it, run it.

This program produces the following output when you type **stop** or **./stop** at the command prompt:

```
Unable to stop: Bad mood error.
```

Ha, ha.

REMEMBER

- `puts()` is not pronounced "putz."
- Like `printf()`, `puts()` slaps a string of text up on the screen. The text is hugged by double quotes and is nestled between two parentheses.
- Like `printf()`, `puts()` understands escape sequences. For example, you can use `\"` if you want to display a string with a double quote in it.
- You don't have to put a `\n` at the end of a `puts()` text string. `puts()` always displays the newline character at the end of its output.
- If you want `puts()` not to display the newline character, you must use `printf()` instead.

`puts()` *and* `gets()` *in action*

The following program is a subtle modification to INSULT1.C. This time, the first `printf()` is replaced with a `puts()` statement:

```
#include <stdio.h>

int main()
{
    char jerk[20];
```

- The equal sign is used to assign a non-string value to a variable. The variable goes on the left side of the equal sign and gets its value from whatever's on the right side.

- String variables cannot be defined in this way, by using an equal sign. You cannot say

  ```
  kitty="Koshka";
  ```

  It just doesn't work! Strings can be read into variables from the keyboard by using the `scanf()`, `gets()`, or other C language keyboard-reading functions. String variables can also be preset, but you cannot use an equal sign with them, like you can with numeric variables!

Entering numeric values from the keyboard

Keep the METHUS1.C program warm in your editor's oven for a few seconds. What does it really do? Nothing. Because the value 969 is already in the program, there's no surprise. The real fun with numbers comes when they're entered from the keyboard. Who knows what wacky value the user may enter? (That's another reason for a variable.)

A small problem arises in reading a value from the keyboard: Only strings are read from the keyboard; the `scanf()` and `gets()` functions you're familiar with have been used to read string variables. And, there's most definitely a difference between the characters "969" and the number 969. One is a value, and the other is a string. (I leave it up to you to figure out which is which.) The object is to covertly transform the string "969" into a value — nay, an *integer* value — of 969. The secret command to do it is `atoi`, the A-to-I function.

The `atoi()` function

The `atoi()` (pronounced "A-to-I") function converts numbers at the beginning of a string into an integer value. The *A* comes from the acronym *ASCII,* which is a coding scheme that assigns secret code numbers to characters. So `atoi` means "convert an ASCII (text) string into an integer value." That's how you can read integers from the keyboard. Here's the format:

```
var=atoi(string);
```

var is the name of a numeric variable, an integer variable created by the `int` keyword. That's followed by an equal sign, which is how you assign a value to a variable.

- Addition symbol: +
- Subtraction symbol: –
- Multiplication symbol: *
- Division symbol: /

Incidentally, the official C language term for these dingbats is *operators.* These are mathematical (or arithmetic — I never know which to use) operators.

+ **Addition:** The addition operator is the plus sign, +. This sign is so basic that I can't really think of anything else you would use to add two numbers:

```
var=value1+value2;
```

Here, the result of adding `value1` to `value2` is calculated by the computer and stored in the numeric variable *var*.

– **Subtraction:** The subtraction operator is the minus sign, –:

```
var=value1-value2;
```


Here, the result of subtracting *value2* from *value1* is calculated and gently stuffed into the numeric variable *var.*

* **Multiplication:** Here's where we get weird. The multiplication operator is the asterisk — not the × character:

```
var=value1*value2;
```

In this line, the result of multiplying `value1` by `value2` is figured out by the computer, and the result is stored in the variable *var.*

/ **Division:** For division, the slash, `/`, is used; the primary reason is that the ÷ symbol is not on your keyboard:

```
var=value1/value2;
```

Here, the result of dividing `value1` by `value2` is calculated by the computer and stored in the variable `var`.

Note that in all cases, the mathematical operation is on the *right* side of the equal sign — something like this:

```
value1+value2=var;
```

The `%d` in the first `printf()` function looks for an integer value to "fill in the blank." The `printf()` function expects to find that value after the comma — and it does! The value is calculated by the C compiler as 65–19, which is 46. The `printf()` statement plugs the value 46 into the `%d`'s placeholder. The same holds true for the second `printf()` function.

You can do the same thing without the math. You can figure out 65–19 and 969–65 in your head and then plug in the values directly:

```
printf("Methuselah contributed to Social Security for
     %d years.\n",46);
printf("Methuselah collected from Social Security for
     %d years.\n",904);
```

Again, the result is the same. The `%d` looks for an integer value, finds it, and plugs it in to the displayed string. It doesn't matter to `printf()` whether the value is a constant, a mathematical equation, or a variable. It must, however, be an integer value.

In C, you can combine both steps into one. For example:

```
int methus=969;
```

This statement creates the integer variable `methus` and assigns it the value `969` — all at once. It's your first peek at C language shortcut. (C is full of shortcuts and alternatives — enough to make you kooky.)

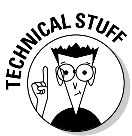
You can do the same thing with string variables — but it's a little weird. Normally, string variables are created and given a size. For example:

```
char prompt[22];
```

Here, a character string variable, `prompt`, is created and given room for 22 characters. Then you use `gets()` or `scanf()` to stick text into that variable. (You don't use an equal sign!) When you create the variable and assign it a string, however, it's given this format:

```
char prompt[] = "So how fat are you, anyway?"
```


This command creates a string variable, `prompt`. That string variable already contains the text "So how fat are you, anyway?" Notice that you see no number in the brackets. The reason is that the compiler is smart enough to figure out how long the string is and use that value automatically. No guesswork — what joy!



- Numeric variables can be assigned a value when they're declared. Just follow the variable name with an equal sign and its value. Remember to end the line with a semicolon.
- You can even assign the variable a value concocted by using math. For example:

  ```
  int video=800*600;
  ```

  This statement creates the integer variable `video` and sets its value equal to 800 times 600, or 480,000. (Remember that `*` is used for multiplication in C.)
- Even though a variable may be assigned a value, that value can still change. If you create the integer variable `methus` and assign it the value `969`, there's nothing wrong with changing that value later in the program. After all, a variable is still a variable.
- Here's a trick that's also possible, but not necessary, to remember:

  ```
  int start = begin = first = count = 0;
  ```

This statement declares four integer variables: `start`, `begin`, `first`, and `count`. Each of them is set equal to 0. `start` is equal to `begin`, which is equal to `first`, which is equal to `count`, which is equal to 0. You probably see this type of declaration used more often than you end up using it yourself.

The old random-sampler variable program

To demonstrate how variables can be defined with specific values, the ICKYGU.C program was concocted. It works like those old Chinese all-you-can-eat places, where steaming trays of yummy glop lie waiting under grease-smeared panes of sneeze-protecting glass. Ah . . . reminds me of my college days and that bowel infection I had. Here's the source code:


```
#include <stdio.h>

int main()
{
        char menuitem[] = "Slimy Orange Stuff \"Icky Woka
              Gu\"";
        int pints=1;
        float price = 1.45;

        printf("Today special - %s\n",menuitem);
        printf("You want %d pint.\n",pints);
        printf("That be $%f, please.\n",price);
        return(0);
}
```

Type this source code into your editor. Double-check everything. Save the program as ICKYGU.C.

Compile the program. Repair any unexpected errors — as well as those you may have been expecting — and recompile if need be.

Run the final program. You see something like the following example displayed:

```
Today special - Slimy Orange Stuff "Icky Woka Gu"
You want 1 pint.
That be $1.450000, please.
```

Whoa! Is that lira or dollars? Of course, it's dollars — the dollar sign in `printf()`'s formatting string is a normal character, not anything special. But the `1.45` value was printed with four extra zeroes. Why? Because you didn't tell the compiler *not* to. That's just the way the `%f`, or floating-point conversion character, displays numbers.

5.878	E12
58.78	E11
587.8	E10
5878.	E9
58780.	E8
587800.	E7
5878000.	E6
58780000.	E5
587800000.	E4
5878000000.	E3
58780000000.	E2
587800000000.	E1
5878000000000.	E0
5,878,000,000,000	

Figure 9-1: Scientific notation and the light year.

When you enter E numbers in the compiler, use the proper E format. To display the numbers in E format with `printf()`, you can use the `%e` placeholder. To see how it works, replace the `%f` in the JUPITER.C program with `%e`, save the change to disk, recompile, and run the result. The output is in E notation, something like the following:

```
Jupiter is 8.223886e-05 light years from the sun.
```

If the E has a negative number in front of it, as shown in this example, you hop the decimal point to the left *nn* places, to indicate very small numbers. You would translate the preceding value into the following:

```
.00008223886
```

- Scientific, or E, notation is required when numbers contain too many digits for the C compiler to eat.
- A negative E number means that the value is very small. Remember to move the decimal point to the *left* rather than to the right when you see this type of number.
- Some compilers allow you to use the `%E` (big E) placeholder in `printf()` to display scientific-notation numbers with a big E in them.

Beyond the `getchar()` dilemma, the program uses seven `putchar()` functions to display `Hello!` (plus a newline character) to the screen. It's a rather silly use of `putchar()`, but it works.

- The `putchar()` function is used to display a single character on the screen.
- You can also specify a character as an escape sequence or a code value with `putchar()` (see the next section).

Character Variables As Values

If you want, you can live your life secure in the knowledge that the `char` keyword sets aside storage space for single-character variables and strings. That's all well and good, and it gets you an A on the quiz. You can stop reading now, if you want.

The horrible truth is that a single-character variable is really a type of integer. It's a tiny integer, but an integer nonetheless. The reason that it isn't obvious is that treating a `char` as an integer is really a secondary function of the single-character variable. The primary purpose of single-character variables is to store characters. But they can be used as integers. It's twisted, so allow me to explain in detail.

The basic unit of storage in a computer is the byte. Your computer has so many bytes (or megabytes) of memory, the hard drive stores so many gigabytes, and so on. Each one of those bytes can be looked at as storing a single character of information. A *byte* is a character.

Without boring you with the details, know that a byte is capable of storing a value, from 0 to 255. That's the range of an unsigned `char` integer: from 0 to 255 (refer to Table 9-1, in Chapter 9). Because a character is a byte, the `char` can also be used to store those tiny integer values.

When the computer deals with characters, it doesn't really know an A from a B. It does, however, know the difference between 65 and 66. Internally, the computer uses the number 65 as a code representing the letter *A.* The letter *B* is code 66. In fact, all letters of the alphabet, number characters, symbols, and other assorted jots and tittles each have their own character codes. The coding scheme is referred to as *ASCII,* and a list of the codes and characters is in Appendix B.

Essentially, when you store the character A in a `char` variable, you place the value 65 into that variable. Internally, the computer sees only the 65 and, lo, it's happy. Externally, when the character is "displayed," an A shows up. That satisfies you and me, supposing that an A is what we want.

Chapter 11

C More Math and the Sacred Order of Precedence

In This Chapter

- Reviewing the C math operators
- Incrementing variables
- Understanding the order of precedence
- Introducing My Dear Aunt Sally
- Using parentheses to control your math

Beware ye the dreadful math chapter! Bwaa-ha-ha!

Math is so terrifying to some people that I'm surprised there isn't some math-themed horror picture, or at least a ride at Disneyland. Pirates. Ghosts. Screaming Dolls. Disneyland *needs* math in order to terrify and thrill children of all ages. Ludwig von Drake would host. But I digress.

This chapter really isn't *the* dreadful math chapter, but it's my first lecture that dwells on math almost long enough to give you a headache. Don't panic! The computer does all the work. You're only required to assemble the math in the proper order for the answers to come out right. And, if you do it wrong, the C compiler tells you and you can start over. No embarrassment. No recriminations. No snickering from the way-too-smart female exchange student from Transylvania.

An All-Too-Brief Review of the Basic C Mathematical Operators

Table 11-1 shows the basic C mathematical operators (or it could be arithmetic operators — whatever). These symbols and scribbles make basic math happen in a C program.

- The math part of the equation is calculated first and is worked from left to right. The result is then transferred to the variable sitting on the left side of the equal sign.

The old "how tall are you" program

You can use "the power of the computer" to do some simple yet annoying math. As an example, I present the HEIGHT.C program, with its source code shown next. This program asks you to enter your height in inches and then spits back the result in centimeters. Granted, it's a typically dull C language program. But, bear with me for a few pages and have some fun with it. Enter this trivial program into your editor:

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

int main()
{
    float height_in_cm;
    char height_in_inches[4];

    printf("Enter your height in inches:");
    gets(height_in_inches);
    height_in_cm = atoi(height_in_inches)*2.54;
    printf("You are %.2f centimeters tall.\n",height_in_cm);
    return(0);
}
```

Be careful with what you type; some long variable names are in there. Also, it's *height*, not *hieght*. (I mention it because I tried to compile the program with that spelling mistake — not once, but twice!) Save the file to disk as HEIGHT.C.

Compile the program. Watch for any syntax or other serious errors. Fix them if they crop up.

Run the HEIGHT program. Your output looks something like this:

```
Enter your height in inches:60
You are 152.40 centimeters tall.
```

If you're 60 inches tall (5 feet exactly), that's equal to 152.40 centimeters — a bigger number, but you're still hovering at the same altitude. The program is good at converting almost any length in inches to its corresponding length in centimeters.

```
#include <stdio.h>

int main()
{
    int total;

    total=100-25*2;
    printf("Tomorrow you will have %d magic
           pellets.\n",total);
    return(0);
}
```

Enter this program in your editor. Double-check everything. Save the file to disk as PELLETS.C.

Compile PELLETS.C. Fix any errors.

Run the PELLETS program. Your output looks like this:

```
Tomorrow you will have 50 magic pellets.
```

Uh-huh. Try explaining that to the IRS. Your computer program, diligently entered, tells you that there are 50 pellets, when tomorrow you will really have 150. The extra 100? They were lost to the order of precedence. In the PELLETS.C program, addition must come first. The way that works is by using parentheses.

Using parentheses to mess up the order of precedence

My Dear Aunt Sally can be quite overbearing. She's insistent. Still, even though she means well, she goofs up sometimes. In the PELLETS.C program, for example, she tells the C compiler to multiply 25 by 2 first and then subtract the result from 100. Anyone who reads the problem knows that you must subtract 25 from 100 first and then multiply what's left by 2. The problem is convincing the C compiler — and Aunt Sally — how to do that.

REMEMBER

You can circumvent the order of precedence by using parentheses. When the C compiler sees parentheses, it quickly darts between them, figures out the math, and then continues with multiplication, division, addition, and subtraction, in that order, from left to right, outside the parentheses.

To fix the PELLETS.C program, you have to change the seventh line to read:

```
total=(100-25)*2;
```

Chapter 12

C the Mighty if Command

In This Chapter

- Using the if statement
- Comparing values with if
- Formatting the if statements
- Handling exceptions with else
- Making multiple decisions

Okay, if isn't a command. It's *another* keyword in the C programming language, one that you can use in your program to make decisions — although it really makes comparisons, not decisions. It's the program that decides what to do based on the results of the comparison.

This chapter is about adding decision-making power to your programs by using the if command.

REMEMBER

Keep in mind that the computer doesn't decide what to do. Instead, it follows a careful path that you set down for it. It's kind of like instructing small children to do something, though with the computer, it always does exactly what you tell it to and never pauses eternally in front of the TV set or wedges a Big Hunk into the sofa.

If Only. . . .

The idea behind the if command is to have the computer handle some predictable yet unknown event: A choice is made from a menu; the little man in some game opens the door with the hydra behind it; or the user types something goofy. These are all events that happen, which the computer must deal with.

The `if` keyword allows you to put these types of decisions into your programs. The decisions are based on a comparison. For example:

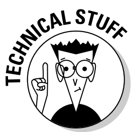
- If the contents of variable *X* are greater than variable *Y*, scream like they're twisting your nose.
- If the contents of the variable *calories* are very high, it must taste very good.
- If it ain't broke, don't fix it.
- If Doug doesn't ask me out to the prom, I'll have to go with Charley.

All these examples show important decisions, similar to those you can make in your C programs by using the `if` keyword. However, in the C programming language, the `if` keyword's comparisons are kind of, sort of — dare I say it? — mathematical in nature. Here are more accurate examples:

- If the value of variable `A` is equal to the value of variable `B`
- If the contents of variable `ch` are less than 132
- If the value of variable `zed` is greater than 1,000,000

These examples are really simple, scales-of-justice evaluations of variables and values. The `if` keyword makes the comparison, and if the comparison is true, your program does a particular set of tasks.

- `if` is a keyword in the C programming language. It allows your programs to make decisions.
- `if` decides what to do based on a comparison of (usually) two items.
- The comparison that `if` makes is mathematical in nature: Are two items equal to, greater than, less than — and so on — to each other? If they are, a certain part of your program runs. If not, that part of the program doesn't run.



- The `if` keyword creates what is known as a *selection statement* in the C language. I wrote this topic down in my notes, probably because it's in some other C reference I have read at some time or another. *Selection statement.* Impress your friends with that term if you can remember it. Just throw your nose in the air if they ask what it means. (That's what I do.)

The computer-genie program example

The following program is GENIE1.C, one of many silly computer guess-the-number programs you write when you find out how to program. Computer scientists used to play these games for hours in the early days of the computer. They would probably drop dead if we could beam a Sony PlayStation back through time.

What GENIE1.C does is to ask for a number, from 0 through 9. You type that number at the keyboard. Then, using the magic of the if statement, the computer tells you whether the number you entered is less than 5. This program was a major thigh-slapper when it was first written in the early 1950s.

Enter the following source code into your text editor. The only new stuff comes with the if statement cluster, near the end of the program. Better double-double-check your typing.

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

int main()
{
    char num[2];
    int number;

    printf("I am your computer genie!\n");
    printf("Enter a number from 0 to 9:");
    gets(num);
    number=atoi(num);

    if(number<5)
    {
        printf("That number is less than 5!\n");
    }

    printf("The genie knows all, sees all!\n");
    return(0);
}
```


Save the file to disk as GENIE1.C.

Compile GENIE1.C. If you see any errors, run back to your editor and fix them. Then recompile.

Run the final program. You see these displayed:

```
I am your computer genie!
Enter a number from 0 to 9:
```

Type a number, somewhere in the range of 0 through 9. For example, you can type 3. Press Enter and you see:

```
That number is less than 5!
The genie knows all, sees all!
```

This time, the test is greater than *or* equal to: Is the number that is entered 5 or more than 5? If the number is greater than or equal to 5, it must be more than 4, and the `printf()` statement goes on to display that important info on the screen.

The following modification to the GENIE1.C program doesn't change the `if` comparison, as in the previous examples. Instead, it shows you that more than one statement can belong to `if`:

```
if(number<5)
{
    printf("That number is less than 5!\n");
    printf("By goodness, aren't I smart?\n");
}
```


Everything between the curly braces is executed when the comparison is true. Advanced C programs may have lots of stuff in there; as long as it's between the curly braces, it's executed only if the comparison is true. (That's why it's indented — so that you know that it all belongs to the `if` statement.)

- The comparison that `if` makes is usually between a variable and a value. It can be a numeric or single-character variable.
- `if` cannot compare strings. For information on comparing strings, refer to my book *C All-in-One Desk Reference For Dummies* (Wiley).
- Less than and greater than and their ilk should be familiar to you from basic math. If not, you should know that you read the symbols from left to right: The > symbol is *greater than* because the big side comes first; the < is *less than* because the lesser side comes first.
- The symbols for less than or equal to and greater than or equal to always appear that way: <= and >=. Switching them the other way generates an error.
- The symbol for "not" in C is the exclamation point. So, != means "not equal." What is !TRUE (not-true) is FALSE. "If you think that it's butter, but it's !." No, I do ! want to eat those soggy zucchini chips.
- When you're making a comparison to see whether two things are equal, you use *two* equal signs. I think of it this way: When you build an `if` statement to see whether two things are equal, you think in your head "is equal" rather than "equals." For example:

  ```
  if(x==5)
  ```

  Read this statement as "If the value of the `x` variable *is equal* to `5`, then. . . ." If you think "equals," you have a tendency to use only one equal sign — which is very wrong.

- If you use one equal sign rather than two, you don't get an error; however, the program is wrong. The nearby Technical Stuff sidebar attempts to explain why.
- If you have programmed in other computer languages, keep in mind that the C language has no `2ewd` or `fi` word. The final curly brace signals to the compiler that the `if` statement has ended.
- Also, no `then` word is used with `if`, as in the `if-then` thing they have in the BASIC or Pascal programming language.

A question of formatting the `if` statement

The `if` statement is your first "complex" C language statement. The C language has many more, but `if` is the first and possibly the most popular, though I doubt that a popularity contest for programming language words has ever been held (and, then again, `if` would be great as Miss Congeniality but definitely come up a little thin in the swimsuit competition).

Though you probably have seen the `if` statement used only with curly braces, it can also be displayed as a traditional C language statement. For example, consider the following — one of the modifications from the GENIE1 program:

```
if(number==5)
{
    printf("That number is 5!\n");
}
```

In C, it's perfectly legitimate to write this as a more traditional type of statement. To wit:

```
if(number==5) printf("That number is 5!\n");
```

This line looks more like a C language statement. It ends in a semicolon. Everything still works the same; if the value of the number variable is equal to `5`, the `printf()` statement is executed. If `number` doesn't equal `5`, the rest of the statement is skipped.

Although all this is legal and you aren't shunned in the C programming community for using it, I recommend using curly braces with your `if` statements until you feel comfortable reading the C language.

Table 12-2 `if` Comparisons and Their Opposites

`if` *Comparison*	`else` *Statement Executed By This Condition*
<	>= (Greater than or equal to)
==	!= (Not equal to)
>	<= (Less than or equal to)
<=	> (Greater than)
>=	< (Less than)
!=	== (Is equal to)

- I don't know about you, but I think that all those symbols in Table 12-2 would certainly make an interesting rug pattern.
- The `else` keyword is used only with `if`.
- Both `if` and `else` can have more than one statement enclosed in their curly braces. `if`'s statements are executed when the comparison is true; `else`'s statements are executed when the comparison is false.
- To *execute* means to run. C programs execute, or run, statements from the top of the source code (the first line) to the bottom. Each line is executed one after the other unless statements like `if` and `else` are encountered. In that case, the program executes different statements, depending on the comparison that `if` makes.
- When your program doesn't require an either-or decision, you don't have to use `else`. For example, the TAXES program has an either-or decision. But, suppose that you're writing a program that displays an error message when something doesn't work. In that case, you don't need `else`; if an error doesn't occur, the program should continue as normal.
- If you're the speaker of another programming tongue, notice that the C language has no `end-else` word in it. This isn't smelly old Pascal, for goodness' sake. The final curly brace signals the end of the `else` statement, just as it does with `if`.

The strange case of `else-if` *and even more decisions*

The C language is rich with decision making. The `if` keyword helps if you need to test for only one condition. True or false, `if` handles it. And, if it's true, a group of statements is executed. Otherwise, it's skipped over. (After the `if`'s group of statements is executed, the program continues as before.)

The `if` command's logical friends

You can use the logical operators `&&` and `||` to help the `if` command make multiple decisions in one statement:

The `&&` is the logical `AND` operator.

The `||` is the logical `OR` operator.

Table 14-1 explains both.

Table 14-1 Logical Operators Used in `if` Comparisons

Operator	*Meaning*	*"True" Examples*
||	Or	true || true
		true || false
		false || true
&&	And	true || true

The logical operator is used on two standard `if` command comparisons. For example:

```
if(temperature>65 && temperature<75)
{
    printf("My, but it's nice weather outside\n");
}
```

In this example, the `if` command makes two comparisons: `temperature>65` and `temperature<75`. If both are true, the `&&`, logical `AND` condition is also true and the entire statement passes; the `printf()` function then displays the string. Table 14-2 shows the possibilities of how you can figure it out.

Table 14-2 Figuring Out a Logical AND Operation

Temperature	*temperature>65*	*(and)*	*temperature<75*	*Logical AND result*
45	45>65		45<75	
	FALSE	&&	TRUE	FALSE
72	72>65		72<75	

Chapter 15

C You Again

In This Chapter

- Understanding the loop
- Repeating chunks of code with `for`
- Using a loop to count
- Displaying an ASCII table by using a loop
- Avoiding the endless loop
- Breaking a loop with `break`

One thing computers enjoy doing more than anything else is repeating themselves. Humans? We think that it's punishment to tell a kid to write "*National Geographic* films are not to be giggled at" 100 times on a chalkboard. Computers? They don't mind a bit. They *enjoy* it, in fact.

Next to making decisions with `if`, the power in your programs derives from their ability to repeat things. The only problem with that is getting them to stop, which is why you need to know how `if` works before you progress into the *looping* statements. This chapter begins your looping journey by introducing you to the most ancient of the loopy commands, `for`.

- To find out about the `if` statement, refer to Chapters 12 though 14.
- It may behoove you to look at Table 12-1, in Chapter 12, which contains comparison functions used by both the `if` and `for` commands.

For Going Loopy

Doing things over and over is referred to as *looping.* When a computer programmer writes a program that counts to one zillion, he writes a loop. The *loop* is called such because it's a chunk of programming instructions — code — that is executed a given number of times. Over and over.

Compile the program. Even though the `for` statement contains a deliberate infinite loop, no error message is displayed (unless you goofed up and typed something else). After all, the compiler may think that you're attempting to do something forever as part of your master plan. How would it know otherwise?

When you run the program, forever, you see the following messages scrolling madly up your screen:

```
The computer has run amok!
```

Indeed, it has! Press Ctrl+C to stop the madness.

- Most loops are designed with a *condition* on which they end. In an endless loop, either they don't have a condition or the condition is set up in some fashion as to be unobtainable. That's bad.
- Infinite loops are insidious! Often, you don't detect them until the program runs, which is a great argument for testing every program you create.
- The Ctrl+C keyboard combination works in both Windows and Unix to cancel a command that is producing standard output, which is what FOREVER.C is doing (over and over). Other types of programs with infinite loops, particularly those that don't produce standard output, are much harder to stop. If Ctrl+C doesn't work, often you have to use your operating system's abilities to kill off the program run amok.
- In the olden days, you often had to restart the entire computer to regain control from a run-amok endlessly looping program.
- The program loops forever because of a flaw in the `for` loop's "while true" part — the second item in the parentheses:

  ```
  for(i=1;i=5;i=i+1)
  ```

  The C compiler sees `i=5` and figures, "Okay, I'll put 5 into the `i` variable." It isn't a true-false comparison, like something you find with an `if` statement, which was expected, so the compiler supposes that it's true and keeps looping — no matter what. Note that the variable `i` is always equal to 5 for this reason; even after it's incremented with `i=i+1`, the `i=5` statement resets it back to `5`.
- Here's what the `for` statement should probably look like:

  ```
      for(i=1;i<=5;i=i+1)
  ```

  This line repeats the loop five times.
- Some compilers may detect the "forever" condition in the `for` statement and flag it as an infinite loop. If so, you're lucky. For example, the old Borland C++ compiler flagged FOREVER.C as having a `Possibly incorrect assignment` error. The compiler still produces the finished (and flawed) program, though.

Counting to 1,000 by fives

The following program is an update to the old 100.C program, from Chapter 15. In this case, the program counts to 1,000 by fives — a task that would literally take days without a computer:

```
#include <stdio.h>

int main()
{
    int i;

    for(i=5;i<=1000;i=i+5)
        printf("%d\t",i);
    return(0);
}
```


Start off with a new, clean slate in your editor. Type the preceding source code. It's nothing fancy. Indeed, it's just a take-off from the old 100.C program. Save the file to disk as 1000.C.

Compile 1000.C and run the result. Your screen fills with values from `5` to `1000`, all lined up in rows and columns.

- This leaping loop counts by fives because of the `i=i+5` part of the `for` statement. The `i=i+5` operation keeps increasing the value of the `i` variable by 5.
- The loop begins counting at 5 because of the `i=5` part of the `for` loop. It stops counting at 1,000 because of the `i<=1000` part of the loop. That's "less than or equal to 1000," which is how you get to 1,000.

Cryptic C operator symbols, Volume III: The madness continues

C is full of shortcuts, and mathematical operations are where you find most of them clustered like bees over a stray Zagnut bar. I feel that the two most cryptic shortcuts are for changing a variable's value by 1: ++ to increment and `--` to decrement. But there are more!

To add 5 to a variable's value, for example, such as in the 1000.C program, you use the following:

```
i=i+5
```

- Technically, these doojabbies are referred to as *assignment operators*. Don't memorize that term. Even I had to look it up.
- Hey: It's a good idea to stick a sticky note on Table 16-2 or flag it by dog-earing the page. These cryptic shortcuts aren't easy to remember.
- One way to remember that the operator (+, -, *, or /) comes first is to look at the wrong way for subtraction:

  ```
  var=-5
  ```

  This is not a shortcut for `var=var-5`. Instead, it sets the value of variable *var* equal to negative-five. Ipso fasto, `var-=5` must be the proper way to do it.
- Remember that these mathematical-shortcut cryptic operators aren't necessarily limited to use in `for` loops. Each of them can be a C language statement unto itself, a mathematical operation to somehow pervert a variable's value. To wit:

  ```
  term+=4;
  ```

  This statement increases the value of the variable `term` by 4.

The answers

In CHANT.C, modify Line 7 to read:

```
for(i=2;i<10;i+=2)
```

In 1000.C, modify Line 7 to read:

```
for(i=2;i<10;i+=5)
```

In both cases, you change the longer equation `i=i+x` to its shorter variation, `i+=x`.

The while keyword (a formal introduction)

The while keyword is used in the C language to repeat a block of statements. Unlike the for loop, while only tells the computer when to end the loop. The loop must be set up before the while keyword, and when it's looping, the ending condition — the sizzling fuse or ticking timer — must be working. Then, the loop goes on, la-de-da, until the condition that while monitors suddenly becomes FALSE. Then, the party's over, and the program goes on, sadder but content with the fact that it was repeating itself for a while (sic).

Here's the rough format:

```
starting;
while(while_true)
{
        statement(s);
        do_this;
}
```


First, the loop must be set up, which is done with the *starting* statement. For example, this statement (or a group of statements) may declare a variable to be a certain value, to wait for a keystroke, or to do any number of interesting things.

while_true is a condition that while examines. If the condition is TRUE, the *statements* enclosed in curly braces are repeated. while examines that condition after each loop is repeated, and only when the statement is FALSE does the loop stop.

Inside the curly braces are *statements* repeated by the while loop. One of those *statements, do_this,* is required in order to control the loop. The *do_this* part needs to modify the *while_true* condition somehow so that the loop eventually stops or is broken out of.

While loops have an advantage over for loops in that they're easier to read in English. For example:

```
while(ch!='~')
```

This statement says "While the value of variable ch does not equal the tilde character, repeat the following statements." For this to make sense, you must remember, of course, that ! means *not* in C. Knowing which symbols to pronounce and which are just decorations is important to understanding C programming.

- Most of the problems Microsoft has had with critical or fatal errors in its software are caused by a *lack* of this type of bounds checking.
- Refer to Chapter 14 for more information about the logical `||` (`OR`) comparison.
- TIP: You may want to insert the following comment into your source code, just above the first loop:

```
/* This loop ensures they type in
a proper value */
```

Nested Loops and Other Bird-Brained Concepts

Glorious loops within loops, wheels within wheels, spinning 'round like some nauseating amusement park ride with a drugged-out, tattooed guy named Craig asleep at the controls. But that's another subject. In the C programming language, spinning two loops is a cinchy and practical thing to do. It's called making a *nested loop,* or with one loop inside another.

Adding a tense, dramatic delay to the COUNTDWN.C program

What's missing from the COUNTDWN.C program is a little tension. In case you haven't noticed, typing any value from 1 to 100 doesn't really affect the speed at which the countdown is displayed; after you press Enter, the text *zips* on up the screen. No suspense!

To help slow down the display, you can insert a delay loop into the program. The purpose of the delay loop is merely to spin the computer's CPU, burning up clock cycles to slow down the program at a certain point. Yes, you do it on purpose.

Modify the second `do while` loop in the COUNTDWN.C program to read:

```
do
{
    printf("T-minus %d\n",start);
    start--;
    for(delay=0;delay<100000;delay++);   /* delay loop */
}
while(start>0);
```

- Having a `for` loop inside a `while` loop is referred to as a *nested loop*. Note that both loops don't need to be of the same type (two `for` loops or two `while` loops).
- A nested loop is basically one loop spinning 'round inside another loop.
- The first loop, or outside loop, ticks off first. Then, the inside loop ticks off, looping as many times as it does. After that, the outside loop ticks off another one, and then the inside loop is repeated entirely again. That's how they work.

WARNING!

- Keep separate the variables associated with one loop or another. For example, the following two `for` loops are nested improperly:

```
for(x=0;x<5;x++)
     for(x=5;x>0;x--);
```

  Because `x` is used in both loops, these nested loops don't behave as you expect. This loop is infinite, in fact, because both are manipulating the same variable in different directions.

WARNING!

- This disaster probably isn't apparent to you. You write some huge program and nest two `for` loops miles apart without thinking about it, by using your favorite variable `x` (or `i`) in each one. Those kind of bugs can wreck your day.

TIP

- The way to avoid messing up nested loops is to use different variables with each one — for example, `a` or `b`, or `i1` and `i2`, or even something descriptive, such as `start` and `delay`, as used in the COUNTDWN.C example.
- That nested `for` loop in COUNTDWN.C ends with a semicolon, indicating that it doesn't "own" any statements that are repeated. Here's another way you could format it:

```
for(delay=0;delay<100000;delay++)
     ;
```

  This example shows you that the `for` loop doesn't have any statements worth repeating. It just sits and spins the microprocessor, wasting time (which is what you want).
- Although delay loops, such as the one shown in COUNTDWN.C, are common, a better way exists. That is to use the computer's internal clock to time a delay of a specific duration. I show you an example in *C All-in-One Desk Reference For Dummies* (Wiley).

TECHNICAL STUFF

- My first IBM PC — some 20 years ago — required a *delay* loop that counted to only 10,000 for about a half-second pause between each line displayed. Today's computers are much, much faster — obviously!

Sleepy time!

The C language does have a built-in delay function, so you really have no need to program a delay loop — as long as you can stand the wait!

The `sleep()` function is used to pause a program for a given number of seconds. Yes — I said *seconds.* You specify the seconds to wait in `sleep()`'s parentheses:

```
sleep(40);
```

You can catch 40 winks — or 40 seconds — of wait time while a program is running.

You can replace the for delay loop in COUNTDWN.C with

```
sleep(1);
```

This line adds a dramatic pause between each line's output — a slow, dramatic, and often maddening pause. But, it works.

Note that in some implementations of GCC, the `sleep()` function apparently uses milliseconds, not seconds, as its argument. To delay one second, for example, you use this command in COUNTDWN.C:

```
sleep(1000);
```

Keep in mind that this implementation of the `sleep()` function is nonstandard.

The nitty GRID.C of nested loops

Nested loops happen all the time. Most often, they happen when you're filling in a grid or an array. In that case, you work on rows and columns, filling up the columns row-by-row, one after the other, or vice versa. An example of how it's done is shown in the GRID.C program, which displays a grid of numbers and letters:

```
#include <stdio.h>

int main()
{
    int a;
    char b;

    printf("Here is thy grid...\n");

    for(a=1;a<10;a++)
    {
        for(b='A';b<'K';b++)
        {
            printf("%d-%c ",a,b);
        }
        putchar('\n');  /* end of line */
    }
    return(0);
}
```

- Keep in mind that although `continue` forces another spin of the loop's wheel, it doesn't reinitialize the loop. It tells the compiler to "go again," not to "start over."
- You should keep in mind only two real warnings about the `continue` command: Don't use it outside a loop or expect it to work on nested loops; and be careful where you put it in a `while` loop, lest you skip over the loop's counter and accidentally create an endless loop.
- As a final, consoling point, this command is rarely used. In fact, many C programmers may be a little fuzzy on what it does or may not know precisely how to use it.

Chapter 19

Switch Case, or, From 'C' to Shining 'c'

In This Chapter

- Solving the endless `else-if` puzzle
- Using `switch-case`
- Creating a `switch-case` structure

Honestly, I don't believe that `switch-case` is really a loop. But the word *loop* works so much better than my alternative, *structure thing.* That's because the statements held inside the `switch-case` structure thing aren't really repeated, yet in a way they are. Well, anyway.

This chapter uncovers the final kind-of-loop thing in the C language, which is called `switch-case`. It's not so much a loop as it's a wonderful method of cleaning up a potential problem with multiple `if` statements. As is true with most things in a programming language, it's just better for me to show you an example than to try to explain it. That's what this chapter does.

The Sneaky `switch-case` Loops

Let's all go to the lobby,
Let's all go to the lobby,
Let's all go to the lobby,
And get ourselves a treat!

— Author unknown

And, when you get to the lobby, you probably order yourself some goodies from the menu. In fact, management at your local theater has just devised an

Run the program again, enter **123412341234xx92431=**, and then press Enter:

```
Beverage        $8.00
Candy           $5.50
Hot dog         $10.00
Popcorn         $7.50
Beverage        $8.00
Candy           $5.50
Hot dog         $10.00
Popcorn         $7.50
Beverage        $8.00
Candy           $5.50
Hot dog         $10.00
Popcorn         $7.50
Improper selection.
Improper selection.
Improper selection.
Candy           $5.50
Popcorn         $7.50
Hot dog         $10.00
Beverage        $8.00
Total of $124.00
Please pay the cashier.
```


This is the last time I'm taking all you guys to the lobby!

- Most programs employ this exact type of loop. The `while(!done)` spins 'round and 'round while a `switch-case` thing handles all the program's input.
- One of the `switch-case` items handles the condition when the loop must stop. In LOBBY3.C, the key is the equal sign. It sets the value of the `done` variable to 1. The `while` loop then stops repeating.
- C views the value `0` as `FALSE`. So, by setting `done` equal to `0`, by using the `!` (not), the `while` loop is executed. The reason for all this is so that the loop `while(!done)` reads "while not done" in English.
- The various `case` structures then examine the keys that were pressed. For each match `1` through `4`, three things happen: The item that is ordered is displayed on the screen; the total is increased by the cost of that item (`total+=3`, for example); and a `break` statement busts out of the `switch-case` thing. At that point, the `while` loop continues to repeat as additional selections are made.
- You may remember the += thing, from Chapter 16. It's a contraction of `total = total + value`.

Chapter 20

Writing That First Function

In This Chapter

- Understanding functions
- Creating the `jerk()` function
- Prototyping functions
- Using the upside-down prototype
- Formatting and naming functions

Functions are where you "roll your own" in the C language. They're nifty little *procedures*, or series of commands, that tell the computer to do something. All that's bundled into one package, which your program can then use repeatedly and conveniently. In a way, writing a function is like adding your own commands to the C language.

If you're familiar with computer programming languages, you should recognize functions as similar to subroutines or procedures. If you're not familiar with computer programming (and *bless you*), think of a function as a shortcut. It's a black box that does something wonderful or mysterious. After you construct the function, the rest of your program can use it — just like any other C language function or keyword. This chapter definitely puts the *fun* into function.

(This chapter is the first one in this book to lack a clever use of the letter *C* in the title. Yeah, I was getting sick of it too — C sick, in fact.)

Meet Mr. Function

Are functions necessary? Absolutely! Every program must have at least one function, the `main()` function. That's required. Beyond that, you don't need to create your own functions. But, without them, often your code would contain a great deal of duplicate instructions.

```
    printf("Not once, or twice, but three times a day!\n");
    jerk();
    printf("He insulted my wife, my cat, my mother\n");
    printf("He irritates and grates, like no other!\n");
    jerk();
    printf("He chuckles it off, his big belly a-heavin'\n");
    printf("But he won't be laughing when I get even!\n");
    jerk();
    return(0);
}

/* This is the jerk() function */

void jerk()
{
    printf("Bill is a jerk\n");
}
```


When you're done, resave BIGJERK2.C to disk. Recompile, and you shan't be bothered by the various warning errors again.

- The prototype is basically a rehash of a function that appears later in the program.
- The prototype must shout out what type of function the program is and describe what kind of stuff should be between the parentheses.
- The prototype must also end with a semicolon. This is *muy importanto.*
- I usually copy the first line of the function to the top of the program, paste it in there, and then add a semicolon. For example, in BIGJERK2.C, I copied Line 21 (the start of the `jerk` function) to the top of the source code and pasted it in, adding the necessary `void`s and semicolon.
- No, the `main()` function doesn't have to be prototyped. The compiler is expecting it and knows all about it. (Well, almost. . . .)
- Required prototyping is something they added to the C language after it was first introduced. You may encounter older C source code files that seem to lack any prototyping. Back in the days when such programs were written (before about 1990), this was a common way of doing things.

A sneaky way to avoid prototyping problems

Only the coolest of the C language gurus do this trick — so don't tell anyone that you found out about it in a *For Dummies* book! Shhhh!

- If your source code has more than one function, the order in which they're listed is important; you cannot use a function inside your source code unless it has first been declared or prototyped. If you have multiple functions in your source code, order them so that if one function calls another, that second function is listed first. Otherwise, you're again saddled with prototyping errors.

The Tao of Functions

The C language allows you to put as many functions as you want in your source code. There really is no limit, though most programmers like to keep their source-code text files to a manageable size.

- What is "manageable size"? It depends.
- The larger the source code file, the longer it takes to compile.
- Oftentimes, it pays to break off functions into their own, separate source code files. It not only aids in debugging, but also makes recompiling larger files easier.
- This book's companion volume, *C All-in-One Desk Reference For Dummies* (Wiley), contains information on creating and managing multimodule source code files.

The function format

Here's the format of a typical function:

```
type name(stuff)
```

The `type` tells the compiler whether the function returns a value. If the type is `void`, the function doesn't return any value. (It merely *functs*.) Otherwise, the type describes which type of value the function returns: `char`, `int`, `float`, or any of the standard C language variable declarations.

The *`name`* is the function's name. It must be a unique name, not any keywords or names of other C language library functions, such as `printf()` or `atio()`. (For more on names, see the next section.)

Parentheses after the function's name are required, as they are on all C language functions. The `stuff` inside the parentheses, if needed, defines whatever value (or values) are sent off to the function for evaluation, manipulation, or mutilation. I cover this subject in Chapter 22. If there's no *stuff,* the parentheses can be left empty or the word `void` can be used.

Chapter 21

Contending with Variables in Functions

In This Chapter

- Naming variables within functions
- Understanding local variables
- Sharing one variable throughout a program
- Using global variables

Each function you create can use its own, private set of variables. It's a must. Just like the `main()` function, other functions require integer or character variables that help the function do its job. A few quirks are involved with this arrangement, of course — a few head-scratchers that must be properly mulled over so that you can understand the enter function/variable gestalt.

This chapter introduces you to the strange concept of variables inside functions. They're different. They're unique. Please avoid the desire to kill them.

Bombs Away with the BOMBER Program!

The `dropBomb()` function in the BOMBER.C program uses its own, private variable *x* in a `for` loop to simulate a bomb dropping. It could be an exciting element of a computer game you may yearn to write, though you probably want to use sophisticated graphics rather than the sloppy console screen used here:

```
#include <stdio.h>

void dropBomb(void);                /* prototype */

int main()
```

- Global variables are declared outside of any function. It's typically done right before the `main()` function.
- Everything you know about creating a variable, other than being declared outside a function, applies to creating global variables: You must specify the type of variable (`int`, `char`, and `float`, for example), the variable's name, and the semicolon.
- You can also declare a group of global variables at one time:

  ```
  int score,tanks,ammo;
  ```

- And, you can preassign values to global variables, if you want:

  ```
  char prompt[]="What?";
  ```

An example of a global variable in a real, live program

For your pleasure, please refer *again* to the BOMBER.C source code. This final modification adds code that keeps a running total of the number of people you kill with the bombs. That total is kept in the global variable `deaths`, defined right up front. Here's the final source code, with specific changes noted just afterward:

```
#include <stdio.h>

#define COUNT 20000000            /* 20,000,000 */

void dropBomb(void);              /* prototype */
void delay(void);

int deaths;                       /* global variable */

int main()
{
    char x;

    deaths=0;
    for(;;)
    {
        printf("Press ~ then Enter to quit\n");
        printf("Press Enter to drop the bomb:");
        x=getchar();
        fflush(stdin);            /* clear input buffer */
        if(x=='~')
        {
            break;
        }
```

Chapter 22

Functions That Actually Funct

In This Chapter

- ▶ Sending a value to a function
- ▶ Sending multiple values to a function
- ▶ Using the `return` keyword
- ▶ Understanding the `main()` function
- ▶ Writing tighter code

A function is like a machine. Although the do-nothing void functions that you probably have read about in earlier chapters are still valid functions, the real value in a function is having it do something. I mean, functions must chew on something and spit it out. Real meat-grinder stuff. Functions that funct.

This chapter explains how functions can be used to manipulate or produce information. It's done by sending a value to a function or by having a function return a value. This chapter explains how all that kooky stuff works.

Marching a Value Off to a Function

Generally speaking, you can write four types of functions:

- **Functions that work all by themselves, not requiring any extra input:** These functions are described in previous chapters. Each one is a ho-hum function, but often necessary and every bit a function as a function can be.
- **Functions that take input and use it somehow:** These functions are passed values, as either constants or variables, which they chew on and then do something useful based on the value received.
- **Functions that take input and produce output:** These functions receive something and give you something back in kind (known as *generating a value*). For example, a function that computed your weight based on your shoe size would swallow your shoe size and cough up your weight. So to speak. Input and output.

TECHNICAL STUFF

Another way to argue with a function

This book shows you the modern, convenient way of declaring variables (or *arguments*) shuffled off to a function. To wit:

```
void jerk(int repeat, char c);
{
and so on. . . .
```

You can also use the original format:

```
void jerk(repeat, c)
int repeat;
char c;
{
and so on...
```

This declaration does the same thing, but it's a little more confusing because the variable name is introduced first and then the "what it is declaration" comes on the following line (or lines). Otherwise, the two are the same.

My advice is to stick with the format used in this book and try not to be alarmed if you see the other format used. Older C references may use the second format, and certain fogey C programmers may adhere to it. Beware!

Functions That Return Stuff

For some functions to properly funct, they must return a value. You pass along your birthday, and the function magically tells you how old you are (and then the computer giggles at you). This process is known as returning a value, and a heck of a lot of functions do that.

Something for your troubles

WARNING!

To return a value, a function must obey these two rules:

Warning! Rules approaching.

- The function has to be *defined* as a certain type (`int`, `char`, or `float`, for example — just like a variable). Use something other than `void`.
- The function has to return a value.

The function type tells you what type of value it returns. For example:

```
int birthday(int date);
```

The function `birthday()` is defined on this line. It's an integer function and returns an integer value. (It also requires an integer parameter, `date`, which it uses as input.)

Finally, the computer tells you how smart it thinks you are

The following program calculates your IQ. Supposedly. What's more important is that it uses a function that has real meaning. If you have read the past few chapters, you have used the following set of C language statements to get input from the keyboard:

```
input=gets();
x=atoi(input);
```

The `gets()` function reads in text that's typed at the keyboard, and `atoi()` translates it into an integer value. Well, ho-ho, the `getval()` function in the IQ.C program does that for you, returning the value happily to the `main()` function:


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

int getval(void);

int main()
{
    int age,weight,area;
    float iq;

    printf("Program to calculate your IQ.\n");
    printf("Enter your age:");
    age=getval();
    printf("Enter your weight:");
    weight=getval();
    printf("Enter the your area code:");
    area=getval();

    iq=(age*weight)/area;
    printf("This computer estimates your IQ to be %f.\n",iq);
    return(0);
}

int getval(void)
{
    char input[20];
    int x;

    gets(input);
    x=atoi(input);
    return(x);
}
```

- Before the ANSI standard, the `main()` function was commonly declared as a `void`:

  ```
  void main()
  ```

  You may see this line in some older programming books or source code examples. Note that *nothing* is wrong with it; it doesn't cause the computer to error, crash, or explode. (Nor has there ever been a documented case of declaring `void main()` ever being a problem on any computer.) Even so, it's the standard now to declare `main()` as an `int`. If you don't, zillions of upset university sophomores will rise from the Internet to point fingers at you. Not that it means anything, but they will point at you.

Give that human a bonus!

The following program, BONUS.C, contains a function that has three — count 'em, three — `return` statements. This program proves that you can stick a `return` plunk in the middle of a function and no one will snicker at you — not even university sophomores.

```
#include <stdio.h>

float bonus(char x);

int main()
{
    char name[20];
    char level;
    float b;

    printf("Enter employee name:");
    gets(name);
    printf("Enter bonus level (0, 1 or 2):");
    level=getchar();
    b=bonus(level);
    b*=100;
    printf("The bonus for %s will be $%.2f.\n",name,b);
    return(0);
}

/* Calculate the bonus */

float bonus(char x)
{
    if(x=='0') return(0.33);         /* Bottom-level bonus */
    if(x=='1') return(1.50);         /* Second-level bonus */
    return(3.10);                    /* Best bonus */
}
```

The long, boring way:

```
float bonus(char x)
{
    int v;

    if(x=='0')
    {
        v=0.33;
    }
    else if(x=='1')
    {
        v=1.50;
    }
    else
    {
        v=3.10;
    }
    return(v);
}
```


The long, boring way minus all the darn curly braces:

```
float bonus(char x)
{
    int v;

    if(x=='0')
        v=0.33;
    else if(x=='1')
        v=1.50;
    else
        v=3.10;
    return(v);
}
```

And, without the integer variable *v*:

```
float bonus(char x)
{
    if(x=='0')
        return(0.33);
    else if(x=='1')
        return(1.50);
    else
        return(3.10);
}
```

```
Able    Baker   Charlie
1       2       3
Alpha   Beta    Gamma
```

Though the \ts in the printf statements look sloppy, the output is definitely organized. Tabular, dude!

- The "tab stops" are preset to every eighth column in C's output. Using a \t inserts a given number of space characters in the output, lining up the next bit of text at the next tab stop. I mention this because some people assume that the tab always moves over eight (or however many) characters. That is not the case.
- The \f and \v characters display special symbols at the Windows command prompt. Rather than a form feed, \f displays the ankh character. Rather than a vertical tab, \v displays the male symbol.
- As long as you know a character's hexadecimal code value, you can always get it displayed by using the \x escape sequence. Just plug in the hexadecimal code and there you go!

The Complex `printf()` Format

The printf() function can also be used to display the contents of variables, which you have been seeing throughout this book with integer variables and the %d placeholder, character variables and %c, and so on. To make it happen, printf() uses this format:

```
printf("format_string"[,var[,...]]);
```

Text still appears in double quotes, but after it's used to display the values in variables, it becomes a *format string*. (It's still the same text in double quotes.) The format string is followed by one or more variables, *var*.

Those *var* variables are plugged in to appropriate spots in the format_string according to special percent-sign placeholders. Those percent-sign placeholders are called *conversion characters*. For example:

```
printf("Yeah, I think %s is a jerk, too.\n",jerk);
```

The format string is text that printf() displays on the screen: Yeah, I think ____ is a jerk, too. The %s is a conversion character — a blank — that must be filled by a string of text. (I call them placeholders, but the lords of C claim that they're conversion characters.)

the random numbers more random. To plant the seed, you use the `srand()` function.

The `srand` function is used to help kick off the computer's random-number machine in a more *random* manner. Here's the format:

```
void srand((unsigned)seed)
```

The `seed` value is an unsigned integer value or variable, ranging from 0 up to 65,000-something. It's that value the compiler uses to help *seed* the random-number-generation equipment located in the bowels of your PC.

You must include the following line at the beginning of your source code to make the `srand()` function behave:

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

Because the `rand()` function already requires this line, you have no need to specify it twice (unless you're just seeding the random-number generator out of some perverse horticultural lust).

- The `(unsigned)` deal is used to ensure that the number `srand()` uses is of the unsigned type (not negative). It's known as *type casting*.
- Using the value `1` (one) to seed the random-number generator causes the compiler to start over, by using the same, uninspirational numbers you witness when `srand()` isn't used. Avoid doing that, if possible.

Randoming up the RANDOM program

Now comes the time for some really random numbers. The following source code is for RANDOM2.C, a mild modification to the original program. This time, a new function is added, `seedrnd()`, which lets you reset the random-number generator and produce more random numbers:

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

int rnd(void);
void seedrnd(void);

int main()
{
    int x;

    seedrnd();
    puts("Behold! 100 Random Numbers!");
```

```
    for(x=0;x<100;x++)
        printf("%d\t",rnd());
    return(0);
}

int rnd(void)
{
    int r;

    r=rand();
    return(r);
}

/* seed the random number */

void seedrnd(void)
{
    int seed;
    char s[6];

    printf("Enter a random number seed (2 - 65000):");
    seed=(unsigned)atoi(gets(s));
    srand(seed);
}
```


Type this program into your editor. You can start by editing the RANDOM1.C source code. Add the prototype for `seedrnd()` up front, and then insert the call to `seedrnd()` in the `main()` function. Finally, tack the `seedrnd()` function itself to the end of the source code. Double-check the whole thing before you save it to make sure that you don't leave anything out.

Use your editor's Save As command to save the file to disk as RANDOM2.C.

Compile and run. You see this line:

```
Enter a random number seed (2-65000):
```

Type a number, from 0 up to 65,000-something. Press Enter and you see a new and more random bunch of numbers displayed.

The true test that it worked is to run the program again. This time, type a different number as the seed. The next batch of random numbers is completely different from the first.

TIP

- You have to seed the randomizer only once, as this program does up in the `main()` function. Some purists insist on calling the `seedrnd()` function (or its equivalent) lots of times. Hey, random is random as random can be with a computer. No sense in wasting time.

Suppose that big is a big number in this statement:

```
m = blah % 5;
```

The values of variable m are in the range of 0 through 4, depending on the remainder of big divided by 5.

The values of variable m for the following statement are either 0 or 1, depending on whether oddoreven is even or odd, respectively:

```
m = oddereven % 2;
```

For example, a die has six sides. Suppose that the computer coughs up the random value 23,415. To pare it to a multiple of 6, you use this line:

```
dice1=23415 % 6;
```

The computer calculates how many times 6 gazinta 23,415. It then places the remainder in the dice1 variable. (The result is the number 3, which is a more realistic roll of a die than 23,415.)

- If the second value is *larger* than the first, as in 5 % 10, the result is always equal to the second value. Therefore, you want the larger value to come first in a modulus operation.
- The modulus operator is %, the percent sign. Pronounce it "mod."
- No math! The modulus is used to help you pare your random numbers. That's all! You can dwell on the mathematical aspects of the % in other C language books.
- *Gazinta* means "goes into." I point it out here because my managing editor loathes it when I use nondictionary words.
- If you want to pare a large random number as a roll of the dice, you need this equation:

  ```
  dice1=(random_value % 6)+1;
  ```

  The *random_value* the computer produces must be pared via % 6 (mod 6). It produces a number in the range of 0 to 5 (0 to 5 as a remainder — you can't have a remainder of 6 when you divide by 6.) After the % calculation, you add 1 to the number and get a value in the range of 1 to 6, which are the true numbers on any given side of a die.
- In the My Dear Aunt Sally theme of things, a modulus operation comes just after division and before addition. See the nearby Technical Stuff sidebar, "Introducing My Dear Mother's Aunt Sally (Ugh!)."
- "Ah, yes, Dr. Modulus. I'm familiar with your work in astrogenetics. Is it true that you got kicked out of the academy for engineering a third gender in mice?" "You read too much, lad."

```
    return(0);
}

int rnd(int range)
{
      int r;

      r=rand()%range;
      return(r);
}

void seedrnd(void)
{
      srand((unsigned)time(NULL));
}
```

Create the source code for RANDOM4.C. Start with your RANDOM3.C program, and make modifications per the source code just shown. Save the file to disk by using the name RANDOM4.C.

Compile and run the program. Here's a sample of the output you may see:

```
 4     1     3     0     6     6     1     0     8     9
 2     9     5     9     8     7     6     8     0     9
 5     6     2     0     5     8     5     5     9     0
 9     9     2     6     1     2     0     2     0     7
 8     4     4     7     1     6     0     0     5     1
 3     7     1     2     1     2     5     0     8     5
 9     2     0     7     9     8     4     5     6     0
 8     8     7     6     0     8     3     9     3     4
 0     4     0     5     5     6     3     0     4     3
 7     6     1     2     2     7     6     7     4     8
```

Everything is in the range of `0` through `9`, which is what the `rnd(10)` call does in Line 14.

- The `rnd()` and `seedrnd()` functions become handy as you write your own C programs — specifically, games. Feel free to copy and paste these functions to other programs you may write. Remember that both require the `#include <stdlib>` directive, with `seedrnd()` also requiring `#include <time.h>`.
- To generate a roll of the dice, you stick the `rnd()` function in your program and use this statement:

```
dice=rnd(6)+1;            /* Roll dem bones! */
```

- Using the ever-collapsing C language function ability, you can rewrite the `rnd()` function to only one statement:

```
return(rand()%range);
```

- You're now only moments from writing your own Monopoly game. . . .

Interacting with the Command Line

In Chapter 22, you may have read briefly about how the `main()` function returns a value to the operating system when the program quits. That's one way that a program can communicate with the operating system. The other way is to read in options directly from the command line. For example:

```
grep pirntf *.c
```

This shell command searches for misspellings in your C language source code. The command has two command-line arguments: `pirntf` and `*.c`. These two strings of text are passed to the `main()` function as arguments as well, which the program can then evaluate and act on, just as arguments passed to any function.

The problem with introducing such a thing in this book is that you need to understand more about arrays and pointers to be able to deal with the information passed to the `main()` function. That too will have to wait for another day.

Disk Access

One of the reasons you have a computer is to store information and work on it later. The C language is equipped with a wide variety of functions to read and write information from and to the disk drives. You can save data to disk using C just as you can with any program that has a File➪Save command — though in C, it is *you* who writes the File➪Save command.

Interacting with the Operating System

The C language also lets you perform operating system functions. You can change directories, make new directories, delete files, rename files, and do a host of other handy tasks.

You can also have your programs run other programs — sometimes two at once! Or, your program can run operating system commands and examine the results.

Finally, you can have your program interact with the environment and examine the state of your computer. You can even run services or prowl out on the network. Just about anything the computer can do, you can add into your program and do it yourself.

Chapter 28

Ten Tips for the Budding Programmer

In This Chapter

- Using the command history
- Keeping your editor open in another window
- Enjoying a color-coded editor
- Knowing your editor's line number commands
- Keeping a command-prompt window open
- Understanding a few commands
- Naming your variables
- Solving incrementing and decrementing riddles
- Breaking out of a loop

Here are some of my top-notch suggestions for programmers just starting out. Man, I wish I had had this list back in the steam-powered computer days, when I first started learning how to program.

Use the Command-Line History

Going back and forth between your editor and compiler at the command prompt gets tedious. Fortunately, most of the command-line shells out there (in both Unix and Windows) have a command-repeat button. Use it!

For example, if you press the up-arrow key, you can recall the preceding command line. Pressing the up-arrow key again recalls the command before that. If you find that you're reediting a lot, use the up-arrow key to recall the command to start your editor, and ditto for the commands to recompile.

uses different colors to present it to you on the screen. This feature is so useful that if you ever go back to a monochrome editor, you notice that it slows you down!

- To activate the colors in Vim, type a colon, **:**, and then **syntax enable**, and press Enter.
- When you're running Vim in a Windows window, choose Syntax➪Automatic so that C language keywords are highlighted.
- In Unix, to keep `syntax enable` activated, edit or create a file named `.vimrc` in your home directory. Into that file, add or include the following command:

  ```
  :syntax enable
  ```

  Then save the `.vimrc` file back to disk.
- Another bonus to highlighted text is that you can easily spot missing quotes; text between quotes is color-coded, so if a quote is missing, the source code looks like blech.
- Turn on auto-indenting if your editor has such a feature. Vim turns on auto-indenting when you use the syntax-enable command, or choose Syntax➪Automatic from the menu.

Know the Line-Number Commands in Your Editor

The C language compiler reports errors in your source code and lists the lines on which the errors occur. If your text editor displays line numbers, you can easily locate the specific line containing the error and then fix the error.

- In Windows Notepad, you can display the line and column number on the status bar. To do so, first ensure that Word Wrap is off (choose Format➪Word Wrap if necessary), and then choose View➪Status Bar. (Note that the Status Bar command may not be available in earlier versions of Notepad.)
- Vim displays the cursor's position on the bottom of the window, toward the right side. (The line number is followed by a comma and the column number, shown as `1,1` in Figure A-1 in Appendix A.)
- In Vim, the command to go to a specific line is G. For example, if the compiler reports an error in Line 64, type **64G** and VIM instantly jumps to Line 64. Think "Line number, Goto" to remember this trick.

Breaking Out of a Loop

All your loops need an exit point. Whether that point is defined in the loop's controlling statement or set inside the loop by using a `break` command, be sure that it's there!

I recall many a time sitting at the computer and waiting for the program to "wake up," only to realize that it was stuck in a loop I had programmed with no escape clause. This is especially easy to do when you work on larger programs with "tall" loops; after the source code for the loop extends past the height of your text editor, it's easy to lose track of things.

Work on One Thing at a Time

Address your bugs one at a time. Even if you're aware that the program has several things wrong with it, fix them methodically.

For example: You notice that the title is too far to the right, random characters are at the bottom of the screen, and the scrolling technique doesn't move the top row. Avoid the temptation to address all three issues in the same editing job. Instead, fix one problem. Compile and run to see how that works. Then fix the next problem.

The problem you run into when you try to fix too much at once is that you may introduce *new* errors. Catching those is easier if you remember that you were working on, for example, only Lines 173 and 174 of your source code.

Break Up Your Code

As your source code gets larger, consider breaking off portions into separate modules. I know that this topic isn't covered in this book — and it probably isn't a problem you will encounter soon — but separate modules can really make tracking bugs easy.

Even if you don't use modules, consider using comments to help visually break up your code into separate sections. Even consider announcing the purpose of each section, such as

```
/*********************************************
Verification function

This function takes the filename passed to it
and confirms that it's a valid filename and
that a file with that name doesn't already
exist.

Returns TRUE/FALSE as defined in the header.
*********************************************/
```

I also put a break between functions, just to keep them visually separated:

```
/*********************************************/
```

Set Breakpoints

You know that the bug is in the `windshield()` function, but you don't know where. Does the bug lurk at the beginning of your code? In the initialization routines? Just before the big math functions? Near the end? Where? Where? Where?

One way to find out is to put breakpoints into your program. At a certain spot, stick in a `return()` or `exit()` function, which immediately stops the program. That way, you can narrow down the pesky code. If the program stops per the breakpoint, the fault lies beyond that point in the program. If the program doesn't stop with the breakpoint, the fault lies before it.

Monitor Your Variables

Sometimes, a program runs amok because the values that you suspected were in your variables just aren't there. To confirm that the variables aren't carrying something outrageous, occasionally toss in a `printf()` statement to display their values to the screen. Never mind if this technique screws up the display; the purpose is debugging.

For example, I had a program with a nasty endless loop in it. I couldn't figure out for the life of me why it repeated and repeated. Talking through the source code did nothing. But after I stuck in a `printf()` statement that displayed the looping variable's value, I noticed that it merrily skipped over the end-of-loop value and kept incrementing itself to infinity and beyond. I added a simple `if` statement to fix the problem, and the program ran just fine afterward.

Document Your Work

At university, they're on you like gum on a theater floor about *comments*. Comment this! Comment that! I remember seeing classmates turn in projects that were three pages of greenbar paper in length, and half of that consisted of the dumb comments at the "top" of the program. Such nonsense impresses no one.

True, document your work. But documentation merely consists of notes to a future version of yourself. It's a reminder to say "This is what I was thinking" or "Here is where my train of thought is going."

You don't have to document every little stupid part of the program. This comment is useless:

The C language compiler

Thanks to the C language's popularity, many compilers are available for you to use with this book. I do, however, recommend the following:

Windows: If you're using Windows, I recommend that you get a GCC-compatible C compiler. A list of compilers is provided on this book's Web page, at `www.c-for-dummies.com`.

For this book, I used the MinGW compiler, which comes with the Dev-C++ IDE (Integrated Development Environment). It's free and available from `www.bloodshed.net`.

Whichever compiler you use, note its location on your PC's hard drive. You have to use this location to create a batch file or modify your system's path so that you can access the compiler from any folder on your disk system. More on that later.

- Other compilers are out there, including the best-selling Microsoft Visual C++ (MSVC). If you have MSVC, fine; you should be okay with running the programs in this book. Note, however, that I'm not familiar with the current version of MSVC and don't refer to it in this book, nor can I answer questions about it via e-mail. If you don't have MSVC, you have no reason to buy it.
- Plenty of free, shareware, and open-source C compilers are available on the Internet.
- If you have other books on the C language, check in the back of the book for a free compiler.
- TIP: Any GCC- or GNU-compatible C compiler works best with this book.

Linux, FreeBSD, or Mac OS X: If you're using any of these variations of Unix, you should already have the GCC compiler installed and ready to use. To confirm, open a terminal window and type the following line at the command prompt:

```
gcc -v
```

The version number of GCC and other information is displayed on the screen. If you get a `Command not found` error, GCC isn't installed; you have to update your operating system to include GCC as well as all the C programming libraries and other materials. (You can generally do that through your operating system's setup or configuration program; it doesn't typically require that the entire operating system be reinstalled.)

Making Programs

To build programs, you need two tools: an editor and a compiler. You use the editor to create or edit the source code — which is merely a text file. Then, you use the compiler to magically transform that text into the language the computer understands, stuffing it all into a program file.

This book illustrates programming techniques by using small programs targeted to showcase specific examples of the C language. Because of that, you can use the command prompt for compiling programs more easily than the IDE that may have come with your compiler. I recommend that you become familiar with the command prompt.

The following steps don't apply to programming on the Macintosh before OS X. If you're using an older Mac, refer to your compiler's documentation to find out how to edit and compile programs. Remember to use the learn folder you created to save all your stuff.

Finding your `learn` directory or folder

The first step to programming is to navigate your way to the `learn` directory (or folder) by using the command prompt. Follow these steps:

1. **Start a terminal or command-prompt window.**

   In Windows, run the CMD.EXE program, also known as the MS-DOS prompt.

   This program is on the Accessories or, often, main Programs menu, off the Start button. Or, you can type **CMD** in the Run dialog box to start the command-prompt window.

   In Linux, OS X, FreeBSD, and other Unix-like operating systems, open a terminal window if you're using a graphical shell. Otherwise, any terminal works.

2. **Change to your home directory.**

   In Windows XP, type this command:

   ```
   cd "my documents"
   ```

   In other versions of Windows, type this command:

   ```
   cd "\My Documents"
   ```

My favorite editor for working with C is vim, a variant on the infamous vi editor in Unix (see Figure A-1). Unlike vi, vim uses colors to code text. When you edit your source code in vim, you see keywords, values, and other parts of the C language highlighted in color.

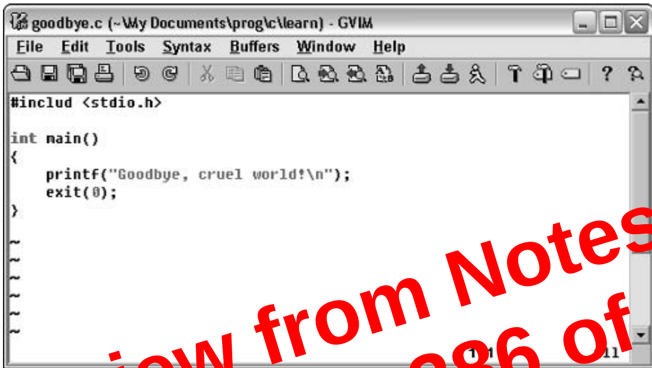


Figure A-1: The vim editor.

TIP

- Versions of vim are available for Linux, FreeBSD, Mac OS X, Windows, and even older Macs. You can pick it up at `www.vim.org`.
- Windows XP may not like the `EDIT` command. As an alternative, you can use Notepad to edit your source code. For example, to edit the `GOODBYE.C` text file, you type this command at the prompt:

  ```
  NOTEPAD GOODBYE.C
  ```

  Notepad opens in another window, where you can edit the text file. Simply close the window when you're done.

Compiling and linking

After the source-code text file is created, your next step is to compile and link. This step transforms the meek and mild text file into a robust and useable program on your computer.

Read the proper subsection for compiling and linking specifics for your operating system. For Macs before OS X, see the reference material that came with your compiler.

Making GCC work in Windows

Heck, for all the advances made with Windows, you may as well be using DOS when it comes to compiling programs at the command prompt. Anyway. . . .

1. **Ensure that you're in the `learn` folder.**

   Heed the steps in the section "Finding your `learn` directory or folder," earlier in this appendix.

2. **Use your text editor to create your source code file.**

   Use vi, ee, or whatever your favorite text editor is to create and save the source code file. For an example, you can refer to the listing of the GOODBYE.C source in Chapter 1; type that text into your editor.

3. **Compile and link the source code.**

   Compiling and linking are both handled by the GCC command. As an example, here's what you need to type to compile and link the GOODBYE.C source code created in Step 1:

   ```
   gcc goodbye.c -o goodbye
   ```

   The code has four items:

   - `gcc`, the command to compile and link the source code
   - `goodbye.c`, the name of the source code file
   - `-o`, the output switch
   - `goodbye`, the name of the final program

   If you leave off the `-o` switch and its option, GCC creates the program file named `a.out`. I don't recommend this. Instead, remember the `-o` option and specify a name for the output program. The name can be the same as the source code file, but without the `.c` extension.

4. **Run the program.**

   Alas, your operating system doesn't run your program if you type its name at the prompt. That's because Unix runs only programs found on the path, and I don't recommend putting your `learn` directory on the path. (If you create your own programs that you want to run, copy them to a `bin` directory beneath your home directory, and put *that* directory on the path.)

   To get the operating system to notice your program, you have to be specific about where the program lives (in the current folder, for example). You do that by prefixing `./` to the program's name. To run the `goodbye` program, type the following at the prompt:

   ```
   ./goodbye
   ```

   And the program runs.

Those steps are the basic ones you take (all in the `learn` folder) to create the program examples in this book. As I have said, it eventually becomes second nature to you.

Code	*Character*	*Hex*	*Binary*	*Notes*
20	^T	14	0001 0100	
21	^U	15	0001 0101	
22	^V	16	0001 0110	
23	^W	17	0001 0111	
24	^X	18	0001 1000	
25	^Y	19	0001 1001	
26	^Z	1A	0001 1010	End of file (DOS)
27	^[	1B	0001 1011	Escape
28	^\	1C	0001 1100	
29	^]	1D	0001 1101	
30	^^	1E	0001 1110	
31	^_	1F	0001 1111	
32		20	0010 0000	Space
33	!	21	0010 0001	
34	"	22	0010 0010	
35	#	23	0010 0011	
36	$	24	0010 0100	
37	%	25	0010 0101	
38	&	26	0010 0110	
39	'	27	0010 0111	
40	(	28	0010 1000	
41	)	29	0010 1001	
42	*	2A	0010 1010	
43	+	2B	0010 1011	
44	,	2C	0010 1100	
45	-	2D	0010 1101	
46	.	2E	0010 1110	