Chapter 4. Basic Declarations and Expressions

A journey of a thousand miles must begin with a single step.

—Lao-zi

If carpenters made buildings the way programmers make programs, the first woodpecker to come along would destroy all of civilization.

—Weinberg's Second Law

4.1 Elements of a Program

If you are going to construct a building, you need two things: the bricks and a blueprint that tells you how to put them together. In computer programming, you need two things: data (variables) and instructions (code or functions). Variables are the basic building blocks of a program. Instructions tell the computer what to do with the variables.

Comments are used to describe the variables and instructions. They are notes by the author documenting the program so that the program is clear and easy to read. Comments are ignored by the computer.

In construction, before we can start, we must order our materials: "We need 500 large bricks, 80 half-size bricks, and 4 flagstones." Similarly, in C, we must declare our variables before we can use them. We must name each one of our "bricks" and tell C what type of brick to use.

After our variables are defined, we can begin to use them. In construction, the basic structure is a room. By combining many rooms, we form a building. In C, the basic structure is a function. Functions can be combined to form a program.

An apprentice builder does not start out building the Empire State Building, but rather starts on a one-room house. In this chapter, we will concentrate on constructing simple one-function programs.

4.2 Basic Program Structure

The basic elements of a program are the data declarations, functions, and comments. Let's see how these can be organized into a simple C program.

The basic structure of a one-function program is:

```
/****************************************************
 * ...Heading comments...                           *
 ****************************************************/
...Data declarations...
int main()
{
    ...Executable statements...
    return (0);
}
```

Heading comments tell the programmer about the program, and *data declarations* describe the data that the program is going to use.

Our single function is named main. The name main is special, because it is the first function called. Other functions are called directly or indirectly from main. The function main begins with:

```
int main()
{
```

and ends with:

```
return (0);
}
```

The line return(0); is used to tell the operating system (UNIX or MS-DOS/Windows) that the program exited normally (Status=0). A nonzero status indicates an error—the bigger the return value, the more severe the error. Typically, a status of 1 is used for the most simple errors, like a missing file or bad command-line syntax.

Now, let's take a look at our Hello World program (Example 3-1).

At the beginning of the program is a comment box enclosed in /* and */. Following this box is the line:

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

This statement signals C that we are going to use the standard I/O package. The statement is a type of data declaration.[1] Later we use the function printf from this package.

[1] Technically, the statement causes a set of data declarations to be taken from an include file. Chapter 10, discusses include files.

Our main routine contains the instruction:

```
printf("Hello World\n");
```

This line is an executable statement instructing C to print the message "Hello World" on the screen. C uses a semicolon (;) to end a statement in much the same way we use a period to end a sentence. Unlike line-oriented languages such as BASIC, an end-of-line does not end a statement. The sentences in this book can span several lines—the end of a line is treated just like space between words. C works the same way. A single statement can span several lines. Similarly, you can put several sentences on the same line, just as you can put several C statements on the same line. However, most of the time your program is more readable if each statement starts on a separate line.

The standard function `printf` is used to output our message. A library routine is a C procedure or function that has been written and put into a library or collection of useful functions. Such routines perform sorting, input, output, mathematical functions, and file manipulation. See your C reference manual for a complete list of library functions.

Hello World is one of the simplest C programs. It contains no computations; it merely sends a single message to the screen. It is a starting point. After you have mastered this simple program, you have done a number of things correctly.

4.3 Simple Expressions

Computers can do more than just print strings—they can also perform calculations. Expressions are used to specify simple computations. The five simple operators in C are listed in Table 4-1.

Table 4-1. Simple Operators

Operator	Meaning
*	Multiply
/	Divide
+	Add
-	Subtract
%	Modulus (return the remainder after division)

Multiply (*), divide (/), and modulus (%) have precedence over add (+) and subtract (-). Parentheses, ( ), may be used to group terms. Thus:

```
(1 + 2) * 4
```

yields 12, while:

```
1 + 2 * 4
```

yields 9.

Example 4-1 computes the value of the expression `(1 + 2) * 4`.

Example 4-1. *simple/simple.c*

```
int main()
{
    (1 + 2) * 4;
    return (0);
}
```

Although we calculate the answer, we don't do anything with it. (This program will generate a "null effect" warning to indicate that there is a correctly written, but useless, statement in the program.)

Think about how confused a workman would be if we were constructing a building and said,

"Take your wheelbarrow and go back and forth between the truck and the building site."

"Do you want me to carry bricks in it?"

"No. Just go back and forth."

We need to store the results of our calculations.

4.4 Variables and Storage

C allows us to store values in *variables* . Each variable is identified by a *variable name*.

In addition, each variable has a *variable type*. The type tells C how the variable is going to be used and what kind of numbers (real, integer) it can hold. Names start with a letter or underscore (_), followed by any number of letters, digits, or underscores. Uppercase is different from lowercase, so the names `sam`, `Sam`, and `SAM` specify three different variables. However, to avoid confusion, you should use different names for variables and not depend on case differences.

Nothing prevents you from creating a name beginning with an underscore; however, such names are usually reserved for internal and system names.

Most C programmers use all-lowercase variable names. Some names like **int**, **while**, **for**, and **float** have a special meaning to C and are considered *reserved words*. They cannot be used for variable names.

The following is an example of some variable names:

```
average           /* average of all grades */
pi               /* pi to 6 decimal places */
number_of_students /* number students in this class */
```

The following are *not* variable names:

```
3rd_entry   /* Begins with a number */
all$done    /* Contains a "$" */
the end     /* Contains a space */
int        /* Reserved word */
```

Avoid variable names that are similar. For example, the following illustrates a poor choice of variable names:

```
total      /* total number of items in current entry */
totals     /* total of all entries */
```

A much better set of names is:

```
entry_total /* total number of items in current entry */
all_total   /* total of all entries */
```

4.5 Variable Declarations

Before you can use a variable in C, it must be defined in a *declaration statement*.

A variable declaration serves three purposes:

1. It defines the name of the variable.
2. It defines the type of the variable (integer, real, character, etc.).
3. It gives the programmer a description of the variable. The declaration of a variable answer can be:

```
    int answer;     /* the result of our expression */
```

The keyword **int** tells C that this variable contains an integer value. (Integers are defined below.) The variable name is `answer`. The semicolon (`;`) marks the end of the statement, and the comment is used to define this variable for the programmer. (The requirement that every C variable declaration be commented is a style rule. C will allow you to omit the comment. Any experienced teacher, manager, or lead engineer will not.)

The general form of a variable declaration is:

```
type  name;   /* comment */
```

where *type* is one of the C variable types (**int**, **float**, etc.) and *name* is any valid variable name. This declaration explains what the variable is and what it will be used for. (In Chapter 9, we will see how local variables can be declared elsewhere.)

Variable declarations appear just before the `main()` line at the top of a program.

4.6 Integers

One variable type is integer. Integer numbers have no fractional part or decimal point. Numbers such as 1, 87, and -222 are integers. The number 8.3 is not an integer because it contains a decimal point. The general form of an integer declaration is:

```
int  name;   /* comment */
```

A calculator with an 8-digit display can only handle numbers between 99999999 and -99999999. If you try to add 1 to 99999999, you will get an overflow error. Computers have similar limits. The limits on integers are implementation dependent, meaning they change from computer to computer.

Calculators use decimal digits (0-9). Computers use binary digits (0-1) called bits. Eight bits make a byte. The number of bits used to hold an integer varies from machine to machine. Numbers are converted from binary to decimal for printing.

On most UNIX machines, integers are 32 bits (4 bytes), providing a range of 2147483647 (2^{31}-1) to -2147483648. On the PC, most compilers use only 16 bits (2 bytes), so the range is 32767 (2^{15}-1) to -32768. These sizes are typical. The standard header file *limits.h* defines constants for the various numerical limits. (See Chapter 18, for more information on header files.)

The C standard does not specify the actual size of numbers. Programs that depend on an integer being a specific size (say 32 bits) frequently fail when moved to another machine.

Question 4-1: *The following will work on a UNIX machine, but will fail on a PC*:

```
int zip;      /* zip code for current address */
.........
zip = 92126;
```

Why does this fail? What will be the result when it is run on a PC? (Click here for the answer Section 4.12)

4.7 Assignment Statements

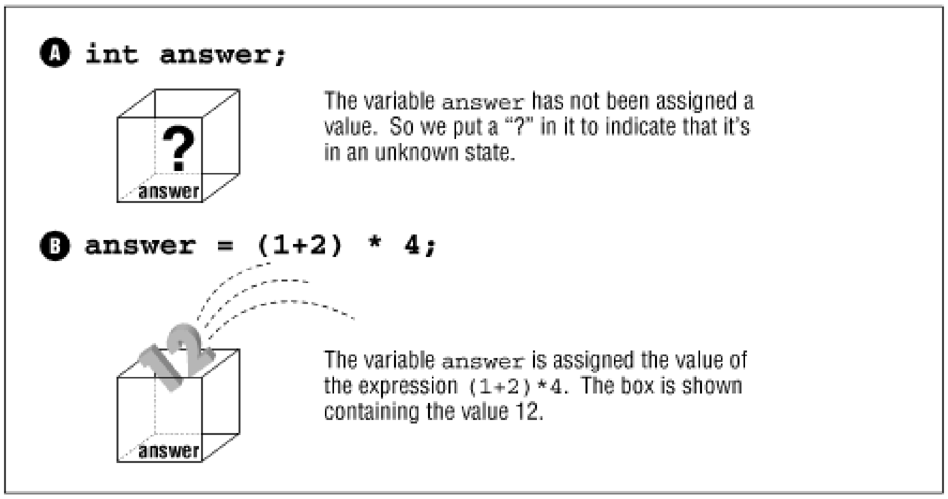
Variables are given a value through the use of assignment statements. For example:

```
answer = (1 + 2) * 4;
```

is an assignment. The variable `answer` on the left side of the equal sign (=) is assigned the value of the expression `(1 + 2) * 4` on the right side. The semicolon (`;`) ends the statement.

Declarations create space for variables. Figure 4-1A illustrates a variable declaration for the variable `answer`. We have not yet assigned it a value so it is known as an *uninitialized variable*. The question mark indicates that the value of this variable is unknown.

Figure 4-1. Declaration of answer and assigning it a value


A int answer;
?
answer
The variable answer has not been assigned a value. So we put a "?" in it to indicate that it's in an unknown state.
B answer = (1+2) * 4;
12
answer
The variable answer is assigned the value of the expression (1+2)*4. The box is shown containing the value 12.

Assignment statements are used to give a variable a value. For example:

```
answer = (1 + 2) * 4;
```

is an assignment. The variable `answer` on the left side of the equals operator (=) is assigned the value of the expression `(1 + 2) * 4`. So the variable `answer` gets the value 12 as illustrated in Figure 4-1B.

The general form of the assignment statement is:

`variable` `=` *`expression`*`;`

The = is used for assignment. It literally means: Compute the expression and assign the value of that expression to the variable. (In some other languages, such as PASCAL, the = operator is used to test for equality. In C, the operator is used for assignment.)

In Example 4-2, we use the variable `term` to store an integer value that is used in two later expressions.

Example 4-2. *term/term.c*

```
[File: term/term.c]
int term;       /* term used in two expressions */
int term_2;     /* twice term */
int term_3;     /* three times term */
int main()
{
    term = 3 * 5;
    term_2 = 2 * term;
    term_3 = 3 * term;
    return (0);
}
```

A problem exists with this program. How can we tell if it is working or not? We need some way of printing the answers.

4.8 printf Function

The library function `printf` can be used to print the results. If we add the statement:

```
printf("Twice %d is %d\n", term, 2 * term);
```

the program will print:

```
Twice 15 is 30
```

The special characters `%d` are called the *integer conversion specification* . When `printf` encounters a `%d`, it prints the value of the next expression in the list following the format string. This is called the *parameter list* .

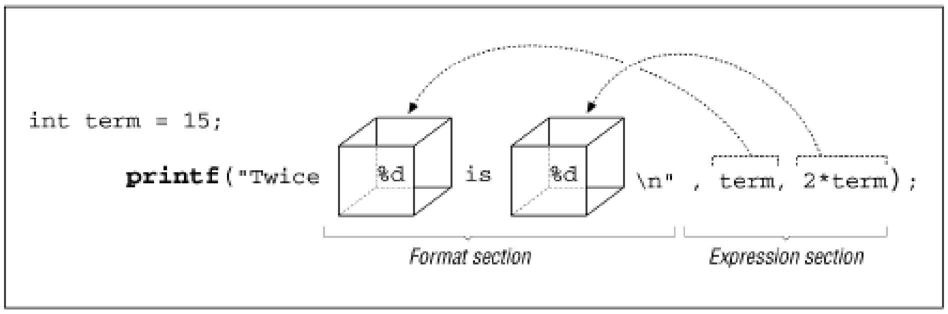
The general form of the `printf` statement is:

printf(*format*, *expression-1*, *expression-2*, ...);

where *format* is the string describing what to print. Everything inside this string is printed verbatim except for the `%d` conversions. The value of *expression-1* is printed in place of the first `%d`, *expression -2* is printed in place of the second, and so o n.

Figure 4 -2 shows how the elements of the `printf` statement work together to generate the final result.

Figure 4-2. printf structure



The format string "Twice %d is %d\n" tells printf to display Twice followed by a space, the value of the first expression, then a space followed by is and a space, the value of the second expression, finishing with an end-of-line (indicated by \n).

Example 4-3 shows a program that computes term and prints it via two printf functions.

Example 4-3. *twice/twice.c*

```
[File: twice/twice.c]
#include <stdio.h>

int term;       /* term used in two expressions */
int main()
{
    term = 3 * 5;
    printf("Twice %d is %d\n", term, 2*term);
    printf("Three times %d is %d\n", term, 3*term);
    return (0);
}
```

The number of %d conversions in the format should exactly match the number of expressions in the printf. C will not verify this. (Actually, the GNU gcc compiler will check printf arguments, if you turn on the proper warnings.) If too many expressions are supplied, the extra ones will be ignored. If there are not enough expressions, C will generate strange numbers for the missing expressions.

4.9 Floating Point

Because of the way they are stored internally, real numbers are also known as floating-point numbers. The numbers 5.5 , 8.3, and -12.6 are all floating-point numbers. C uses the decimal point to distinguish between floating-point numbers and integers. So 5.0 is a floating-point number, while 5 is an integer. Floating-point numbers must contain a decimal point. Floating-point numbers include: 3.14159, 0.5, 1.0, and 8.88.

Although you could omit digits before the decimal point and specify a number as .5 instead of 0.5, the extra clearly indicates that you are using a floating-point number. A similar rule applies to 12. versus 12.0. A floating-point zero should be written as 0.0.

Additionally, the number may include an exponent specification of the form:

`e +` *`exp`*

For example, 1.2e34 is a shorthand version of 1.2 x 10^{34}.

The form of a floating-point declaration is:

`float` *`variable`*`;` *`/* comment */`*

Again, there is a limit on the range of floating-point numbers that the computer can handle. This limit varies widely from computer to computer. Floating-point accuracy will be discussed further in Chapter 16.

When a floating-point number using `printf` is written, the `%f` conversion is used. To print the expression `1.0/3.0`, we use this statement:

```
printf("The answer is %f\n", 1.0/3.0);
```

4.10 Floating Point Versus Integer Divide

The division operator is special. There is a vast difference between an integer divide and a floating-point divide. In an integer divide, the result is truncated (any fractional part is discarded). So the value of 19/10 is 1.

If either the divisor or the dividend is a floating-point number, a floating-point divide is executed. So 19.0/10.0 is 1.9. (19/10.0 and 19.0/10 are also floating-point divides; however, 19.0/10.0 is preferred for clarity.) Several examples appear in Table 4 -2.

Table 4-2. Expression Examples

Expression	Result	Result type
1 + 2	3	Integer
1.0 + 2.0	3.0	Floating Point
19 / 10	1	Integer
19.0 / 10.0	1.9	Floating Point

C allows the assignment of an integer expression to a floating-point variable. C will automatically perform the conversion from integer to floating point. A similar conversion is performed when a floating-point number is assigned to an integer. For example:

```
int   integer;  /* an integer */
float floating; /* a floating-point number */

int main()
{
    floating = 1.0 / 2.0;         /* assign "floating" 0.5 */
    integer = 1 / 3;              /* assign integer 0 */
    floating = (1 / 2) + (1 / 2); /* assign floating 0.0 */
    floating = 3.0 / 2.0;         /* assign floating 1.5 */
    integer = floating;           /* assign integer 1 */
    return (0);
}
```

Notice that the expression `1 / 2` is an integer expression resulting in an integer divide and an integer result of `0`.

Question 4-2: *Why is the result of* Example 4-4 *0.0"? What must be done to this program to fix it?* (Click here for the answer Section 4.12)

Example 4-4. *q_zero/q_zero.c*

```
#include <stdio.h>

float answer;   /* The result of our calculation */

int main()
{
    answer = 1/3;
    printf("The value of 1/3 is %f\n", answer);
    return (0);
}
```

Question 4-3: *Why does 2 + 2 = 5928? (Your results may vary. See Example 4-5.)* (Click here for the answer Section 4.12)

Example 4-5. *two/two.c*

```
[File: two/two.c]
#include <stdio.h>

/* Variable for computation results */
int answer;

int main()
{
    answer = 2 + 2;

    printf("The answer is %d\n");
    return (0);
}
```

Question 4-4: *Why is an incorrect result printed? (See Example 4-6.)* (Click here for the answer Section 4.12)

Example 4-6. *div/div.c*

```
[File: div/div.c]
#include <stdio.h>

float result;   /* Result of the divide */

int main()
{
    result = 7.0 / 22.0;

    printf("The result is %d\n", result);
    return (0);
}
```

4.11 Characters

The type **char** represents single characters. The form of a character declaration is:

```
char   variable;   /* comment */
```

Characters are enclosed in single quotes (©). ©A©, ©a©, and ©!© are character constants. The backslash character (\) is called the *escape character*. It is used to signal that a special character follows. For example, the characters `\"` can be used to put a double quote inside a string. A single quote is represented by `\©`. `\n` is the newline character. It causes the output device to go to the beginning of the next line (similar to a return key on a typewriter). The characters `\\` are the backslash itself. Finally, characters can be specified by `\`*nnn*, where *nnn* is the octal code for the character. Table 4-3 summarizes these special characters. Appendix A contains a table of ASCII character codes.

Table 4-3. Special Characters

Character	Name	Meaning
`\b`	Backspace	Move the cursor to the left by one character
`\f`	Form Feed	Go to top of new page
`\n`	Newline	Go to next line
`\r`	Return	Go to beginning of current line
`\t`	Tab	Advance to next tab stop (eight column boundary)
`\©`	Apostrophe	Character ©
`\"`	Double quote	Character ".
`\\`	Backslash	Character \\.
`\`*nnn*		Character number *nnn* (octal)

While characters are enclosed in single quotes (©), a different data type, the string, is enclosed in double quotes ("). A good way to remember the difference between these two types of quotes is to remember that single characters are enclosed in single quotes. Strings can have any number of characters (including one), and they are enclosed in double quotes.

Characters use the `printf` conversion `%c`. Example 4-7 reverses three characters.

Example 4-7. *rev/rev.c*

```
[File: rev/rev.c]
#include <stdio.h>

char char1;     /* first character */
char char2;     /* second character */
char char3;     /* third character */

int main()
{
    char1 = 'A';
    char2 = 'B';
    char3 = 'C';
    printf("%c%c%c reversed is %c%c%c\n",
        char1, char2, char3,
        char3, char2, char1);
    return (0);
}
```

When executed, this program prints:

```
ABC reversed is CBA
```

4.12 Answers

Answer 4 -1 : The largest number that can be stored in an **int** on most UNIX machines is 2147483647. When using Turbo C++, the limit is 32767. The zip code 92126 is larger than 32767, so it is mangled, and the result is 26590.

This problem can be fixed by using a **long int** instead of just an **int**. The various types of integers will be discussed in Chapter 5.

Answer 4 -2 : The problem concerns the division: 1/3. The number 1 and the number 3 are both integers, so this question is an integer divide. Fractions are truncated in an integer divide. The expression should be written as:

```
answer = 1.0 / 3.0
```

Answer 4 -3 : The `printf` statement:

```
printf("The answer is %d\n");
```

tells the program to print a decimal number, but there is no variable specified. C does not check to make sure `printf` is given the right number of parameters. Because no value was specified, C makes one up. The proper `printf` statement should be:

```
printf("The answer is %d\n", answer);
```

Answer 4-4: The problem is that in the `printf` statement, we used a `%d` to specify that an integer was to be printed, but the parameter for this conversion was a floating-point number. The `printf` function has no way of checking its parameters for type. So if you give the function a floating-point number, but the format specifies an integer, the function will treat the number as an integer and print unexpected results.

4.13 Programming Exercises

Exercise 4-1: Write a program to print your name, social security number, and date of birth.

Exercise 4-2: Write a program to print a block E using asterisks (*), where the E has a height of seven characters and a width of five characters.

Exercise 4-3: Write a program to compute the area and perimeter of a rectangle with a width of three inches and a height of five inches. What changes must be made to the program so that it works for a rectangle with a width of 6.8 inches and a length of 2.3 inches?

Exercise 4-4: Write a program to print "HELLO" in big block letters; each letter should have a height of seven characters and width of five characters.

Exercise 4-5: Write a program that deliberately makes the following mistakes:

- Prints a floating-point number using the `%d` conversion.
- Prints an integer using the `%f` conversion.
- Prints a character using the `%d` conversion.