Rai Technology University

ENGINEERING MINDS

PROGRAMMING FUNDAMENTALS USING "C"

Subject: PROGRAMMING FUNDAMENTALS USING "C" **Credits: 4**

SYLLABUS

Introduction to „C' Language
Structures of 'C' Programming. Constants and Variables, C Tokens, Operators, Types of operators, Precedence and Associativity, Expression , Statement and types of statements

Built-in Operators and function
Console based I/O and related built-in I/O function, Concept of header files, Preprocessor directives, Decision Control structures, The if Statement, Use of Logical Operators, The else if Clause.

Loop Control structures
Nesting of loops , Case Control Structure, using Switch Introduction to problem solving, Problem solving techniques (Trial & Error, Brain storming, Divide & Conquer), Steps in problem solving (Define Problem, Analyze Problem, Explore Solution), Algorithms and Flowcharts (Definitions, Symbols), Characteristics of an algorithm

Simple Arithmetic Problems
Addition / Multiplication of integers, Functions, Basic types of function, Declaration and definition, Function call, Types of function, Introduction to Pointers, Pointer Notation, Recursion.

Storage Class
Automatic Storage Class , Register Storage Class, Static Storage Class, External Storage Class

Suggested Readings:

1. Mastering C by Venugopal, Prasad – TMH
2. Complete reference with C Tata McGraw Hill
3. C – programming E.Balagurusamy Tata McGray Hill
4. How to solve it by Computer : Dromey, PHI
5. Schaums outline of Theory and Problems of programming with C : Gottfried
6. The C programming language : Kerninghan and Ritchie
7. Programming in ANSI C : Ramkumar Agarwal
8. Mastering C by Venugopal, Prasad – TMH
9. Let Us C by Kanetkar

COURSE OVERVIEW

The computing world has undergone a revolution since the publication of The C Programming Language in 1978. Big computers are much bigger, and personal computers have capabilities that rival mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its origins as the language of the UNIX operating system.

The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce "an unambiguous and machine-independent definition of the language C", while still retaining its spirit. The result is the ANSI standard for C.

It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks.

It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent.

Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard.

C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all examples have been tested directly, which is in machine-readable form.

As we said in the preface to the C "wears well as one's experience with it grows". With a decade more experience, we still feel that way. We hope that this book will help you learn C and use it well.

PROGRAMMING FUNDAMENTALS USING 'C'

CONTENT

PROGRAMMING FUNDAMENTALS USING 'C'

CONTENT

LESSON 1
CONCEPTS OF DATA STORAGE WITHIN A COMPUTER PROGRAM

Objectives

- Know how the computer memory is organized.
- Know about the memory space required to store data
- Know what a variable is? And the different types of variables

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C has two ways of storing number values—variables and constants—with many options for each. A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that can't change.

Today you will learn:

- How to create variable names in C
- The use of different types of numeric variables
- The differences and similarities between character and numeric values
- How to declare and initialize numeric variables
- C's two types of numeric constants
- Before you get to variables, however, you need to know a little about the operation of your computer's memory.

Computer Memory

If you already know how a computer's memory operates, you can skip this section. If you're not sure, however, read on. This information will help you better understand certain aspects of C programming.

A computer uses random-access memory (RAM) to store information while it's operating. RAM is located in integrated circuits, or chips, inside your computer. RAM is volatile, which means that it is erased and replaced with new information as often as needed. Being volatile also means that RAM "remembers" only while the computer is turned on and loses its information when you turn the computer off.

Each computer has a certain amount of RAM installed. The amount of RAM in a system is usually specified in kilobytes (KB) or megabytes (MB), such as 512KB, 640KB, 2MB, 4MB, or 8MB. One kilobyte of memory consists of 1,024 bytes. Thus, a system with 640KB of memory actually has 640 * 1,024, or 65,536, bytes of RAM. One megabyte is 1,024 kilobytes. A machine with 4MB of RAM would have 4,096KB or 4,194,304 bytes of RAM.

The byte is the fundamental unit of computer data storage.

Memory space required to store data.

Data Bytes Required

The letter x 1
The number 500 2
The number 241.105 4

The phrase Teach Yourself C 17

One typewritten page Approximately 3,000

The RAM in your computer is organized sequentially, one byte following another. Each byte of memory has a unique address by which it is identified—an address that also distinguishes it from all other bytes in memory. Addresses are assigned to memory locations in order, starting at zero and increasing to the system limit. For now, you don't need to worry about addresses; it's all handled automatically by the C compiler.

What is your computer's RAM used for? It has several uses, but only data storage need concern you as a programmer. Data is the information with which your C program works. Whether your program is maintaining an address list, monitoring the stock market, keeping a household budget, or tracking the price of hog bellies, the information (names, stock prices, expense amounts, or hog futures) is kept in your computer's RAM while the program is running.

Now that you understand a little about the nuts and bolts of memory storage, you can get back to C programming and how C uses memory to store information.

Variables

A variable is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

Variable Names

To use variables in your C programs, you must know how to create variable names. In C, variable names must adhere to the following rules:

- The name can contain letters, digits, and the underscore character (_).
- The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended.
- Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables.
- C keywords can't be used as variable names. A keyword is a word that is part of the C language. (A complete list of 33 C keywords can be found in Appendix B, "Reserved Words.")

The following list contains some examples of legal and illegal C variable names:

Variable Name	Legality
Percent	Legal
y2x5__fg7h	Legal
annual_profit	Legal
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
double	Illegal: Is a C keyword
9winter	Illegal: First character is a digit

Because C is case-sensitive, the names percent, PERCENT, and Percent would be considered three different variables. C programmers commonly use only lowercase letters in variable names, although this isn't required.

Using all-uppercase letters is usually reserved for the names of constants.

For many compilers, a C variable name can be up to 31 characters long. (It can actually be longer than that, but the compiler looks at only the first 31 characters of the name.) With this flexibility, you can create variable names that reflect the data being stored. For example, a program that calculates loan payments could store the value of the prime interest rate in a variable named interest rate. The variable name helps make its usage clear. You could also have created a variable named x or even Johnny Carson; it doesn't matter to the C compiler. The use of the variable, however, wouldn't be nearly as clear to someone else looking at the source code. Although it might take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.

Many naming conventions are used for variable names created from multiple words.

You've seen one style:

interest_rate. Using an underscore to separate words in a variable name makes it easy to interpret. The second style is called camel notation. Instead of using spaces, the first letter of each word is capitalized. Instead of interest_rate, the variable would be named InterestRate. Camel notation is gaining popularity, because it's easier to type a capital letter than an underscore. We use the underscore in this book because it's easier for most people to read. You should decide which style you want to adopt.

- **DO** use variable names that are descriptive.
- **DO** adopt and stick with a style for naming your variables.
- **DON'T** start your variable names with an underscore unnecessarily.
- **DON'T** name your variables with all capital letters unnecessarily.

Numeric Variable Types

C provides several different types of numeric variables. You need different types of variables because different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them.

Small integers (for example, 1, 199, and -8) require less memory to store, and your computer can perform mathematical operations (addition, multiplication, and so on) with such numbers very quickly. In contrast, large integers and floating-point values (123,000,000 or 0.000000871256, for example) require more storage space and more time for mathematical operations. By using the appropriate variable types, you ensure that your program runs as efficiently as possible.

C's numeric variables fall into the following two main categories:

- Integer variables hold values that have no fractional part (that is, whole numbers only).
- Integer variables come in two flavors: signed integer variables can hold positive or negative values, whereas unsigned integer variables can hold only positive values (and 0).
- Floating-point variables hold values that have a fractional part (that is, real numbers).

Within each of these categories are two or more specific variable types. These are summarized in the table given below. It also shows the amount of memory, in bytes, required to hold a single variable of each type when you use a microcomputer with 16-bit architecture.

Table. C's numeric data types.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Integer	int	2	-32768 to 32767
Short integer	short	2	-32768 to 32767
Long integer	long	4	-7,483,648 to2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294, 967,295
Single-precision floating-point	Float	4	1.2E-38 to3.4E381
Double-precision f loating-point	Double	8	2.2E-308 to1.8E3082

1. Approximate range; precision = 7 digits
2. Approximate range; precision = 19 digits.

Approximate range means the highest and lowest values a given variable can hold. (Space limitations prohibit listing exact ranges for the values of these variables.)

Precision means the accuracy with which the variable is stored. (For example, if you evaluate 1/3, the answer is 0.33333... with 3s going to infinity. A variable with a precision of 7 stores seven 3s.)

Looking at Table, you might notice that the variable types int and short are identical. Why are two different types necessary? The int and short variable types are indeed identical on 16-bit IBM PC-compatible systems, but they might be different on other types of hardware. On a VAX system, a short and an int aren't the same size. Instead, a short is 2 bytes, whereas an int is 4. Remember that C is a flexible, portable language, so it provides different keywords for the two types. If you're working on a PC, you can use int and short interchangeably.

No special keyword is needed to make an integer variable signed; integer variables are signed by default. You can, however, include the signed keyword if you wish. The keywords

shown in Table are used in variable declarations, which are discussed in the next section.

There are five things you can count on:

- The size of a char is one byte.
- The size of a short is less than or equal to the size of an int.
- The size of an int is less than or equal to the size of a long. L
- The size of an unsigned is equal to the size of an int.

The size of a float is less than or equal to the size of a double.

Notes

LESSON 2
CONCEPT OF VARIABLE, CONSTANT AND PREPROCESSOR DIRECTIVE STATEMENTS

Objectives

- Know how to declare a variable.
- Know how to initialize a variable.
- Know what a constant is? And the different types of constants.
- Know about the pre-processor directive statements.

Variable Declarations

Before you can use a variable in a C program, it must be declared. A variable declaration tells the compiler the name and type of a variable and optionally initializes the variable to a specific value. If your program attempts to use a variable that hasn't been declared, the compiler generates an error message. A variable declaration has the following form:

typename varname;

typename specifies the variable type and must be one of the keywords listed in Table *varname* is the variable name, which must follow the rules mentioned earlier.

You can declare multiple variables of the same type on one line by separating the variable names with commas:

```
int count, number, start; /* three integer variables */
float percent, total; /* two float variables */
```

When we learn about "Understanding Variable Scope," you'll learn that the location of variable declarations in the source code is important, because it affects the ways in which your program can use the variables. For now, you can place all the variable declarations together just before the start of the main() function.

Excercise

1. Declare an integer called sum

   int sum;

2. Declare a character called letter

   char letter;

3. Define a constant called TRUE which has a value of 1

   #define TRUE 1

4. Declare a variable called money which can be used to hold currency

   float money;

5. Declare a variable called arctan which will hold scientific notation values (+e)

   double arctan;

6. Declare an integer variable called total and initialise it to zero.

   int total;

   total = 0;

7. Declare a variable called loop, which can hold an integer value.

   int loop;

8. Define a constant called GST with a value of .125

   #define GST 0.125

The typedef Keyword

The typedef keyword is used to create a new name for an existing data type. In effect, typedef creates a synonym. For example, the statement

typedef int integer;

creates integer as a synonym for int. You then can use integer to define variables of type int, as in this example:

integer count;

Note that typedef doesn't create a new data type; it only lets you use a different name for a predefined data type.

The most common use of typedef concerns aggregate data types, "Structures."

An aggregate data type consists of a combination of data types presented in this lesson.

Initializing Numeric Variables

When you declare a variable, you instruct the compiler to set aside storage space for the variable. However, the value stored in that space—the value of the variable—isn't defined. It might be zero, or it might be some random "garbage" value. Before using a variable, you should always initialize it to a known value. You can do this independently of the variable declaration by using an assignment statement, as in this example:

```
int count; /* Set aside storage space for count */
count = 0; /* Store 0 in count */
```

Note that this statement uses the equal sign (=), which is C's assignment operator. The one in programming is not the same as the equal sign in algebra. If you write x = 12in an algebraic statement, you are stating a fact:

"x equals 12."

In C, however, it means something quite different: "Assign the value 12 to the variable named x."

You can also initialize a variable when it's declared. To do so, follow the variable name in the declaration statement with an equal sign and the desired initial value:

```
int count = 0;
double percent = 0.01, taxrate = 28.5;
```

Be careful not to initialize a variable with a value outside the allowed range. Here are two examples of out-of-range initializations:

```
int weight = 100000;
unsigned int value = -2500;
```

The C compiler doesn't catch such errors. Your program might compile and link, but you might get unexpected results when the program is run.

- **DO** understand the number of bytes that variable types take for your computer.
- **DO** use typedef to make your programs more readable.
- **DO** initialize variables when you declare them whenever possible.
- **DON'T** use a variable that hasn't been initialized. Results can be unpredictable.
- **DON'T** use a float or double variable if you're only storing integers. Although they will work, using them is inefficient.
- **DON'T** try to put numbers into variable types that are too small to hold them.
- **DON'T** put negative numbers into variables with an unsigned type.

Now let us what are constants in C. Don't worry they are the same, as they are known by their name. You know it as something, which has fixed value, and it is the same here.

Constants

Like a variable, a *constant* is a data storage location used by your program. Unlike a variable, the value stored in a constant can't be changed during program execution. C has two types of constants, each with its own specific uses.

Literal Constants

A *literal constant* is a value that is typed directly into the source code wherever it is needed. Here are two examples:

int count = 20;

float tax_rate = 0.28;

The 20 and the 0.28 are literal constants. The preceding statements store these values in the variables count and tax_rate. Note that one of these constants contains a decimal point, whereas the other does not. The presence or absence of the decimal point distinguishes floating-point constants from integer constants.

A literal constant written with a decimal point is a floating-point constant and is represented by the C compiler as a double-precision number. Floating-point constants can be written in standard decimal notation, as shown in these examples:

123.456

0.019

100.

Note that the third constant, 100., is written with a decimal point even though it's an integer (that is, it has no fractional part). The decimal point causes the C compiler to treat the constant as a double-precision value.

Without the decimal point, it is treated as an integer constant. Floating-point constants also can be written in scientific notation. You might recall from high school mathematics that scientific notation represents a number as a decimal part multiplied by 10 to a positive or negative power.

Scientific notation is particularly useful for representing extremely large and extremely small values. In C, scientific notation is written as a decimal number followed immediately by an E or e and the exponent:

1.23E2 1.23 times 10 to the 2nd power, or 123

4.08e6 4.08 times 10 to the 6th power, or 4,080,000

0.85e-4 0.85 times 10 to the -4th power, or 0.000085

A constant written without a decimal point is represented by the compiler as an integer number. Integer constants can be written in three different notations:

A constant starting with any digit other than 0 is interpreted as a decimal integer (that is, the standard base-10 number system). Decimal constants can contain the digits 0 through 9 and a leading minus or plus sign. (Without a leading minus or plus, a constant is assumed to be positive.)

- A constant starting with the digit 0 is interpreted as an octal integer (the base-8 number system). Octal constants can contain the digits 0 through 7 and a leading minus or plus sign.
- A constant starting with 0x or 0X is interpreted as a hexadecimal constant (the base-16 number system).
- Hexadecimal constants can contain the digits 0 through 9, the letters A through F, and a leading minus or plus sign.

Now let us see what a symbolic constant is?

Symbolic Constants

A *symbolic constant* is a constant that is represented by a name (symbol) in your program. Like a literal constant, a symbolic constant can't change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it is first defined.

Symbolic constants have two significant advantages over literal constants, as the following example shows.

- Suppose that you're writing a program that performs a variety of geometrical calculations. The program frequently needs the value ,, (3.14159) for its calculations. (You might recall from geometry class that ,, is the ratio of a circle's circumference to its diameter.) For example, to calculate the circumference and area of a circle with a known radius, you could write

Circumference = 3.14159 * (2 * radius);

Area = 3.14159 * (radius)*(radius);

The asterisk (*) is C's multiplication operator. Thus, the first of these statements means

"Multiply 2 times the value stored in the variable radius, and then multiply the result by 3.14159. Finally, assign the result to the variable named circumference."

If, however, you define a symbolic constant with the name PI and the value 3.14, you could write

circumference = PI * (2 * radius);

area = PI * (radius)*(radius);

The resulting code is clearer. Rather than puzzling over what the value 3.14 is for, you can see immediately that the constant PI is being used.

- The second advantage of symbolic constants becomes apparent when you need to change a constant. Continuing with the preceding example, you might decide that for greater

accuracy your program needs to use a value of PIwith more decimal places: 3.14159 rather than 3.14. If you had used literal constants for PI, you would have to go through your source code and change each occurrence of the value from 3.14 to 3.14159. With a symbolic constant, you need to make a change only in the place where the constant is defined.

C has two methods for defining a symbolic constant:

the #define directive and the const keyword. The #define directive is one of C's preprocessor directives, and will be discussed later.

The #define directive is used as follows:

#define *CONSTNAME literal*

This creates a constant named *CONSTNAME* with the value of *literal*. *literal* represents a literal constant, as described earlier. *CONSTNAME* follows the same rules described earlier for variable names. By convention, the names of symbolic constants are uppercase. This makes them easy to distinguish from variable names, which by convention are lowercase. For the previous example, the required #define directive would be

#define PI 3.14159

Note that #define lines don't end with a semicolon (;). #defines can be placed anywhere in your source code, but they are in effect only for the portions of the source code that follow the #define directive. Most commonly, programmers group all #defines together, near the beginning of the file and before the start of main().

The C Preprocessor

C allows for commands to the compiler to be included in the source code. These commands are then called preprocessor commands and are defined by the ANSI standard to be;

#if
#ifdef
#ifndef
#else
#elif
#include
#define
#undef
#line
#error
#pragma

All preprocessor commands start with a hash symbol, "#", and must be on a line on their own (although comments may follow).

#define

The #define command specifies an identifier and a string that the compiler will substitute every time it comes accross the identifier within that source code module. For example;

#define FALSE 0

#define TRUE !FALSE

The compiler will replace any subsequent occurence of 'FALSE' with '0' and any subsequent occurence of 'TRUE' with '!0'. The substitution does NOT take place if the compiler finds that the identifier is enclosed by quotation marks, so

```
printf("TRUE");
```

would NOT be replaced, but

```
printf("%d",FALSE);
```

would be.

The #define command also can be used to define macros that may include parameters. The parameters are best enclosed in parenthesis to ensure that correct substitution occurs.

This example declares a macro 'larger()' that accepts two parameters and returns the larger of the two;

```
#include <stdio.h>
#define larger(a,b) (a > b) ? (a) : (b)
int main()
{
        printf("\n%d is largest",larger(5,7));
}
```

#error

The #error command causes the compiler to stop compilation and to display the text following the #error command. For example;

#error REACHED MODULE B

will cause the compiler to stop compilation and display;

REACHED MODULE B

#include

The #include command tells the compiler to read the contents of another source file. The name of the source file must be enclosed either by quotes or by angular brackets thus;

```
#include "module2.c"
#include <stdio.h>
```

Generally, if the file name is enclosed in angular brackets, then the compiler will search for the file in a directory defined in the compiler's setup. Whereas if the file name is enclosed in quotes then the compiler will look for the file in the current directory.

#if, #else, #elif, #endif

The #if set of commands provide conditional compilation around the general form;

```
#if constant_expression
        statements
#else
        statements
#endif
```

#elif stands for '#else if' and follows the form;

```
#if expression
        statements
#elif expression
        statements
#endif
```

#ifdef, #ifndef

These two commands stand for '#if defined' and '#if not defined' respectively and follow the general form;

```
#ifdef macro_name
        statements
#else
        statements
#endif
#ifndef macro_name
        statements
#else
        statements
#endif
```

where 'macro_name' is an identifier declared by a #define statement.

#undef

Undefines a macro previously defined by #define.

#line

Changes the compiler declared global variables __LINE__ and __FILE__. The general form of #line is;

```
#line number "filename"
```

where number is inserted into the variable '__LINE__' and 'filename' is assigned to '__FILE__'.

#pragma

This command is used to give compiler specific commands to the compiler. The compiler's manual should give you full details of any valid options to go with the particular implementation of #pragma that it supports.

How a #define Works

The precise action of the #define directive is to instruct the compiler as follows: "In the source code, replace *CONSTNAME* with *literal*." The effect is exactly the same as if you had used your editor to go through the source code and make the changes manually. Note that #define doesn't replace instances of its target that occur as parts of longer names, within double quotes, or as part of a program comment. For example, in the following code, the instances of PI in the second and third lines would not get changed:

```
#define PI 3.14159
/* You have defined a constant for PI. */
#define PIPETTE 100
```

Defining Constants with the const Keyword

The second way to define a symbolic constant is with the const keyword. const is a modifier that can be applied to any variable declaration. A variable declared to be const can't be modified during program execution—only initialized at the time of declaration.

Here are some examples:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

const affects all variables on the declaration line. In the last line, debt and tax_rate are symbolic constants. If your program tries to modify a const variable, the compiler generates an error message, as shown here:

```
const int count = 100;
count = 200; /* Does not compile! Cannot reassign or alter */
/* the value of a constant. */
```

What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword?

The differences have to do with pointers and variable scope. Pointers and variable scope will be covered later.

- **DO** use constants to make your programs easier to read.
- **DON'T** try to assign a value to a constant after it has already been initialized.

The **define** statement is used to make programs more readable. Consider the following examples,

```
#define TRUE    1    /* Don't use a semi-colon , # must be first character on line */
#define FALSE   0
#define NULL    0
#define AND     &
#define OR      |
#define EQUALS  ==
game_over = TRUE;
while( list_pointer != NULL )
        ..............
```

Note that preprocessor statements begin with a # symbol, and are NOT terminated by a semi-colon. Traditionally, preprocessor statements are listed at the beginning of the source file.

Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like TRUE) which occur in the C program are replaced by their value (like 1). Once this substitution has taken place by the preprocessor, the program is then compiled.

In general, preprocessor constants are written in **UPPER-CASE**.

Exercise

Use pre-processor statements to replace the following constants

```
0.312
W
37
```

Literal Substitution of Symbolic Constants Using #define

Lets now examine a few examples of using these symbolic constants in our programs. Consider the following program, which defines a constant called TAX_RATE.

```
#include <stdio.h>
#define TAX_RATE  0.10
main()
```

```
        {
                float balance;
                float tax;
                balance = 72.10;
                tax = balance * TAX_RATE;
printf("The tax on %.2f is %.2f\n", balance, tax );
        }
```

The pre-processor first replaces all symbolic constants before the program is compiled, so after preprocessing the file (and before its compiled), it now looks like,

```
        #include <stdio.h>
        #define TAX_RATE  0.10
        main()
        {
                float balance;
                float tax;
                balance = 72.10;
                tax = balance * 0.10;
printf("The tax on %.2f is %.2f\n", balance, tax );
        }
```

You Cannot Assign Values to the Symbolic Constants

Considering the above program as an example, look at the changes we have made below. We have added a statement, which tries to change the TAX_RATE to a new value.

```
        #include <stdio.h>
        #define TAX_RATE  0.10
        main()
        {
                float balance;
                float tax;
                balance = 72.10;
                TAX_RATE = 0.15;
                tax = balance * TAX_RATE;
printf("The tax on %.2f is %.2f\n", balance, tax );
        }
```

This is **illegal**. You cannot re-assign a new value to a symbolic constant.

Its Literal Substitution, So Beware of Errors

As shown above, the preprocessor performs literal substitution of symbolic constants. Lets modify the previous program slightly, and introduce an error to highlight a problem.

```
        #include <stdio.h>
        #define TAX_RATE  0.10;
        main()
        {
                float balance;
                float tax;
                balance = 72.10;
                tax = (balance * TAX_RATE )+ 10.02;
printf("The tax on %.2f is %.2f\n", balance, tax );
        }
```

In this case, the error that has been introduced is that the *#define* is terminated with a semi-colon. The preprocessor performs the substitution and the offending line (which is flagged as an error by the compiler) looks like

```
tax = (balance * 0.10; )+ 10.02;
```

However, you do not see the output of the preprocessor. If you are using TURBO C, you will only see

```
tax = (balance * TAX_RATE )+ 10.02;
```

flagged as an error, and this actually looks okay (but its not! after substitution takes place).

Making Programs Easy to Maintain by Using #define

The whole point of using *#define* in your programs is to make them easier to read and modify. Considering the above programs as examples, what changes would you need to make if the TAX_RATE was changed to 20%.

Obviously, the answer is once, where the *#define* statement which declares the symbolic constant and its value occurs. You would change it to read

```
        #define TAX_RATE = 0.20
```

Without the use of symbolic constants, you would hard code the value 0.20 in your program, and this might occur several times (or tens of times).

This would make changes difficult, because you would need to search and replace every occurrence in the program. However, as the programs get larger, **what would happen if you actually used the value 0.20 in a calculation that had nothing to do with the TAX_RATE!**

Summary of #define

- Allow the use of symbolic constants in programs
- In general, symbols are written in uppercase
- Are not terminated with a semi-colon
- Generally occur at the beginning of the file
- Each occurrence of the symbol is replaced by its value
- Makes programs readable and easy to maintain

Summary

This lesson explored numeric variables, which are used by a C program to store data during program execution. You've seen that there are two broad classes of numeric variables, integer and floating-point. Within each class are specific variable types. Which variable type—int, long, float, or double—you use for a specific application depends on the nature of the data to be stored in the variable. You've also seen that in a C program, you must declare a variable before it can be used. A variable declaration informs the compiler of the name and type of a variable.

This lesson also covered C's two constant types, literal and symbolic. Unlike variables, the value of a constant can't change

during program execution. You type literal constants into your source code whenever the value is needed. Symbolic constants are assigned a name that is used wherever the constant value is needed. Symbolic constants can be created with the #define directive or with the const keyword.

Q&A

Q. long int variables hold bigger numbers, so why not always use them instead of int variables?

A. A long int variable takes up more RAM than the smaller int. In smaller programs, this doesn't pose a problem. As programs get bigger, however, you should try to be efficient with the memory you use.

Q. What happens if I assign a number with a decimal to an integer?

A. You can assign a number with a decimal to an int variable. If you're using a constant variable, your compiler probably will give you a warning. The value assigned will have the decimal portion truncated. For example, if you assign 3.14 to an integer variable called pi, pi will only contain 3. The .14 will be chopped off and thrown away.

Q. What happens if I put a number into a type that isn't big enough to hold it?

A. Many compilers will allow this without signaling any errors. The number is wrapped to fit, however, and it isn't correct. For example, if you assign 32768 to a two-byte signed integer, the integer really contains the value -32768. If you assign the value 65535 to this integer, it really contains the value -1. Subtracting the maximum value that the field will hold generally gives you the value that will be stored.

Q. What happens if I put a negative number into an unsigned variable?

A. As the preceding answer indicated, your compiler might not signal any errors if you do this. The compiler does the same wrapping as if you assigned a number that was too big. For instance, if you assign -1 to an unsigned int variable that is two bytes long, the compiler will put the highest number possible in the variable (65535).

Q. What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword?

A. The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C programming and are covered on Days 9 and 12. For now, know that by using #define to create constants, you can make your programs much easier to read.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What's the difference between an integer variable and a floating-point variable?
2. Give two reasons for using a double-precision floating-point variable (type double) instead of a single-precision floating-point variable (type float).
3. What are five rules that the ANSI Standard states are always true when allocating size for variables?
4. What are the two advantages of using a symbolic constant instead of a literal constant?
5. Show two methods for defining a symbolic constant named MAXIMUM that has a value of 100.
6. What characters are allowed in C variable names?
7. What guidelines should you follow in creating names for variables and constants?
8. What's the difference between a symbolic and a literal constant?
9. What's the minimum value that a type int variable can hold?

Exercises

1. In what variable type would you best store the following values?
   - **a.** A person's age to the nearest year.
   - **b.** A person's weight in pounds.
   - **c.** The radius of a circle.
   - **d.** Your annual salary.
   - **e.** The cost of an item.
   - **f.** The highest grade on a test (assume it is always 100).
   - **g.** The temperature.
   - **h.** A person's net worth.
   - **i.** The distance to a star in miles.
2. Determine appropriate variable names for the values in exercise 1.
3. Write declarations for the variables in exercise 2.
4. Which of the following variable names are valid?
   - **a.** 123variable
   - **b.** x
   - **c.** total_score
   - **d.** Weight_in_#s
   - **e.** one.0
   - **f.** gross-cost
   - **g.** RADIUS
   - **h.** Radius
   - **i.** radius
   - **j.** this_is_a_variable_to_hold_the_width_of_a_box

LESSON 3
ELEMENTS OF LANGUAGE : EXPRESSIONS, STATEMENTS, OPERATORS

Objectives

- Know what a statement is?
- Know what an expression is? And more about expressions.
- Know what an operator is? And the different types of operators

C programs consist of statements, and most statements are composed of expressions and operators. You need to understand these three topics in order to be able to write C programs. Today you will learn

What a statement is?

What an expression is?

C's mathematical, relational, and logical operators!

What operator precedence is!

The if statement!

Statements

A statement is a complete direction instructing the computer to carry out some task. In C, statements are usually written one per line, although some statements span multiple lines. C statements always end with a semicolon

(Except for preprocessor directives such as #define and #include). You've already been introduced to some of C's statement types.

For example:

```
x = 2 + 3;
```

is an assignment statement. It instructs the computer to add 2 and 3 and to assign the result to the variable x. Other types of statements will be introduced as needed throughout this book.

Statements and White Space

The term *white space* refers to spaces, tabs, and blank lines in your source code. The C compiler isn't sensitive to white space. When the compiler reads a statement in your source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space. Thus, the statement

```
x=2+3;
```

is equivalent to this statement:

```
x = 2 + 3;
```

It is also equivalent to this:

```
x =
2
+
3;
```

This gives you a great deal of flexibility in formatting your source code. You shouldn't use formatting like the previous example, however. Statements should be entered one per line with a standardized scheme for spacing around variables and operators. As you become more experienced, you might discover that you prefer slight variations. The point is to keep your source code readable. However, the rule that C doesn't care about white space has one exception: Within literal string constants, tabs and spaces aren't ignored; they are considered part of the string. A *string* is a series of characters. Literal string constants are strings that are enclosed within quotes and interpreted literally by the compiler, space for space.

Although it's extremely bad form, the following is legal:

```
printf(
"Hello, world!"
);
```

This, however, is not legal:

```
printf("Hello,
world!");
```

To break a literal string constant line, you must use the backslash character (\) just before the break. Thus, the following is legal:

```
printf("Hello,world!");
```

Null Statements

If you place a semicolon by itself on a line, you create a *null statement*—a statement that doesn't perform any action. This is perfectly legal in C. Later, you will learn how the null statement can be useful.

Compound Statements

A compound statement, also called a *block*, is a group of two or more C statements enclosed in braces. Here's an example of a block:

```
{
printf("Hello, ");
printf("world!");
}
```

In C, a block can be used anywhere a single statement can be used. Note that the enclosing braces can be positioned in different ways. The following is equivalent to the preceding example:

```
{printf("Hello, ");
printf("world!");}
```

It's a good idea to place braces on their own lines, making the beginning and end of blocks clearly visible. Placing braces on their own lines also makes it easier to see whether you've left one out.

- **DO** stay consistent with how you use white space in statements.
- **DO** put block braces on their own lines. This makes the code easier to read.

- **DO** line up block braces so that it's easy to find the beginning and end of a block.
- **DON'T** spread a single statement across multiple lines if there's no need to do so. Limit statements to one line if possible.

Now let us see what an expression is and the different types of expressions?

Expressions

In C, an *expression* is anything that evaluates to a numeric value. C expressions come in all levels of complexity.

Simple Expressions

The simplest C expression consists of a single item: a simple variable, literal constant, or symbolic constant.

Here are four expressions:

Expression	*Description*
PI	A symbolic constant (defined in the program)
20	A literal constant
rate	A variable
-1.25	Another literal constant

A *literal constant* evaluates to its own value. A *symbolic constant* evaluates to the value it was given when you created it using the #define directive. A variable evaluates to the current value assigned to it by the program.

Complex Expressions

Complex expressions consist of simpler expressions connected by operators. For example:

2 + 8

is an expression consisting of the sub expressions 2 and 8 and the addition operator +.

The expression 2 + 8 evaluates, as you know, to 10. You can also write C expressions of great complexity:

1.25 / 8 + 5 * rate + rate * rate / cost

When an expression contains multiple operators, the evaluation of the expression depends on operator precedence. This concept is covered later in this lesson details about all of C's operators.

C expressions get even more interesting. Look at the following assignment statement:

x = a + 10;

This statement evaluates the expression a + 10 and assigns the result to x. In addition, the entire statement x = a + 10 is itself an expression that evaluates to the value of the variable on the left side of the equal sign.Thus, you can write statements such as the following, which assigns the value of the expression a + 10 to both variables, x and y:

y = x = a + 10;

In the given table below, an assignment statement is itself an expression.

You can also write statements such as this:

x = 6 + (y = 4 + 5);

The result of this statement is that y has the value 9 and x has the value 15. Note the parentheses, which are required in order for the statement to compile. We'll discuss the use of parentheses later.

Operators

An *operator* is a symbol that instructs C to perform some operation, or action, on one or more operands. An *operand* is something that an operator acts on. In C, all operands are expressions. C operators fall into several categories:

- The assignment operator
- Mathematical operators
- Relational operators
- Logical operators

The Assignment Operator

The *assignment operator* is the equal sign (=). Its use in programming is somewhat different from its use in regular math. If you write

x = y;

in a C program, it doesn't mean "x is equal to y." Instead, it means "assign the value of y to x." In a C assignment statement, the right side can be any expression, and the left side must be a variable name. Thus, the form is as follows:

variable = expression;

When executed, *expression* is evaluated, and the resulting value is assigned to *variable.*

Mathematical Operators

C's mathematical operators perform mathematical operations such as addition and subtraction. C has two unary mathematical operators and five binary mathematical operators.

Unary Mathematical Operators

The *unary* mathematical operators are so named because they take a single operand. C has two unary mathematical operators as listed in the given Table.

C's unary mathematical operators.

Operator	Symbol	Action	Examples
Increment	++	Increments the operand by one	++x, x++
Decrement	--	Decrements the operand by one	--x, x--

The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. In other words, the statements

++x;

—y;

are the equivalent of these statements:

x = x + 1;

y = y - 1;

We can see from Table that either unary operator can be placed before its operand (*prefix* mode) or after its operand (*postfix*

mode). These two modes are not equivalent. They differ in terms of when the increment or decrement is performed:

- When used in prefix mode, the increment and decrement operators modify their operand before it's used.
- When used in postfix mode, the increment and decrement operators modify their operand after it's used.

An example should make this clearer. Look at these two statements:

```
x = 10;
y = x++;
```

After these statements are executed, x has the value 11, and y has the value 10. The value of x was assigned to y, and then x was incremented. In contrast, the following statements result in both y and x having the value 11.

x is incremented, and then its value is assigned to y.

```
x = 10;
y = ++x;
```

Remember that = is the assignment operator, not a statement of equality. As an analogy, think of = as the "photocopy" operator. The statement y = x means to copy x into y. Subsequent changes to x, after the copy has been made, have no effect on y.

The following program illustrates the difference between prefix mode and postfix mode.

UNARY.C: Demonstrates prefix and postfix modes.

```
1: /* Demonstrates unary operator prefix and postfix modes
*/
2:
3: #include <stdio.h>
4:
5: int a, b;
6:
7: main()
8: {
9: /* Set a and b both equal to 5 */
10:
11: a = b = 5;
12:
13: /* Print them, decrementing each time. */
14: /* Use prefix mode for b, postfix mode for a */
15:
16: printf("\n%d %d", a—, —b);
17: printf("\n%d %d", a—, —b);
18: printf("\n%d %d", a—, —b);
19: printf("\n%d %d", a—, —b);
20: printf("\n%d %d\n", a—, —b);
21:
22: return 0;
23: }
```

Output

```
5 4
4 3
3 2
2 1
1 0
```

ANALYSIS: This program declares two variables, a and b, in line 5. In line 11, the variables are set to the value of 5. With the execution of each printf() statement (lines 16 through 20), both a and b are decremented by 1. After a is printed, it is decremented, whereas b is decremented before it is printed.

Notes

LESSON 4 Binary & Relational Operators

Objectives

- Know what is a binary mathematical operator?
- Know what is the operator precedence in 'C'
- Know how to evaluate an expression.
- Know more about relational operators.

Binary Mathematical Operators

C's binary operators take two operands. The binary operators, which include the common mathematical operations found on a calculator, are listed in the table.

C's Binary Mathematical Operators

Operator	Symbol	Action	Example
Addition	+	Adds two operands	x + y
Subtraction	-	Subtracts the second operand from the first operand	x - y
Multiplication	*	Multiplies two operands	x * y
Division	/	Divides the first operand by the second operand	x / y
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % y

The first four operators listed in the above table should be familiar to you, and you should have little trouble using them. The fifth operator, modulus, might be new. Modulus returns the remainder when the first operand is divided by the second operand.

For example,

11 modulus 4 equals 3 (that is, 4 goes into 11 two times with 3 left over).

Here are some more examples:

100 modulus 9 equals 1

10 modulus 5 equals 0

40 modulus 6 equals 4

Operator Precedence and Parentheses

In an expression that contains more than one operator, what is the order in which operations are performed?

The following assignment statement illustrates the importance of this question:

```
x = 4 + 5 * 3;
```

Performing the addition first results in the following, and x is assigned the value 27:

```
x = 9 * 3;
```

In contrast, if the multiplication is performed first, you have the following, and x is assigned the value 19:

```
x = 4 + 15;
```

Clearly, some rules are needed about the order in which operations are performed. This order, called *operator precedence*, is strictly spelled out in C. Each operator has a specific precedence.

When an expression is evaluated, operators with higher precedence are performed first. Table given below lists the precedence of C's mathematical operators. Number 1 is the highest precedence and thus is evaluated first.

The Precedence of C's Mathematical Operators

Operators	Relative Precedence
++ --	1
* / %	2
+ -	3

Looking at the above table, you can see that in any C expression, operations are performed in the following order:

- Unary increment and decrement
- Multiplication, division, and modulus
- Addition and subtraction

If an expression contains more than one operator with the same precedence level, the operators are performed in left-to-right order as they appear in the expression. For example, in the following expression, the % and * have the same precedence level, but the % is the leftmost operator, so it is performed first:

```
12 % 5 * 2
```

The expression evaluates to 4 (12 % 5 evaluates to 2; 2 times 2 is 4).

Returning to the previous example, you see that the statement x = 4 + 5 * 3; assigns the value 19 to x because the multiplication is performed before the addition.

What if the order of precedence doesn't evaluate your expression as needed? Using the previous example, what if you wanted to add 4 to 5 and then multiply the sum by 3?

C uses parentheses to modify the evaluation order.

A subexpression enclosed in parentheses is evaluated first, without regard to operator precedence. Thus, you could write

```
x = (4 + 5) * 3;
```

The expression 4 + 5 inside parentheses is evaluated first, so the value assigned to x is 27.

You can use multiple and nested parentheses in an expression. When parentheses are nested, evaluation proceeds from the innermost expression outward. Look at the following complex expression:

```
x = 25 - (2 * (10 + (8 / 2)));
```

The evaluation of this expression proceeds as follows:

1. The innermost expression, 8 / 2, is evaluated first, yielding the value 4:

25 - (2 * (10 + 4))

2. Moving outward, the next expression, 10 + 4, is evaluated, yielding the value 14:

25 - (2 * 14)

3. The last, or outermost, expression, 2 * 14, is evaluated, yielding the value 28:

25 – 28

4. The final expression, 25 - 28, is evaluated, assigning the value -3 to the variable x:

x = -3

You might want to use parentheses in some expressions for the sake of clarity, even when they aren't needed for modifying operator precedence. Parentheses must always be in pairs, or the compiler generates an error message.

Order of Subexpression Evaluation

As was mentioned in the previous section, if C expressions contain more than one operator with the same precedence level, they are evaluated left to right. For example, in the expression w is multiplied by x,

w * x / y * z

the result of the multiplication is then divided by y, and the result of the division is then multiplied by z. Across precedence levels, however, there is no guarantee of left-to-right order. Look at this expression:

w * x / y + z / y

Because of precedence, the multiplication and division are performed before the addition. However, C doesn't specify whether the Subexpression w * x / y is to be evaluated before or after z / y. It might not be clear to you why this matters. Look at another example:

w * x / ++y + z / y

If the left Subexpression is evaluated first, y is incremented when the second expression is evaluated. If the right expression is evaluated first, y isn't incremented, and the result is different. Therefore, you should avoid this sort of indeterminate expression in your programming.

- **DO** use parentheses to make the order of expression evaluation clear.
- **DON'T** overload an expression. It is often more clear to break an expression into two or more statements. This is especially true when you're using the unary operators (—) or (++).

Relational Operators

C's relational operators are used to compare expressions, asking questions such as,

"Is x greater than 100?" or "Is y equal to 0?"

An expression containing a relational operator evaluates to either true (1) or false (0). C's six relational operators are listed in the table. Table shows some examples of how relational operators might be used. These examples use literal constants, but the same principles hold with variables.

NOTE: "True" is considered the same as "yes," which is also considered the same as 1. "False" is considered the same as "no," which is considered the same as 0.

C's Relational Operators

Operator	Symbol	Question Asked	Example
Equal	==	Is operand 1 equal to operand 2?	x == y
Greater than	>	Is operand 1 greater than operand 2?	x > y
Less than	<	Is operand 1 less than operand 2?	x < y
Greater than or equal to	>=	Is operand 1 greater than or equal to operand 2?	x >= y
Less than or equal to	<=	Is operand 1 less than or equal to operand 2?	x <= y
Not equal	!=	Is operand 1 not equal to operand 2?	x != y

Relational Operators in Use.

Expression	How It Reads	What It Evaluates To
5 == 1	Is 5 equal to 1?	0 (false)
5 > 1	Is 5 greater than 1?	1 (true)
5 != 1	Is 5 not equal to 1?	1 (true)
(5 + 10) == (3 * 5)	Is (5 + 10) equal to (3 * 5)?	1 (true)

Expression How It Reads What It Evaluates To

5 == 1 Is 5 equal to 1? 0 (false)

5 > 1 Is 5 greater than 1? 1 (true)

5! = 1 Is 5 not equal to 1? 1 (true)

(5 + 10) == (3 * 5) Is (5 + 10) equal to (3 * 5)? 1 (true)

- **DO** learn how C interprets true and false. When working with relational operators, true is equal to 1, and false is equal to 0.
- **DON'T** confuse ==, the relational operator, with =, the assignment operator. This is one of the most common errors that C programmers make.

Notes

LESSON 5 Branching Constructs

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with branching statements?
- Know how a if statement works.
- Know how to evaluate a relational expression.

Let us see how we can use if statement for branching. Computers are programmed or we can say that the software is developed in such a way that they think like humans. But still they don't have the intelligence character that is present in humans. We humans still think that computers are superior to humans, but it is not so. These looping and branching statements whatever we are going to encounter in the forthcoming lessons are a result of this programming, which is made to make computers par to humans.

Branching Constructs

The if Statement

Relational operators are used mainly to construct the relational expressions used in if and while statements, "Basic Program Control." For now, I'll explain the basics of the if statement to show how relational operators are used to make *program control statements*. You might be wondering what a program control statement is. Statements in a C program normally execute from top to bottom, in the same order as they appear in your source code file. A program control statement modifies the order of statement execution. Program control statements can cause other program statements to execute multiple times or to not execute at all, depending on the circumstances. The if statement is one of C's program control statements. Others, such as do and while, are covered later. In its basic form, the if statement evaluates an expression and directs program execution depending on the result of that evaluation. The form of an if statement is as follows:

if (*expression*)

statement;

If *expression* evaluates to true, *statement* is executed. If *expression* evaluates to false, *statement* is not executed. In either case, execution then passes to whatever code follows the if statement. You could say that execution of *statement* depends on the result of *expression*. Note that both the line if (*expression*) and the line *statement*; are considered to comprise the complete if statement; they are not separate statements.

An if statement can control the execution of multiple statements through the use of a compound statement, or block. As defined earlier in this lesson, a block is a group of two or more statements enclosed in braces. A block can be used anywhere a single statement can be used. Therefore, you could write an if statement as follows:

if (*expression*)

{

statement1;

statement2;

/* additional code goes here */

statementn;

}

- **DO** remember that if you program too much in one day, you'll get C sick.
- **DO** indent statements within a block to make them easier to read. This includes the statements within a block in an if statement.
- **DON'T** make the mistake of putting a semicolon at the end of an if statement. An if statement should end with the conditional statement that follows it. In the following, *statement1* executes whether or not x equals 2, because each line is evaluated as a separate statement, not together as intended:

if(x == 2); /* semicolon does not belong! */

statement1;

In your programming, you will find that if statements are used most often with relational expressions; in other words, "Execute the following statement(s) only if such-and-such a condition is true." Here's an example:

```
if (x > y)
y = x;
```

This code assigns the value of x to y only if x is greater than y. If x is not greater than y, no assignment takes place. The program given below illustrates the use of if statements.

```
if( students < 65 )
        ++student_count;
```

In the above example, the variable *student_count* is incremented by one only if the value of the integer variable *students* is less than 65.

Program 1: Demonstrates if statements

```
1: /* Demonstrates the use of if statements */
2:
3: #include <stdio.h>
4:
5: int x, y;
6:
7: main()
8: {
9: /* Input the two values to be tested */
```

```
10:
11: printf("\nInput an integer value for x: ");
12: scanf("%d", &x);
13: printf("\nInput an integer value for y: ");
14: scanf("%d", &y);
15:
16: /* Test values and print result */
17:
18: if (x == y)
19: printf("x is equal to y\n");
20:
21: if (x > y)
22: printf("x is greater than y\n");
23:
24: if (x < y)
25: printf("x is smaller than y\n");
26:
27: return 0;
28: }
Input an integer value for x: 100
Input an integer value for y: 10
x is greater than y
Input an integer value for x: 10
Input an integer value for y: 100
x is smaller than y
Input an integer value for x: 10
Input an integer value for y: 10
x is equal to y
```

The above program shows three if statements in action (lines 18 through 25). Many of the lines in this program should be familiar. Line 5 declares two variables, x and y, and lines 11 through 14 prompt the user for values to be placed into these variables. Lines 18 through 25 use if statements to determine whether x is greater than, less than, or equal to y. Note that line 18 uses an if statement to see whether x is equal to y. Remember that ==, the equal operator, means "is equal to" and should not be confused with =, the assignment operator. After the program checks to see whether the variables are equal, in line 21 it checks to see whether x is greater than y, followed by a check in line 24 to see whether x is less than y. If you think this is inefficient, you're right. In the next program, you will see how to avoid this inefficiency. For now, run the program with different values for x and y to see the results.

NOTE: You will notice that the statements within an if clause are indented. This is a common practice to aid readability.

The else Clause

An if statement can optionally include an else clause. The else clause is included as follows:

```
if (expression)
statement1;
else
statement2;
```

If *expression* evaluates to true, *statement1* is executed. If *expression* evaluates to false, *statement2* is executed. Both *statement1* and *statement2* can be compound statements or blocks.

The program given below is rewritten to show how to use an if statement with an else clause.

Program 2: An if statement with an else clause.

```
1: /* Demonstrates the use of if statement with else clause */
2:
3: #include <stdio.h>
4:
5: int x, y;
6:
7: main()
8: {
9: /* Input the two values to be tested */
10:
11: printf("\nInput an integer value for x: ");
12: scanf("%d", &x);
13: printf("\nInput an integer value for y: ");
14: scanf("%d", &y);
15:
16: /* Test values and print result */
17:
18: if (x == y)
19: printf("x is equal to y\n");
20: else
21: if (x > y)
22: printf("x is greater than y\n");
23: else
24: printf("x is smaller than y\n");
25:
26: return 0;
27: }
Input an integer value for x: 99
Input an integer value for y: 8
x is greater than y
Input an integer value for x: 8
Input an integer value for y: 99
x is smaller than y
Input an integer value for x: 99
Input an integer value for y: 99
x is equal to y
```

ANALYSIS: Lines 18 through 24 are slightly different from the previous listing. Line 18 still checks to see whether x equals y. If x does equal y, x is equal to y appears on-screen, just as in Program 1. However, the program then ends, and lines 20

through 24 aren't executed. Line 21 is executed only if x is not equal to y , or, to be more accurate, if the expression "x equals y" is false. If x does not equal y, line 21 checks to see whether x is greater than y. If so, line 22 prints x is greater than y; otherwise (else), line 24 is executed.

Program 2 uses a nested if statement. Nesting means to place (nest) one or more C statements inside another C statement. In the case of Program 2, an if statement is part of the first if statement's else clause.

The if Statement

Form 1

if(*expression*)

statement1;

next_statement;

This is the if statement in its simplest form. If *expression* is true, *statement1* is executed. If *expression* is not true, *statement1* is ignored.

Form 2

if(*expression*)

statement1;

else

statement2;

next_statement;

This is the most common form of the if statement. If *expression* is true, *statement1* is executed; otherwise, *statement2* is executed.

Form 3

if(*expression1*)

statement1;

else if(*expression2*)

statement2;

else

statement3;

next_statement;

This is a nested if. If the first expression, *expression1*, is true, *statement1* is executed before the program continues with the *next_statement*. If the first expression is not true, the second expression, *expression2*, is checked. If the first expression is not true, and the second is true, *statement2* is executed. If both expressions are false, *statement3* is executed. Only one of the three statements is executed.

Example 1

```
if( salary > 45,0000 )
tax = .30;
else
tax = .25;
```

Example 2

```
if( age < 18 )
printf("Minor");
else if( age < 65 )
printf("Adult");
else
printf( "Senior Citizen");
```

Evaluating Relational Expressions

Remember that expressions using relational operators are true C expressions that evaluate, by definition, to a value. Relational expressions evaluate to a value of either false (0) or true (1). Although the most common use of relational expressions is within if statements and other conditional constructions, they can be used as purely numeric values. This is illustrated in the given below program.

Evaluating relational expressions.

```
1: /* Demonstrates the evaluation of relational expressions */
2:
3: #include <stdio.h>
4:
5: int a;
6:
7: main()
8: {
9: a = (5 == 5); /* Evaluates to 1 */
10: printf("\na = (5 == 5)\na = %d", a);
11:
12: a = (5 != 5); /* Evaluates to 0 */
13: printf("\na = (5 != 5)\na = %d", a);
14:
15: a = (12 == 12) + (5 != 1); /* Evaluates to 1 + 1 */
16: printf("\na = (12 == 12) + (5 != 1)\na = %d\n", a);
17: return 0;
18: }
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) + (5 != 1)
a = 2
```

ANALYSIS: The output from this listing might seem a little confusing at first. Remember, the most common mistake people make when using the relational operators is to use a single equal sign—the assignment operator—instead of a double equal sign. The following expression evaluates to 5 (and also assigns the value 5 to x): x = 5

In contrast, the following expression evaluates to either 0 or 1 (depending on whether x is equal to 5) and doesn't change the value of x: x == 5

If by mistake you write

```
if (x = 5)
printf("x is equal to 5");
```

the message always prints because the expression being tested by the if statement always evaluates to true, no matter what the original value of x happens to be.

Looking at the program, you can begin to understand why a takes on the values that it does. In line 9, the value 5 does equal 5, so true (1) is assigned to a. In line 12, the statement "5 does not equal 5" is false, so 0 is assigned to a.To reiterate, the relational operators are used to create relational expressions that ask questions about relationships between expressions. The answer returned by a relational expression is a numeric value of either 1 (representing true) or 0 (representing false).

Consider the following program which determines whether a character entered from the keyboard is within the range A to Z.

```
#include <stdio.h>
main()
{
        char letter;
        printf("Enter a character —>");
        scanf(" %c", &letter );
        if( letter >= 'A' ) {
                if( letter <= 'Z' )
                printf("The character is within A to
Z \n");
        }
}
```

Sample Program Output

```
Enter a character —> C
The character is within A to Z
```

The program does not print any output if the character entered is not within the range A to Z. This can be addressed on the following pages with the *if else* construct.

Please note use of the leading space in the statement (before %c)

```
scanf(" %c", &letter );
```

This enables the skipping of leading TABS, Spaces, (collectively called whitespaces) and the ENTER KEY. If the leading space was not used, then the first entered character would be used, and *scanf* would not ignore the whitespace characters.

Notes

LESSON 6 Precedence of Operators

Objectives

Upon completion of this Lesson, you should be able to:

- Know about the precedence of relational operators.
- Know about the various other operators.

The Precedence of Relational Operators

Like the mathematical operators, the relational operators each have a precedence that determines the order in which they are performed in a multiple-operator expression. Similarly, you can use parentheses to modify precedence in expressions that use relational operators. First, all the relational operators have a lower precedence than the mathematical operators. Thus, if you write the following, 2 is added to x, and the result is compared to y:

```
if (x + 2 > y)
```

This is the equivalent of the following line, which is a good example of using parentheses for the sake of clarity:

```
if ((x + 2) > y)
```

Although they aren't required by the C compiler, the parentheses surrounding (x + 2) make it clear that it is the sum of x and 2 that is to be compared with y. There is also a two-level precedence within the relational operators, as shown below.

The Order of Precedence of C's Relational Operators.

Operators	Relative Precedence
< <= > >=	1
!= ==	2

Thus, if you write

```
x == y > z
```

it is the same as

```
x == (y > z)
```

because C first evaluates the expression y > z, resulting in a value of 0 or 1. Next, C determines whether x is equal to the 1 or 0 obtained in the first step. You will rarely, if ever, use this sort of construction, but you should know about it.

- **DON'T** put assignment statements in if statements. This can be confusing to other people who look at your code. They might think it's a mistake and change your assignment to the logical equal statement.
- **DON'T** use the "not equal to" operator (!=) in an if statement containing an else. It's almost always clearer to use the "equal to" operator (==) with an else. For instance, the following code:

```
if ( x != 5 )
statement1;
else
statement2;
```

would be better written as this:

```
if (x == 5 )
statement2;
else
statement1;
```

Logical Operators

Sometimes you might need to ask more than one relational question at once. For example, "If it's 7:00 a.m. and a weekday and not my vacation, ring the alarm." C's logical operators let you combine two or more relational expressions into a single expression that evaluates to either true or false. The table given below lists C's three logical operators.

C's Logical Operators.

Operator	Symbol	Example
AND	&&	*exp1* && *exp2*
OR	||	*exp1* || *exp2*
NOT	!	!*exp1*

The way these logical operators work is explained below.

C's Logical Operators in Use.

Expression	What It Evaluates To
(*exp1* && *exp2*)	True (1) only if both *exp1* and *exp2* are true; false (0) otherwise
(*exp1* || *exp2*)	True (1) if either *exp1* or *exp2* is true; false (0) only if both are false
(!*exp1*)	False (0) if *exp1* is true; true (1) if *exp1* is false

You can see that expressions that use the logical operators evaluate to either true or false, depending on the true/false value of their operand(s). Table 4.9 shows some actual code examples.

Code examples of C's Logical

Expression	What It Evaluates To
(5 == 5) && (6 != 2)	True (1), because both operands are true
(5 > 1) || (6 < 1)	True (1), because one operand is true
(2 == 1) && (5 == 5)	False (0), because one operand is false
!(5 == 4)	True (1), because the operand is false

You can create expressions that use multiple logical operators. For example, to ask the question "Is x equal to 2, 3, or 4?" you would write

```
(x == 2) || (x == 3) || (x == 4)
```

The logical operators often provide more than one way to ask a question. If x is an integer variable, the preceding question also could be written in either of the following ways:

(x > 1) && (x < 5)

(x >= 2) && (x <= 4)

More on True/False Values

You've seen that C's relational expressions evaluate to 0 to represent false and to 1 to represent true. It's important to be aware, however, that any numeric value is interpreted as either true or false when it is used in a C expression or statement that is expecting a logical value (that is, a true or false value). The rules are as follows:

- A value of zero represents false.
- Any nonzero value represents true.

This is illustrated by the following example, in which the value of x is printed:

```
x = 125;
if (x)
printf("%d", x);
```

Because x has a nonzero value, the if statement interprets the expression (x) as true. You can further generalize this because, for any C expression, writing

(*expression*)

is equivalent to writing

(*expression* != 0)

Both evaluate to true if *expression* is nonzero and to false if *expression* is 0. Using the not (!) operator, you can also write

(!*expression*)

which is equivalent to

(*expression* == 0)

The Precedence of Operators

As you might have guessed, C's logical operators also have a precedence order, both among themselves and in relation to other operators. The ! operator has a precedence equal to the unary mathematical operators ++ and —. Thus, ! has a higher precedence than all the relational operators and all the binary mathematical operators.

In contrast, the && and || operators have much lower precedence, lower than all the mathematical and relational operators, although && has a higher precedence than ||. As with all of C's operators, parentheses can be used to modify the evaluation order when using the logical operators. Consider the following example:

You want to write a logical expression that makes three individual comparisons:

1. Is a less than b?

2. Is a less than c?

3. Is c less than d?

You want the entire logical expression to evaluate to true if condition 3 is true and if either condition 1 or condition 2 is true. You might write

a < b || a < c && c < d

However, this won't do what you intended. Because the && operator has higher precedence than ||, the expression is equivalent to

a < b || (a < c && c < d)

and evaluates to true if (a < b) is true, whether or not the relationships (a < c) and (c < d) are true. You need to write

(a < b || a < c) && c < d

which forces the || to be evaluated before the &&. The variables are set so that, if written correctly, the expression should evaluate to false (0).

Compound Assignment Operators

C's compound assignment operators provide a shorthand method for combining a binary mathematical operation with an assignment operation. For example, say you want to increase the value of x by 5, or, in other words, add 5 to x and assign the result to x. You could write

x = x + 5;

Using a compound assignment operator, which you can think of as a shorthand method of assignment, you would write

x += 5;

In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

exp1 op= *exp2*

This is equivalent to writing

exp1 = *exp1* op *exp2*;

You can create compound assignment operators using the five binary mathematical operators discussed earlier. The table given below lists some examples.

Examples of Compound Assignment Operators

When You Write This...	It Is Equivalent To This
x *= y	x = x * y
y -= z + 1	y = y - z + 1
a /= b	a = a / b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

The compound operators provide convenient shorthand, the advantages of which are particularly evident when the variable on the left side of the assignment operator has a long name. As with all other assignment statements, a compound assignment statement is an expression and evaluates to the value assigned to the left side. Thus, executing the following statements results in both x and z having the value 14:

```
x = 12;
z = x += 2;
```

The Conditional Operator

The conditional operator is C's only *ternary* operator, meaning that it takes three operands. Its syntax is

exp1 ? *exp2* : *exp3*;

If *exp1* evaluates to true (that is, nonzero), the entire expression evaluates to the value of *exp2*. If *exp1* evaluates to false (that is, zero), the entire expression evaluates as the value of *exp3*. For example, the following statement assigns the value 1 to x if y is true and assigns 100 to x if y is false:

```
x = y ? 1 : 100;
```

Likewise, to make z equal to the larger of x and y, you could write

```
z = (x > y) ? x : y;
```

Perhaps you've noticed that the conditional operator functions somewhat like an if statement. The preceding statement could also be written like this:

```
if (x > y)
z = x;
else
z = y;
```

The conditional operator can't be used in all situations in place of an if...else construction, but the conditional operator is more concise. The conditional operator can also be used in places you can't use an if statement, such as inside a single printf() statement:

```
printf( "The larger value is %d", ((x > y) ? x : y) );
```

The Comma Operator

The comma is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on. In certain situations, the comma acts as an operator rather than just as a separator. You can form an expression by separating two subexpressions with a comma. The result is as follows:

- Both expressions are evaluated, with the left expression being evaluated first.
- The entire expression evaluates to the value of the right expression.

For example, the following statement assigns the value of b to x, then increments a, and then increments b:

```
x = (a++ , b++);
```

Because the ++ operator is used in postfix mode, the value of b—before it is incremented—is assigned to x. Using parentheses is necessary because the comma operator has low precedence, even lower than the assignment operator. The most common use of the comma operator is in for statements.

- **DO** use (expression == 0) instead of (!expression). When compiled, these two expressions evaluate the same; however, the first is more readable.
- **DO** use the logical operators && and || instead of nesting if statements.
- **DON'T** confuse the assignment operator (=) with the equal to (==) operator.

Operator Precedence Revisited

The table given below lists all the C operators in order of decreasing precedence. Operators on the same line have the same precedence.

C Operator Precedence

Level	Operators
1	() [] -> .
2	! ~ ++ -- * (indirection) & (address-of) (type)
	sizeof + (unary) - (unary)
3	* (multiplication) / %
4	+ -
5	<< >>
6	< <= > >=
7	== !=
8	& (bitwise AND)
9	^
10	|
11	&&
12	||
13	?:
14	= += -= *= /= %= &= ^= |= <<= >>=
15	,

() is the function operator; [] is the array operator.

TIP: This is a good table to keep referring to until you become familiar with the order of precedence. You might find that you need it later.

Summary

This lesson covered a lot of material. You learned what a C statement is, that white space doesn't matter to a C compiler, and that statements always end with a semicolon. You also learned that a compound statement (or block), which consists of two or more statements enclosed in braces, can be used anywhere a single statement can be used. Many statements are made up of some combination of expressions and operators. Remember that an expression is anything that evaluates to a numeric value. Complex expressions can contain many simpler expressions, which are called sub expressions.

Operators are C symbols that instruct the computer to perform an operation on one or more expressions. Some operators are unary, which means that they operate on a single operand. Most of C's operators are binary, however, operating on two operands. One operator, the conditional operator, is ternary. C's operators have a defined hierarchy of precedence that determines the order in which operations are performed in an expression that contains multiple operators.

The C operators covered in this lesson fall into three categories:

- Mathematical operators perform arithmetic operations on their operands (for example, addition).
- Relational operators perform comparisons between their operands (for example, greater than).
- Logical operators operate on true/false expressions. Remember that C uses 0 and 1 to represent false and true, respectively, and that any nonzero value is interpreted as being true.

You've also been introduced to C's if statement, which lets you control program execution based on the evaluation of relational expressions.

Q&A

Q What effect do spaces and blank lines have on how a program runs?

A White space (lines, spaces, tabs) makes the code listing more readable. When the program is compiled, white space is stripped and thus has no effect on the executable program. For this reason, you should use white space to make your program easier to read.

Q Is it better to code a compound if statement or to nest multiple if statements?

A You should make your code easy to understand. If you nest if statements, they are evaluated as shownin this lesson. If you use a single compound statement, the expressions are evaluated only until the entire statement evaluates to false.

Q What is the difference between unary and binary operators?

A As the names imply, unary operators work with one variable, and binary operators work with two.

Q Is the subtraction operator (-) binary or unary?

A It's both! The compiler is smart enough to know which one you're using. It knows which form to use based on the number of variables in the expression that is used. In the following statement, it is unary: x = -y; versus the following binary use:

x = a - b;

Q Are negative numbers considered true or false?

A Remember that 0 is false, and any other value is true. This includes negative numbers.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the following C statement called, and what is its meaning?

x = 5 + 8;

2. What is an expression?

3. In an expression that contains multiple operators, what determines the order in which operations are performed?

4. If the variable x has the value 10, what are the values of x and a after each of the following statements is executed separately?

a = x++;

a = ++x;

5. To what value does the expression 10 % 3 evaluate?

6. To what value does the expression 5 + 3 * 8 / 2 + 2 evaluate?

7. Rewrite the expression in question 6, adding parentheses so that it evaluates to 16.

8. If an expression evaluates to false, what value does the expression have?

9. In the following list, which has higher precedence?

a. == or <

b. * or +

c. != or ==

d. >= or >

10. What are the compound assignment operators, and how are they useful?

Exercises

1. The following code is not well-written. Enter and compile it to see whether it works.

```
#include <stdio.h>
int x,y;main(){ printf(
"\nEnter two numbers");scanf(
"%d %d",&x,&y);printf(
"\n\n%d is bigger",(x>y)?x:y);return 0;}
```

2. Rewrite the code in exercise 1 to be more readable.

3. Change Listing 4.1 to count upward instead of downward.

4. Write an if statement that assigns the value of x to the variable y only if x is between 1 and 20. Leave y unchanged if x is not in that range.

5. Rewrite the following nested if statements using a single if statement and compound operators.

```
if (x < 1)
if ( x > 10 )
statement;
```

6. To what value do each of the following expressions evaluate?

a. (1 + 2 * 3)

b. 10 % 3 * 3 - (1 + 2)

c. ((1 + 2) * 3)

d. (5 == 5)

e. (x = 5)

7. If x = 4, y = 6, and z = 2, determine whether each of the following evaluates to true or false.

a. if(x == 4)

b. if(x != y - z)

c. if(z = 1)

d. if(y)

8. Write an if statement that determines whether someone is legally an adult (age 21), but not a senior citizen (age 65).

9. **BUG BUSTER:** Fix the following program so that it runs correctly.

```
/* a program with problems... */
#include <stdio.h>
int x= 1:
main()
{
if( x = 1);
printf(" x equals 1" );
otherwise
printf(" x does not equal 1");
return 0;
}
```

LESSON 7 Controlling Program Execution

Objectives

Upon completion of this Lesson, you should be able to:

- Know about controlling program execution
- Know about the for statement.
- Know about the while statement.

Controlling Program Execution

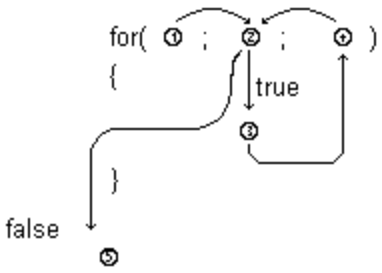
The default order of execution in a C program is top-down. Execution starts at the beginning of the main() function and progresses, statement by statement, until the end of main() is reached. However, this order is rarely encountered in real C programs. The C language includes a variety of program control statements that let you control the order of program execution. You have already learned how to use C's fundamental decision operator, the if statement, so let's explore three additional control statements you will find useful.

The for Statement

The for statement is a C programming construct that executes a block of one or more statements a certain number of times. It is sometimes called the for *loop* because program execution typically loops though the statement more than once. You've seen a few for statements used in programming examples earlier in this book.

Now you're ready to see how the for statement works.

A for statement has the following structure:

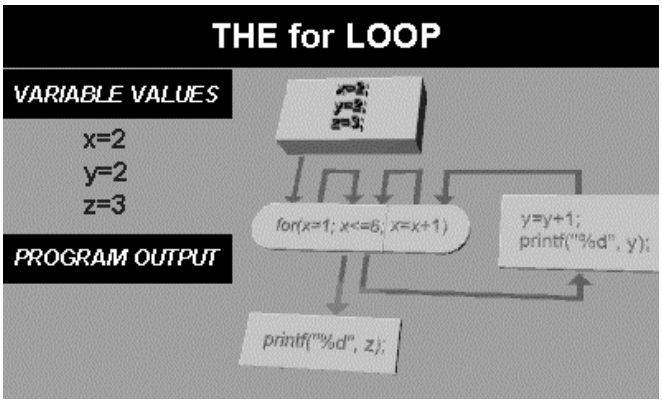


for (initial; *condition*; *increment)*

statement;

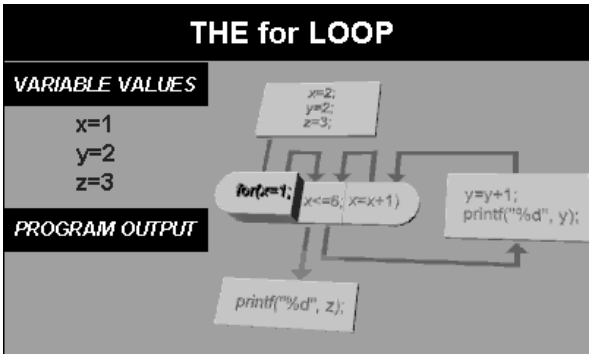
initial, *condition*, and *increment* are all C expressions, and *statement* is a single or compound C statement.

The following diagram shows the order of processing each part of a *for*

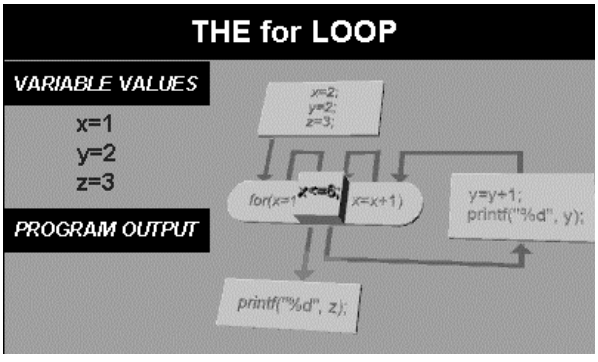


The following diagram shows the initial state of the program, after the initialization of the variables *x*, *y*, and *z*.

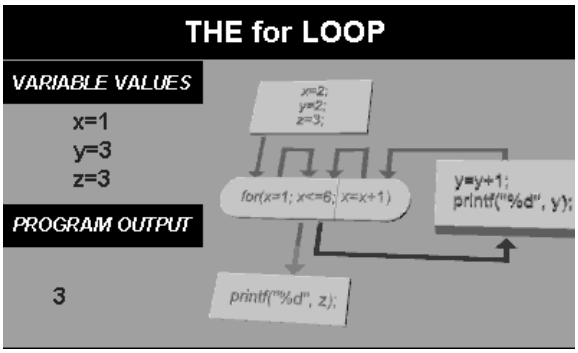
On entry to the *for* statement, the first expression is executed, which in our example assigns the value 1 to *x*. This can be seen in the graphic shown below (Note: see the Variable Values: section)



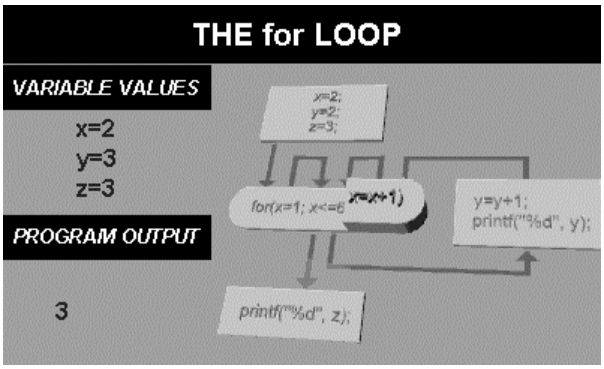
The next part of the *for* is executed, which tests the value of the loop variable *x* against the constant **6**.



It can be seen from the variable window that *x* has a current value of 1, so the test is successful, and program flow branches to execute the statements of the *for body*, which prints out the value of *y*, then adds 1 to *y*. You can see the program output and the state of the variables shown in the graphic below.

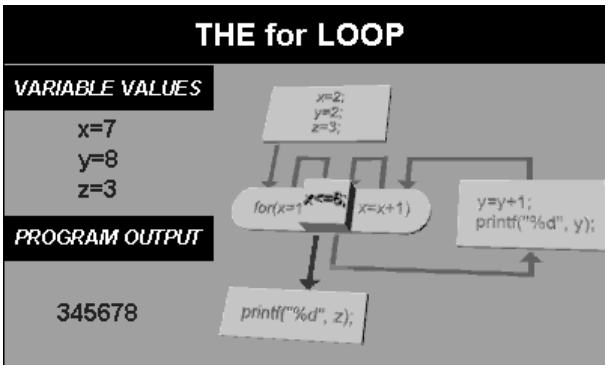


After executing the statements of the *for body*, execution returns to the last part of the *for* statement. Here, the value of *x* is incremented by 1. This is seen by the value of *x* changing to 2.



Next, the condition of the *for* variable is tested again. It continues because the value of it (2) is less than 6, so the body of the loop is executed again.

Execution continues till the value of *x* reaches 7. Lets now jump ahead in the animation to see this. Here, the condition test will fail, and the *for* statement finishes, passing control to the statement which follows.



When a for statement is encountered during program execution, the following events occur:

1. The expression *initial* is evaluated. *initial* is usually an assignment statement that sets a variable to a particular value.
2. The expression *condition* is evaluated. *condition* is typically a relational expression.
3. If *condition* evaluates to false (that is, as zero), the for statement terminates, and execution passes to the first statement following *statement*.
4. If *condition* evaluates to true (that is, as nonzero), the C statement(s) in *statement* are executed.
5. The expression *increment* is evaluated, and execution returns to step 2.

Here is a simple example. The program given below uses a for statement to print the numbers 1 through 20. You can see that the resulting code is much more compact than it would be if a separate printf() statement were used for each of the 20 values.

Program. A simple for statement.

```
1: /* Demonstrates a simple for statement */
2:
3: #include <stdio.h>
4:
5: int count;
6:
7: main()
8: {
9: /* Print the numbers 1 through 20 */
10:
11: for (count = 1; count <= 20; count++)
12: printf("%d\n", count);
13:
14: return 0;
15: }
```

```
1: /*Calculation of simple interestfor 3 sets of p,n and r */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:  int p,n,count;
8:  float r, si;
9:  for (count = 1; count <= 3; count=count+1)
10:  {
11:  printf("Enter values of p,n, and r");
12:  scanf("%d%d%f",&p,&n,&r);
13:
14:  si=p*n*r / 100;
15:  printf("Simple Interest = Rs.%f/n", si);
```

```
16:  }
17: }
```

ANALYSIS: Line 3 includes the standard input/output header file. Line 5 declares a type int variable, named count, that will be used in the for loop. Lines 11 and 12 are the for loop. When the for statement is reached, the initial statement is executed first. In this listing, the initial statement is count = 1. This initializes count so that it can be used by the rest of the loop. The second step in executing this for statement is the evaluation of the condition count <= 20. Because count was just initialized to 1, you know that it is less than 20, so the statement in the for command, the printf(), is executed. After executing the printing function, the increment expression, count++, is evaluated. This adds 1 to count, making it 2. Now the program loops back and checks the condition again. If it is true, the printf() re-executes, the increment adds to count (making it 3), and the condition is checked. This loop continues until the condition evaluates to false, at which point the program exits the loop and continues to the next line (line 14), which returns 0 before ending the program. The for statement is frequently used, as in the previous example, to "count up," incrementing a counter from one value to another. You also can use it to "count down," decrementing (rather than incrementing) the counter variable.

```
for (count = 100; count > 0; count—)
```

You can also "count by" a value other than 1, as in this example:

```
for (count = 0; count < 1000; count += 5)
```

The for statement is quite flexible. For example, you can omit the initialization expression if the test variable has been initialized previously in your program. (You still must use the semicolon separator as shown, however.)

```
count = 1;
for ( ; count < 1000; count++)
```

The initialization expression doesn't need to be an actual initialization; it can be any valid C expression. Whatever it is, it is executed once when the for statement is first reached.

```
for (count = 0; count < 100; )
printf("%d", count++);
```

The test expression that terminates the loop can be any C expression. As long as it evaluates to true (nonzero), the for statement continues to execute. You can use C's logical operators to construct complex test expressions. For example, the following for statement prints the elements of an array named array[], stopping when all elements have been printed or an element with a value of 0 is encountered:

```
for (count = 0; count < 1000 && array[count] != 0; count++)
printf("%d", array[count]);
```

You could simplify this for loop even further by writing it as follows.

```
for (count = 0; count < 1000 && array[count]; )
printf("%d", array[count++]);
```

You can follow the for statement with a null statement, allowing all the work to be done in the for statement itself. Remember, the null statement is a semicolon alone on a line. For example, to initialize all elements of a 1,000-element array to the value 50, you could write

```
for (count = 0; count < 1000; array[count++] = 50);
```

In this for statement, 50 is assigned to each member of the array by the increment part of the statement. You can create an expression by separating two sub expressions with the comma operator. The two sub expressions are evaluated (in left-to-right order), and the entire expression evaluates to the value of the right subexpression. By using the comma operator, you can make each part of a for statement perform multiple duties.

Imagine that you have two 1,000-element arrays, a[] and b[]. You want to copy the contents of a[] to b[] in reverse order so that after the copy operation, b[0] = a[999], b[1] = a[998], and so on. The following for statement does the trick:

```
for (i = 0, j = 999; i < 1000; i++, j—)
b[j] = a[i];
```

The comma operator is used to initialize two variables, i and j. It is also used to increment part of these two variables with each loop.

The for Statement

```
for (initial; condition; increment)
statement(s)
```

initial is any valid C expression. It is usually an assignment statement that sets a variable to a particular value. *condition* is any valid C expression. It is usually a relational expression. When *condition* evaluates to false (zero), the for statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the C *statement(s)* in *statement(s)* are executed. *increment* is any valid C expression. It is usually an expression that increments a variable initialized by the initial expression.

statement(s) are the C statements that are executed as long as the condition remains true.

A for statement is a looping statement. It can have an initialization, test condition, and increment as parts of its command. The for statement executes the initial expression first. It then checks the condition. If the condition is true, the statements execute. Once the statements are completed, the increment expression is evaluated. The forstatement then rechecks the condition and continues to loop until the condition is false.

Example 1

```
/* Prints the value of x as it counts from 0 to 9 */
int x;
for (x = 0; x <10; x++)
printf( "\nThe value of x is %d", x );
```

Example 2

```
/*Obtains values from the user until 99 is entered */
int nbr = 0;
for ( ; nbr != 99; )
scanf( "%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops */
int value[10];
int ctr,nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
puts("Enter a number, 99 to quit ");
scanf("%d", &nbr);
value[ctr] = nbr;
}
```

Exercise C9

Create a C program which calculates the triangular number of the users request, read from the keyboard using **scanf()**. A triangular number is the sum of the preceding numbers, so the triangular number 7 has a value of

7 + 6 + 5 + 4 + 3 + 2 + 1

Nesting for Statements

A for statement can be executed within another for statement. This is called *nesting*. By nesting for statements, you can do some complex programming. The program given below is not a complex program, but it illustrates the nesting of two for statements.

Nested for statements.

```
1: /* Demonstrates nesting two for statements */
2:
3: #include <stdio.h>
4:
5: void draw_box( int, int);
6:
7: main()
8: {
9: draw_box( 8, 35 );
10:
11: return 0;
12: }
13:
14: void draw_box( int row, int column )
15: {
16: int col;
17:         for ( ; row > 0; row—)
18:         {
19:           for (col = column; col > 0; col—)
20:           printf("X");
21:           printf("\n");
23:         }
24:  }
```

Output:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ANALYSIS: The main work of this program is accomplished on line 20. When you run this program, 280 Xs are printed on-screen, forming an 8*35 square. The program has only one command to print an X, but it is nested in two loops. In this listing, a function prototype for draw box() is declared on line 5. This function takes two-type int variables, row and column, which contain the dimensions of the box of Xs to be drawn. In line 9, main() calls draw_box() and passes the value 8 as the row and the value 35 as the column. Looking closely at the draw_box() function, you might see a couple things you don't readily understand. The first is why the local variable col was declared. The second is why the second printf() in line 22 was used. Both of these will become clearer after you look at the two for loops. Line 17 starts the first for loop. The initialization is skipped because the initial value of row was passed to the function. Looking at the condition, you see that this for loop is executed until the row is 0. On first executing line 17, row is 8; therefore, the program continues to line 19. Line 19 contains the second for statement. Here the passed parameter, column, is copied to a local variable, col, of type int. The value of col is 35 initially (the value passed via column), and column retains its original value. Because col is greater than 0, line 20 is executed, printing an X. col is then decremented, and the loop continues. When col is 0, the for loop ends, and control goes to line 22. Line 22 causes the on-screen printing to start on a new line. After moving .0000to a new line on the screen, control reaches the end of the first for loop's statements, thus executing the increment expression, which subtracts 1 from row, making it 7. This puts control back at line 19. Notice that the value of col was 0 when it was last used. If column had been used instead of col, it would fail the condition test, because it will never be greater than 0. Only the first line would be printed. Take the initializer out of line 19 and change the two col variables to column to see what actually happens.

- **DON'T** put too much processing in the for statement. Although you can use the comma separator, it is often clearer to put some of the functionality into the body of the loop.
- **DO** remember the semicolon if you use a for with a null statement. Put the semi-colon placeholder on a separate line, or place a space between it and the end of the for statement. It's clearer to put it on a separate line.

```
for (count = 0; count < 1000; array[count] = 50) ;
/* note space! */
```

Practise Exercise: for loops

1. Write a for loop to print out the values 1 to 10 on separate lines.

2. Write a for loop which will produce the following output (hint: use two nested for loops)

```
1
22
333
4444
55555
```

3. Write a for loop which sums all values between 10 and 100 into a variable called *total*. Assume that *total* has NOT been initialized to zero.

Answers

1. Write a for loop to print out the values 1 to 10 on separate lines.

```
for( loop = 1; loop <= 10; loop = loop + 1 )
printf("%d\n", loop) ;
```

2. Write a for loop which will produce the following output (hint: use two nested for loops)

```
1
22
333
4444
55555
for( loop = 1; loop <= 5; loop = loop + 1 )
{
        for( count = 1; count <= loop; count  = count + 1 )
         printf("%d", loop );
        printf("\n");
}
```

3. Write a for loop which sums all values between 10 and 100 into a variable called *total*. Assume that *total* has NOT been initialized to zero.

```
for( loop = 10, total = 0; loop <= 100; loop = loop + 1 )
total = total + loop;
```

4. Write a for loop to print out the character set from A-Z.

```
for( ch = 'A'; ch <= 'Z'; ch = ch + 1 )
printf("%c", ch );
printf("\n");
int out the character set from A-Z.
```

The while Statement

The while statement, also called the while *loop*, executes a block of statements as long as a specified condition is true. The while statement has the following form:

while (*condition*)

statement

condition is any C expression, and *statement* is a single or compound C statement. When program execution reaches a while statement, the following events occur:

1. The expression *condition* is evaluated.
2. If *condition* evaluates to false (that is, zero), the while statement terminates, and execution passes to the first statement following *statement*.
3. If *condition* evaluates to true (that is, nonzero), the C statement(s) in *statement* are executed.
4. Execution returns to step 1.

The program given below is a simple program that uses a while statement to print the numbers 1 through 20. (This is the same task that is performed by a for statement in the previous program.)

A simple while statement.

```
1: /* Demonstrates a simple while statement */
2:
3: #include <stdio.h>
4:
5: int count;
6:
7: int main()
8: {
9: /* Print the numbers 1 through 20 */
10:
11: count = 1;
12:
13: while (count <= 20)
14: {
15: printf("%d\n", count);
16: count++;
17: }
18: return 0;
19: }
```

ANALYSIS: Examine the while loop program and compare it with program of for loop, which uses a for statement to perform the same task. In line 11, count is initialized to 1. Because the while statement doesn't contain an initialization section, you must take care of initializing any variables before starting the while. Line 13 is the actual while statement, and it contains the same condition statement from the while loop program, count <= 20. In the while loop, line 16 takes care of incrementing count. What do you think would happen if you forgot to put line 16 in this program? Your program wouldn't know when to stop, because count would always be 1, which is always less than 20.

You might have noticed that a while statement is essentially a for statement without the initialization and increment components. Thus,

for (; *condition* ;)

is equivalent to

while (*condition*)

Because of this equality, anything that can be done with a for statement can also be done with a while statement. When you use a while statement, any necessary initialization must first be performed in a separate statement, and the updating must be performed by a statement that is part of the while loop. When initialization and updating are required, most experienced C programmers prefer to use a for statement rather than a while statement. This preference is based primarily on source code readability. When you use a for statement, the initialization, test, and increment expressions are located together and are easy to find and modify. With a while statement, the initialization and update expressions are located separately and might be less obvious.

The while Statement

while (*condition*)

statement(s)

condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following *statement(s)*; otherwise, the first C statement in *statement(s)* is executed. *statement(s)* is the C statement(s) that is executed as long as *condition* remains true. A while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). If the condition is not true when the while command is first executed, the *statement(s)* is never executed.

Example 1

```
int x = 0;
while (x < 10)
{
printf("\nThe value of x is %d", x );
x++;
}
```

Example 2

```
/* get numbers until you get one greater than 99 */
int nbr=0;
while (nbr <= 99)
scanf("%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops */
int value[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
puts("Enter a number, 99 to quit ");
scanf("%d", &nbr);
value[ctr] = nbr;
ctr++;
}
```

Answers: Practise Exercise : while loops and if else

1. Use a while loop to print the integer values 1 to 10 on the screen

```
12345678910
#include <stdio.h>
main()
{
        int loop;
        loop = 1;
        while( loop <= 10 ) {
         printf("%d", loop);
         loop++;
        }
        printf("\n");
}
```

2. Use a nested while loop to reproduce the following output

```
1
22
333
4444
55555
```

```
#include <stdio.h>
main()
{
        int loop;
        int count;
        loop = 1;
        while( loop <= 5 ) {
                count = 1;
                while( count <=
loop ) {
printf("%d", count);
                                count++;
                }
                loop++;
        }
        printf("\n");
}
```

3. Use an if statement to compare the value of an integer called sum against the value 65, and if it is less, print the text string "Sorry, try again".

```
if( sum < 65 )
        printf("Sorry, try again.\n");
```

4. If total is equal to the variable good_guess, print the value of total, else print the value of good_guess.

```
if( total == good_guess )
        printf("%d\n", total );
else
```

```
printf("%d\n", good_guess );
```

Compound Relationals (And, Not, Or, Eor)

Combining more than one condition

These allow the testing of more than one condition as part of selection statements. The symbols are

LOGICAL AND &&

Logical and requires all conditions to evaluate as TRUE (non-zero).

LOGICAL OR | |

Logical or will be executed if any ONE of the conditions is TRUE (non-zero).

LOGICAL NOT !

logical not negates (changes from TRUE to FALSE, vsvs) a condition.

LOGICAL EOR ^

Logical eor will be excuted if either condition is TRUE, but NOT if they are all true.

The following program uses an *if* statement with logical OR to validate the users input to be in the range 1-10.

```
#include <stdio.h>
main()
{
        int number;
        int valid = 0;
        while( valid == 0 ) {
                printf("Enter a number between 1 and 10 —>");
                scanf("%d", &number);
                if( (number < 1 ) | | (number > 10) ){
printf("Number is outside range 1-10. Please re-enter\n");
                        valid = 0;
                }
                else
                        valid = 1;
        }
        printf("The number is %d\n", number );
}
```

Sample Program Output

Enter a number between 1 and 10 —> *56*

Number is outside range 1-10. Please re-enter

Enter a number between 1 and 10 —> *6*

The number is 6

This program is slightly different from the previous example in that a LOGICAL OR eliminates one of the else clauses.

Compound Relationals (And, Not, Or, Eor)

Negation

```
#include <stdio.h>

main()
{
        int flag = 0;
        if( ! flag ) {
                printf("The flag is not set.\n");
                flag = ! flag;
        }
        printf("The value of flag is %d\n", flag);
}
```

Sample Program Output

The flag is not set.

The value of flag is 1

The program tests to see if *flag* is not (!) set; equal to zero. It then prints the appropriate message, changes the state of *flag*; *flag* becomes equal to not *flag*; equal to 1. Finally the value of *flag* is printed.

Compound Relationals (And, Not, Or, Eor)

Range checking using Compound Relationals

Consider where a value is to be inputted from the user, and checked for validity to be within a certain range, lets say between the integer values 1 and 100.

```
#include <stdio.h>

main()
{
        int number;
        int valid = 0;

        while( valid == 0 ) {
printf("Enter a number between 1 and 100");
                scanf("%d", &number );
        if( (number < 1) | | (number > 100) )
        printf("Number is outside legal range\n");
                else
                        valid = 1;
        }
        printf("Number is %d\n", number );
}
```

Sample Program Output

Enter a number between 1 and 100

203

Number is outside legal range

Enter a number between 1 and 100

-2

Number is outside legal range

Enter a number between 1 and 100

37

Number is 37

LESSON 8 While & Do While Loop

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with looping statements?
- Know how a do while statement works.
- Know how to handle nested loops.

Nesting while Statements

Just like the for and if statements, while statements can also be nested.

- **DON'T** use the following convention if it isn't necessary: while (x). Instead, use this convention: while (x != 0)Although both work, the second is clearer when you're debugging (trying to find problems in) the code. When compiled, these produce virtually the same code.
- **DO** use the for statement instead of the while statement if you need to initialize and increment within your loop. The for statement keeps the initialization, condition, and increment statements all together. The while statement does not.

The do...while Loop

C's third loop construct is the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop. The structure of the do...while loop is as follows:

do

statement

while (*condition*);

condition is any C expression, and *statement* is a single or compound C statement. When program execution reaches a do...while statement, the following events occur:

1. The statements in *statement* are executed.

2. *condition* is evaluated. If it's true, execution returns to step 1. If it's false, the loop terminates. The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In contrast, for loops and while loops evaluate the test condition at the start of the loop, so the associated statements are not executed at all if the test condition is initially false. The do...while loop is used less frequently than while and for loops. It is most appropriate when the statement(s) associated with the loop must be executed at least once. You could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop. A do...while loop probably would be more straightforward, however. The program given below shows an example of a do...while loop.

A simple do...while loop.

```
1: /* Demonstrates a simple do...while statement */
2:
3: #include <stdio.h>
4:
5: int get_menu_choice( void );
6:
7: main()
8: {
9: int choice;
10:
11: choice = get_menu_choice();
12:
13: printf("You chose Menu Option %d\n", choice );
14:
15: return 0;
16: }
17:
18: int get_menu_choice( void )
19: {
20: int selection = 0;
21:
22: do
23: {
24: printf("\n" );
25: printf("\n1 - Add a Record" );
26: printf("\n2 - Change a record");
27: printf("\n3 - Delete a record");
28: printf("\n4 - Quit");
29: printf("\n" );
30: printf("\nEnter a selection: " );
31:
32: scanf("%d", &selection );
33:
34: }while ( selection < 1 || selection > 4 );
35:
36: return selection;
37: }
```

```
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
```

```
Enter a selection: 8
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit

Enter a selection: 4

You chose Menu Option 4
```

ANALYSIS: This program provides a menu with four choices. The user selects one of the four choices, and then the program prints the number selected. Programs later in this book use and expand on this concept. For now, you should be able to follow most of the listing. The main() function (lines 7 through 16) adds nothing to what you already know.

NOTE: The body of main() could have been written into one line, like this:

```
printf( "You chose Menu Option %d", get_menu_option() );
```

If you were to expand this program and act on the selection, you would need the value returned by get_menu_choice(), so it is wise to assign the value to a variable (such as choice).

Lines 18 through 37 contain get_menu_choice(). This function displays a menu on-screen (lines 24 through 30) and then gets a selection. Because you have to display a menu at least once to get an answer, it is appropriate to use a do...while loop. In the case of this program, the menu is displayed until a valid choice is entered. Line 34 contains the while part of the do...while statement and validates the value of the selection, appropriately named selection. If the value entered is not between 1 and 4, the menu is redisplayed, and the user is prompted for a new value. When a valid selection is entered, the program continues to line 36, which returns the value in the variable selection.

The do...while Statement

```
do
{
statement(s)
}while (condition);
```

condition is any valid C expression, usually a relational expression. When *condition* evaluates to false (zero), the while statement terminates, and execution passes to the first statement following the while statement; otherwise, the program loops back to the do, and the C statement(s) in *statement(s)* is executed. *statement(s)* is either a single C statement or a block of statements that are executed the first time through the loop and then as long as *condition* remains true. A do...while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). Unlike the while statement, a do...while loop executes its statements at least once.

Example 1

```
/* prints even though condition fails! */
int x = 10;
do
{
printf("\nThe value of x is %d", x );
}while (x != 10);
```

Example 2

```
/* gets numbers until the number is greater than 99 */
int nbr;
do
{
scanf("%d", &nbr );
}while (nbr <= 99);
```

Example 3

```
/* Enables user to enter up to 10 integer values */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops */
int value[10];
int ctr = 0;
int nbr;
do
{
puts("Enter a number, 99 to quit ");
scanf( "%d", &nbr);
value[ctr] = nbr;
ctr++;
}while (ctr < 10 && nbr != 99);
```

Nested Loops

The term *nested loop* refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops, except that each inner loop must be enclosed completely in the outer loop; you can't have overlapping loops. Thus, the following is not allowed:

```
for ( count = 1; count < 100; count++)
{
do
{
/* the do...while loop */
} /* end of for loop */
}while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem:

```
for (count = 1; count < 100; count++)
{
do
{
/* the do...while loop */
}while (x != 0);
} /* end of for loop */
```

When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. Note, however, that the inner loop might be independent from any variables in the outer loop; in this example, they are not. In the previous example, if the inner do...while loop modifies the

value of count, the number of times the outer for loop executes is affected. Good indenting style makes code with nested loops easier to read. Each level of loop should be indented one step further than the last level. This clearly labels the code associated with each loop.

- **DON'T** try to overlap loops. You can nest them, but they must be entirely within each other.
- **DO** use the do...while loop when you know that a loop should be executed at least once.

Summary

Now you are almost ready to start writing real C programs on your own. C has three loop statements that control program execution: for, while, and do...while. Each of these constructs lets your program execute a block of statements zero times, one time, or more than one time, based on the condition of certain program variables. Many programming tasks are well served by the repetitive execution allowed by these loop statements.

Although all three can be used to accomplish the same task, each is different. The for statement lets you initialize, evaluate, and increment all in one command. The while statement operates as long as a condition is true. The do...while statement always executes its statements at least once and continues to execute them until a condition is false. Nesting is the placing of one command within another. C allows for the nesting of any of its commands. Nesting the if statement was demonstrated on Day 4. In this lesson, the for, while, and do...while statements were nested.

Q&A

Q How do I know which programming control statement to use—the for, the while, or the do...while?

A If you look at the syntax boxes provided, you can see that any of the three can be used to solve a looping problem. Each has a small twist to what it can do, however. The for statement is best when you know that you need to initialize and increment in your loop. If you only have a condition that you want to meet, and you aren't dealing with a specific number of loops, while is a good choice. If you know that a set of statements needs to be executed at least once, a do...while might be best. Because all three can be used for most problems, the best course is to learn them all and then evaluate each programming situation to determine which is best.

Q How deep can I nest my loops?

A You can nest as many loops as you want. If your program requires you to nest more than two loops deep, consider using a function instead. You might find sorting through all those braces difficult, so perhaps a function would be easier to follow in code.

Q Can I nest different loop commands?

A You can nest if, for, while, do...while, or any other command. You will find that many of the programs you try to write will require that you nest at least a few of these.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the index value of the first element in an array?
2. What is the difference between a for statement and a while statement?
3. What is the difference between a while statement and a do...while statement?
4. Is it true that a while statement can be used and still get the same results as coding a for statement?
5. What must you remember when nesting statements?
6. Can a while statement be nested in a do...while statement?
7. What are the four parts of a for statement?
8. What are the two parts of a while statement?
9. What are the two parts of a do...while statement?

Exercises

1. Write a declaration for an array that will hold 50 type long values.
2. Show a statement that assigns the value of 123.456 to the 50th element in the array from exercise 1.
3. What is the value of x when the following statement is complete?

   ```
   for (x = 0; x < 100, x++) ;
   ```

4. What is the value of ctr when the following statement is complete?

   ```
   for (ctr = 2; ctr < 10; ctr += 3) ;
   ```

5. How many Xs does the following print?

   ```
   for (x = 0; x < 10; x++)
   for (y = 5; y > 0; y—)
   puts("X");
   ```

6. Write a for statement to count from 1 to 100 by 3s.
7. Write a while statement to count from 1 to 100 by 3s.
8. Write a do...while statement to count from 1 to 100 by 3s.
9. **BUG BUSTER:** What is wrong with the following code fragment?

   ```
   record = 0;
   while (record < 100)
   {
   printf( "\nRecord %d ", record );
   printf( "\nGetting next number..." );
   }
   ```

10. **BUG BUSTER:** What is wrong with the following code fragment? (MAXVALUES is not the problem!)

    ```
    for (counter = 1; counter < MAXVALUES; counter++);
    printf("\nCounter = %d", counter );
    ```

LESSON 9 Break Statement

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with break and continue statements
- Know how a go to statement works.
- Know how to work with the menu system.

In the previous lesson we have introduced some of C's program control statements that govern the execution of other statements in your program. This lesson covers more advanced aspects of program control, including the goto statement and some of the more interesting things you can do with loops. Today you will learn :

- How to use the break and continue statements
- What infinite loops are and why you might use them
- What the goto statement is and why you should avoid it
- How to use the switch statement
- How to control program exits
- How to execute functions automatically upon program completion
- How to execute system commands in your program

Ending Loops Early

In the previous lessons, we have learnt how the for loop, the while loop, and the do...while loop can control program execution. These loop constructions execute a block of C statements never, once, or more than once, depending on conditions in the program. In all three cases, termination or exit of the loop occurs only when a certain condition occurs. At times, however, you might want to exert more control over loop execution. The break and continue statements provide this control.

The Break Statement

The break statement can be placed only in the body of a for loop, while loop, or do...while loop. (It's valid in a switch statement too). When a break statement is encountered, execution exits the loop. The following is an example:

```
for ( count = 0; count < 10; count++ )
{
if ( count == 5 )
break;
}
```

Left to itself, the for loop would execute 10 times. On the sixth iteration, however, count is equal to 5, and the break statement executes, causing the for loop to terminate. Execution then passes to the statement immediately following the for loop's closing brace. When a break statement is encountered inside a nested loop, it causes the program to exit the innermost loop only.

The Break Statement

```
break;
```

break is used inside a loop or switch statement. It causes the control of a program to skip past the end of the current loop (for, while, or do...while) or switch statement. No further iterations of the loop execute; the first command following the loop or switch statement executes.

Example

```
int x;
printf ( "Counting from 1 to 10\n" );
/* having no condition in the for loop will cause it to loop forever */
for( x = 1; ; x++ )
{
if( x == 10 ) /* This checks for the value of 10 */
break; /* This ends the loop */
printf( "\n%d", x );
}
```

The Continue Statement

Like the break statement, the continue statement can be placed only in the body of a for loop, a while loop, or a do...while loop. When a continue statement executes, the next iteration of the enclosing loop begins immediately. The statements between the continue statement and the end of the loop aren't executed.

The program given below uses the continue statement. This program accepts a line of input from the keyboard and then displays it with all lowercase vowels removed.

Using the Continue Statement

```
1: /* Demonstrates the continue statement. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: /* Declare a buffer for input and a counter variable. */
8:
9: char buffer[81];
10: int ctr;
11:
12: /* Input a line of text. */
13:
14: puts("Enter a line of text:");
15: gets(buffer);
```

```
16:
17: /* Go through the string, displaying only those */
18: /* characters that are not lowercase vowels. */
19:
20: for (ctr = 0; buffer[ctr] !='\0'; ctr++)
21: {
22:
23: /* If the character is a lowercase vowel, loop back */
24: /* without displaying it. */
25:
26: if (buffer[ctr] == 'a' || buffer[ctr] == 'e'
27:         || buffer[ctr] == 'i' || buffer[ctr] == 'o'
28:         || buffer[ctr] == 'u')
29: continue;
30:
31: /* If not a vowel, display it. */
32:
33: putchar(buffer[ctr]);
34: }
35: return 0;
36: }
```

```
Enter a line of text:
This is a line of text
Ths s ln f txt
```

ANALYSIS:

Although this isn't the most practical program, it does use a continue statement effectively. Lines 9 and 10 declare the program's variables. buffer[] holds the string that the user enters in line 15. The other variable, ctr, increments through the elements of the array buffer[], while the for loop in lines 20 through 34 searches for vowels. For each letter in the loop, an if statement in lines 26 through 28 checks the letter against lowercase vowels. If there is a match, a continue statement executes, sending control back to line 20, the for statement. If the letter isn't a vowel, control passes to the if statement, and line 33 is executed. Line 33 contains a new library function, putchar(), which displays a single character on-screen.

The Continue Statement

```
continue;
```

continue is used inside a loop. It causes the control of a program to skip the rest of the current iteration of a loop and start the next iteration.

Example

```
int x;
printf("Printing only the even numbers from 1 to 10\n");
for( x = 1; x <= 10; x++ )
{
if( x % 2 != 0 ) /* See if the number is NOT even */
continue; /* Get next instance x */
printf( "\n%d", x );
}
```

The Goto Statement

The goto statement is one of C's *unconditional jump*, or *branching*, statements. When program execution reaches a goto statement, execution immediately jumps, or branches, to the location specified by the goto statement. This statement is unconditional because execution always branches when a goto statement is encountered; the branch doesn't depend on any program conditions (unlike if statements, for example).

A goto statement and its target must be in the same function, but they can be in different blocks. The simple program that is given below uses a goto statement.

Using the goto statement.

```
1: /* Demonstrates the goto statement */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: int n;
8:
9: start: ;
10:
11: puts("Enter a number between 0 and 10: ");
12: scanf("%d", &n);
13:
14: if (n < 0 ||n > 10 )
15: goto start;
16: else if (n == 0)
17: goto location0;
18: else if (n == 1)
19: goto location1;
20: else
21: goto location2;
22:
23: location0: ;
24: puts("You entered 0.\n");
25: goto end;
26:
27: location1: ;
28: puts("You entered 1.\n");
29: goto end;
30:
31: location2: ;
32: puts("You entered something between 2 and 10.\n");
33:
34: end: ;
35: return 0;
36: }
```

Output

```
Enter a number between 0 and 10:
1
You entered 1.
Enter a number between 0 and 10:
```

```
9
```

```
You entered something between 2 and 10.
```

ANALYSIS

This is a simple program that accepts a number between 0 and 10. If the number isn't between 0 and 10, the program uses a goto statement on line 15 to go to start, which is on line 9. Otherwise, the program checks on line 16 to see whether the number equals 0. If it does, a goto statement on line 17 sends control to location0 (line 23), which prints a statement on line 24 and executes another goto. The goto on line 25 sends control to end at the end of the program. The program executes the same logic for the value of 1 and all values between 2 and 10 as a whole. The target of a goto statement can come either before or after that statement in the code. The only restriction, as mentioned earlier, is that both the goto and the target must be in the same function. They can be in different blocks, however. You can use goto to transfer execution both into and out of loops, such as a for statement, but you should never do this. In fact, I strongly recommend that you never use the goto statement anywhere in your programs.

There are two reasons:

- You don't need it. No programming task requires the goto statement. You can always write the needed code using C's other branching statements.
- It's dangerous. The goto statement might seem like an ideal solution for certain programming problems, but it's easy to abuse. When program execution branches with a goto statement, no record is kept of where the execution came from, so execution can weave through the program willy-nilly. This type of programming is known as *spaghetti code*.

Some careful programmers can write perfectly fine programs that use goto. There might be situations in which a judicious use of goto is the simplest solution to a programming problem. It's never the only solution, however. If you're going to ignore this warning, at least be careful!

- **DO** avoid using the goto statement if possible.
- **DON'T** confuse break and continue. break ends a loop, whereas continue starts the next iteration.

The goto Statement

goto *location*;

location is a label statement that identifies the program location where execution is to branch. A *label statement* consists of an identifier followed by a colon and a C statement:

location: *a C statement*; If you want the label by itself on a line, you can follow it with the null statement (a semicolon by itself):

location: ;

Infinite Loops

What is an infinite loop, and why would you want one in your program? An infinite loop is one that, if left to its own devices, would run forever. It can be a for loop, a while loop, or a do...while loop. For example, if you write

```
while (1)
{
/* additional code goes here */
}
```

you create an infinite loop. The condition that the while tests is the constant 1, which is always true and can't be changed by the program. Because 1 can never be changed on its own, the loop never terminates.

In the preceding section, you saw that the break statement can be used to exit a loop. Without the break statement, an infinite loop would be useless. With break, you can take advantage of infinite loops. You can also create an infinite for loop or an infinite do...while loop, as follows:

```
for (;;)
{
/* additional code goes here */
}
do
{
/* additional code goes here */
} while (1);
```

The principle remains the same for all three-loop types. This section's examples use the while loop. An infinite loop can be used to test many conditions and determine whether the loop should terminate. It might be difficult to include all the test conditions in parentheses after the while statement. It might be easier to test the conditions individually in the body of the loop, and then exit by executing a break as needed.

An infinite loop can also create a menu system that directs your program's operation. A program's main() function often serves as a sort of "traffic cop," directing execution among the various functions that do the real work of the program. This is often accomplished by a menu of some kind: The user is presented with a list of choices and makes an entry by selecting one of them. One of the available choices should be to terminate the program. Once a choice is made, one of C's decision statements is used to direct program execution accordingly.

The program given below demonstrates a menu system.

Using an infinite loop to implement a menu system.

```
1: /* Demonstrates using an infinite loop to implement */
2: /* a menu system. */
3: #include <stdio.h>
4: #define DELAY 1500000 /* Used in delay loop. */
5:
6: int menu(void);
7: void delay(void);
8:
9: main()
10: {
11: int choice;
12:
13: while (1)
14: {
```

```
15:
16: /* Get the user's selection. */
17;
18: choice = menu();
19:
20: /* Branch based on the input. */
21:
22; if (choice == 1)
23: {
24: puts("\nExecuting choice 1.");
25: delay();
26: }
27: else if (choice == 2)
28: {
29: puts("\nExecuting choice 2.");
30: delay();
31: }
32; else if (choice == 3)
33: {
34: puts("\nExecuting choice 3.");
35: delay();
36: }
37: else if (choice == 4)
38: {
39: puts("\nExecuting choice 4.");
40: delay();
41: }
42: else if (choice == 5) /* Exit program. */
43: {
44: puts("\nExiting program now...\n");
45: delay();
46: break;
47: }
48: else
49: {
50: puts("\nInvalid choice, try again.");
51: delay();
52: }
53: }
54: return 0;
55: }
56:
57: /* Displays a menu and inputs user's selection. */
58: int menu(void)
59: {
60: int reply;
61:
62: puts("\nEnter 1 for task A.");
63: puts("Enter 2 for task B.");
64: puts("Enter 3 for task C.");
65: puts("Enter 4 for task D.");
66: puts("Enter 5 to exit program.");
67:
68: scanf("%d", &reply);
69:
70: return reply;
71: }
72:
73: void delay( void )
74: {
75: long x;
76: for ( x = 0; x < DELAY; x++ )
77: ;
78: }
```

Output

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

ANALYSIS

In the above program, a function named menu() is called on line 18 and defined on lines 58 through 71. menu() displays a menu on-screen, accepts user input, and returns the input to the main program. In main(), a series of nested if statements tests the returned value and directs execution accordingly. The only thing this program does is display messages on-screen. In a real program, the code would call various functions to perform the selected task. This program also uses a second function named

delay(). delay() is defined on lines 73 through 78 and really doesn't do much. Simply stated, the for statement on line 76 loops, doing nothing (line 77). The statement loops DELAY times. This is an effective method of pausing the program momentarily. If the delay is too short or too long, the defined value of DELAY can be adjusted accordingly.

Both Borland and Symantec offer a function similar to delay(), called sleep(). This function pauses program execution for the number of seconds that is passed as its argument. To use sleep(), a program must include the header file TIME.H if you're using the Symantec compiler. You must use DOS.H if you're using a Borland compiler. If you're using either of these compilers or a compiler that supports sleep(), you could use it instead of delay().

Warning

There are better ways to pause the computer than what is shown above. If you choose to use a function such as sleep(), as just mentioned, be cautious. The sleep() function is not ANSI-compatible. This means that it might not work with other compilers or on all platforms.

Notes

LESSON 10 Switch Statement

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with switch statements?
- Know how a execute operating system commands in a program.

The switch Statement

C's most flexible program control statement is the switch statement, which lets your program execute different statements based on an expression that can have more than two values. Earlier control statements, such as if, were limited to evaluating an expression that could have only two values: true or false. To control program flow based on more than two values, you had to use multiple nested if statements, as shown above. The switch statement makes such nesting unnecessary.

The general form of the switch statement is as follows:

switch (*expression*)

{

case *template_1*: *statement(s)*;

case *template_2*: *statement(s)*;

...

case *template_n*: *statement(s)*;

default: *statement(s)*;

}

In this statement, *expression* is any expression that evaluates to an integer value: type long, int, or char. The switch statement evaluates *expression* and compares the value against the templates following each case label, and then one of the following happens:

- If a match is found between *expression* and one of the templates, execution is transferred to the statement that follows the case label.
- If no match is found, execution is transferred to the statement following the optional default label.
- If no match is found and there is no default label, execution passes to the first statement following the switch statement's closing brace.

The switch statement is demonstrated in the program given below, which displays a message based on the user's input.

Using the switch statement

```
1: /* Demonstrates the switch statement. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: int reply;
8:
9: puts("Enter a number between 1 and 5:");
10: scanf("%d", &reply);
11:
12: switch (reply)
13: {
14: case 1:
15: puts("You entered 1.");
16: case 2:
17: puts("You entered 2.");
18: case 3:
19: puts("You entered 3.");
17:44:29]
20: case 4:
21: puts("You entered 4.");
22: case 5:
23: puts("You entered 5.");
24: default:
25: puts("Out of range, try again.");
26: }
27:
28: return 0;
29: }
```

Output

```
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

ANALYSIS

Well, that's certainly not right, is it? It looks as though the switch statement finds the first matching template and then executes everything that follows (not just the statements associated with the template). That's exactly what does happen, though. That's how switch is supposed to work. In effect, it performs a goto to the matching template. To ensure that only the statements associated with the matching template are executed, include a break statement where needed. The program

given below shows the rewritten form of the above program with break statements. Now it functions properly.

Correct use of switch, including break statements as needed.

```
1: /* Demonstrates the switch statement correctly. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: int reply;
8:
9: puts("\nEnter a number between 1 and 5:");
10: scanf("%d", &reply);
11:
12: switch (reply)
13: {
14: case 0:
15: break;
16 case 1:
17: {
18: puts("You entered 1.\n");
19: break;
20: }
21: case 2:
22: {
23: puts("You entered 2.\n");
24: break;
25: }
26: case 3:
27: {
28: puts("You entered 3.\n");
29: break;
30: }
31: case 4:
32: {
33: puts("You entered 4.\n");
34: break;
35: }
36: case 5:
37: {
38: puts("You entered 5.\n");
39: break;
40: }
41: default:
42: {
43: puts("Out of range, try again.\n");
44: }
45: } /* End of switch */
46:
47: }
```

Output

```
Enter a number between 1 and 5:
1
You entered 1.
Enter a number between 1 and 5:
6
Out of range, try again.
```

Compile and run this version; it runs correctly.

The program given below uses switch instead of if to implement a menu. Using switch is much better than using nested if statements, which were used in the earlier version of the menu program

Using the switch statement to execute a menu system

```
1: /* Demonstrates using an infinite loop and the switch */
2: /* statement to implement a menu system. */
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: #define DELAY 150000
7:
8: int menu(void);
9: void delay(void);
10:
11: main()
12: {
13:
14: while (1)
15: {
16: /* Get user's selection and branch based on the input. */
17:
18: switch(menu())
19: {
20: case 1:
21: {
22: puts("\nExecuting choice 1.");
23: delay();
24: break;
25: }
26: case 2:
27: {
28: puts("\nExecuting choice 2.");
29: delay();
30: break;
31: }
32: case 3:
33: {
34: puts("\nExecuting choice 3.");
35: delay();
36: break;
37: }
38: case 4:
39: {
40: puts("\nExecuting choice 4.");
41: delay();
42: break;
43: }
44: case 5: /* Exit program. */
45: {
46: puts("\nExiting program now...\n");
47: delay();
48: exit(0);
```

```
49: }
50: default:
51: {
52: puts("\nInvalid choice, try again.");
53: delay();
54: }
55: } /* End of switch */
56: } /* End of while */
57:
58: }
59:
60: /* Displays a menu and inputs user's selection. */
61: int menu(void)
62: {
63: int reply;
64:
65: puts("\nEnter 1 for task A.");
66: puts("Enter 2 for task B.");
67: puts("Enter 3 for task C.");
68: puts("Enter 4 for task D.");
69: puts("Enter 5 to exit program.");
70:
71: scanf("%d", &reply);
72:
73: return reply;
74: }
75:
76: void delay( void )
77: {
78: long x;
79: for( x = 0; x < DELAY; x++ )
80: ;
81: }
```

Output

```
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5

Exiting program now...
```

One other new statement is in this program: the exit() library function in the statements associated with case 5: on line 48. You can't use break here, as you did in Listing 13.4. Executing a break would merely break out of the switch statement; it wouldn't break out of the infinite while loop. As you'll learn in the next section, the exit() function terminates the program.

However, having execution "fall through" parts of a switch construction can be useful at times. Say, for example, that you want the same block of statements executed if one of several values is encountered. Simply omit the break statements and list all the case templates before the statements. If the test expression matches any of the case conditions, execution will "fall through" the following case statements until it reaches the block of code you want executed.

The switch Statement

switch (*expression*)

{

case *template_1*: *statement(s)*;

case *template_2*: *statement(s)*;

...

case *template_n*: *statement(s)*;

default: *statement(s)*;

}

The switch statement allows for multiple branches from a single expression. It's more efficient and easier to follow than a multileveled if statement. A switch statement evaluates an expression and then branches to the case statement that contains the template matching the expression's result. If no template matches the expression's result, control goes to the default statement. If there is no default statement, control goes to the end of the switch statement.

Program flow continues from the case statement down unless a break statement is encountered. In that case, control goes to the end of the switch statement.

Example 1

```
switch( letter )
{
case 'A':
case 'a':
printf( "You entered A" );
break;
case 'B':
case 'b':
printf( "You entered B");
break;
...
...
default:
printf( "I don't have a case for %c", letter );
}
```

Example 2

```
switch( number )
{
case 0: puts( "Your number is 0 or less.");
case 1: puts( "Your number is 1 or less.");
case 2: puts( "Your number is 2 or less.");
case 3: puts( "Your number is 3 or less.");
...
...
case 99: puts( "Your number is 99 or less.");
break;
default: puts( "Your number is greater than 99.");
}
```

Because there are no break statements for the first case statements, this example finds the case that matches the number and prints every case from that point down to the break in case 99. If the number was 3, you would be told that your number is equal to 3 or less, 4 or less, 5 or less, up to 99 or less. The program continues printing until it reaches the break statement in case 99.

- **DON'T** forget to use break statements if your switch statements need them.
- **DO** use a default case in a switch statement, even if you think you've covered all possible cases.
- **DO** use a switch statement instead of an if statement if more than two conditions are being evaluated for the same variable.
- **DO** line up your case statements so that they're easy to read.

Exiting the Program

A C program normally terminates when execution reaches the closing brace of the main() function. However, you can terminate a program at any time by calling the library function exit(). You can also specify one or more functions to be automatically executed at termination.

The exit() Function

The exit() function terminates program execution and returns control to the operating system. This function takes a single type int argument that is passed back to the operating system to indicate the program's success or failure. The syntax of the exit() function is

exit(*status*);

If *status* has a value of 0, it indicates that the program terminated normally. A value of 1 indicates that the program terminated with some sort of error. The return value is usually ignored. In a DOS system, you can test the return value with a DOS batch file and the **if errorlevel** statement. If you're using an operating system other than DOS, you should check its documentation to determine how to use a return value from a program.

To use the exit() function, a program must include the header file **stdlib.h**. This header file also defines two symbolic constants for use as arguments to the exit() function:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Thus, to exit with a return value of 0,

```
call exit(EXIT_SUCCESS);
```

for a return value of 1,

call exit(EXIT_FAILURE).

- **DO** use the exit() command to get out of the program if there's a problem.
- **DO** pass meaningful values to the exit() function.

Executing Operating System Commands in a Program

The C standard library includes a function, system(), that lets you execute operating system commands in a running C program. This can be useful, allowing you to read a disk's directory listing or format a disk without exiting the program. To use the system() function, a program must include the header file **stdlib.h**. The format of system() is

system(*command*);

The argument *command* can be either a string constant or a pointer to a string. For example, to obtain a directory listing in DOS, you could write either system("dir");

Or char *command = "dir"; system(command); After the operating system command is executed, execution returns to the program at the location immediately following the call to system(). If the command you pass to system() isn't a valid operating system command, you get a Bad command or file name error message before returning to the program.

Summary

You learned about the goto statement and why you should avoid using it in your programs. You saw that the break and continue statements give additional control over the execution of loops and that these statements can be used in conjunction with infinite loops to perform useful programming tasks. Also you have learnt how to use the exit() function to control program termination. Finally, you saw how to use the system() function to execute system commands from within your program.

Q&A

Q Is it better to use a switch statement or a nested loop?

A If you're checking a variable that can take on more than two values, the switch statement is almost always better. The resulting code is easier to read, too. If you're checking a true/false condition, go with an if statement.

Q Why should I avoid a goto statement?

A When you first see a goto statement, it's easy to believe that it could be useful. However, goto can cause you more problems than it fixes. A goto statement is an unstructured command that takes you to another point in a program. Many debuggers (software that helps you trace program problems) can't interrogate a goto properly. goto statements also lead to spaghetti code—code that goes all over the place.

Q Why don't all compilers have the same functions?

A In this lesson, you saw that certain C functions aren't available with all compilers or all computer systems. For example, sleep() is available with the Borland C compilers but not with the Microsoft compilers. Although there are standards that all ANSI compilers follow, these standards don't prohibit compiler manufacturers from adding additional functionality. They do this by creating and including new functions. Each compiler manufacturer usually adds a number of functions that they believe will be helpful to their users.

Q Isn't C supposed to be a standardized language?

A C is, in fact, highly standardized. The American National Standards Institute (ANSI) has developed the ANSI C Standard, which specifies almost all details of the C language, including the functions that are provided. Some compiler vendors have added more functions—ones that aren't part of the ANSI standard—to their C compilers in an effort to one-up the competition. In addition, you sometimes come across a compiler that doesn't claim to meet the ANSI standard. If you limit yourself to ANSI-standard compilers, however, you'll find that 99 percent of program syntax and functions are common among them.

Q Is it good to use the system() function to execute system functions?

A The system() function might appear to be an easy way to do such things as lists the files in a directory, but you should be cautious. Most operating system commands are specific to a particular operating system. If you use a system() call, your code probably won't be portable. If you want to run another program (not an operating system command), you shouldn't have portability problems.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. When is it advisable to use the goto statement in your programs?
2. What's the difference between the break statement and the continue statement?
3. What is an infinite loop, and how do you create one?
4. What two events cause program execution to terminate?
5. What variable types can a switch evaluate to?
6. What does the default statement do?
7. What does the exit() function do?
8. What does the system() function do?

Exercises

1. Write a statement that causes control of the program to go to the next iteration in a loop.
2. Write the statement(s) that send control of a program to the end of a loop.
3. Write a line of code that displays a listing of all the files in the current directory (for a DOS system).
4. **BUG BUSTER:** Is anything wrong with the following code?

```
switch( answer )
{
case 'Y': printf("You answered yes");
break;
case 'N': printf( "You answered no");
}
```

5. **BUG BUSTER:** Is anything wrong with the following code?

```
switch( choice )
{
default:
printf("You did not choose 1 or 2");
case 1:
printf("You answered 1");
break;
case 2:
printf( "You answered 2");
break;
}
```

6. Rewrite exercise 5 using if statements.
7. Write an infinite do...while loop.

Because of the multitude of possible answers for the following exercises, answers are not provided. These are exercises for you to try "on your own."

8. **ON YOUR OWN:** Write a program that works like a calculator. The program should allow for addition, subtraction, multiplication, and division.
9. **ON YOUR OWN:** Write a program that provides a menu with five different options. The fifth option should quit the program. Each of the other options should execute a system command using the system() function.

LESSON 11 Functions

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to handle functions.
- Know how a function works.
- Know what are the advantages of structured programming

Functions are central to C programming and to the philosophy of C program design. C's library functions are complete functions supplied as part of your compiler. This lesson covers user-defined functions, which, as the name implies, are functions that you, the programmer, create.

Today you will learn

- What a function is and what its parts are ?
- About the advantages of structured programming with functions
- How to create a function
- How to declare local variables in a function
- How to return a value from a function to the program
- How to pass arguments to a function

What is a Function?

This lesson approaches the question "What is a function?" in two ways.

First, it tells you what functions are, and then it shows you how they're used.

A Function Defined

First the definition: A *function* is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

A function is named. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as *calling* the function.

A function can be called from within another function.

- *A function is independent*. A function can perform its task without interference from or interfering with other parts of the program.
- *A function performs a specific task*. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.
- *A function can return a value to the calling program*. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

That's all there is to the "telling" part. Keep the previous definition in mind as you look at the next section.

A Function Illustrated

The Program given below contains a user-defined function.

A program that uses a function to calculate the cube of a number.

```
1: /* Demonstrates a simple function */
2: #include <stdio.h>
3:
4: long cube(long x);
5:
6: long input, answer;
7:
8: main()
9: {
10: printf("Enter an integer value: ");
11: scanf("%d", &input);
12: answer = cube(input);
13: /* Note: %ld is the conversion specifier for */
14: /* a long integer */
15: printf("\nThe cube of %ld is %ld.\n", input, answer);
16:
17: return 0;
18: }
19:
20: /* Function: cube() - Calculates the cubed value of a variable */
21: long cube(long x)
22: {
23: long x_cubed;
24:
25: x_cubed = x * x * x;
26: return x_cubed;
27: }
```

Output

```
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3
The cube of 3 is 27.
```

NOTE: The following analysis focuses on the components of the program that relate directly to the function rather than explaining the entire program.

Analysis

Line 4 contains the *function prototype*, a model for a function that will appear later in the program. A function's prototype contains the name of the function, a list of variables that must be passed to it, and the type of variable it returns, if any. Looking at line 4, you can tell that the function is named cube, that it requires a variable of the type long, and that it will return a value of type long. The variables to be passed to the function are called *arguments*, and they are enclosed in parentheses following the function's name. In this example, the function's argument is long x. The keyword before the name of the function indicates the type of variable the function returns. In this case, a type long variable is returned. Line 12 calls the function cube and passes the variable input to it as the function's argument. The function's return value is assigned to the variable answer. Notice that both input and answer are declared on line 6 as long variables, in keeping with the function prototype on line 4.The function itself is called the *function definition*. In this case, it's called cube and is contained in lines 21 through 27. Like the prototype, the function definition has several parts. The function starts out with a function header on line 21. The *function header* is at the start of a function, and it gives the function's name (in this case, the name is cube). The header also gives the function's return type and describes its arguments. Note that the function header is identical to the function prototype (minus the semicolon). The body of the function, lines 22 through 27, is enclosed in braces. The body contains statements, such as on line 25, that are executed whenever the function is called. Line 23 is a variable declaration that looks like the declarations you have seen before, with one difference: it's local. *Local* variables are declared within a function body. (Local declarations are discussed later) Finally, the function concludes with a return statement on line 26, which signals the end of the function. A return statement also passes a value back to the calling program. In this case, the value of the variable x_cubed is returned. If you compare the structure of the cube() function with that of the main() function, you'll see that they are the same. main() is also a function. Other functions that you already have used are printf() and scanf(). Although printf() and scanf() are library functions (as opposed to user-defined functions), they are functions that can take arguments and return values just like the functions you create.

How a Function Works

A C program doesn't execute the statements in a function until the function is called by another part of the program. When a function is called, the program can send the function information in the form of one or more arguments. An *argument* is program data needed by the function to perform its task. The statements in the function then execute, performing whatever task each was designed to do. When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

Figure shows a program with three functions, each of which is called once. Each time a function is called, execution passes to that function. When the function is finished, execution passes back to the place from which the function was called. A function can be called as many times as needed, and functions can be called in any order.

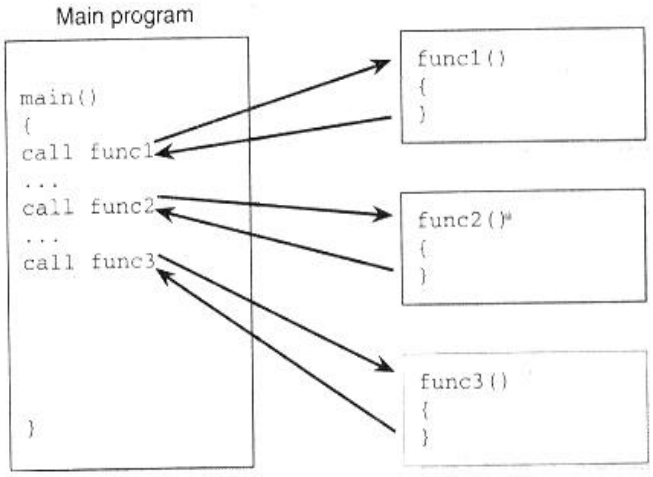
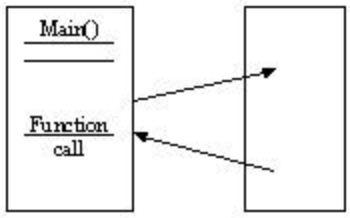
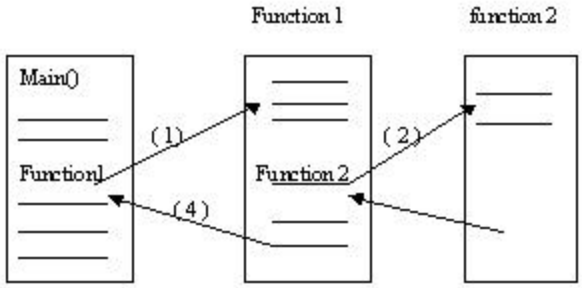


Figure *When a program calls a function, execution passes to the function and then back to the calling program.*

You now know what a function is and the importance of functions. Lessons on how to create and use your own functions follow.



But there is no restriction that the main only should call a function. A function can call another, that another and so on. See figure below:



The number indicate the sequence of control flow:

When a program calls a function, execution passes to the function and then back to the calling program.

You now know what a function is and the importance of functions. Lessons on how to create and use your own functions follow.

Functions

Function Prototype

return_type function_name(*arg-type name-1*,...,*arg-type name-n*);

Function Definition

return_type function_name(*arg-type name-1*,...,*arg-type name-n*)

{

/* *statements*; */

}

A *function prototype* provides the compiler with a description of a function that will be defined at a later point in the program. The prototype includes a return type indicating the type of variable that the function will return. It also includes the function name, which should describe what the function does. The prototype also contains the variable types of the arguments (arg type) that will be passed to the function. Optionally, it can contain the names of the variables that will be passed. A prototype should always end with a semicolon.

A *function definition* is the actual function. The definition contains the code that will be executed. The first line of a function definition, called the *function header*, should be identical to the function prototype, with the exception of the semicolon. A function header shouldn't end with a semicolon. In addition, although the argument variable names were optional in the prototype, they must be included in the function header. Following the header is the function body, containing the statements that the function will perform. The function body should start with an opening bracket and end with a closing bracket. If the function return type is anything other than void, a return statement should be included, returning a value matching the return type.

Function Prototype Examples

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Function Definition Examples

```
double squared( double number ) /* function header */
{ /* opening bracket */
return( number * number ); /* function body */
} /* closing bracket */
void print_report( int report_number )
{
if( report_number == 1 )
puts( "Printing Report 1" );
else
puts( "Not printing Report 1" );
}
```

Functions and Structured Programming

By using functions in your C programs, you can practice *structured programming*, in which individual program tasks are performed by independent sections of program code. "Independent sections of program code" sounds just like part of the definition of functions given earlier, doesn't it? Functions and structured programming are closely related.

The Advantages of Structured Programming

Why is structured programming so great?
There are two important reasons:

- It's easier to write a structured program, because complex programming problems are broken into a number of smaller, simpler tasks. Each task is performed by a function in which code and variables are isolated from the rest of the program. You can progress more quickly by dealing with these relatively simple tasks one at a time.
- It's easier to debug a structured program. If your program has a *bug* (something that causes it to work improperly), a structured design makes it easy to isolate the problem to a specific section of code (a specific function).

A related advantage of structured programming is the time you can save. If you write a function to perform a certain task in one program, you can quickly and easily use it in another program that needs to execute the same task. Even if the new program needs to accomplish a slightly different task, you'll often find that modifying a function you created earlier is easier than writing a new one from scratch. Consider how much you've used the two functions printf() and scanf() even though you probably haven't seen the code they contain. If your functions have been created to perform a single task, using them in other programs is much easier.

Planning a Structured Program

If you're going to write a structured program, you need to do some planning first. This planning should take place before you write a single line of code, and it usually can be done with nothing more than pencil and paper. Your plan should be a list of the specific tasks your program performs. Begin with a global idea of the program's function. If you were planning a program to manage your name and address list,

- what would you want the program to do? Here are some obvious things:
- Enter new names and addresses.
- Modify existing entries.
- Sort entries by last name.
- Print mailing labels.

With this list, you've divided the program into four main tasks, each of which can be assigned to a function. Now you can go a step further, dividing these tasks into subtasks. For example, the "Enter new names and addresses" task can be subdivided into these subtasks:

- Read the existing address list from disk.
- Prompt the user for one or more new entries.

- Add the new data to the list.
- Save the updated list to disk.

Likewise, the "Modify existing entries" task can be subdivided as follows:

- Read the existing address list from disk.
- Modify one or more entries.
- Save the updated list to disk.

You might have noticed that these two lists have two subtasks in common—the ones dealing with reading from and saving to disk. You can write one function to "Read the existing address list from disk," and that function can be called by both the "Enter new names and addresses" function and the "Modify existing entries" function. The same is true for "Save the updated list to disk."

Already you should see at least one advantage of structured programming. By carefully dividing the program into tasks, you can identify parts of the program that share common tasks. You can write "double-duty" disk access functions, saving yourself time and making your program smaller and more efficient.

This method of programming results in a *hierarchical,* or layered, program structure.

Figure illustrates hierarchical programming for the address list program.

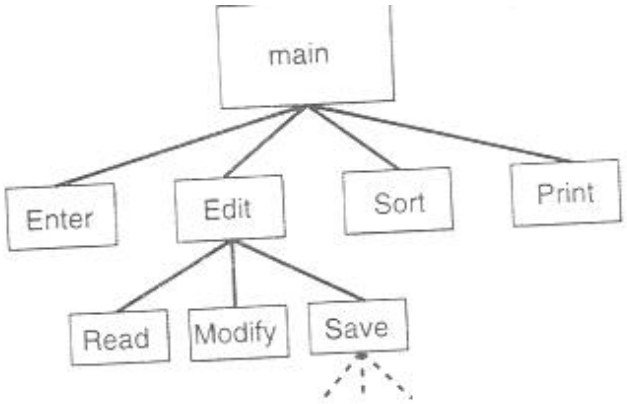


Figure *A structured program is organized hierarchically.*

When you follow this planned approach, you quickly make a list of discrete tasks that your program needs to perform. Then you can tackle the tasks one at a time, giving all your attention to one relatively simple task. When that function is written and working properly, you can move on to the next task. Before you know it, your program starts to take shape.

The Top-Down Approach

By using structured programming, C programmers take the *top-down approach,* where the program's structure resembles an inverted tree. Many times, most of the real work of the program is performed by the functions at the tips of the "branches." The functions closer to the "trunk" primarily direct program execution among these functions.

As a result, many C programs have a small amount of code in the main body of the program—that is, in main(). The bulk of the program's code is found in functions. In main(), all you might find are a few dozen lines of code that direct program execution among the functions. Often, a menu is presented to the person using the program. Program execution is branched according to the user's choices. Each branch of the menu uses a different function.

This is a good approach to program design. Previously we have seen how you can use the switch statement to create a versatile menu-driven system. Now that you know what functions are and why they're so important, the time has come for you to learn how to write your own.

- **DO** plan before starting to code. By determining your program's structure ahead of time, you can save time writing the code and debugging it.
- **DON'T** try to do everything in one function. A single function should perform a single task, such as reading information from a file.

Notes

LESSON 12 Writing a Function

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to write a function.
- Know what is the difference between arguments and parameters..
- Know what a local variable is?

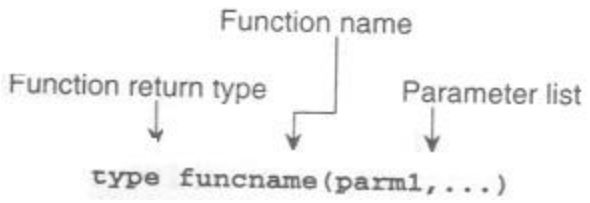
Writing a Function

The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function aren't particularly difficult.

The Function Header

The first line of every function is the function header, which has three components, each serving a specific function.

The three components of a function header.



The Function Return Type

The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types:

char, int, long, float, or double.

You can also define a function that doesn't return a value by using a return type of void. Here are some examples:

int *func1*(...) /* Returns a type int. */

float *func2*(...) /* Returns a type float. */

void *func3*(...) /* Returns nothing. */

The Function Name

You can name a function anything you like, as long as you follow the rules for C variable names. A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.

The Parameter List

Many functions use *arguments*, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect—the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list. For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter. For example, here's the header from the function specified in the first program of lesson function:

long cube(long x)

The parameter list consists of long x, specifying that this function takes one type long argument, represented by the parameter x. If there is more than one parameter, each must be separated by a comma. The function header

void *func1*(int x, float y, char z)

specifies a function with three arguments: a type int named x, a type float named y, and a type char named z. Some functions take no arguments, in which case the parameter list should consist of void, like this:

void *func2*(void)

NOTE: You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler will generate an error message.

Sometimes confusion arises about the distinction between a parameter and an argument.

A *parameter* is an entry in a function header; it serves as a "placeholder" for an argument. A function's parameters are fixed; they do not change during program execution.

An *argument* is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. In C, a function must be passed the same number and type of arguments each time it's called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name.

An example will make this clearer. Program given below presents a very simple program with one function that is called twice.

The difference between arguments and parameters:

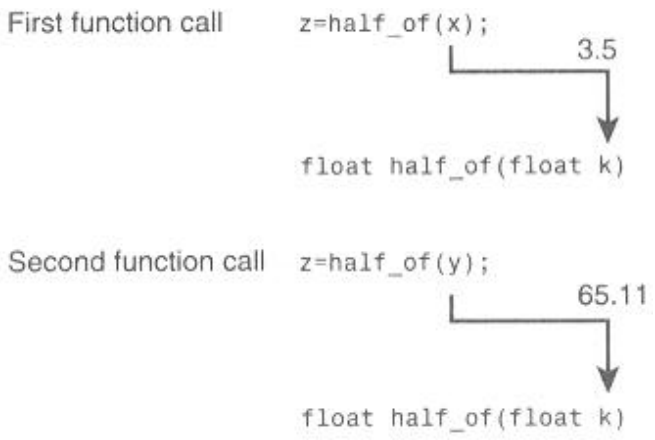
```
1: /* Illustrates the difference between arguments and parameters. */
2:
3: #include <stdio.h>
4:
5: float x = 3.5, y = 65.11, z;
6:
7: float half_of(float k);
8:
9: main()
10: {
11: /* In this call, x is the argument to half_of(). */
12: z = half_of(x);
13: printf("The value of z = %f\n", z);
14:
15: /* In this call, y is the argument to half_of(). */
16: z = half_of(y);
17: printf("The value of z = %f\n", z);
```

```
18:
19: return 0;
20: }
21:
22: float half_of(float k)
23: {
24: /* k is the parameter. Each time half_of() is called, */
25: /* k has the value that was passed as an argument. */
26:
27: return (k/2);
28: }
```

Output

```
The value of z = 1.750000
The value of z = 32.555000
```

Figure shows the relationship between argument and parameters.



Analysis

Looking at above program , you can see that the half_of() function prototype is declared on line 7. Lines 12 and 16 call half_of(), and lines 22 through 28 contain the actual function. Lines 12 and 16 each send a different argument to half_of(). Line 12 sends x, which contains a value of 3.5, and line 16 sends y, which contains a value of 65.11. When the program runs, it prints the correct number for each. The values in x and y are passed into the argument k of half_of(). This is like copying the values from x to k, and then from y to k. half_of() then returns this value after dividing it by 2 (line 27).

- **DO** use a function name that describes the purpose of the function.
- **DON'T** pass values to a function that it doesn't need.
- **DON'T** try to pass fewer (or more) arguments to a function than there are parameters.

In C programs, the number of arguments passed must match the number of parameters.

The Function Body

The *function body* is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Local Variables

You can declare variables within the body of a function. Variables declared in a function are called *local variables*. The term *local* means that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This will be explained shortly; for now, you should learn how to declare local variables. A local variable is declared like any other variable, using the same variable types and rules for names. Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function. Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
int a, b = 10;
float rate;
double cost = 12.55;
/* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available. When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name

A demonstration of local variables.

```
1:  /* Demonstrates local variables. */
2:
3:  #include<stdio.h>
4:
5:  int x = 1, y = 2;
6:
7:  void demo(void);
8:
9:  int main()
10: {
11: printf("\nBefore calling demo() , x = %d and y = %d.", x,
    y);
12: demo();
13: printf("\nAfter calling demoO, x = %d and y = %d\n.", x,
    y};
14:
15: return 0;
16: }
17:
18: void demo(void)
```

```
19: {
20: /* Declare and initialize two local variables. */
21:
22: int x = 88, y = 99;
23:
24: /* Display their values. */
25:
26: printf("\nWithin demo() , x = %d and y = %d.", x, y);
27: }
```

Output

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

Analysis

Listing 5.3 is similar to the previous programs in this chapter. Line 5 declares variables x and y.

These are declared outside of any functions and therefore are considered global. Line 7 contains the prototype for our demonstration function, named demo(). It doesn't take any parameters, so it has void in the prototype. It also doesn't return any values, giving it a type of void. Line 9 starts our main() function, which is very simple.

First, printf() is called on line 11 to display the values of x and y, and then the demo() function is called. Notice

that demo() declares its own local versions of x and y on line 22. Line 26 shows that the local variables take

precedence over any others. After the demo function is called, line 13 again prints the values of x and y.

Because you are no longer in demo(), the original global values are printed.

As you can see, local variables x and y in the function are totally independent from the global variables x and y declared outside the function. Three rules govern the use of variables in functions:

- To use a variable in a function, you must declare it in the function header or the function body (except for global variables).
- In order for a function to obtain a value from the calling program, the value must be passed as an argument.
- In order for a calling program to obtain a value from a function, the value must be explicitly returned from the function.

Keeping the function's variables separate from other program variables is one way in which functions are independent. A function can perform any sort of data manipulation you want, using its own set of local variables. There's no worry that these manipulations will have an unintended effect on another part of the program.

Function Statements

There is essentially no limitation on the statements that can be included within a function. The only thing you can't do inside a function is define another function. You can, however, use all other C statements, including loops, if statements, and assignment statements. You can call library functions and other user-defined functions.

What about function length?

C places no length restriction on functions, but as a matter of practicality, you should keep your functions relatively short. Remember that in structured programming, each function is supposed to perform a relatively simple task. If you find that a function is getting long, perhaps you're trying to perform a task too complex for one function alone. It probably can be broken into two or more smaller functions.

How long is too long? There's no definite answer to that question, but in practical experience it's rare to find a function longer than 25 or 30 lines of code. You have to use your own judgment. Some programming tasks require longer functions, whereas many functions are only a few lines long. As you gain programming experience, you will become more adept at determining what should and shouldn't be broken into smaller functions.

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
int x;
/* Function code goes here... */
return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements can be an efficient way to return different values from a function, as demonstrated in the program given below.

Using multiple return statements in a function

```
1: /* Demonstrates using multiple return statements in a function. */
2:
3: #include <stdio.h>
4:
5: int x, y, z;
6:
7: int larger_of( int , int );
8:
9: main()
10: {
```

```
11: puts("Enter two different integer values: ");
12: scanf("%d%d", &x, &y);
13:
14: z = larger_of(x,y);
15:
16: printf("\nThe larger value is %d.", z);
17:
18: return 0;
19: }
20:
21: int larger_of( int a, int b)
22: {
23: if (a > b)
24: return a;
25: else
26: return b;
27: }
```

Output

```
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

Analysis

As in other examples, the program starts with a comment to describe what the program does (line 1). The STDIO.H header file is included for the standard input/output functions that allow the program to display information to the screen and get user input. Line 7 is the function prototype for larger_of(). Notice that it takes two int variables for parameters and returns an int. Line 14 calls larger_of() with x and y. The function larger_of() contains the multiple return statements. Using an if statement, the function checks to see whether a is bigger than b on line 23. If it is, line 24 executes a return statement, and the function immediately ends. Lines 25 and 26 are ignored in this case. If a isn't bigger than b, line 24 is skipped, the else clause is instigated, and the return on line 26 executes. You should be able to see that, depending on the arguments passed to the function larger_of(), either the first or the second return statement is executed, and the appropriate value is passed back to the calling function. One final note on this program. Line 11 is a new function that you haven't seen before. puts()—meaning *putstring*—is a simple function that displays a string to the standard output, usually the computer screen.

Remember that a function's return value has a type that is specified in the function header and function prototype. The value returned by the function must be of the same type, or the compiler generates an error message.

Note

Structured programming suggests that you have only one entry and one exit in a function. This means that you should try to have only one return statement within your function. At times, however, a program might be much easier to read and maintain with more than one return statement. In such cases, maintainability should take precedence.

Notes

LESSON 13 Function Prototype & Recursive Function

Objectives

Upon completion of this Lesson, you should be able to:

- Know what a function prototype is?
- Know how to call functions.
- Know what recursion is?

The Function Prototype

A program must include a prototype for each function it uses.

What is a function prototype, and why is it needed?

You can see from the earlier examples that the prototype for a function is identical to the function header, with a semicolon added at the end. Like the function header, the function prototype includes information about the function's return type, name, and parameters. The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code

calls the function and verify that you're passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message. Strictly speaking, a function prototype doesn't need to exactly match the function header. The parameter names can be different, as long as they are the same type, number, and in the same order. There's no reason for the header and prototype not to match; having them identical makes source code easier to understand. Matching the two also makes writing a program easier. When you complete a function definition, use your editor's cut-and-paste feature to copy the function header and create the prototype. Be sure to add a semicolon at the end.

Where should function prototypes be placed in your source code?

They should be placed before the start of main() or before the first function is defined. For readability, it's best to group all prototypes in one location.

- **DON'T** try to return a value that has a type different from the function's type.
- **DO** use local variables whenever possible.
- **DON'T** let functions get too long. If a function starts getting long, try to break it into separate, smaller tasks.
- **DO** limit each function to a single task.
- **DON'T** have multiple return statements if they aren't needed. You should try to have one return when possible; however, sometimes having multiple return statements is easier and clearer.

Passing Arguments to a Function

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype. For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments—no more, no less—and no other type. If you try to pass a function an incorrect number and/or type of argument, the compiler will detect it, based on the information in the function prototype.

If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on.

Calling Functions

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement, as in the following example. If the function has a return value, it is discarded.

```
wait(12);
```

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. You've already seen an expression with a return value used as the right side of an assignment statement. Here are some more examples.

In the following example, half_of() is a parameter of a function:

```
printf("Half of %d is %d.", x, half_of(x));
```

First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

In this second example, multiple functions are being used in an expression:

```
y = half_of(x) + half_of(z);
```

Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions. Here, a function is being used with the if statement:

```
if ( half_of(x) > 10 )
{
/* statements; */ /* these could be any statements! */
}
```

If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned

value doesn't meet the criteria, the if's statements are not executed.

The following example is even better:

```
if ( do_a_process() != OKAY )
{
/* statements; */ /* do error routine */
}
```

Again, I haven't given the actual statements, nor is do_a_process() a real function; however, this is an important example that checks the return value of a process to see whether it ran all right. If it didn't, the statements take care of any error handling or cleanup. This is commonly used with accessing information in files, comparing values, and allocating memory. If you try to use a function with a void return type as an expression, the compiler generates an error message.

- **DO** pass parameters to functions in order to make the function generic and thus reusable.
- **DO** take advantage of the ability to put functions into expressions.
- **DON'T** make an individual statement confusing by putting a bunch of functions in it.

You should put functions into your statements only if they don't make the code more confusing.

Recursion

The term *recursion* refers to a situation in which a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program given below uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

Using a recursive function to calculate factorials.

```
1: /* Demonstrates function recursion. Calculates the */
2: /* factorial of a number. */
3:
4: #include <stdio.h>
5:
6: unsigned int f, x;
7: unsigned int factorial(unsigned int a);
8:
9: main()
10: {
11: puts("Enter an integer value between 1 and 8: ");
12: scanf("%d", &x);
13:
14: if( x > 8 || x < 1)
15: {
16: printf("Only values from 1 to 8 are acceptable!");
17: }
18: else
19: {
20: f = factorial(x);
21: printf("%u factorial equals %u\n", x, f);
22: }
23:
24: return 0;
25: }
26:
27: unsigned int factorial(unsigned int a)
28: {
29: if (a == 1)
30: return 1;
31: else
32: {
33: a *= factorial(a-1);
34: return a;
35: }
36: }
```

Output

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

Analysis

The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 and 2. On line 4, the appropriate header file is included for the input/output routines. Line 6 declares a couple of unsigned integer values. Line 7 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 9 through 25 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8 and then accept an entered value. Lines 14 through 22 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it's greater than 8, an error message is printed; otherwise, the program figures the factorial on line 20 and prints the result on line 21. When you know there could be a

problem, such as a limit on the size of a number, add code to detect the problem and prevent it. Our recursive function, factorial(), is located on lines 27 through 36. The value passed is assigned to a. On line 29, the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 29 is true. If the value of the factorial is 3, the factorial is evaluated to the following:

3 * (3-1) * ((3-1)-1)

- **DO** understand and work with recursion before you use it.
- **DON'T** use recursion if there will be several iterations. (An iteration is the repetition of a program statement.) Recursion uses many resources, because the function has to remember where it is.

Where the Functions Belong

You might be wondering where in your source code you should place your function definitions. For now, they should go in the same source code file as main() and after the end of main(). Figure 5.6 shows the basic structure of a program that uses functions.You can keep your user-defined functions in a separate source-code file, apart from main(). This technique is useful with large programs and when you want to use the same set of functions in more than one program.

Summary

This lesson introduced you to functions, an important part of C programming. Functions are independent sections of code that perform specific tasks. When your program needs a task performed, it calls the function that performs that task. The use of functions is essential for structured programming—a method of program design that emphasizes a modular, top-down approach. Structured programming creates more efficient programs and also is much easier for you, the programmer, to use. You also learned that a function consists of a header and a body. The header includes information about the function's return type, name, and parameters. The body contains local variable declarations and the C statements that are executed when the function is called. Finally, you saw that local variables—those declared within a function—are totally independent of any other program variables declared elsewhere.

Q&A

Q What if I need to return more than one value from a function?

A Many times you will need to return more than one value from a function, or, more commonly, you will want to change a value you send to the function and keep the change after the function ends. This process is covered on Day 18, "Getting More from Functions."

Q How do I know what a good function name is?

A A good function name describes as specifically as possible what the function does.

Q When variables are declared at the top of the listing, before main(), they can be used anywhere, but local variables can be used only in the specific function. Why not just declare everything before main()?

A Variable scope is discussed in more detail on Day 12.

Q What other ways are there to use recursion?

A The factorial function is a prime example of using recursion. The factorial number is needed in many statistical calculations. Recursion is just a loop; however, it has one difference from other loops. With recursion, each time a recursive function is called, a new set of variables is created. This is not true of the other loops that you will learn about in the next lesson.

Q Does main() have to be the first function in a program?

A No. It is a standard in C that the main() function is the first function to execute; however, it can be placed anywhere in your source file. Most people place it first so that it's easy to locate.

Q What are member functions?

A Member functions are special functions used in C++ and Java. They are part of a class—which is a special type of structure used in C++ and Java.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Will you use structured programming when writing your C programs?
2. How does structured programming work?
3. How do C functions fit into structured programming?
4. What must be the first line of a function definition, and what information does it contain?
5. How many values can a function return?
6. If a function doesn't return a value, what type should it be declared?
7. What's the difference between a function definition and a function prototype?
8. What is a local variable?
9. How are local variables special?
10. Where should the main() function be placed?

Exercises

1. Write a header for a function named do_it() that takes three type char arguments and returns a type float to the calling program.
2. Write a header for a function named print_a_number() that takes a single type int argument and doesn't return anything to the calling program.
3. What type value do the following functions return?

   a. int print_error(float err_nbr);

   b. long read_record(int rec_nbr, int size);

4. **BUG BUSTER:** What's wrong with the following listing?

```
#include <stdio.h>
void print_msg( void );
main()
{
print_msg( "This is a message to print" );
return 0;
}
void print_msg( void )
{
puts( "This is a message to print" );
return 0;
}
```

5. **BUG BUSTER:** What's wrong with the following function definition?

```
int twice(int y);
{
return (2 * y);
}
```

6. Write a function that receives two numbers as arguments and returns the value of their product.
7. Write a function that receives two numbers as arguments. The function should divide the first number by the second. Don't divide by the second number if it's zero. (Hint: Use an if statement.)
8. Write a function that calls the functions in exercises 7 and 8.
9. Write a program that uses a function to find the average of five type float values entered by the user.
10. Write a recursive function to take the value 3 to the power of another number. For example, if 4 is passed, the function will return 81.

Arrays

Arrays are a type of data storage that you often use in C programs.

Today you will learn

- What an array is?
- The definition of single- and multidimensional numeric arrays.

How to declare and initialize arrays.

Notes

LESSON 14 Introduction to Arrays

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with arrays?
- Know what a single dimensional array is?
- Know how to work with multidimensional arrays.

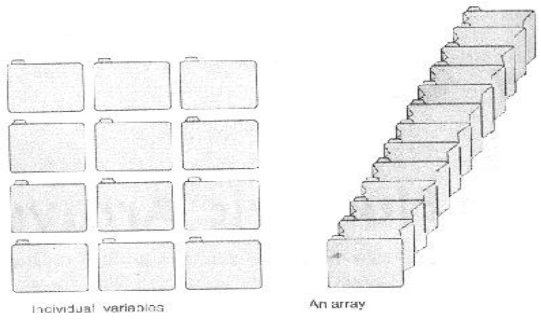
What Is an Array?

An *array* is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an *array element*.

Why do you need arrays in your programs?

This question can be answered with an example. If you're keeping track of your business expenses for 1998 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments. Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments.

Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments. The following figure illustrates the difference between using individual variables and an array.



Variables are like individual folders, whereas an array is like a single folder with many *compartments.*

So far, our initial attempt will be to create a specific variable for each user. This might look like,

```
int name1 = 101;
int name2 = 232;
int name3 = 231;
```

It becomes increasingly more difficult to keep track of this as the number of variables increase. Arrays offer a solution to this problem.

An array is a multi-element box, a bit like a filing cabinet, and uses an indexing system to find each variable stored within it. In C, indexing starts at **zero**.

Arrays, like other variables in C, must be declared before they can be used.

The replacement of the above example using arrays looks like,

```
int names[4];
names[0] = 101;
names[1] = 232;
names[2] = 231;
names[3] = 0;
```

We created an array called *names*, which has space for four integer variables. You may also see that we stored 0 in the last space of the array. This is a common technique used by C programmers to signify the end of an array.

Arrays have the following syntax, using square brackets to access each indexed value (called an **element**).

```
x[i]
```

so that *x[5]* refers to the sixth element in an array called *x*. In C, array elements start with 0. Assigning values to array elements is done by,

```
x[10] = g;
```

and assigning array elements to a variable is done by,

```
g = x[10];
```

In the following example, a character-based array named *word* is declared, and each element is assigned a character. The last element is filled with a zero value, to signify the end of the character string (in C, there is no string type, so character based arrays are used to hold strings). A printf statement is then used to print out all elements of the array.

```
/* Introducing array's, 2 */
#include <stdio.h>
main()
{
char word[20];
```

```
        word[0] = 'H';
        word[1] = 'e';
        word[2] = 'l';
        word[3] = 'l';
        word[4] = 'o';
        word[5] = 0;
        printf("The contents of word[] is —
>%s\n", word );
}
```

Sample Program Output

```
The contents of word[] is Hello
```

Single-Dimensional Arrays

A *single-dimensional array* has only a single subscript. A *subscript* is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C's data types can be used for arrays. C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11. In the preceding example, January's expense total would be stored in expenses[0], February's in expenses[1], and so on.

When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations as shown in figure .

The location of array declarations in your source code is important. As with nonarray variables, the declaration's location affects how your program can use the array. For now, place your array declarations with other variable declarations, just before the start of main(). An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):

```
expenses[1] = 89.95;
```

Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10]. When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a C integer variable or expression, or even another array element.

Here are some examples:

```
float expenses[100];
int a[10];
/* additional statements go here */
expenses[i] = 100; /* i is an integer variable */
expenses[2 + 3] = 100; /* equivalent to expenses[5] */
expenses[a[2]] = 100; /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing expenses[a[2]] has the same effect as writing

```
expenses[8];
```

When you use arrays, keep the element numbering scheme in mind: In an array of *n* ele-ments, the allowable subscripts range from 0 to *n*-1. If you use the subscript value *n*, you might get program errors. The C compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

Warning

Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9. Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through *n*. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows.

You could also store some related data in element 0 (the yearly expense total, perhaps).

```
float expenses[13];
```

the program expenses.c in below program demonstrates the use of an array. This is a simple program with no real practical use; however, it helps demonstrate the use of an array.

```
1:  /* expenses.c . Demonstrates use of an array */
2:
3:  #include <stdio.h>
4:
5:  /* Declare an array to hold expenses, and a counter variable
    *'
6:
7:  float expenses[13];
8:  int     count;
9:
10: int main()
11: {
12: /* Input data from keyboard into array */
13:
14: for (count = 1; count < 13; count++)
15: {
16: printf("Enter expenses for month %d: ", count);
```

```
17: scarnf("%f", &expenses [count]);
18: }
19:
20: /*Print array contents*/
21:
22: for (count =1; count < 13; count++)
23: {
24: printf("Month %d  =  $%.2f\n", count, expenses[count]);
25: }
26: return 0;
27: }
```

Analysis

When you run EXPENSES.C, the program prompts you to enter expenses for months 1 through 12. The values you enter are stored in an array. You must enter a value for each month. After the 12th value is entered, the array contents are displayed on-screen. The flow of the program is similar to listings you've seen before. Line 1 starts with a comment that describes what the program does. Notice that the name of the program, EXPENSES.C, is included. When the name of the program is included in a comment, you know which program you're viewing. This is helpful when you're reviewing printouts of a listing. Line 5 contains an additional comment explaining the variables that are being declared. In line 7, an array of 13 elements is declared. In this program, only 12 elements are needed, one for each month, but 13 have been declared. The for loop in lines 14 through 18 ignores element 0. This lets the program use elements 1 through12, which relate directly to the 12 months. Going back to line 8, a variable, count, is declared and is used throughout the program as a counter and an array index.

The program's main() function begins on line 10. As stated earlier, this program uses a for loop to print a message and accept a value for each of the 12 months. Notice that in line 17, the scanf() function uses an array element. In line 7, the **expenses array was declared as float, so %f is used. The** *address-of operator* (&) also is placed before the array element, just as if it were a regular type float variable and not an array element.

Lines 22 through 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the printf() function so that the expenses values print in a more orderly fashion. For now, know that %.2f prints a floating number with two digits to the right of the decimal.

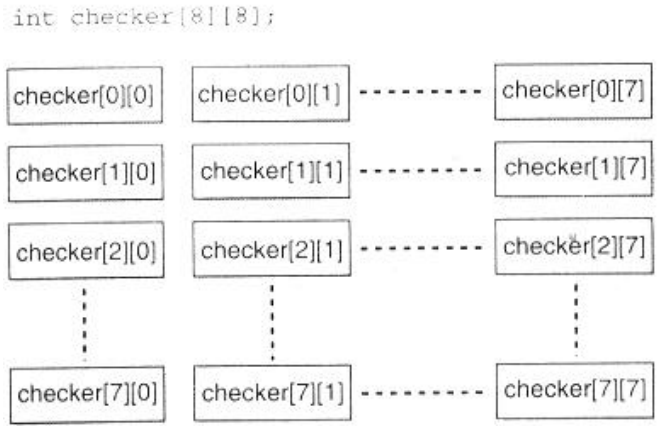
- **DON'T** forget that array subscripts start at element 0.
- **DO** use arrays instead of creating several variables that store the same thing. For example, if you want to store total sales for each month of the year, create an array with 12 elements to hold sales rather than creating a sales variable for each month.

Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts; a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have. (There *is* a limit on total array size, as discussed later in this chapter.) For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7]. The structure of this two-dimensional array is illustrated in Figure.



A two-dimensional array has a row-and-column structure.

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory.

Notes

LESSON 15 Naming and Declaring Arrays

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to name and declare an array.
- Know how to initialize an array.
- A demonstration of how an array program really works.

Naming and Declaring Arrays

The rules for assigning names to arrays are the same as for variable names. An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately following the array name. When you declare an array, you can specify the number of elements with a literal constant (as was done in the earlier examples) or with a symbolic constant created with the #define directive. Thus, the following:

```
#define MONTHS 12

int array[MONTHS];
```

is equivalent to this statement:

```
int array[12];
```

With most compilers, however, you can't declare an array's elements with a symbolic constant created with the const keyword:

```
const int MONTHS = 12;

int array[MONTHS]; /* Wrong! */
```

```
1:  /*grades.c Sample progr3mwith array */
2:  /* Get 10 grades and then average them */
3:
4:  #include<stdio.h>
5:
6:  #define MAX_GRADE 100
7:  #define STUDENTS 10
9:  int grades[STUDENTS];
10:
11: int idx;
12: int total = 0;     /* used for average */
13:
14: int main()
15: {
16: for( idx = 0; idx<STUDENTS; idx ++)
17: {
18: printf( "Enter Person %d's grade: ", idx +1);
19: scanf( "%d", &grades[idx]);
20:
21: while ( grades[idx] > MAX_GRADE)
22: {
23: printf( "\nThe highest grade possible is %d',
24  MAX_GRADE );
25: printf( "\nEnter correct grade: ");
26: scanf( "%d", &grades[idx] );
27: }
28:
29: total += grades[idx];
30: }
31:
32: printf("\n\nThe average score is %d\n", ( total /
   STUDENTS) ) ;
33:
34: return;
35: }
```

Input/ Output

```
Enter person 1's grade:95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade. 95
```

Analysis

Like EXPENSES.C, this listing prompts the user for input. It prompts for 10 people's grades. Instead of printing each grade, it prints the average score. As you learned earlier, arrays are named like regular variables. On line 9, the array for this program is named grades. It should be safe to assume that this array holds grades. On lines 6 and 7, two constants, MAX_GRADE and STUDENTS, are defined. These constants can be changed easily. Knowing that STUDENTS is defined as 10, you then know that the grades array has 10 elements. Two other variables are declared, idx and total. An abbreviation of *index*, idx is used as a counter and array subscript. A running total of all grades is kept in total.

The heart of this program is the for loop in lines 16 through 30. The for statement initializes idx to 0, the first subscript for an array. It then loops as long as idx is less than the number of

students. Each time it loops, it increments idx by 1. For each loop, the program prompts for a person's grade (lines 18 and 19). Notice that in line 18, 1 is added to idx in order to count the people from 1 to 10 instead of from 0 to 9. Because arrays start with subscript 0, the first grade is put in grade[0]. Instead of confusing users by asking for Person 0's grade,they are asked for Person 1's grade. Lines 21 through 27 contain a while loop nested within the for loop. This is an edit check that ensures that the grade isn't higher than the maximum grade, MAX_GRADE. Users are prompted to enter a correct grade if they enter a grade that is too high. You should check program data whenever you can. Line 29 adds the entered grade to a total counter. In line 32, this total is used to print the average score (total/STUDENTS).

- **DO** use #define statements to create constants that can be used when declaring arrays. Then you can easily change the number of elements in the array. In GRADES.C, for example, you could change the number of students in the #define, and you wouldn't have to make any other changes in the program.
- **DO** avoid multidimensional arrays with more than three dimensions. Remember, multidimensional arrays can get very big very quickly.

Initializing Arrays

You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0. For example, the following code assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3]:

```
int array[4] = { 100, 200, 300, 400 };
```

If you omit the array size, the compiler creates an array just large enough to hold the initialization values. Thus, the following statement would have exactly the same effect as the previous array declaration statement:

```
int array[] = { 100, 200, 300, 400 };
```

You can, however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If you don't explicitly initialize an array element, you can't be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

Initializing Multidimensional Arrays

Multidimensional arrays can also be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. For example:

```
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
array[3][1] is equal to 11
array[3][2] is equal to 12
```

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
                    { 7, 8, 9 } , { 10, 11, 12 } };
```

Remember, initialization values must be separated by a comma—even when there is a brace between them. Also, be sure to use braces in pairs—a closing brace for every opening brace—or the compiler becomes confused. Now look at an example that demonstrates the advantages of arrays. The program then displays the array elements on-screen. Imagine how many lines of source code you would need to perform the same task with non-array variables. You see a new library function, getch(), in this program. The getch() function reads a single character from the keyboard.

```
1:  /*random.c· Demonstrates using a multidimensional array
    *1
2
3:  include <stdio.h:>
4:  include <stdlib.h:>
5:  /* Declare a three-dimensional array with 1000 elements */
6
7:  int random_array[10][10][10];
8:  int a,b, c;
9
10: int main()
11: {
12: /* Fill the array with ralldom Ilumbers. The C library */
13: /*function rand() returns a random number. Use one */
14: /*for loop for each array subscript. *1
15:
16: for(a = 0; a<10; a++)
17: {
18: for (b = 0; b < 10; b++)
19:  {
20: for (c = 0; c < 10; c++)
21: {
22: random_array[a][b][C] = rand();
23: }
24: }
25: }
26:
27: /* Now display the array elements 10 at a time */
28:
29: for (a = 0; a < 10; a++)
30: {
```

```
31: for (b=0; b<10; b++)
32: {
33: for (e = 0; c < 10; e++)
34: {
35: printf("\n random_array[%d] [%d] [%dJ = " a, b, c);
36: printf("%d", random_array[a][b][c]);
37: }
38: printf(" \n Press Enter to continue, CTRL·C to quit. ");
39:
40: getchar();
41: }
42: }
43: return 0;
44: }      /* end of main()  */
```

Output

```
random_array[0][0][0]=346
random_array[0][0][1]=130
random_array[0][0][2] =10982
random_array[0][0][3] = 1090
random_array[0][0][4] =11656
random_array[0][0][5] =7117
random_array[0][0][6] =17595
random_array[0][0][7] =6415
random_array[0][0][8] =22948
random_array[0][0][9] = 31126
Press Enter to continue, CTRL·C to quit.
random_array[0][1][0] =9004
random_array[0][1][1] =14558
random_array[0][1][2] = 3571
random_array[0][1][3] = 22879
random_array[0][1][4] = 18492
random_array[0][1][5] = 1360
random_array[0][1][6] = 5412
random_array[0][1][7] = 26721
random_array[0][1][8] = 22463
random_array[0][1][9] = 25047
Press Enter to continue, CTRL·C to quit
random_array[9][8][0] =6287
random_array[9][8][1]=26957
random_array[9][8][2] = 1530
```

Analysis

We have already seen a program that used a nested for statement; this program has two nested for loops. Before you look at the for statements in detail, note that lines 7 and 8 declare four variables. The first is an array named random_array, used to hold random numbers. random_array is a three-dimensional type int array that is 10-by-10-by-10, giving a total of 1,000 type int elements (10 * 10 * 10). Imagine coming up with 1,000 unique variable names if you couldn't use arrays! Line 8 then declares three variables, a, b, and c, used to control the for loops.

This program also includes the header file STDLIB.H (for standard library) on line 4. It is included to provide the prototype for the rand() function used on line 22.

The bulk of the program is contained in two nests of for statements. The first is in lines 16 through 25, and the second is in lines 29 through 42. Both for nests have the same structure. They work just like the loops in Listing 6.2, but they go one level deeper. In the first set of for statements, line 22 is executed repeatedly. Line 22 assigns the return value of a function, rand(), to an element of the random_array array, where rand() is a library function that returns a random number.

Going backward through the listing, you can see that line 20 changes variable c from 0 to 9. This loops through the farthest right subscript of the random_array array. Line 18 loops through b, the middle subscript of the random array. Each time b changes, it loops through all the c elements. Line 16 increments variable a, which loops through the farthest left subscript. Each time this subscript changes, it loops through all 10 values of subscript b, which in turn loop through all 10 values of c. This loop initializes every value in the random array to a random number.

Lines 29 through 42 contain the second nest of for statements. These work like the previous for statements, but this loop prints each of the values assigned previously. After 10 are displayed, line 38 prints a message and waits for Enter to be pressed. Line 40 takes care of the keypress using getchar(). If Enter hasn't been pressed, getchar() waits until it is. Run this program and watch the displayed values.

Maximum Array Size

Because of the way memory models work, you shouldn't try to create more than 64 KB of data variables for now. An explanation of this limitation is beyond the scope of this book, but there's no need to worry: None of the programs in this book exceed this limitation. To understand more, or to get around this limitation, consult your compiler manuals. Generally, 64 KB is enough data space for programs, particularly the relatively simple programs you will write as you work through this book. A single array can take up the entire 64 KB of data storage if your program uses no other variables. Otherwise, you need to apportion the available data space as needed.

Note

Some operating systems don't have a 64 KB limit. DOS does. The size of an array in bytes depends on the number of elements it has, as well as each element's size. Element size depends on the data type of the array and your computer. The sizes for each numeric data type, given below. These are the data type sizes for many PCs.

Table Storage space requirements for numeric data types for many PCs.

Element Data Type	Element Size (Bytes)

To calculate the storage space required for an array, multiply the number of elements in the array by the element size. For

Element Data Type	Element Size (Bytes)
int	2 or 4
short	2
long	4
float	4
double	8

example, a 500-element array of type float requires storage space of 500 * 4 = 2000 bytes. You can determine storage space within a program by using C's sizeof() operator; sizeof() is a unary operator, not a function. It takes as its argument a variable name or the name of a data type and returns the size, in bytes, of its argument. The use of sizeof() is illustrated in the following program given below.

Using the sizeof() operator to determine storage space requirements for an array.

```
1:  /*'Demonstrates the sizeof() operator*/
2:
3:  include<stdio.h>
4:
5:  /*Declare several 100-element arrays */
6:
7: int intarray[100];
8: float floatarray[180];
9: double doublearray[100];
10:
11: int main();
12: {
13: /* Display the sizes of numeric data types */
14:
15: printf("\n\nSize of int = %d bytes", sizeof(int));
16: print1("\nSize of short = %d bytes", sizeof(short));
17: printf("\nSize of long = %d bytes", sizeof(long));
18: printf( "\nSize of float = %d bytes', sizeof(float)) j
19: print1("\nSize of double =%d bytes', sizeof(double));
20:
21: /*Displaythesizesofthethreearrays*/
22:
23: printf( "\nSize of intarray =%d bytes", sizeof(intarray));
24: printf("\nSize of floatarray = %d bytes",
25: sizeof(floatarray));
26: printf("\nSize of doublearray = %d bytes\n",
27: sizeof(doublearray));
28:
29: return 0;
31: }
```

The following output is from a 16-bit Windows 3.1 machine:

```
Size of int = 2 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 200 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

You would see the following output on a 32-bit Windows NT machine, as well as a 32-bit UNIX machine:

```
Size of int = 4 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 400 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Analysis

Enter and compile the program in this listing by using the procedures you learned on Day 1, "Getting Started with C." When the program runs, it displays the sizes—in bytes—of the three arrays and five numeric data types. On Day 3 you ran a similar program; however, this listing uses sizeof() to determine the storage size of arrays. Lines 7, 8, and 9 declare three arrays, each of different types. Lines 23 through 27 print the size of each array. The size should equal the size of the array's variable type times the number of elements. For example, if an int is 2 bytes, intarray should be 2 * 100, or 200 bytes. Run the program and check the values. As you can see from the output, different machines or operating systems might have different sized data types.

Summary

In this lesson we spoke about numeric arrays in detail, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage. Like non-array variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared.

Q&A

Q What happens if I use a subscript on an array that is larger than the number of elements in the array?

A If you use a subscript that is out of bounds with the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable.

This can be a difficult error to find once it starts causing problems, so make sure you're careful when initializing and accessing array elements.

Q What happens if I use an array without initializing it?

A This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize variables and arrays so that you know exactly what's in them. Day 12 introduces you to one exception to the need to initialize. For now, play it safe.

Q How many dimensions can an array have?

A As stated in this chapter, you can have as many dimensions as you want. As you add more dimensions, you use more data storage space. You should declare an array only as large as you need to avoid wasting storage space.

Q Is there an easy way to initialize an entire array at once?

A Each element of an array must be initialized. The safest way for a beginning C programmer to Initialize an array is either with a declaration, as shown in this chapter, or with a for statement. There are other ways to initialize an array, but they are beyond the scope of this book.

Q Can I add two arrays together (or multiply, divide, or subtract them)?

A If you declare two arrays, you can't add the two together. Each element must be added individually.

Q Why is it better to use an array instead of individual variables?

A With arrays, you can group like values with a single name. In Listing 8.3, 1,000 values were stored. Creating 1,000 variable names and initializing each to a random number would have taken a tremendous amount of typing. By using an array, you made the task easy.

Q What do you do if you don't know how big the array needs to be when you're writing the program?

A There are functions within C that let you allocate space for variables and arrays on-the-fly. These functions are covered on Day 15.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. Which of C's data types can be used in an array?
2. If an array is declared with 10 elements, what is the subscript of the first element?
3. In a one-dimensional array declared with *n* elements, what is the subscript of the last element?
4. What happens if your program tries to access an array element with an out-of-range subscript?
5. How do you declare a multidimensional array?
6. An array is declared with the following statement. How many total elements does the array have?

   ```
   int array[2][3][5][8];
   ```

7. What would be the name of the 10th element in the array in question 6?

Exercises

1. Write a C program line that would declare three one-dimensional integer arrays, named one, two, and three, with 1,000 elements each.
2. Write the statement that would declare a 10-element integer array and initialize all its elements to 1.
3. Given the following array, write code to initialize all the array elements to 88:

   ```
   int eightyeight[88];
   ```

4. Given the following array, write code to initialize all the array elements to 0:

   ```
   int stuff[12][10];
   ```

5. **BUG BUSTER:** What is wrong with the following code fragment?

   ```
   int x, y;
   int array[10][3];
   main()
   {
   for ( x = 0; x < 3; x++ )
   for ( y = 0; y < 10; y++ )
   array[x][y] = 0;
   return 0;
   }
   ```

6. **BUG BUSTER:** What is wrong with the following?

   ```
   int array[10];
   int x = 1;
   main()
   {
   for ( x = 1; x <= 10; x++ )
   array[x] = 99;
   return 0;
   }
   ```

7. Write a program that puts random numbers into a two-dimensional array that is 5 by 4.

   Print the values in columns on-screen. (Hint: Use the rand() function)

8. Write a program that initializes an array of 10 elements. Each element should be equal to its subscript. The program should then print each of the 10 elements.
9. Modify the program from exercise 9. After printing the initialized values, the program should copy the values to a new array and add 10 to each value. Then the new array values should be printed.

LESSON 16 Types of I/O & Console I/O Functions

Objectives

Upon completion of this Lesson, you should be able to:

- Know what are the different types of I/O.
- Know what a console I/O function is?.
- Know what a formatted console I/O function is?
- Know what conversion specification is?

Thus, C simply has no provision for receiving data from any of the input devices (like say keyboard, floppy etc.), nor for sending data to the output devices (like say VDU, floppy etc.). Then how do we manage I/O, and how is it that we were we able to use printf() and scanf() if C has nothing to offer for ***I/O?*** This is what we, intend to explore in this lesson.

Types of I/O

Though C has no provision for I/O, it of course has to be dealt with at some point or the other. There is not much use writing a program that spends all it's time telling itself a secret. Each Operating System has it's own facility for inputting and outputting data from and to the files and devices. It's a simple matter for a system programmer to write a few small programs that would link the C compiler for particular Operating system's I/O facilities. The developers of C Compilers do just that. They write several standard I/O functions and put them in libraries. Though these functions are not part of C's formal definition, they have become a standard feature of C language, and have been imitated more or less faithfully by every designer of a DOS based or a UNIX based C compiler. Whatever version of C you are using it's almost certain that you have access to such a library of I/O functions.

Do understand that the I/O facilities with different operating systems would be different. Thus, the way DOS displays output on screen would be different than the way UNIX does it. Fortunately, the printf( ) function that we use on a DOS based C compiler works even with a UNIX based C compiler. This is because, the standard library function printf() for DOS based C compiler has been written keeping in mind the way DOS outputs characters to screen. Similarly, the printf() function for a UNIX based compiler has been written Keeping in mind the way UNIX outputs characters to screen. We as users do not have to bother about which **printf()** has been written in what manner. We should just use **printf()** and it would take care of the rest of the details. Same is true about all other standard library functions available for I/O.

There are numerous library functions available for I/O. These can be classified into three broad categories:

a. Console I/O functions - functions to receive input from Keyboard and write output to VDU.

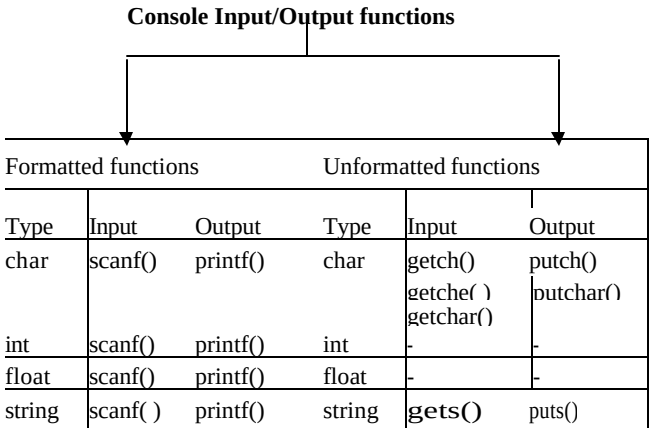
b. Disk 1/0 functions - functions to perform I/O operations on a floppy disk or hard disk.

c. Port I/O functions - Functions to perform I/O opera-tions on various ports.

Let us now take a close look at each of these categories of I/O functions.

Console I/O Functions

Console I/O functions can be further classified into two categories: formatted and unformatted console I/O functions. The basic dif-ference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points etc. can be controlled using for-matted functions. The functions available under each of these two categories are shown in Figure. Now let us discuss these console I/O functions in detail.



Figure

Console keyboard device are described as either standard or extended, based on the number and configuration of keys, while video display device typically are classed as either mono-chrome or colour. Because the DOS device CON is by default the predefined standard input (stdin) and output (stdout) device, all functions described in this chapter, except_bios_ keyboard (), may be subject to redirection or piping from the DOS command line.

Even through console I/O is affected by redirection.

Formatted Console I/O Functions

As can be seen from the above figure the functions printf(), and scanf() fall under the category of formatted console I/O functions. These functions allow us to supply the input in a

fixed format and let us obtain the output in the specified form. Let us discuss these functions one by one.

We have talked a lot about printf(), used it regularly, but without having introduced it formally. Well, better late than never. It's general form looks like this ...

printf ("format string", list of variables);

The format string can contain:

1. Characters that are simply printed as they are Conversion specifications that begin with a % sign Escape sequences that begin with a \ sign

For example, look at the following program:

```
main()
{
 int avg = 346;
 float per = 69.2 ;
 printf ("Average = %d\n Percentage = %f ", avg, per) :
}
```

The output of the program would be ...

Average = 346

Percentage = 69.200000

How does **printf()** function interpret the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a **%** or a **** it continues to dump the characters that it encounters, on to the screen. In this example **Average =** is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format. In this example, the moment **%d** is met the variable **avg** is picked up and its value is printed. Similarly, the moment an escape sequence is met it takes the appropriate action. In this example, the moment **\n** is met it places the cursor at the beginning of the next line. This process continues till the end of format string is reached.

Conversion Specifications

The **%d** and **%f** used in the printf() are called conversion characters. They tell printf() to print the value of avg as a decimal integer and the value of per as a float. Following is the list of conversion characters that can be used with the printf( ) funcion.

Data type		Conversion character
Integer	Short signed	%d or %i
	Short unsigned	%u
	Long signed	%ld
	Long unsigned	%Iu
	Unsigned hexadecimal	%x
	Unsigned octal	%0
Real	Float	%f
	Double	%If
Character	Signed character	%c
	Unsigned character	%c
String		%s

We can provide following optional specifiers in the conversion specifications.

Specifier	Description
dd	Digits specifying field width
.	Decimal point separating field width from precision (Precision stands for the number of places after the Decimal point)
dd	Digits specifying precision
–	Minus sign for left justifying the output in the specified Field width.

Now a short explanation about these conversion specifications. The field width specifier tells printf( ) how many columns on screen should be used while printing a value. For example, %10d says, "print the variable as a decimal integer in a field of 10 columns." If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in conversion specification (as in %-10d), this means left justification is desired and the value will be padded with blanks on the right. Here is an example, which should make this point clear.

```
main()
{
int weight = 63;
printf ( '\nweight is %d kg", weight) ;
printl ("\nweight is %2d kg", weight) ;
```

```
printl ( '\nweight is %4d kg', weight) ;
printl ( "\nweight is %6d kg', weight) ;
printl ( '\nweight is %-6d kg', weight) ;
}
```

The output of the program would look like this ...

```
                             1          2    3
Columns         1234567890123456789012345678901234567890

weight is 63 kg
weight is 63 kg
weight is 63 kg
                weight is 63 kg
                weight is 63 kg
```

Specifying the field width can be useful in creating tables of numeric values, as the following program demonstrates.

```
main()
{
printf ("\n %f%f %f, 5.0,13.5,133.9);
printf ('\n %f%l%f, 305.0,1200.9,3005.3);
}
```

And here is the output ...

```
5.000000 13.500000 133.900000
305.000000 1200.900000 3005.100000
```

Even though the numbers have been printed, the numbers have not been lined up properly .nd hence are hard to read. A better way would be something like this ...

```
main()
{
printf("ln%10.1f%10.1f%10.1f, 5.0, 13.5, 133.9);
printf("\n%10.1f%10.11 %10.11', 305.0,1200.9,3005.3);
}
```

This results into a much better output...

```
0123456789012345678901234567890 1
        5.0      13.5     133.9
        305.0    1200.9   3005.3
```

The conversion specifications could be used even while displaying string of characters. The following program would clarify this point:

```
/* Formatting string with printf() */
main()
{
 char firstname1[ ]= "Sandy' ;
 char surname1 [ ] = 'Malya";
 char firstname2[ ] = 'AjayKumar" ;
 char surname2[ ] = "Gurubaxani' ;
 printf ( "'\n%2Os%20s", firstname1, surname1 ) ;
 printf ( "\n%2Os%20s", firstname2, surname2) ;
}
```

and here's the output….

```
01234567890123456789012345678901234567890123456789 0
             Sandy                   Malya
AjayKumar          Gurubaxani
```

The conversion specification **%20s** reserves 20 columns for printing a string and then prints the string in these 20 columns with right justification. This helps lining up names of different lengths properly. Obviously, the format **%-20s** would have left justified the string.

Notes

LESSON 17 Escape Sequences

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with escape sequences.
- Know how sprintf() and sscanf() function works.
- Know how unformatted console I/O and formatted I/O works.

Escape Sequences

We saw earlier how the newline character**, \n**, when inserted in a **printf( )'**s format string, takes the cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol **(\)** is considered an 'escape' character: it causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

The following example shows usage of \n and a new escape sequence **\t**, called 'tab'. A **\t** moves the cursor to the next tab stop. A 80 column screen usually has 10 tab stops. In other words, the screen is divided into 10 zones of 8 columns each. Printing a tab takes the cursor to the beginning of next printing zone. For example, if cursor is positioned in column 5, then printing a lab takes it 10 column 8.

```
main()
{
 printf(" you\tmust\tbe\tcrazy\nto\thate\tthis\tbook" );
}
```

Output

```
                1       2       3       4
01234567890123456789012345678901234567890
You     must    be      crazy
to      hate    this    book
```

The cursor jumps over eight columns when it encounters the **'\t'** character. This is another useful technique for lining up columns of output. Most compilers allow the user to change the number of print zones available on the screen, thereby changing the tab width. The **\n** character causes a new line to begin following 'crazy'. The tab and newline are probably the most commonly used escape se-quences, but there are others as well.

The following figure shows a complete list of these escape sequences.

Esc.Seq.	Purpose	Esc.Seq.	Purpose
\n	Newline	\t	Tab
\b	Backspace	\r	Carriage return
\f	Fonn feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

The first few of these escape sequences are more or less self-explanatory. \b moves the cursor-one position to the left of its current position. \r takes the cursor to the beginning of the line in which it is currently placed. \3 alerts the user by sounding the speaker inside the computer. Form feed advances the computer stationery attached to the printer to the top of the next page. Characters that are ordinarily used as delimiters ... the single quote, double quote, and the backslash can be printed by preceding them with the backslash. Thus, the statement,

```
printf ("He said, \"Let's do it!\" " );
```

will print...

He said, "Let's do it!"

So far we have been describing printf()'s specification as if we are forced to use only %d for an integer, only %c for a char, only %s for a string and so on. This is not true at all. In fact, printf() uses the specification that we mention and attempts to perform the specified conversion, and does its best to produce a proper result. Sometimes the result is nonsensical, as in case when we ask it to print a string using %d. Sometimes the result is useful, as in the case we ask printf( ) to print ASCII value of a character using %d. Sometimes the result is disastrous and the entire program blows up.

The following program shows a few of these conversions, some sensible, some weird.

```
main()
{
char ch = 'z' ;
int i= 125;
float a = 12.55;
char s[] = "hello there !" ;
printf ( "In%c %d %f, ch, ch, ch) ;
printf ("\n%s %d %f, s,s,s);
printf ( "\n%c %d %f,i ,i, i) ;
printf ('\n%f%d\n", a, a);
}
```

And here's the output ...

```
z 122-93628317825017830OOOOOOOOOOOOO.00000o
hello there! 3280-
9362831782501783000000000000000000.000000
} 125-93628317825017830OOOOOOOOOOOOO.00000o
12.550000
```

I will leave it to you to analyse the results by yourselves. Some of the conversions you would find are quite sensible.

Let us now turn our attention to **scanf()**. **scanf( )** allows us to enter data from keyboard that will be formatted in a certain way.

The general form of scanf( ) statement is as follows:

scanf ("format string", list of addresses of variables) ;

For example:

scanf("%d%f%c",&c,&a,&ch);

Note that we are sending addresses of variables (addresses are obtained by using **'&'** the 'address of operator') to **scanf()** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.

All the format specifications that we learnt in **printf()** function apply for **scanf( )** function as well.

sprintf( ) and sscanf( ) Functions

The **sprintf( )** function works similar to the **printf( )** function except for one small difference. Instcad of sending the output to the screen as **printf( )** does, this function writes the output to an array of characters. The following program illustrates this.

```
main()
{
 int I=10;
char ch = 'A';
float a= 3.14;
char str[20];
printf("\n%d %c %f, i, ch, a) ;
sprintf (str,"%d %c %f", i, ch, a) ;
printf("\n%s",str) ;
}
```

In this Program the **printf( )** prints out the va1ues of **i, ch** and **a** on the screen, whereas **sprintf( )** stores these values in the character array **str**. Since the string **str** is present in memory what is written into str using sprintf() doesn't get displayed on the screen. Once sir has been built, its contents can be displayed on the screen. In our program this was achieved by the second printf( ) statement.

The counterpart of **sprintf()** is the **sscanf()** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats. The scanf() function comes in handy for in-memory conversion of characters to values. You may find it convenient to read in strings from a file and then extract values from a string by using scanf(). The usage of sscanf() is same as scanf(), except that the first argument is the string from which reading is to take place.

Unformatted Console I/O Functions

There are several standard library functions available under this, category those, which can deal with a single character, and those, which can deal with a string of characters. For openers let us look at those which handle one character at a time.

So far for input we have consistently used the scanf( ) function. However, for some situations the scanf() function has one glaring weakness ... you need to hit the Enter key before the function can digest what you have typed. However, we often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. getch() and getche() are two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in getche() function means it echoes (displays) the character that you typed to the screen. As against this getch() just returns the character that you typed without echoing it on the screen. getchar() works similarly and echos the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed. The difference between getchar() and fgetchar() is that the former is a macro whereas the latter is a function.

Here is a sample program, which illustrates the use of these functions.

```
main()
{
char ch ;
prinf ( " \nPress any key to continue");
getch() ; /* will not echo the character */
printf ( "\nType any character" ) ;
ch = getche() ; /* will echo the character typed */
printf ( "\nType any characta" ) ;
getchar() ; /* will echo character, must be followed by enter key*/
printf ( "\nContinue Y/N') ;
fgetchar() ; /*will echo character, must be followed by enter key */
 }
```

And here is a sample run of this program ...

Press any key to continue

Type any character B

Type any character W Continue Y/N Y

putcb() and putchar( ) form the other side of the coin. They print a character on the screen. As far as the working of putch( ) putchar( ) and fputchar( ) is concerned it's exactly same.

The following pro-gram illustrates this.

```
main()
        {
char ch='A';
putch (ch);
putchar ( ch ) ;
fputchar ( ch ) ;
putch ('Z');
putchar ('Z');
fputchar ('Z');
}
```

And here is the output...

AAAZZZ

The limitation of **putch( ), putcbar( )** and **fputchar( )** is that they can output only one character at a time.

gets( ) and puts()

gets() receives a string from the keyboard.

Why is it needed?

Because scanf( ) function has some limitations while receiving string of characters, as the following example illustrates ...

```
main()
{
  char namel50] ;
  printf ( " \n Enter name") ;
  scanf ( "%s", name) ;
  printf ( "%s", name) ;
 }
```

And here is the output...

Enter name Jonty Rhodes

Jonty

Surprised? Where did "Rhodes" go? It never got stored in the array **name []**, because the moment the blank was typed after "Jonty" scanf( ) assumed that the name being entered has ended. The result is that there is no way (at least not without a lot of trouble on the programmer's part) to enter a multi-word string into a single variable (name in this case) using scanf(). The solution to this problem is to use gets() function. As said earlier, it gets a string from the keyboard. It is terminated when an Enter key is hit. Thus, spaces and tabs are perfectly acceptable as part of the input string. More exactly, gets() gets a newline (\n) terminated string of characters from the keyboard and replaces the \n with a \0. The puts( ) function works exactly opposite to gets( ) function. It outputs a string to the screen.

Here is a program, which illustrates the usage of these functions:

```
main()
{
 char footballer[40];
 puts(" Enter name");
 gets(footballer);    /* sends base address of array */
 puts("happy footballing!");
 puts(footballer);
}
```

Following is the sample output:

Enter name

Jonty Rhodes

Happy footballing!

Tendulkar

Why did we use two **puts()** functions to print "Happy footballing!" and "Jonty Rhodes"?

Because, unlike **printf(), puts()** can output only one string at a time. If we attempt to print two strings using **puts(),** only the first one gets printed. Similarly, unlike scanf( ), gets( ) can be used to read only one string at a time.

Formatted I/O

Formatted input conversion specifiers

c Matches a fixed number of characters. If you specify a maximum field width (see below), that is how many characters will be matched; otherwise, %c matches one character. This conversion does not append a null character to the end of the text it reads, as does the %s conversion. It also does not skip whitespace characters, but reads precisely the number of characters it was told to, or generates a matching error if it cannot.

d Matches an optionally signed decimal integer, containing the following sequence:

1. An optional plus or minus sign (+ or -).
2. One or more decimal digits.

Note that %d and %i are not synonymous for scanf, as they are for `printf`.

e Matches an optionally signed floating-point number, containing the following sequence:

An optional plus or minus sign (+ or -).

A floating-point number in decimal or hexadecimal format.

The decimal format is a sequence of one or more decimal digits, optionally containing a decimal point character (usually .), followed by an optional exponent part, consisting of a character e or E, an optional plus or minus sign, and a sequence of decimal digits.

The hexadecimal format is a 0x or 0X, followed by a sequence of one or more hexadecimal digits, optionally containing a decimal point character, followed by an optional binary-exponent part, consisting of a character p or P, an optional plus or minus sign, and a sequence of digits.

E Same as e.

f Same as e.

g Same as e.

G Same as e.

i Matches an optionally signed integer, containing the following sequence:

An optional plus or minus sign (+ or -).

A string of characters representing an unsigned integer.
If the string begins with 0x or 0X, the number is assumed to be in hexadecimal format, and the rest of the string must contain hexadecimal digits. Otherwise, if the string begins with 0, the number is assumed to be in octal format (base eight), and the rest of the string must contain octal digits. Otherwise, the number is assumed to be in decimal format, and the rest of the string must contain decimal digits.
Note that %d and %i are not synonymous for scanf, as they are for printf. You can print integers in this syntax with printf by using the # flag character with the %x or %d output conversions.

s Matches a string of non-whitespace characters. It skips initial whitespace, but stops when it meets more whitespace after it has read something. It stores a null character at the end of the text that it reads, to mark the end of the string.

x Matches an unsigned integer in hexadecimal format. The string matched must begin with 0x or 0X, and the rest of the string must contain hexadecimal digits.

X Same as x.

[Matches a string containing an arbitrary set of characters.

For example, %12[0123456789] means to read a string with a maximum field width of 12, containing characters from the set 0123456789 — in other words, twelve decimal digits. An embedded - character means a range of characters; thus %12[0-9] means the same thing as the last example. Preceding the characters in the square brackets with a caret (^) means to read a string *not* containing the characters listed. Thus, %12[^0-9] means to read a twelve-character string not containing any decimal digit.

% Matches a percent sign. Does not correspond to an argument, and does not permit flags, field width, or type modifier to be specified (see below).

In between the percent sign (%) and the input conversion character, you can place some combination of the following modifiers, in sequence. (Note that the percent sign conversion (%%) doesn't use arguments or modifiers.)

An optional * flag. This flag specifies that a match should be made between the conversion specifier and an item in the input stream, but that the value should *not* then be assigned to an argument.

An optional a flag, valid with string conversions only. This is a GNU extension to scanf that requests allocation of a buffer long enough to safely store the string that was read

An optional ' flag. This flag specifies that the number read will be grouped according to the rules currently specified on your system. For example, in the United States, this usually means that 1,000 will be read as one thousand.

An optional decimal integer that specifies the maximum field width. The scanf function will stop reading characters from the input stream either when this maximum is reached, or when a non-matching character is read, whichever comes first. Discarded initial whitespace does not count toward this width; neither does the null character stored by string input conversions to mark the end of the string.

An optional type modifier character from the following table. (The default type of the corresponding argument is int * for the %d and %i conversions, unsigned int * for %x and %X, and float * for %e and its synonyms. You can use these type modifiers to specify otherwise.)

h Specifies that the argument to which the value read should be assigned is of type short int * or unsigned short int *. Valid for the %d and %i conversions.

l For the %d and %i conversions, specifies that the argument to which the value read should be assigned is of type long int * or unsigned long int *.

For the %e conversion and its synonyms, specifies that the argument is of type double *.

L For the %d and %i conversions, specifies that the argument to which the value read should be assigned is of type long long int * or unsigned long long int *. On systems that do not have extra-long integers, this has the same effect as l. For the %e conversion and its synonyms, specifies that the argument is of type long double *.

ll Same as L, for the %d and %i conversions.

q Same as L, for the %d and %i conversions.

z Specifies that the argument to which the value read should be assigned is of type size_t. (The size_t type is used to specify the sizes of blocks of memory, and many functions in this chapter use it.) Valid for the %d and %i conversions.

Notes

LESSON 18 Formatted output conversion specifiers

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with formatted output conversion specifiers.
- Know what a space character is?
- Know about Line oriented output.

Formatted output conversion specifiers

There are many different conversion specifiers that can be used for various data types. Conversion specifiers can become quite complex; for example, %-17.7ld specifies that printf should print the number left-justified (-), in a field at least seventeen characters wide (17), with a minimum of seven digits (.7), and that the number is a long integer (l) and should be printed in decimal notation (%d).

In this section, we will examine the basics of printf and its conversion specifiers. A conversion specifier begins with a percent sign, and ends with one of the following *output conversion characters*. The most basic conversion specifiers simply use a percent sign and one of these characters, such as %d to print an integer. (Note that characters in the template string that are not part of a conversion specifier are printed as-is.)

c Print a single character.

D Print an integer as a signed decimal number.

e Print a floating-point number in exponential notation, using lower-case letters. The exponent always contains at least two digits.

E Same as e, but uses upper-case letters.

F Print a floating-point number in normal, fixed-point notation.

i Same as d.

m Print the string corresponding to the specified value of the system errno variable. (See Usual file name errors.) GNU systems only.

s Print a string.

u Print an unsigned integer.

x Print an integer as an unsigned hexadecimal number, using lower-case letters.

X Same as x, but uses upper-case letters.

% Print a percent sign (%).

In between the percent sign (%) and the output conversion character, you can place some combination of the following *modifiers*. (Note that the percent sign conversion (%%) doesn't use arguments or modifiers.)

Zero or more flag characters, from the following table:

- Left-justify the number in the field (right justification is the default). Can also be used for string and character conversions (%s and %c).

+ Always print a plus or minus sign to indicate whether the number is positive or negative. Valid for %d, %e, %E, and %i.

Space Character

If the number does not start with a plus or minus sign, prefix it with a space character instead. This flag is ignored if the + flag is specified.

For %e, %E, and %f, forces the number to include a decimal point, even if no digits follow. For %x and %X, prefixes 0x or 0X, respectively.

' Separate the digits of the integer part of the number into groups, using a locale-specific character. In the United States, for example, this will usually be a comma, so that one million will be rendered 1,000,000. GNU systems only.

0 Pad the field with zeroes instead of spaces; any sign or indication of base (such as 0x) will be printed before the zeroes. This flag is ignored if the - flag or a precision is specified.

In the example given above, %-17.7ld, the flag given is -

An optional non-negative decimal integer specifying the minimum field width within which the conversion will be printed. If the conversion contains fewer characters, it will be padded with spaces (or zeroes, if the 0 flag was specified). If the conversion contains more characters, it will not be truncated, and will overflow the field. The output will be right justified within the field, unless the - flag was specified. In the example given above, %-17.7ld, the field width is 17.

For numeric conversions, an optional precision that specifies the number of digits to be written. If it is specified, it consists of a dot character (.), followed by a non-negative decimal integer (which may be omitted, and defaults to zero if it is). In the example given above, %-17.7ld, the precision is .7. Leading zeroes are produced if necessary. If you don't specify a precision, the number is printed with as many digits as necessary (with a default of six digits after the decimal point). If you supply an argument of zero with and explicit precision of zero, printf will not print any characters. Specifying a precision for a string conversion (%s) indicates the maximum number of characters to write.

An optional *type modifier character* from the table below. This character specifies the data type of the argument if it is different from the default. In the example given above, %-17.7ld, the type modifier character is l; normally, the d output conversion character expects a data type of int, but the l specifies that a long int is being used instead. The numeric conversions usually expect an argument of either type int, unsigned int, or double. (The %c conversion converts its argument to unsigned char.) For the integer conversions (%d and %i), char and short arguments are automatically converted to type int, and for the unsigned integer conversions (%u, %x, and %X), they are

converted to type unsigned int. For the floating-point conversions (%e, %E, and %f), all float arguments are converted to type double. You can use one of the type modifiers from the table below to specify another type of argument.

l Specifies that the argument is a long int (for %d and %i), or an unsigned long int (for %u, %x, and %X).

L Specifies that the argument is a long double for the floating-point conversions (%e, %E, and %f). Same as ll, for integer conversions (%d and %i).

ll Specifies that the argument is a long long int (for %d and %i). On systems that do not have extra-long integers, this has the same effect as l.

q Same as ll; comes from calling extra-long integers "quad ints".

z Same as Z, but GNU only, and deprecated.

Z Specifies that the argument is of type size_t. (The size_t type is used to specify the sizes of blocks of memory, and many functions in this chapter use it.) Make sure that your conversion specifiers use valid syntax; if they do not, if you do not supply enough arguments for all conversion specifiers, or if any arguments are of the wrong type, unpredictable results may follow. Supplying too many arguments is not a problem, however; the extra arguments are simply ignored.

Here is a code example that shows various uses of printf.

```
#include <stdio.h>
#include <errno.h>
int main()
{
 int my_integer = -42;
 unsigned int my_ui = 23;
 float my_float = 3.56;
 double my_double = 424242.171717;
 char my_char = 'w';
 char my_string[] = "Pardon me, may I borrow your nose?";
 printf ("Integer: %d\n", my_integer);
 printf ("Unsigned integer: %u\n", my_ui);
 printf ("The same, as hexadecimal: %#x %#x\n",
my_integer, my_ui);
 printf ("Floating-point: %f\n", my_float);
 printf ("Double, exponential notation: %17.11e\n",
my_double);
 printf ("Single character: %c\n", my_char);
 printf ("String: %s\n", my_string);
 errno = EACCES;
 printf ("errno string (EACCES): %m\n");
 return 0;
}
```

The code example above produces the following output on a GNU system:

```
Integer: -42
Unsigned integer: 23
The same, as hexadecimal: 0xffffffd6 0x17
Floating-point: 3.560000
Double, exponential notation: 4.24242171717e+05
Single character: w
String: Pardon me, may I borrow your nose?
errno string (EACCES): Permission denied
```

Line-Oriented Input

Since many programs interpret input on the basis of lines, it is convenient to have functions to read a line of text from a stream. Standard C has functions to do this, but they aren't very safe: null characters and even (for gets) long lines can confuse them. So the GNU library provides the nonstandard getline function that makes it easy to read lines reliably. Another GNU extension, getdelim, generalizes getline. It reads a delimited record, defined as everything through the next occurrence of a specified delimiter character. All these functions are declared in 'stdio.h'.

Function:

ssize_t **getline** *(char **lineptr, size_t *n, FILE *stream)*

This function reads an entire line from *stream*, storing the text (including the newline and a terminating null character) in a buffer and storing the buffer address in **lineptr*. Before calling getline, you should place in **lineptr* the address of a buffer **n* bytes long, allocated with malloc. If this buffer is long enough to hold the line, getline stores the line in this buffer. Otherwise, getline makes the buffer bigger using realloc, storing the new buffer address back in **lineptr* and the increased size back in **n*. If you set **lineptr* to a null pointer, and **n* to zero, before the call, then getline allocates the initial buffer for you by calling malloc. In either case, when getline returns, **lineptr* is a char * which points to the text of the line. When getline is successful, it returns the number of characters read (including the newline, but not including the terminating null). This value enables you to distinguish null characters that are part of the line from the null character inserted as a terminator.

This function is a GNU extension, but it is the recommended way to read lines from a stream. The alternative standard functions are unreliable. If an error occurs or end of file is reached without any bytes read, getline returns -1.

Function:

ssize_t **getdelim** *(char **lineptr, size_t *n, int delimiter, FILE *stream)*

This function is like getline except that the character which tells it to stop reading is not necessarily newline. The argument *delimiter* specifies the delimiter character; getdelim keeps reading until it sees that character (or end of file). The text is stored in *lineptr*, including the delimiter character and a terminating null. Like getline, getdelim makes *lineptr* bigger if it isn't big enough.

getline is in fact implemented in terms of getdelim, just like this:

```
ssize_t
getline (char **lineptr, size_t *n, FILE *stream)
{
return getdelim (lineptr, n, '\n', stream);
}
```

Function:

char * **fgets** *(char *s, int count, FILE *stream)*

The fgets function reads characters from the stream *stream* up to and including a newline character and stores them in the string *s*, adding a null character to mark the end of the string. You must supply *count* characters worth of space in *s*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null character at the end of the string. If the system is already at end of file when you call fgets, then the contents of the array *s* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *s*.

Warning: If the input data has a null character, you can't tell. So don't use fgets unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message. We recommend using getline instead of fgets.

Function:

wchar_t * **fgetws** *(wchar_t *ws, int count, FILE *stream)*

The fgetws function reads wide characters from the stream *stream* up to and including a newline character and stores them in the string *ws*, adding a null wide character to mark the end of the string. You must supply *count* wide characters worth of space in *ws*, but the number of characters read is at most *count* - 1. The extra character space is used to hold the null wide character at the end of the string. If the system is already at end of file when you call fgetws, then the contents of the array *ws* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *ws*.

Warning: If the input data has a null wide character (which are null bytes in the input stream), you can't tell. So don't use fgetws unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message.

Function:

char * **fgets_unlocked** *(char *s, int count, FILE *stream)*

The fgets_unlocked function is equivalent to the fgets function except that it does not implicitly lock the stream. This function is a GNU extension.

Function:

wchar_t * **fgetws_unlocked** *(wchar_t *ws, int count, FILE *stream)*

The fgetws_unlocked function is equivalent to the fgetws function except that it does not implicitly lock the stream. This function is a GNU extension.

Deprecated function:

char * **gets** *(char *s)*

The function gets reads characters from the stream stdin up to the next newline character, and stores them in the string *s*. The newline character is discarded (note that this differs from the behavior of fgets, which copies the newline character into the string). If gets encounters a read error or end-of-file, it returns a null pointer; otherwise it returns *s*.

Warning: The gets function is **very dangerous** because it provides no protection against overflowing the string *s*. The GNU library includes it for compatibility only. You should **always** use fgets or getline instead. To remind you of this, the linker (if using GNU ld) will issue a warning whenever you use gets.

Notes

LESSON 19 Character Input &Character Output

Objectives

Upon completion of this Lesson, you should be able to:

- Know more about functions for performing character and line oriented output.
- Know more about character input and character output.

Simple Output by Characters or Lines

This section describes functions for performing character- and line-oriented output.

These narrow streams functions are declared in the header file 'stdio.h' and the wide stream functions in 'wchar.h'.

Function: int **fputc** (int c, FILE *stream)

The fputc function converts the character c to type unsigned char, and writes it to the stream stream. EOF is returned if a write error occurs; otherwise the character c is returned.

Function: wint_t **fputwc** (wchar_t wc, FILE *stream)

The fputwc function writes the wide character wc to the stream stream. WEOF is returned if a write error occurs; otherwise the character wc is returned.

Function: int **fputc_unlocked** (int c, FILE *stream)

The fputc_unlocked function is equivalent to the fputc function except that it does not implicitly lock the stream.

Function: wint_t **fputwc_unlocked** (wint_t wc, FILE *stream)

The fputwc_unlocked function is equivalent to the fputwc function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **putc** (int c, FILE *stream)

This is just like fputc, except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the stream argument more than once, which is an exception to the general rule for macros. putc is usually the best function to use for writing a single character.

Function: wint_t **putwc** (wchar_t wc, FILE *stream)

This is just like fputwc, except that it can be implement as a macro, making it faster. One consequence is that it may evaluate the stream argument more than once, which is an exception to the general rule for macros. putwc is usually the best function to use for writing a single wide character.

Function: int **putc_unlocked** (int c, FILE *stream)
The putc_unlocked function is equivalent to the putc function except that it does not implicitly lock the stream.

Function: wint_t **putwc_unlocked** (wchar_t wc, FILE *stream)

The putwc_unlocked function is equivalent to the putwc function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **putchar** (int c)

The putchar function is equivalent to putc with stdout as the value of the stream argument.

Function: wint_t **putwchar** (wchar_t wc)

The putwchar function is equivalent to putwc with stdout as the value of the stream argument.

Function: int **putchar_unlocked** (int c)

The putchar_unlocked function is equivalent to the putchar function except that it does not implicitly lock the stream.

Function: wint_t **putwchar_unlocked** (wchar_t wc)

The putwchar_unlocked function is equivalent to the putwchar function except that it does not implicitly lock the stream.This function is a GNU extension.

Function: int **fputs** (const char *s, FILE *stream)

The function fputs writes the string s to the stream stream. The terminating null character is not written. This function does not add a newline character, either. It outputs only the characters in the string. This function returns EOF if a write error occurs, and otherwise a non-negative value.

For example:

```
fputs ("Are ", stdout);
fputs ("you ", stdout);
fputs ("hungry?\n", stdout);
```

outputs the text 'Are you hungry?' followed by a newline.

Function: int **fputws** (const wchar_t *ws, FILE *stream)

The function fputws writes the wide character string ws to the stream stream. The terminating null character is not written. This function does not add a newline character, either. It outputs only the characters in the string. This function returns WEOF if a write error occurs, and otherwise a non-negative value.

Function: int **fputs_unlocked** (const char *s, FILE *stream)

The fputs_unlocked function is equivalent to the fputs function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **fputws_unlocked** (const wchar_t *ws, FILE *stream)

The fputws_unlocked function is equivalent to the fputws function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **puts** (const char *s)

The puts function writes the string s to the stream stdout followed by a newline. The terminating null character of the string is not written. (Note that fputs does not write a newline as this function does.) puts is the most convenient function for printing simple messages. For example:

```
puts ("This is a message.");
```

outputs the text 'This is a message.' followed by a new line.

Function: int **putw** (int w, FILE *stream)

This function writes the word w (that is, an int) to stream. It is provided for compatibility with SVID, but we recommend you use fwrite instead (see section

Character Input

This section describes functions for performing character-oriented input. These narrow streams functions are declared in the header file 'stdio.h' and the wide character functions are declared in 'wchar.h'.

These functions return an int or wint_t value (for narrow and wide stream functions respectively) that is either a character of input, or the special value EOF/WEOF (usually -1). For the narrow stream functions it is important to store the result of these functions in a variable of type int instead of char, even when you plan to use it only as a character. Storing EOF in a char variable truncates its value to the size of a character, so that it is no longer distinguishable from the valid character '(char) -1'. So always use an int for the result of getc and friends, and check for EOF after the call; once you've verified that the result is not EOF, you can be sure that it will fit in a 'char' variable without loss of information.

Function: int **fgetc** *(FILE *stream)*

This function reads the next character as an unsigned char from the stream *stream* and returns its value, converted to an int. If an end-of-file condition or read error occurs, EOF is returned instead.

Function: wint_t **fgetwc** *(FILE *stream)*

This function reads the next wide character from the stream *stream* and returns its value. If an end-of-file condition or read error occurs, WEOF is returned instead.

Function: int **fgetc_unlocked** *(FILE *stream)*

The fgetc_unlocked function is equivalent to the fgetc function except that it does not implicitly lock the stream.

Function: wint_t **fgetwc_unlocked** *(FILE *stream)*

The fgetwc_unlocked function is equivalent to the fgetwc function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **getc** *(FILE *stream)*

This is just like fgetc, except that it is permissible (and typical) for it to be implemented as a macro that evaluates the *stream* argument more than once. getc is often highly optimized, so it is usually the best function to use to read a single character.

Function: wint_t **getwc** *(FILE *stream)*

This is just like fgetwc, except that it is permissible for it to be implemented as a macro that evaluates the *stream* argument more than once. getwc can be highly optimized, so it is usually the best function to use to read a single wide character.

Function: int **getc_unlocked** *(FILE *stream)*

The getc_unlocked function is equivalent to the getc function except that it does not implicitly lock the stream.

Function: wint_t **getwc_unlocked** *(FILE *stream)*

The getwc_unlocked function is equivalent to the getwc function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: int **getchar** *(void)*

The getchar function is equivalent to getc with stdin as the value of the *stream* argument.

Function: wint_t **getwchar** *(void)*

The getwchar function is equivalent to getwc with stdin as the value of the *stream* argument.

Function: int **getchar_unlocked** *(void)*

The getchar_unlocked function is equivalent to the getchar function except that it does not implicitly lock the stream.

Function: wint_t **getwchar_unlocked** *(void)*

The getwchar_unlocked function is equivalent to the getwchar function except that it does not implicitly lock the stream. This function is a GNU extension.

Here is an example of a function that does input using fgetc. It would work just as well using getc instead, or using getchar () instead of fgetc (stdin). The code would also work the same for the wide character stream functions.

```
int y_or_n_p (const char *question)
{
fputs (question, stdout);
while (1)
{
    int c, answer;      /* Write a space to separate answer from question. */
fputc (' ', stdout);      /* Read the first character of the line. This should be the answer character, but might not be. */
c = tolower (fgetc (stdin));
answer = c;      /* Discard rest of input line. */while (c != '\n' && c != EOF)
 c = fgetc (stdin);      /* Obey the answer if it was valid. */
if (answer == 'y') return 1;
if (answer == 'n')  return 0;      /* Answer was invalid: ask for valid answer. */
fputs ("Please answer y or n:", stdout);    } }
```

Function: int **getw** *(FILE *stream)*

This function reads a word (that is, an int) from *stream*. It's provided for compatibility with SVID. We recommend you use fread instead. Unlike getc, any int value could be a valid result. getw returns EOF when it encounters end-of-file or an error, but there is no way to distinguish this from an input word with value -1.

A number of functions provide for character oriented I/O. Their declarations are:

```
#include <stdio.h>
/* character input */
int fgetc(FILE *stream);
int getc(FILE *stream);
```

```
int getchar(void);
int ungetc(int c, FILE *stream);
/* character output */
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
/* string input */
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
/* string output */
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

Their descriptions are as follows.

Character Input

These read an unsigned char from the input stream where specified, or otherwise stdin. In each case, the next character is obtained from the input stream. It is treated as an unsigned char and converted to an int, which is the return value. On End of File, the constant EOF is returned, and the end-of-file indicator is set for the associated stream. On error, EOF is returned, and the error indicator is set for the associated stream. Successive calls will obtain characters sequentially. The functions, if implemented as macros, may evaluate their stream argument more than once, so do not use side effects here.

There is also the supporting ungetc routine, which is used to push back a character on to a stream, causing it to become the next character to be read. This is not an output operation and can never cause the external contents of a file to be changed. A fflush, fseek, or rewind operation on the stream between the pushback and the read will cause the pushback to be forgotten. Only one character of pushback is guaranteed, and attempts to pushback EOF are ignored. In every case, pushing back a number of characters then reading or discarding them leaves the file position indicator unchanged. The file position indicator is decremented by every successful call to ungetc for a binary stream, but unspecified for a text stream, or a binary stream, which is positioned at the beginning of the file.

Character Output

These are identical in description to the input functions already described, except performing output. They return the character written, or EOF on error. There is no equivalent to End of File for an output file.

Notes

LESSON 20 Stream I/O

Objectives

Upon completion of this Lesson, you should be able to:

- Know about type conversion.
- Know more about stream I/O
- Know more about unreading and block I/O.

Many operators cause conversion of operands into a common type, yielding a result of the same type. This pattern denotes the usual arithmetic conversions. In any evaluation, the type conversion is done as follows (in the given order).

If one of the operands is a long double , the other is converted to a long double.

else if one of the operand is a double, then the other is converted into a double.

else if one of the operands is a float, the other is converted to a float.

else if one of the operands is an unsigned long int, then the other is converted to an unsigned long int.

else if one operands is a long int and the other is an unsigned int, the effect depends on whether a long int.

else if one operand is a long int , the other an unsigned int, the effect depends on whether a long int can be represent all values of an unsigned int ; if so, the unsigned int, operand is converted to a long int; if not, both are converted to an unsigned long int.

else if one operand is a long int, the other is converted to a long int.

else if either operand is an unsigned int, the other is converted to an unsigned int.

else if one of the operand is an int, then the other is converted to an int.

else if an one of the operands is an unsigned char then the other is converted to an unsigned char.

else both operands are char

unexpected results may occur when an unsigned expression is compared with a signed expression of the same size.

An expression results may occur when an unsigned expression is compared with a signed expression of the same size.

An expression of the integral type may be added to or subtracted from a pointer variable; the results of such an operation is an address offset. The actual address can be obtained by multiplying this offset by the size of the object to which the pointer points. The following is an example:

int a[10];

(a+5)=a[0] + a[2]; / adds the first and third element of array a[10] and places the

result at the position 5 in the array */

when 2 pointers to objects of the same type, in the same array , are subtracted, the result obtained is an integer. This gives the displacement between the two pointed to objects.

An integral constant expression with a value 0, or such an expression cast to type void *, may be converted to a pointer of any type, by type casting, by assignment, or by comparison. This produces a null pointer that is equal to another null pointer of the same type, but unequal to any pointer to function or pointer-to-object.

Stream I/O

Input and Output (I/O): stdio.h

This chapter will look at many forms of I/O. We have briefly mentioned some forms before will look at these in much more detail here.

Your programs will need to include the standard I/O header file so do:

#include <stdio.h>

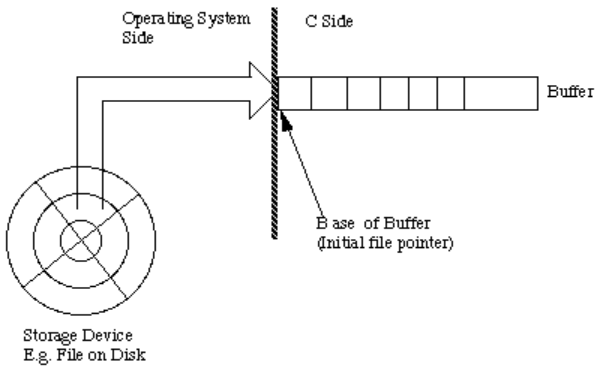
Streams

Streams are a portable way of reading and writing data. They provide a flexible and efficient means of I/O. A Stream is a file or a physical device (e.g. printer or monitor), which is manipulated with a pointer to the stream. There exists an internal C data structure, FILE, which represents all streams and is defined in stdio.h.

- We simply need to refer to the FILE structure in C programs when performing I/O with streams.
- We just need to declare a variable or pointer of this type in our programs.
- We do not need to know any more specifics about this definition.
- We must open a stream before doing any I/O, then access it and then close it.

Stream I/O is BUFFERED: That is to say a fixed "chunk" is read from or written to a file via some temporary storage area (the buffer). This is illustrated in the following figure.

NOTE the file pointer actually points to this buffer.



Stream I/O Model this leads to efficient I/O but beware: data written to a buffer does not appear in a file (or device) until the buffer is flushed or written out. (n does this). Any abnormal exit of code can cause problems.

Predefined Streams

UNIX defines 3 predefined streams (in stdio.h):

stdin, stdout, stderr

They all use text a the method of I/O.

stdin and stdout can be used with files, programs, I/O devices such as keyboard, console, etc.. stderr always goes to the console or screen.

The console is the default for stdout and stderr. The keyboard is the default for stdin.

Predefined stream are automatically open.

Redirection

This how we override the UNIX default predefined I/O defaults.

This is not part of C but operating system dependent. We will do redirection from the command line.

> — redirect stdout to a file.

So if we have a program, out, that usually prints to the screen then

out > file1 will send the output to a file, file1.

< — redirect stdin from a file to a program.

So if we are expecting input from the keyboard for a program, in we can read similar input from a file

in < file2.

|— pipe: puts stdout from one program to stdin of another

prog1 | prog2

e.g. Sent output (usually to console) of a program direct to printer:

out | lpr

Basic I/O

There are a couple of function that provide basic I/O facilities. probably the most common are: getchar() and putchar(). They are defined and used as follows:

1. int getchar(void) — reads a char from stdin
2. int putchar(char ch) — writes a char to stdout, returns character written.

```
int ch;
ch = getchar();
(void) putchar((char) ch);
```

Related Functions

int getc(FILE *stream),
int putc(char ch,FILE *stream)

Formatted I/O

We have seen examples of how C uses formatted I/O already. Let's look at this in more detail.

printf

The function is defined as follows:

int printf(char *format, arg list ...) —
prints to stdout the list of arguments according specified format string. Returns number of characters printed. The format string has 2 types of object:

1. ordinary characters — these are copied to output.
2. conversion specifications — denoted by % and listed in Table

Table: printf/scanf format characters

Format Spec (%)	Type	Result
C	char	single character
i,d	int	decimal number
O	int	octal number
x,X	int	hexadecimal number
		lower/uppercase notation
U	int	unsigned int
S	char *	print string
		terminated by 0
F	double/float	format -m.ddd...
e,E	"	Scientific Format
		-1.23e002
g,G	"	e or f whichever
		is most compact
%	-	print % character

Between % and format char we can put:

- (Minus sign)

- Left justify.

Integer number

— field width.

m.d

— m = field width, d = precision of number of digits after decimal point or number of chars from a string.

So:

```
printf("%-2.3fn",17.23478);
```

The output on the screen is:

```
17.235
```

and:

```
printf("VAT=17.5%%n");
```

...outputs:

VAT=17.5%

Unreading

In parser programs it is often useful to examine the next character in the input stream without removing it from the

stream. This is called "peeking ahead" at the input because your program gets a glimpse of the input it will read next. Using stream I/O, you can peek ahead at input by first reading it and then unreading it (also called pushing it back on the stream). Unreading a character makes it available to be input again from the stream, by the next call to fgetc or other input function on that stream.

What Unreading Means

Here is a pictorial explanation of unreading. Suppose you have a stream reading a file that contains just six characters, the letters 'foobar'. Suppose you have read three characters so far. The situation looks like this:

```
f o o b a r
^
```

so the next input character will be 'b'.

If instead of reading 'b' you unread the letter 'o', you get a situation like this:

```
f o o b a r
|   o—
^
```

so that the next input characters will be 'o' and 'b'.

If you unread '9' instead of 'o', you get this situation:

```
f o o b a r
|   9—
^
```

so that the next input characters will be '9' and 'b'.

Using ungetc to do Unreading

The function to unread a character is called ungetc, because it reverses the action of getc.

Function: int **ungetc** (int c, FILE *stream)

The ungetc function pushes back the character c onto the input stream stream. So the next input from stream will read c before anything else. If c is EOF, ungetc does nothing and just returns EOF. This lets you call ungetc with the return value of getc without needing to check for an error from getc. The character that you push back doesn't have to be the same as the last character that was actually read from the stream. In fact, it isn't necessary to actually read any characters from the stream before unreading them with ungetc! But that is a strange way to write a program; usually ungetc is used only to unread a character that was just read from the same stream. The GNU C library supports this even on files opened in binary mode, but other systems might not. The GNU C library only supports one character of pushback—in other words, it does not work to call ungetc twice without doing input in between. Other systems might let you push back multiple characters; then reading from the stream retrieves the characters in the reverse order that they were pushed.

Pushing back characters doesn't alter the file; only the internal buffering for the stream is affected. If a file positioning functionis called, any pending pushed-back characters are discarded. Unreading a character on a stream that is at end of file clears the end-of-file indicator for the stream, because it makes the character of input available. After you read that character, trying to read again will encounter end of file.

Function: wint_t **ungetwc** (wint_t wc, FILE *stream)

The ungetwc function behaves just like ungetc just that it pushes back a wide character.

Here is an example showing the use of getc and ungetc to skip over whitespace characters. When this function reaches a non-whitespace character, it unreads that character to be seen again on the next read operation on the stream.

```
#include <stdio.h>#include <ctype.h>  void skip_whitespace (FILE *stream) {  int c;  do  /* No need to check for EOF  because it is not isspace, and  ungetc ignores EOF.  */  c = getc (stream);  while (isspace (c)); ungetc (c, stream); }
```

Block Input/Output

This section describes how to do input and output operations on blocks of data. You can use these functions to read and write binary data, as well as to read and write text in fixed-size blocks instead of by characters or lines. Binary files are typically used to read and write blocks of data in the same format as is used to represent the data in a running program. In other words, arbitrary blocks of memory—not just character or string objects—can be written to a binary file, and meaningfully read in again by the same program. Storing data in binary form is often considerably more efficient than using the formatted I/O functions. Also, for floating-point numbers, the binary form avoids possible loss of precision in the conversion process. On the other hand, binary files can't be examined or modified easily using many standard file utilities (such as text editors), and are not portable between different implementations of the language, or different kinds of computers.

These functions are declared in 'stdio.h'.

Function: size_t **fread** (void *data, size_t size, size_t count, FILE *stream)

This function reads up to count objects of size size into the array data, from the stream stream. It returns the number of objects actually read, which might be less than count if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either size or count is zero.

If fread encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.

Function: size_t **fread_unlocked** (void *data, size_t size, size_t count, FILE *stream)

The fread_unlocked function is equivalent to the fread function except that it does not implicitly lock the stream. This function is a GNU extension.

Function: size_t **fwrite** (const void *data, size_t size, size_t count, FILE *stream)

This function writes up to count objects of size size from the array data, to the stream stream. The return value is normally count, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

Function: size_t **fwrite_unlocked** (const void *data, size_t size, size_t count, FILE *stream)

The fwrite_unlocked function is equivalent to the fwrite function except that it does not implicitly lock the stream. This function is a GNU extension

Notes

LESSON 21 Scope of Veriables

Objectives

Upon completion of this Lesson, you should be able to:

- Know about the scope of a variable.
- Know more about ins and outs of local variables.
- Know about static and automatic variables.

You have already seen that a variable defined within a function is different from a variable defined outside a function. Without knowing it, you were being introduced to the concept of *variablescope*, an important aspect of C programming.

Today you will learn

- About scope and why it's important
- What external variables are and why you should usually avoid them
- The ins and outs of local variables
- The difference between static and automatic variables
- About local variables and blocks
- How to select a storage class

What Is Scope?

The *scope* of a variable refers to the extent to which different parts of a program have access to the variable—in other words, where the variable is *visible*. When referring to C variables, the terms *accessibility* and *visibility* are used interchangeably. When speaking about scope, the term *variable* refers to all C data types: simple variables, arrays, structures, pointers, and so forth. It also refers to symbolic constants defined with the const keyword.

Scope also affects a variable's *lifetime:* how long the variable persists in memory, or when the variable's storage is allocated and deallocated. First, we will examine visibility.

A Demonstration of Scope

Let us have a look at the program given below. It defines the variable x in line 5, uses printf() to display the value of x in line 11, and then calls the function print_value() to display the value of x again. Note that the function print_value() is not passed the value of x as an argument; it simply uses x as an argument to printf() in line 19.

Program 1.The variable x is accessible within the function print_value().

```
1: /* Illustrates variable scope. */
2:
3: #include <stdio.h>
4:
5: int x = 999;
6:
7: void print_value(void);
8:
9: main()
10: {
11: printf("%d\n", x);
12: print_value();
13:
14: return 0;
15: }
16:
17: void print_value(void)
18: {
19: printf("%d\n", x);
20: }
```

Output

```
999
999
```

This program compiles and runs with no problems. Now make a minor modification in the program, moving the definition of the variable x to a location within the main() function. The new source code is shown below.

Program 2. The variable x is not accessible within the function print_value().

```
1: /* Illustrates variable scope. */
2:
3: #include <stdio.h>
4:
5: void print_value(void);
6:
7: main()
8: {
9: int x = 999;
10:
11: printf("%d\n", x);
12: print_value();
13:
14: return 0;
15: }
16:
17: void print_value(void)
18: {
19: printf("%d\n", x);
20: }
```

Analysis

If you try to compile the above program, the compiler generates an error message similar to the following:

list1202.c(19) : Error: undefined identifier 'x'.

Remember that in an error message, the number in parentheses refers to the program line where the error was found. Line 19 is the call to printf() within the print_value() function. This error message tells you that within the print_value() function, the variable x is undefined or, in other words, not visible.

Note : However, that the call to printf() in line 11 doesn't generate an error message; in this part of the program, the variable x *is* visible. The only difference between the two programs is where variable x is defined. By moving the definition of x, you change its scope. In Program 1, x is an *external variable*, and its scope is the entire program. It is accessible within both the main() function and the print_value() function. In Program 2, x is a *local variable,* and its scope is limited to within the main() function. As far as print_value() is concerned, x doesn't exist. Later, you'll learn more about local and external variables, but first you need to understand the importance of scope.

Why is Scope Important?

In the structured approach, you might remember, it divides the program into independent functions that perform a specific task. The key word here is *independent*. For true independence, it's necessary for each function's variables to be isolated from interference caused by other functions. Only by isolating each function's data can you make sure that the function goes about its job without some other part of the program throwing a monkey wrench into the works. If you're thinking that complete data isolation between functions isn't always desirable, you are correct. You will soon realize that by specifying the scope of variables, a programmer has a great deal of control over the degree of data isolation.

External Variables

An *external variable* is a variable defined outside of any function. This means outside of main() as well, because main() is a function, too. Until now, most of the variable definitions in this book have been external, placed in the source code before the start of main(). External variables are sometimes referred to as *global variables*.

NOTE: If you don't explicitly initialize an external variable when it's defined, the compiler initializes it to 0.

External Variable Scope

The scope of an external variable is the entire program. This means that an external variable is visible throughout main() and every other function in the program. For example, the variable x in Program 1 is an external variable. As you saw when you compiled and ran the program, x is visible within both functions, main() and print_value(). Strictly speaking, it's not accurate to say that the scope of an external variable is the entire program. Instead, the scope is the entire source code file that contains the variable definition. If the entire program is contained in one source code file, the two scope definitions are equivalent. Most small-to-medium-sized C programs are contained in one file, and that's certainly true of the programs you're writing now. It's possible, however, for a program's source code to be contained in two or more separate files.

When to Use External Variables

Although the sample programs to this point have used external variables, in actual practice you should use them rarely. Why? Because when you use external variables, you are violating the principle of *modular independence* that is central to structured programming. Modular independence is the idea that each function, or module, in a program contains all the code and data it needs to do its job. With the relatively small programs you're writing now, this might not seem important, but as you progress to larger and more complex programs, over reliance on external variables can start to cause problems.

When should you use external variables?

Make a variable external only when all or most of the program's functions need access to the variable. Symbolic constants defined with the const keyword are often good candidates for external status. If only some of your functions need access to a variable, pass the variable to the functions as an argument rather than making it external.

The extern Keyword

When a function uses an external variable, it is good programming practice to declare the variable within the function using the extern keyword. The declaration takes the form

extern *type name*;

in which *type* is the variable type and *name* is the variable name. For example, you would add the declaration of x to the functions main() and print_value() in Program 1. The resulting program is shown below.

Program 3. The external variable x is declared as extern within the functions main() and print_value().

```
1: /* Illustrates declaring external variables. */
2:
3: #include <stdio.h>
4:
5: int x = 999;
6:
7: void print_value(void);
8:
9: main()
10: {
11: extern int x;
12:
13: printf("%d\n", x);
14: print_value();
15:
16: return 0;
17: }
18:
19: void print_value(void)
20: {
21: extern int x;
22: printf("%d\n", x);
23: }
```

Output

```
999
999
```

Analysis

This program prints the value of x twice, first in line 13 as a part of main(), and then in line 21 as a part of print_value(). Line 5 defines x as a type int variable equal to 999. Lines 11 and 21 declare x as anextern int. Note the distinction between a variable definition, which sets aside storage for the variable, and an extern declaration. The latter says: "This function uses an external variable with such-and-such a name and type that is defined elsewhere." In this case, the extern declaration isn't needed, strictly speaking—the program will work the same without lines 11 and 21. However, if the function print_value() were in a different code module than the global declaration of the variable x (in line 5), the extern declaration would be required.

Local Variables

A *local variable* is one that is defined within a function. The scope of a local variable is limited to the function in which it is defined. Local variables aren't automatically initialized to 0 by the compiler. If you don't initialize a local variable when it's defined, it has an undefined or *garbage* value. You must explicitly assign a value to local variables before they're used for the first time. A variable can be local to the main() function as well. This is the case for x in Program.2. It is defined within main(), and as compiling and executing that program illustrates, it's also only visible within main().

- **DO** use local variables for items such as loop counters.
- **DON'T** use external variables if they aren't needed by a majority of the program's functions.
- **DO** use local variables to isolate the values the variables contain from the rest of the program.

Static Versus Automatic Variables

Local variables are *automatic* by default. This means that local variables are created a new each time the function is called, and they are destroyed when execution leaves the function. What this means, in practical terms, is that an automatic variable doesn't retain its value between calls to the function in which it is defined.

Suppose your program has a function that uses a local variable x. Also suppose that the first time it is called, the function assigns the value 100 to x. Execution returns to the calling program, and the function is called again later.

Does the variable x still hold the value 100? No, it does not. The first instance of variable x was destroyed when execution left the function after the first call. When the function was called again, a new instance of x had to be created. The old x is gone. What if the function needs to retain the value of a local variable between calls?

For example, a printing function might need to remember the number of lines already sent to the printer to determine when a new page is needed. In order for a local variable to retain its value between calls, it must be defined as *static* with the static keyword.

For example:

```
void func1(int x)
{
static int a;
/* Additional code goes here */
}
```

Program.4 illustrates the difference between automatic and static local variables.

Program 4. The difference between automatic and static local variables.

```
1: /* Demonstrates automatic and static local variables. */
2: #include <stdio.h>
3: void func1(void);
4: main()
5: {
6: int count;
7:
8: for (count = 0; count < 20; count++)
9: {
10: printf("At iteration %d: ", count);
11: func1();
12: }
13:
14: return 0;
15: }
16:
17: void func1(void)
18: {
19: static int x = 0;
20: int y = 0;
21:
22: printf("x = %d, y = %d\n", x++, y++);
23: }
```

Output

```
At iteration 0: x = 0, y = 0
At iteration 1: x = 1, y = 0
At iteration 2: x = 2, y = 0
At iteration 3: x = 3, y = 0
At iteration 4: x = 4, y = 0
At iteration 5: x = 5, y = 0
At iteration 6: x = 6, y = 0
At iteration 7: x = 7, y = 0
At iteration 8: x = 8, y = 0
At iteration 9: x = 9, y = 0
At iteration 10: x = 10, y = 0
At iteration 11: x = 11, y = 0
At iteration 12: x = 12, y = 0
At iteration 13: x = 13, y = 0
At iteration 14: x = 14, y = 0
At iteration 15: x = 15, y = 0
```

```
At iteration 16: x = 16, y = 0
At iteration 17: x = 17, y = 0
At iteration 18: x = 18, y = 0
At iteration 19: x = 19, y = 0
```

Analysis

This program has a function that defines and initializes one variable of each type. This function is func1() in lines 17 through 23. Each time the function is called, both variables are displayed on-screen and incremented (line 22). The main() function in lines 4 through 15 contains a for loop (lines 8 through 12) that prints a message (line 10) and then calls func1() (line 11). The for loop iterates 20 times. In the output, note that x, the static variable, increases with each iteration because it retains its value between calls. The automatic variable y, on the other hand, is reinitialized to 0 with each call. This program also illustrates a difference in the way explicit variable initialization is handled (that is, when a variable is initialized at the time of definition). A static variable is initialized only the first time the function is called. At later calls, the program remembers that the variable has already been initialized and therefore doesn't reinitialize. Instead, the variable retains the value it had when execution last exited the function. In contrast, an automatic variable is initialized to the specified value every time the function is called.

If you experiment with automatic variables, you might get results that disagree with what you've read here. For example, if you modify Program 4 so that the two local variables aren't initialized when they're defined, the function func1() in lines 17 through 23 reads

```
17: void func1(void)
18: {
19: static int x;
20: int y;
21:
22: printf("x = %d, y = %d\n", x++, y++);
23: }
```

When you run the modified program, you might find that the value of y increases by 1 with each iteration. This means that y is keeping its value between calls to the function. Is what you've read here about automatic variables losing their value a bunch of malarkey?

No, what you read is true (Have faith!). If you get the results described earlier, in which an automatic variable keeps its value during repeated calls to the function, it's only by chance. Here's what happens: Each time the function is called, a new y is created. The compiler might use the same memory location for the new y that was used for y the preceding time the function was called. If y isn't explicitly initialized by the function, the storage location might contain the value that y had during the preceding call. The variable seems to have kept its old value, but it's just a chance occurrence; you definitely can't count on it happening every time. Because automatic is the default for local variables, it doesn't need to be specified in the variable definition. If you want to, you can include the auto keyword in the definition before the type keyword, as shown here:

```
void func1(int y)
{
auto int count;
/* Additional code goes here */
}
```

Notes

LESSON 22 Scope of Function Parameters

Objectives

Upon completion of this Lesson, you should be able to:

- Know about the scope of function parameters.
- Know about extern static variables and register variables.
- Know where to use which storage class and when.

The Scope of Function Parameters

A variable that is contained in a function heading's parameter list has *local scope*.

For example, look at the following function:

```
void func1(int x)
{
int y;
/* Additional code goes here */
}
```

Both x and y are local variables with a scope that is the entire function func1(). Of course, x initially contains whatever value was passed to the function by the calling program. Once you've made use of that value, you can use x like any other local variable.

Because parameter variables always start with the value passed as the corresponding argument, it's meaningless to think of them as being either static or automatic.

External Static Variables

You can make an external variable static by including the static keyword in its definition:

```
static float rate;
main()
{
/* Additional code goes here */
}
```

The difference between an ordinary external variable and a static external variable is one of scope. An ordinary external variable is visible to all functions in the file and can be used by functions in other files. A static external variable is visible only to functions in its own file and below the point of definition. These distinctions obviously apply mostly to programs with source code that is contained in two or more files.

Register Variables

The register keyword is used to suggest to the compiler that an automatic local variable be stored in a *processor register* rather than in regular memory. What is a processor register, and what are the advantages of using it?

The central processing unit (CPU) of your computer contains a few data storage locations called *registers*. It is in the CPU registers that actual data operations, such as addition and division, take place. To manipulate data, the CPU must move the data from memory to its registers, perform the manipulations, and then move the data back to memory. Moving data to and from memory takes a finite amount of time. If a particular variable could be kept in a register to begin with, manipulations of the variable would proceed much faster. By using the register keyword in the definition of an automatic variable, you ask the compiler to store that variable in a register.

Look at the following example:

```
void func1(void)
{
register int x;
/* Additional code goes here */
}
```

Note that I said *ask*, not *tell*. Depending on the program's needs, a register might not be available for the variable. In this case, the compiler treats it as an ordinary automatic variable. The register keyword is a suggestion, not an order. The benefits of the register storage class are greatest for variables that the function uses frequently, such as the counter variable for a loop. The register keyword can be used only with simple numeric variables, not arrays or structures. Also, it can't be used with either static or external storage classes. You can't define a pointer to a register variable.

- **DO** initialize local variables, or you won't know what value they will contain.
- **DO** initialize global variables even though they're initialized to 0 by default. If you always initialize your variables, you'll avoid problems such as forgetting to initialize local variables.
- **DO** pass data items as function parameters instead of declaring them as global if they're needed in only a few functions.
- **DON'T** use register variables for nonnumeric values, structures, or arrays.

Local Variables and the main() Function

Everything said so far about local variables applies to main() as well as to all other functions. Strictly speaking, main() is a function like any other. The main() function is called when the program is started from your operating system, and control is returned to the operating system from main() when the program terminates. This means that local variables defined in main() are created when the program begins, and their lifetime is over when the program ends. The notion of a static local variable retaining its value between calls to main() really makes no sense: A variable can't remain in existence between program executions. Within main(), therefore, there is no difference between automatic and static local variables. You can define a local variable in main() as being static, but it has no effect.

- **DO** remember that main() is a function similar in most respects to any other function.
- **DON'T** declare static variables in main(), because doing so gains nothing.

Which Storage Class Should You Use?

When you're deciding which storage class to use for particular variables in your programs, it might be helpful to refer to table given below which summarizes the five storage classes available in C.

C's five variable storage classes.

Storage Lifetime	Where It's Defined	Scope	Class	Keyword
Automatic	None[1]	Temporary	In a function	Local
Static	static	Temporary	In a function	Local
Register	register	Temporary	In a function	Local
External	None[2]	Permanent	Outside a function	Global (all files)
External	static	Permanent	Outside a function	Global (one file)
[1]The auto keyword is optional.				
[2]The extern keyword is used in functions to declare a static external variable that is defined elsewhere.				

When you're deciding on a storage class, you should use an automatic storage class whenever possible and use other classes only when needed. Here are some guidelines to follow:

- Give each variable an automatic local storage class to begin with
- If the variable will be manipulated frequently, add the register keyword to its definition.
- In functions other than main(), make a variable static if its value must be retained between calls to thefunction.
- If a variable is used by most or all of the program's functions, define it with the external storage class.

Local Variables and Blocks

So far, we have discussed only variables that are local to a function. This is the primary way local variables are used, but you can define variables that are local to any program block (any section enclosed in braces). When declaring variables within the block, you must remember that the declarations must be first.

Program 5 shows an example.

Program 5. Defining local variables within a program block.

Analysis

From this program, you can see that the count defined within the block is independent of the count defined outside the block. Line 9 defines count as a type int variable equal to 0. Because it is declared at the beginning of main(), it can be used throughout the entire main() function. The code in line 11 shows that the variable count has been initialized to 0 by printing its value. A block is declared in lines 14 through 19, and within the block, another count variable is defined as a type int variable. This count variable is initialized to 999 in line 17. Line 18 prints the block's count variable value of 999. Because the block ends on line 19, the print statement in line 21 uses the original count initially declared in line 9 of main(). The use of this type of local variable isn't common in C programming, and you may never find a need for it. Its most common use is probably when a programmer tries to isolate a problem within a program. You can temporarily isolate sections of code in braces and establish local variables to assist in tracking down the problem. Another advantage is that the variable declaration/initialization can be placed closer to the point where it's used, which can help in understanding the program.

- **DON'T** try to put variable definitions anywhere other than at the beginning of a function or at the beginning of a block.
- **DON'T** use variables at the beginning of a block unless it makes the program clearer.
- **DO** use variables at the beginning of a block (temporarily) to help track down problems.

Summary

This lesson covered C's variable storage classes. Every C variable, whether a simple variable, an array, a structure, or whatever, has a specific storage class that determines two things: its scope, or where in the program it's visible; and its lifetime, or how long the variable persists in memory. Proper use of storage classes is an important aspect of structured programming. By keeping most variables local to the function that uses them, you enhance functions' independence from each other. A variable should be given automatic storage class unless there is a specific reason to make it external or static.

Q&A

Q If global variables can be used anywhere in the program, why not make all variables global?

A As your programs get bigger, you will begin to declare more and more variables. As stated in this lesson, there are limits on the amount of memory available. Variables declared as global take up memory for the entire time the program is running; however, local variables don't. For the most part, a local variable takes up memory only while the function to which it is local is active. (A static variable takes up memory from the time it is first used to the end of the program.) Additionally, global variables are subject to unintentional alteration by other functions. If this occurs, the variables might not contain the values you expect them to when they're used in the functions for which they were created.

Q."Structures," stated that scope affects a structure instance but not a structure tag or body. Why doesn't scope affect the structure tag or body?

A When you declare a structure without instances, you are creating a template. You don't actually declare any variables. It isn't until you create an instance of the structure that you declare a variable. For this reason, you can leave a structure body external to any functions with no real effect on external memory. Many programmers put commonly used structure bodies with tags into header files and then include these header files when they need to create an instance of the structure.

Q How does the computer know the difference between a global variable and a local variable that have the same name?

A The answer to this question is beyond the scope of this chapter. The important thing to know is that when a local variable is declared with the same name as a global variable, the program temporarily ignores the global variable. It continues to ignore the global variable until the local variable goes out of scope.

Q Can I declare a local variable and a global variable that have the same name, as long as they have different variable types?

A Yes. When you declare a local variable with the same name as a global variable, it is a completely different variable. This means that you can make it whatever type you want. You should be careful, however, when declaring global and local variables that have the same name.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What does scope refer to?
2. What is the most important difference between local storage class and external storage class?
3. How does the location of a variable definition affect its storage class?
4. When defining a local variable, what are the two options for the variable's lifetime?
5. Your program can initialize both automatic and static local variables when they are defined. When do the initializations take place?
6. True or false: A register variable will always be placed in a register.
7. What value does an uninitialized global variable contain?
8. What value does an uninitialized local variable contain?
9. What will line 21 of Listing 12.5 print if lines 9 and 11 are removed? Think about this, and then try the program to see what happens.
10. If a function needs to remember the value of a local type int variable between calls, how should the variable be declared?
11. What does the extern keyword do?
12. What does the static keyword do?

Exercises

1. Write a declaration for a variable to be placed in a CPU register.
2. Change Listing 12.2 to prevent the error. Do this without using any external variables.
3. Write a program that declares a global variable of type int called var. Initialize var to any value. The program should print the value of var in a function (not main()). Do you need to pass var as a parameter to the function?
4. Change the program in exercise 3. Instead of declaring var as a global variable, change it to a local variable in main(). The program should still print var in a separate function. Do you need to pass var as a parameter to the function?
5. Can a program have a global and a local variable with the same name? Write a program that uses a global and a local variable with the same name to prove your answer.
6. **BUG BUSTER:** Can you spot the problem in this code? Hint: It has to do with where a variable is declared.

```
void a_sample_function( void )
{
int ctr1;
for ( ctr1 = 0; ctr1 < 25; ctr1++ )
printf( "*" );
puts( "\nThis is a sample function" );
{
char star = '*';
puts( "\nIt has a problem\n" );
for ( int ctr2 = 0; ctr2 < 25; ctr2++ )
{
printf( "%c", star);
}
}
}
```

7. **BUG BUSTER:** What is wrong with the following code?

```
/*Count the number of even numbers between 0 and 100.
*/
#include <stdio.h>
main()
{
int x = 1;
static int tally = 0;
for (x = 0; x < 101; x++)
{
if (x % 2 == 0) /*if x is even...*/
tally++;.. /*add 1 to tally.*/
}
printf("There are %d even numbers.\n", tally);
return 0;
}
```

8. **BUG BUSTER:** Is anything wrong with the following program?

```
#include <stdio.h>
void print_function( char star );
int ctr;
main()
{
char star;
print_function( star );
```

```
return 0;
}
void print_function( char star )
{
char dash;
for ( ctr = 0; ctr < 25; ctr++ )
{
printf( "%c%c", star, dash );
}
}
```

9. What does the following program print? Don't run the program—try to figure it out by reading the code.

```
#include <stdio.h>
void print_letter2(void); /* function prototype */
int ctr;
char letter1 = 'X';
char letter2 = '=';
main()
{
for( ctr = 0; ctr < 10; ctr++ )
{
printf( "%c", letter1 );
print_letter2();
}
return 0;
}
void print_letter2(void)
{
for( ctr = 0; ctr < 2; ctr++ )
printf( "%c", letter2 );
}
```

10. **BUG BUSTER:** Will the preceding program run? If not, what's the problem? Rewrite it so that it is -correct.

Notes

LESSON 23 Input and Output Redirection

Objectives

Upon completion of this Lesson, you should be able to:

- Know how to work with input and output redirection.
- Know how printf() and standard error.
- Know what redirecting input is?

Input and Output Redirection

A program that uses stdin and stdout can utilize an operating-system feature called redirection. Redirection allows you to do the following:

- Output sent to stdout can be sent to a disk file or the printer rather than to the screen.
- Program input from stdin can come from a disk file rather than from the keyboard.

You don't code redirection into your programs; you specify it on the command line when you run the program. In DOS, as in UNIX, the symbols for redirection are < and >. I'll discuss redirection of output first.

Remember your first C program, HELLO.C? It used the printf() library function to display the message Hello, world on-screen. As you now know, printf() sends output to stdout, so it can be redirected. When you enter the program name at the command-line prompt, follow it with the > symbol and the name of the new destination:

hello > destination

Thus, if you enter hello >prn, the program output goes to the printer instead of to the screen (prn is the DOS name for the printer attached to port LPT1:). If you enter hello >hello.txt, the output is placed in a disk file with the name HELLO.TXT.

When you redirect output to a disk file, be careful. If the file already exists, the old copy is deleted and replaced with the new file. If the file doesn't exist, it is created. When redirecting output to a file, you can also use the >> symbol. If the specified destination file already exists, the program output is appended to the end of the file. The program demonstrates redirection.

The redirection of input and output.

```
1: /* Can be used to demonstrate redirection of stdin and
   stdout. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: char buf[80];
8:
9: gets(buf);
10: printf("The input was: %s\n", buf);
11: return 0;
12: }
```

Analysis

This program accepts a line of input from stdin and then sends the line to stdout, preceding it with The input was:. After compiling and linking the program, run it without redirection (assuming that the program is named LIST1414) by entering LIST1414 at the command-line prompt. If you then enter I am teaching myself C, the program displays the following on-screen:

The input was: I am teaching myself C

If you run the program by entering LIST1414 >test.txt and make the same entry, nothing is displayed on-screen. Instead, a file named TEST.TXT is created on the disk. If you use the DOS TYPE (or an equivalent) command to display the contents of the file:

type test.txt

you'll see that the file contains only the line The input was: I am teaching myself C. Similarly, if you had run the program by entering LIST1414 >prn, the output line would have been printed on the printer (prn is a DOS command name for the printer).

Run the program again, this time redirecting output to TEST.TXT with the >> symbol. Instead of the file's getting replaced, the new output is appended to the end of TEST.TXT.

Redirecting Input

Now let's look at redirecting input. First you need a source file. Use your editor to create a file named INPUT.TXT that contains the single line William Shakespeare. Now run Listing 14.14 by entering the following at the DOS prompt:

list1414 < INPUT.TXT

The program doesn't wait for you to make an entry at the keyboard. Instead, it immediately displays the following message on-screen:

The input was: William Shakespeare

The stream stdin was redirected to the disk file INPUT.TXT, so the program's call to gets() reads one line of text from the file rather than the keyboard. You can redirect input and output at the same time. Try running the program with the following command to redirect stdin to the file INPUT.TXT and redirect stdout to JUNK.TXT:

list1414 < INPUT.TXT > JUNK.TXT

Redirecting stdin and stdout can be useful in certain situations. A sorting program, for example, could sort either keyboard input or the contents of a disk file. Likewise, a mailing list program could display addresses on-screen, send them to the

printer for mailing labels, or place them in a file for some other use.

Note

Remember that redirecting stdin and stdout is a feature of the operating system and not of the C language itself. However, it does provide another example of the flexibility of streams. You can check your operating system documentation for more information on redirection.

When to Use fprintf()

As mentioned earlier, the library function fprintf() is identical to printf(), except that you can specify the stream to which output is sent. The main use of fprintf() involves disk files, as explained on Day 16. There are two other uses, as explained here.

Using stderr

One of C's predefined streams is stderr (standard error). A program's error messages traditionally are sent to the stream stderr and not to stdout. Why is this?

As you just learned, output to stdout can be redirected to a destination other than the display screen. If stdout is redirected, the user might not be aware of any error messages the program sends to stdout. Unlike stdout, stderr can't be redirected and is always connected to the screen (at least in DOS—UNIX systems might allow redirection of stderr). By directing error messages to stderr, you can be sure the user always sees them. You do this with fprintf():

```
fprintf(stderr, "An error has occurred.");
```

You can write a function to handle error messages and then call the function when an error occurs rather than calling fprintf():

```
error_message("An error has occurred.");

void error_message(char *msg)
{
fprintf(stderr, msg);
}
```

By using your own function instead of directly calling fprintf(), you provide additional flexibility (one of the advantages of structured programming). For example, in special circumstances you might want a program's error messages to go to the printer or a disk file. All you need to do is modify the error_message() function so that the output is sent to the desired destination.

Printer Output Under DOS

On a DOS or Windows system, you send output to your printer by accessing the predefined stream stdprn. On IBM PCs and compatibles, the stream stdprn is connected to the device LPT1: (the first parallel printer port). The below program presents a simple example.

Note

To use stdprn, you need to turn ANSI compatibility off in your compiler. Consult your compiler's manuals for more information.

Sending output to the printer.

```
1: /* Demonstrates printer output. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: float f = 2.0134;
8:
9: fprintf(stdprn, "\nThis message is printed.\r\n");
10: fprintf(stdprn, "And now some numbers:\r\n");
11: fprintf(stdprn, "The square of %f is %f.", f, f*f);
12:
13: /* Send a form feed. */
14: fprintf(stdprn, "\f");
15:
16: return 0;
17: }
```

Output

```
This message is printed.
And now some numbers:
The square of 2.013400 is 4.053780.
```

Note

This output is printed by the printer. It won't appear on-screen.

Analysis

If your DOS system has a printer connected to port LPT1:, you can compile and run this program.It prints three lines on the page. Line 14 sends an "\f" to the printer. \f is the escape sequence for a form feed, the command that causes the printer to advance a page (or, in the case of a laser printer, to eject the current page).

DON'T ever try to redirect stderr.

DO use fprintf() to create programs that can send output to stdout, stderr, stdprn, or any other stream.

DO use fprintf() with stderr to print error messages to the screen.

DON'T use stderr for purposes other than printing error messages or warnings.

DO create functions such as error_message to make your code more structured and maintainable.

Summary

This was a long day full of important information on program input and output. You learned how C uses streams, treating all input and output as a sequence of bytes. You also learned that C has five predefined streams:

stdin	The keyboard
stdout	The screen
stderr	The screen
stdprn	The printer
stdaux	The communications port

Input from the keyboard arrives from the stream stdin. Using C's standard library functions, you can accept keyboard input character by character, a line at a time, or as formatted numbers and strings. Character input can be buffered or unbuffered, echoed or unechoed.

Output to the display screen is normally done with the stdout stream. Like input, program output can be by character, by line, or as formatted numbers and strings. For output to the printer, you use fprintf() to send data to the stream stdprn.

When you use stdin and stdout, you can redirect program input and output. Input can come from a disk file rather than the keyboard, and output can go to a disk file or to the printer rather than to the display screen.

Finally, you learned why error messages should be sent to the stream stderr instead of stdout. Because stderr is usually connected to the display screen, you are assured of seeing error messages even when the program output is redirected.

Q&A

Q What happens if I try to get input from an output stream?

A You can write a C program to do this, but it won't work. For example, if you try to use stdprn with fscanf(), the program compiles into an executable file, but the printer is incapable of sending input, so your program doesn't operate as intended.

Q What happens if I redirect one of the standard streams?

A Doing this might cause problems later in the program. If you redirect a stream, you must put it back if you need it again in the same program. Many of the functions described in this chapter use the standard streams. They all use the same streams, so if you change the stream in one place, you change it for all the functions. For example, assign stdout equal to stdprn in one of the listings in this chapter and see what happens.

Q Is there any danger in using non-ANSI functions in a program?

A Most compilers come with many useful functions that aren't ANSI-standard. If you plan on always using that compiler and not porting your code to other compilers or platforms, there won't be a problem. If you're going to use other compilers and platforms, you should be concerned with ANSI compatibility.

Q Why shouldn't I always use fprintf() instead of printf()? Or fscanf() instead of scanf()?

A If you're using the standard output or input streams, you should use printf() and scanf(). By using these simpler functions, you don't have to bother with any other streams.

Workshop The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is a stream, and what does a C program use streams for?
2. Are the following input devices or output devices?
   a. Printer
   b. Keyboard
   c. Modem
   d. Monitor
   e. Disk drive
3. List the five predefined streams and the devices with which they are associated.
4. What stream do the following functions use?
   a. printf()
   b. puts()
   c. scanf()
   d. gets()
   e. fprintf()
5. What is the difference between buffered and unbuffered character input from stdin?
6. What is the difference between echoed and unechoed character input from stdin?
7. Can you "unget" more than one character at a time with ungetc()? Can you "unget" the EOF character?
8. When you use C's line input functions, how is the end of a line determined?
9. Which of the following are valid type specifiers?
   a. "%d"
   b. "%4d"
   c. "%3i%c"
   d. "%q%d"
   e. "%%%i"
   f. "%9ld"
10. What is the difference between stderr and stdout?

Exercises

1. Write a statement to print "Hello World" to the screen.
2. Use two different C functions to do the same thing the function in exercise 1 did.
3. Write a statement to print "Hello Auxiliary Port" to the standard auxiliary port.
4. Write a statement that gets a string 30 characters or shorter. If an asterisk is encountered, truncate the string.
5. Write a single statement that prints the following:

   Jack asked, "What is a backslash?"

   Jill said, "It is `\'"

   Because of the multitude of possibilities, answers are not provided for the following exercises; however, you should attempt to do them.

6. **ON YOUR OWN**: Write a program that redirects a file to the printer one character at a time.
7. **ON YOUR OWN:** Write a program that uses redirection to accept input from a disk file, counts the number of times each letter occurs in the file, and then displays the results on-screen. (A hint is provided in Appendix G, "Answers.")

8. **ON YOUR OWN:** Write a program that prints your C source files. Use redirection to enter the source file, and use fprintf() to do the printing.
9. **ON YOUR OWN:** Modify the program from exercise 8 to put line numbers at the beginning of the listing when it is printed. (A hint is provided in Appendix G.)
10. **ON YOUR OWN:** Write a "typing" program that accepts keyboard input, echoes it to the screen, and then reproduces this input on the printer. The program should count lines and advance the paper in the printer to a new page when necessary. Use a function key to terminate the program.

Notes

LESSON 24 Command Line Arguments

Objectives

Upon completion of this Lesson, you should be able to:

- Know about command line arguments.
- Know about Passing command-line arguments to main()

Command Line Arguments

Your C program can access arguments passed to the program on the command line. This refers to information entered after the program name when you start the program. If you start a program named PROGNAME from the C:\> prompt, for example, you could enter

C:\>progname smith jones

The two command-line arguments smith and jones can be retrieved by the program during execution. You can think of this information as arguments passed to the program's main() function. Such command-line arguments permit information to be passed to the program at startup rather than during execution, which can be convenient at times. You can pass as many command-line arguments as you like. Note that command-line arguments can be retrieved only within main(). To do so, declare main() as follows:

```
main(int argc, char *argv[])
        {
/* Statements go here */
        }
```

The first parameter, argc, is an integer giving the number of command-line arguments available. This value is always at least 1, because the program name is counted as the first argument. The parameter argv[] is an array of pointers to strings. The valid subscripts for this array are 0 through argc - 1. The pointer argv[0] points to the program name (including path information), argv[1] points to the first argument that follows the program name, and so on. Note that the names argc and argv[] aren't required—you can use any valid C variable names you like to receive the command-line arguments. However, these two names are traditionally used for this purpose, so you should probably stick with them. The command line is divided into discrete arguments by any white space. If you need to pass an argument that includes a space, enclose the entire argument in double quotation marks. For example, if you enter

C:>progname smith "and jones"

smith is the first argument (pointed to by argv[1]), and and jones is the second (pointed to by argv[2]). The program given below demonstrates how to access command-line arguments.

Passing command-line arguments to main().

```
1: /* Accessing command-line arguments. */
2:
3: #include <stdio.h>
4:
5: main(int argc, char *argv[])
6: {
7: int count;
8:
9: printf("Program name: %s\n", argv[0]);
10:
11: if (argc > 1)
12: {
13: for (count = 1; count < argc; count++)
14: printf("Argument %d: %s\n", count, argv[count]);
15: }
16: else
17: puts("No command line arguments entered.");
18: return(0);
19: }
```

Output

```
list21_6
Program name: C:\LIST21_6.EXE
No command line arguments entered.
list21_6 first second "3 4"
Program name: C:\LIST21_6.EXE
Argument 1: first
Argument 2: second
Argument 3: 3  4
```

Analysis

This program does no more than print the command-line parameters entered by the user. Notice that line 5 uses the argc and argv parameters shown previously. Line 9 prints the one command-line parameter that you always have, the program name. Notice this is argv[0]. Line 11 checks to see whether there is more than one command-line parameter. Why more than one and not more than zero? Because there is always at least one—the program name. If there are additional arguments, a for loop prints each to the screen (lines 13 and 14).Otherwise, an appropriate message is printed (line 17).

Command-line arguments generally fall into two categories: those that are required because the program can't operate without them, and those that are optional, such as flags that instruct the program to act in a certain way. For example, imagine a program that sorts the data in a file. If you write the program to receive the input filename from the command line, the name is required information. If the user forgets to enter the input filename on the command line, the program must somehow deal with the situation. The program could also look for the argument /r, which signals a reverse-order sort. This

argument isn't required; the program looks for it and behaves one way if it's found and another way if it isn't.

- **DO** use argc and argv as the variable names for the command-line arguments for main(). Most C programmers are familiar with these names.
- **DON'T** assume that users will enter the correct number of command-line parameters. Check to be sure they did, and if not, display a message explaining the arguments they should enter.

Summary

This chapter covered some of the more advanced programming tools available with C compilers. You learned how to write a program that has source code divided among multiple files or modules. This practice, called modular programming, makes it easy to reuse general-purpose functions in more than one program. You saw how you can use preprocessor directives to create function macros, for conditional compilation, and other tasks. Finally, you saw that the compiler provides some function macros for you.

Q&A

Q When compiling multiple files, how does the compiler know which filename to use for the executable file?

A You might think the compiler uses the name of the file containing the main() function; however, this isn't usually the case. When compiling from the command line, the first file listed is used to determine the name. For example, if you compiled the following with Borland's Turbo C, the executable would be called FILE1.EXE:

```
tcc file1.c main.c prog.c
```

Q Do header files need to have an .h extension?

A No. You can give a header file any name you want. It is standard practice to use the .H extension.

Q When including header files, can I use an explicit path?

A Yes. If you want to state the path where a file to be included is, you can. In such a case, you put the name of the include file between quotation marks.

Notes

LESSON 25 Introduction to Structures and Unions

Objectives

Upon completion of this Lesson, you should be able to:

- Know about structures and unions.
- Know how to access structure members.
- Know about structures within structures.

Structures and Unions

Structure

Many programming tasks are simplified by the C data constructs called structures. A structure is a data storage method designed by you, the programmer, to suit your programming needs exactly. Today you will learn What simple and complex structures are?

- How to define and declare structures.
- How to access data in structures.
- How to create structures that contain arrays and arrays of structures.
- How to declare pointers in structures and pointers to structures.
- How to pass structures as arguments to functions.
- How to define, declare, and use unions.
- How to use type definitions with structures.

Simple Structures

A structure is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a member of the structure. The next section shows a simple example. You should start with simple structures. Note that the C language makes no distinction between simple and complex structures, but it's easier to explain structures in this way.

Defining and Declaring Structures

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

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

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or tag (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.

The preceding statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. In other words, they don't declare (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here:

```
struct coord
{
    int x;
    int y;
 } first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. First and second are each instances of type coord; first contains two integer members named x and y, and so does second.

This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord:

```
struct coord
{
     int x;
     int y;
 };
/* Additional code may go here */
struct coord first, second;
```

Accessing Structure Members

Individual structure members can be used like other variables of the same type. Structure members are accessed using the structure member operator (.), also called the dot operator, between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major

advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

The struct Keyword

```
struct tag
{
structure_member(s);
/* additional statements may go here */
} instance;
```

The struct keyword is used to declare structures. A structure is a collection of one or more variables (structure_members) that have been grouped under a single name for easy man-ipulation. The variables don't have to be of the same variable type, nor do they have to be simple variables. Structures also can hold arrays, pointers, and other structures. The keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure. Following the tag are the structure members, enclosed in braces. An instance, the actual declaration of a structure, can also be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. Here is a template's format:

```
struct tag
 {
structure_member(s);
/* additional statements may go here */
  };
```

To use the template, you use the following format:

```
struct tag instance;
```

To use this format, you must have previously declared a structure with the given tag.

Example 1

```
/* Declare a structure template called SSN */
struct SSN
 {
int first_three;
char dash1;
int second_two;
char dash2;
int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Example 2

```
/* Declare a structure and instance together */
struct date
{
char month[2];
char day[2];
char year[4];
 } current_date;
```

Example 3

```
/* Declare and initialize a structure */
struct time
 {
int hours;
int minutes;
int seconds;
} time_of_birth = { 8, 45, 0 };
```

More-Complex Structures

Now that you have been introduced to simple structures, you can get to the more interesting and complex types of structures. These are structures that contain other structures as members and structures that contain arrays as members.

Structures that Contain Structures

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this.

Assume that your graphics program needs to deal with rectangles. A rectangle can be defined by the coordinates of two diagonally opposite corners. You've already seen how to define a structure that can hold the two coordinates required for a single point. You need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle
 {
   struct coord topleft;
   struct coord bottomrt;
 };
```

This statement defines a structure of type rectangle that contains two structures of type coord. These two type coord structures are named topleft and bottomrt.

The preceding statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as

```
struct rectangle mybox;
```

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle
 {
struct coord topleft;
struct coord bottomrt;
 } mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus,the expression

mybox.topleft.x

refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Maybe this is getting a bit confusing. You might understand better if you look at Figure given below, which shows the relationship between the type rectangle structure, the two type coord structures it contains, and the two type int variables each type coord structure contains. These structures are named as in the preceding example.

mybox.topleft.x

A type rectangle structure

(mybox)

A type coord

Structure(topleft)

mybox.topleft.y

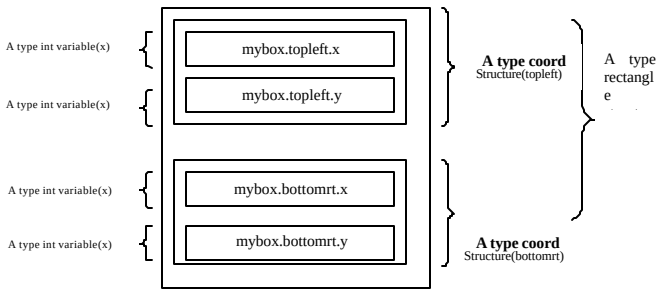
mybox.topleft.x

mybox.bottomrt.x

A type coord

Structure(bottomrt)

mybox.bottomrt.y



Let's look at an example of using structures that contain other structures. The program given below takes input from the user for the coordinates of a rectangle and then calculates and displays the rectangle's area. Note the program's assumptions, given in comments near the start of the program (lines 3 through 8). In the figure given above gives the relationship between a structure, structures within a structure, and the structure members.

A demonstration of structures that contain other structures.

```
1:  /* Demonstrates structures that contain other structures. */
2:
3:  /* Receives input for corner coordinates of a rectangle and
4:  calculates the area. Assumes that the y coordinate of the
5:  upper-left corner is greater than the y coordinate of the
6:  lower-right corner, that the x coordinate of the lower-
7:  right corner is greater than the x coordinate of the upper-
8: left corner, and that all coordinates are positive. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16: int x;
17: int y;
18: };
19:
20: struct rectangle{
21: struct coord topleft;
22: struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27: /* Input the coordinates */
28:
29: printf("\nEnter the top left x coordinate: ");
30: scanf("%d", &mybox.topleft.x);
31:
32: printf("\nEnter the top left y coordinate: ");
33: scanf("%d", &mybox.topleft.y);
34:
35: printf("\nEnter the bottom right x coordinate: ");
36: scanf("%d", &mybox.bottomrt.x);
37:
38: printf("\nEnter the bottom right y coordinate: ");
39: scanf("%d", &mybox.bottomrt.y);
40:
41: /* Calculate the length and width */
42:
43: width = mybox.bottomrt.x - mybox.topleft.x;
```

```
44: length = mybox.bottomrt.y - mybox.topleft.y;
45:
46: /* Calculate and display the area */
47:
48: area = width * length;
49: printf("\nThe area is %ld units.\n", area);
50:
51: return 0;
52: }
```

Output

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

Analysis

The coord structure is defined in lines 15 through 18 with its two members, x and y. Lines 20 through 23 declare and define an instance, called mybox, of the rectangle structure. The two members of the rectangle structure are topleft and bottomrt, both structures of type coord. Lines 29 through 39 fill in the values in the mybox structure. At first it might seem that there are only two values to fill, because mybox has only two members. However, each of mybox's members has its own members. topleft and bottomrt have two members each, x and y from the coord structure. This gives a total of four members to be filled. After the members are filled with values, the area is calculated using the structure and member names. When using the x and y values, you must include the structure instance name. Because x and y are in a structure within a structure, you must use the instance names of both structures—mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x, and mybox.topleft.y—in the calculations.

C places no limits on the nesting of structures. While memory allows, you can define structures that contain structures that contain structures that contain structures—well, you get the idea! Of course, there's a limit beyond which nesting becomes unproductive. Rarely are more than three levels of nesting used in any C program.

Structures that Contain Arrays

You can define a structure that contains one or more arrays as members. The array can be of any C data type (int, char, and so on). For example, the statements define a structure of

```
struct data
{
int x[4];
char y[10];
};
```

type data that contains a four-element integer array member named x and a 10-element character array member named y. You can then declare a structure named record of type data as follows:

```
struct data record;
```

The organization of this structure is shown in figure given below. Note that, in this figure, the elements of array x take up twice as much space as the elements of array y. This is because a type int typically requires two bytes of storage, whereas a type char usually requires only one byte

```
record.x[0]
record
record.x
record.y
record.y[8]
```

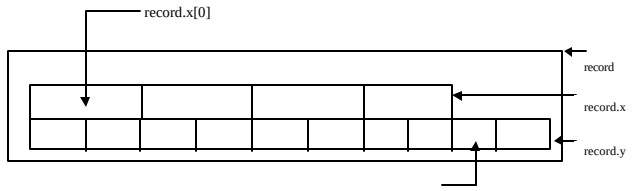


Figure: The organization of a structure that contains arrays as members.

You access individual elements of arrays that are structure members using a combination of the member operator and array subscripts:

```
record.x[2] = 100;
record.y[1] = 'x';
```

You probably remember that character arrays are most frequently used to store strings. You should also remember (from Day 9, "Understanding Pointers") that the name of an array, without brackets, is a pointer to the array. Because this holds true for arrays that are structure members, the expression

```
record.y
```

is a pointer to the first element of array y[] in the structure record. Therefore, you could print the contents of y[] on-screen using the statement

```
puts(record.y);
```

Now look at another example. The program given below uses a structure that contains a type float variable and two type char arrays.

A structure that contains array members.

```
1:  /* Demonstrate a structure that has array members. */
2:  #include<stdio.h>
3:
4:
5:  /* define and declare a structure to hold the data. */
6:  /* it contain one float variable and two char arrays. */
7:
8:  struct data {
9:  float amount;
10: char frame[30];
11: char lname[30];
```

```
12: } rec;
13:
14: int main()
15: {
16: /* Input the data from the keyboard. */
17:
18: printf("Enter the door`s first and last names, \n");
19: printf("separated by a space: ");
20: scanf(" %s %s", rec.fname, rec.lname);
21:
22: printf("\n enter the donation amount:");
23: scanf("%f", &rec.amount);
24:
25: /* Display the information */
26: /*note  : %.2f specifies a floating-point value */
27: /* to be displayed with two digits to the right */
28: /*of the decimal point. /*
29:
30: /* Display the data on the screen. */
31:
32: printf(" \n Donor  %s %s gave  $%.2f.\n ", rec.fname,
    rec.lname,
33: rec.amount);
34:
35: return 0;
36: }
```

Analysis

This program includes a structure that contains array members named fname[30] and lname[30]. Both are arrays of characters that hold a person's first name and last name, respectively. The structure declared in lines 8 through 12 is called data. It contains the fname and lname character arrays with a type float variable called amount. This structure is ideal for holding a person's name (in two parts, first name and last name) and a value, such as the amount the person donated to a charitable organization.

An instance of the array, called rec, has also been declared in line 12. The rest of the program uses rec to get values from the user (lines 18 through 23) and then print them (lines 32 and 33).

Notes

LESSON 26 Arrays of Structures

Objectives

Upon completion of this Lesson, you should be able to:

- Know about arrays of structures.
- Know how to initialize structure.
- Know about bit fields in structures.

Arrays of Structures

If you can have structures that contain arrays, can you also have arrays of structures? You bet you can! In fact, arrays of structures are very powerful programming tools. Here's how it's done.

You've seen how a structure definition can be tailored to fit the data your program needs to work with. Usually a program needs to work with more than one instance of the data. For example, in a program to maintain a list of phone numbers, you can define a structure to hold each person's name and number:

```
struct entry{
char fname[10];
char lname[12];
char phone[8];
};
```

A phone list must hold many entries, however, so a single instance of the entry structure isn't of much use. What you need is an array of structures of type entry. After the structure has been defined, you can declare an array as follows:

```
struct entry list[1000];
```

This statement declares an array named list that contains 1,000 elements. Each element is a structure of type entry and is identified by subscript like other array element types. Each of these structures has three elements, each of which is an array of type char. This entire complex creation is diagrammed in Figure given below.

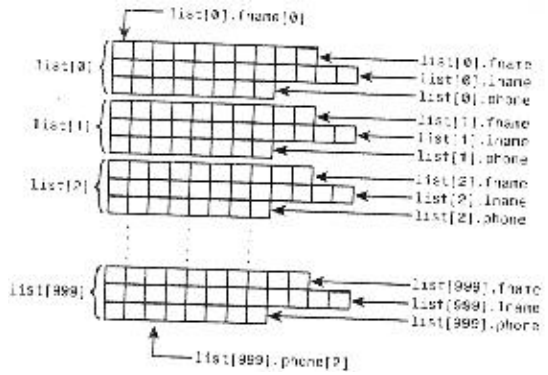


Figure: The organization of the array of structures defined in the text.

When you have declared the array of structures, you can manipulate the data in many ways. For example, to assign the data in one array element to another array element, you would write

```
list[1] = list[5];
```

This statement assigns to each member of the structure list[1] the values contained in the corresponding members of list[5]. You can also move data between individual structure members. The statement

```
strcpy(list[1].phone, list[5].phone);
```

copies the string in list[5].phone to list[1].phone. (The strcpy() library function copies one string to another string.) If you want to, you can also move data between individual elements of the structure member arrays:

```
list[5].phone[1] = list[2].phone[3];
```

This statement moves the second character of list[5]'s phone number to the fourth position in list[2]'s phone number. (Don't forget that subscripts start at offset 0.)

The program given below demonstrates the use of arrays of structures. Moreover, it demonstrates arrays of structures that contain arrays as members.

Arrays of structures

```
1: /* Demonstrates using arrays of structures. */
2:
3: #include <stdio.h>
4:
5: /* Define a structure to hold entries. */
6:
7: struct entry {
8: char fname[20];
9: char lname[20];
10: char phone[10];
11: };
12:
13: /* Declare an array of structures. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22: /* Loop to input data for four people. */
```

```
23:
24: for (i = 0; i < 4; i++)
25: {
26: printf("\nEnter first name: ");
27: scanf("%s", list[i].fname);
28: printf("Enter last name: ");
29: scanf("%s", list[i].lname);
30: printf("Enter phone in 123-4567 format: ");
31: scanf("%s", list[i].phone);
32: }
33:
34: /* Print two blank lines. */
35:
36: printf("\n\n");
37:
38: /* Loop to display data. */
39:
40: for (i = 0; i < 4; i++)
41: {
42: printf("Name: %s %s", list[i].fname, list[i].lname);
43: printf("\t\tPhone: %s\n", list[i].phone);
44: }
45:
46: return 0;
47: }
```

Output

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434
Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Deanna
Enter last name: Townsend
Enter phone in 123-4567 format: 555-1234
Name: Bradley Jones                    Phone: 555-1212
Name: Peter Aitken                     Phone: 555-3434
Name: Melissa Jones                    Phone: 555-1212
Name: Deanna Townsend          Phone: 555-1234
```

Analysis

This program follows the same general format as most of the other listings. It starts with the comment in line 1 and, for the input/output functions, the #include file STDIO.H in line 3. Lines 7 through 11 define a template structure called entry that contains three character arrays: fname, lname, and phone. Line 15 uses the template to define an array of four entry structure variables called list. Line 17 defines a variable of type int to be used as a counter throughout the program. main() starts in line 19. The first function of main() is to perform a loop four times with a for statement. This loop is used to get information for the array of structures. This can be seen in lines 24 through 32. Line 36 provides a break from the input before starting with the output. It prints two new lines in a manner that shouldn't be new to you. Lines 40 through 44 display the data that the user entered in the preceding step. The values in the array of structures are printed with the subscripted array name followed by the member operator (.) and the structure member name.

Familiarize yourself with the techniques used in program given above. Many real-world programming tasks are best accomplished by using arrays of structures containing arrays as members.

- **DON'T** forget the structure instance name and member operator (.) when using a structure's members.
- **DON'T** confuse a structure's tag with its instances! The tag is used to declare the structure's template, or format. The instance is a variable declared using the tag.
- **DON'T** forget the struct keyword when declaring an instance from a previously defined structure.
- **DO** declare structure instances with the same scope rules as other variables.

Initializing Structures

Like other C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values separated by commas and enclosed in braces.

For example, look at the following statements:

```
1: struct sale {
2: char customer[20];
3: char item[20];
4: float amount;
5: } mysale = { "Acme Industries",
6: "Left-handed widget",
7: 1000.00
8: };
```

When these statements are executed, they perform the following actions:

1. Define a structure type named sale (lines 1 through 5).
2. Declare an instance of structure type sale named mysale (line 5).
3. Initialize the structure member mysale.customer to the string "Acme Industries"(line 5).
4. Initialize the structure member mysale.item to the string "Left-handed widget" (line 6).
5. Initialize the structure member mysale.amount to the value 1000.00 (line 7).

For a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

```
1: struct customer {
2: char firm[20];
3: char contact[25];
4: }
5:
6: struct sale {
7: struct customer buyer;
8: char item[20];
9: float amount;
10: } mysale = { { "Acme Industries", "George Adams"},
11: "Left-handed widget",
12: 1000.00
13: };
```

These statements perform the following initializations:

1. The structure member mysale.buyer.firm is initialized to the string "Acme Industries" (line 10).
2. The structure member mysale.buyer.contact is initialized to the string "George Adams" (line 10).
3. The structure member mysale.item is initialized to the string "Left-handed widget" (line 11).
4. The structure member mysale.amount is initialized to the amount 1000.00 (line 12).

You can also initialize arrays of structures. The initialization data that you supply is applied, in order, to the structures in the array. For example, to declare an array of structures of type sale and initialize the first two array elements (that is, the first two structures), you could write

```
1: struct customer {
2: char firm[20];
3: char contact[25];
4: };
5:
6: struct sale {
7: struct customer buyer;
8: char item[20];
9: float amount;
10: };
11:
12:
13: struct sale y1990[100] = {
14: { { "Acme Industries", "George Adams"},
15: "Left-handed widget",
16: 1000.00
17: }
18: { { "Wilson & Co.", "Ed Wilson"},
19: "Type 12 gizmo",
20: 290.00
21: }
22: };
```

This is what occurs in this code:

1. The structure member y1990[0].buyer.firm is initialized to the string "Acme Industries" (line 14).
2. The structure member y1990[0].buyer.contact is initialized to the string "George Adams" (line 14).
3. The structure member y1990[0].item is initialized to the string "Left-handed widget" (line 15).
4. The structure member y1990[0].amount is initialized to the amount 1000.00 (line 16).
5. The structure member y1990[1].buyer.firm is initialized to the string "Wilson & Co." (line 18).
6. The structure member y1990[1].buyer.contact is initialized to the string "Ed Wilson" (line 18).
7. The structure member y1990[1].item is initialized to the string "Type 12 gizmo" (line 19).
8. The structure member y1990[1].amount is initialized to the amount 290.00 (line 20).

Bit Fields in Structures

The final bit-related topic is the use of bit fields in structures. In "Structures," you learned how to define your own data structures, customizing them to fit your program's data needs. By using bit fields, you can accomplish even greater customization and save memory space as well.

A *bit field* is a structure member that contains a specified number of bits. You can declare a bit field to contain one bit, two bits, or whatever number of bits are required to hold the data stored in the field. What advantage does this provide?

Suppose that you're programming an employee database program that keeps records on your company's employees. Many of the items of information that the database stores are of the yes/no variety, such as "Is the employee enrolled in the dental plan?" or "Did the employee graduate from college?" Each piece of yes/no information can be stored in a single bit, with 1 representing yes and 0 representing no.Using C's standard data types, the smallest type you could use in a structure is a type char. You could indeed use a type char structure member to hold yes/no data, but seven of the char's eight bits would be wasted space. By using bit fields, you could store eight yes/no values in a single char.

Bit fields aren't limited to yes/no values. Continuing with this database example, imagine that your firm has three different health insurance plans. Your database needs to store data about the plan in which each employee is enrolled (if any). You could use 0 to represent no health insurance and use the values 1, 2, and 3 to represent the three plans. A bit field containing two bits is sufficient, because two binary bits can represent values of 0 through 3. Likewise, a bit field containing three bits could hold values in the range 0 through 7, four bits could hold values in the range 0 through 15, and so on.

Bit fields are named and accessed like regular structure members. All bit fields have type unsigned int, and you specify the size of the field (in bits) by following the member name with a colon and the number of bits. To define a structure with a one-bit member named dental, another one-bit member named college, and a two-bit member named health, you would write the following:

```
struct emp_data
{
unsigned dental : 1;
unsigned college : 1;
unsigned health : 2;
...
};
```

The ellipsis (...) indicates space for other structure members. The members can be bit fields or fields made up of regular data types. Note that bit fields must be placed first in the structure definition, before any nonbit field structure members. To access the bit fields, use the structure member operator just as you do with any structure member. For the example, you can expand the structure definition to something more useful:

```
struct emp_data
{
unsigned dental : 1;
unsigned college : 1;
unsigned health : 2;
char fname[20];
char lname[20];
char ssnumber[10];
};
```

You can then declare an array of structures:

```
struct emp_data workers[100];
```

To assign values to the first array element, write something like this:

```
workers[0].dental = 1;
workers[0].college = 0;
workers[0].health = 2;
strcpy(workers[0].fname, "Mildred");
```

Your code would be clearer, of course, if you used symbolic constants YES and NO with values of 1 and 0 when working with one-bit fields. In any case, you treat each bit field as a small, unsigned integer with the given number of bits. The range of values that can be assigned to a bit field with *n* bits is from 0 to 2*n*-1. If you try to assign an out-of-range value to a bit field, the compiler won't report an error, but you will get unpredictable results.

- **DO** use defined constants YES and NO or TRUE and FALSE when working with bits. These are much easier to read and understand than 1 and 0.
- **DON'T** define a bit field that takes 8 or 16 bits. These are the same as other available variables such as type char or int. In a scalar data object, the smallest object that can be addressed directly is usually a byte.

In a structure, you can define data objects from 1 to 16 bits long. Suppose your program contains a number of TRUE/FALSE variables grouped in a structure called Status, as follows:

```
struct
{
unsigned int bIsValid;
unsigned int bIsFullSize;
unsigned int bIsColor;
unsigned int bIsOpen;
unsigned int bIsSquare;
unsigned int bIsSoft;
unsigned int bIsLong;
unsigned int bIsWide;
unsigned int bIsBoxed;
unsigned int bIsWindowed;
} Status;
```

This structure requires 20 bytes of storage, which is a lot of memory for saving a few TRUE/FALSE variables. It would be better to save each variable using only one bit. Perhaps you could use a single, bit-mapped variable. Sometimes, however, your flags must keep the identity that a unique name offers.

C offers the capability to define the width of a variable, but only when the variable is in a structure called a *bit field*. For example, you could rewrite the definition of Status as follows:

```
struct
{
unsigned int bIsValid:1;
unsigned int bIsFullSize:1;
unsigned int bIsColor:1;
unsigned int bIsOpen:1;
unsigned int bIsSquare:1;
unsigned int bIsSoft:1;
unsigned int bIsLong:1;
unsigned int bIsWide:1;
unsigned int bIsBoxed:1;
unsigned int bIsWindowed:1;
} Status;
```

The :1 that appears after each variable's name tells the compiler to allocate one bit to the variable. Thus, the variable can hold only a 0 or a 1. This is exactly what is needed, however, because the variables are TRUE/FALSE variables. The structure is only two bytes long (one tenth the size of the previous example). A bit field can hold more than a single bit.

For example, it can hold a definition of a structure member, such as

```
unsigned int nThreeBits:3;
```

In this example, nThreeBits can hold any value from 0 to 7. The most critical limitation to using bit fields is that you cannot

determine the address of a bit field variable. If you use the address of operator, a compile-time error results. This means that you cannot pass a bit-field's address as a parameter to a function. When the compiler stores bit fields, it packs them into storage without regard to alignment. Therefore, storage is used most efficiently when all your bit fields are grouped together in the structure. You can force the compiler to pad the current word so that the next bit field starts on a word boundary. To do so, specify a dummy bit field with a width of 0,

For example:

```
struct
 {
unsigned int bIsValid:1;
unsigned int bIsFullSize:1;
unsigned int bReserved1:0;
unsigned int bIsBoxed:1;
unsigned int bIsWindowed:1;
} Status;
```

The bReserved1 bit field tells the compiler to pad to the next word boundary, which results in the bIsBoxed bit field starting on a known boundary. This techniqueis useful when the compiler is packing structures and you need to know that the alignment is as optimal as possible. (Some computers access objects faster when the objects are aligned on word or double word boundaries.)

Notes

LESSON 27 Introduction to typedef & Macro

Objectives

Upon completion of this Lesson, you should be able to:

- Know about how to use the typeedef keyword
- Know about macros.

Using the typedef Keyword

I think that the typedef keyword is one of the best parts of the C language. It enables you to create any data type from simple variables, arrays, structures, or unions. The typedef keyword is used to define a type of variable, just as its name implies. You can define any type from any other type. A variable created with typedef can be used just like any other variable. The program given below, CREATEDB.C, is a simple example of using typedef with structures.

```
/* CREATEDB.C
* This program demonstrates typedef. The program
* has minimal error checking; it will fail if
* you enter a field value that is too long for
* the structure member that holds the value.
* Use with caution!
*/.
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#define CUSTOMER_RECORD 1
#define SUPPLIER_RECORD 2
/* Define the structure for the customer database */
typedef struct _CUSTNAME
 {
int nRecordType; / / 1 == Customer record
char szName[61]; / / 60 chars for name; 1 for null at end
char szAddr1[61]; / / 60 chars for address; 1 for null at end
char szAddr2[61]; / / 60 chars for address; 1 for null at end
char szCity[26]; / / 25 characters for city; 1 for null at end
char szState[3]; / / 2-character state abbrev. plus null
int nZip; / / Use integer. Print as %5.5d for leading 0
int nRecordNumber; / / Which record number?
double dSalesTotal; / / Amount customer has purchased
} CUSTNAME;
typedef CUSTNAME near *NPCUSTNAME;
typedef CUSTNAME *PCUSTNAME;
void main()
{
FILE *DataFile;
CUSTNAME Customer;
char szFileName[25];
char szBuffer[129];
int i;
int nResult;
double dSales = 0.0; / / Forces loading of floating-point
support
printf("Please enter customer database name: ");
gets(szFileName);
DataFile = fopen(szFileName, "wb");
if (DataFile == NULL)
{
printf("ERROR: File '%s' couldn't be opened.\n",
szFileName);
exit(4);
}
Customer.szName[0] = 'A'; / / To get past while() the first time
i = 0;
Customer.nRecordNumber = 0;
while (strlen(Customer.szName) > 0)
{
memset(&Customer, 0, sizeof(CUSTNAME));
printf("Enter the Customer's name: ");
gets(Customer.szName);
if (strlen(Customer.szName) > 0)
{
Customer.nRecordNumber = i;
do
{
printf("Enter 1 for customer, 2 for supplier ");
gets(szBuffer);
sscanf(szBuffer, "%d", &Customer.nRecordType);
}
while (Customer.nRecordType != CUSTOMER_RECORD &&
Customer.nRecordType != SUPPLIER_RECORD);
printf("Enter address line 1: ");
gets(Customer.szAddr1);
printf("Enter address line 2: ");
gets(Customer.szAddr2);
printf("Enter City: ");
```

```
gets(Customer.szCity);
printf("Enter state postal abbreviation: ");
gets(Customer.szState);
printf("Enter ZIP code: ");
gets(szBuffer);
sscanf(szBuffer, "%d", &Customer.nZip);
printf("Enter total sales: ");
gets(szBuffer);
sscanf(szBuffer, "%f", &Customer.dSalesTotal);
nResult = fwrite((char *)&Customer, sizeof(CUSTNAME), 1,
DataFile);
if (nResult != 1)
{
printf("ERROR: File '%s', write error.\n",
szFileName);
fclose(DataFile);
exit(4);
}
++i;
}
}
fclose(DataFile);
}
```

In the program, the lines that define the structure that holds the customer's name and address use the typedef keyword. This enables us to define the data object using only one line of code:

```
CUSTNAME Customer;
```

This line creates a structure named Customer. As many different structures as needed could have been created using the name CUSTNAME. You access a structure created by a typedef in the same way as you access a structure created by any other method. However, now the compiler has a data type that it can work with, so you can obtain the size of the structure type by referring to its name. This is valuable when you must allocate memory for the structure—you cannot get the size from the object because it doesn't exist yet! The program clears the structure's contents to 0 by using sizeof() with the name:

```
memset(&Customer, 0, sizeof(CUSTNAME));
```

In the call to memset(), you must pass the address of the structure (&Customer), the value

that you are setting all the bytes to (0), and the size of the structure

(sizeof(CUSTNAME)).

The memset() C library function then stores the specified value in all the bytes in Customer. The rest of CREATEDB is straightforward. The program reads from the keyboard each field in the structure. Fields that are not character fields (such as dSalesTotal) are converted to the correct type for the field before being saved in the structure.

Using the offsetof() Macro

ANSI C introduced a new macro, called offsetof(), that you use to determine the offset of a member in a structure. There are many reasons for wanting to know the location of a member in a structure. You might want to write part of a structure to a disk file or read part of a structure in from the file. Using the offsetof() macro and simple math, it is easy to compute the amount of storage used by individual members of a structure. An example use of the offsetof() macro is shown in the program given below.

OFFSETOF.C.

```
/* OFFSETOF, written 1992 by Peter D. Hipson
* This program illustrates the use of the
* offsetof() macro.
*/
#include <stdio.h> // Make includes first part of file
#include <string.h> // For string functions
#include <stddef.h> // For offsetof()
#define MAX_SIZE 35
int main(void); // Define main(), and the fact that this
program doesn't
// use any passed parameters
int main()
{
int i;
typedef struct
{
char *szSaying[MAX_SIZE];
int nLength[MAX_SIZE];
} SAYING;
typedef struct
{
SAYING Murphy;
SAYING Peter;
SAYING Peter1;
SAYING Peter2;
SAYING Peter3;
SAYING Peter4;
} OURSAYING;
OURSAYING OurSaying = {{
"Firestone's Law of Forecasting:",
" Chicken Little has to be right only once.",
"",
"",
"Manly's Maxim:",
" Logic is a systematic method of coming to",
" the wrong conclusion with confidence.",
"",
```

```
“”,
“Moer’s truism:”,
“ The trouble with most jobs is the job holder’s”,
“ resemblance to being one of a sled dog team. No one”,
“ gets a change of scenery except the lead dog.”,
“”,
“”,
“Cannon’s Comment:”,
“ If you tell the boss you were late for work because you”,
“ had a flat tire, the next morning you will have a flat tire.”,
NULL /* Flag to mark the last saying */
}, {
“David’s rule:”,
“ Software should be as easy to use as a Coke machine.”,
“”,
“”,
“Peter’s Maxim:”,
“ To be successful, you must work hard, but”,
“ Hard work doesn’t guarantee success.”,
“”,
“”,
“Teacher’s truism:”,
“ Successful people learn.”,
“”,
“”,
“Player’s Comment:”,
“ If you don’t play to win,”,
“ you don’t win.”,
NULL /* Flag to mark the last saying */
}};
printf(
“sizeof(SAYING) = %d (each member’s size)\n”
“offsetof(OURSAYING, Peter) = %d (the second member)\n”
“offsetof(OURSAYING, Peter3) = %d (the fifth member)\n”,
sizeof(SAYING),
offsetof(OURSAYING, Peter),
offsetof(OURSAYING, Peter3));
return (0);
}
```

To use the offsetof() macro, you supply both the structure and the member name. In addition, the structure name must be created using typedef because the offsetof() macro must create the pointer type with a value of 0, and an identifier— not a variable name—is required.

Here is another use of the offsetof() macro. Suppose that a structure has 75 members that consist of strings, structures, and scalar variables. You want to save the middle 30 members in a file. You have to know the starting address and how many bytes to write to the file.

You could use the sizeof() keyword to compute the size of the block of memory to write, but this would be difficult and complex. You would have to get the size of each member that you want to save to the file, then add the results. Also, serious problems would result if members contained packing bytes (to align them on word boundaries). A better solution is to take the offsetof() of the first member to write and the offsetof() of the member just after the last member to write. Subtract one from the other, and you have the number of bytes to save. As you can see, this method is quick and easy.

Notes

LESSON 28 Details of Unions

Objectives

Upon completion of this Lesson, you should be able to:

- Know about unions.
- Know how to access union members.
- Know how to define, declare and initialize unions.

Unions

Unions are similar to structures. A union is declared and used in the same ways that a structure is. A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the same area of memory. They are laid on top of each other.

Defining, Declaring, and Initializing Unions

Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct. To define a simple union of a char variable and an integer variable, you would write the following:

```
union shared {
char c;
int i;
};
```

This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. Unlike a structure that would hold both values, the union can hold only one value at a time. Figure given below, illustrates how the shared union would appear in memory.

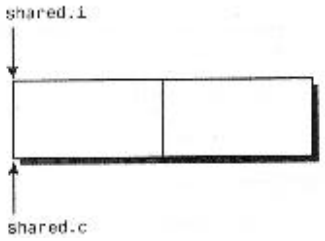


Figure : The union can hold only one value at a time.

A union can be initialized on its declaration. Because only one member can be used at a time, only one can be initialized. To avoid confusion, only the first member of the union can be initialized. The following code shows an instance of the shared union being declared and initialized:

```
union shared generic_variable = {'@'};
```

Notice that the generic_variable union was initialized just as the first member of a structure would be initialized.

Accessing Union Members

Individual union members can be used in the same way that structure members can be used—by using the member operator (.). However, there is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. The program given below presents an example.

An example of the wrong use of unions.

```
1: /* Example of using more than one union member at a
   time */
2: #include <stdio.h>
3:
4: main()
5: {
6: union shared_tag {
7: char c;
8: int i;
9: long l;
10: float f;
11: double d;
12: } shared;
13:
14: shared.c = '$';
15:
16: printf("\nchar c = %c", shared.c);
17: printf("\nint i = %d", shared.i);
18: printf("\nlong l = %ld", shared.l);
19: printf("\nfloat f = %f", shared.f);
20: printf("\ndouble d = %f", shared.d);
21:
22: shared.d = 123456789.8765;
23:
24: printf("\n\nchar c = %c", shared.c);
25: printf("\nint i = %d", shared.i);
26: printf("\nlong l = %ld", shared.l);
27: printf("\nfloat f = %f", shared.f);
28: printf("\ndouble d = %f\n", shared.d);
29:
30: return 0;
31: }
```

Output

```
char c = $
int i = 4900
long l = 437785380
float f = 0.000000
double d = 0.000000
char c = 7
int i = -30409
long l = 1468107063
float f = 284852666499072.000000
double d = 123456789.876500
```

Analysis

In this listing, you can see that a union named shared is defined and declared in lines 6 through 12. shared contains five members, each of a different type. Lines 14 and 22 initialize individual members of shared. Lines 16 through 20 and 24 through 28 then present the values of each member using printf() statements.

Note that, with the exceptions of char c = $ and double d = 123456789.876500, the output might not be the same on your computer. Because the character variable, c, was initialized in line 14, it is the only value that should be used until a different member is initialized. The results of printing the other union member variables (i, l, f, and d) can be unpredictable (lines 16 through 20). Line 22 puts a value into the double variable, d. Notice that the printing of the variables again is unpredictable for all but d. The value entered into c in line 14 has been lost because it was overwritten when the value of d in line 22 was entered. This is evidence that the members all occupy the same space.

The union Keyword

```
union tag {
union_member(s);
/* additional statements may go here */
}instance;
```

The union keyword is used for declaring unions. A union is a collection of one or more variables (*union_members*) that have been grouped under a single name. In addition, each of these union members occupies the same area of memory.

The keyword union identifies the beginning of a union definition. It's followed by a tag that is the name given to the union. Following the tag are the union members enclosed in braces. An *instance*, the actual declaration of a union, also can be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. The following is a template's format:

```
union tag {
union_member(s);
/* additional statements may go here */
};
```

To use the template, you would use the following format:

union *tag instance*;

To use this format, you must have previously declared a union with the given tag.

Example 1

```
/* Declare a union template called tag */
union tag {
int nbr;
char character;
}
/* Use the union template */
union tag mixed_variable;
```

Example 2

```
/* Declare a union and instance together */
union generic_type_tag {
char c;
int i;
float f;
double d;
} generic;
```

Example 3

```
/* Initialize a union. */
union date_tag {
char full_date[9];
struct part_date_tag {
char month[2];
char break_value1;
char day[2];
char break_value2;
char year[2];
} part_date;
}date = {"01/01/97"};
```

The program given below shows a more practical use of a union. Although this use is simplistic, it's one of the more common

Listing: A practical use of a union.

```
1: /* Example of a typical use of a union */
2:
3: #include <stdio.h>
4:
5: #define CHARACTER 'C'
6: #define INTEGER 'I'
7: #define FLOAT 'F'
8:
9: struct generic_tag{
10: char type;
11: union shared_tag {
12: char c;
```

```
13: int i;
14: float f;
15: } shared;
16: };
17:
18: void print_function( struct generic_tag generic );
19:
20: main()
21: {
22: struct generic_tag var;
23:
24: var.type = CHARACTER;
25: var.shared.c = '$';
26: print_function( var );
27:
28: var.type = FLOAT;
29: var.shared.f = (float) 12345.67890;
30: print_function( var );
31:
32: var.type = 'x';
33: var.shared.i = 111;
34: print_function( var );
35: return 0;
36: }
37: void print_function( struct generic_tag generic )
38: {
39: printf("\n\nThe generic value is...");
40: switch( generic.type )
41: {
42: case CHARACTER: printf("%c", generic.shared.c);
43: break;
44: case INTEGER: printf("%d", generic.shared.i);
45: break;
46: case FLOAT: printf("%f", generic.shared.f);
47: break;
48: default: printf("an unknown type: %c\n",
49: generic.type);
50: break;
51: }
52: }
```

Output

```
The generic value is...$

The generic value is...12345.678711

The generic value is...an unknown type: x
```

Analysis

This program is a very simplistic version of what could be done with a union. This program provides a way of storing multiple data types in a single storage space. The generic_tag structure lets you store either a character, an integer, or a floating-point number within the same area. This area is a union called shared that operates just like the examples in Listing 11.6. Notice that the generic_tag structure also adds an additional field called type. This field is used to store information on the type of variable contained in shared. type helps prevent shared from being used in the wrong way, thus helping to avoid erroneous data such as that presented in Listing 11.6.

A formal look at the program shows that lines 5, 6, and 7 define constants CHARACTER, INTEGER, and FLOAT. These are used later in the program to make the listing more readable. Lines 9 through 16 define a generic_tag structure that will be used later. Line 18 presents a prototype for the print_function(). The structure var is declared in line 22 and is first initialized to hold a character value in lines 24 and 25. A call to print_function() in line 26 lets the value be printed. Lines 28 through 30 and 32 through 34 repeat this process with other values.

The print_function() is the heart of this listing. Although this function is used to print the value from a generic_tag variable, a similar function could have been used to initialize it. print_function() will evaluate the type variable in order to print a statement with the appropriate variable type. This prevents getting erroneous data such as that in Listing 11.6.

DON'T try to initialize more than the first union member.

DO remember which union member is being used. If you fill in a member of one type and then try to use a different type, you can get unpredictable results.

DON'T forget that the size of a union is equal to its largest member.

DO note that unions are an advanced C topic.

Summary

This chapter covered a variety of C programming topics. You learned how to allocate, reallocate, and free memory at runtime, and you saw commands that give you flexibility in allocating storage space for program data. You also saw how and when to use typecasts with variables and pointers. Forgetting about typecasts, or using them improperly, is a common cause of hard-to-find program bugs, so this is a topic worth reviewing! You also learned how to use the memset(), memcpy(), and memmove() functions to manipulate blocks of memory. Finally, you saw the ways in which you can manipulate and use individual bits in your programs.

Q&A

Q What's the advantage of dynamic memory allocation? Why can't I just declare the storage space I need in my source code?

A If you declare all your data storage in your source code, the amount of memory available to your program is fixed. You have to know ahead of time, when you write the program, how much memory will be needed. Dynamic memory allocation lets your program control the amount of memory used to suit the current conditions and user input. The program can use as much memory as it needs, up to the limit of what's available in the computer.

Q Why would I ever need to free memory?

A When you're first learning to use C, your programs aren't very big. As your programs grow, their use of memory also grows. You should try to write your programs to use memory as efficiently as possible. When you're done with memory, you should release it. If you write programs that work in a multitasking environment, other applications might need memory that you aren't using.

Q What happens if I reuse a string without calling realloc()?

A You don't need to call realloc() if the string you're using was allocated enough room. Call realloc() when your current string isn't big enough. Remember, the C compiler lets you do almost anything, even things you shouldn't! You can overwrite one string with a bigger string as long as the new string's length is equal to or smaller than the original string's allocated space. However, if the new string is bigger, you will also overwrite whatever came after the string in memory. This could be nothing, or it could be vital data. If you need a bigger allocated section of memory, call realloc().

Q What's the advantage of the memset(), memcpy(), and memmove() functions? Why can't I just use a loop with an assignment statement to initialize or copy memory?

A You can use a loop with an assignment statement to initialize memory in some cases. In fact, sometimes this is the only way to do it—for example, setting all elements of a type float array to the value 1.23. In other situations, however, the memory will not have been assigned to an array or list, and the mem...() functions are your only choice. There are also times when a loop and assignment statement would work, but the mem...() functions are simpler and faster.

Q When would I use the shift operators and the bitwise logical operators?

A The most common use for these operators is when a program is interacting directly with the computer hardware—a task that often requires specific bit patterns to be generated and interpreted. This topic is beyond the scope of this book. Even if you never need to manipulate hardware directly, you can use the shift operators, in certain circumstances, to divide or multiply integer values by powers of 2.

Q Do I really gain that much by using bit fields?

A Yes, you can gain quite a bit with bit fields. (Pun intended!) Consider a circumstance similar to the example in this chapter in which a file contains information from a survey. People are asked to answer TRUE or FALSE to the questions asked. If you ask 100 questions of 10,000 people and store each answer as a type char as T or F, you will need 10,000 * 100 bytes of storage (because a character is 1 byte). This is 1 million bytes of storage. If you use bit fields instead and allocate one bit for each answer, you will need 10,000 * 100 bits. Because 1 byte holds 8 bits, this amounts to 130,000 bytes of data, which is significantly less than 1 million bytes.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the difference between the malloc() and calloc() memory-allocation functions?
2. What is the most common reason for using a typecast with a numeric variable?
3. What variable type do the following expressions evaluate to? Assume that c is a type char variable, i is a type int variable, l is a type long variable, and f is a type float variable.
   - **a.** (c + i + l)
   - **b.** (i + 32)
   - **c.** (c + 'A')
   - **d.** (i + 32.0)
   - **e.** (100 + 1.0)
4. What is meant by dynamically allocating memory?
5. What is the difference between the memcpy() function and the memmove() function?
6. Imagine that your program uses a structure that must (as one of its members) store the day of the week as a value between 1 and 7. What's the most memory-efficient way to do so?
7. What is the smallest amount of memory in which the current date can be stored? (Hint: month/day/year—think of year as an offset from 1900.)
8. What does 10010010 << 4 evaluate to?
9. What does 10010010 >> 4 evaluate to?
10. Describe the difference between the results of the following two expressions:

    (01010101 ^ 11111111)

    (~01010101)

Exercises

1. Write a malloc() command that allocates memory for 1,000 longs.
2. Write a calloc() command that allocates memory for 1,000 longs.
3. Assume that you have declared an array as follows:

   float data[1000]; Show two ways to initialize all elements of the array to 0. Use a loop and an assignment statement for one method, and the memset() function for the other.
4. **BUG BUSTER:** Is anything wrong with the following code?

```
void func()
{
int number1 = 100, number2 = 3;
float answer;
answer = number1 / number2;
printf("%d/%d = %lf", number1, number2, answer)
}
```

5. **BUG BUSTER:** What, if anything, is wrong with the following code?

```
void *p;
```

```
p = (float*) malloc(sizeof(float));
*p = 1.23;
```

6. **BUG BUSTER:** Is the following structure allowed?

```
struct quiz_answers {
char student_name[15];
unsigned answer1 : 1;
unsigned answer2 : 1;
unsigned answer3 : 1;
unsigned answer4 : 1;
unsigned answer5 : 1;
}
```

Answers are not provided for the following exercises.

7. Write a program that uses each of the bitwise logical operators. The program should apply the bitwise operator to a number and then reapply it to the result. You should observe the output to be sure you understand what's going on.
8. Write a program that displays the binary value of a number. For instance, if the user enters 3, the program should display 00000011. (Hint: You will need to use the bitwise operators.

Notes

LESSON 29 Introductions to Bits & Its Operators

Objectives

Upon completion of this Lesson, you should be able to:

- Know about how to work with bits.
- Know about the shift and bitwise operators.
- Know about the complement operators.

Working with Bits

As you may know, the most basic unit of computer data storage is the bit. There are times when being able to manipulate individual bits in your C program's data is very useful. C has several tools that let you do this.

The C bitwise operators let you manipulate the individual bits of integer variables. Remember, a *bit* is the smallest possible unit of data storage, and it can have only one of two values: 0 or 1. The bitwise operators can be used only with integer types: char, int, and long. Before continuing with this section, you should be familiar with binary notation—the way the computer internally stores integers.

The bitwise operators are most frequently used when your C program interacts directly with your system's hardware. They do have other uses, we'll see what it is.

The Shift Operators

Two shift operators shift the bits in an integer variable by a specified number of positions. The << operator shifts bits to the left, and the >> operator shifts bits to the right. The syntax for these binary operators is

x << n

and

x >> n

Each operator shifts the bits in x by n positions in the specified direction. For a right shift, zeros are placed in the n high-order bits of the variable; for a left shift, zeros are placed in the n low-order bits of the variable. Here are a few examples:

- Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).
- Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).
- Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).
- Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

Under certain circumstances, the shift operators can be used to multiply and divide an integer variable by a power of 2. Left-shifting an integer by *n* places has the same effect as multiplying it by 2*n*, and right-shifting an integer has the same effect as dividing it by 2*n*. The results of a left-shift multiplication are accurate only if there is no overflow—that is, if no bits are "lost" by being shifted out of the high-order positions. A right-shift division is an integer division, in which any fractional part of the result is lost. For example, if you right-shift the value 5 (binary 00000101) by one place, intending to divide by 2, the result is 2 (binary 00000010) instead of the correct 2.5, because the fractional part (the .5) is lost. The program given below demonstrates the shift operators.

Using the shift operators.

```
1: /* Demonstrating the shift operators. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7: unsigned int y, x = 255;
8: int count;
9:
10: printf("Decimal\t\tshift left by\tresult\n");
11:
12: for (count = 1; count < 8; count++)
13: {
14: y = x << count;
15: printf("%d\t\t%d\t\t%d\n", x, count, y);
16:         }
17: printf("\n\nDecimal\t\tshift right by\tresult\n");
18:
19: for (count = 1; count < 8; count++)
20: {
21: y = x >> count;
22: printf("%d\t\t%d\t\t%d\n", x, count, y);
23: }
24: return(0);
25: }
```

Output

Decimal	Shift left by result	Result
255	1	254
255	2	252
255	3	248
255	4	240
255	5	224
255	6	192
255	7	128

Decimal	Shift right by result	Results
255	1	127
255	2	63
255	3	31
255	4	15
255	5	7
255	6	3
255	7	1

The Bitwise Logical Operators

Three bitwise logical operators are used to manipulate individual bits in an integer data type, as shown in Table. These operators have names similar to the TRUE/FALSE logical operators, but their operations differ.

Table. The bitwise logical operators.

Operator	Description
&	AND
|	Inclusive OR
^	Exclusive OR

These are all binary operators, setting bits in the result to 1 or 0 depending on the bits in the operands.

They operate as follows:

- Bitwise AND sets a bit in the result to 1 only if the corresponding bits in both operands are 1; otherwise, the bit is set to 0. The AND operator is used to turn off, or clear, one or more bits in a value.
- Bitwise inclusive OR sets a bit in the result to 0 only if the corresponding bits in both operands are 0; otherwise, the bit is set to 1. The OR operator is used to turn on, or set, one or more bits in a value.
- Bitwise exclusive OR sets a bit in the result to 1 if the corresponding bits in the operands are different (if one is 1 and the other is 0); otherwise, the bit is set to 0.

The following are examples of how these operators work:

Operation	Example
AND	11110000 & 01010101 ——————— 01010000
Inclusive OR	11110000 | 01010101 ——————— 11110101
Exclusive OR	11110000 ^ 01010101 ——————— 10100101

You just read that bitwise AND and bitwise inclusive OR can be used to clear or set, respectively, specified bits in an integer value. Here's what that means. Suppose you have a type char variable, and you want to ensure that the bits in positions 0 and 4 are cleared (that is, equal to 0) and that the other bits stay at their original values. If you AND the variable with a second value that has the binary value 11101110, you'll obtain the desired result.

Here's how this works:

- In each position where the second value has a 1, the result will have the same value, 0 or 1, as was present in that position in the original variable:

  0 & 1 == 0

  1 & 1 == 1
- In each position where the second value has a 0, the result will have a 0 regardless of the value that was present in that position in the original variable:

  0 & 0 == 0

  1 & 0 == 0
- Setting bits with OR works in a similar way. In each position where the second value has a 1, the result will have a 1, and in each position where the second value has a 0, the result will be unchanged:

  0 | 1 == 1

  1 | 1 == 1

  0 | 0 == 0

  1 | 0 == 1

The Complement Operator

The final bitwise operator is the complement operator, ~. This is a unary operator. Its action is to reverse every bit in its operand, changing all 0s to 1s, and vice versa. For example, ~254 (binary 11111110) evaluates to 1 (binary 00000001).

All the examples in this section have used type char variables containing 8 bits. For larger variables, such as type int and type long, things work exactly the same.

LESSON 30 Dynamic Memory Allocation

Objectives

Upon completion of this Lesson, you should be able to:

- Know about Dynamic Memory allocation.
- Know about the malloc function.
- Know about the calloc operators.

Dynamic Memory Allocation

Allocating large data objects at compile time is seldom practical—especially if the data objects are used infrequently and for a short time. Instead, you usually allocate these data objects at runtime. To make more memory available to the programmer, ANSI C offers a number of memory allocation functions, including malloc(), realloc(), calloc(), and free(). Many compiler suppliers complement these functions with similar functions that optimize the allocation of memory based on the specifics of your computer's architecture. In this lesson, you look at these four functions and Microsoft's enhanced versions of them.

Using the malloc() Function

The memory allocation functions in Table include both the ANSI C standard malloc() functions and Microsoft's extensions.

Table Microsoft C malloc( ) functions.

Function	Description
`void * malloc( size_t size );`	The ANSI C standard memory allocation function.
`void __based( void ) *_bmalloc ( __segment seg, size_t size );`	Does based memory allocation. The memory is allocated from the segment you specify.
`void __far *_fmalloc( size_tsize );`	Allocates a block of memory outside the default data segment, returning a `far` pointer. This function is called by `malloc()` when the `large` or `compact` memory model is specified.
`void __near *_nmalloc ( size_t size );`	Allocates a block of memory inside the default data segment, returning a `near` pointer. This function is called by `malloc()` when the `small` or `medium` memory model is specified.

The malloc() library function allocates a block of memory up to the size allowed by size_t. To use malloc(), you must follow a few simple rules:

- The malloc() function returns a pointer to the allocated memory or NULL if the memory could not be allocated. You should always check the returned pointer for NULL.
- The pointer returned by malloc() should be saved in a static variable, unless you are sure that the memory block will be freed before the pointer variable is discarded at the end of the block or the function.
- You should always free a block of memory that has been allocated by malloc() when you are finished with it. If you rely on the operating system to free the block when your program ends, there may be insufficient memory to satisfy additional requests for memory allocation during the rest of the program's run.
- Avoid allocating small blocks (that is, less than 25 or 50 bytes) of memory. There is always some overhead when malloc() allocates memory—16 or more bytes are allocated in addition to the requested memory.

The malloc() function requires only one parameter: the size of the block of memory to allocate. As mentioned, the length of this parameter is size_t, which on many systems is a short int (16 bits).

You could assume that you cannot allocate a block of memory larger than the ANSI C maximum of 32,767 bytes. Another method is to check the defined identifier (usually in malloc.h) for the maximum for the particular system. With Microsoft C compilers, for example, the maximum is approximately 65,500 bytes. If you assume the worst case (the ANSI C value), however, your program has a better chance of working if the limit changes.

The constraint on the size of a data object may seem unreasonable, but you will rarely reach the 32K limit imposed by ANSI C. If you have large data objects, it is always possible (and desirable) to break them into smaller, more manageable pieces.

If you are determined to define a data object larger than the allowed size (something I do not recommend) and are using a Microsoft C compiler, you can use the halloc() function. This function allocates an array that can be any size (up to the amount of available free memory). You must define the array element size as a power of two, which is not an issue if the array is type char, int, or long. If the array is a structure, type union, or a floating-point long double, this constraint may need to be addressed with padding. If you use the halloc() function, your code will not be portable, but you could probably create a workaround if necessary.

When you use the malloc() function, remember that the block of allocated memory is not initialized. If you want initialized memory, use memset() after the memory is allocated or use calloc() (discussed in the next section). I recommend that you

always initialize any memory allocated with the malloc() function.

Program, MALLOC2.C, allocates blocks of memory. There is no way to determine the size of the largest available block, so the program begins with the largest size (32,767). If malloc() fails, the program reduces this size by 50 percent; this continues until the requested size is less than 2 bytes. The program stops when there is no more memory, or a total of 2M has been allocated.

MALLOC2.C.

```
/* MALLOC2,
* This program allocates memory.
*/
#include <io.h> // I/O functions
#include <stdio.h> // Make includes first in program
#include <string.h> // For string functions
#include <malloc.h> // For memory allocation functions
int main(void); // Define main() and the fact that this
// program doesn't use any passed parameters
int main()
{
int i = 0;
int j = 0;
int *nPointer[100] = {NULL};
int nSize = 32767;
long lTotalBytes = 0;
while(nSize > 0 && // Make nSize valid
nSize <= 32767 &&
lTotalBytes < 2000000) // Not more than 2M will be allocated
{
nPointer[i] = (int *)malloc(nSize);
if (nPointer[i] != NULL)
{
++i;
lTotalBytes += nSize;
printf("Allocated %5u bytes, total
%10ld\n",
nSize,
lTotalBytes);
}
else
{
printf("Couldn't allocate %5u
bytes, total %10ld\n",
nSize,
lTotalBytes);
nSize /= 2;
}
}
for (j = 0; j < i; j++)
{
free(nPointer[j]);
nPointer[j] = NULL;
}
return (0);
}
```

The program is system dependent. If you are using a PC under DOS in real mode, for example, about 400,000 bytes of memory might be allocated. Under a protectedmode environment such as OS/2 or Windows, you can allocate much more memory.

For example, on a system running in protected mode with 10M of free memory, 2M of memory might be allocated, as follows:

Allocated 32767 bytes, total 32767

Allocated 32767 bytes, total 65534

Allocated 32767 bytes, total 98301

and so on...

Allocated 32767 bytes, total 1966020

Allocated 32767 bytes, total 1998787

Allocated 32767 bytes, total 2031554

If you are not sure of the environment in which your application will be running, assume the worst case—less than 32K of free memory. Notice that a loop at the end of the program frees the memory that malloc() has allocated. This loop is performing housekeeping—something that every well-written program should do.

Using the calloc() Function

Because malloc() does not initialize memory and calloc() does, programmers often prefer calloc(). When using Microsoft's C compilers, the array memory allocation functions in Table are used with calloc().

Table : Microsoft C calloc( ) Functions.

Function	Description
`void *calloc( size_t num, size_t size );`	The ANSI C standard array memory allocation function.
`void __based( void ) *_bcalloc( __segment seg, size_t num, size_t size );`	Does based memory allocation. You provide the segment that the data will be allocated from.
`void __far *_fcalloc ( size_t num, size_t size );`	Allocates a block of memory outside the default data segment, returning a far pointer. This function is called by `calloc()` when the `large` or `compact` memory model is specified.
`void __near *_ncalloc ( size_t num, size_t size );`	Allocates a block of memory inside the default data segment, returning a near pointer. This function is called by `calloc()` when the `small` or `medium` memory model is specified.

The calloc() library function allocates memory much like the malloc() function, with two main differences. With the calloc() function, you specify two parameters, not one: the number of elements and the size of each element. The product of these parameters determines the size of the memory block to allocate, and must fit in type size_t, which on many systems is a short int (16 bits). If you specify an element size of 1, the calloc() num parameter functions similarly to the malloc() size parameter.

The second difference is that the calloc() function initializes the memory it allocates to zero. The value used to initialize the memory is an absolute zero, which usually—but not always—evaluates to a floating-point zero or a NULL pointer value. This is fine if the memory will be used for string storage or integers. If the memory will be used for floating-point values, you should explicitly initialize the block of memory after calloc() returns. I recommend that you always initialize memory allocated with calloc if you do not know the format of the data that you will be storing in it.

To use calloc(), you follow the same rules for using malloc(). These rules are outlined in the first section, "Using the malloc() Function."

The program CALLOC1.C, allocates blocks of memory. The size of the largest available block cannot be determined, so the program begins with the largest size possible (using the size of int) and tries to allocate an array of 32,767 members. If calloc() fails, the program reduces the size by 50 percent; this continues until the requested size is less than 2 bytes. The program allocates buffers, each containing 32,767 2-byte integers. When an allocation request fails, the program decreases the size of the array until more memory can be allocated. It stops when there is no more memory or 2M have been allocated. The only major difference between MALLOC2.C and CALLOC1.C is the call to the memory allocation function.

CALLOC1.C.

```
/* CALLOC1, written 1992 by Peter D. Hipson
* This program allocates arrays of memory.
*/
#include <stdio.h> // Make includes first part of file
#include <string.h> // For string functions
#include <malloc.h> // For memory allocation functions
int main(void); // Define main() and establish that this
// program does not use any passed parameters
int main()
{
int i = 0;
int j = 0;
int *nPointer[100] = {NULL};
int nSize = 32767;
long lTotalBytes = 0;
while(nSize > 0 && // Make nSize valid
nSize <= 32767 &&
lTotalBytes < 2000000) // No more than 2M will be allocated
{
nPointer[i] = (int *)calloc(nSize, sizeof(int));
if (nPointer[i] != NULL)
{
++i;
lTotalBytes += (nSize * sizeof(int));
printf("Allocated %5u short int, total %10ld\n",
nSize,
lTotalBytes);
}
else
{
printf("Couldn't allocate %5u short int, total %10ld\n",
nSize,
lTotalBytes);
nSize /= 2;
}
}
for (j = 0; j < i; j++)
{
free(nPointer[j]);
nPointer[j] = NULL;
}
return (0);
}
```

When CALLOC1 was run, it could not allocate an integer array of 32,767 members, as the following output shows:

```
Couldn't Allocate 32767 bytes, total 0
Allocated 16383 bytes, total 32766
Allocated 16383 bytes, total 65532
Allocated 16383 bytes, total 98298
and so on...
Allocated 16383 bytes, total 1965960
Allocated 16383 bytes, total 1998726
Allocated 16383 bytes, total 2031492
```

The reason for this is not the ANSI C limit of 32,767 bytes in a data object—my C compiler does not enforce this limit. The limit in my compiler is that a data object created by calloc() or malloc() cannot be larger than 65,510 bytes. The array of integers consisted of 32,767 members (each 2 bytes long), for a total of 65,534 bytes, which is too large.

CALLOC1 then attempted to allocate the next size, 16,383, and was successful.

Using the free() Function

The free() functions in Table can be used with a Microsoft C compiler.

Table. Microsoft C free( ) Functions.

Function	Description
`void free( void *memblock );`	The ANSI C standard array memory deallocation function.
`void _bfree( __segment seg, void __based( void ) *memblock );`	Based memory deallocation.
`void _ffree( void __far *memblock );`	Frees a block of memory outside the default data segment.
`void _nfree( void __near *memblock );`	Frees a block of memory inside the default data segment.

The free() memory allocation function was shown in the previous program. Its function is to return to the operating system memory that you have allocated. (You could think of the memory as borrowed.) Memory is usually a limited resource—even when you are running a system with virtual memory—so you must give memory back when you are finished with it.

The free() function is almost foolproof. Errors could occur, however, when you try to free memory that

- Was not allocated with one of the memory allocation functions;
- Has been released through a prior call to free() or a call to realloc();
- Is currently in use by another thread in a multithreaded operating system;
- Is not yours to free.

When free() is called, be sure you are passing a valid pointer to it . To make sure that the pointer is valid, check that it contains NULL or points to a properly allocated block of memory. Note that free() considers a NULL pointer to be always valid, and treats a call with a NULL pointer as a no-operation, which means free() simply returns without freeing any memory.

Look at the following variable declarations in the CALLOC1 program:

```
int j = 0;
int *nPointer[100] = {NULL};
int nSize = 32767;
```

The pointer array is initialized to NULL, which is a safe value for initialization because it will not cause an error if it is passed to free(). In addition, because the loop that allocates the memory uses an index into the array of memory pointers (nPointer[]), only valid pointers to allocated blocks of memory will be stored in nPointer[]. In the final cleanup, the memory pointed to by nPointer[] is freed. The following loop resets each pointer to NULL after the block of memory is freed:

```
for (j = 0; j < i; j++)
{
free(nPointer[j]);
nPointer[j] = NULL;
}
```

If the program tries to free one of the already freed nPointer[]s later (because of a programming error), the NULL pointer prevents an error that would be caused by trying to free a block of memory twice.

Notes

LESSON 31 Functions of Dynamic Memory Allocation

Objectives

Upon completion of this Lesson, you should be able to:

- Know about how to use the realloc function.
- Know how to allocate arrays.
- Know about memory vs local memory

Using the realloc() Function

When using Microsoft's C compilers, the array memory allocation functions in Table are used with realloc().

Table. Microsoft C realloc( ) functions.

Function	Description
`void *realloc( void *memblock,size_t size );`	The ANSI C standard array memory reallocation function.
`void __based( void ) *_brealloc( __segment seg, void __based( void ) *memblock, size_t size );`	Does based memory reallocation. You must provide the segment that the data will be allocated from.
`void __far *_frealloc ( void __far *memblock, size_t size );`	Reallocates a block of memory outside the default data segment, returning a far pointer. This function is called by realloc() when the large or compact memory model is specified.
`void __near *_nrealloc ( void __near *memblock, size_t size );`	Reallocates a block of memory inside the default data segment, returning a near pointer. This function is called by realloc() when the small or medium memory model is specified.

Assume that you have a program that reads customer records from the keyboard and stores each in a structure. When the user is finished entering the names, the program saves them to disk. You want to be sure that there is enough memory (within reason) to hold the entered names, but you do not want to allocate more memory than necessary.

You could call calloc() and allocate all available free memory for the structures. This might work, but it wastes a lot of memory. Another method is to call calloc() and allocate a small block, but the program would have to pause to save the information, something that might irritate the user. Or you could call calloc(), allocate a small block, call calloc() again when the block was filled and get a bigger block of memory, copy the small block of memory to the larger one, then free the small block. As you can see, that would require a lot of work. The best solution is to call calloc() to allocate the initial array, then call realloc() to make the block larger. The realloc() function copies the contents of the original block of memory to the new block, then frees the original block, so your work is minimized.

The program given below is the CDB program.Unlike CREATEDB, CDB writes the records entered by the user to the file only after the user has finished entering the names.

CDB.C.

```
/* CDB
* This program uses calloc() and realloc(). It has
* better error checking than the CREATEDB program,
* which was presented in Chapter 7.
*/
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#define INCREMENT_AMOUNT 2
#define CUSTOMER_RECORD 1
#define SUPPLIER_RECORD 2
/* Define our structure for the customer database. */
typedef struct _CUSTNAME
{
int nRecordType; // 1 == Customer record
char szName[61]; // 60 chars for name, 1 for null at end
char szAddr1[61]; // 60 chars for address, 1 for null at end
char szAddr2[61]; // 60 chars for address, 1 for null at end
char szCity[26]; // 25 chars for city, 1 for null at end
char szState[3]; // 2-char state abbreviation, plus null
long lZip; // Use integer, print as %5.5ld for leading 0
int nRecordNumber; // Which record number?
double dSalesTotal; // How much customer has purchased
} CUSTNAME;
typedef CUSTNAME near *NPCUSTNAME;
typedef CUSTNAME *PCUSTNAME;
void main()
{
FILE *DataFile;
PCUSTNAME Customer = NULL;
PCUSTNAME TempCustomer = NULL;
char szFileName[25];
char szBuffer[257];
int i;
```

```
int nNumberRecords = 0;
int nRecord = 0;
int nResult = 0;
double dSales = 0.0; // Forces loading of floating-point
support
Customer = (PCUSTNAME)calloc(sizeof(CUSTNAME),
INCREMENT_AMOUNT);
nNumberRecords += INCREMENT_AMOUNT;
printf("Please enter customer database name: ");
gets(szFileName);
DataFile = fopen(szFileName, "wb");
if (DataFile == NULL)
{
printf("ERROR: File '%s' couldn't be opened.\n",
szFileName);
exit(4);
}
printf("Demo of calloc() and realloc(). sizeof(CUSTNAME) =
%d\n",
sizeof(CUSTNAME));
nRecord = 0;
Customer[nRecord].szName[0] = 'A'; // To get past while()
first time
while (strlen(Customer[nRecord].szName) > 0)
{
memset(&Customer[nRecord], 0, sizeof(CUSTNAME));
printf("Enter name %d: ", nRecord + 1);
gets(szBuffer);
szBuffer[sizeof(Customer[nRecord].szName) - 1] = '\0';
strcpy(Customer[nRecord].szName, szBuffer);
if (strlen(Customer[nRecord].szName) > 0)
{
Customer[nRecord].nRecordNumber = i;
do
{
printf("Enter 1 for customer, 2 for supplier ");
gets(szBuffer);
                sscanf(szBuffer,"%d",&Customer[nRecord].nRecordType);
}
while (Customer[nRecord].nRecordType !=
CUSTOMER_RECORD &&
Customer[nRecord].nRecordType != SUPPLIER_RECORD);
printf("Enter address line 1: ");
gets(szBuffer);
szBuffer[sizeof(Customer[nRecord].szAddr1) - 1] = '\0';
strcpy(Customer[nRecord].szAddr1, szBuffer);
printf("Enter address line 2: ");
gets(szBuffer);
szBuffer[sizeof(Customer[nRecord].szAddr2) - 1] = '\0';
strcpy(Customer[nRecord].szAddr2, szBuffer);
printf("Enter City: ");
gets(szBuffer);
szBuffer[sizeof(Customer[nRecord].szCity) - 1] = '\0';
strcpy(Customer[nRecord].szCity, szBuffer);
printf("Enter state postal abbreviation: ");
gets(szBuffer);
szBuffer[sizeof(Customer[nRecord].szState) - 1] = '\0';
strcpy(Customer[nRecord].szState, szBuffer);
printf("Enter ZIP code: ");
gets(szBuffer);
sscanf(szBuffer, "%ld", &Customer[nRecord].lZip);
printf("Enter total sales: ");
gets(szBuffer);
sscanf(szBuffer, "%f", &Customer[nRecord].dSalesTotal);
++nRecord;
if (nRecord == nNumberRecords)
{
TempCustomer = (PCUSTNAME)realloc(Customer,
sizeof(CUSTNAME) * (nNumberRecords +
INCREMENT_AMOUNT));
if (TempCustomer)
{
nNumberRecords += INCREMENT_AMOUNT;
printf("realloc() added records, now total is %d\n",
nNumberRecords);
Customer = TempCustomer;
Customer[nRecord].szName[0] = 'A'; // To get past
while()
}
else
{
printf("ERROR: Couldn't realloc the buffers\n\n\g");
—nRecord;
Customer[nRecord].szName[0] = '\0';
}
}
else
{
Customer[nRecord].szName[0] = 'A'; // To get past while()
}
}
}
for (i = 0; i < nRecord; i++)
{
printf("Name '%10s' City '%10s' State '%2s' ZIP '%5.5ld'\n",
```

```
Customer[i].szName,
Customer[i].szCity,
Customer[i].szState,
Customer[i].lZip);
}
nResult = fwrite((char *)Customer,
sizeof(CUSTNAME),
nRecord,
DataFile);
if (nResult != nRecord)
{
printf("ERROR: File '%s', write error, record
%d.\n",szFileName,i);
fclose(DataFile);
exit(4);
}
fclose(DataFile);
}
```

By expanding the buffers used for storing data, the data can be saved in memory and written to the disk at one time. In addition, summary information such as totals could be displayed, the user could edit the entered information, and the information could be processed if necessary. The one hitch is that all the user's data that is in RAM and not written to the disk will be lost if the computer fails. With CREATEDB,at most one record would be lost.

When you write a program in which the user will be entering substantial amounts of data from the keyboard, you should plan for events that might cause the loss of information just entered. One solution to retaining this information is to write to the file after the user inputs a record. Summary information can be presented, records can be edited, and so on, and the records the user entered can be rewritten by the program to a master file later as necessary.

The realloc() function enables you to have some control over the size of your dynamic data objects. Sometimes, however, the data objects will become too large for available memory. In CDB, for example, each data object is 228 bytes long. If 40,000 bytes of free memory were available, the user could enter about 176 records before using up free memory. Your program must be able to handle the problem of insufficient memory in a way that does not inconvenience the user or lose data.

Allocating Arrays

Allocating an array is an easy process when you use calloc(). Its parameters are the size for each element of the array and a count of the number of array elements. To dynamically allocate an array at runtime, you simply make a call.

The program given below prompts the user for a number of integers, in the range 10 to 30,000. It then creates a list of integers, sorts them, and prints the result.

SORTALOC.C.

```
/* SORTALOC, written 1992 by Peter D. Hipson
* This program prompts for the number of integers to sort,
* allocates the array, fills the array with random numbers,
* sorts the array, then prints it, using 10 columns.
*/
#include <search.h>
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <time.h>
int compare(const void *, const void *);
int main()
{
int i;
int *nArray = NULL;
int nArraySize = 0;
while(nArraySize < 10 || nArraySize > 30000)
{
printf("Enter the number of random integers to sort (10 to \
30,000): ");
scanf("%d", &nArraySize);
if(nArraySize < 10 || nArraySize > 30000)
{
printf("Error: must be between 10 and 30,000!\n");
}
nArray = (int *)calloc(sizeof(int), nArraySize);
if (nArray == NULL)
{
printf("Error: couldn't allocate that much memory!\n");
nArraySize = 0;
}
}
srand((unsigned)time(NULL));
for (i = 0; i < nArraySize; i++)
{
nArray[i] = rand();
}
qsort(nArray, nArraySize, sizeof(int), compare);
for (i = 0; i < nArraySize; i += 10)
{
printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d
%5d\n",
nArray[i],
nArray[i + 1],
nArray[i + 2],
nArray[i + 3],
```

```
nArray[i + 4],
nArray[i + 5],
nArray[i + 6],
nArray[i + 6],
nArray[i + 7],
nArray[i + 8],
nArray[i + 9]);
}
free(nArray);
return(0);
}
int compare(const void * a,const void * b)
{
return (*(int *)a - *(int *)b);
}
```

SORTALOC illustrates several important points about using the memory allocation functions. First, the array is simply declared as an integer pointer called nArray. This pointer is initialized with NULL to prevent an error when the free() function frees the pointer. Although always initializing variables may seem excessive, using an uninitialized variable is a common programming error. After calloc() allocates the array, it can be accessed in the same way as any other array.

For example, standard array indexing can be used, as shown in the following:

```
for (i = 0; i < nArraySize; i++)
{
nArray[i] = rand();
}
```

The loop assigns a random number to each element (indexed by i).

After the array is filled, it is passed to the qsort() function like any other array. The qsort() function can sort almost any type of data. You just give qsort() the size of the array's elements, the number of elements, and the compare function. (Note:The compare function in the program given is valid for integers but not floating-point values.This is because the compare must return an integer, and a floating-point value may differ by less than the truncation value of an integer.)

Finally, the array is printed in ten columns. There is nothing tricky about this portion of the code—one print statement prints ten elements, then the index is incremented by ten.

Global Memory versus Local Memory

The discussion of local memory and global memory is applicable to computers with Intel 80x86 CPUs. These CPUs use segmented architecture, in which a data object can be addressed with a full address (consisting of both a segment and an offset from the segment) or as an offset (where the segment used is the default data segment).Not all operating systems and compilers offer access to both local memory (found in the default data segment) and global memory (located outside the default data segment, usually in its own segment). A compiler that offers memory models, such as small, medium, large, and compact, is generally found on a PC-type of computer. The discussion in this section pertains to compilers used on an 80x86 CPU.

For most compilers, the memory model determines the area from which memory

will be allocated. If your program uses the large or compact memory model, the default

memory pool is global. If your program is a small or medium model program, the default

memory pool is local. You can always override the compiler's default memory allocation area.

When running in real mode, Intel 80x86 CPUs can access a maximum of 64K in each segment. This limitation, and the way the default data segment is allocated (it is often used for the stack, initialized data variables, constants, literals, and the heap, which is where memory is allocated from when using local memory), affects how much data a program can have.

Global memory has its own segment, and thus can have up to 64K in a single data object (or more than 64K by using several contiguous segments). To use global memory, however, your program must use far (4-byte) pointers rather than near (2-byte) pointers, and this can slow program execution. If you need to determine the effect this has on performance, you could create one version of your program with small data blocks and near pointers, and the other with large data blocks and far pointers, then run simple benchmarks.

Summary

In these lessons, you learned about memory allocation, how to change the size of an allocated block of memory, and how to free memory after it is no longer needed.

- The malloc() function is the ANSI standard method for allocating memory. It accepts a single parameter that specifies how much memory should be allocated.
- The calloc() function allocates memory based on the size of the object and the number of objects. It is typically used to allocate an array.
- When memory allocated with one of the memory allocation functions is no longer needed, the free() function returns the memory to the operating system.
- The realloc() function changes the size of a block of memory allocated with one of the memory allocation functions. The object's size can be increased or decreased.
- When programming on the PC (and other systems), you can often choose the size of the pointer that accesses the allocated memory. The pointer size affects the size of the executable program and the performance of the program.

LESSON 32 Verification and Validation

Objectives

Upon completion of this Lesson, you should be able to:

- Know about verification and validation.
- Know how to use different types of testing using different types of test data.
- Know what is the testing process.

Verification and Validation

Verification and validation (V & V) is the generic name given to checking processes, which ensure that software conforms to its specification and meets the needs of the software customer. The system should be verified and validated at each stage of the software process using documents produced during the previous stage. Verification and validation therefore starts with requirements reviews and continues through design and code reviews to product testing.

Verification and validation are sometimes confused. They are, in fact, different activities. The difference between them is succinctly summarized by Boehm (1979).

'Validation: Are we building the right product ?'

'Verification: Are we building the product right ?'

Verification involves checking that the program conforms to its specification, Validation involves checking that the program as implemented meets the expectations of the software customer. Requirements validation techniques, such as prototyping, help in this respect. However, flaws and deficiencies in the requirements can sometimes only be discovered when the system implementation is complete.

To satisfy the objectives of the V & V process, both static and dynamic techniques of system checking and analysis should be used. Static techniques arc concerned with the analysis and checking of system representations such as the requirements document, design diagrams and the program source code. They may be applied at all stages of the process through structured reviews. Dynamic techniques or tests involve exercising an implementation.

Figure 1 shows the place of static and dynamic techniques in the software process. Static techniques can be used at all stages of the software process. Dynamic techniques, however, can only be used when a prototype or an executable program is available.

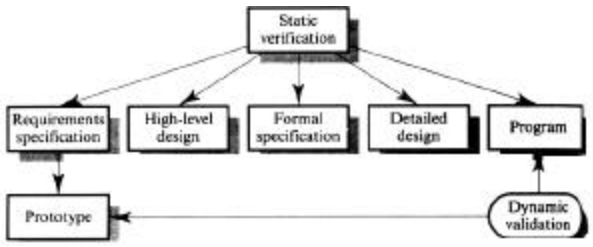


Figure 1: Static and dynamic Verification and Validation

Static techniques include program inspections, analysis and formal verifica-tion. Some purists have suggested that these techniques should completely replace dynamic techniques in the verification and validation process and that testing is unnecessary. This is nonsense. Static techniques can only check the correspondence between a program and its specification (verification); they cannot demonstrate that the software is operationally useful.

Although static verification techniques are becoming more widely used, program testing is still the predominant verification and validation technique. Testing involves exercising the program using data like the real data processed by the program. The existence of program defects or inadequacies is inferred from unexpected system outputs. Testing may be carried out during the implementation phase to verify that the software behaves, as intended by its designer and after the implementation is complete. This later testing phase checks conformance with requirements and assesses the reliability of the system.

Different kinds of testing use different types of test data:

- Statistical testing may be used to test the program's performance and reliability. Tests are designed to reflect the frequency of actual user inputs. After running the tests. an estimate of the operational reliability of the system can be made. Program performance may be judged by measuring the execution of the statistical tests.
- Defect testing is intended to find areas where the program does not conform to its specification. Tests are designed to reveal the presence of defects in the system.

When defects have been found in a program, these must be discovered and removed. This is called *debugging.* Defect testing and debugging are sometimes considered to be parts of the same process. In fact, they are quite different. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

Figure 2 illustrates a possible debugging process. Defects in the code must be located and the program modified to meet its requirements. Testing must then be related to ensure that the change has been made correctly.

The debugger must generate hypotheses about the observable behavior of the program then test these hypotheses in the hope of finding the fault which caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debug-ging tools which show the intermediate values of program variables and a trace of the statements executed may be used to help the debugging process.

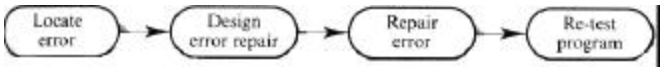


Figure 2: The debugging process

It is impossible to present a set of instructions for program debugging. The skilled debugger looks for patterns in the test output where the defect is exhibited and uses knowledge of the defect, the pattern and the programming process to locate the defect. Process knowledge is important. Debuggers know of common programmer errors (such as failing to increment a counter) and match these against the observed patterns.

After a defect in the program has been discovered, it must be corrected and the system should then be re-tested. This form of testing is called regression testing. Regression testing is used to check that the changes made to a program have not introduced new faults into the system.

In principle, all tests should be repeated after every defect repair; in practice this is too expensive. As part of the test plan, dependencies between parts of the system and the tests associated with each part should be identified. When a change is made, it may only be necessary to run a subset of the entire test data set to check the modified component and its dependants.

The Testing Process

Except for small programs, systems should not be tested as a single, monolithic unit large systems are built out of sub-systems, which are built out of modules, which are composed of procedures and functions. The testing process should therefore processed in stages where testing is carried out incrementally in conjunction with system implementation.

The most widely used testing process consists of five stages as shown in Figure 3. In general, the sequence of testing activities is component testing, integration testing then user testing. However, as defects are discovered at anyone stage, they require program modifications to correct them and this may require other stages

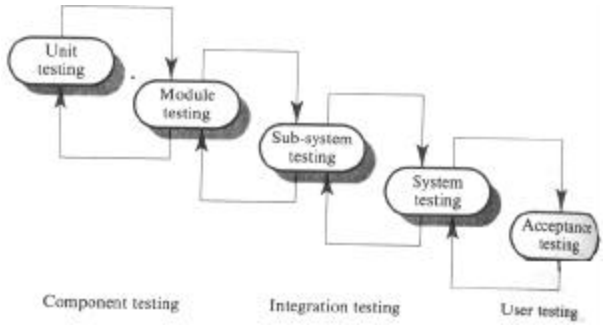


Figure 3: The testing process.

in the testing process to be repeated. Errors in program components, say, may come to light at a later stage of the testing process. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

In Figure 3, the arrows from the top of the boxes indicate the normal sequence of testing. The arrows returning to the previous box indicate that previous testing stages may have to be repeated. The stages in the testing process are:

- Unit testing Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.
- Module testing A module is a collection of dependent components such as an object class, an abstract data type or some looser collection of procedures and functions. A module encapsulates related components so can be tested without other system modules.
- Sub-system testing this phase involves testing collections of modules which have been integrated into sub-systems. Sub-systems may be independently designed and implemented. The most common problems which arise in large software systems are sub-system interface mismatches. The sub-system test process should therefore concentrate on the detection of interface errors by rigorously exercising these interfaces.
- System testing The sub-systems are integrated to make up the entire system. The testing process is concerned with finding errors, which result from un-anticipated interactions between sub-systems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.
- Acceptance testing This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system procurer rather than simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercises the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's -facilities do not really meet the user's needs or the system performance is unacceptable.

Acceptance testing is sometimes called *alpha testing.* Bespoke systems are developed for a single client. The alpha testing process continues until the s)'stem developer and the client agree that the delivered system is an acceptable implemen-tation of the system requirements.

When a system is to be marketed as a software product, a testing process called *beta testing* is often used. Beta testing involves delivering a system, to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors, which may not have been anticipated by the system builders. After this feedback, the system is modified and either released for further beta testing or for general sale.

Object oriented system testing

The model of system testing shown in Figure3 is based on the notion of incremental system integration where simple components are integrated to form modules, These modules are integrated into sub-systems and finally the sub-systems are inte-grated into a complete system. In essence, we should finish testing at one integration level before moving on to the next level.

When object-oriented systems are developed, the levels of integration are less distinct. Clearly, operations and data are integrated to form objects and object classes. Testing these object classes corresponds 10 units testing. There is no direct equivalent to module testing in object-oriented systems. However, Murphy *ef al*. (1994) suggest that groups of classes

which act in combination to provide a set of services should be tested together, They can this *cluster testing.*

At higher levels of integration, namely sub-system and system levels, thread testing may be used later in this chapter. Thread testing is based on testing the system's response to a particular input or set of input events. Object-oriented systems are often event-driven so this is a particularly appropriate form of testing to use.

A related approach to testing groups of interacting objects is proposed by Jorgensen and Erickson (1994). They suggest that an intermediate level of integration testing can be based on identifying 'method-message' (MM) paths. These are traces through a sequence of object interactions, which stop when an object operation does not call on the services of any other object. They also identify a related construct, which they call an 'Atomic System Function' (ASF). An ASF consists of some input event followed by a sequence of MM-paths, which is terminated, by an output event. This is similar to a thread in a real"time system, discussed later in this chapter.

Test planning

System testing is expensive. For some large systems. such as real-time systems with complex non-functional constraints, half the system development budget may be spent on testing. Careful planning is needed to get the most out Of testing and 10 control testing costs.

Test planning is concerned with setting out standards for the testing process rather than describing product tests. Test plans are not just management documents. They are also intended for software engineers involved in designing and carrying out system tests. They allow technical staff to get an overall picture of the system tem and to place their own work in this context. Test plans also provide information to staff who are responsible for ensuring that appropriate hardware and software resources are available to the testing team.

The major components of a test plan are shown in Figure 4. This plan should include significant amounts of contingency so that slippages in design and implementation can be accommodated and staff allocated to testing can be deployed in other activities. A good description of test plans and their relation to more general quality plans is given in Frewin and Hatton (1986).

Figure 4:Test plan contents.

Like other plans, the test plan is not a static document. It should be revised regularly as

testing is an activity which is dependent on implementation being complete. If part of a system is incomplete, the system testing process cannot begin.

The preparation of the test plan should begin when the system requirements are formulated and it should be developed in detail as the software is designed.

figure 5 shows the relationships between test plans and software process activities.

The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process which are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

Hardware and software requirements
This section should set out software tools required and estimated hardware utilization.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Figure 4: component of the test plan

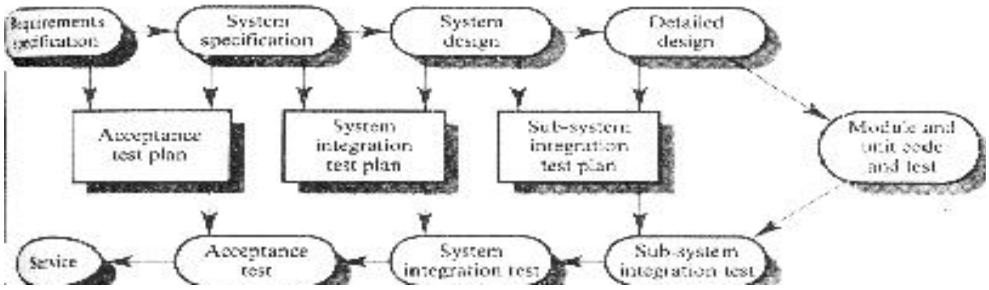


Figure 5 : Testing phases in the software process.

Figure.5 is a horizontal version of what is sometimes called the V-model of the software process. This V-model is an extension of the simple waterfall model where each process phase concerned with development has an associated verification and validation phase. The individual test plans are the links between these activities.

Unit testing and module testing may be the responsibility of the programmer developing the component. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach as the programmer knows the component best and is most able to generate test data. Unit testing is part of the implementation process and it is expected that a component conforming to its specification will be delivered as part of that process, as it is a natural human trait for individuals to feel an affinity with objects they have constructed, programmers responsible for system implementation may feel that testing threatens their creations. Psychologically, programmers do not usually want to 'destroy' their work. Consciously or subconsciously, tests may be selected which will not demonstrate the presence of system defects.

If unit testing is left to the component developer, it should be subject to some monitoring procedure to ensure that the components have been properly tested. Some of the components should be re-tested by an independent tester using a different set of test cases. If independent testing and programmer testing come to the same conclusions, it may be assumed that the programmer's testing methods are adequate.

Later stages of testing involve integrating work from a number of programmers and must be planned **in** advance. They should be undertaken by an independent team of testers. Module and sub-system testing should be planned as the design of the sub-system is formulated. Integration tests should be developed in conjunction with the system design. Acceptance tests should be designed with the program specification. They may be written into the Contract for the system development.

Notes

LESSON 33 Testing strategies

Objectives

Upon completion of this Lesson, you should be able to:

- Know about the various test strategies in detail.

Testing strategies

A testing strategy is a general approach to the testing process rather than a method of devising particular system or component tests. Different testing strategies may be adopted depending on the type of system to be tested and the development Process used.

The testing strategies, which I discuss in this section, are;

1. Top-down testing where testing starts with the most abstract component and works downwards .
2. Bottom-up testing where testing starts with the fundamental components and works upwards.
3. Thread resting which is used for systems with multiple processes where the processing of a transaction threads its way through these processes.
4. Stress testing which relies on stressing the system by going beyond its Specified limits and hence testing how well the system can cope with over-load situations.
5. Back-to-back testing, which is used when versions of a system are available. The systems are tested together and their outputs are compared. Large systems are usually tested using a mixture of these testing strategies rather than any single approach. Different strategies may be needed for different pans of the system and at different stages in the testing process.

Whatever testing strategy is adopted, it is always sensible to adopt an incremental approach 10 sub-system and system testing (Figure 6). Rather than integrate all components into a system and then start testing, the system should be tested incrementally. Each increment should be tested before the next increment is added to the system.

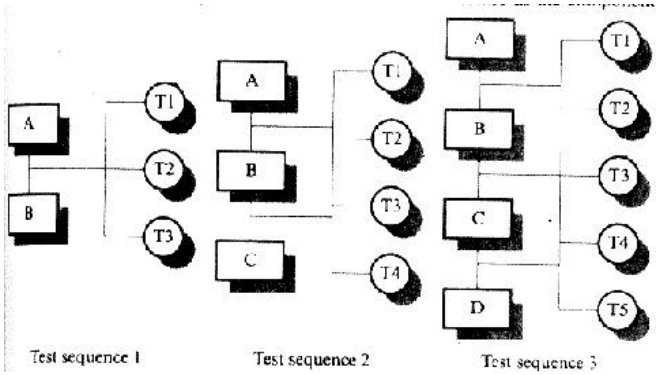
In the example shown in Figure 6, tests TI, T2 and T3 are first run on a system composed of module A and module B. Module C is integrated and tests T1 and T2 are repeated 10 ensure that there have not been unexpected interactions with A and B. TestT4 is also run on the system. Finally, module D is integrated and tested using existing and new tests.

The process should continue until all modules have been incorporated into the system. When a module is introduced at some stage in this process, tests, which were previously unsuccessful, may now detect defects. These defects are probably due to interactions with the new module. The source of the problem is localized to some extent, thus simplifying defect location and repair.

Top-down testing

Top-down testing tests the high levels of a system before testing its detailed components. The program is represented as a single abstract component with sub- components represented by stubs. Stubs have the same interface as the component but very limited functionality. After the top-level component has been tested, its sub-components are implemented and tested in the same way. This process continues recursively until the bottom-level components are implemented. The whole system may then be completely tested. Figure 7 illustrates this sequence.

Top-down testing should be used with top-down program development so that a system component is tested as soon as it is coded. Coding and testing are a single activity with no separate component or module-testing phase.



If top-down testing is used, unnoticed design errors may be detected at an early stage in the testing process. As these errors are usually structural errors, early detection means that they can be corrected without undue costs. Early error detection means that extensive re-design and re-implementation may be avoided.

Top-down testing has the further advantage that a limited, working system is available at an early stage in the development. This is an important psychological boost to those involved in the system development. It demonstrates the feasibility of the system to management. Validation, as distinct from verification, can begin early in the testing process, as a demonstrable system can be made available to users.

Strict top-down testing is difficult to implement because of the requirement that program stubs, simulating lower levels of the system, must be produced. These program stubs may either be implemented, as a simplified version of the component required which returns some random value of the correct type or by manual simulation. The stub simply requests the tester to input an appropriate value or to simulate the action of the component.

If the component is a complex one, it may be impractical to produce a program stub, which simulates it accurately. Consider a function which relies on the conversion of an array of objects into a linked list. Computing its result involved internal program objects, the pointers linking elements in the list. It is unrealistic 10 generate a random list and return that object. The

list components must correspond to the array elements. It is equally unrealistic for the programmer to input the created list. This requires knowledge of the internal pointer representation Therefore, the routine to perform the conversion from array to list must exist before top-down testing is possible.

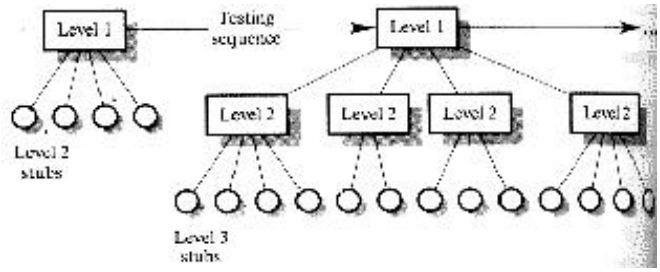


Figure 7 : Top-down testing.

Another disadvantage of top-down testing is that test output may be difficult to observe. In many systems, the higher levels of that system do not generate output but, to test these levels, they must be forced to do so. The tester must create an artificial environment to generate the test results.

Collections of objects are not usually integrated in a strictly hierarchical way so a strict top-down testing strategy is not appropriate for object-oriented systems. However, individual objects may be tested using this approach where operations are replaced by stubs.

Bottom-up testing

Bottom-up testing is the converse of cop-down testing; it involves testing the modules at the lower levels **in** the hierarchy, and then working up the hierarchy of modules until the final module is tested (Figure 8). The advantages of bottom-up testing are the disadvantages of top-down testing and vice versa.

When using bottom-up testing, test drivers must be written to exercise the lower-level components. These test drivers simulate the components' environment and are valuable components in their own right. If the components being tested are reusable components, the test drivers and test data should be distributed with the component. Potential reusers can then run these tests to satisfy themselves that the component behaves as expected in their environment.

If top-down development is combined with bottom-up testing, all parts of the system must be implemented before testing can begin. Architectural faults are unlikely to be discovered until much of the system has been tested. Collection of these faults might involve the rewriting and consequent re-testing of lower-level modules in the system.

Because of this problem, bottom-up testing was criticized by the proponents of top-down functional development in the 1970s. However, a strict top-down devel-opment process including testing is an impractical-approach, particularly if existing software components are to be reused. Bottom-up testing of critical, low-level system components is almost always necessary.

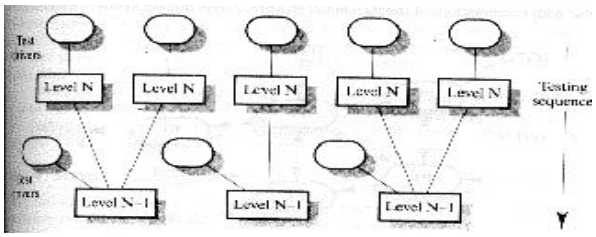


Figure 8: Bottom-up testing.

Bottom-up testing is appropriate for object-oriented systems in that individual objects may be tested using their own test drivers. They are then integrated and the object collection is tested. The testing of these collections should focus on object interactions. An approach such as the 'method-message' path strategy, discussed in this chapter earlier, may be used.

Thread testing

Thread testing is a testing strategy, which was devised for testing real-time systems. It is an event-based approach where tests are based on the events, which trigger system actions. A comparable approach may be used to test object-oriented system as they may be modeled as event-driven systems. Bezier (1990) discusses this approach in detail but he calls it 'transaction-flow testing' rather than thread testing,.

Thread testing is a testing strategy, which may be used after processes, or objects have been individually tested and integrated into sub-systems. The processinig of each external event 'threads' its way through the system processes or objects with some processing carried out at each stage. Thread testing involves identifying and executing each possible processing 'thread'. Of course, complete thread testing may be impossible because of the number of possible input and output combinations. In such cases, the most commonly exercised threads should be identified and selected for testing.

Consider the real-time system made up of five interacting processes shown in Figure 9. Some processes accept inputs from their environment and generate outputs to that environment. These inputs may be from sensors, keyboards or from some other computer system. Similarly, outputs may be to control lines, other computers or user terminals. Inputs from the environment are labeled with an l outputs with an O. The mows connecting processes mean that an event of some kind (with associated data) is generated by the source of the mow and processed by the process at the head of the arrow.

As part of the testing process, the system should be analyzed to identify as many threads as possible. Threads are not just. Associated with individual events but also with combinations of inputs, which can arise. These threads should be identified

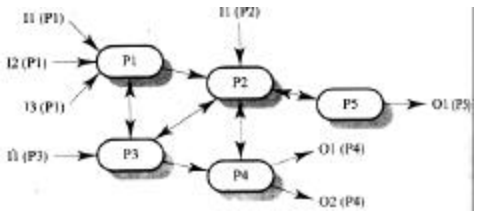


Figure 9 : Real-time process interaction

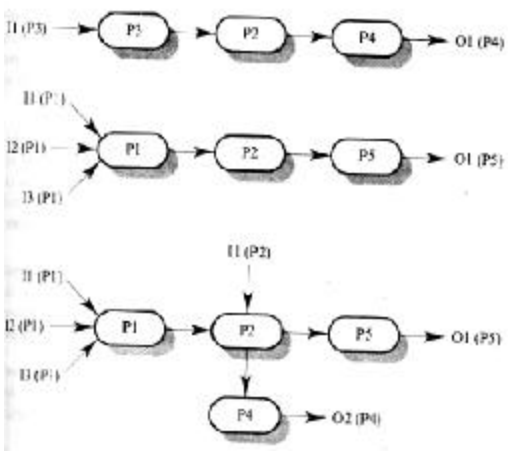


Figure 10: Single thread testing

Figure 11: Multiple-input thread testing

Figure 12: Multiple thread testing

from an architectural model of the system which shows process interactions and from descriptions of system input and output.

A possible thread is shown in Figure 10 where an input is transformed by a number of processes in turn to produce an output. In Figure10, the thread of control passes through a sequence of processes from P3 to P2 to P4. Threads can be recursive so that processes or objects appear more than once in the thread. This is normal in object-oriented systems where control returns to the calling object from the called object. The same object therefore appears at different places in the thread.

After each thread has been tested with a single event, the processing of multiple events of the same type should be tested without events of any other type. For example, a multi-user system might first be tested using a single terminal then multiple terminal testing gradually introduced. In the above model. This type of testing might involve processing all inputs in multiple-input processes. Figure 11 illustrate an example 'of this process where three inputs to the same process are used as test data.

After the system's reaction to each class of event has been tested, it can then be tested for its reactions to more than one class of simultaneous event. At this stage, new event tests should be introduced gradually so that system errors can be localized. This might be tested as shown in Figure 12.

Stress testing

Some classes of system are designed to handle a specified load. For example, a transaction processing system may be designed to process up to 100 transactions per second; an operating system may be designed to handle up to 200 separate tenninals, Tests have to be designed to ensure that the system can process its intended load. This usually involves planning a series of tests where the load is steadily increased.

Stress testing continues these tests beyond the maximum design load of the system until the system fails. This type of testing has two functions:

1. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.
2. It stresses the system and may cause defects to come to light, which would not normally manifest themselves. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unused combinations of normal circumstances which the stress testing replicates,.

Stress testing is particularly relevant to distributed systems based on a network *of* processors. These systems often exhibit severe degradation when they are heavily loaded as the network becomes swamped with data, which the different processes must exchange.

Back-to-back testing

Back-to-back testing may be used when more than one version of a system is available for testing. The same tests are presented to both versions of the system and the test results compared. Differences between these test results highlight potential system problems (Figure 13).

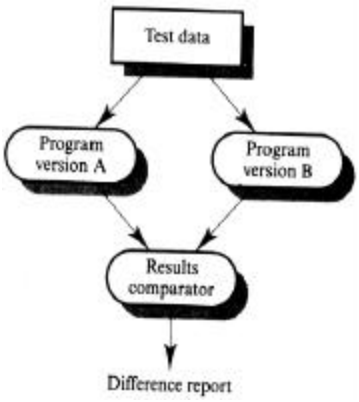
Of course, it is not usually realistic to generate a completely new system just for testing so back-to-back testing is usually only possible in the following situations

1. When a system prototype is available.
2. When reliable systems are developed-using N-version programming.
3 When different versions of a system have been developed for different types of computer.

Alternatively, where a new version of a system has been produced with some functionality in common with previous versions, the tests on this new version can be compared with previous test runs using the older version.

The steps involved in back-to-back testing are:

1. Prepare a general-purpose set of test cases.
2. Run one version of the program with these test, cases and save the results in one or more files



3. Run another version of the program with the same test cases, saving the results to a different file.
4. Automatically compare the files produced by the modified and unmodified program versions.

If the programs behave in the same way, the file comparison should show the output files to be identical. Although this does not guarantee that they are valid (the imple-mentors of both versions may have made the same mistake), it is probable that the programs are behaving correctly. Differences between the outputs suggest problems which should be investigated in more detail.

Notes

LESSON 34 Error Handling Functions

Objectives

Upon completion of this Lesson, you should be able to:

- Know about various kinds of errors
- Know about the error handling functions.

Errors

Compile time errors also called syntax errors. These errors occur when we do not obey the syntax or grammar of the C language. They are discovered by the compiler in statements it does not recognize as valid C statement.

Compilation Errors

A compilation error occurs when the compiler finds something in the source code that it can't compile. A misspelling, typographical error, or any of a dozen other things can cause the compiler to choke. Fortunately, modern compilers don't just choke; they tell you what they're choking on and where it is! This makes it easier to find and correct errors in your source code.

This point can be illustrated by introducing a deliberate error into HELLO.C. If you worked through that example (and you should have), you now have a copy of HELLO.C on your disk. Using your editor, move the cursor to the end of the line containing the call to printf(), and erase the terminating semicolon. HELLO.C should now look like below program.

Program: HELLO.C with an error.

```
1: #include <stdio.h>
2:
3: main()
4: {
5: printf("Hello, World!")
6: return 0;
7: }
```

Next, save the file. You're now ready to compile it. Do so by entering the command for your compiler. Because of the error you introduced, the compilation is not completed. Rather, the compiler displays a message similar to the following:

hello.c(6) : Error: ';' expected

Looking at this line, you can see that it has three parts:

hello.c The name of the file where the error was found

(6) : The line number where the error was found

Error: ';' expected A description of the error

This message is quite informative, telling you that in line 6 of HELLO.C the compiler expected to find a semicolon but didn't. However, you know that the semicolon was actually omitted from line 5, so there is a discrepancy. You're faced with the puzzle of why the compiler reports an error in line 6 when, in fact, a semicolon was omitted from line 5. The answer lies in the fact that C doesn't care about things like breaks between lines. The semicolon that belongs after the printf() statement could have been placed on the next line (although doing so would be bad programming practice). Only after encountering the next command (return) in This points out an undeniable fact about C compilers and error messages. Although the compiler is very clever about detecting and localizing errors, it's no Einstein. Using your knowledge of the C language, you must interpret the compiler's messages and determine the actual location of any errors that are reported. They are often found on the line reported by the compiler, but if not, they are almost always on the preceding line. You might have a bit of trouble finding errors at first, but you should soon get better at it.

NOTE: The errors reported might differ depending on the compiler. In most cases, the error message should give you an idea of what or where the problem is. Before leaving this topic, let's look at another example of a compilation error. Load HELLO.C into your editor again and make the following changes:

1. Replace the semicolon at the end of line 5.

2. Delete the double quotation mark just before the word Hello.

Save the file to disk and compile the program again. This time, the compiler should display error messages

similar to the following:

hello.c(5) : Error: undefined identifier 'Hello'

hello.c(7) : Lexical error: unterminated string

Lexical error: unterminated string

Lexical error: unterminated string

Fatal error: premature end of source file

The first error message finds the error correctly, locating it in line 5 at the word Hello. The error message undefined identifier means that the compiler doesn't know what to make of the word Hello, because it is no longer enclosed in quotes. However, what about the other four errors that are reported? These errors, the meaning of which you don't need to worry about now, illustrate the fact that a single error in a C program can sometimes cause multiple error messages.

The lesson to learn from all this is as follows: If the compiler reports multiple errors, and you can find only one, go ahead and fix that error and recompile. You might find that your single correction is all that's needed, and the program will compile without errors.

Linker Error Messages

Linker errors are relatively rare and usually result from misspelling the name of a C library function. In this case, you get an Error: undefined symbols: error message, followed by the misspelled name (preceded by an underscore). Once you correct the spelling, the problem should go away.

Exercises

1. Use your text editor to look at the object file created by Listing 1.1. Does the object file look like the source file? (Don't save this file when you exit the editor.)
2. Enter the following program and compile it. What does this program do? (Don't include the line numbers or colons.)

```
1: #include <stdio.h>
2:
3: int radius, area;
4:
5: main()
6: {
7: printf( "Enter radius (i.e. 10): " );
8: scanf( "%d", &radius );
9: area = (int) (3.14159 * radius * radius);
10: printf( "\n\nArea = %d\n", area );
11: return 0;
12: }
```

3. Enter and compile the following program. What does this program do?

```
1: #include <stdio.h>
2:
3: int x,y;
4:
5: main()
6: {
7: for ( x = 0; x < 10; x++, printf( "\n" ) )
8: for ( y = 0; y < 10; y++ )
9: printf( "X" );
10:
11: return 0;
12: }
```

4. BUG BUSTER: The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
1: #include <stdio.h>
2:
3: main();
4: {
5: printf( "Keep looking!" );
6: printf( "You\'ll find it!\n" );
7: return 0;
8: }
```

5. BUG BUSTER: The following program has a problem. Enter it in your editor and compile it. Which lines generate problems?

```
1: #include <stdio.h>
2:
3: main()
4: {
5: printf( "This is a program with a " );
6: do_it( "problem!");
7: return 0;
8: }
```

6. Make the following change to the program in exercise 3. Recompile and rerun this program. What does the program do now?

```
9: printf( "%c", 1 );
```

Error-Handling Functions

The C standard library contains a variety of functions and macros that help you deal with program errors.

The assert() Function

The macro assert() can diagnose program bugs. It is defined in ASSERT.H, and its prototype is void assert(int *expression*); The argument *expression* can be anything you want to test—a variable or any C expression. If *expression* evaluates to TRUE, assert() does nothing. If *expression* evaluates to FALSE, assert() displays an error message on stderr and aborts program execution.

How do you use assert()? It is most frequently used to track down program bugs (which are distinct from compilation errors). A bug doesn't prevent a program from compiling, but it causes it to give incorrect results or to run improperly (locking up, for example). For instance, a financial-analysis program you're writing might occasionally give incorrect answers. You suspect that the problem is caused by the variable interest_rate taking on a negative value, which should never happen. To check this, place the statement

```
assert(interest_rate >= 0);
```

at locations in the program where interest_rate is used. If the variable ever does become negative, the assert() macro alerts you. You can then examine the relevant code to locate the cause of the problem.

To see how assert() works, run program given below. If you enter a nonzero value, the program displays the value and terminates normally. If you enter zero, the assert() macro forces abnormal program termination. The exact error message you see will depend on your compiler, but here's a typical example:

```
Assertion failed: x, file list19_3.c, line 13
```

Note that, in order for assert() to work, your program must be compiled in debug mode. Refer to your compiler documentation for information on enabling debug mode (as explained in a moment). When you later compile the final version in release mode, the assert() macros are disabled.

Program: Using the assert() macro.

```
1: /* The assert() macro. */
2:
3: #include <stdio.h>
4: #include <assert.h>
5:
6: main()
7: {
```

```
8: int x;
9:
10: printf("\nEnter an integer value: ");
11: scanf("%d", &x);
12:
13: assert(x >= 0);
14:
15: printf("You entered %d.\n", x);
16: return(0);
17: }
```

Output

```
Enter an integer value: 10
You entered 10.
Enter an integer value: -1
Assertion failed: x, file list19_3.c, line 13
Abnormal program termination
```

Your error message might differ, depending on your system and compiler, but the general idea is the same.

Analysis

Run this program to see that the error message displayed by assert() on line 13 includes the expression whose test failed, the name of the file, and the line number where the assert() is located.

The action of assert() depends on another macro named NDEBUG (which stands for "no debugging"). If the macro NDEBUG isn't defined (the default), assert() is active. If NDEBUG is defined, assert() is turned off and has no effect. If you placed assert() in various program locations to help with debugging and then solved the problem, you can define NDEBUG to turn assert() off. This is much easier than going through the program and removing the assert() statements (only to discover later that you want to use them again). To define the macro NDEBUG, use the #define directive. You can demonstrate this by adding the line

```
#define NDEBUG
```

to above program, on line 2. Now the program prints the value entered and then terminates normally, even if you enter -1.

Note that NDEBUG doesn't need to be defined as anything in particular, as long as it's included in a #define directive.

The ERRNO.H Header File

The header file ERRNO.H defines several macros used to define and document runtime errors. These macros are used in conjunction with the perror() function,

The ERRNO.H definitions include an external integer named errno. Many of the C library functions assign a value to this variable if an error occurs during function execution. The file ERRNO.H also defines a group of symbolic constants for these errors, listed in Table 1.

Table 1: The symbolic error constants defined in ERRNO.H.

Name	Value	Message and Meaning
E2BIG	1000	Argument list too long (list length exceeds 128 bytes).
EACCES	5	Permission denied (for example, trying to write to a file opened for read only).
EBADF	6	Bad file descriptor.
EDOM	1002	Math argument out of domain (an argument passed to a math function was outside the allowable range).
EEXIST	80	File exists.
EMFILE	4	Too many open files.
ENOENT	2	No such file or directory.
ENOEXEC	1001	Exec format error.
ENOMEM	8	Not enough core (for example, not enough memory to execute the exec() function).
ENOPATH	3	Path not found.
ERANGE	1003	Result out of range (for example, result returned by a math function is too large or too small for the return data type).

You can use errno two ways. Some functions signal, by means of their return value, that an error has occurred.If this happens, you can test the value of errno to determine the nature of the error and take appropriate action.Otherwise, when you have no specific indication that an error occurred, you can test errno. If it's nonzero, anerror has occurred, and the specific value of errno indicates the nature of the error. Be sure to reset errno to zeroafter handling the error. The next section explains perror(), and then in program given below illustrates the use of errno.

The perror() Function

The perror() function is another of C's error-handling tools. When called, perror() displays a message on stderr describing the most recent error that occurred during a library function call or system call. The prototype, in STDIO.H, is

```
void perror(char *msg);
```

The argument msg points to an optional user-defined message. This message is printed first, followed by a colon and the implementation-defined message that describes the most recent error. If you call perror() when no error has occurred, the message displayed is no error.

A call to perror() does nothing to deal with the error condition. It's up to the program to take action, which might consist of prompting the user to do something such as terminate the program. The action the program takes can be determined by testing the value of errno and by the nature of the error. Note that a program need not include the header file ERRNO.H to use the external variable errno. That header file is required only if your program uses the symbolic error constants listed in Table 1. The below program illustrates the use of perror() and errno for handling runtime errors.

Program: Using perror() and errno to deal with runtime errors.

```
1: /* Demonstration of error handling with perror() and errno.
   */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <errno.h>
6:
7: main()
8: {
9: FILE *fp;
10: char filename[80];
```

```
11:
12: printf("Enter filename: ");
13: gets(filename);
14:
15: if (( fp = fopen(filename, "r")) == NULL)
16: {
17: perror("You goofed!");
18: printf("errno = %d.\n", errno);
19: exit(1);
20: }
21: else
22: {
23: puts("File opened for reading.");
24: fclose(fp);
25: }
26: return(0);
27: }
```

Output

```
Enter file name: list19_4.c
File opened for reading.
Enter file name: notafile.xxx
You goofed!: No such file or
r directory
errno = 2.
```

Analysis

This program prints one of two messages based on whether a file can be opened for reading. Line 15 tries to open a file. If the file opens, the else part of the if loop executes, printing the following message:

```
File opened for reading.
```

If there is an error when the file is opened, such as the file not existing, lines 17 through 19 of the if loop execute. Line 17 calls the perror() function with the string "You goofed!". This is followed by printing the error number. The result of entering a file that does not exist is

```
You goofed!: No such file or directory.
errno = 2
```

- DO include the ERRNO.H header file if you're going to use the symbolic errors listed in Table 1.
- DON'T include the ERRNO.H header file if you aren't going to use the symbolic error constants listed in Table 1.
- DO check for possible errors in your programs. Never assume that everything is okay.

Notes

LESSON 35 Types of Errors

Objectives

Upon completion of this Lesson, you should be able to:

- Know about beginner errors
- Know about the various other errors.

This document lists the common C programming errors that the author sees time and time again. Solutions to the errors are also presented.

Beginner Errors

These are errors that beginning C students often make. However, the professionals still sometimes make them too!

Forgetting to put a break in a switch statement

Remember that C does not break out a switch statement if a case is encountered. For example:

```
int x = 2;
switch(x) {
case 2:
  printf("Two\n");
case 3:
  printf("Three\n");
}
```

prints out:

Two

Three

Put a break to break out of the switch:

```
int x = 2;
switch(x) {
case 2:
  printf("Two\n");
  break;
case 3:
  printf("Three\n");
  break;   /* not necessary
```

Using = instead of ==

C's = operator is used exclusively for assignment and returns the value assigned. The == operator is used exclusively for comparison and returns an integer value (0 for *false*, not 0 for *true*). Because of these return values, the C compiler often does not flag an error when = is used when one really wanted an ==. For example:

```
int x = 5;
if ( x = 6 )
  printf("x equals 6\n");
```

This code prints out x equals 6! Why? The assignment inside the if sets x to 6 and returns the value 6 to the if. Since 6 is not 0, this is interpreted as *true*.

One way to have the compiler find this type of error is to put any constants (or any r-value expressions) on the left side. Then if an = is used, it will be an error:

```
if ( 6 = x)
```

scanf() errors

There are two types of common scanf() errors:

Forgetting to put an ampersand (&) on arguments

scanf() must have the address of the variable to store input into. This means that often the ampersand address operator is required to compute the addresses. Here's an example:

```
int x;
char * st = malloc(31);
scanf("%d", &x);   /* & required to pass address to scanf()   */
scanf("%30s", st); /* NO & here, st itself points to variable! */
```

As the last line above shows, sometimes no ampersand is correct!

Using the wrong format for operand C compilers do *not* check that the correct format is used for arguments of a scanf() call. The most common errors are using the %f format for doubles (which must use the %lf format) and mixing up %c and %s for characters and strings.

Size of Arrays

Arrays in C always start at index 0. This means that an array of 10 integers defined as:

```
int a[10];
```

has valid indices from 0 to 9 *not* 10! It is very common for students go one too far in an array. This can lead to unpredictable behavior of the program.

Integer Division

Unlike Pascal, C uses the / operator for both real and integer division. It is important to understand how C determines which it will do. If both operands are of an integal type, integer division is used, else real division is used. For example:

```
double half = 1/2;
```

This code sets half to 0 not 0.5! Why? Because 1 and 2 are integer constants. To fix this, change at least one of them to a real constant.

```
double half = 1.0/2;
```

If both operands are integer variables and real division is desired, cast one of the variables to double (or float).

```
int x = 5, y = 2;
double d = ((double) x)/y;
```

Loop errors

In C, a loop repeats the very next statement after the loop statement. The code:

```
int x = 5;
while( x > 0 );
  x—;
```

is an infinite loop. Why? The semicolon after the while defines the statement to repeat as the null statement (which does nothing). Remove the semicolon and the loop works as expected.

Another common loop error is to iterate one too many times or one too few. Check loop conditions carefully!

Not using prototypes

Prototypes tell the important features of a function: the return type and the parameters of the function. If no prototype is given, the compiler *assumes* that the function returns an int and can take any number of parameters of any type.

One important reason to use prototypes is to let the compiler check for errors in the argument lists of function calls. However, a prototype *must* be used if the function does not return an int. For example, the sqrt() function returns a double, not an int. The following code:

```
double x = sqrt(2);
```

will not work correctly if a prototype:

```
double sqrt(double);
```

does not appear above it. Why? Without a prototype, the C compiler assumes that sqrt() returns an int. Since the returned value is stored in a double variable, the compiler inserts code to convert the value to a double. This conversion is not needed and will result in the wrong value. The solution to this problem is to include the correct C header file that contains the sqrt() prototype, math.h. For functions you write, you must either place the prototype at the top of the source file or create a header file and include it.

Not initializing pointers

Anytime you use a pointer, you should be able to answer the question: *What variable does this point to?* If you can not answer this question, it is likely it doesn't point to *any* variable. This type of error will often result in a Segmentation fault/coredump error on the AIX or a general protection fault under Windows. (Under good old DOS (ugh!), anything could happen!)

Here's an example of this type of error.

```
#include <string.h>
int main()
{
 char * st;   /* defines a pointer to a char or char array */
 strcpy(st, "abc");  /* what char array does st point to?? */
 return 0;
}
```

How to do this correctly? Either use an array or dynamically allocate an array.

```
#include <string.h>
int main()
{
 char st[20];   /* defines an char array */
 strcpy(st, "abc");  /* st points to char array */
 return 0;
}
```

or

```
#include <string.h>
#include <stdlib.h>
int main()
{
 char *st = malloc(20);   /* st points to allocated array*/
 strcpy(st, "abc");  /* st points to char array */
 free(st);
 return 0;
}
```

Actually, the first solution is much preferred for what this code does. Why? Dynamical allocation should only be used when it is required. It is slower and more error prone than just defining a normal array.

"The lesson content has been compiled from various sources in public domain including but not limited to the internet for the convenience of the users. The university has no proprietary right on the same."

Rai Technology University

ENGINEERING MINDS

Rai Technology University Campus

Dhodballapur Nelmangala Road, SH -74, Off Highway 207, Dhodballapur Taluk, Bangalore - 561204

E-mail: info@raitechuniversity.in | Web: www.raitechuniversity.in