C Programming Tutorial

Lesson 1: Writing And Compiling A First Program
Hello World

The lessons in this tutorial will take you from being a beginner to being able to write real programs in C.

C is a compiled language. The C compiler is a program that reads source code, which is the C code written by a programmer, and produces an executable or binary file that in a format that can be read and executed (run) by a computer. The source file is a plain text file containing your code. The executable file consists of machine code, 1's and 0's that are not meant to be understood or read by people, but only by computers.

The best way to learn anything is to jump right in, so let's start by writing a simple C program.

First Program - Hello World

```
#include <stdio.h>
main()
{
    printf("Hello World From About\n");
}
```

Line 1: #include <stdio.h>
As part of compilation, the C compiler runs a program called the C preprocessor. The preprocessor is able to add and remove code from your source file. In this case, the directive #include tells the preprocessor to include code from the file stdio.h. This file contains declarations for functions that the program needs to use. A declaration for the printf function is in this file.

Line 2: void main()
This statement declares the main function. A C program can contain many functions but must always have one main function. A function is a self-contained module of code that can accomplish some task. The "void" specifies the return type of main. In this case, nothing is returned to the operating system.

Line 3: {
This opening bracket denotes the start of the program.

Line 4: printf("Hello World \n");
printf is a function from a standard C library that is used to print strings to the standard output, normally your screen. The compiler links code from these standard libraries to the code you have written to produce the final executable. The "\n" is a special format modifier that tells the printf to put a line feed at the end of the line. If there were another printf in this program, its string would print on the next line.

Line 5: }
This closing bracket denotes the end of the program.

That's it. To get the most of this series of tutorials, you should get access to both a text editor and a C compiler. Some instructions for doing this are in our tutorials on compiling.

Lesson 2: Variables

This lesson will teach you how to declare and use variables in C. A variable is used to hold data within your program. A variable represents a location in your computer's memory. You can put data into this location and retrieve data out of it. Every variable has two parts, a name and a data type.

Variable Names

Valid names can consist of letters, numbers and the underscore, but may not start with a number. A variable name may not be a C keyword such as if, for, else, or while. Variable names are case sensitive. So, Age, AGE, aGE and AgE could be names for different variables, although this is not recommended since it would probably cause confusion and errors in your programs. Let's look at some variable declarations to better understand these rules. Note that int, float and double are built in C data types as explained latter in this lesson.

Which of the following are valid variable names?

```
int idnumber;
int transaction_number;
int __my_phone_number__;
float 4myfriend;
float its4me;
double VeRyStRaNgE;
float while;
float myCash;
int CaseNo;
int CASENO;
int caseno;
```

Data Types

C provides built in data types for character, float and integer data. A mechanism, using the keyword typedef, exists for creating user-defined types. As an aside, in C you may assign a value to a variable when you declare it.

Integer variables are used to store whole numbers. There are several keywords used to declare integer variables, including int, short, long, unsigned short, unsigned long. The difference deals with the number of bytes used to store the variable in memory, long vs. short, or whether negative and positive numbers may be stored, signed vs. unsigned. These differences will be explained in more advanced tutorials. For now, use int to declare integer variables. On most 32-bit systems, int is synonymous with signed long.

Examples:

```
int count;
int number_of_students = 30;
```

Float variables are used to store floating point numbers. Floating point numbers may contain both a whole and fractional part, for example, 52.7 or 3.33333333. There are several keywords used to declare floating point numbers in C including float, double and long double. The difference here is the number of bytes used to store the variable in memory. Double allows larger values than float. Long double allows even larger values. These differences will be explained in more advanced tutorials. For now, use float to declare floating point variables.

Examples:

```
float owned = 0.0;
float owed = 1234567.89;
```

Character variables are used to store character values. The use of characters and strings will be covered in a latter tutorial. Character variables are declared with the keyword char.

Examples:

```
char firstInitial = 'J';
char secondInitial = 'K';
```

Typedef

It is sometimes very convenient to define new data type. In C this can be done using the typedef keyword. Its syntax is:

```
typedef data_type new_name;
```

Typedef is creating a synonym "new_name" for "data_type" Here's an example of its use. Suppose we need to create a program to track the grades of students in a school. Each student has an ID number, and a name to store and manipulate in many places and functions through out the program. The ID number will be a position small integer. The name will be stored in a character array of length 60. Arrays, which will be fully introduced in a latter lesson, are used to store multiple items of the same type sequentially in memory. In this case, a space in memory of 60 bytes will be allocated to store each name. Here are declarations for these variables for one student.

```
unsigned short int myID;    /* ID numbers are positive (unsigned) and small (short) */
char myName[60];    /* Character array of length 60 */
```

It is hard and potentially error prone to type out "unsigned short int" and the character array declaration. This is particularly true if there are multiple functions in our program and variables of this type need to be declared in each. Typedef can be used to create short meaningful synonyms for these types. Here's an example of the use of typedef.

```
#include <stdio.h>

int main()
```

```
{
    typedef unsigned short int ID;
    typedef char Name[60];

    ID myID = 123;
    Name myName = "John K";

    printf("Good work %s or should I say %d\n",myName,myID);

    return 0;
}
```

Lvalues/Rvalues
C has the notion of lvalues and rvalues associated with variables and constants. The rvalue is the data value of the variable, that is, what information it contains. The "r" in rvalue can be thought of as "read" value. A variable also has an associated lvalue. The "l" in lvalue can be though of as location, meaning that a variable has a location that data or information can be put into. This is contrasted with a constant. A constant has some data value, that is, an rvalue. But, it cannot be written to. It does not have an lvalue.

Another view of these terms is that objects with an rvalue, namely a variable or a constant can appear on the right hand side of a statement. They have some data value that can be manipulated. Only objects with an lvalue, such as variable, can appear on the left hand side of a statement. An object must be addressable to store a value.

Here are two examples.

```
int x;

x = 5;  /* This is fine, 5 is an rvalue, x can be an lvalue. */
5 = x;  /* This is illegal. A literal constant such as 5 is not */
        /* addressable. It cannot be a lvalue. */
```

Lesson 3: Constants

This lesson will teach you how to declare and use constants in C. A constant is similar to a variable in the sense that it represents a memory location. It differs, as I sure you can guess, in that it cannot be reassigned a new value after initialization. In general, constants are a useful feature that can prevent program bugs and logic errors. Unintended modifications are prevented from occurring. The compiler will catch attempts to reassign new values to constants.

Using #define
There are three techniques used to define constants in C. First, constants may be defined using the preprocessor directive #define. The preprocessor is a program that modifies your source file prior to compilation. Common preprocessor directives are #include, which is used to include additional code into your source file, #define, which is used to define a constant and #if/#endif, which can be used to conditionally determine which parts of your code will be compiled. The #define directive is used as follows.

```
#define pi 3.1415
#define id_no 12345
```

Wherever the constant appears in your source file, the preprocessor replaces it by its value. So, for instance, every "pi" in your source code will be replace by 3.1415. The compiler will only see the value 3.1415 in your code, not "pi". Every "pi" is just replaced by its value. Here is a simple program illustrating the preprocessor directive #define.

```
#include <stdio.h>

#define monday 1
#define tuesday 2
#define wednesday 3
#define thursday 4
#define friday 5
#define saturday 6
#define sunday 7

int main()
{

    int today = monday;

    if ((today == saturday) || (today == sunday))

    {
        printf("Weekend\n");
    }
    else
    {
        printf("Go to work or school\n");
    }

    return 0;
}
```

Using const variables

The second technique is to use the keyword const when defining a variable. When used the compiler will catch attempts to modify variables that have been declared const.

```
const float pi = 3.1415;
const int id_no = 12345;
```

There are two main advantages over the first technique. First, the type of the constant is defined. "pi" is float. "id_no" is int. This allows some type checking by the compiler. Second, these constants are variables with a definite scope. The scope of a variable relates to parts of your program in which it is defined. Some variables may exist only in certain functions or in certain blocks of code. You may want to use "id_no" in one function and a completely unrelated "id_no" in your main program. Sorry if this is confusing, the scope of variables will be covered in a latter lesson.

Using enumerations

The third technique to declare constants is called enumeration. An enumeration defines a set of constants. In C an enumerator is of type int. C++ allows variables to be a new enumeration type, which allows the compiler to perform type and bound checking. Unfortunately, C does not provide this. But the enumeration is still useful to define a set of constants instead of using multiple #defines.

```
#include <stdio.h>

enum days {monday=1,tuesday,wednesday,thursday,friday,saturday,sunday};

int main()
{
   enum days today = monday;

   if ((today == saturday) || (today == sunday))
   {
       printf("Weekend\n");
   }
   else
   {
       printf("Go to work or school\n");
   }

   return 0;

}
```

There are a few things to note in this example. First, is clearer to use the enumeration than the #defines used in the earlier example. Next, the variable today is of type int, as are all enumerators. Any expression that can be assigned to an int variable can be assigned to "today". The compiler will not catch bound errors as would occur in C++. For instance, the follow assignment, an error, could appear in the program and not be flagged by the compiler.

```
today = 9999; /* Error, today should be between 1 and 7 */
```

Look at the following examples to understand how to assign values to the constants in enumerations.

```
enum COLOR { RED, BLUE, GREEN};
enum SHAPE {SQUARE, RECTANGLE, TRIANGLE, CIRCLE, ELLIPSE};
```

Each enumerated constant (sometimes called an enumerator) has an integer value. Unless specified, the first constant has a value of zero. The values increase by one for each additional constant in the enumeration. So, RED equals 0, BLUE equals 1, and GREEN = 2. SQUARE equals 0, RECTANGLE equals 1, TRIANGLE equals 2 and so forth. The values of each constant can also be specified.

```
enum SHAPE {SQUARE=5,RECTANGLE,TRIANGLE=17,CIRCLE,ELLIPSE};
```

Here, SQUARE equals 5, RECTANGLE equals 6, TRIANGLE equals 17, CIRCLE equals 18 and ELLIPSE equals 19.

Lesson 4: Input and Output

This lesson covers basic input and output. Usually i/o, input and output, form an important part of any program. To do anything useful your program needs to be able to accept input data and report back your results. In C, the standard library provides routines for input and output. The standard library has functions for i/o that handle input, output, and character and string manipulation. In this lesson, all the input functions described read from standard input and all the output functions described write to standard output. Standard input is usually the keyboard. Standard output is usually the monitor.

Formatted Output

The standard library function printf is used for formatted output. It takes as arguments a format string and an optional list of variables or literals to output. The variables and literals are output according to the specifications in the format string. Here is the prototype for printf.

```
int printf(const char *format, arg1, arg2, arg3, ......);
```

A prototype is used to declare a function in C. It specifies the functions signature, which is the number and type of its arguments and also the return type of the function. So, printf returns an integer value. It will either be the number of characters output, or some negative number signifying that an error has occurred. The format is a character string that can contain ordinary characters, which are output unmodified and conversion specifications, which control how the additional arguments are converted to characters for output. The easiest way to understand this is by example.

```
#include <stdio.h>

int main()
{
   int luckyNumber = 5;
   float radius = 2;
   char myName[15] = "John";
   char initial = 'J';

   printf("Hello World\n");    /* The format string contains only ordinary
                characters. Ordinary characters are output unmodified.
                A character string in C is of type "char *". */

   printf("My lucky number is %d\n", luckyNumber);    /* The "%d" specifies
               that an integer value will be output. */

   printf("My name is %s\n",myName);    /* The %s specifies that a character
               string will be output. */

   printf("My first initial is %c\n",initial);    /* The %c specifies that a
```

```
                character will be output. */

    printf("The area of a circle with radius %f is %f\n", radius, 3.14*radius*radius);
                /* %f specifies that a float will be output. */

    printf("Hello %s or should I say %c\n",myName,initial);
                /* Multiple arguments of different types may be output. */
    return(0);
}
```

Here are the more common conversion specifiers.

Specifier	Argument Type
%d	int
%f	float or double
%e	float or double, output in scientific notation.
%c	character
%s	character string (char *)

Formatted Input

The standard library function scanf is used for formatted input. It takes as its arguments a format string and a list of pointers to variables to store the input values. Similar to its use in printf, the format string can contain literals and conversion specifiers. In C, to return a value from a function to a calling routine, a pointer to a variable is passed into the function. A pointer stores the memory address of another variable. The methods of passing and returning values from functions are described in Passing Arguments to Functions. Don't worry if you don't understand pointers completely. For now, all this means is a slightly different notation in the call to scanf. In later lessons, pointers and functions will be fully explained. Here is the prototype of scanf and a program illustrating its use.

```
int scanf(const char *format, arg1, arg2, ....);
```

Scanf returns an integer, either the number of values read in, or EOF if an end of file is reached. EOF is a special termination character, specified in stdio.h, which designates the end of a file. If no values are successfully read, scanf returns 0. To use scanf in a program, the file stdio.h must be included.

```
#include <stdio.h>

int main()
{
    float radius;
    char yourName[15];
    int number;

    printf("Enter the radius of circle: ");
    scanf("%f",&radius);   /* & is the "address of" operator.
```

```
                The address of radius is passed into scanf. Passing
                the address of a variable is equivalent to passing
                a pointer containing the address of the variable. */
   printf("A circle of radius %f has area %f\n",radius,3.14*radius*radius);

   printf("Enter a number from 1 to 1000: ");
   scanf("%d",&number);
                /* The address of number is passed to scanf */
   printf("Your number is %d\n",number);

   printf("Enter your name: ");
   scanf("%s",yourName);    /* yourName is a character array.
                yourName[0] specifies the first element of the array.
                &yourName[0] specifies the address of the first element
                of the array. In C, an array name by itself is shorthand
                for the address of the first element. So, yourName is
                equivalent to &yourName[0], which is what must be
                passed into scanf. This will be made clearer in the
                lesson covering arrays. */
   printf("Hello %s\n",yourName);

   return(0);
}
```

With one exception scanf will skip over white space such as blanks, tabs and newlines in the input stream. The exception is when trying to read single characters with the conversion specifier %c. In this case, white space is read in. So, it is more difficult to use scanf for single characters. An alternate technique, using getchar, will be described later in this lesson.

Other Useful Standard Library Functions for Input and Output

getchar

getchar reads a single character from standard input. Its prototype is:

```
int getchar();
```

It returns int rather than char because the "end of file", EOF, character must be handled. EOF is an int and is too large to fit in a char variable. A newline character in C is '\n'. This designates the end of a line.

putchar

putchar writes a single character to standard output. Its prototype is:
int putchar(int value);

Here is a simple program which echoes back everything that is typed in. Note that depending on which operating system you are using, EOF is entered by typing control-z or control-d. So, if you try running this code, type control-z or control-d to end execution

```
#include <stdio.h>

int main()
{
    int c;

    while ((c = getchar()) != EOF)
    {
        putchar(c);
    }

    return(0);
}
```

gets
gets reads a line of input into a character array. Its prototype is:

```
char *gets(char *buffer);
```

It returns a pointer to the character string if successful, or NULL if end of file is reached or if an error has occurred. The string is also read into the character array specified as an argument. The character string will be terminated be a "\0", which is standard for C.

puts
puts writes a line of output to standard output. Its prototype is:

```
int puts(const char *buffer);
```

It terminates the line with a newline, '\n'. It will return EOF is an error occurred. It will return a positive number on success.

Here is the "echo" program implemented using the functions gets and puts. If you run this, remember to type control-z or control-d to end execution.

```
#include <stdio.h>

int main()
{
    char buffer[120]; /* Holds input and output strings */
    char *pt; /* A pointer to datatype char */
    int returnCode;

    while ((pt = gets(buffer)) != NULL)
```

```
    {
        returnCode = puts(buffer);
        if (returnCode == EOF)
        {
            printf("Error on output\n");
    }
    }

    return(0);
}
```

There are some very important things to note in the this example. First, gets returns a NULL pointer when EOF is reached. This is the condition that must be checked in while loop. Second, puts returns EOF on failure. A check is made on this, and an appropriate error message is output if an error has occurred. Finally, the character array buffer is declared to be of length 120. The function gets makes no check on the array size to see if it is large enough to hold the string entered. It is up to the programmer to create a character array that is large enough. If the entered string is larger than the array, gets will still put the string in memory starting at the location of the first byte of the designated array. Since the string is longer than the array, the result is that memory past the end of the array will be overwritten with the entered string. This will cause either erratic behavior or failure (core dump) of the program. This weakness in the C language can lead to program bugs. On the other hand, the low level access to memory and data can be of great utility and is a strength of C and C++ that is lacking in some other languages. C++ allows C style handling of strings but also provides a string class with more sophisticated string handling to alleviate the need for the programmer to address this issue.

Practice

1) Try compiling and running some of the examples presented in this lesson.
2) Write a program to read in the name and address of the user and echo back that information.

3) Write a program to prompt a user for a nickname, the year they were born as an integer, and their weight in pounds as a float. Calculate their age. Calculate their weight in kilograms. Write this information to standard output. Note there are 2.2046 pounds per kilogram.

1)

```
#include <stdio.h>

int main()
{
    /* character arrays are sized properly for data */
    char name[60];
    char address[120];
    char city[60];
    char state[20];
    char zip[15];
    char *pt;    /* Used to check return value from gets */

    /* For responses that can include white space such as blanks
    use gets. For single word responses scanf is easier */

    printf("Enter your name: ");
    if ((pt = gets(name)) == NULL)
    {
        printf("Error on read\n");
        return(1);
    }

    printf("Enter your address: ");
    if ((pt = gets(address)) == NULL)
    {
        printf("Error on read\n");
        return(1);
    }

    printf("Enter your city: ");
    scanf("%s",city);

    printf("Enter your state: ");
    scanf("%s",state);

    printf("Enter your zip: ");
    scanf("%s",zip);

    /* Output user information */
    printf("%s\n",name);
    printf("%s\n",address);
    printf("%s, %s %s\n",city,state,zip);

    return(0);
}
```

2)

```
#include <stdio.h>

int main()
{
    /* Note the data types used. */
```

```
    char nickname[60];
    int presentYear = 2002;
    int birthYear = 2002;
    int age;
    float weightlbs = 0;
    float poundsPerKilo = 2.2046;
    float weightkgs;

    /* For responses that can include white space such as blanks
    use gets. For single word responses scanf is easier */

    printf("Enter your nickname: ");
    scanf("%s",nickname);

    printf("Enter the year you were born: ");
    scanf("%d",&birthYear);    /* Notice that a pointer to birthYear, or
                        its address must be passed to scanf */

    printf("Enter your weight: ");
    scanf("%f",&weightlbs);    /* Note: address of weightlbs passed to scanf */

    age = presentYear - birthYear; /* Close but really depends on
                        month and day of birth and present
                        month and day as well */

    weightkgs = weightlbs / poundsPerKilo;

    /* Output user information */
    printf("Hello %s\n",nickname);
    printf("You are %d years old\n",age);
    printf("You weigh %f kilograms\n",weightkgs);

    return(0);
}
```

3)

```
#include <stdio.h>

int main()
{
    /* Note the data types used. */
    char nickname[60];
    int presentYear = 2002;
    int birthYear = 2002;
    int age;
    float weightlbs = 0;
    float poundsPerKilo = 2.2046;
    float weightkgs;

    /* For responses that can include white space such as blanks
    use gets. For single word responses scanf is easier */

    printf("Enter your nickname: ");
    scanf("%s",nickname);
```

```
    printf("Enter the year you were born: ");
    scanf("%d",&birthYear);    /* Notice that a pointer to birthYear, or
                        its address must be passed to scanf */

    printf("Enter your weight: ");
    scanf("%f",&weightlbs);    /* Note: address of weightlbs passed to scanf */

    age = presentYear - birthYear; /* Close but really depends on
                        month and day of birth and present
                        month and day as well */

    weightkgs = weightlbs / poundsPerKilo;

    /* Output user information */
    printf("Hello %s\n",nickname);
    printf("You are %d years old\n",age);
    printf("You weigh %f kilograms\n",weightkgs);

    return(0);
}
```

Lesson 5: Conditional Processing, Part 1

If/Else Statements and Relational Operators

This lesson introduces conditional processing. In previous tutorials, all the code in the examples executed, that is, from the first line of the program to the last, every statement was executed in the order it appeared in the source code. This may be correct for some programs, but others need a way to choose which statements will be executed or run. Conditional processing extends the usefulness of programs by allowing the use of simple logic or tests to determine which blocks of code are executed. In this lesson, a simple guessing game will be developed to illustrate the use of conditional execution.

The if statement is used to conditionally execute a block of code based on whether a test condition is true. If the condition is true the block of code is executed, otherwise it is skipped.

```
#include <stdio.h>

int main()
{

    int number = 5;
    int guess;

    printf("I am thinking of a number between 1 and 10\n");
    printf("Enter your guess, please \n");
    scanf("%d",&guess);
    if (guess == number)
    {
        printf("Incredible, you are correct\n");
    }

    return 0;
```

```
}
```

Please try compiling and executing the above. The "==" is called a relational operator. Relational operators, ==, !-, >, >=, <, and <=, are used to compare two operands. The program works, but it needs some improvements. If the user enters 5 as a choice, he gets back a nice message, "Incredible, you are correct". But what happens if the user puts in an incorrect choice? Nothing. No message, no suggestions, nothing. Luckily, for our program user, C has a solution.

The else statement provides a way to execute one block of code if a condition is true, another if it is false.

```
#include <stdio.h>

int main()
{

    int number = 5;
    int guess;

    printf("I am thinking of a number between 1 and 10\n");
    printf("Enter your guess, please\n");
    scanf("%d",&guess);
    if (guess == number)
    {
        printf("Incredible, you are correct\n");
    }
   else
    {
        printf("Sorry, try again\n");
    }

    return 0;
}
```

This is a big improvement. Regardless of whether the guess is correct or not, the user gets some response. But let's try to really get the program to work in a way that's even closer to the real game. When playing this highly enjoyable game for hours with our friends and family, what do we say when an incorrect guess is made? Higher or lower. C has an if/else if construct that can be used to implement this functionality in our program.

```
#include <stdio.h>

int main()
{

    int number = 5;
    int guess;

    printf("I am thinking of a number between 1 and 10\n");
    printf("Enter your guess, please ");
    scanf("%d",&guess);
```

```
    if (guess == number)
    {
        printf("Incredible, you are correct\n");
    }
    else if (guess < number)
    {
        printf("Higher, try again\n");
    }
    else // guess must be too high
    {
        printf("Lower, try again\n");
    }
    return 0;
}
```

It is interesting that note that there is no C keyword "else if", as may exist in other languages. For instance, Perl has a keyword "elsif". The if/else if construct is created out of if and else statements. To see this, the above code can be rewritten as follows:

```
    if (guess == number)
    {
        printf("Incredible, you are correct\n");
    }
    else
        if (guess < number)
        {
            printf("Higher, try again\n");
        }
        else /* guess must be too high */
        {
            printf("Lower, try again\n");
        }
```

This code is identical to the code in the program. Only the spacing has been changed to illustrate how if/else if statements are constructed.

Practice

Write a game program that allows a user to guess the day of your birthday. Add logic to your program to limit the guesses between 1 and 31, since the days of a month are in this limit. Print out appropriate error messages if the guess is outside of this range. Add logic and messages to guide the user towards the correct answer, that is, output "Higher" or "Lower" if the guess if wrong.

Solution to Practice Problem - If/Else Example

```
#include <stdio.h>

int main()
{
```

```
    int myBirthday = 13;
    int guess;

    printf("Please guess the day of my birth, from 1 to 31\n");
    printf("Enter your guess, please ");
    scanf("%d",&guess);
    /* Test for out of range guesses first */
    if (guess <= 0)
    {
       printf("Months have at least one day, Einstein\n");
    }
    else if (guess > 31)
    {
       printf("Pretty long month, genius\n");
    }
    else if (guess == myBirthday)
    {
      printf("Incredible, you are correct\n");
    }
    else if (guess < myBirthday)
    {
       printf("Higher, try again\n");
    }
    else /* Guess is in range and not greater than or equal */
    {
       printf("Lower, try again\n");
    }
    return 0;
}
```

Summary of Relational Operators

Operator	Description	Example	Evaluation
==	equal	5 == 4	FALSE
		5 == 5	TRUE
!=	not equal	5 != 4	TRUE
		5 != 5	FALSE
>	greater than	5 > 4	TRUE
		5 > 5	FALSE
>=	greater than or equal	5 >= 4	TRUE
		5 >= 5	TRUE
<	less than	5 < 4	FALSE
		5 < 5	FALSE
<=	less than or equal	5 <= 4	FALSE
		5 <= 5	TRUE

Lesson 6: Conditional Processing, Part 2

Switch Statements and Logical Operators

This second lesson on conditional processing introduces both the switch statement and logical operators. The switch statement is a construct that is used to replace deeply nested or chained if/else statements. Nested if/else statements arise when there are multiple alternative threads of execution based on some condition. Here's an example. Suppose that an ice cream store has asked us to write a program that will automate the taking of orders. We will need to present a menu and then based on the customer's choice take an appropriate action.

```
#include <stdio.h>

int main()
{
    int choice;

    printf("What flavor ice cream do want?\n");
    printf("Enter 1 for chocolate\n");
    printf("Enter 2 for vanilla\n");
    printf("Enter 3 for strawberry\n");
    printf("Enter 4 for green tea flavor, yuck\n");
    printf("Enter you choice: ");

    scanf("%d",&choice);

   if (choice == 1) {
        printf("Chocolate, good choice\n");
    }
    else if (choice == 2) {
        printf("Vanillarific\n");
    }
    else if (choice == 3) {
        printf("Berry Good\n");
    }
    else if (choice == 4) {
        printf("Big Mistake\n");
    }
    else {
       printf("We don't have any\n");
        printf("Make another selection \n");
    }

    return 0;
}
```

This program will work fine, but the if/else block is cumbersome. It would be easy, particularly if there were more choices and maybe sub choices involving more if/else's to end up with program that doesn't perform the actions intended. Here's the same program with a switch.

```
#include <stdio.h>

int main()
{
    int choice;
```

```
    printf("What flavor ice cream do want?\n");
    printf("Enter 1 for chocolate\n");
    printf("Enter 2 for vanilla\n");
    printf("Enter 3 for strawberry\n");
    printf("Enter 4 for green tea flavor, yuck\n");
    printf("Enter you choice: \n");

    scanf("%d",&choice);

    switch (choice) {
    case 1:
        printf("Chocolate, good choice\n");
        break;
    case 2:
        printf("Vanillarific\n");
        break;
    case 3:
        printf("Berry Good\n");
        break;
    case 4:
        printf("Big Mistake\n");
        break;
    default:
        printf("We don't have any\n");
        printf("Make another selection\n");
    }

    return 0;
}
```

The general form of a switch statement is:

```
switch (variable) {
    case expression1:
        do something 1;
        break;
    case expression2:
        do something 2;
        break;
      ....
    default:
        do default processing;
}
```

Each expression must be a constant. The variable is compared for equality against each expression. It is not possible to use the other relational operators discussed in the last lesson or the logical operators that will be introduced latter in this lesson. When an expression is found that is equal to the tested variable, execution continues until a break statement is encountered. It is possible to have a case without a break. This causes execution to fall through into the next case. This is sometimes very useful. Suppose we need to determine if a letter stored in a variable is a vowel or consonant.

```
switch (myLetter) {
```

```
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U':
        vowelCnt++; /* increments vowel count */
            /* same as, vowelCnt = vowelCnt + 1; */
        break;
    default:
        consonantCnt = consonantCnt + 1;
}
```

If any vowels are found execution drops through each case until the count is incremented and the "break" is encountered. For some coding practice, how could the same logic be implemented using if and else's?
ANSWER

In more realistic examples, it is probably necessary to evaluate multiple conditions to determine what parts of code should execute. For instance, if condition 1 is true and condition 2 is true process one way, if condition 1 is true and condition two is false process another way. C provides several logical operators that allow more complex relational expressions to be formed and evaluated.

Logical Operators

Operator	Description	Example	Evaluation
&&	AND	(5 > 3) AND (5 > 6)	FALSE
&&	AND	(5 > 3) AND (5 > 4)	TRUE
||	OR	(5 > 3) OR (5 > 6)	TRUE
||	OR	(5 > 3) OR (5 > 4)	TRUE
!	NOT	!(5 > 3)	FALSE
!	NOT	!(5 > 6)	TRUE

As can be seen in this table, && will return true only if both expressions are true, while || will be true if either expression is true. The operator "!" provides logical negation. One very important consideration when forming expressions is the order of precedence of the relational and logical operators. Relational operators are of higher precedence than the logical and the order of evaluation is from left to right. Here are some examples that illustrate what this means.

```
if (myChoice == 'A' and myAge < 25) is evaluated as
if ((myChoice == 'A') and (myAge < 25))
```

```
Suppose x = 8, y = 49, z = 1.
if (x < 7 && y > 50 || z < 2) is evaluated as
if (((x < 7) && (y > 50))   ||   (z < 2)) which is TRUE, not as
if ((x < 7)   &&   ((y > 50) || (z < 2)) which is FALSE.
```

Now, here are a few final points to wrap up this lesson. First, even if you are sure about the order of precedence of an expression, use explicit parenthesis. This serves to increase readability and will help avoid errors. Second, there is such a thing as green tea ice cream and I recommend that you not buy it.

Lesson 7: Looping

Do, While and For Constructs

This lesson covers three constructs that are used to create loops in C programs. Loops can be created to execute a block of code for a fixed number of times. Alternatively, loops can be created to repetitively execute a block of code until a boolean condition changes state. For instance, the loop may continue until a condition changes from false to true, or from true to false. In this case, the block of code being executed must update the condition being tested in order for the loop to terminate at some point. If the test condition is not modified somehow within the loop, the loop will never terminate. This creates a programming bug known as an infinite loop.

While

The while loop is used to execute a block of code as long as some condition is true. If the condition is false from the start the block of code is not executed at all. Its syntax is as follows.

```
while (tested condition is satisfied) {
   block of code
}
```

Here is a simple example of the use of while. This program counts from 1 to 100.

```
#include <stdio.h>

int main()
{
   int count = 1;

   while (count <= 100)
   {
      printf("%d\n",count);
      count += 1; /* Shorthand for count = count + 1 */
   }

   return 0;
}
```

Here is a more realistic use of while. This program determines the minimum number of bits needed to store a positive integer. The largest unsigned number that can be stored in N bits is (2^N - 1).

```
#include <stdio.h>

int main()
{
```

```
    int bitsRequired = 1; //the power of 2
    int largest = 1; //largest number that can be stored
    int powerOf2 = 2;
    int number;

    printf("Enter a positive integer: ");
    scanf("%d",&number);

    while (number > largest)
    {
        bitsRequired += 1; //Shorthand for bitsRequired = bitsRequired + 1
        powerOf2 = powerOf2 * 2;
        largest = powerOf2 - 1;
    }

    printf("To store %d requires %d bits",number,bitsRequired);

    return 0;
}
```

Do

The do loop also executes a block of code as long as a condition is satisfied. The difference between a "do" loop and a "while" loop is that the while loop tests its condition at the top of its loop; the "do" loop tests its condition at the bottom of its loop. As noted above, if the test condition is false as the while loop is entered the block of code is skipped. Since the condition is tested at the bottom of a do loop, its block of code is always executed at least once. The "do" loops syntax is as follows

```
do {
    block of code
} while (condition is satisfied)
```

Here is an example of the use of a do loop. The following program is a game that allows a user to guess a number between 1 and 100. A "do" loop is appropriate since we know that winning the game always requires at least one guess.

```
#include <stdio.h>

int main()
{

    int number = 44;
    int guess;

    printf("Guess a number between 1 and 100\n");
    do {
        printf("Enter your guess: ");
        scanf("%d",&guess);

        if (guess > number) {
            printf("Too high\n");
        }
        if (guess < number) {
```

```
            printf("Too low\n");
        }
    } while (guess != number);

    printf("You win. The answer is %d",number);

    return 0;
}
```

For

The third looping construct in C is the for loop. The for loop can execute a block of code for a fixed number of repetitions. Its syntax is as follows.

```
for (initializations;test conditions;actions)
{
    block of code
}
```

The simplest way to understand for loops is to study several examples.

First, here is a for loop that counts from 1 to 10.

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

The test conditions may be unrelated to the variables being initialized and updated (assigned). Here is a loop that counts until a user response terminates the loop.

```
for (count = 1; response != 'N'; count++)
{
    printf("%d\n",count);
    printf("Continue (Y/N): \n");
    scanf("%c",&response);
}
```

More complicated test conditions are also allowed. Suppose the user of the last example never enters "N", but the loop should terminate when 100 is reached, regardless.

```
for (count = 1; (response != 'N') && (count <= 100); count++)
{
    printf("%d\n",count);
    printf("Continue (Y/N): \n");
    scanf("%c",&response);
}
```

It is also possible to have multiple initializations and multiple actions. This loop starts one counter at 0 and another at 100, and finds a midpoint between them.

```
for (i = 0, j = 100; j != i; i++, j--)
{
    printf("i = %d, j = %d\n",i,j);
}
printf("i = %d, j = %d\n",i,j);
```

Initializations are optional. For instance, suppose we need to count from a user specified number to 100. The first semicolon is still required as a place keeper.

```
printf("Enter a number to start the count: ");
scanf("%d",&count);
for ( ; count < 100 ; count++)
{
    printf("%d\n",count);
}
```

The actions are also optional. Here is a silly example that will repeatedly echo a single number until a user terminates the loop;

```
for (number = 5; response != 'Y';) {
    printf("%d\n",number);
    printf("Had Enough (Y/N) \n");
    scanf("%c",&response);
}
```

Practice For practice, try implementing programs that will count down from 10 to 1 using while, do and for loops.

Solution using while

```
#include <stdio.h>

int main()
{

    int i = 10;

    while (i > 0)
    {
        printf("%d\n",i);
        i = i - 1;
    }

    return 0;
}
```

Solution using do

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

```
int main()
{

    int i = 10;

    do {
        printf("%d\n",i);
        i = i - 1;
    } while (i > 0);

    return 0;
}
```

Solution using for

```
#include <stdio.h>

int main()
{

    int i;

    for (i = 10; i > 0; i--)
    {
        printf("%d\n",i);
    }

    return 0;
}
```

Lesson 8: An Introduction To Pointers

Pointers are variables that hold addresses in C and C++. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages. They are also useful for passing parameters into functions in a manner that allows a function to modify and return values to the calling routine. When used incorrectly, they also are a frequent source of both program bugs and programmer frustration.

Introduction
As a program is executing all variables are stored in memory, each at its own unique address or location. Typically, a variable and its associated memory address contain data values. For instance, when you declare:

```
int count = 5;
```

The value "5" is stored in memory and can be accessed by using the variable "count". A pointer is a special type of variable that contains a memory address rather than a data value. Just as

data is modified when a normal variable is used, the value of the address stored in a pointer is modified as a pointer variable is manipulated.

Usually, the address stored in the pointer is the address of some other variable.

```
int *ptr;
ptr = &count /* Stores the address of count in ptr */
   /* The unary operator & returns the address of a variable */
```

To get the value that is stored at the memory location in the pointer it is necessary to dereference the pointer. Dereferencing is done with the unary operator "*".

```
int total;
total = *ptr;
   /* The value in the address stored in ptr is assigned to total */
```

The best way to learn how to use pointers is by example. There are examples of the types of operations already discussed below. Pointers are a difficult topic. Don't worry if everything isn't clear yet.

Declaration and Initialization

Declaring and initializing pointers is fairly easy.

```
int main()
{
   int j;
   int k;
   int l;
   int *pt1;    /* Declares an integer pointer */
   int *pt2;    /* Declares an integer pointer */
   float values[100];
   float results[100];
   float *pt3;    /* Declares a float pointer */
   float *pt4;    /* Declares a float pointer */

   j = 1;
   k = 2;
   pt1 = &j;    /* pt1 contains the address of the variable j */
   pt2 = &k;    /* pt2 contains the address of variable k */
   pt3 = values;
      /* pt3 contains the address of the first element of values */
   pt3 = &values[0];
      /* This is the equivalent of the above statement */

   return 0;
}
```

Pointer Dereferencing/Value Assignment

Dereferencing allows manipulation of the data contained at the memory address stored in the pointer. The pointer stores a memory address. Dereferencing allows the data at that memory

address to be modified. The unary operator "*" is used to dereference.
For instance:

```
*pt1 =*pt1 + 2;
```

This adds two to the value "pointer to" by pt1. That is, this statement adds 2 to the contents of the memory address contained in the pointer pt1. So, from the main program, pt1 contains the address of j. The variable "j" was initialized to 1. The effect of the above statement is to add 2 to j.

The contents of the address contained in a pointer may be assigned to another pointer or to a variable.

```
*pt2 = *pt1;
   /* assigns the contents of the memory pointed to by pt1 */
   /* to the contents of the memory pointer to by pt2; */
k = *pt2;
   /* assigns the contents of the address pointer to by pt2 to k. */
```

Pointer Arithmetic

Part of the power of pointers comes from the ability to perform arithmetic on the pointers themselves. Pointers can be incremented, decremented and manipulated using arithmetic expressions. Recall the float pointer "pt3" and the float array "values" declared above in the main program.

```
pt3 = &values[0];
   /* The address of the first element of "values" is stored in pt3*/
pt3++;
   /* pt3 now contains the address of the second element of values */
*pt3 = 3.1415927;
   /* The second element of values now has pie (actually pi)*/
pt3 += 25;
   /* pt3 now points to the 27th element of values */
*pt3 = 2.22222;
   / The 27th element of values is now 2.22222 */

pt3 = values;
   /*pt3 points to the start of values, now */

for (ii = 0; ii < 100; ii++)
{
    *pt3++ = 37.0;    /* This sets the entire array to 37.0 */
}

pt3 = &values[0];
   /* pt3 contains the address of the first element of values */
pt4 = &results[0];
   /* pt4 contains the address of the first element of results */

for (ii=0; ii < 100; ii++)
{
```

```
    *pt4 = *pt3;
       /* The contents of the address contained in pt3 are assigned to
       the contents of the address contained in pt4 */
     pt4++;
     pt3++;
}
```

Lesson 9: Arrays

Introduction to Arrays

This lesson introduces arrays. Arrays are a data structure that is used to store a group of objects of the same type sequentially in memory. All the elements of an array must be the same data type, for example float, char, int, pointer to float, pointer to int, a structure or function. Structures provide a way to organize related data and will be studied in a later lesson. Functions provide a way to define a new operation. They are used to calculate a result or update parameters. Functions will be covered in a later lesson. The elements of an array are stored sequentially in memory. This allows convenient and powerful manipulation of array elements using pointers.

Defining Arrays

An array is defined with this syntax.

```
datatype arrayName[size];
```

Examples:

```
int ID[30];
            /* Could be used to store the ID numbers of students in a class */

float temperatures[31];
            /* Could be used to store the daily temperatures in a month */

char name[20];
            /* Could be used to store a character string.
            Character strings in C are terminated by the null character, '\0'.
            This will be discussed later in the this lesson. */

int *ptrs[10];
            /* An array holding 10 pointers to integer data */

unsigned short int[52];
            /* Holds 52 unsigned short integer values */
```

Using Arrays

Arrays in C are zero based. Suppose an array named myExample contains N elements. This array is indexed from 0 to (N-1). The first element of myExample is at index 0 and is accessed as myExample[0]. The second element is at index 1 and is accessed as myExample[1]. The last element is at index (N-1) and is accessed as myExample[N-1]. As a concrete example, suppose N equals 5 and that myExample will store integer data.

```
int myExample[5];     /* Defines myExample to be of length 5 and to contain integer data */

myExample[0]     /* First element of myExample */
myExample[1]     /* Second element of myExample */
myExample[2]     /* Third element of myExample */
myExample[3]     /* Fourth element of myExample */
myExample[4]     /* Fifth and final element of myExample */
```

Here is a sample program that calculates and stores the squares of the first one hundred positive integers.

```
#include <stdio.h>

int main()
{
    int square[100];
    int i;    /* loop index */;
    int k;    /* the integer */

    /* Calculate the squares */
    for (i = 0; i < 100; i++) {
        k = i + 1;    /* i runs from 0 to 99 */
                      /* k runs from 1 to 100 */
        square[i] = k*k;
        printf("The square of %d is %d\n",k,square[i]);
    }

    return 0;
}
```

Practice Problem 1: A Fibonacci sequence is a numerical sequence such that after the second element all numbers are equal to the sum of the previous two elements.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Write a program to calculate and print the first 20 elements of this sequence.

Solution to practice problem 1: Write a program to calculate and print the first 20 elements of this sequence.

```
#include <stdio.h>

int main()
{
    int i;
    int fib[20];

    fib[0] = 1;
    fib[1] = 1;
```

```
    for (i = 2; i < 20; i++)
    {
        fib[i] = fib[i-1] + fib[i-2];
    }

    for (i = 0; i < 20; i++)
    {
        printf("The %d element is %d\n", i+1, fib[i]);
    }

return 0;
}
```

Relationship Between Pointers and Arrays

In some cases, a pointer can be used as a convenient way to access or manipulate the data in an array. Please see Lesson 8 if you need a refresher on pointers.

Suppose the following declarations are made.

```
float temperatures[31];
    /* An array of 31 float values, the daily temperatures in a month */

float *temp;    /* A pointer to type float */
```

Since temp is a float pointer, it can hold the address of a float variable. The address of the first element of the array temperatures can be assigned to temp in two ways.

```
temp = &temperatures[0];
temp = temperatures;
    /* This is an alternate notation for the first
    element of the array. Same as temperatures = &temperatures[0]. */
```

The temperature of the first day can be assigned in two ways.

```
temperatures[0] = 29.3;    /* brrrrrrr */
*temp = 15.2;    /* BRRRRRRR */
```

Other elements can be updated via the pointer, as well.

```
temp = &temperatures[0];

*(temp + 1) = 19.0;
    /* Assigns 19.0 to the second element of temperatures */

temp = temp + 9;
    /* temp now has the address of the 10th element of the array */
```

```
*temp = 25.0;
   /* temperatures[9] = 25, remember that arrays are zero based,
   so the tenth elementis at index 9. */

temp++;    /* temp now points at the 11th element */

*temp = 40.9;    /* temperatures[10] = 40.9 */
```

Pointers are particularly useful for manipulating strings, which are stored as null terminated character arrays in C

Character Arrays

Strings are stored in C as character arrays terminated by the null character, '\0'. The array length must be at least one greater than the length of the string to allow storage of the terminator. String constants or literals are stored internally as a null character terminated character array.

Assigning a character literal to an array is done as follows.

```
char str1[] = "Hello World";
char str2[] = "Goodbye World";
```

The compiler automatically sizes the arrays correctly. For this example, str1 is of length 12, str2 is of length 14. These lengths include space for the null character that is added at the end of the string.

A character pointer can also be assigned the address of a string constant or of a character array.

```
char *lpointer = "Hello World";
   /* Assigns the address of the literal to lpointer */

char *apointer = str1;
   /* Assigns the starting address of str1 to apointer */

char *apointer = &str1[0];
   /* Assigns the starting address of str1 to apointer */
```

There is no direct means in the C language to copy one array to another, or one string to another. It must be done either with a standard library function or element wise in a loop. Let's try to copy on string to another.

```
#include <stdio.h>

int main()
{
   char str1[] = "Hello World";
   char str2[] = "Goodbye World";

   str2 = str1;
```

```
    return 0;
}
```

Can you see what's wrong with this code. As stated, there is no operation to assign one array to another in C. This code produced this compiler error.

error: '=' : cannot convert from 'char [12]' to 'char [14]'
There is no context in which this conversion is possible.

Now let's make another attempt using character pointers.

```
#include <stdio.h>

int main()
{
    char str1[] = "Hello World";
    char str2[] = "Goodbye World";
    char *cpt1;
    char *cpt2;

    cpt1 = &str1[0];
    cpt2 = &str2[0];

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);
    printf("cpt1 is %s\n",cpt1);
    printf("cpt2 is %s\n",cpt2);

    cpt2 = cpt1;

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);
    printf("cpt1 is %s\n",cpt1);
    printf("cpt2 is %s\n",cpt2);

    return 0;
}
```

Results:
str1 is Hello World
str2 is Goodbye World
cpt1 is Hello World
cpt2 is Goodbye World
str1 is Hello World
str2 is Goodbye World
cpt1 is Hello World
cpt2 is Hello World

As can be seen from the results, all that happened is that the pointer cpt2 was assigned the value of cpt1, that is, the address of str1. The contents of the array str2 were not changed. The only way to copy a string or any array in C is element by element. Here is a program that correctly copies on string to another.

```
#include <stdio.h>

int main()
{
    int i;
    char str1[] = "Hello World";
    char str2[] = "Goodbye World";

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);

    i = 0;
    while ((str2[i] = str1[i]) != '\0') {
        i++;
    }

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);

    return 0;
}
```

Results:
str1 is Hello World
str2 is Goodbye World
str1 is Hello World
str2 is Hello World

Practice Problem 2: Try reimplementing the above program using pointers in the copy loop.
Hints:
    cpt1 = &str1[0];
    cpt2 = &str2[0];
    Use these pointers in the while loop, remember to dereference.

Solution to practice problem 2: Implement the string copy program using pointers

```
#include <stdio.h>

int main()
{
    char str1[] = "Hello World";
    char str2[] = "Goodbye World";
    char *cpt1;
    char *cpt2;

    cpt1 = &str1[0];
    cpt2 = &str2[0];

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);

    while ((*cpt2 = *cpt1) != '\0') {
        cpt2++;
        cpt1++;
```

```
    }

    printf("str1 is %s\n",str1);
    printf("str2 is %s\n",str2);

    return 0;
}
```

Multidimensional Arrays
The C language also allows multidimensional arrays. They are defined as follows.

```
float temperatures[12][31];    /* Used to store temperature data for a year */

float altitude[100][100];    /* Used to store the altitudes of 10000 grid points
            of a 100 by 100 mile square */
```

A common way to access the elements of a multidimensional arrays is with nested for loops.

```
#define MAXI 50
#define MAXJ 75

int i;
int j;
float values[MAXI][MAXJ];

for (i = 0; i < MAXI; i++) {
    for (j = 0; j < MAXJ; j++) {
        values[i][j] = whatever;
    }
}
```

Important Note About Array Dimensions

The C language performs no error checking on array bounds. If you define an array with 50 elements and you attempt to access element 50 (the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task alone to check that all attempts to access or write to arrays are done only at valid array indexes. Writing or reading past the end of arrays is a common programming bug and can be hard to isolate.

What will happen if a program accesses past the end of an array? Suppose a program has the following code.

```
int val;

int buffer[10];

val = buffer[10];
    /* Bug, remember that the indexes of buffer run from 0 to 9. */
```

What value will be in val? Whatever happens to be in memory at the location right after the end of the array. This value could be anything. Worse yet, the program may continue to run with the incorrect value and no warnings are issued.

What will happen if a program writes past the end of an array? Suppose a program has the following code.

```
int buffer[10];

buffer[593] = 99;
```

The value of 99 will be written at the memory location, buffer + 593. "buffer" is a pointer to the beginning of the array. buffer + 593 is pointer arithmetic for the address equal to the starting address of the array plus the size of 593 integers. The overwriting of the value at this memory location will change the value of whatever variable is stored there. Some other variable may have its value changed unintentionally. If the program writes unintentionally to memory locations that not valid, the program may crash.

The most common cause of writing/reading to invalid array indexes are errors in loop limits.

```
int i;
float b[10];

for (i < 0 ; i <= 10; i++) {
    b[i] = 3.14 * i * i;
}
```

This loop should use "<" rather than "<="

Lesson 10: Strings

This lesson covers the use of strings in C. It begins by presenting some basic ways to analyze and manipulate string data and then presents some C library functions that are useful for string manipulation. Character data is stored as the intrinsic data type char. String data is stored as a

null character terminated character array. You may wish to review the lessons on input and output and arrays before studying this lesson.

Strings

Stings in C are stored as null character, '\0', terminated character arrays. This means that the length of a string is the number of characters it contains plus one to store the null character. Common string operations include finding lengths, copying, searching, replacing and counting the occurrences of specific characters and words. Here is a simple way to determine the length of a string.

```
#include <stdio.h>

int main()
{
    char sentence[] = "Hello World";
    int count = 0;
    int i;

    for (i = 0; sentence[i] != '\0'; i++)
    {
        count++;
    }
    printf("The string %s has %d characters ",
        sentence,count);
    printf("and is stored in %d bytes\n",
        count+1);

    return 0;
}
```

Each character within the array sentence is compared with the null character terminator until the end of the string is encountered. This technique can be generalized to search for any character.

As an example, here is a program that counts the occurrences of each lowercase letter of the alphabet in a string.

```
#include <stdio.h>

int main()
{
    int i,j;
    char buffer[120];    /* Holds input strings */
    char *pt;    /* A pointer to data type char */
    char alphabet[27] = "abcdefghijklmnopqrstuvwxyz";
    int alphaCount[26];    /* an array of counts */

    /* Initialize counts */
    for (i = 0; i < 26; i++)
    {
        alphaCount[i] = 0;
    }

    while ((pt = gets(buffer)) != NULL)
```

```
    {
        for (i = 0; i < 120 && buffer[i] != '\0'; i++)
        {
            for (j = 0; j < 26; j++)
            {
                if (buffer[i] == alphabet[j])
                {
                    alphaCount[j]++;
                }
            }
        }
    }

    for (i = 0; i < 26; i++)
    {
        printf("Found %d occurrences of %c\n",
            alphaCount[i],alphabet[i]);
    }

    return 0;
}
```

gets reads a line of input into a character array. Its prototype is:
char *gets(char *buffer);

It returns a pointer to the character string if successful, or NULL if end of file is reached or if an error has occurred. The string is also read into the character array specified as an argument. The character string will be terminated be a "\0", which is standard for C.

The first step in this program is to initialize the count array. When you use an automatic variable, you must initialize it. Otherwise, it contains whatever data happened to be at the memory location it is stored in. This will be discussed further in the lesson on variable scope.

The while loop in this program reads lines of input until an end of file is encountered. If you run this program, remember that an end of file, EOF, is entered as a control-d or a control-z from you keyboard. This is, after you type the last line of text, type control-z or control-d to enter the End of File character.

For each line read in, each character in the line is compared against each letter in the alphabet. If a match is found the appropriate count is incremented. The last step in this program writes out the results.

Practice:
1) Type in the above program. Compile and run.
2) Modify the above program to count the occurrences of the digits 0-9, rather than letters.

```
#include <stdio.h>

int main()
{
    int i,j;
    char buffer[120];    /* Holds input strings */
    char *pt;    /* A pointer to data type char */
    char digits[11] = "0123456789";
```

```
    int digitCount[10];    /* an array of counts */

    /*Initialize counts */
    for (i = 0; i < 10; i++)
    {
        digitCount[i] = 0;
    }

    while ((pt = gets(buffer)) != NULL)
    {
        for (i = 0; i < 120 && buffer[i] != '\0'; i++)
        {
            for (j = 0; j < 10; j++)
            {
                if (buffer[i] == digits[j])
                {
                    digitCount[j]++;
                }
            }
        }
    }

    for (i = 0; i < 10; i++)
    {
        printf("Found %d occurences of %c\n",
            digitCount[i],digits[i]);
    }

    return 0;
}
```

Library Functions for String Manipulation
To use any of these functions in your code, the header file "strings.h" must be included. It is the programmer's responsibility to check that the destination array is large enough to hold either what is copied or appended to it. C performs no error checking in this respect. See the array tutorial for more details. The data type size_t is equivalent to unsigned integer.

strcpy
strcpy copies a string, including the null character terminator from the source string to the destination. This function returns a pointer to the destination string, or a NULL pointer on error. Its prototype is:

```
char *strcpy(char *dst, const char *src);
```

strncpy
strncpy is similar to strcpy, but it allows the number of characters to be copied to be specified. If the source is shorter than the destination, than the destination is padded with null characters up to the length specified. This function returns a pointer to the destination string, or a NULL pointer on error. Its prototype is:

```
char *strncpy(char *dst, const char *src, size_t len);
```

strcat
This function appends a source string to the end of a destination string. This function returns a pointer to the destination string, or a NULL pointer on error. Its prototype is:

char *strcat(char *dst, const char *src);

strncat

This function appends at most N characters from the source string to the end of the destination string. This function returns a pointer to the destination string, or a NULL pointer on error. Its prototype is:

char *strncat(char *dst, const char *src, size_t N);

strcmp

This function compares two strings. If the first string is greater than the second, it returns a number greater than zero. If the second string is greater, it returns a number less than zero. If the strings are equal, it returns 0. Its prototype is:

int strcmp(const char *first, const char *second);

strncmp

This function compares the first N characters of each string. If the first string is greater than the second, it returns a number greater than zero. If the second string is greater, it returns a number less than zero. If the strings are equal, it returns 0. Its prototype is:

int strncmp(const char *first, const char *second, size_t N);

strlen

This function returns the length of a string, not counting the null character at the end. That is, it returns the character count of the string, without the terminator. Its prototype is:

size_t strlen(const char *str);

memset

memset is useful to initialize a string to all nulls, or to any character. It sets the first N positions of the sting to the value passed. The destination may be specified by a pointer to any type, char, int, float, etc. A void pointer is used to allow different types to be specified. Its prototype is:

void *memset(const void *dst,int c, site_t N);

Here is a simple program to illustrate the use of some of these functions.

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

int main()
{
   char str1[] = "Hello";
   char str2[] = "World";
   char str3[20];
   char *cpt;
   int length;
   int rc;

   /* Find length of str1 */
   length = strlen(str1);
   printf("String 1, %s, is of length %d\n",str1,length);

   /* Copy str1 to str3, print both */
```

```
    printf("\nCopying str1 to str3.\n");
    if ((cpt = strcpy(str3,str1)) == NULL)
    {
        printf("Error on strcpy");
        return 1;
    }
    printf("String 1 is %s\n",str1);
    printf("String 3 is %s\n",str3);

    /* Clear str3 */
    memset(str3,'\0',20);
    printf("\nstr3 is cleared\n");

    /* Copy first 2 characters of str1 to str3 */
    printf("Copying the first two characters of str1 to str3\n");
    strncpy(str3,str1,2);
    printf("String 1 is %s\n",str1);
    printf("String 3 is %s\n",str3);

    /* Compare the first 2 characters of str1, str3 */
    printf("\nComparing the first two characters of str1 and str3\n");
    if ((rc = strncmp(str1,str3,2)) == 0)
    {
        printf("First two characters of str1 and str3 match\n");
    }
    else
    {
        printf("First two characters of str1 and str3 don't match\n");
    }

    /* Compare all characters of str1 and str3 */
    printf("\nComparing all characters of str1 and str3\n");
    rc = strcmp(str1,str3);
    if (rc == 0)
    {
        printf("str1 equals str3\n");
    }
    else if (rc > 0)
    {
        printf("str1 is greater\n");
    }
    else /* rc < 0 */
    {
        printf("str3 is greater\n");
    }

    /* Append to "he" in str3 */
    printf("\nAppending to str3\n");
    str3[2] = 'y';
    str3[3] = ' ';
    strcat(str3,str2);
    printf("%s\n",str3);

    return 0;
}
```

In this example error checking on the library calls was not done in order to simplify the presentation. As an example, only the first call was checked.

Lesson 11: Structures

A structure provides a means of grouping variables under a single name for easier handling and identification. Complex hierarchies can be created by nesting structures. Structures may be copied to and assigned. They are also useful in passing groups of logically related data into functions.

Declaring Structures

A structure is declared by using the keyword struct followed by an optional structure tag followed by the body of the structure. The variables or members of the structure are declared within the body. Here is an example of a structure that would be useful in representing the Cartesian coordinates of a point on a computer screen, that is, the pixel position.

```
struct point {
    int x;
    int y;
};
```

The struct declaration is a user defined data type. Variables of type point may be declared similarly to the way variables of a built in type are declared.

```
struct point {
    int x;
    int y;
} upperLeft;
```

is analogous to

```
float rate;
```

The structure tag provides a shorthand way of declaring structures.

```
struct point {
    int x;
    int y;
};

struct point left,right;
struct point origin;
```

The C language allows data types to be named using the keyword typedef. For example:

```
typedef double Money;
typedef unsigned long int ulong;

Money paycheck;
    /* This declares paycheck to be of type Money, or double */
ulong IDnumber;
    /* This declared IDnumber to be of type ulong, or unsigned long */
```

User defined data types such as struct may also be named using typedef.

```
typedef struct point {
    int x;
    int y;
} Dot;

Dot left,right;
    /* Declares left and right to be Dots, or structures of type point */
```

The examples in the next section will present the different ways of declaring structures.

Using Structures

The individual members of a structure can be accessed using ".", the member access operator. Given

```
typedef struct point {
    int x;
    int y;
} Dot;
....
Dot location;
```

The x coordinate is location.x
The y coordinate is location.y

It is also possible to create hierarchies by nesting structures. A rectangle could be represented as follows.

```
typedef struct rect {
    Dot upperLeft;
    Dot upperRight;
    Dot lowerLeft;
    Dot lowerRight;
} Box;

Box myRectangle;
```

Pointers to structures may also be declared and assigned. Given the above typedef of Dot, a pointer may be declared and initialized as follows.

```
Dot location;    /* Defines an object of type Dot */
Dot *lpt;    /* Declares a pointer of type Dot */
....
lpt = &location;
    /* Assigns the address of location to the pointer, pt */
```

A special member access operator, "->", is used to access the members of a structure via a pointer.

The x coordinate of location is given by lpt->x
The y coordinate of location is given by lpt->y

Here is a sample program illustrating the use of structures.

```
#include <stdio.h>

typedef struct point {
    int x;
    int y;
} DOT;

struct rect {
    DOT ul;
    DOT ur;
    DOT ll;
    DOT lr;
};

int main()
{

    DOT location;    /* One way to declare structures */
    DOT *lpt;

    struct rect myRect;
         /* Second way to declare structures */

    struct circle {    /* Third way */
        DOT origin;
        int radius;
    } myCircle;

    location.x = 5;
    location.y = 4;
```

```
    lpt = &location;

    printf("Starting point %d %d\n",location.x, lpt->y);

    return 0;
}
```

Arrays of Structures

The data structures needed to solve some problems are best represented as an array of structures. Consider, for instance, the problem of counting the occurrence of each letter in a string. One possible approach would be to declare two separate arrays. One would hold the letters to compare against. The second would hold a count of the occurrences of each letter. Their declarations would be:

```
#define NUMLETTERS 26
....
char letter[NUMLETTERS];
int count[NUMLETTERS];
```

This approach would work, but each letter and its count are closely related. It would be convenient and less error prone to keep this closely linked data together in a structure.

```
#define NUMLETTERS 26
....
/* Declares a structure with a structure tag lettertype */
struct lettertype {
    char letter;
    int count;
};

typedef struct lettertype Letter;
    /* Declares the structure to be a new data type, Letter */

Letter alphabet[NUMLETTERS];
    /* Defines an array, alphabet, of type letter */
```

Note that the above declarations could also be written as follows.

```
typedef struct lettertype {
    char letter;
    int count;
} Letter;

Letter alphabet[NUMLETTERS];
```

Here is a program using the Letter data structure to count letters in a string.

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

```
#include <string.h>

#define NUMLETTERS 26

typedef struct lettertype {
    char letter;
    int count;
} Letter;

int main()
{
    /* Variable Declarations */
    int i,j;
    int length;
    Letter alphabet[] = {
        'a',0,
        'b',0,
        'c',0,
        'd',0,
        'e',0,
        'f',0,
        'g',0,
        'h',0,
        'i',0,
        'j',0,
        'k',0,
        'l',0,
        'm',0,
        'n',0,
        'o',0,
        'p',0,
        'q',0,
        'r',0,
        's',0,
        't',0,
        'u',0,
        'v',0,
        'w',0,
        'x',0,
        'y',0,
        'z',0
    };
    char sampleString[] =
        "now is the time for all good men to come to the aid of their country";


    /* Find length of string */
    length = strlen(sampleString);

    for (i = 0; i < length; i++)
    {
        for (j = 0; j < NUMLETTERS; j++)
        {
            if (sampleString[i] == alphabet[j].letter)
            {
                alphabet[j].count++;
            }
        }
```

```
    }

    for (j = 0; j < NUMLETTERS; j++)
    {
        printf("%c found %d times\n",alphabet[j].letter,
                alphabet[j].count);
    }

    return 0;
}
```

Practice:
Try compiling the above example. Experiment with the different ways to declare structures described at the beginning of this lesson. Extend it to count upper and lower case letters.

Lesson 12: Memory Allocation

If my recollection is correct, this lesson covers the use of memory allocation. :-). Memory allocation provides a way to dynamically create buffers and arrays. Dynamic means that the space is allocated in memory as the program is executing. On many occasions the sizes of objects will not be known until run time. For instance, the length of a string a user inputs will not be known prior to execution. The size of an array may depend on parameters unknown until program execution. Certain data structures such as linked lists utilize dynamic memory allocation.

How to Allocate and Free Memory

The standard library function malloc is commonly used to allocate memory. Its prototype is:

```
void *malloc(size_t nbytes);
```

Malloc returns a void pointer to the allocated buffer. This pointer must be cast into the proper type to access the data to be stored in the buffer. On failure, malloc returns a null pointer. The return from malloc should be tested for error as shown below.

```
char *cpt;
...
if ((cpt = (char *) malloc(25)) == NULL)
{
    printf("Error on malloc\n");
}
```

The data type size_t is equivalent to unsigned integer. The number of bytes of memory needed depends on the number of objects to be stored and the size of each object. Char data is stored in one byte. Float data is stored in four bytes. On most 32-bit machines, ints are stored in four

bytes. The easiest way to determine the size of the object to be stored is to use the sizeof operator, which returns the size in bytes of an object. Sizeof can be called with either an object or a data type as its argument. Here is a simple program illustrating its use.

```
#include <stdio.h>

typedef struct employee_st {
    char name[40];
    int id;
} Employee;

int main()
{
    int myInt;
    Employee john;

    printf("Size of int is %d\n",sizeof(myInt));
        /* The argument of sizeof is an object */
    printf("Size of int is %d\n",sizeof(int));
        /* The argument of sizeof is a data type */

    printf("Size of Employee is %d\n",sizeof(Employee));
        /* The argument of sizeof is an object */
    printf("Size of john is %d\n",sizeof(john));
        /* The argument of sizeof is a data type */

    printf("Size of char is %d\n",sizeof(char));
    printf("Size of short is %d\n",sizeof(short));
    printf("Size of int is %d\n",sizeof(int));
    printf("Size of long is %d\n",sizeof(long));
    printf("Size of float is %d\n",sizeof(float));
    printf("Size of double is %d\n",sizeof(double));

    return 0;

}
```

When using dynamically allocated memory, it is necessary for the programmer to free the memory after its use. This is done, surprisingly enough, using the standard library function free. Its prototype is:

```
void free(void *pt);
```

It is not necessary to cast the pointer argument back to void. The compiler handles this conversion.

Example of Use
One common use of malloc is to allocate a buffer to hold string data. Here is a somewhat contrived example of this.

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

int main()
```

```
{

    char s1[] = "This is a sentence";
    char *s2;

    s2 = (char *) malloc(strlen(s1) + 1);
        /* Remember that stings are terminated by the null
        terminator, "\0', and the strlen returns the length of
        a string not including the terminator */
    if (s2 == NULL)
    {
        printf("Error on malloc");
        return 1;
        /* Use a nonzero return to indicate an error has occurred */
    }

    strcpy(s2,s1);

    printf("s1: %s\n", s1);
    printf("s2: %s\n", s2);

    return 0;
}
```

Practice: Write a program that dynamically allocates an array of type Employee. Prompt the user for the number of Employee's to create. Hire a few employees (fill in their names and id numbers).

The data type Employee is defined as:

```
typedef struct employee_st {
    char name[40];
    int id;
} Employee;
```

Solution

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

typedef struct employee_st {
    char name[40];
    int id;
} Employee;

int main()
{

    Employee *workers, *wpt;
    int num;

    printf("How many employees do you want\n");
    scanf("%d",&num);
```

```
    workers = (Employee *) malloc(num * sizeof(Employee));
    if (workers == NULL)
    {
        printf("Unable to allocated space for employees\n");
        return 1;
        /* A nonzero return is usually used to indicate an error */
    }

    wpt = workers;

    strcpy(wpt->name,"John");
    wpt->id = 12345;

    wpt++;
    strcpy(wpt->name,"Justin");
    wpt->id = 12346;

    wpt = workers;
    printf("Employee %d is %s\n", wpt->id, wpt->name);
    wpt++;
    printf("Employee %d is %s\n", wpt->id, wpt->name);

    free(workers);
    workers = NULL;

    return 0;
}
```

Common Errors

There are several common errors to be wary of when using dynamic memory allocation. The first is neglecting to check if the malloc was successful. If unable to allocate memory, malloc returns a null pointer. If this occurs, when the pointer is accessed later in the program, most likely, the program will crash.

```
int *pt;

pt = (int *) malloc(500 * sizeof(int));
...
...
*pt = 4002;    /* Program dies here */
```

If the return from malloc were checked it would be possible to either gracefully exit the program, or perhaps skip the section of code that used this buffer.

Another common error is neglecting to free memory after use. There is a limit to the amount of memory available for use by a program. If a long running program neglects to free memory, but continues to allocate the memory available to it will run out. This error is called a memory leak. It is a very serious bug. As the errant program consumes more and more memory, other processes may also be deprived of memory they require. It can impact the performance of the entire system.

A third common error is referred to as a dangling pointer. After freeing memory, the pointer passed to free still contains the memory address that was the start of the buffer being deallocated. If this pointer is used later in the program, whatever values happen to be in that

area of memory may be used. This could happen if the programmer incorrectly believes that some other code that runs after the "free" resets the pointer to some valid address. The best way to avoid this bug is to immediately set the pointer to NULL after the free.

```
free(pt);
pt = NULL;
```

Lesson 13: File I/O and Command Line Arguments

This lesson covers reading and writing to files and passing command line arguments into your code. An important part of any program is the ability to communicate with the world external to it. Reading input from files and writing results to files are simple, but effective ways to achieve that. Command line arguments provide a way to pass a set of arguments into a program at run time. These arguments could be the names of files on which to operate, user options to modify program behavior, or data for the program to process.

Library Functions for File I/O.
This section introduces the library functions used for file input and output (I/O). These routines are used to open and close files and read and write files using formatted I/O. Formatted I/O was introduced in lesson 4. You may wish to review that material.

Fopen is used to open a file for formatted I/O and to associate a stream with that file. A stream is a source or destination of data. It may be a buffer in memory, a file or some hardware device such as a port. The prototype for fopen is:

```
FILE *fopen(const char *filename, const char *mode);
```

Fopen returns a file pointer on success or NULL on failure. The file pointer is used to identify the stream and is passed as an argument to the routines that read, write or manipulate the file. The filename and mode arguments are standard null-terminated strings. The valid modes are shown below.

Mode	Use
r	open for reading
w	open or create for writing. Truncate (discard) any previous contents.
a	open or create for writing. Append (write after) any previous contents.
r+	open file for update (reading and writing).
w+	open or create file for update. Truncate (discard) any previous data.
a+	open or create file for update. Append (write after) any previous data.

Fflush is used to flush any buffered data out of an output stream. Output streams may be buffered or unbuffered. With buffered output, as data is written to the stream, the operating

system saves this data to an intermediate buffer. When this buffer is full, the data is then written to the file. This is done to reduce the number of system calls needed to write out the data. The whole buffer is written at once. This is done to make the program more efficient and without any programmer involvement. Fflush forces the buffer to be written out to the associated file. Its prototype is:

int fflush(FILE *stream);

It accepts a file pointer as an argument. It returns zero on success and EOF on failure. EOF is a special constant used to designate the end of a file.

Files are closed with the function fclose. Its prototype is:

int fclose(FILE *stream);

Fclose returns zero on success and EOF on failure.

Data is written to a file using fprintf. This function is very similar to printf, which is described fully in lesson 4. Printf was used to write to standard output, stdout. Fprintf has one additional argument to specify the stream to send data. Its prototype is:

int fprintf(FILE *stream, const char* format, ...);

Fprintf returns the number of characters written if successful or a negative number on failure.

Data is read from a file using fscanf. This function is very similar to scanf, which was described in lesson 4 and is used to read from standard input, stdin. Fscanf has one additional argument to specify the stream to read from. Remember that the argument to store data must be pointers. The prototype for fscanf is:

int fscanf(FILE *stream, const char* format, ...);

Other Useful Standard Library Functions for Input and Output

int fgetc(FILE *stream);
This function returns the next character in the input stream, or EOF if the end of the file is encountered. It returns int rather than char because the "end of file", EOF, character must be handled. EOF is an int and is too large to fit in a char variable.

int *fgets(char *s, int n, FILE *stream);
It returns a pointer to the character string if successful, or NULL if end of file is reached or if an error has occurred. The string is also read into the character array specified as an argument. The character string will be terminated be a "\0", which is standard for C.
At most n-1 characters will be read. This allows for storage of the string terminator, '\0'.

int fputs(const char s*, FILE *stream);
This function writes a string to the stream. It returns EOF on failure.

int fputc(int c, FILE *stream);
Fputc writes a single character to the output stream. It returns EOF on failure.

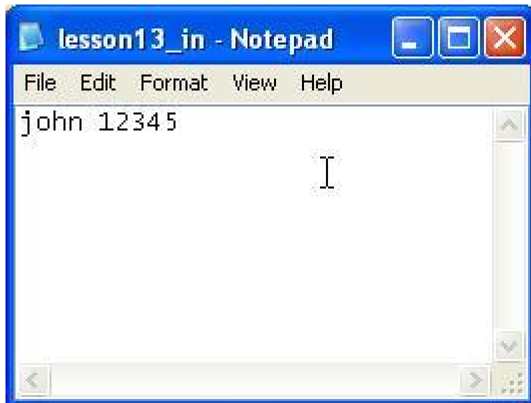
int sprintf(char *buffer, const char *format, ...);
This function is the same as fprintf, except data is written into a character buffer rather than an output stream.

int sscanf(char *buffer, const char *format, ...);

This function is the same as fscanf, except data is read from a character buffer rather than an input stream.

Example

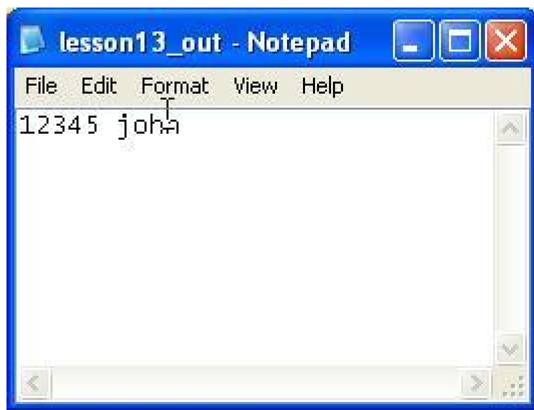
The following program opens a file containing a name and an ID number. It then opens an output file and writes this data to an output file. This input file was created with a text editor.



Here is the example program.

```
#include <stdio.h>

int main()
{
    char ifilename[] = "c:/lesson13_in.txt";
    char ofilename[] = "c:/lesson13_out.txt";
    char name[30];
    int idNum;
    FILE *ofp, *ifp;

    /* Open file for input */
    ifp = fopen(ifilename,"r");

    /* Read data */
    fscanf(ifp,"%s %d",name,&idNum);

    /* Open file for output */
    ofp = fopen(ofilename,"w");

    /* Write out data */
    fprintf(ofp,"%d %s\n",idNum, name);

    /* Close Files */
    fclose(ifp);
    fclose(ofp);

    return 0;
}
```

Here is the output.



Practice Problem
Extend the above example so it will read and write multiple pairs of names and ids. Hint: Create a loop and check the return code of fscanf for EOF.

```
#include <stdio.h>

int main()
{
    char ifilename[] = "c:/lesson13_in.txt";
    char ofilename[] = "c:/lesson13_out.txt";
    char name[30];
    int idNum;
    FILE *ofp, *ifp;

    /* Open file for input */
    ifp = fopen(ifilename,"r");

    /* Open file for output */
    ofp = fopen(ofilename,"w");

    /* Read and write data */
    while (fscanf(ifp,"%s %d",name,&idNum) != EOF)
    {
    /* Write out data */
    fprintf(ofp,"%d %s\n",idNum, name);
    }

    /* Close Files */
    fclose(ifp);
    fclose(ofp);

    return 0;
}
```

stderr, stdin and stdout
Stderr and stdout are predefined streams available for output. Both are usually defined to be the computer's screen. Stdout is a buffered stream; stderr is unbuffered. Stdin is a predefined input stream, usually defined to be the keyboard. In previous lessons, output was written to stdout using the printf function.

```
printf("Hello World\n"); /* Buffered output */
```

Output may also be directed to stdout or stderr using the fprintf function.

```
fprintf(stdout,"Hello World\n"); /* Buffered output, same as printf() */
fprintf(stderr,"Hello World\n"); /* Unbuffered output */
```

Buffered vs. unbuffered output has important implications for error messages and prints used for debugging. If a program dies, core dumps, while running, buffers are not flushed. That means error messages sent to stdout may not be seen. Since stderr is unbuffered, all error messages are immediately output. Thus, error messages and debugging statements should be directed to stderr rather than stdout.

Stdin is usually the keyboard. In previous lessons, input was read from stdin using scanf().

```
scanf("%d",&someInt);
```

This may also be done using fscanf as follows.

```
fscanf(stdin,"%d",&someInt);
```

Command Line Arguments

C provides a mechanism to pass command line arguments into a program when it begins to execute. When execution begins, "main" is called with two arguments, a count and a pointer to an array of character strings. The count is by convention called argc. The pointer is usually called argv. The use of argv is somewhat tricky. Since argv is a pointer to an array of character strings, the first character string is referenced by argv[0] (or by *argv). The second character string is referenced by argv[1] (or by *(argv + 1)), the third by argv[2], and so on. The first character string, argv[0], contains the program name. Command line arguments begin with argv[1]. If the number of arguments will be fixed, the count, argc, should always be checked. Here is a simple example program that performs an echo. Note that the number of strings to echo need not be hard coded. Argc is used to determine the number of strings passed as command line arguments.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    fprintf(stdout,
```

```
        "The number of command line arguments is %d\n",
        argc);
    fprintf(stdout,"The program name is %s\n",argv[0]);

    for (i = 1; i < argc; i++)
    {
        fprintf(stdout,"%s",argv[i]);
    }
    fprintf(stdout,"\n");
    return 0;
}
```

Practice: Try compiling and running the above program.

Lesson 14: Functions

This lesson covers functions. Functions are used to encapsulate a set of operations and return information to the main program or calling routine. Encapsulation is detail, information or data hiding. Once a function is written, we need only be concerned with what the function does. That is, what data it requires and what outputs it produces. The details, "how" the function works, need not be known.

The use of functions provides several benefits. First, it makes programs significantly easier to understand and maintain. The main program can consist of a series of function calls rather than countless lines of code. A second benefit is that well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions. A third benefit of using functions is that different programmers working on one large project can divide the workload by writing different functions.

Defining and Declaring Functions

A function is declared with a prototype. The function prototype, which has been seen in previous lessons, consists of the return type, a function name and a parameter list. The function prototype is also called the function declaration. Here are some examples of prototypes.

```
return_type function_name(list of parameters);

int max(int n1, int n2);    /* A programmer-defined function */
int printf(const char *format,...);    /* From the standard library */
int fputs(const char *buff, File *fp);    /* From the standard library */
```

The function definition consist of the prototype and a function body, which is a block of code enclosed in parenthesis. A declaration or prototype is a way of telling the compiler the data types of the any return value and of any parameters, so it can perform error checking. The definition creates the actual function in memory. Here are some examples of functions.

```
int FindMax(int n1, int n2)
{
    if (n1 > n2)
    {
        return n1;
    }
    else
    {
```

```
        return n2;
    }
}

void PrintMax(int someNumber)
{
    printf("The max is %d\n",someNumber);
}

void PrintHW()
{
    printf("Hello World\n");
}

float FtoC(float faren)
{
    float factor = 5./9.;
    float freezing = 32.0;
    float celsius;

    celsius = factor * (faren - freezing);

    return celsius;
}
```

There are a few significant things to notice in these examples. The parameter list of a function may have parameters of any data type and may have from no to many parameters. The return statement can be used to return a single value from a function. The return statement is optional. For instance, the function PrintHW does not return a value. Techniques for returning multiple values from a function will be covered later in the lesson. Finally, observe that variables can be declared within a function. These variables are local variables. They have local scope. Scope refers to the section of code where a variable name is valid and may be used.

Using Functions

A function should always be declared prior to its use to allow the compiler to perform type checking on the arguments used in its call. Depending on the compiler and compiler options used, it is possible to not declare functions prior to use but then error checking on the arguments in the function call will not be performed. Let's use the functions defined in the previous section in a simple program.

```
/* Include Files */
#include <stdio.h>

/* Function Declarations */
int FindMax(int n1, int n2);
void PrintMax(int someNumber);
void PrintHW();
float FtoC(float faren);

int main()
{
    int i = 5;
    int j = 7;
```

```
    int k;
    float tempInF = 85.0;    /* A nice sunny day */
    float tempInC;

    PrintHW();    /* Prints Hello World */

    k = FindMax(i,j);
    PrintMax(k);    /* Prints Max Value */

    tempInC = FtoC(tempInF);
    printf("%f Fahrenheit equals %f Celsius \n",tempInF,tempInC);

    return 0;
}

/* Function Definitions */
int FindMax(int n1, int n2)
{
    if (n1 > n2)
    {
        return n1;
    }
    else
    {
        return n2;
    }
}

void PrintMax(int someNumber)
{
    printf("The max is %d\n",someNumber);
}

void PrintHW()
{
    printf("Hello World\n");
}

float FtoC(float faren)
{
    float factor = 5./9.;
    float freezing = 32.0;
    float celsius;

    celsius = factor * (faren - freezing);

    return celsius;
}
```

Notice that the functions are declared prior to their use. The function definitions are actually below main in this file. Also, notice that the function definitions and declarations match. They have the same return type, names, and parameters. The function definitions need not even be in this file. For instance, when you use library functions, the definitions are not in your file. A

program may be separated into multiple files. Main, along with the needed function declarations may be in one file. The functions themselves, that is, their definitions, may be separated, into multiple files.

Practice Problem

1) Write functions to convert feet to inches, convert inches to centimeters, and convert centimeters to meters. Write a program that prompts a user for a measurement in feet and converts and outputs this value in meters.

Facts to use: 1 ft = 12 inches, 1 inch = 2.54 cm, 100 cm = 1 meter.

```
#include <stdio.h>

double feetToInches(double feet);
double inchesToCentimeters(double inches);
double centimetersToMeters(double cm);

int main()
{
   double feet,inches,cms,meters;

   printf("Enter your measurement in feet\n");
   scanf("%lf",&feet);

   inches = feetToInches(feet);
   cms = inchesToCentimeters(inches);
   meters = centimetersToMeters(cms);

   printf("%f feet equals %f inches\n",feet,inches);
   printf("%f feet equals %f centimeters\n",feet,cms);
   printf("%f feet equals %f meters\n",feet,meters);

   return 0;
}

double feetToInches(double feet)
{
   return 12.0 * feet;
}

double inchesToCentimeters(double inches)
{
   return 2.54 * inches;
}

double centimetersToMeters(double cm)
{
   return cm/100.0;
}
```

Returning Multiple Values From Functions

So far, all the examples shown have either not returned any values, or have returned a single value using the return statement. In practice, it will be common to need to return multiple values from a function. Let's see how this can be done by developing a function to swap two integer values. Here's a first attempt.

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

```
void swap(int x, int y);
        /* Note that the variable names in the
           prototype and function
           definition need not match.
           Only the types and number of
           variables must match */

int main()
{
    int x = 4;
    int y = 2;

    printf("Before swap, x is %d, y is %d\n",x,y);
    swap(x,y);
    printf("After swap, x is %d, y is %d\n",x,y);

}

void swap(int first, int second)
{
    int temp;

    temp = second;
    second = first;
    first = temp;
}
```

What happened? The values weren't swapped. The function didn't work as expected. In C, all arguments are passed into functions by value. This means that the function receives a local copy of the argument. Any modifications to the local copy do not change the original variable in the calling program.

If a variable is to be modified within a function, and the modified value is desired in the calling routine, a pointer to the variable should be passed to the function. The pointer can then be manipulated to change the value of the variable in the calling routine. It is interesting to note that the pointer itself is passed by value. The function cannot change the pointer itself since it gets a local copy of the pointer. However, the function can change the contents of memory, the variable, to which the pointer refers. The advantages of passing by pointer are that any changes to variables will be passed back to the calling routine and that multiple variables can be changed. Here is the same example with pointers being passed into the function.

```
#include <stdio.h>

void swap(int *x, int *y);

int main()
{
    int x = 4;
    int y = 2;
```

```
    printf("Before swap, x is %d, y is %d\n",x,y);
    swap(&x,&y);
    printf("After swap, x is %d, y is %d\n",x,y);

    return 0;

}

void swap(int *first, int *second)
{
    int temp;

    temp = *second;
    *second = *first;
    *first = temp;
}
```

Practice Problem

1) Write a function that will calculate the area and circumference of a circle. Write a program to prompt a user for a radius and write out the values calculated by the function. Hint: Pass values that will be modified by pointer.

Useful facts:

pi = 3.14

area = pi * $radius^2$

circumference = 2 * pi * radius

```
#include <stdio.h>
#define PI 3.1415

void calcCircle(float radius, float *area, float *circum);

int main()
{

    float radius;
    float area;
    float circum;
    float *apt;

    printf("Enter the radius of your circle: ");
    scanf("%f",&radius);

    apt = &area;
    calcCircle(radius,apt,&circum);
    /* Notice that for the area an explicit pointer is passed.
       For the circumference, the address of operator is used.
       Passing in the address of a variable is the same as passing
       a pointer */

    printf("A circle of radius %f\n \t has area %f\n \t and circumference %f\n",
           radius, area, circum);

    return 0;

}
```

```
void calcCircle(float radius, float *area, float *circum)
{
    float pi = PI;

    *area = pi * radius * radius;
    *circum = 2.0 * pi * radius;
}
```

Passing Arrays to Functions

When an array is passed into a function, the function receives not a copy the array, but instead the address of the first element of the array. The function receives a pointer to the start of the array. Any modifications made via this pointer will be seen in the calling program. Let's see how this works. Suppose the main program has an array of 10 integers and that a function has been written to double each of these.

```
void doubleThem(int a[], int size);
int main()
{
    int myInts[10] = {1,2,3,4,5,6,7,8,9,10};

    doubleThem(myInts, 10);

    return 0;
}

void doubleThem(int a[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        a[i] = 2 * a[i];
    }
}
```

Note that due to relationship between pointers and arrays in C, this function could also be written as follows and the rest of the program could be used without modification.

```
void doubleThem(int *pt, int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        *pt = 2 * *pt;
        pt++;
    }
}
```

Strings are stored in C as null terminated character arrays. To conclude this lesson, let's see

how to pass strings into and out of functions by implementing a string copy function. Note that a string copy function, strcpy, is part of the C standard library.

```
#include <stdio.h>
void strcpy(char *dest, char *source);

int main()
{
    char a[100] = "Hello From About C/C++";
    char b[100];

    strcpy(b,a);
    printf("%s\n",b);
}

void strcpy(char d[], char s[])
{
    int i = 0;
    for (i = 0; s[i] != '\0'; i++)
    {
        d[i] = s[i];
    }
    d[i] = s[i]; /* Copy null terminator to dest */
}
```

Practice Problem
1) Try compiling and running this example.
2) Try rewriting the strcpy function using pointer notation. Compile and run.

```
void strcpy(char *d, char *s)
{
    while (*s != '\0')
    {
        *d = *s;
        d++;
        s++;
    }
    *d = *s; /* Copy null terminator to dest */
}
```

Here is a shorter solution.

```
void strcpy(char *d, char *s)
{
    while ((*d++ = *s++) != '\0');
}
```

Lesson 15: Scope and Program Structure

This is the final lesson. It covers several topics that are relevant to creating real C programs to solve real problems: variable scope, static variables and program structure.

Local Variables

The scope of a variable is simply the part of the program where it may be accessed or written. It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,

```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```

Here, the local variable "a" in fun1 is distinct from the local variable "a" in fun2. Changes made to "a" in one function have no effect on the "a" in the other function. Also, note that "b" exists and can be used only in fun1. "C" exists and can be used only in fun2. The scope of b is fun1. The scope of c is fun2. Note that main is also a function. Variables declared after the opening bracket of main will have all of main as their scope.

```
int fun1();
int fun2();

int main()
{
   int a;
   int b;
   int x,y,z;

   x = fun1();
   y = fun2();

   return 0;
}

int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
}
```

So here, a, b, x, y and z are local to main, a and b are local to fun1 and a and c are local to fun2. Notice that in this example there are three distinct local variables all named "a". Local variables are also referred to as automatic variables. They come to life at the beginning of a function and die at the end automatically.

External Variables

Variables may also be defined outside of any function. These are referred to as global or external variables. The scope of an external variable is from its declaration to the end of the file. For instance:

```
int j;
...
int main()
{
   ....
}

int k;
float funA()
{
}

int l;
float funB()
{
}
```

The variable "j" will be visible in main, funA and funB. The variable "k" will be visible in funA and funB only. The variable "l" will be visible only in function funB.

An important distinction between automatic (local) variables and external (global) variables is how they are initialized. External variables are initialized to zero. Automatic variables are undefined. They will have whatever random value happens to be at their memory location. Automatic, or local, variables must always be initialized before use. It is a serious error, a bug, to use a local variable without initialization.

The scope for a function is similar to that of an external variable. Its scope is from the functions declaration to the end of the file.

```
int fun1();
int fun2();
int main()
{
   ....
}
int fun3();

int fun1()
{
   int i;
   ....
   i = fun3();
   ....
}
```

```
int fun2()
{
    .....
}
int fun3()
{
    .....
}
```

Notice that here fun1 and fun2 may be called from main but fun3 may not. The function fun3 can be called from fun1 and fun2 or from anywhere after its declaration.

Static Variables

The keyword "static" has two different uses, depending on whether it is applied to an external variable or function or to an automatic variable. When applied to an external variable (global) variable the scope of that variable to the file in which it is declared. This is useful to hide buffers and variables that are used only by functions in a particular file. Likewise, when used with a function, the scope of that function is limited to the file in which it is defined. For instance, if several input related routines in a file required a buffer to store data, then

```
static char inputbuff[10000];
```

would declare a buffer that had its scope limited to that file.

When used with an automatic variable (local variable), the keyword static gives persistence to the variable. This is particularly useful to maintain information in a function between calls. All non-static automatic variables in a function are created when it is called and destroyed when it exists. Static variables will maintain their values between calls. This is useful if the function needs to do special initializations on its first call.

```
int example()
{
    static int first = 1;

    if (first)
    {
        first = 0;
        /* This code is executed only with the first call */
        /* Perform initializations, initialize hardware, etc. */
    }
    /* Rest of the function will be executed every time */
    ....
    ....
}
```

Multiple Files

Up to now, all the examples in this tutorial have shown an entire program within one file. This is fine for short simple programs but any large real world program is likely to have its code distributed among multiple files. There are several practical reasons for organizing a program into multiple files. First, it allows parts of the program to be developed independently, possible by different programmers. This separation also allows independent compiling and testing of the modules in each file. Second, it allows greater reuse. Files containing related functions can become libraries of routines that can be used in multiple programs.

Having a program distributed in multiple files raises some important issues. First, let's look at global, or external, variables. Within one file the scope of a global (external) variable is from its definition to the end of the file. The keyword "extern" makes it possible to use a global variable in multiple files. Suppose a program is in three files and it is desired to use a global variable, myState, to communicate some information between routines in each of the files. The variable, myState, is defined in one file and declared in the other two files using the extern keyword.

```
File One:
int myState;
int main()
{
   ....
}

File Two:
extern int myState;
float foo1()
{
   ....
}
float foo2()
{
   ....
}
.....

File Three:
extern int myState;
float fooA()
{
   .....
}
....
```

Notice that a variable may only be defined once. It may be declared multiple times. Remember that a definition creates space, or reserves memory, for the variable. The declaration just informs the compiler that a variable or function exists. The keyword extern tells the compiler that the variable is defined in another file.

The second issue with multiple files is that of function declarations, or prototypes. Function prototypes should always be used even if the compiler used does not enforce their use. They allow the compiler to perform type checking on the arguments in function calls and can prevent many errors. Suppose a function that is defined in one file is to be called in several others. Each of these other files should contain a prototype for this function. One approach would be as follows.

```
fileone.c:
int fun1(float x, float y);
int main()
{
   float x,y,z;
   x = 5.0;
   y = 6.0;
   z = fun1(x,y);
   ....
```

```
}

filetwo.c:
int fun1(float x, float y);
float fun2()
{
   float a = 4.0;
   float b = 6.0;
   float c;
   c = fun1(a,b);
   ....
}
.....

filethree.c:
int fun1(float x, float y);
....
....
float fun1(float x, float y)
{
   return (x * y);
}
....
```

Include Files and Program Structure

This works but is error prone and tedious. The function prototype must be typed into each file. Suppose we had many functions and hated typing. Fortunately, there is a simple solution, "include" files. All function prototypes and global variables can be put into an include file, and code in this file will be sourced into each file using the #include preprocessor directive.

```
Include File: filethree.h
int fun1(float x, float y);
......

fileone.c:
#include "filethree.h"
int main()
{
   float x,y,z;
   x = 5.0;
   y = 6.0;
   z = fun1(x,y);
   ....
}

filetwo.c:
#include "filethree.h";
float fun2()
{
   float a = 4.0;
   float b = 6.0;
   float c;
   c = fun1(a,b);
```

```
   ....
}
.....

filethree.c:
#include "filethree.h";
....
....
float fun1(float x, float y)
{
   return (x * y);
}
....
```

In general, I recommend putting the main program in a single file and related functions in their own files. So, you might have a large program divided into several files such as main.c, input.c, output.c and calc.c. Here, main.c would contain the main program. Input functions would be in input.c. Output functions would be in output.c and calculation functions in calc.c. Each of the "function" files should have its own include file containing the prototypes of all the functions in that file. So in this example, you would also have the include files input.h, output.h and calc.h. Assume that main uses functions from all three files and that calc.c requires routines from output.c to log error messages. Overall the program looks like:

```
main.c:
#include "input.h"
#include "output.h"
#include "calc.h"
main()
{
   ....
}

input.c:
function1() /* Functions can, of course, have any name */
{
}
function2()
{
}

output.c:
functionA()
{
}
functionB()
{
}

calc.c:
#include "output.h"
functiony()
{
}
functionz()
{
}
```