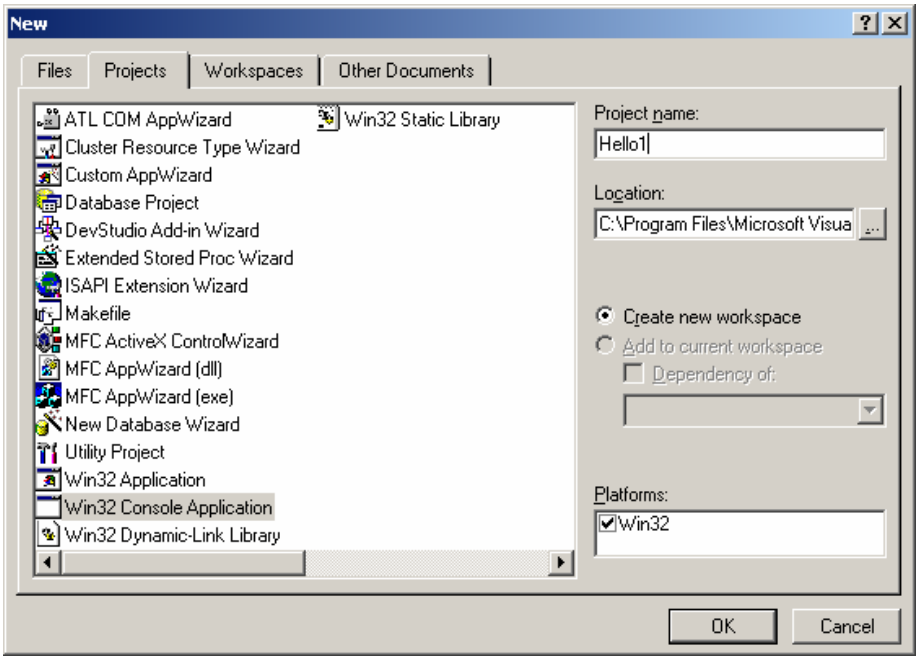
# Introduction to Programming in C Language

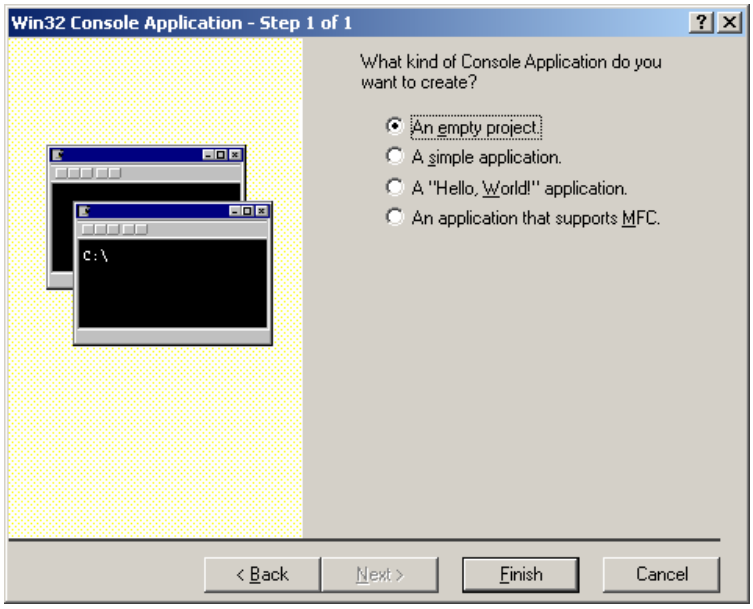
***© Dr. Zulfikar H. A. Kassam***
***zulfikarkassam@shaw.ca***

# Creating a C Program using Microsoft Visual C++

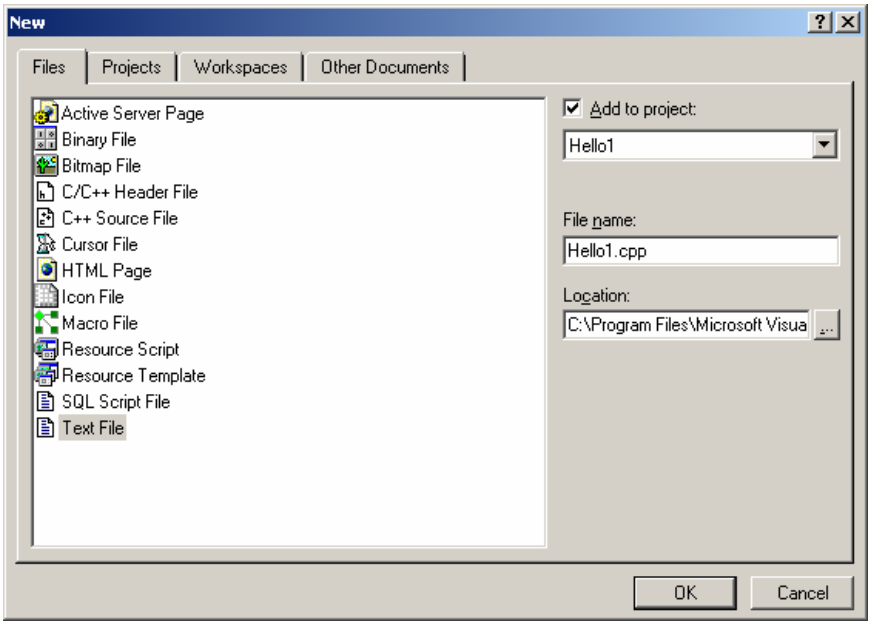
1. Go to Start | Programs | Microsoft Visual C++ 6.0 | Microsoft Visual C++ 6.0
2. Choose **File | New** from the main menu.




3. Select the **Projects** Tab.
4. Click the **Win32 Console Application** icon.
5. Specify the Project name in the **Project name:** dialog box.
6. Change the **Location** if necessary.
7. Click **OK** to continue.



8. Select **An empty project**.
9. Click **Finish**.
10. Click **OK** after verifying the New Project Information window.
11. Click **File | New** from the main menu.




12. Select **Text File**.
13. Select **Add to project** check mark.
14. Type in the **Filename** in the format *filename.cpp*.
15. Type in the program listing.
16. Click the **Save All** button to save the whole project.
17. Select **Build | Build Hello.exe** from the main menu.
18. Use the **command prompt** to go the directory called **debug** within the project path. (usually **C:\Program Files\Microsoft Visual Studio\MyProjects\Hello\debug**>
19. Run **hello.exe** from the **command prompt**.

# Section I

The best way to learn C programming is to enter, compile and run small to medium sized listings.

## *Exercise 1 - The smallest C Program*

Please type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. This program does do nothing. However, it illustrates the most important fundamental aspects of the C Language.

```
1:      main()
2:      {
3:              return 0;
4:      /* This is a comment */
5:      }
```

### Explanation of each line

**Line 1:**
**Main()** is a Function. All C programs have the **main()** function that is the body of the program. It is defined with empty brackets because it has no parameters to pass. Parameters are values (integers, real numbers, strings, etc.) that you can pass when calling a function. We shall discuss more about Functions and Parameter passing during the later stages.

**Line 2:**
All functions start with an open Parenthesis.

**Line 3:**
All functions have a **return** statement that passes the end result back. In this case, there is no end result. Therefore, it uses the return 0 statement.

**Line 4:**
Shows how one can use comments within a program to explain the code. This is extremely useful when you come back to the program later on. Note that Slash-Star combination starts the comment and the Star-Slash combination ends a comment. The compiler ignores all comments.

**Line 5:**
All functions end with a close Parenthesis.

## *The Standard Input Output*

C Language supplies a standard package for performing input and output to files or the terminal. This contains most of the functions that will be introduced in this

section, along with definitions of the data types required to use them. To use these facilities, your program must include these definitions by adding the line

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

at the beginning of the program file.

## *Formatted Input / Output*

### printf

This offers a structured output. Its arguments consist of a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string. Example

```
printf("My favorite number is %i.\n",num);
```

where the value of variable **num** would be substituted in the position of **%i**. Note that **%i** yields the same result as **%d** since **%i** is an integer and **%d** is a decimal integer.

| Control String Entry | What Gets Printed |
|---|---|
| %d | A Decimal Integer |
| %f | A Floating Point Value |
| %c | A Character |
| %s | A Character String |

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example **%6d** would print a decimal value in a field 6 spaces wide, **%8.2f** would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is *left justified* by default, but can be right justified by putting a - before the format information, for example **%-6d**, a decimal integer *right justified* in a 6 space field.

### scanf

**scanf** allows formatted reading of data from the keyboard. Like **printf** it has a control string, followed by the list of items to be read. However **scanf** wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the **&** sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.
The percentage sign precedes control string entries that match values to be read in a similar way to their **printf** equivalents.

Example:

```
scanf("%i",&variablename);
```

where **variablename** in this case is an integer variable.

## *Exercise 2 - Output onto the Screen*

Type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. This program does outputs a few statements onto the screen. While being simple, it builds on the concepts introduced earlier.

```
1:      #include <stdio.h>          /* include header file stdio.h */
2:      main()
3:      {
4:              printf("I am a simple ");     /* print the following lines */
5:              printf("computer.\n");
6:              printf("My favorite number is 1");
7:              return 0;
8:      }
```

### Explanation of each line

**Line 1:**
The **include** statement is used when one wants to include a *header file*. A header file is a file written in C Language that has predefined commands that one can use whenever necessary. An example in this case is the command **printf** that is included in the **stdio.h** file. C Language has a number of header files that can be used for various purposes. We shall cover them as we go along.

Note also the use of comments beside the line itself. It has a similar syntax to that described previously.

**Line 2:**
**Main()** is a Function. All C programs have the **main()** function that is the body of the program. It is defined with empty brackets because it has no parameters to pass. Parameters are values (integers, real numbers, strings, etc.) that you can pass when calling a function. We shall discuss more about Functions and Parameter passing during the later stages.

**Line 3:**
All functions start with an open Parenthesis.

**Line 4:**
The **printf** statement is used to output data onto the screen. The output you want to display on the screen is enclosed in brackets. If the output is a *string*, then it needs to be enclosed in quotes.

**Line 5:**
Similar to line 4 except that it includes **\n** parameter that places a carriage return after the period.

**Line 6:**
Same as line 4. Note that the text now appears on a separate line due to the \n parameter in the previous line.

**Line 7:**
All functions have a **return** statement that passes the end result back. In this case, there is no end result. Therefore, it uses the return 0 statement.

**Line 8:**
All functions end with a close Parenthesis.

## *Exercise 3 - Defining Variables and Displaying them on Screen*

Type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. This program does outputs a few statements onto the screen. While being simple, it builds on the concepts introduced earlier.

```
1:      #include <stdio.h>          /* include header file stdio.h */
2:      main()
3:      {
4:              int num;            /* declare integer num */
5:              num=1;              /* assign the value 1 to num*/
6:              printf("I am a simple ");    /* print the following lines */
7:              printf("computer.\n");
8:              printf("My favorite number is %i because it is first.\n",num);
9:              return 0;
10:     }
```

The main differences between this Exercise and the previous one is lines 4,5 and 8 which will be explained below.

**Line 4:**
Declares an integer variable. We shall discuss integer variables in more detail later on.

**Line 5:**
Assigns value to the integer variable.

**Line 8:**
Uses **%i** as part of the output string and has **num** after the comma to designate that the variable **num** should be placed at the position where **%i** appears.

## *Exercise 4 - Multiplication using Integer Variables*

Type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. This program does outputs a few statements onto the screen. While being simple, it builds on the concepts introduced earlier.

```
1:      #include <stdio.h>          /* include header file stdio.h */
2:      main()
3:      {
4:              int feet, inches;
5:              feet=5;
6:              inches=feet*12;
7:              printf("There are %i inches in %i feet.\n", inches, feet);
8:              return 0;
9:      }
```

## *Character Input / Output*

This is the lowest level of input and output. Most computers perform buffering of input and output. This means that they'll not start reading any input until the return key is pressed.

### getchar

**getchar** returns the next character of keyboard input until a certain programmed condition is encountered.

### putchar

**putchar** puts its character argument on the standard output (usually the screen).

## *Exercise 5 - Output onto the Screen using getchar and putchar*

Type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. Upon running the program, type in any word or series of words, or numbers, followed by pressing *Enter* key.

This program does outputs a few statements onto the screen. While being simple, it builds on the concepts introduced earlier. Some of the commands have not been covered as yet, but most of it should be apparent.

```
1:      #include <stdio.h>
2:      main()
3:      {
4:              char ch;
5:              int i=0;
6:              ch = getchar();
7:              putchar(ch);
8:              while(ch != '\n')
9:              {
10:                     i++;
11:                     ch=getchar();
12:                     putchar(ch);
13:             }
14:             printf("\n%d\n", i);
15:             return 0;
16:     }
```

## *Practice Exercises I*

1. Add your comments to the listings shown in Exercise 4 and Exercise 5.
2. Explain each and every line in Exercise 4 and Exercise 5 thoroughly.
3. The purpose of this exercise to create a program that will calculate the total purchase cost (not including taxes) of buying a computer with monitor and printer. To achieve follow the steps described below.

   Write a program that does the following:
   - Defines 4 integer variables with the following names: computer, monitor, printer and total.
   - Assigns the following values to 3 variables, i.e, computer = 1100, monitor = 300 and printer = 100
   - Calculates the value of the total variable by adding all the other 3 variables.
   - Has a final output that reads "The total purchase price is yyyy" where yyyy is the value of the total variable.
   - Add comments to enhance understanding.

4. Create a program that is similar to the one above except that use the following command to accept input from the keyboard for prices of the computer, monitor and printer and use these values to calculate the total cost. The command is

   **scanf("%i",&variablename);**

   e.g., to read in the price of the computer, you would include the following statement:

   **printf("Please input the cost the computer > ");**
   **scanf("%i",&computer);**

   Important: Note the **&** required before the variable name.

# Section II

## *Whole Lines of Input and Output*

Where we are not too interested in the format of our data, or perhaps we cannot predict its format in advance, we can read and write whole lines as character strings. This approach allows us to read in a line of input, and then use various string-handling functions to analyze it at our leisure.

### gets

**gets** reads a whole line of input into a string until a newline (**\n**) or **EOF** is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.
When all input is finished, NULL as defined in stdio.h is returned.

### puts

puts writes a string to the output, and follows it with a newline character.

## *Exercise 6 - Output onto the Screen using gets and puts*

Type in the listing shown below without the Line numbers and colons. Thereafter, compile it and run it. Upon running the program, type in any word or series of words, or numbers, followed by pressing *Enter* key.

This program does outputs a few statements onto the screen. While being simple, it builds on the concepts introduced earlier.

```
1:      #include <stdio.h>
2:      main()
3:      {
4:              char line[256];     /* Define string sufficiently large to
                                       store a line of input */
5:              gets(line);         /* Get line input */
6:              puts(line);         /* Print line */
7:              printf("\n");       /* Print blank line */
8:              return 0;
9:      }
```

## *Variables*

In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.

A declaration begins with the type, followed by the name of one or more variables. For example,

```
int high, low;
```

Declarations can be spread out, allowing space for an explanatory comment. Variables can also be initialized when they are declared, this is done by adding an equals sign and the required value after the declaration.

```
int high = 250;     /* Maximum Temperature */
int low = -40;      /* Minimum Temperature */
```

C provides a wide range of types. The most common are

| | |
|---|---|
| `int` | An Integer |
| `float` | A floating point (real) number |
| `char` | A single byte of memory, enough to hold a character |

Example,

```
float feet, inches;
```

There are also several variants on these types.

| | |
|---|---|
| `short` | An integer, possibly of reduced range |
| `long` | An integer, possibly of increased range |
| `unsigned` | An integer with no negative range, the spare capacity being used to increase the positive range |
| `unsigned long` | Like unsigned, possibly of increased range |
| `double` | A double precision floating point number. |

| Data Type | Size in Bytes | Size in Bits | Minimum Value | Maximum Value |
|---|---|---|---|---|
| char (signed char) | 1 | 8 | -128 | 127 |
| unsigned char | 1 | 8 | 0 | 255 |
| short (signed short) | 2 | 16 | -32,768 | 32,767 |
| unsigned short | 2 | 16 | 0 | 65,535 |
| int (signed int) | 2 | 16 | -32,768 | 32,767 |
| unsigned int | 2 | 16 | 0 | 65,535 |
| long (signed long) | 4 | 32 | -2,147,483,648 | 2,147,483,647 |
| unsigned long | 4 | 32 | 0 | 4,294,967,295 |

| Data Type | Size in Bytes | Size in Bits | Minimum Value | Maximum Value |
|---|---|---|---|---|
| float | 4 | 32 | 3.40E-38 | 3.40E+38 |
| double | 8 | 64 | 1.7E-308 | 1.7E+308 |
| long double | 10 | 80 | 3.4E-4932 | 3.4E+4932 |

| Conversion specifiers | Variable type |
|---|---|
| %c | single character |
| %d | Signed decimal integer |
| %e | floating-point number, e-notation |
| %E | floating-point number, E-notation |
| %f | floating-point number, decimal notation |
| %g | General format, uses %f or %e whichever is shorter. |
| %G | General format, uses %f or %E whichever is shorter. |
| %i | Signed integer |
| %p | Pointer |
| %s | Multiple character string |

All of the integer types plus the char are called the integral types. float and double are called the real types.

## Variable Names

Every variable has a name and a value. The name identifies the variable and the value stores the data. There is a limitation on what these names can be. Every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc. as variable names.
Examples of legal variable names include

```
x          result          outfile          bestyet
x1         x2              out_file         best_yet
power      impetus         gamma            hi_score
```

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name. Most modern ones don't apply this limit though. The rules governing variable names also apply to the names of functions. We shall meet functions later on in the course.

## Global Variables

Local variables are declared within the body of a function, and can only be used within that function. This is usually no problem, since when another function is called, all required data is passed to it as arguments. Alternatively, a variable can be declared globally so it is available to all functions. Modern programming practice recommends against the excessive use of global variables. They can lead to poor program structure, and tend to clog up the available name space.
A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions that access it.

## External Variables

Where a global variable is declared in one file, but used by functions from another, then the variable is called an external variable in these functions, and must be declared as such. The word **extern** must precede the declaration. The declaration is required so the compiler can find the type of the variable without having to search through several source files for the declaration.
Global and external variables can be of any legal type. They can be initialized, but the initialization takes place when the program starts up, before entry to the main function.

## Static Variables

Another class of local variables is the **static** type. A **static** variable can only be accessed from the function in which it was declared (similar to a local variable). The static variable is not destroyed on exit from the function; instead its value is preserved, and becomes available again when the function is next called. Static variables are declared as local variables, but the declaration is preceded by the word static.

```
static int counter;
```

Static variables can be initialized as normal, the initialization is performed once only, when the program starts up.

## Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.
Octal constants are written with a leading zero, e.g., 015.
Hexadecimal constants are written with a leading 0x, e.g., 0x1ae.
Long constants are written with a trailing L, e.g., 890L.
Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2-character sequence.

| | |
|---|---|
| '\n' | newline |
| '\t' | tab |
| '\\' | backslash |
| '\'' | single quote |
| '\0' | null (used automatically to terminate character strings). |

In addition, a required bit pattern can be specified using its octal equivalent. '\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes e.g. "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.
A character is a different type to a single character string. This is important.

## *Exercise 7 - Getting Keyboard Input and using Real (Floating Point) variables*

Type in the listing shown below. Thereafter, compile it and run it. Upon running the program, type in the number of feet (as a real variable) e.g., 10.56, followed by pressing *Enter* key.

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
        float feet, inches, conversion;
        conversion = 12.0;
        printf("This program allows you to convert from feet to inches\n");
        printf("How many feet?\n");
        scanf("%f",&feet);
        inches=feet*conversion;
        printf("There are %f inches in %f feet.\n", inches, feet);
        return 0;
}
```

## *Exercise 8 - Getting Keyboard Input and using Real (Floating Point) variables and using %G*

This is similar to Exercise 7 except replace %f with %G. Upon running the program, type in the number of feet (as a real variable, e.g., 10.56, followed by pressing *Enter* key.

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
        float feet, inches, conversion;
        conversion = 12.0;
        printf("This program allows you to convert from feet to inches\n");
        printf("How many feet?\n");
        scanf("%G",&feet);
        inches=feet*conversion;
        printf("There are %G inches in %G feet.\n", inches, feet);
        return 0;
}
```

**What differences do you observe in this output?**

## *Exercise 9 - More on String Input/Output*

Type in the listing shown below. Thereafter, compile it and run it. Upon running the program, type in your first name or your last name.

Then run it again by typing in your first and last name. What happens to the output?

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
    char name[40];     /* declaring a string or character variable */
    printf("This program allows you to input character strings.\n");
    printf("What is your name?\n");
    scanf("%s",&name);
    printf("Hi %s! Nice to meet you.\n", name);
    return 0;
}
```

## *Exercise 10 - REVIEW - More on String Input/Output*

Write a program similar to Exercise 9 but using **gets** instead of **scanf** statement. What differences do you observe when you do the same test described in Exercise 9?

## *Expressions and Operators*

One reason for the power of C is its wide range of useful operators. An operator is a function that is applied to values to give a result. You should be familiar with operators such as +, -, /.
Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits.
Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

### Assignment Statement

The easiest example of an expression is in the assignment statement. An expression is evaluated, and the result is saved in a variable. A simple example might look like

```
y = (m * x) + c;
```

This assignment will save the value of the expression in variable y.

### Arithmetic operators

Here are the most common arithmetic operators

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulo Reduction (Remainder from integer division)

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash.
Here are some arithmetic expressions used within assignment statements.

```
velocity = distance / time;

force = mass * acceleration;

count = count + 1;
```

C has some operators that allow abbreviation of certain types of arithmetic assignment statements.

| Shorthand | Equivalent |
|---|---|
| i++; or ++i; | i = i + 1; |
| i--; or --i; | i = i - 1; |

These operations are usually very efficient. They can be combined with another expression.

```
x = a * b++;   is equivalent to   x = a * b;
                                  b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

```
x = --i * (a + b);   is equivalent to   i = i - 1;
                                        x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - to ensure that your programs stay readable.
Another shorthand notation is listed below:

| Shorthand | Equivalent |
|---|---|
| i += 10; | i = i + 10; |
| i -= 10; | i = i - 10; |
| i *= 10; | i = i * 10; |
| i /= 10; | i = i / 10; |

These are simple to read and use.

## Type conversion

You can mix the types of values in your arithmetic expressions. char types will be treated as int. Otherwise where types of different size are involved, the result will usually be of the larger size, so a float and a double would produce a double result. Where integer and real types meet, the result will be a double.
There is usually no trouble in assigning a value to a variable of different type. The value will be preserved as expected except where;
The variable is too small to hold the value. In this case it will be corrupted (this is bad).
The variable is an integer type and is being assigned a real value. The value is rounded down. This is often done deliberately by the programmer.
Values passed as function arguments must be of the correct type. The function has no way of determining the type passed to it, so automatic conversion cannot take place. This can lead to corrupt results. The solution is to use a method called casting, which temporarily disguises a value as a different type.
e.g., The function **sqrt** finds the square root of a double.

```
int i = 256;
int root

root = sqrt( (double) i);
```

Putting the bracketed name of the required type just before the value makes the cast. Look at (double) in this example. The result of sqrt( (**double**) i); is also a double, but this is automatically converted to an **int** on assignment to root.

## Comparison

C has no special type to represent logical or Boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

| C notation | Meaning |
|---|---|
| == | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| != | Not equal to |

Note that == is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

```
x == y

i > 10

a + b != c
```

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement or a for or a while loop. These will be introduced in a later chapter.

## Logical Connectors

These are the usual And, Or and Not operators.

| Symbol | Meaning |
|---|---|
| && | And |
| \|\| | Or |
| ! | Not |

They are frequently used to combine relational operators, for example

```
if (x < 20 && x >= 10)
```

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.
Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
        printf("Not Acceptable !!\n");
```

## *Practice Exercises II*

You are asked to design a program for a hotel check-in counter. The program should ask you for the following items in sequence:

- First Name
- Last Name
- Street Address
- Apt Number
- City
- Province
- Postal Code
- Number of nights of stay
- Flat rates of $50 a night will be applied.

The idea is to take in all the information listed above, calculate the total payment that will be billed to the customer including taxes (both GST and PST).

The final display output will be the full customer information including the amount owing which shows the subtotal, each of taxes separately and the final total.

# Section III

## *Control Statements*

A program consists of a number of statements that are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.
Statements fall into three general types

- Assignment: where values (usually the results of calculations) are stored in variables.
- Input / Output: data is read in or printed out.
- Control: the program makes a decision about what to do next.

This section will discuss the use of control statements in C. We will show how they can be used to write powerful programs by:

- Repeating important sections of the program.
- Selecting between optional sections of a program.

## The if else Statement

This is used to decide whether to do something at a special point, or to decide between two courses of action.
The following test decides whether a student has passed an exam with a pass mark of 60

```
if (result >= 60)
        printf("Pass\n");
else
        printf("Fail\n");
```

It is possible to use the **if** statement without the else.

```
if (temperature < 0)
        print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the **if**). If the test is true then the next statement is obeyed. If it is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.
If we wish to have more than one statement following the **if** or the **else**, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{
   printf("Passed\n");
   printf("Congratulations\n")
}
else
{
```

```
   printf("Failed\n");
   printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 90)
        printf("Passed: Grade A\n");
else if (result >= 70)
        printf("Passed: Grade B\n");
else if (result >= 60)
        printf("Passed: Grade C\n");
else
        printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else **if** statements and the final **else** statement may be left out. It is up to the programmer to devise the correct structure for each programming problem.

## The switch Statement

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable.
- The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch.
- A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function that converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small. An *estimate* function is described below utilizing the **switch** statement.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{
   switch(number)
   {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
```

```
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
        }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.
Both if and switch constructs allow the programmer to make a selection from a number of possible actions.
The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times, the loop is C's way of achieving this.

## Loops

C gives you a choice of three types of loop, **while**, **do while** and **for**.

- The **while** loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The **do while** loops is similar, but the test occurs after the loop body is executed. *This ensures that the loop body is run at least once*.
- The **for** loop is frequently used, usually where the loop will be traversed a fixed number of times.

## The while Loop

The while loop repeats a statement until the test at the top proves false.
As an example, here is a function to return the length of a string.

Remember that the string is represented as an array of characters terminated by a **NULL** character '**\0**'.

```
int string_length(char string[])
{
         int i = 0;

        while (string[i] != '\0')
                i++;

        return(i);
}
```

The string is passed to the function as an argument. The size of the array is not specified; the function will work for a string of any size.
The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't NULL, the index is incremented and the test is repeated.

Do the following exercise:

## *Exercise 11 - The while loop*

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
    float feet, inches, conversion;
    conversion = 12.0;

    int response;
    printf("This program allows you to convert feet into inches.\n");
            response=1;

    while (response == 1)

            {
                    printf("Please input your value in feet -> ");
                    scanf("%f",&feet);
                    inches=feet*conversion;
                    printf("There are %f inches in %f feet.\n", inches, feet);
                    printf("Would you like to continue?\n");
                    printf("Press [1] to continue or Press [0] to stop -> ");
                    scanf("%d",&response);
                    printf("Your response was %d\n \n", response);
            }

    return 0;
}
```

### The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

## *Exercise 12 - The do while loop*

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
    float feet, inches, conversion;
    conversion = 12.0;

    int response;
    printf("This program allows you to convert feet into inches.\n");
            response=1;
    do
            {
            printf("Please input your value in feet -> ");
            scanf("%f",&feet);
            inches=feet*conversion;
            printf("There are %6.2f inches in %6.2f feet.\n", inches, feet);
            printf("Would you like to continue?\n");
            printf("Press [1] to continue or Press [0] to stop -> ");
            scanf("%d",&response);
            printf("Your response was %d\n \n", response);
    }
    while (response == 1);

    return 0;

}
```

### The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialization of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

An example is a function that calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{       float total = 0.0;
        int i;

        for(i = 0; i < count; i++)
                total += array[i];
```

```
        return(total / count);
}
```

The **for** loop ensures that the correct number of array elements are added up before calculating the average.
The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialization or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.
It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The **for** loop is extremely flexible and allows many types of program behavior to be specified simply and quickly.

Do the following exercise:

## *Exercise 13 - The for loop*

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
    float inches, feet, conversion, counter_inc;

    conversion = 12.0;
    counter_inc = 0.5;

    int counter;      /* this is a counter for the feet */

    printf("This program allows you to convert feet into inches.\n");

    for (counter=1; counter <= 15; counter++)

    {
            feet=counter*counter_inc;
            inches=feet*conversion;
            printf("There are %.1f inches in %.1f feet.\n", inches, feet);
    }

    return 0;
}
```

### The break Statement

We have already met **break** in the discussion of the **switch** statement. It is used to exit from a **loop** or a **switch**, control passing to the first statement beyond the loop or a switch.

With loops, **break** can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

## The continue Statement

It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a **while** loop, jump to the test statement.
- In a **do while** loop, jump to the test statement.
- In a **for** loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

## *Practice Exercises III*

You are asked to design a program for a hotel check-in counter. The program should ask you for the following items in sequence:

- First Name
- Last Name
- Street Address
- Apt Number
- City
- Province
- Postal Code
- Number of nights of stay
- Room type (single, double)

The rates are as follows

| **Number of Nights** | **Single** | **Double** |
|---|---|---|
| 1-2 | $50 | $70 |
| 3-7 | $45 | $65 |
| 7+ | $40 | $60 |

The idea is to take in all the information listed above, calculate the total payment that will be billed to the customer including taxes (both GST and PST).

The final display output will be the full customer information including the amount owing which shows the subtotal, each of taxes separately and the final total.

# Section IV

## Arrays

An array is a collection of variables of the same type. Individual array elements are identified by an integer index. In C the index begins at zero and is always written inside square brackets.
We have already met single dimensioned arrays that are declared like this

```
int results[20];
```

Arrays can have more than one dimension, in which case they might be declared as

```
int results_2d[20][5];
int results_3d[20][5][3];
```

Each index has its own set of square brackets.

Where an array is declared in the main function it will usually have details of dimensions included. It is possible to use another type called a pointer in place of an array. This means that dimensions are not fixed immediately, but space can be allocated as required. This is an advanced technique which is only required in certain specialized programs.

When passed as an argument to a function, the receiving function need not know the size of the array. So for example if we have a function which sorts a list (represented by an array) then the function will be able to sort lists of different sizes. The drawback is that the function is unable to determine what size the list is, so this information will have to be passed as an additional argument.
As an example, here is a simple function to add up all of the integers in a single dimensioned array.

```
int add_array(int array[], int size)
{       int i;
        int total = 0;

        for(i = 0; i < size; i++)
                total += array[i];

        return(total);
}
```

## *Exercise 14 - To find a maximum value, introducing ARRAYS*

```
#include <stdio.h>          /* include header file stdio.h */
main()
{
        int input[5], maximum, counter;
        printf("This program allows you to input 5 positive integer numbers
and find the maximum.\n");

        maximum=0;
        for (counter=0; counter <= 4; counter++)
        {
                scanf("%d",&input[counter]);
                if (input[counter]>maximum)
                maximum=input[counter];
        }
        printf("The maximum number is %d", maximum);
        return 0;

}
```

## *Functions in C*

Almost all programming languages have some equivalent of the function. You may have met them under the alternative names, i.e., subroutine or procedure. In C language, if the programmer wants a return value, this is achieved using the return statement. If no return value is required, the function can be defined as a **void function**.
Here is a function that raises a double to the power of an unsigned, and returns the result.

```
1: double power(double val, unsigned pow)
2: {
3:       double ret_val = 1.0;
4:       unsigned i;
5:       for(i = 0; i < pow; i++)
6:       ret_val *= val;
7:       return(ret_val);
8: }
```

The function follows a simple algorithm, multiplying the value by itself pow times. A **for** loop is used to control the number of multiplications, and variable ret_val stores the value to be returned.

Let us examine the details of this function.

**Line 1:**
This line begins the function definition. It tells us the type of the return value, the name of the function, and a list of arguments used by the function. The arguments and their types are enclosed in brackets, each pair separated by commas.

**Line 2:**
The body of the function is bounded by a set of curly brackets. Any variables declared here will be treated as local unless specifically declared as static or extern types.

**Line 7:**
On reaching a return statement, control of the program returns to the calling function. The bracketed value is the value that is returned from the function. If the final closing curly bracket is reached before any return value, then the function will return automatically, any return value will then be meaningless.
A line can call the example function shown above by using the following statement.

```
result = power(val, pow);
```

This calls the function power assigning the return value to variable **result**.
Here is an example of a function that does not return a value.

```
void error_line(int line)
{
   fprintf(stderr, "Error in input data: line %d\n", line);
}
```

The definition uses type **void**, which is optional. The usage of **void** indicates that no return value is required. Otherwise the function is much the same as the previous example, except that there is no return statement. Some void type functions might use return, but only to force an early exit from the function, and not to return any value.

The function would be called as follows

```
error_line(line_number);
```

## *Scope of Function Variables*

### Local variables

Only a limited amount of information is available within each function. Variables declared within the calling function are called **local variables**. Local variables cannot be accessed from outside of the function unless they are passed to the called function as arguments.
Local variables are declared within a function. They are created anew each time the function is called, and deleted on return from the function. Values passed to the function as arguments can also be treated like local variables.

### Static variables

Static variables are slightly different; they are not deleted on return from the function. Instead their last value is retained, and it becomes available when the function is called again.

## Global variables

The only other contact a function might have with the outside world is through global variables. Global variables also retain their values, and are available to any other function that accesses them.

## *Exercise 15 - Introducing Functions*

Type in the following exercise and compile it and run it.

```
1:   #include <stdio.h>          /* include header file stdio.h */
2:   float CanadaToUSDollars(float CdnDollars);
3:   main()
4:   {
5:   /* This program converts Canadian Dollars to US Dollars */
6:   float CanadianDollars,USDollars;
7:   printf("This program converts Canadian Dollars to US Dollars\n");
8:   printf("Please enter the Canadian Dollars Amount -> ");
9:   scanf("%f",&CanadianDollars);
10:  USDollars = CanadaToUSDollars(CanadianDollars);
11:  printf("$%.2f Canadian can be converted to $%.2f
     US",CanadianDollars,USDollars);
12:  return 0;
13:  }

14:  float CanadaToUSDollars(float CdnDollars)
15:  {
16:        float ExchangeRate= 1.6;
17:        return(CdnDollars*ExchangeRate);
18:  }
```

**Line 2:**
Declaring the function **CanadaToUSDollars** that will be defined later (in line 14)

**Line 10:**
Calls the **CanadaToUSDollars** function and assigns the return value to **USDollars**. The parenthesis defines the parameter that will be passed to the function. In this case, it is the amount in Canadian dollars.

**Line 14:**
Defines the function **CanadaToUSDollars**. The return value will be a floating point number and hence the function is defines as **float CanadaToUSDollars**. The parenthesis defines the parameters that will be passed to the function when calling the function. In this case, it is the amount in Canadian dollars.

**Line 17:**
Return the result of the operation back to the calling line (line 10) and assigns the resulting value to **USDollars**.

## *Modifying Function Arguments*

Some functions work by modifying the values of their arguments. This may be done in order to pass more than one value back to the calling routine, or because the return value is already being used in some way. C language requires special arrangements for arguments whose values will be changed.
You can treat the arguments of a function as variables, **however direct manipulation of these arguments won't change the values of the arguments in the calling function.** The value passed to the function is a copy of the calling value. This value is stored like a local variable, it disappears on return from the function.
**There is a way to change the values of variables declared outside the function. It is achieved by passing the addresses of variables to the function, i.e., by using pointers.** These addresses, or pointers, behave a bit like integer types, except that only a limited number of arithmetic operators can be applied to them. They are declared differently to normal types, and we are rarely interested in the value of a pointer. It is what lies at the address which the pointer references that interest us.
To get back to our original function, we pass it the address of a variable whose value we wish to change. The function must now be written to use the value at that address (or at the end of the pointer). On return from the function, the desired value will have changed. We manipulate the actual value using a copy of the pointer.

## *Pointers in C*

Pointers are not exclusive to functions, but this seems a good place to introduce the pointer type.
Imagine that we have an int called i. Its address could be represented by the symbol &i. If the pointer is to be stored as a variable, it should be stored like this.

```
int *pi = &i;
```

**int *** is the notation for a pointer to an int. **&** is the operator which returns the address of its argument. When it is used, as in **&i** we say it is referencing **i**.
The opposite operator, which gives the value at the end of the pointer is *. An example of use, known as de-referencing pi, would be

```
i = *pi;
```

Take care not to confuse the many uses of the * sign: Multiplication, pointer declaration and pointer de-referencing.
This is a very confusing subject, so let us illustrate it with an example. The following function fiddle takes two arguments, x is an int while y is a pointer to int. It changes both values.

```
fiddle(int x, int *y)
{   printf(" Starting fiddle: x = %d, y = %d\n", x, *y);
    x ++;
    (*y)++;
```

```
    printf("Finishing fiddle: x = %d, y = %d\n", x, *y);
}
```

since y is a pointer, we must de-reference it before incrementing its value.
A very simple program to call this function might be as follows.

```
main()
{   int i = 0;
    int j = 0;

    printf(" Starting main  : i = %d, j = %d\n", i, j);
    printf("Calling fiddle now\n");.
    fiddle(i, &j);
    printf("Returned from fiddle\n");
    printf("Finishing main  : i = %d, j = %d\n", i, j);
}
```

Note here how a pointer to int is created using the & operator within the call fiddle(i, &j);.
The result of running the program will look like this.

```
Starting main  : i =   0 ,j =   0
Calling fiddle now
Starting fiddle: x =   0, y =   0
Finishing fiddle: x =   1, y =   1
Returned from fiddle
Finishing main  : i =   0, j =   1
```

After the return from fiddle the value of i is unchanged while j, which was passed as a pointer, has changed.

To summarize, if you wish to use arguments to modify the value of variables from a function, these arguments must be passed as pointers, and de-referenced within the function.
Where the value of an argument isn't modified, the value can be passed without any worries about pointers.

## *Arrays and Pointers*

To fully understand the workings of C you must know that pointers and arrays are related.
**An array is actually a pointer to the 0th element of the array**.
Dereferencing the array name will give the 0th element. This gives us a range of equivalent notations for array access. In the following examples, arr is an array.

| Array Access | Pointer Equivalent |
|---|---|
| arr[0] | *arr |
| arr[2] | *(arr + 2) |
| arr[n] | *(arr + n) |

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the values in the array, but not the array itself, so statements like `arr ++` are illegal, but `arr[n] ++` is legal.

Since an array is like a pointer, we can pass an array to a function, and modify elements of that array without having to worry about referencing and de-referencing. Since the array is implemented as a hidden pointer, all the difficult stuff gets done automatically.
A function that expects to be passed an array can declare that parameter in one of two ways.

```
int arr[];  or int *arr;
```

Either of these definitions is independent of the size of the array being passed. This is met most frequently in the case of character strings, which are implemented as an array of type char. This could be declared as char string[]; but is most frequently written as char *string; In the same way, the argument vector argv is an array of strings which can be supplied to function main. It can be declared as one of the following.

```
char **argv;  or  char *argv[];
```

Don't panic if you find pointers confusing. While you will inevitably meet pointers in the form of strings, or as variable arguments for functions, they need not be used in most other simple types of programs.

## *Practice Exercises IV*

Expand Exercise 14 so that it can accept 10 numbers (instead of 5) and calculate the minimum and maximum input value, sum of all the input values and find the average. The program should have 4 functions defined called CalculateMinumum, CalculateMaximum, CalculateSum, CalculateAverage.

# Section V

## *Handling Files in C*

This section describes the use of C's input / output facilities for reading and writing files. There is also a brief description of string handling functions here. The functions are all variants on the forms of input / output which were introduced in the previous section.

### C File Handling

C communicates with files using a new datatype called a file pointer. This type is defined within **stdio.h**, and written as FILE *. A file pointer called output_file is declared in a statement like

```
FILE *output_file;
```

### Opening a file pointer using fopen

Your program must open a file before it can access it. This is done using the fopen function, which returns the required file pointer. If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows

```
if ((output_file = fopen("output_file", "w")) == NULL)
        fprintf(stderr, "Cannot open %s\n", "output_file");
```

**fopen** takes two arguments, both are strings, the first is the name of the file to be opened, the second is an access character, which is usually one of:

| | |
|---|---|
| "r" | Open file for reading |
| "w" | Create file for writing |
| "a" | Open file for appending |

### Standard file pointers

C Language provides three file descriptors that are automatically open to all C programs. These are

| | |
|---|---|
| stdin | The standard input. The keyboard or a redirected input file. |
| stdout | The standard output. The screen or a redirected output file. |
| stderr | The standard error. This is the screen, even when output is redirected. This is the conventional place to put any error messages. |

Since these files are already open, there is no need to use **fopen** on them.

### Closing a file using fclose

The **fclose** command can be used to disconnect a file pointer from a file. This is usually done so that the pointer can be used to access a different file. Systems

have a limit on the number of files which can be open simultaneously, so it is a good idea to close a file when you have finished using it.
This would be done using a statement like

```
fclose(output_file);
```

If files are still open when a program exits, the system will close them for you. However it is usually better to close the files properly.

## *Input and Output using file pointers*

Having opened a file pointer, you will wish to use it for either input or output. C supplies a set of functions to allow you to do this. All are very similar to input and output functions that you have already met.

### Character Input and Output with Files

This is done using equivalents of **getchar** and **putchar** that are called **getc** and **putc**. Each takes an extra argument, which identifies the file pointer to be used for input or output.

```
putchar(c)   is equivalent to   putc(c, stdout)
getchar()    is equivalent to   getc(stdin)
```

### Formatted Input Output with File Pointers

Similarly there are equivalents to the functions **printf** and **scanf** which read or write data to files. These are called **fprintf** and **fscanf**. You have already seen **fprintf** being used to write data to **stderr**.
The functions are used in the same way, except that the **fprintf** and **fscanf** take the file pointer as an additional first argument.

### Formatted Input Output with Strings

These are the third set of the **printf** and **scanf** families. They are called **sprintf** and **sscanf**.

**sprintf**
**puts** formatted data into a string which must have sufficient space allocated to hold it. This can be done by declaring it as an array of char. The data is formatted according to a control string of the same form as that for **printf**.

**sscanf**
takes data from a string and stores it in other variables as specified by the control string. This is done in the same way that **scanf** reads input data into variables. **sscanf** is very useful for converting strings into numeric v values.

### Whole Line Input and Output using File Pointers

Predictably, equivalents to gets and puts exist called **fgets** and **fputs**. The programmer should be careful in using them, since they are incompatible with gets and puts. **fgets** requires the programmer to specify the maximum number of characters to be read. fgets and fputs retain the trailing newline character on the line they read or write, whereas gets and puts discard the newline.

When transferring data from files to standard input / output channels, the simplest way to avoid incompatibility with the newline is to use fgets and fputs for files and standard channels too.
For Example, read a line from the keyboard using

```
fgets(data_string, 80, stdin);
```

and write a line to the screen using

```
fputs(data_string, stdout);
```

## *Exercise 16 - Reads a text file one line at a time and output the lines onto the screen*

Create a text file using Notepad. This file can contain a few lines, e.g., a few sentences, one on each line. Thereafter save it. Type in the following C program and compile it and run it. When prompted, type in the name of the text file you created using Notepad (including full path if the file is not in the same folder as the executable file). The program will read 1 line at a time from the text file and output the contents onto the screen.

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

main()
{
    FILE *fp; /* Defining a pointer to a file */
    char buffer[256], filename[50];   /* filename stores the name of the file */

    puts("Please enter the name of the text file to be read -> ");
    gets(filename);
    fp=fopen(filename,"r");  /* Open the specified filename for read purposes */

    while (fgets(buffer,255,fp)!= NULL)
            fputs(buffer, stdout);

    fclose(fp);

    return 0;
}
```

## *Exercise 17 - Writes typed info into a text file*

Type in the following C program and compile it and run it. When prompted, type in the name of the text file you want to write to (including full path if the file is not in the same folder as the executable file). The program will prompt you for text. Please type in a sentence followed by the Enter key. Open the file that you had specified when you were prompted. You will see the sentence you typed in an additional line which is part of the program.

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

main()
{
    FILE *fp; /* Defining a pointer to a file */
    char buffer[256], filename[50];   /* filename stores the name of the file */

    puts("Please enter the name of the text file to be written to -> ");
    gets(filename);
    fp=fopen(filename,"w");  /* Open the specified filename for writing purposes */

    puts("Enter text to written to file -> ");
    gets(buffer);

    fputs(buffer, fp);          /*Write the typed info into the file */
    fputc('\n', fp);            /*Write a newline character to file */
    fputs("The above text was typed in by you", fp);          /*Write this line to file */

    fclose(fp);    /* Close file */

    return 0;
}
```

## *Special Characters*

C makes use of some 'invisible' characters which have already been mentioned. However a fuller description seems appropriate here.

### NULL, The Null Pointer or Character

NULL is a character or pointer value. If it is a pointer, then the pointer variable does not reference any object (i.e. a pointer to nothing). It is usual for functions that return pointers to return NULL if they failed in some way. The return value can be tested. See the section on fopen for an example of this.
NULL is returned by read commands of the gets family when they try to read beyond the end of an input file.
Where it is used as a character, NULL is commonly written as '\0'. It is the string termination character that is automatically appended to any strings in your C program. You usually need not bother about this final '\0', since it is handled automatically. However it sometimes makes a useful target to terminate a string search. There is an example of this in the string_length function example in the section on Functions in C.

### EOF, The End of File Marker

EOF is a character that indicates the end of a file. It is returned by read commands of the getc and scanf families when they try to read beyond the end of a file.

## *Other String Handling Functions*

As well as sprintf and sscanf, the UNIX system has a number of other string handling functions within its libraries. A number of the most useful ones are contained in the <strings.h> file, and are made available by putting the line

```
#include <strings.h>
```

near to the head of your program file.
A couple of the functions are described below.

| | |
|---|---|
| `strcpy(str1, str2)` | copies `str2` into `str1`. |
| `strcmp(str1, str2)` | compares the contents of `str1` and `str2`. Returns 0 (false) if both are equal. |

## *Structures in C*

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name that is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.
A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

### Defining a Structure

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.
Here is an example structure definition.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
} student;
```

This defines a new type student variables of type student can be declared as follows.

```
student st_rec;
```

Notice how similar this is to declaring an int or float.
The variable name is st_rec, it has members called name, course, age and year.

### Accessing Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure st_rec will behave just like a normal array of char, however we refer to it by the name

```
st_rec.name
```

Here the dot is an operator that selects a member from a structure.
Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own

operator -> which acts as follows. Assume that st_ptr is a pointer to a structure of type student We would refer to the name member as

```
st_ptr -> name
```

## Structures as Function Arguments

A structure can be passed as a function argument just like any other variable. This raises a few practical issues.
Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.
If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.
When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

## Further Uses of Structures

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct {
        double real;
        double imag;
} complex;
```

doubles have been used for each field because their range is greater than floats and because the majority of mathematical library functions deal with doubles by default.
In a similar way, structures could be used to hold the locations of points in multi-dimensional space. Mathematicians and engineers might see a storage efficient implementation for sparse arrays here.
Apart from holding data, structures can be used as members of other structures. Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases.
Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other connected structures. These are rather daunting to the new programmer, so we won't deal with them here.

# *The C Preprocessor*

The C preprocessor is a tool that filters your source code before it is compiled. The preprocessor allows constants to be named using the #define notation. The preprocessor provides several other facilities that will be described here. It is particularly useful for selecting machine dependent pieces of code for different computer types, allowing a single program to be compiled and run on several different computers.

The C preprocessor isn't restricted to use with C programs, and programmers who use other languages may also find it useful, however it is tuned to recognize features of the C language like comments and strings, so its use may be restricted in other circumstances.
The preprocessor is called cpp, however it is called automatically by the compiler so you will not need to call it while programming in C.

## Using #define to Implement Constants

We have already met this facility, in its simplest form it allows us to define textual substitutions as follows.

```
#define MAXSIZE 256
```

This will lead to the value 256 being substituted for each occurrence of the word MAXSIZE in the file.

## *Using #define to Create Functional Macros*

#define can also be given arguments which are used in its replacement. The definitions are then called macros. Macros work rather like functions, but with the following minor differences.
Since macros are implemented as a textual substitution, there is no effect on program performance (as with functions).
Recursive macros are generally not a good idea.
Macros don't care about the type of their arguments. Hence macros are a good choice where we might want to operate on reals, integers or a mixture of the two. Programmers sometimes call such type flexibility polymorphism.
Macros are generally fairly small.
Macros are full of traps for the unwary programmer. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules.
Here is an example of a macro that won't work.

```
#define DOUBLE(x) x+x
```

Now if we have a statement

```
a = DOUBLE(b) * c;
```

This will be expanded to

```
a = b+b * c;
```

And since * has a higher priority than +, the compiler will treat it as.

```
a = b + (b * c);
```

The problem can be solved using a more robust definition of DOUBLE

```
#define DOUBLE(x) (x+x)
```

Here the brackets around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.
In general it is better to write a C function than risk using a macro.

## Reading in Other Files using #include

The preprocessor directive #include is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions.
Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is /usr/include for most UNIX systems.
For example

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

Another use for #include for the programmer is where multi-file programs are being written. Certain information is required at the beginning of each program file. This can be put into a file called globals.h and included in each program file. Local header file names are usually enclosed by double quotes, " ". It is conventional to give header files a name that ends in .h to distinguish them from other types of file.
The following line would include our globals.h file.

```
#include "globals.h"
```

## Conditional selection of code using #ifdef

The preprocessor has a conditional statement similar to C's if else. It can be used to selectively include statements in a program. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program which will run on either type.
The keywords for conditional selection are; #ifdef, #else and #endif.

**#ifdef**
takes a name as an argument, and returns true if the the name has a current definition. The name may be defined using a #define, the -d option of the compiler, or certain names which are automatically defined by the UNIX environment.

**#else**
is optional and ends the block beginning with #ifdef. It is used to create a 2 way optional selection.

**#endif**
ends the block started by #ifdef or #else.
Where the #ifdef is true, statements between it and a following #else or #endif are included in the program. Where it is false, and there is a following #else, statements between the #else and the following #endif are included.
This is best illustrated by an example.

## Using #ifdef for Different Computer Types

Conditional selection is rarely performed using #defined values. A simple application using machine dependent values is illustrated below.

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

```
main()
{
#ifdef vax
      printf("This is a VAX\n");
#endif
#ifdef sun
      printf("This is a SUN\n");
#endif
}
```

sun is defined automatically on SUN computers. vax is defined automatically on VAX computers.

### Using #ifdef to Temporarily Remove Program Statements

#ifdef also provides a useful means of temporarily `blanking out' lines of a program. The lines in question are preceeded by #ifdef NEVER and followed by #endif. Of course you should ensure that the name NEVER isn't defined anywhere.

## *Some Useful Library Functions*

The following functions may be useful to you. Each manual page typically describes several functions, so if you see something similar to what you want, try looking in that manual page.

| Function | Description |
|---|---|
| abs | integer absolute value |
| ctime | convert date and time |
| fopen | open a stream |
| printf | formatted output |
| fputc | put character or word on a stream |
| getwd | get current working directory path name |
| strcat | string concatenation |
| ctype | character classification and conversion macros and functions |
| mktemp | make a unique file name |
| puts | put a string on a stream |
| sleep | suspend execution for interval |
| stdio | standard buffered input/output package |

### Precedence of C operators

The following table shows the precedence of operators in C. Where a statement involves the use of several operators, those with the lowest number in the table will be applied first.

| | Description | Represented By |
|---|---|---|
| 1 | Parenthesis | () [] |
| 1 | Structure Access | . -> |
| 2 | Unary | ! ~ ++ -- - * & |
| 3 | Mutiply, Divide, Modulus | * / % |
| 4 | Add, Subtract | + - |
| 5 | Shift Right, Left | >> << |
| 6 | Greater, Less Than, etc | > < = |
| 7 | Equal, Not Equal | == != |
| 8 | Bitwise AND | & |
| 9 | Bitwise Exclusive OR | ^ |
| 10 | Bitwise OR | \| |
| 11 | Logical AND | && |
| 12 | Logical OR | \|\| |
| 13 | Conditional Expression | ?: |
| 14 | Assignment | = += -= etc |
| 15 | Comma | , |

Some of these operators have not been described in this course, consult a textbook if you want details about them.

## Special Characters

The following special patterns are used to represent a single character in C programs. The leading backslash in the single quotes indicates that more information is to follow.

| C code | Meaning |
|---|---|
| '\014' | Bit pattern for Form Feed |
| '\n' | Newline |
| '\t' | Tab |
| '\\ ' | Backslash |
| '\'' | Single Quote |
| '\"' | Double Quote |
| '\b' | Backspace |
| '\r' | Carriage Return |
| '\f' | Form Feed |
| '\0' | NULL (String Terminator) |

## Formatted Input and Output Function Types

The following format strings can be used to specify data to be printed or read using the printf or scanf functions, or any of their variants; fprintf, sprintf, fscanf or sscanf .

Note that though all can be used with the printf variants, some are not available for scanf.

| FORMAT STRING | DESCRIPTION | scanf() |
|---|---|---|
| | Numeric Values | |
| %d | Decimal integer | y |
| %f | Floating point decimal(Real number) | y |
| %lf | Double (Long Real number) | y |
| %e | Exponential representation | n |
| %u | Unsigned (positive) integer | n |
| %x | Hexadecimal integer | y |
| %o | Octal integer | y |
| %g | Automatically Selects Shorter of %f and %e | n |
| | Character Types | |
| %c | Single character | y |
| %s | Character String, Defined as `char *` | y |