

# **GE2112 - FUNDAMENTALS OF COMPUTING AND PROGRAMMING**

## **UNIT IV**

### **INTRODUCTION TO C**

Overview of C – Constants, Variables and Data Types – Operators and Expressions – Managing Input and Output operators – Decision Making - Branching and Looping.

### **OVERVIEW OF C**

As a programming language, C is rather like Pascal or Fortran.. Values are stored in variables. Programs are structured by defining and calling functions. Program flow is controlled using loops, if statements and function calls. Input and output can be directed to the terminal or to files. Related data can be stored together in arrays or structures.

Of the three languages, C allows the most precise control of input and output. C is also rather more terse than Fortran or Pascal. This can result in short efficient programs, where the programmer has made wise use of C's range of powerful operators. It also allows the programmer to produce programs which are impossible to understand. Programmers who are familiar with the use of pointers (or indirect addressing, to use the correct term) will welcome the ease of use compared with some other languages. Undisciplined use of pointers can lead to errors which are very hard to trace. This course only deals with the simplest applications of pointers.

### **A Simple Program**

The following program is written in the C programming language.

```
#include <stdio.h>

main()
{
    printf("Programming in C is easy.\n");
}
```

### **A NOTE ABOUT C PROGRAMS**

In C, lowercase and uppercase characters are very important! All commands in C must be lowercase. The C programs starting point is identified by the word

*main()*

This informs the computer as to where the program actually starts. The brackets that follow the keyword *main* indicate that there are no arguments supplied to this program (this will be examined later on).

The two braces, { and }, signify the begin and end segments of the program. The purpose of the statement

*include <stdio.h>*

is to allow the use of the *printf* statement to provide program output. Text to be displayed by *printf()* must be enclosed in double quotes. The program has only one statement

```
printf("Programming in C is easy.\n");
```

*printf()* is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes "", it is printed without modification. There are some exceptions however. This has to do with the \

and % characters. These characters are modifier's, and for the present the \ followed by the n character represents a newline character. Thus the program prints

Programming in C is easy.

and the cursor is set to the beginning of the next line. As we shall see later on, what follows the \ character will determine what is printed, ie, a tab, clear screen, clear line etc. Another important thing to remember is that all C statements are terminated by a semi-colon ;

### Summary of major points:

- program execution begins at *main()*
- keywords are written in lower-case
- statements are terminated with a semi-colon
- text strings are enclosed in double quotes
- C is case sensitive, use lower-case and try not to capitalise variable names
- \n means position the cursor on the beginning of the next line
- *printf()* can be used to display text to the screen
- The curly braces { } define the beginning and end of a program block.

## **BASIC STRUCTURE OF C PROGRAMS**

C programs are essentially constructed in the following manner, as a number of well defined sections.

```
/* HEADER SECTION */
/* Contains name, author, revision number*/

/* INCLUDE SECTION */
/* contains #include statements */

/* CONSTANTS AND TYPES SECTION */
/* contains types and #defines */

/* GLOBAL VARIABLES SECTION */
/* any global variables declared here */

/* FUNCTIONS SECTION */
/* user defined functions */

/* main() SECTION */

int main()
{

}
```

## VARIABLE

User defined variables must be declared before they can be used in a program. Variables must begin with a character or underscore, and may be followed by any combination of characters, underscores, or the digits 0 - 9.

### LOCAL AND GLOBAL VARIABLES

#### Local

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

#### Global

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

### DEFINING GLOBAL VARIABLES

*/\* Demonstrating Global variables \*/*

#### **Example:**

```
#include <stdio.h>
int add_numbers( void );          /* ANSI function prototype */

/* These are global variables and can be accessed by functions from this point on */
int value1, value2, value3;

int add_numbers( void )
{
    auto int result;
    result = value1 + value2 + value3;
    return result;
}

main()
{
    auto int result;

    result = add_numbers();
    printf("The sum of %d + %d + %d is %d\n",
           value1, value2, value3, final_result);
}
```

The scope of global variables can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

**Example:**

```

#include <stdio.h>
void no_access( void ); /* ANSI function prototype */
void all_access( void );

static int n2;           /* n2 is known from this point onwards */

void no_access( void )
{
    n1 = 10;    /* illegal, n1 not yet known */
    n2 = 5;     /* valid */
}

static int n1;           /* n1 is known from this point onwards */

void all_access( void )
{
    n1 = 10;     /* valid */
    n2 = 3;     /* valid */
}

```

**AUTOMATIC AND STATIC VARIABLES**

C programs have a number of segments (or areas) where data is located. These segments are typically,

- \_DATA Static data
- \_BSS Uninitialized static data, zeroed out before call to main()
- \_STACK Automatic data, resides on stack frame, thus local to functions
- \_CONST Constant data, using the ANSI C keyword const

The use of the appropriate keyword allows correct placement of the variable onto the desired data segment.

**Example:**

```

/* example program illustrates difference between static and automatic variables */
#include <stdio.h>
void demo( void );      /* ANSI function prototypes */

void demo( void )
{
    auto int avar = 0;
    static int svar = 0;

    printf("auto = %d, static = %d\n", avar, svar);
    ++avar;
    ++svar;
}

main()
{
    int i;

```

```

        while( i < 3 ) {
            demo();
            i++;
        }
    }

```

## AUTOMATIC AND STATIC VARIABLES

### Example:

```

/* example program illustrates difference between static and automatic variables */
#include <stdio.h>
void demo( void );      /* ANSI function prototypes */

void demo( void )
{
    auto int avar = 0;
    static int svar = 0;

    printf("auto = %d, static = %d\n", avar, svar);
    ++avar;
    ++svar;
}

main()
{
    int i;

    while( i < 3 ) {
        demo();
        i++;
    }
}

```

### Program output

```

auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2

```

The basic format for declaring variables is

```
data_type var, var, ... ;
```

where *data\_type* is one of the four basic types, an *integer*, *character*, *float*, or *double* type.

Static variables are created and initialized once, on the first call to the function. Subsequent calls to the function do not recreate or re-initialize the static variable. When the function terminates, the variable still exists on the `_DATA` segment, but cannot be accessed by outside functions.

Automatic variables are the opposite. They are created and re-initialized on each entry to the function. They disappear (are de-allocated) when the function terminates. They are created on the `_STACK` segment.

## DATA TYPES AND CONSTANTS

The four basic data types are

- **INTEGER**

These are whole numbers, both positive and negative. Unsigned integers (positive values only) are supported. In addition, there are short and long integers.

The keyword used to define integers is,

*int*

An example of an integer value is 32. An example of declaring an integer variable called **sum** is,

*int sum;*

*sum = 20;*

- **FLOATING POINT**

These are numbers which contain fractional parts, both positive and negative. The keyword used to define float variables is,

- *float*

An example of a float value is 34.12. An example of declaring a float variable called **money** is,

*float money;*  
*money = 0.12;*

- **DOUBLE**

These are exponential numbers, both positive and negative. The keyword used to define double variables is,

- *double*

An example of a double value is 3.0E2. An example of declaring a double variable called **big** is,

*double big;*  
*big = 312E+7;*

- **CHARACTER**

These are single characters. The keyword used to define character variables is,

- *char*

An example of a character value is the letter **A**. An example of declaring a character variable called **letter** is,

```
char letter;  
letter = 'A';
```

Note the assignment of the character *A* to the variable *letter* is done by enclosing the value in **single quotes**. Remember the golden rule: Single character - Use single quotes.

### **Sample program illustrating each data type**

#### **Example:**

```
#include <stdio.h>  
  
main()  
{  
    int sum;  
    float money;  
    char letter;  
    double pi;  
  
    sum = 10;           /* assign integer value */  
    money = 2.21;       /* assign float value */  
    letter = 'A';       /* assign character value */  
    pi = 2.01E6;        /* assign a double value */  
  
    printf("value of sum = %d\n", sum );  
    printf("value of money = %f\n", money );  
    printf("value of letter = %c\n", letter );  
    printf("value of pi = %e\n", pi );  
}
```

#### ***Sample program output***

```
value of sum = 10  
value of money = 2.210000  
value of letter = A  
value of pi = 2.010000e+06
```

### **INITIALISING DATA VARIABLES AT DECLARATION TIME**

In C variables may be initialised with a value when they are declared. Consider the following declaration, which declares an integer variable *count* which is initialised to 10.

```
int count = 10;
```

## **SIMPLE ASSIGNMENT OF VALUES TO VARIABLES**

The = operator is used to assign values to data variables. Consider the following statement, which assigns the value 32 an integer variable *count*, and the letter **A** to the character variable *letter*

```
count = 32;  
letter = 'A'
```

## **Variable Formatters**

%d	decimal integer
%c	character
%s	string or character array
%f	float
%e	double

## **HEADER FILES**

Header files contain definitions of functions and variables which can be incorporated into any C program by using the pre-processor *#include* statement. Standard header files are provided with each compiler, and cover a range of areas, string handling, mathematical, data conversion, printing and reading of variables.

To use any of the standard functions, the appropriate header file should be included. This is done at the beginning of the C source file. For example, to use the function *printf()* in a program, the line

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

should be at the beginning of the source file, because the definition for *printf()* is found in the file *stdio.h*. All header files have the extension .h and generally reside in the /include subdirectory.

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

The use of angle brackets <> informs the compiler to search the compilers include directory for the specified file. The use of the double quotes "" around the filename inform the compiler to search in the current directory for the specified file.



## OPERATORS AND EXPRESSIONS

An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

### ARITHMETIC OPERATORS:

The symbols of the arithmetic operators are:-

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6
Subtraction	-	sum = sum - 2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

### **Example:**

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

```
main()
{
    int sum = 50;
    float modulus;

    modulus = sum % 10;
    printf("The %% of %d by 10 is %f\n", sum, modulus);
}
```

### PRE/POST INCREMENT/DECREMENT OPERATORS

PRE means do the operation first followed by any assignment operation. POST means do the operation after any assignment operation. Consider the following statements

```
++count;    /* PRE Increment, means add one to count */
count++;    /* POST Increment, means add one to count */
```

**Example:**

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

```
main()
{
    int count = 0, loop;

    loop = ++count; /* same as count = count + 1; loop = count; */
    printf("loop = %d, count = %d\n", loop, count);

    loop = count++; /* same as loop = count; count = count + 1; */
    printf("loop = %d, count = %d\n", loop, count);
}
```

If the operator precedes (is on the left hand side) of the variable, the operation is performed first, so the statement

```
loop = ++count;
```

really means increment *count* first, then assign the new value of *count* to *loop*.

**THE RELATIONAL OPERATORS**

These allow the comparison of two or more variables.

```
==  equal to
!=  not equal
<   less than
<=  less than or equal to
>   greater than
>=  greater than or equal to
```

**Example:**

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

```
main() /* Program introduces the for statement, counts to ten */
{
    int count;

    for( count = 1; count <= 10; count = count + 1 )
        printf("%d ", count );

    printf("\n");
}
```

## RELATIONALS (AND, NOT, OR, EOR)

### Combining more than one condition

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

LOGICAL AND    &&

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

LOGICAL OR    ||

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

LOGICAL NOT    !

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

LOGICAL EOR    ^

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

### Example:

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

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

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

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

### Example:

## NEGATION

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

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

```

        flag = !flag;
    }
    printf("The value of flag is %d\n", flag);
}

```

### Example:

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

```

#include <stdio.h>

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

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

```

## THE CONDITIONAL EXPRESSION OPERATOR or TERNARY OPERATOR

This conditional expression operator takes THREE operators. The two symbols used to denote this operator are the ? and the :. The first operand is placed before the ?, the second operand between the ? and the :, and the third after the :. The general format is,

*condition ? expression1 : expression2*

If the result of condition is TRUE ( non-zero ), expression1 is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then expression2 is evaluated and its result becomes the result of the operation. An example will help,

*s = ( x < 0 ) ? -1 : x \* x;*

*If x is less than zero then s = -1*

*If x is greater than zero then s = x \* x*

**Example:**

```
#include <stdio.h>

main()
{
    int input;

    printf("I will tell you if the number is positive, negative or zero!\n");
    printf("please enter your number now--->");
    scanf("%d", &input );
    (input < 0) ? printf("negative\n") : ((input > 0) ? printf("positive\n") : printf("zero\n"));
}

```

**BIT OPERATIONS**

C has the advantage of direct bit manipulation and the operations available are,

Operation	Operator	Comment	Value of Sum before	Value of sum after
AND	&	sum = sum & 2;	4	0
OR		sum = sum   2;	4	6
Exclusive OR	^	sum = sum ^ 2;	4	6
1's Complement	~	sum = ~sum;	4	-5
Left Shift	<<	sum = sum << 2;	4	16
Right Shift	>>	sum = sum >> 2;	4	0

**Example:**

```
/* Example program illustrating << and >> */
#include <stdio.h>

main()
{
    int n1 = 10, n2 = 20, i = 0;

    i = n2 << 4; /* n2 shifted left four times */
    printf("%d\n", i);
    i = n1 >> 5; /* n1 shifted right five times */
    printf("%d\n", i);
}

```

**Example:**

```
/* Example program using EOR operator */
#include <stdio.h>

```

```

main()
{
    int value1 = 2, value2 = 4;

    value1 ^= value2;
    value2 ^= value1;
    value1 ^= value2;
    printf("Value1 = %d, Value2 = %d\n", value1, value2);
}

```

#### Example:

```

/* Example program using AND operator */
#include <stdio.h>

main()
{
    int loop;

    for( loop = 'A'; loop <= 'Z'; loop++ )
        printf("Loop = %c, AND 0xdf = %c\n", loop, loop & 0xdf);
}

```

## MANAGING INPUT AND OUTPUT OPERATORS

### Printf ():

*printf()* is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes "", it is printed without modification. There are some exceptions however. This has to do with the \ and % characters. These characters are modifier's, and for the present the \ followed by the n character represents a newline character.

#### Example:

```

#include <stdio.h>

main()
{
    printf("Programming in C is easy.\n");
    printf("And so is Pascal.\n");
}

```

@ Programming in C is easy.  
And so is Pascal.

### FORMATTERS for printf are,

#### Cursor Control Formatters

\n	newline
\t	tab

\r	carriage return
\f	form feed
\v	vertical tab

### Scanf ():

Scanf () is a function in C which allows the programmer to accept input from a keyboard.

### Example:

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

```
main() /* program which introduces keyboard input */
{
    int number;

    printf("Type in a number \n");
    scanf("%d", &number);
    printf("The number you typed was %d\n", number);
}
```

### FORMATTERS FOR scanf()

The following characters, after the % character, in a scanf argument, have the following effect.

d	read a decimal integer
o	read an octal value
x	read a hexadecimal value
h	read a short integer
l	read a long integer
f	read a float value
e	read a double value
c	read a single character
s	read a sequence of characters
[...]	Read a character string. The characters inside the brackets

### ACCEPTING SINGLE CHARACTERS FROM THE KEYBOARD

#### Getchar, Putchar

*getchar()* gets a single character from the keyboard, and *putchar()* writes a single character from the keyboard.

### Example:

The following program illustrates this,

```

#include <stdio.h>

main()
{
    int i;
    int ch;

    for( i = 1; i <= 5; ++i ) {
        ch = getchar();
        putchar(ch);
    }
}

```

The program reads five characters (one for each iteration of the for loop) from the keyboard. Note that *getchar()* gets a single character from the keyboard, and *putchar()* writes a single character (in this case, *ch*) to the console screen.

## DECISION MAKING

### IF STATEMENTS

The *if* statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```

if( expression )
    program statement;

```

#### **Example:**

```

if( students < 65 )
    ++student_count;

```

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

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

#### **Example:**

```

#include <stdio.h>

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

    while( valid == 0 ) {

```



```

        printf("Enter a number between 1 and 10 -->");
        scanf("%d", &number);
        /* assume number is valid */
        valid = 1;
        if( number < 1 ) {
            printf("Number is below 1. Please re-enter\n");
            valid = 0;
        }
        if( number > 10 ) {
            printf("Number is above 10. Please re-enter\n");
            valid = 0;
        }
    }
    printf("The number is %d\n", number );
}

```

## **IF ELSE**

The general format for these are,

```

if( condition 1 )
    statement1;
else if( condition 2 )
    statement2;
else if( condition 3 )
    statement3;
else
    statement4;

```

The *else* clause allows action to be taken where the condition evaluates as false (zero).

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

### **Example:**

```

#include <stdio.h>

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

    while( valid == 0 ) {
        printf("Enter a number between 1 and 10 -->");
        scanf("%d", &number);
        if( number < 1 ) {
            printf("Number is below 1. Please re-enter\n");
            valid = 0;
        }
    }
}

```

```

        else if( number > 10 ) {
            printf("Number is above 10. Please re-enter\n");
            valid = 0;
        }
        else
            valid = 1;
    }
    printf("The number is %d\n", number );
}

```

This program is slightly different from the previous example in that an *else* clause is used to set the variable *valid* to 1. In this program, the logic should be easier to follow.

### **NESTED IF ELSE**

*/\* Illustrates nested if else and multiple arguments to the scanf function. \*/*

#### **Example:**

```

#include <stdio.h>

main()
{
    int invalid_operator = 0;
    char operator;
    float number1, number2, result;

    printf("Enter two numbers and an operator in the format\n");
    printf(" number1 operator number2\n");
    scanf("%f %c %f", &number1, &operator, &number2);

    if(operator == '*')
        result = number1 * number2;
    else if(operator == '/')
        result = number1 / number2;
    else if(operator == '+')
        result = number1 + number2;
    else if(operator == '-')
        result = number1 - number2;
    else
        invalid_operator = 1;

    if( invalid_operator != 1 )
        printf("%f %c %f is %f\n", number1, operator, number2, result );
    else
        printf("Invalid operator.\n");
}

```

## BRANCHING AND LOOPING

### ITERATION, FOR LOOPS

The basic format of the for statement is,

```
for( start condition; continue condition; re-evaluation )  
program statement;
```

#### **Example:**

```
/* sample program using a for statement */  
#include <stdio.h>  
  
main() /* Program introduces the for statement, counts to ten */  
{  
    int count;  
  
    for( count = 1; count <= 10; count = count + 1 )  
        printf("%d ", count );  
  
    printf("\n");  
}
```

The program declares an integer variable *count*. The first part of the *for* statement

```
for( count = 1;
```

initialises the value of *count* to 1. The *for* loop continues whilst the condition

```
count <= 10;
```

evaluates as TRUE. As the variable *count* has just been initialised to 1, this condition is TRUE and so the program statement

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

is executed, which prints the value of *count* to the screen, followed by a space character.

Next, the remaining statement of the *for* is executed

```
count = count + 1 );
```

which adds one to the current value of *count*. Control now passes back to the conditional test,

```
count <= 10;
```

which evaluates as true, so the program statement

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

is executed. *Count* is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test

```
count <= 10;
```

evaluates as FALSE, and the *for* loop terminates, and program control passes to the statement

```
printf("n");
```

which prints a newline, and then the program terminates, as there are no more statements left to execute.

## **THE WHILE STATEMENT**

The *while* provides a mechanism for repeating C statements whilst a condition is true. Its format is,

```
while( condition )  
    program statement;
```

Somewhere within the body of the *while* loop a statement must alter the value of the condition to allow the loop to finish.

### **Example:**

```
/* Sample program including while */  
#include <stdio.h>  
  
main()  
{  
    int loop = 0;  
  
    while( loop <= 10 ) {  
        printf("%d\n", loop);  
        ++loop;  
    }  
}
```

The above program uses a *while* loop to repeat the statements

```
printf("%d\n", loop);  
++loop;
```

whilst the value of the variable *loop* is less than or equal to 10.

Note how the variable upon which the *while* is dependant is initialised prior to the *while* statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the *while* loop will terminate.

This program is functionally equivalent to the earlier *for* program which counted to ten.

## **THE DO WHILE STATEMENT**

The *do { } while* statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero). The loop is executed as least once.

### **Example:**

```
/* Demonstration of DO...WHILE */
#include <stdio.h>

main()
{
    int value, r_digit;

    printf("Enter the number to be reversed.\n");
    scanf("%d", &value);
    do {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
    } while( value != 0 );
    printf("\n");
}
```

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digit into the variable *r\_digit*. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

## **SWITCH CASE:**

The *switch case* statement is a better way of writing a program when a series of *ifelses* occurs. The general format for this is,

```
switch ( expression ) {
    case value1:
        program statement;
        program statement;
        .....
        break;
    case valuen:
        program statement;
        .....
        break;
    default:
        .....
        .....
        break;
}
```

The keyword *break* must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

**Example:**

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

```
main()
{
    int menu, numb1, numb2, total;

    printf("enter in two numbers -->");
    scanf("%d %d", &numb1, &numb2 );
    printf("enter in choice\n");
    printf("1=addition\n");
    printf("2=subtraction\n");
    scanf("%d", &menu );
    switch( menu ) {
        case 1: total = numb1 + numb2; break;
        case 2: total = numb1 - numb2; break;
        default: printf("Invalid option selected\n");
    }
    if( menu == 1 )
        printf("%d plus %d is %d\n", numb1, numb2, total );
    else if( menu == 2 )
        printf("%d minus %d is %d\n", numb1, numb2, total );
}
```

The above program uses a *switch* statement to validate and select upon the users input choice, simulating a simple menu of choices.