Chapter 8: Function

- In this chapter, you will learn about
  - Introduction to function
  - User-defined function
  - Formal and Actual Parameters
  - Parameter passing by value
  - ~~Parameter passing by reference~~
  - Local and Global Variables
  - Storage classes

Introduction

- A function is a block of code which is used to perform a specific task.
- It can be written in one program and used by another program without having to rewrite that piece of code. Hence, it promotes usability!!!
- Functions can be put in a library. If another program would like to use them, it will just need to include the appropriate header file at the beginning of the program and link to the correct library while compiling.

Introduction

- Functions can be divided into two categories :
  - Predefined functions (standard functions)
    - Built-in functions provided by C that are used by programmers without having to write any code for them. i.e: printf( ), scanf( ), etc
  - User-Defined functions
    - Functions that are written by the programmers themselves to carry out various individual tasks.

Standard/Pre-defined Functions

- Standard functions are functions that have been pre-defined by C and put into standard C libraries.
  - Example: printf(), scanf(), pow(), ceil(), rand(), etc.
- What we need to do to use them is to include the appropriate header files.
  - Example: #include <stdio.h>, #include <math.h>
- What contained in the header files are the prototypes of the standard functions. The function definitions (the body of the functions) has been compiled and put into a standard C library which will be linked by the compiler during compilation.

User-defined Functions

- A programmer can create his/her own function(s).
- It is easier to plan and write our program if we divide it into several functions instead of writing a long piece of code inside the main function.
- A function is **reusable** and therefore prevents us (programmers) from having to unnecessarily rewrite what we have written before.
- In order to write and use our own function, we need to do the following:
  - create a function prototype (declare the function)
  - define the function somewhere in the program (implementation)
  - call the function whenever it needs to be used

Function Definition

- It is also called as *function implementation*
- A function definition is where the actual code for the function is written. This code will determine what the function will do when it is called.
- A function definition consists of the following elements:
  - A function return data type (return value)
  - A function name
  - An optional list of formal parameters enclosed in parentheses (function arguments)
  - A compound statement (function body)

Function Definition

- A function definition has this format:

```
rdtype fName (dt1 var1, dt2 var2, dt3 var3, …..)
{
        local variable declarations;
        statements;
}
```

- The return data type (rdtype) indicates the type of data that will be returned by the function fName to the calling function. There can be **only one return value**.
- A function that does not returning any value must be have the return data type written as 'void'.
- If the function does not receive any parameter from the calling function, 'void' is also used in the place of the parameter.
- Note that if the function is returning a value, it needs to use the keyword **`return`**.

Function definition example 1

- A simple function is :

```
void print_message (void)
{
  printf("Hello, are you having fun?\n");
}
```

- Note the function name is **print_message**. No arguments are accepted by the function, this is indicated by the keyword **void** enclosed in the parentheses. The return_value is **void**, thus data is not returned by the function.
- So, when this function is called in the program, it will simply perform its intended task which is to print

  Hello, are you having fun?

Function definition example 2

- Consider the following example:

```
int calculate (int num1, int num2)
{
 int sum;
 sum = num1 + num2;
 return sum;
}
```

- The above code segments is a function named **calculate**. This function accepts two arguments i.e. **num 1** and **num2** of the type **int**. The return_value is **int**, thus this function will return an integer value.
- So, when this function is called in the program, it will perform its task which is to **calculate the sum of any two numbers** and **return the result of the summation.**

Function Prototype

- A function prototype will tell the compiler that there exist a function with this name defined somewhere in the program and therefore it can be used even though the function has not yet been defined at that point.
- If the function receives some arguments, the variable names for the arguments are not needed. Only the data type need to be stated.
- Function prototypes need to be written at the beginning of the program.

Function Prototype

- For the following function definition,

```
void print_message (void)
{
  printf("Hello, are you having fun?\n");
}
```

- The corresponding function prototype is

```
void print_message (void);
```

Function Prototype

- For the following function definition,

```
int calculate (int num1, int num2)
  {
   int sum;
   sum = num1 + num2;
   return sum;
  }
```

- The corresponding function prototype is

```
int calculate (int, int);
```

Function Call

- Any function (including main) could utilise any other function definition that exist in a program – hence it is said to be calling the function and known as calling function.
- To call a function (i.e. to ask another function to do something for the calling function), it requires the FunctionName followed by a list of actual parameters (or arguments), if any, enclosed in parenthesis.
- For example, a function call to the function calculate defined in the previous slide will look something like below.

```
void main(void){
   ...
   calculate (6, 7);
   ...
}
```

Basic skeleton…

```
#include <stdio.h>

//function prototype
void fn1(void);

int main(void)
{
   ...
   //function call
   fn1( );
   ...
   return (0);
}

//function definition
void fn1(void)
{
   local variable declaration;
   statements;
}
```

main is the *calling function*

fn1 is the *called function*

Function Call cont…

- If the function returns a value, then the returned value need to be assigned to a variable so that it can be stored. For example:

```
int GetUserInput (void); /* function prototype*/
int main(void)
{
      int input;
      input = GetUserInput( );
      return(0);
}
```

- However, it is perfectly okay (syntax wise) to just call the function without assigning it to any variable if we want to ignore the returned value.
- We can also call a function inside another function. For example:

```
printf("User input is: %d", GetUserInput( ));
```

Types of function

- Type 1: Receive no input parameter and return nothing
- Type 2: Receive no input parameter but return a value
- Type 3: Receive input parameter(s) and return nothing
- Type 4: Receive input parameters(s) and return a value

Type 1 Example

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

int main(void)
{
  greeting( );
  greeting( );
  return(0);
}

void greeting(void)
{
  printf("Have fun!! \n");
}
```

In this example, function greeting does not receive any arguments from the calling function (main), and does not return any value to the calling function, hence type 'void' is used for both the input arguments and return data type.

```
Have fun!!
Have fun!!
Press any key to continue
```

Type 2 Example

```
#include <stdio.h>
int getInput(void);

int main(void)
{
   int num1, num2, sum;

   num1 = getInput( );
   num2 = getInput( );
   sum = num1 + num2;

   printf("Sum is %d\n",sum);
   return(0);
}

int getInput(void)
{
   int number;
   printf("Enter a number:");
   scanf("%d",&number);

   return number;
}
```

```
Enter a number: 5
Enter a number: 4
Sum is 9
Press any key to continue
```

Type 3 Example

```
#include <stdio.h>
int getInput(void);
void displayOutput(int);

int main(void)
{
    int num1, num2, sum;

    num1 = getInput();
    num2 = getInput();
    sum = num1 + num2;
    displayOutput(sum);
    return(0);
}

int getInput(void)
{
    int number;
    printf("Enter a number:");
    scanf("%d",&number);

    return number;
}

void displayOutput(int sum)
{
    printf("Sum is %d \n",sum);
}
```

```
Enter a number: 5
Enter a number: 4
Sum is 9
Press any key to continue
```

Type 4 Example

```
#include <stdio.h>
int calSum(int,int);        /*function prototype*/

int main(void)
{
   int sum, num1, num2;

   printf("Enter two numbers to calculate its sum:\n");

   scanf("%d%d",&num1,&num2);

   sum = calSum(num1,num2);  /* function call */
   printf("\n %d + %d = %d", num1, num2, sum);

   return(0);
}

int calSum(int val1, int val2)  /*function definition*/
{
   int sum;
   sum = val1 + val2;
   return sum;
}
```

```
Enter two numbers to calculate its sum:
4
9
4 + 9 = 13
Press any key to continue
```

Type 4 Example (Explanation)

- In this example, the `calSum` function receives input parameters of type int from the calling function (main).
- The calSum function returns a value of type int to the calling function.
- Therefore, the **function definition** for `calSum`:

```
int calSum(int val1, int val2)
```

- Note that the **function prototype** only indicates the type of variables used, not the names.

```
int calSum(int,int);
```

- Note that the function call is done by (main) calling the function name (calSum), and supplying the variables (num1,num2) needed by the calSum function to carry out its task.

Type 4 Example (Complete Flow)

```
#include<stdio.h>
int calSum(int, int);
int main(void)
{
   int num1, num2, sum;
   printf("Enter 2 numbers to calculate its sum:");
   scanf("%d %d", &num1, &num2);

   sum = calSum (num1, num2);
   printf ("\n %d + %d = %d", num1, num2, sum);
   return (0);
}

int calSum(int val1, int val2)
{
   int sum;
   sum = val1 + val2;
   return sum;
}
```

Function with parameter (Type 3 & 4)

- When a function calls another function to perform a task, the calling function may also send data to the called function. After completing its task, the called function may pass the data it has generated back to the calling function.
- Two terms used:
  - Formal parameter
    - Variables declared in the formal list of the function header (written in function prototype & function definition)
  - Actual parameter
    - Constants, variables, or expression in a function call that correspond to its formal parameter

Formal and Actual parameters

- The three important points concerning functions with parameters are: (number, order and type)
  - The number of actual parameters in a function call must be the same as the number of formal parameters in the function definition.
  - A one-to-one correspondence must occur among the actual and formal parameters. The first actual parameter must correspond to the first formal parameter and the second to the second formal parameter, an so on.
  - The type of each actual parameter must be the same as that of the corresponding formal parameter.

Formal and Actual parameters

```
#include <stdio.h>
int calSum(int,int);                    /*function prototype*/

int main(void)
{
    .....
    .....
    sum = calSum(num1,num2); /* function call */
    .....
}

int calSum(int val1, int val2) /*function header*/
{
    ......
    ......
    ......
}
```

Formal Parameters

Actual Parameters

Formal Parameters

Function Call cont…

- If the function requires some arguments to be passed along, then the arguments need to be listed in the bracket ( ) according to the specified order. For example:

```
void Calc(int, double, char, int);

int main(void)
{
        int a, b;
        double c;
        char d;
        …
        Calc(a, c, d, b);
        return (0);
}
```

Function Call

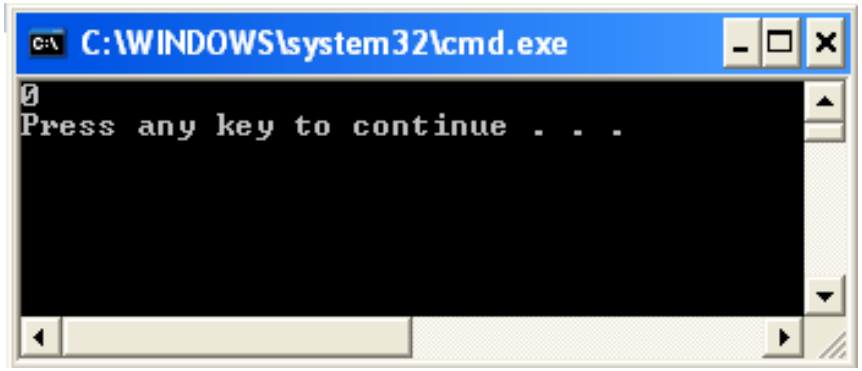
Types of Parameter Passing

- There are 2 ways to call a function:
  - Call by value (parameter passing by value)
    - In this method, only the **copy of variable's value** (copy of actual parameter's value) is passed to the function. Any modification to the passed value inside the function **will not** affect the actual value.
    - In all the examples that we have seen so far, this is the method that has been used.
  - ~~Call by reference (parameter passing by reference)~~
    - ~~In this method, the **reference** (memory address) of the variable, i.e. the pointer to the variable, is passed to the function. Any modification passed done to the variable inside the function **will** affect the actual value.~~

Limitation of Parameter Passing by Value

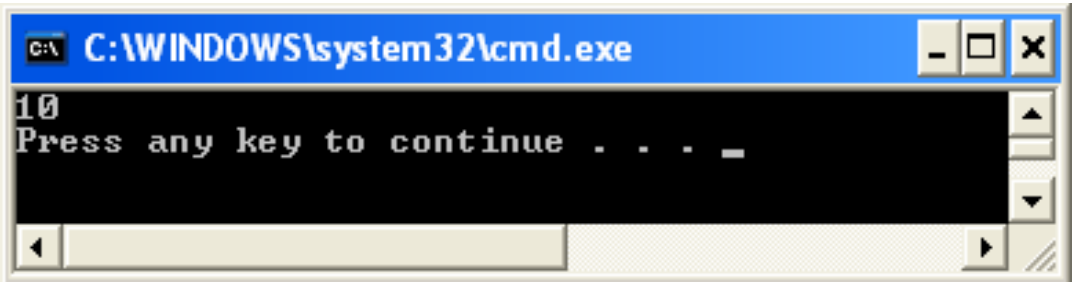
- Consider the following program.

```
#include<stdio.h>
void getValue(int);

void main(void){
  int a=0;
  getValue(a);
  printf("%d\n", a);
}

void getValue(int b){
  b = 10;
}
```



Limitation of Parameter Passing by Value

```
#include<stdio.h>
int getValue(int);

void main(void){
  int a=0;
  a=getValue(a);
  printf("%d\n", a);
}

int getValue(int b){
  b = 10;
  return b;
}
```

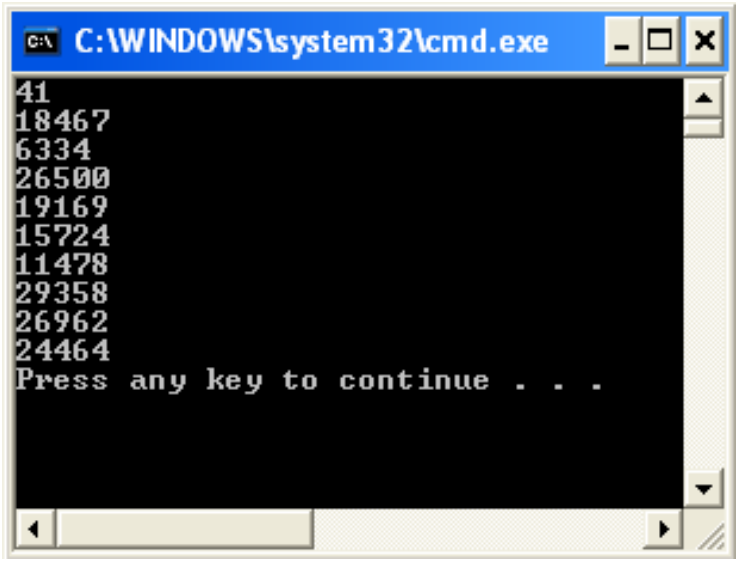


Passing array as parameter

- The name of an array is the pointer to the first element of the array. Due to this, passing an array as parameter to a function is considered as passing by reference.

Passing 1D array as parameter

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

void fillArray(int []);

void main(void){
   int intarray[10]={0}, i;

   fillArray(intarray);

   for(i=0;i<10;i++){
       printf("%d  ", intarray[i]);
   }

void fillArray(int a[]){
   int i;
   for (i=0;i<10;i++)
   a[i]=rand();
}
```



Passing 2D array as parameter

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

void fill2DArray(int [], int[]);

void main(void){
   int intarray[10][5]={0}, i, j;

   fill2DArray(intarray);

   for(i=0;i<10;i++){
        for(j=0;j<5;j++)
              printf("%d  ", intarray[i][j]);
   }

void fill2DArray(int a[10], int b[]){
   int i;
   for (i=0;i<10;i++){
        for(j=0;j<5;j++)
             a[i][j]=rand();
   }
}
```

Using Header File

- Function prototypes can also be put in a header file.Header files are files that have a **.h** extension.
- The header file can then be included at the beginning of our program.
- To include a user defined header file, type:

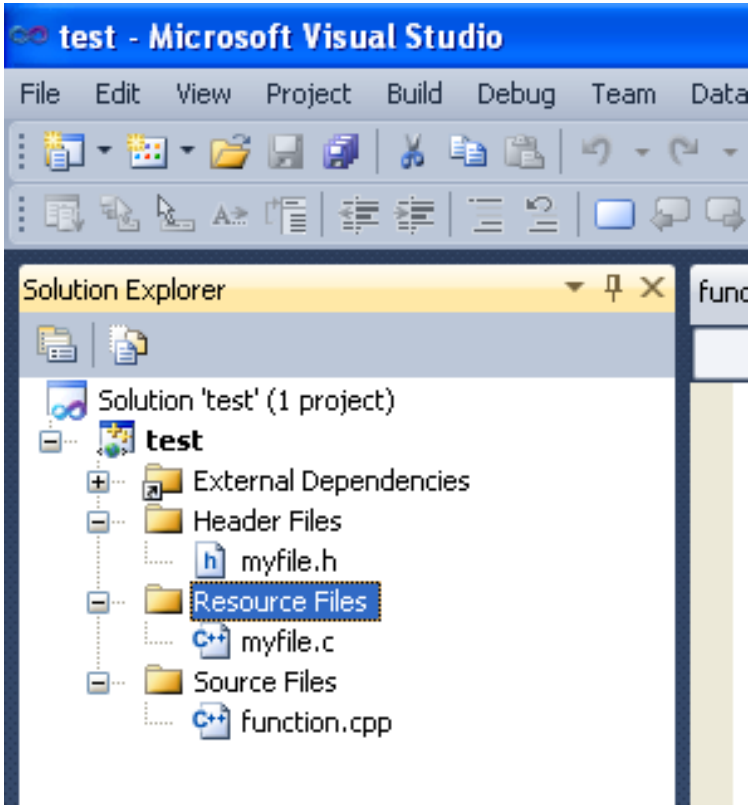
```
#include "header_file.h "
```

- Notice that instead of using < > as in the case of standard header files, we need to use " ". This will tell the compiler to search for the header file in the same directory as the program file instead of searching it in the directory where the standard library header files are stored.

Using Header File

```
#include<stdio.h>

void main(void){
printMessage();
}

void printMessage(void){
  printf("Hello!  Welcome!  Good Morning!\n");
}
```

Using Header File



Using Header File

- Write the following and save as myfile.c

```
#include<stdio.h>

void printMessage(void){
   printf("Hello!  Welcome!  Good Morning!\n");
}
```

- In your myfile.h file,

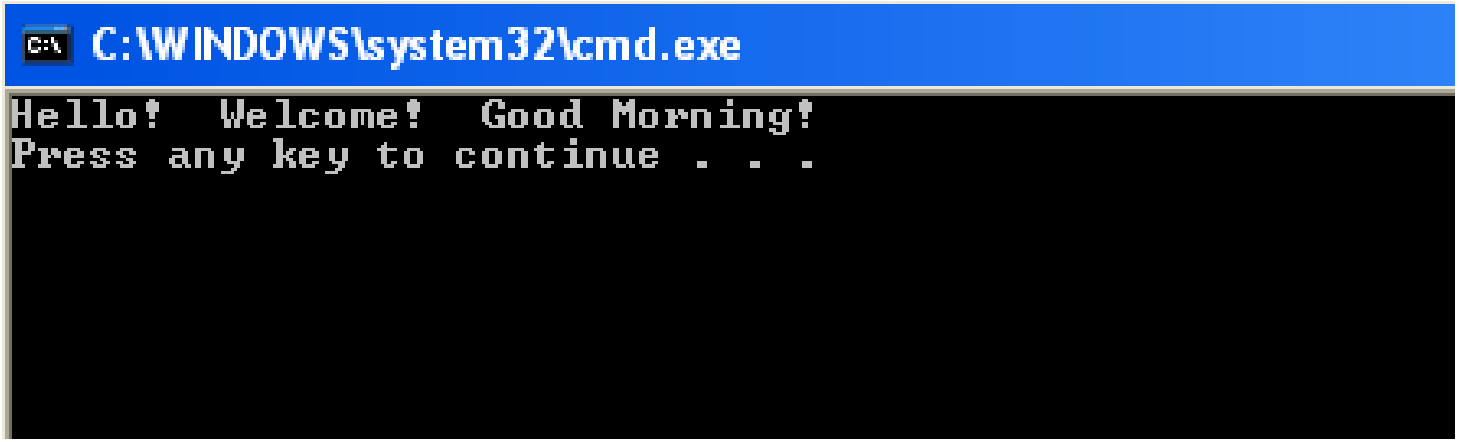
```
#include"myfile.c"

void printMessage(void);
}
```

- In your function.c file,

```
#include"myfile.h"

void main(void){
   printMessage();
}
```

Using Header File

- The output …



Macros

- A simple operation can also be specified by a preprocessing directive called a **macro**.

- Example:

```
#define degrees_C(x) (((x) - 32) * (5.0/9.0))

int main(void){
    double temp;
    printf("Enter temperature in degrees Fahrenheit: );
    scanf("%f", &temp);

    printf("%f degrees Centigrade\n", degrees_C(temp));

    return 0;
 }
```

Exercise

- Write macros to compute the following values.

1. Area of a square, $A = side^2$
2. Area of a rectangle, $A = side_1 \times side_2$
3. Area of a trapezoid, $A = \frac{1}{2} \times base \times (height_1 + height_2)$ where $height_1$ and $height_2$ are parallel

Declaration of Variables

- Up until now, we have learnt to declare our variables within the braces of segments (or a function) including the main.

- It is also possible to declare variables outside a function. These variables can be accessed by all functions throughout the program.

Local and Global Variables

- Local variables only exist within a function. After leaving the function, they are 'destroyed'. When the function is called again, they will be created (reassigned).
- Global variables can be accessed by any function within the program. While loading the program, memory locations are reserved for these variables and any function can access these variables for read and write (overwrite).
- If there exist a local variable and a global variable with the same name, the compiler will refer to the local variable.

Global variable - example

```
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);

int sum, num1, num2;

int main(void)
{
    /* initialise sum to 0 */
    initialise( );

    /* read num1 and num2 */
    getInputs( );

    calSum( );

    printf("Sum is %d\n",sum);
    return (0);
}

void initialise(void)
{
    sum = 0;
}

void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}

void calSum(void)
{
    sum = num1 + num2;
}
```

Global variables

```
Enter num1 and num2:
4
9
Sum is 13
Press any key to continue
```

Global variable – example explained

- In the previous example, no variables are passed between functions.
- Each function could have access to the global variables, hence having the right to read and write the value to it.
- Even though the use of global variables can simplify the code writing process (promising usage), it could also be dangerous at the same time.
- Since any function can have the right to overwrite the value in global variables, a function reading a value from a global variable can not be guaranteed about its validity.

Global variable – the dangerous side

```
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);

int sum, num1, num2;

int main(void)
{
    /* initialise sum to 0 */
    initialise( );

    /* read num1 and num2 */
    getInputs( );

    calSum( );

    printf("Sum is %d\n",sum);
    return(0);
}

void initialise(void)
{
    sum = 0;
}

void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}

void calSum(void)
{
    sum = num1 + num2;
    initialise();
}
```

```
Enter num1 and num2:
4
9
Sum is 0
Press any key to continue
```

Imagine what would be the output of this program if someone 'accidently' write the following function call inside calSum?

Storage Classes

- Storage class indicates the lifetime of a variable used in a program.
- Local variables only exist within a function by default. When calling a function repeatedly, we might want to
  - Start from scratch – re-initialise the variables
    - The storage class is ‘auto’
  - Continue where we left off – remember the last value
    - The storage class is ‘static’
- Another two storage classes (seldomly used)
  - register (ask to use hardware registers if available)
  - extern (to make local variables external i.e. global variables)

Auto storage class

- Variables with automatic storage duration are created when the block in which they are declared is entered, exist when the block is active and destroyed when the block is exited.

- The keyword **auto** explicitly declares variables of automatic storage duration. It is rarely used because when we declare a **local variable**, by default it has class storage of type **auto**.
  - `int a, b;` is the same as `auto int a, b;`

Static storage class

- Variables with static storage duration exist from the point at which the program begin execution.
- All the global variables are static, and all local variables and functions formal parameters are automatic by default. Since all global variables are static by default, in general it is not necessary to use the **static** keyword in their declarations.
- However the **static** keyword can be applied to a local variable so that the variable still exist even though the program has gone out of the function. As a result, whenever the program enters the function again, the value in the **static** variable still holds.

Auto - Example

```
#include <stdio.h>
void auto_example(void);

int main(void)
{
   int i;

   printf("Auto example:\n");

   auto_example( );
   auto_example( );
   auto_example( );

   return(0);

}

void auto_example(void)
{
   auto int num = 1;

   printf(" %d\n",num);
   num = num + 2;
}
```

```
Auto example:
1
1
1
Press any key to continue
```

Static - Example

```
#include <stdio.h>
void auto_example(void);

int main(void)
{
   int i;

   printf("Static example:\n");

   static_example( );
   static_example( );
   static_example( );

   return(0);

}

void static_example(void)
{
   static int num = 1;

   printf(" %d\n",num);
   num = num + 2;
}
```

```
Static example:
1
3
5
Press any key to continue
```

Summary

- In this chapter, you have learnt:
  - Standard vs User Define functions
  - Function prototype, function definition and function call
  - Formal vs Actual parameters
  - Parameter passing by values
  - Local vs Global variables
  - Auto vs Static storage class