**Reference** (**Alias**)

Normally when ever we create a variable in C++ program, a new memory location is created and some value is stored in that memory location. An example is given below

```
int x=35;
int y=x;
```

Variable x is created and assigned a value 35. Variable is y created as a copy of x. Variables x and y are two separate memory locations.

But some times in our program we need give new name to an existing variable. When we do that, we create a **Reference** or an **Alias** of an existing variable.

Rule: `DataType& NewVariableName=OldVariableName;`

Kindly note the use of ampersand operator (`&`) just after the `DataType`. Ampersand operator (`&`) works as **Reference** operator. An example is given below:

```
int x=35;
int& y=x;
```

Variable x is created and assigned a value 35. Variable is y created as an alias of x. Variable names x and y represent same memory location.

Variable y is a **Reference** of x (y is an **Alias** of x and vice versa). Any change in variable x will change variable y and vice versa. A **Reference** (**Alias**) can only be created for an existing variable.

**User Defined Function**

C++ programs consist of functions. Most important function in C++ program is the `main`() function. The Standard C++ Library has a reach collection of built-in functions, which are accessed, through header files. Here is a simple program that uses predefined function `sqrt`() used for calculating square root of non-negative value.

```
#include<iostream.h>
#include<math.h>
void main()
{
  for (int x=1; x<=4; x++)
  {
    double r=sqrt(x);
    cout<<r<<endl;
  }
}
```

Program consist of two functions, `main`() function and built-in function `sqrt`(). This program prints the square roots of the numbers 1 through 4. Each time the function `sqrt(x)` is called to calculate square root of x. Statement like `double r=sqrt(x);` is called **invoking** a function or **calling** a function. When a function is invoked, program control jumps to that particular function.

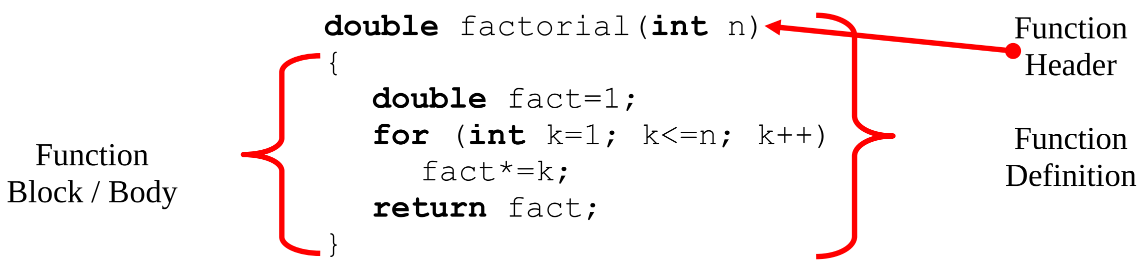
Running of the program

```
1
1.41421
1.73205
2
```

Great variety of function provided by the C++ library (through header files) is still insufficient for most programming tasks. Programmers need User Defined Function to break up a large program into small modules. C++ function (including user defined function) has two parts:

a) **Function Header**: it contains the **name** of the function, **return value** of the function and **optional** list of formal parameters, that is, it is not necessary that every function must have parameters. Function header is also called **Function Declarator**.

b) **Function Body**: it is the block after the function header. Function block contains statement that carries out action inside the function including the optional **`return`** statement. If the return value of a user defined function is **`void`**, then return statement is not required. Function Body is also called **Function Block**. Function header along with function block defines a complete function (also called **Function Definition**). An example of a user defined function is given below:



```
double factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   return fact;
}
```

Name of the function is `factorial`
Return value of the function is **`double`**
Function has a formal parameter **`int`** `n`
Block after the function header is the body of the function. Function header plus function block is the function definition. Function's **`return`** statement serves three (3) purposes:

a) It terminates the function
b) Returns a value to the calling function

Given below is the complete C++ program including user defined function `factorial()`.

```
#include<iostream.h>
double factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   return fact;
}
void main()
{
   int m;
   cout<<"Input integer? ";
   cin>>m;
   double f1=factorial(m);
   double f2=factorial(8);
   double f3=factorial(m+3);
   cout<<m<<"!="<<f1<<endl;
   cout<<8<<"!="<<f2<<endl;
   cout<<(m+3)<<"!="<<f3<<endl;
}
```

- Function **`void`** `main()` is the calling function.
- **`double`** `factorial()` is the called function. Called function is invoked from calling function.
- Formal parameter – **`int`** `n`
  Formal parameter is always a variable
- Function invocation is function name and parameter within parenthesis on the right hand side of assignment operator.
  ```
  double f1=factorial(m);
  double f2=factorial(8);
  double f3=factorial(m+3);
  ```
- Actual parameter:
  i) 1st call – `m` (variable)
  ii) 2nd call – `8` (constant)
  iii) 3rd call – `m+3`(expression)
  Actual parameter can be either variable or constant or expression

Running of the program

```
Input integer? 4
4!=24
8!=40320
7!=5040
```

**Explanation of the output**

Inputted value of m is 4. User defined function factorial() is invoked first time with actual parameter 4. Parameter used during function invocation is called **actual** parameter. Value stored in actual parameter (m) is transferred to **formal** parameter (**int** n). Parameter used in the function definition is called **formal** parameter. Body of the function calculates factorial of n. The **return** statement of the function factorial() terminates the function and returns calculated value to the calling function main(). Return value of the function is stored in variable **double** f1. Function factorial is invoked second time with actual parameter 8. Return value of the function is stored in variable **double** f2. Function factorial is invoked third time with actual parameter 2*m. Return value of the function is stored in variable **double** f3.

Return value of a function could be **void** as well. A user defined function with return value **void**, does not return a value to the calling function. Given below is the complete C++ program including user defined function factorial() whose return value is **void**.

```
#include<iostream.h>
void factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   cout<<n<<"!="<<fact<<endl;
}
void main()
{
   int m;
   cout<<"Input integer? ";
   cin>>m;
   factorial(m);
   factorial(8);
   factorial(m+3);
}
```

- Function **void** main() is the calling function.
- **void** factorial() is the called function and is invoked from main() function.
- Formal parameter – **int** n
  No **return** statement in the user defined function. Calculated value (fact) is displayed inside the user defined function.
- Function invocation is function name and parameter within parenthesis.
  factorial(m);
  factorial(8);
  factorial(m+3);
- Actual parameter:
  i) 1st call – m (variable)
  ii) 2nd call – 8 (constant)
  iii) 3rd call – m+3(expression)

**Running of the program produces following output**

```
Input integer? 4
4!=24
8!=40320
7!=5040
```

**Explanation of the output**

Inputted value of m is 4. User defined function factorial() is invoked first time with actual parameter 4. Body of the function factorial() calculates factorial of n and stored in an accumulator fact. Factorial of n is displayed inside the user defined function. Since there is no **return** statement, after the end of the function, program control returns to the calling function. Function factorial is invoked second time with actual parameter 8. Function factorial is invoked third time with actual parameter 2*m.

Why is the return value of the function factorial() is **double** or the variable fact used as an accumulator to calculate factorial is **double**? What happens if we use **int** in place of **double**? A four bytes integer data type (**int**) of C++ can store maximum 2147483647.

Factorial of 12 is 479001600 and is less than 2147483647. Factorial of 13 is 6227020800 and greater than 2147483647. This leads to memory overflow and therefore value stored in an integer accumulator is incorrect. Instead of using **`int`** as a data type of the accumulator if we use **`double`** (**`float`**) then memory overflow problem is solved since a variable of the type **`double`** (**`float`**) can store maximum $1.7x10^{308}$ ($3.4x10^{38}$).

Comparing factorial() function with return value **`double`** and with return value **`void`**.

| Return value of a function is not void | Return value of a function is void |
|---|---|
| `double factorial(int n)`<br>`{`<br>`   double fact=1;`<br>`   for (int k=1; k<=n; k++)`<br>`      fact*=k;`<br>`   return fact;`<br>`}` | `void factorial(int n)`<br>`{`<br>`   double fact=1;`<br>`   for (int k=1; k<=n; k++)`<br>`      fact*=k;`<br>`   cout<<fact<<endl;`<br>`}` |
| **Correct function invocation** | **Correct function invocation** |
| `Var=FunctName(ActualParam);`<br>`cout<<FunctName(ActualParam);`<br>`double f1=factorial(m);`<br>`cout<<factorial(m);` | `FunctName(ActualParam);`<br>`factorial(m);`<br>`factorial(8);`<br>`factorial(m+3);` |
| **Logically incorrect function invocation** | **Incorrect function invocation** |
| `FunctName(ActualParam);`<br>`factorial(m);`<br>This kind of function invocation is acceptable by the compiler. The return value of the function is discarded or ignored. A common example is use of built-in function `getch()`. Return value of `getch()` is **`int`**. But when we use `getch()`, it seems that the return value of the function is **`void`**. | `Var=FunctName(ActualParam);`<br>`cout<<FunctName(ActualParam);`<br>`double f1=factorial(m);`<br>`cout<<factorial(m);`<br>In both the cases, compiler will flag **syntax errors**. Since the return value of the function is **`void`**, return value of the function cannot be assigned to a variable name nor can return value of the function be displayed with `cout`. |

Now that we know that a user defined function, like `factorial()` may or may not have a return value. So which one (with return value or without return value) is better? The answer to this question depends on the usage of the function `factorial()`. If the function `factorial()` is only used for calculation factorial and displaying the value, then it does not really matter whether we defined the function whose return value is void or double. But suppose we want to calculate either $^nP_r$ or $^nC_r$.

$$^nP_r = \frac{n!}{(n-r)!} = factorial(n)/\,factorial(n-r);$$

$$^nC_r = \frac{n!}{r!(n-r)!} = factorial(n)/(\,factorial(r) *\, factorial(n-r));$$

To calculate either $^nP_r$ or $^nC_r$, first we need to calculate factorial and then only we can calculate either $^nP_r$ or $^nC_r$. So we will be using the return value of `factorial()` function to calculate either $^nP_r$ or $^nC_r$. This is only possible when return value of `factorial()` function is **`double`** (**`float`**). Generally, if the return value of a function has to be further used in any calculation in the calling function, then the return value of the function should not be **`void`**.

## Function Declaration (Function Prototype)

One method of defining the function is above the main() function (Calling function). This is the simplest arrangement and works well with small programs. But what happens when called function is defined after calling function? An example is given below:

```
#include<iostream.h>
void main()
{
   int m;
   cout<<"Input an integer? ";
   cin>>m;
   double f=factorial(m);
   cout<<m<<"!="<<f<<endl;
}
double factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   return fact;
}
```

Function **void** main() is the calling function. **double** factorial() is the called function and is defined after main() function. Called function is invoked from calling function. When the compiler is compiling the program, it will flag **syntax error** in the red highlighted line. Compiler at that point does know anything about factorial() function since factorial() is defined after the calling function. To remove the **syntax error** following statement will be added just after the header files:
**double** factorial(**int**);
In C++ this is known as **Function Declaration** or **Function Prototype**.

Corrected program is given below:

```
#include<iostream.h>
double factorial(int);
void main()
{
   int m;
   cout<<"Input an integer? ";
   cin>>m;
   double f=factorial(m);
   cout<<m<<"!="<<f<<endl;
}
double factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   return fact;
}
```

Blue highlighted line is the **Function Declaration** or **Function Prototype**. Pink highlighted line is the **Function Invocation**. When the compiler encounters the function declaration, it knows that somewhere in the block of the main() function, it will come across a function invocation which will match function declaration but the function definition will be after the main() function. This is another common practice to first declare the function and then define the function after the main() function. In this, function's declaration is separated from its definition and when coding large program this type of methodology is followed. C++ header files contain **Function Declarations** of C++ **Built-in** functions.

A function declaration contains Function name, Return value of the function, Data type of optional list of formal parameters and a Semi-colon at the end. Name of the formal parameters are not important in a function declaration. But if the formal parameter names are included in the function declaration, then they are ignored by the compiler.

## Function Definition

A function definition is the complete function, that is, header and the body. A function declaration must appear above any use of the function's name. But function's definition, when listed separately from the function's declaration, may appear anywhere outside the body or block of main() function.
Two unusual function definitions and the possible consequences:

| Return value of the function is **`double`** (not **`void`**) but without **`return`** statement | Return value of the function is **`void`** but with a **`return`** statement |
|---|---|

```
#include<iostream.h>
double factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
}
void main()
{
   int m;
   cout<<"Input integer? ";
   cin>>m;
   double f=factorial(m);
   cout<<f<<endl;
}
```

```
#include<iostream.h>
void factorial(int n)
{
   double fact=1;
   for (int k=1; k<=n; k++)
      fact*=k;
   cout<<fact<<endl;
   return fact;
}
void main()
{
   int m;
   cout<<"Input integer? ";
   cin>>m;
   factorial(m);
}
```

| | |
|---|---|
| Return value of the `factorial`() function is **`double`**. But return statement is missing from the function definition. When compiler compiles the program, it flags a **warning** (no syntax error), "Function should return a value". During the run-time the program will come to an abrupt halt. `cout<<f<<endl;` will not display any output on the screen or will display a garbage value on the screen. | Return value of the `factorial`() function is **`void`**. But function definition contains a **`return`** statement. When compiler compiles the program, it flags a **syntax error**, "`factorial`(**`int`**) cannot return a value". To remove the syntax error, **`return`** statement has to be removed from the function definition `factorial`() function and program needs to be recompiled. |

**Parameter**

Generally a C++ function has parameter or parameters. With help of parameters data is transferred between a calling function and called function. C++ function with parameters will need parameters at two places, at the point of function invocation (actual parameter) and at the point of function definition (formal parameter). Value of an actual parameter is transferred to formal parameter. An example is given below showing the use of parameter:

```
#include<iostream.h>
void moo(int a, int b)
{
   b*=a+=b;
   cout<<a<<','<<b<<endl;
}
void main()
{
   int x=5, y=8;
   moo(x, y);
   cout<<x<<','<<y<<endl;
}
```

**Explanation of the output**

User defined function `moo`() is invoke with actual parameters `x=5` and `y=8`. Values of actual parameter `x` and `y` are copied to formal parameter `a` and `b`. Inside the function `moo`(), formal parameters `a` and `b` are updated; function displays `13` and `104`. After the end of function, program control returns to the `main`() function. Since values of actual parameters were copied to formal parameters, changes in the formal parameters do not update actual parameters. So `main`() function displays `5` and `8`.

Running of the program produces following output

```
13,104
5,8
```

Since formal parameters a and b receive values as copy of actual parameter, these type of formal parameters are called **Value Parameter**.

A C++ function can have any number of parameters but return value is only one. So this is a kind of handicap in some cases. Suppose we want to write a C++ function to calculate AM and GM of two values and want to return both the values to calling function. But this is not possible since a function can return only one value. But an alternative solution exists where if it was possible to update actual parameter by changing formal parameters and this can be achieved through **Reference Parameter**. An example is given below to illustrate use of reference parameter.

```
#include<iostream.h>
void moo(int& a, int& b)
{
   a+=b;
   b*=a;
   cout<<a<<','<<b<<endl;
}
void main()
{
   int x=5, y=8;
   moo(x, y);
   cout<<x<<','<<y<<endl;
}
```

**Explanation of the output**
Note the changes in the formal parameters. An ampersand introduced between the data type and the formal parameter name. **Reference Parameter** is an **alias** of actual parameter. User defined function moo() is invoke with actual parameters x=5 and y=8. Formal parameter a is alias of x and formal parameter y is alias of b. Inside the function moo(), formal parameters a and b are updated and therefore x and y are also updated. Hence both the functions (moo() and main()) display 13 and 104 on the screen. Using reference parameter, a function can return more than one value.

Running of the program following output

```
13,104
13,104
```

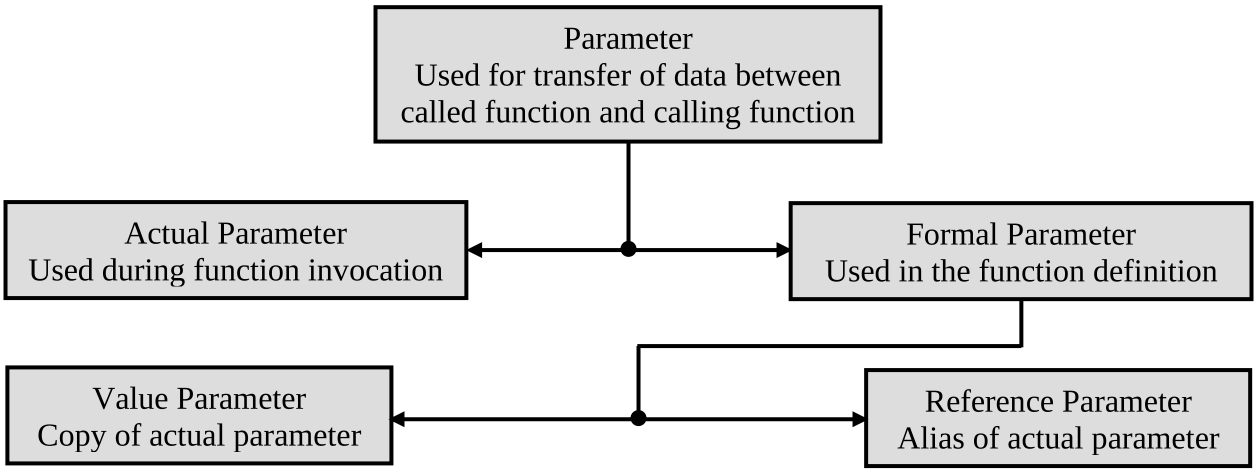
Program to calculate AM, GM and HM of two values using a function and reference parameter.

```
#include<iostream.h>
#include<math.h>
void mean(double x, double y, double& a, double& g, double& h)
{
   a=(x+y)/2;
   g=sqrt(x*y);
   h=2*x*y/(x+y);
}
void main()
{
   double num1, num2, am, gm, hm;
   cout<<"Input 1st value? "; cin>>num1;
   cout<<"Input 2nd value? "; cin>>num2;
   mean(num1, num2, am, gm, hm);
   cout<<"AM="<<am<<endl;
   cout<<"GM="<<gm<<endl;
   cout<<"HM="<<hm<<endl;
}
```

Running of the program following output

```
Input 1st value? 12.5
Input 2nd value? 14.7
AM=13.6
GM=13.5554
HM=13.511
```

Tree diagram of parameters


Parameter
Used for transfer of data between called function and calling function
Actual Parameter
Used during function invocation
Formal Parameter
Used in the function definition
Value Parameter
Copy of actual parameter
Reference Parameter
Alias of actual parameter

**Difference between Actual Parameter and Formal Parameter:**

| Actual Parameter | Formal Parameter |
|---|---|
| • Parameter used in function invocation | • Parameter created in the function header |
| • Actual parameter may be either be a variable or an expression or a constant | • Formal parameter is always a variable (or an alias) |

**Difference between Value Parameter and Reference Parameter:**

| Value Parameter | Reference Parameter |
|---|---|
| • Copy of actual parameter | • Alias of actual parameter |
| • Change in value parameter does not change actual parameter | • Change in reference parameter, updates actual parameter |
| • Transfer of data is one way, from calling function to called function | • Transfer of data is two ways, from calling function to called function and vice-versa |
| • Actual parameter may either be a variable or an expression or a constant | • Actual parameter can only be a variable, it cannot be constant or expression |

**Local Variable**

A variable created inside a block (block of **if-else**, block of **while** loop, block of **for** loop, block of **do-while** loop and function block) is called a **Local Variable**. A local variable is visible and accessible within that block and block nested below. An example of Local variables is given below:

```
#include<iostream.h>
void myfunct(int a, int b)
{
    int s=a+b;
    int p=a*b;
    cout<<s<<','<<p<<endl;
}
void main()
{
    int x=5, y=8;
    myfunct(x, y);
    cout<<x<<','<<y<<endl;
}
```

Variables x and y created in the main() function block is local to main() function. Variables x and y can only used in the main() function. Variables s and p created in the myfunct() function block is local to func() function block. If we try to use variables s and p in main() function (or if we try to use variables x and y in function myfunct()) then compiler will flag **Syntax Error**. Scope of formal parameter is local to a function, that is, formal parameters **int** a and **int** b are local to function myfunct().

Characteristics of **Local variable**:

a) Default value of a Local Variable is **garbage**
b) **Visible inside** the block and blocks nested blow
c) **Longevity** is as long as the **block is active**

**Global Variable**

A variable created immediately after the header files and before any block is called **Global Variable**. A global variable may be used in anywhere in the program and it is visible through out the program. Example of Global variable is given below:

```
#include<iostream.h>
int m=5;
void main()
{
   int n=30;
   n+=m; m+=n;
   cout<<m<<n<<endl;
   {
      int t=20;
      m+=t; n+=t;
      cout<<m<<n<<t<<endl;
   }
   n+=m; m+=n;
   cout<<m<<n<<endl;
}
```

- Variable `int m=5` is a global variable. Variable `m` can be used through out the program, in any block (inside outer block and inside inner block). Every time a global variable receives a value, its value gets updated.
- Variable `int n=30` is a local variable created in the outer block. Variable `n` can be used in the inner block also.
- Variable `int t=20` is a local variable created in the inner block. Variable `t` can only be used inside inner block. If we try to use variable `t` outside the inner block, C++ compiler will flag **Syntax Error**.

Characteristics of **Global Variable**:

a) Default value of a Global Variable is **Zero**
b) **Visible throughout** the program – main() function and all other user defined functions
c) **Longevity** is as long as the **program is active**

**Scope Resolution operator (::)**

As mentioned above, a global variable can be though out the program (in any block) but always this is not true. Inside a block, if there is a local variable and global variable having same name then the local variable is preferred to the global variable. An example is given below:

```
#include<iostream.h>
int m=100;
void main()
{
   int m=10;
   cout<<m<<endl;
}
```

Global variable `int m=100` will not be visible in the `main`() function block. Program display `10` since local variable `int m` has a value `10`. If we want to use both global variable and local variable inside the `main`() function block then we need to use **Scope Resolution** operator (**::**) with the global variable name.

```
#include<iostream.h>
int m=7;
void main()
{
   int m=8;
   ::m*=m;
   m*=::m;
   cout<<::m<<m<<endl;
}
```

Variable `int m=7` created after the header file is the global variable. Variable `int m=7` created in the `main`() function block is the local variable. Inside the `main`() function block scope resolution operator (**::**) is used with the global variable name. Running of the program displays `56` and `448` on the screen. Scope resolution operator (**::**) is used as a **unary** operator with a global variable name.

```
#include<iostream.h>
int m=5;
void foo()
{
   int m=50;
   m+=::m;
   ::m+=m;
   cout<<::m<<','<<m<<endl;
}
void main()
{
   int m=100;
   ::m+=m;
   m+=::m;
   cout<<::m<<','<<m<<endl;
   foo();
   m+=::m;
   ::m+=m;
   cout<<::m<<','<<m<<endl;
}
```

Running of the program

```
105,205
260,155
725,465
```

Variable **int** m=5 created after the header file is the global variable. Variable **int** m=100 created in the main() function block is local to main() function. Variable **int** m=50 created in function foo() is local to function foo(). To use both global variable and local variable (inside main() function, and inside function moo()) scope resolution operator (**::**)is used with the global variable name. Local variable **int** m of main() function cannot be used in the function moo() (or local variable **int** m created in moo() cannot be used in main() function).

**Function with Default Parameter**

So far we have seen that formal parameter obtains value from actual parameter. Therefore actual parameters list must match with formal parameter list. But in C++ it is possible to invoke C++ function with variable number of actual parameters. This is possible if we assign some default values to the formal parameters. Default values assigned to formal parameters is known as function with **Default Parameters**. An example is given below:

```
#include<iostream.h>
void print(char c='*', int n=5)
{
   for(int x=0; x<n; x++)
      cout<<c;
   cout<<endl;
}
void main()
{
   print();
   print('@');
   print('#', 10);
   print(65);
}
```

**Explanation of the output**

1st call of print(), no actual parameter, formal parameters obtain the default values. Displays *****. 2nd call of print(), default value of **char** c is replaced by actual parameter '@'. Displays @@@@@. 3rd call of print(), default value of **char** c is replaced by actual parameter '#' and default value of **int** n is replaced by 10. Displays ##########. 4th call of print(), default value of **char** c is replaced by 'A', ASCII code of 65. Displays AAAAA.

Running of the program following output

```
*****
@@@@@
##########
AAAAA
```

The function assigns default values to the formal parameters which does not have matching actual parameters in the function call, that is, default values of formal parameters are only relevant when there are no actual parameters. If actual parameters are present then values stored in actual parameters replace default values of the formal parameters.

If a function has default parameters, then the function's parameter list must show all the parameters with default values to the right of all the parameters that have no default values, that is, we must add default values to the formal parameters from **right to left**. We **cannot** provide default value to a particular parameter in the middle or in the beginning of the formal parameter list. Compiler flags **Syntax Error** if default values are not assigned from right to left.

Examples of **correct** function definitions with default parameters are given below (default values are assigned from right to left):

```
double simpleinterest(double p=10000, double r=0.1, double y=5)
{
   return p*r*y;
}
```

All the parameters are default parameters

```
double simpleinterest(double p, double r=0.1, double y=5)
{
   return p*r*y;
}
```

First two parameters from right are default parameters

```
double simpleinterest(double p, double r, double y=5)
{
   return p*r*y;
}
```

First parameter from right is default parameter

Examples of **incorrect** function definitions with default parameters are given below:

```
double simpleinterest(double p=10000, double r=0.1, double y)
{
   return p*r*y;
}
```

First two parameters from the left are default parameter.

```
double simpleinterest(double p, double r=0.1, double y)
{
   return p*r*y;
}
```

The middle parameter is the default parameter

```
double simpleinterest(double p=10000, double r, double y=5)
{
   return p*r*y;
}
```

Middle parameter is not a default parameter, where as the parameter from left and the parameter from the are default parameter

```
double simpleinterest(double p=10000, double r, double y)
{
   return p*r*y;
}
```

The first parameter from the left is the default parameter

## Function overloading

In C++ we can use same function name to perform variety of different task. In C++ this is known as function overloading. This is done by defining two or more functions having same name but differentiated on the basis of formal parameters, that is, either the formal parameter list must contain different number of parameters or there must be at least one position in the list of formal parameters where data type is different. The function will perform different operations depending on list of formal parameters. The correct function to be invoked is determined by checking the formal parameter list. **Return value of a function does not play any role in function overloading**. An example of overloaded functions is given below:

```
#include<iostream.h>
void add(int a, int b)
{
   int s=a+b;
   cout<<"Sum="<<s<<endl;
}
void add(double a, double b)
{
   double s=a+b;
   cout<<"Sum="<<s<<endl;
}
void add(double a, double b, double c)
{
   double s=a+b+c;
   cout<<"Sum="<<s<<endl;
}
void main()
{
   add(2.5, 3.7);
   add(4, 5);
   add(4, 5, 8);
   add(2.5, 4, 6);
   add(5.4, 7.2, 9);
}
```

First invocation of `add(2.5, 3.7)` finds **Exact Match** with second definition of `add(`**`double`**`, `**`double`**`)`. Second invocation of `add(4, 5)` finds **Exact Match** with first definition of `add(`**`int`**`, `**`int`**`)`. Third invocation of `add(4, 5, 8)` finds **Exact Match** with third definition of `add(`**`double`**`, `**`double`**`, `**`double`**`)`.
Fourth invocation of `add(2.5, 4, 6)` does not find exact match. Now compiler will use **Integral Promotion** to find a match, that is, internally actual parameters `4` and `6` will be converted to `4.0` and `6.0`. So a match will be found with third definition of `add(`**`double`**`, `**`double`**`, `**`double`**`)`.
Fifth invocation of add(`5.4, 7.2, 9`) finds match with third definition of `add(`**`double`**`, `**`double`**`, `**`double`**`)` through **Integral Promotion**.

The function selection from family functions during function call involves following steps:

1. Compiler first tries to find an **Exact Match** in which the list actual parameters matches exactly with list of formal parameters
2. If an exact match is not found, the compiler uses the **Integral Promotion** to convert actual parameters, such as, **`char`** promoted to **`int`** or **`int`** is promoted to **`double`** to find the match
3. When either of them fails then compiler tries to use the **Implicit Type Conversion** (**`int`** will be converted to **`char`** or **`double`** will be truncated to **`int`**) to convert actual parameters to find the match
4. If during the function call, compiler finds that there are two or more functions whose list of parameters matches with list of actual parameters, then the compiler flags a Syntax error called "Ambiguity error".

Suppose in the above example, a statement is added in the `main`() function as `add(10.2, 20)`;, what will the compiler do? No exact match will be found from the list of overloaded functions. Either the compiler promotes integer `20` to floating point `20` through **Integral Promotion** so that a match will be found with `add(`**`double`**`, `**`double`**`)` or the compiler convert floating point `10.2` to integer `10` through **Implicit type Conversion** so that a match will be found with `add(int, int)` and this lead to **Ambiguity Error**. The compiler will flag an error message 'Ambiguity between `add(int, int)` and `add(double, double)`'. To remove the syntax error we need two more function definitions of `add()` as:

```
void add(int a, double b)
{
   double s=a+b;
   cout<<"Sum="<<s<<endl;
}
```

```
void add(double a, int b)
{
   double s=a+b;
   cout<<"Sum="<<s<<endl;
}
```

Why two more definitions of `add()`? This is to remove any further ambiguity error in future. Now it will be perfectly alright to add following two statements:

```
   add(10.2, 20);
   add(20, 10.2);
```

in the main() function without compiler flagging any syntax error.

**Return by reference**

In C++ a function can also return by reference. In that case function name acts as an alias of one of the formal parameters. In this case formal parameters must be of the type reference. Since reference parameters are alias of actual parameters, therefore indirectly function name acts as an alias of one on the actual parameters. An example of function return by reference is given below:

```
int& getmin(int& x, int& y)
{
   return x<y ? x : y;
}
void main()
{
   int a=20, b=10;
   int lo=getmin(a, b);
   cout<<lo<<endl;
   getmin(a, b)=50;
   cout<<a<<','<<b<<endl;
   getmin(a, b)=100;
   cout<<a<<','<<b<<endl;
}
```

**Explanation of output**

Function `getmin()` returns minimum of the two actual parameters passed by reference. First invocation of `getmin()` return `10` and return value is stored in variable lo. Here the function is invoked like a normal function. Second invocation of `getmin()` is on the left hand side of the assignment operator. Function `getmin()` is an alias of formal parameter `y`, `y` is an alias of `b` and therefore `getmin()` is an alias of `b`. Therefore `b`'s value is updated to `50`. Therefore `20` and `50` are displayed. In the third invocation of `getmin()`, it is an alias of `a`, therefore `a`'s value is updated. So last the output is `100` and `50`.

Running of the program produces following output

```
10
20,50
100,50
```

```
int countdigit(int n)
{
   int count=0;
   while (n!=0)
   {
      count++;
      n/=10;
   }
   return count;
}
```

```
int sumofdigit(int n)
{
   int sum=0;
   while (n!=0)
   {
      sum+=n%10;
      n/=10;
   }
   return sum;
}
```

```
int reverseint(int n)
{
   int num=0;
   while (n!=0)
   {
      int digit=n%10;
      num=10*num+digit;
      n/=10;
   }
   return num;
}
```

```
int checkprime(int n)
{
   int x=2;
   int prime=1;
   while (x<n && prime==1)
      if (n%x==0)
         prime=0;
      else
         x++;
   return prime;
}
```

```
int checkarmstrong(int n)
{
   int sum=0, temp=n;
   while (n>0)
   {
      int digit=n%10;
      sum+=digit*digit*digit;
      n/=10;
   }
   return sum==temp;
}
```

```
int checkpalidrome(int n)
{
   int num=0, temp=n;
   while (n!=0)
   {
      int digit=n%10;
      num=10*num+digit;
      n/=10;
   }
   return temp==num;
}
```

```
int productofdigit(int n)
{
   int prod=1;
   while (n!=0)
   {
      int digit=n%10;
      prod*=digit;
      n/=10;
   }
   return prod;
}
```

```
int productofdigit(int n)
{
   int prod=1;
   while (n!=0)
   {
      int digit=n%10;
      if (digit!=0)
         prod*=digit;
      n/=10;
   }
   return prod;
}
```

```
// Displays Armstrong Nos
// between 1 and n
void generatearmstrong(int n)
{
   for (int k=1; k<=n; k++)
   {
      int sum=0, temp=k;
      while (temp>0)
      {
         int digit=temp%10;
         sum+=digit*digit*digit;
         temp/=10;
      }
      if (sum==k)
         cout<<k<<endl;
   }
}
```

```
// Displays Prime Nos between
// 2 and n
void generateprime(int n)
{
   for (int k=2; k<=n; k++)
   {
      int x=2, prime=1;
      while (x<k && prime==1)
      {
         if (k%x==0)
            prime=0;
         x++;
      }
      if (prime==1)
         cout<<k<<endl;
   }
}
```

```
//Display Prime between 2 & n
#include<iostream.h>
void sumofprime(int n)
{
   int sum=0;
   for (int k=2; k<=n; k++)
   {
      int x=2, prime=1;
      while (x<k && prime==1)
         if (k%x==0)
            prime=0;
         else
            x++;
      if (prime==1)
      {
         cout<<k<<endl;
         sum+=k;
      }
   }
   cout<<"Sum Of Prime="<<sum;
}
void main()
{
   int n;
   cout<<"Input n? "; cin>>n;
   sumofprime(n);
}
```

```
//Displays first n Prime Nos
#include<iostream.h>
void generateprime(int n)
{
   int k=2, count=0;
   while (count<n)
   {
      int x=2, prime=1;
      while (x<k && prime==1)
         if (k%x==0)
            prime=0;
         else
            x++;
      if (prime==1)
      {
         cout<<k<<endl;
         count++;
      }
      k++;
   }
}
void main()
{
   int n;
   cout<<"Input n? "; cin>>n;
   generateprime(n);
}
```

```
int hcf(int a, int b)
{
   int r;
   do
   {
      r=a%b;
      a=b;
      b=r;
   }
   while (r>0);
   return a;
}
```

```
int lcm(int a, int b)
{
   int r, p=a*b;
   do
   {
      r=a%b;
      a=b;
      b=r;
   }
   while (r>0);
   return p/a;
}
```

```
//Displays Fibonacci Prime Nos
void fibonacciprime(int n)
{
   int f1=1;
   int f2=1;
   for (int k=3; k<=n; k++)
   {
      int f3=f1+f2;
      cout<<f3;
      f1=f2;
      f2=f3;
      int x=2, pri=1;
      while (x<f3 && pri==1)
      {
         if (f3%x==0)
            pri=0;
         x++;
      }
      if (pri==1)
          cout<<"Prime"<<endl;
      else
         cout<<endl;
   }
}
```

```
//HCF and LCM of two numbers
void hcflcm(int a, int b)
{
   int p=a*b;
   int r;
   do
   {
      r=a%b;
      a=b;
      b=r;
   }
   while (r>0);
   cout<<"HCF="<<a<<endl;
   cout<<"LCM="<<(p/a)<<endl;
}
```

```
void checkfibo(int n)
{
   int f1=1, f2=1;
   while (f1+f2<=n)
   {
      int f3=f1+f2;
      f1=f2;
      f2=f3;
   }
   if (f2==n)
      cout<<"Fibonacci\n";
   else
      cout<<"Not Fibonacci\n";
}
```

```
// Fibonacci Nos between 1 & n
void genfibo1(int n)
{
   int f1=1, f2=1;
   cout<<f1<<endl<<f2<<endl;
   while (f1+f2<=n)
   {
      int f3=f1+f2;
      cout<<f3<<endl;
      f1=f2;
      f2=f3;
   }
}
```

```
// First n Fibonacci Nos
void genfibo2(int n)
{
   int f1=1, f2=1;
   cout<<f1<<endl<<f2<<endl;
   for (int x=3; x<=n; x++)
   {
      int f3=f1+f2;
      cout<<f3<<endl;
      f1=f2;
      f2=f3;
   }
}
```