



Modular Programming

Lengthier programs

- Prone to errors*
- tedious to locate and correct the errors*

To overcome this

Programs broken into a number of smaller logical components, each of which serves a specific task.

Advantages of modularization

- Reusability
- Debugging is easier
- Build library

Functions

- ◆ A **function** is a set of instructions to carryout a particular task.
- ◆ Using functions we can structure our programs in a more modular way.

Functions

- ◆ *Standard functions*
(library functions or built in functions)
- ◆ *User-defined functions*
Written by the user(programmer)

Defining a Function

✓ **Name**

- *You should give functions descriptive names*
- *Same rules as variable names, generally*

✓ **Return type**

- *Data type of the value returned to the part of the program that activated (called) the function.*

✓ **Parameter list**

- *A list of variables that hold the values being passed to the function*

✓ **Function body**

- *Statements enclosed in curly braces that perform the function's operations(tasks)*



The general form of a function definition

```
return_type function_name( parameter_definition )  
{  
    variable  declaration;  
    statement1;  
    statement2;  
    .  
    .  
    .  
    return(value_computed);  
}
```

Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
```

} Body


➔ int main()
{
 cout << "Hello from main";
 DisplayMessage(); // **FUNCTION CALL**
 cout << "Back in function main again.\n";
 return 0;
}

Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

} Body



Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
} Body
```

```
int main()
{
    ➡ cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    ➡ DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

} Body

Functions

Return type

Function name

Parameter List

Body

void **DisplayMessage(void)**

{

cout << "Hello from the function" ;

}

int main()

{

cout << "Hello from main";

DisplayMessage(); // FUNCTION CALL

cout << "Back in function main again.\n";

return 0;

}

Functions

Return type Function name Parameter List

→ void DisplayMessage(void)

{

 cout << "Hello from the function" ;

}

} Body

int main()

{

 cout << "Hello from main";

 DisplayMessage(); **// FUNCTION CALL**

 cout << "Back in function main again.\n";

 return 0;

}

Functions

Return type Function name Parameter List

void DisplayMessage(void)

→ {
 cout << "Hello from the function" ;
}

} Body

int main()
{
 cout << "Hello from main";
 DisplayMessage(); // **FUNCTION CALL**
 cout << "Back in function main again.\n";
 return 0;
}

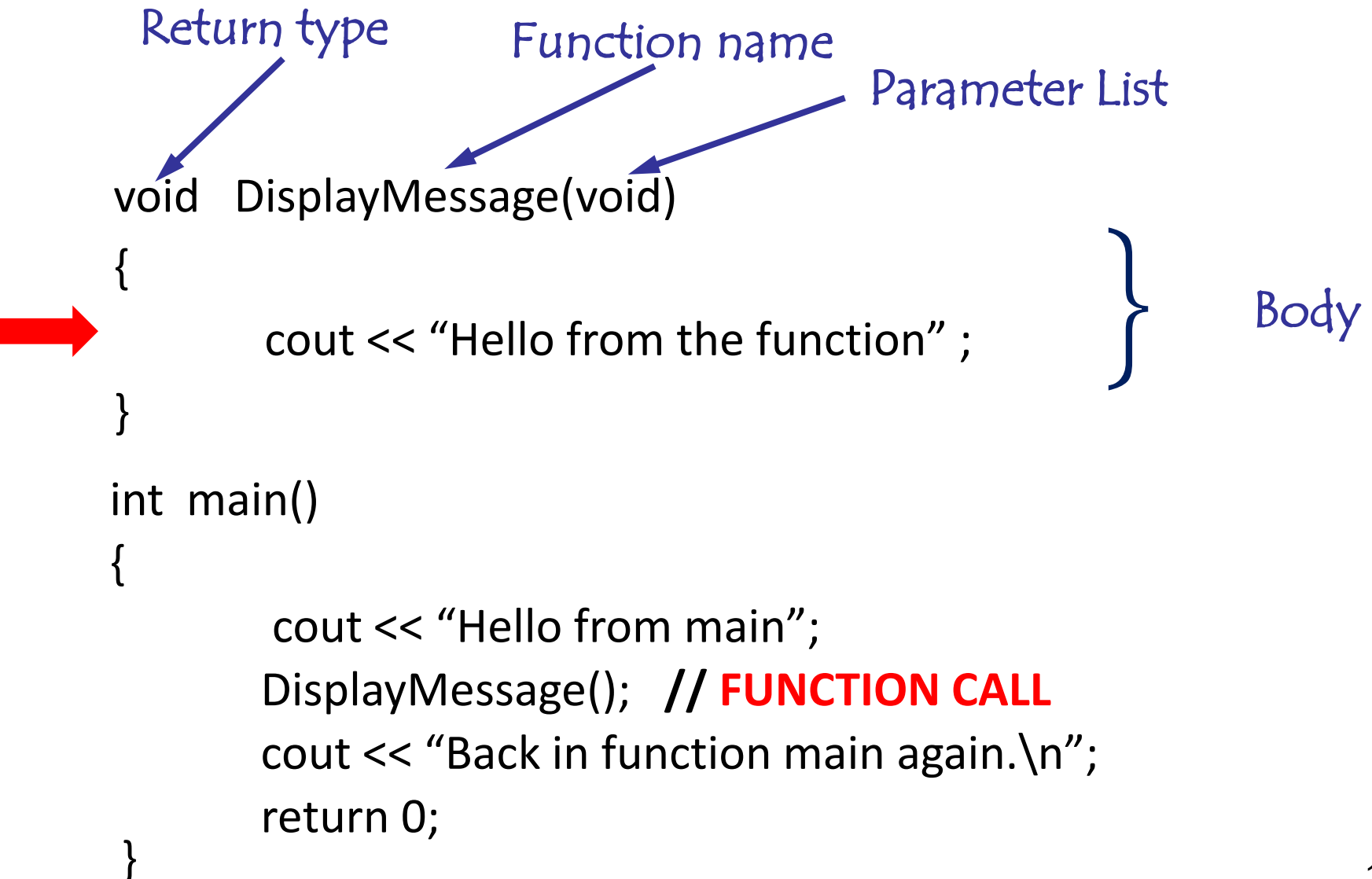
Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}

int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

Body




Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

Body



Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

Body

Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

} Body


➔

Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

} Body



Functions

Return type Function name Parameter List

```
void DisplayMessage(void)
{
    cout << "Hello from the function" ;
}
int main()
{
    cout << "Hello from main";
    DisplayMessage(); // FUNCTION CALL
    cout << "Back in function main again.\n";
    return 0;
}
```

} Body

➔

```
void DisplayMessage(void); //fn declaration
```

```
int main()  
{  
    cout << "Hello from main";  
    DisplayMessage(); // FUNCTION CALL  
    cout << "Back in function main again.\n";  
}
```

```
// FUNCTION DEFINITION
```

```
void DisplayMessage(void)  
{  
    cout << "Hello from the function" ;  
}
```

```
graph TD
    main["void main ()  
{  
  cout << \"I am starting in function main\\n\";  
  First ();  
  Second ();  
  cout << \"Back in function main again.\\n\";  
}"]
    First["void First (void)  
{  
  cout << \"I am now inside function First\\n\";  
}"]
    Second["void Second (void)  
{  
  cout << \"I am now inside function Second\\n\";  
}"]
    main -- blue arrow --> First
    First -- blue arrow --> main
    main -- red arrow --> Second
    Second -- red arrow --> main
```

void **First** (void)
{ cout << "I am now inside function First\\n";}

void **Second** (void)
{ cout << "I am now inside function Second\\n";
}

void main ()
{
 cout << "I am starting in function main\\n";
 First ();
 Second ();
 cout << "Back in function main again.\\n";
}

Arguments and parameters

- Both arguments and parameters are variables used in a *program* & *function*.
- Variables used in the *function reference* or *function call* are called as *arguments*. These are written within the parenthesis followed by the name of the function. They are also called *actual parameters*.
- Variables used in *function definition* are called parameters, They are also referred to as *formal parameters*.

```
3  int    sum( int    x , int    y );
4
5  int main()
6  {
7      int  a = 10 , b = 20 , c;
8      c = sum(a,b);
9      cout<<c;
10     return 0;
11 }
12
13 int sum( int  x, int  y )
14 {
15     return  x + y;
16 }
```

```
3  int    sum( int p , int q );
4
5  int main()
6  {
7      int  a = 10 , b = 20 , c;
8      c = sum(a,b);
9      cout<<c;
10     return 0;
11 }
12
13 int sum( int  x, int  y )
14 {
15     return  x + y;
16 }
```



```
3  int    sum( int , int );
4
5  int main()
6  {
7      int  a = 10 , b = 20 , c;
8      c = sum(a,b);
9      cout<<c;
10     return 0;
11 }
12
13 int sum( int  x, int  y )
14 {
15     return  x + y;
16 }
```

Functions- *points to note*

1. The parameter list must be separated by commas.

```
dispChar( int n, char c);
```

2. The parameter names do not need to be the same in the prototype declaration and the function definition.

3. The types must match the types of parameters in the function definition, in number and order.

```
void dispChar ( int n, char c);//proto-type
```

```
void dispChar ( int num, char letter) // function definition
```

```
{ cout<<" You have entered " << n<< "&" <<c; }
```

4. Use of parameter names in the declaration is optional.

```
void dispChar ( int , char);//proto-type
```

Functions- *points to note*

4. If the function has no formal parameters, the list is written as (void).
5. The return type is optional, when the function returns **int** type data.
6. The return type must be **void** if no value is returned.
7. When the declared types do not match with the types in the function definition, compiler will produce error.

Functions- Categories

1. Functions with no parameters and no return values.
2. Functions with parameters and no return values.
3. Functions with parameters and one return value.
4. Functions with no parameters but return a value.

Fn with No parameters & No return values

```
3 void dispPattern()  
4 {  
5     int i;  
6     for ( i = 1 ; i <= 20 ; i++ )  
7         cout<<"*";  
8 }  
9  
10 int main()  
11 {  
12     cout<<"\nfn to display a line of stars\n";  
13     dispPattern();  
14     return 0;  
15 }
```

Fn with parameters & No return values

```
3 void dispPattern( char c )
4 {
5     int i;
6     for ( i = 1 ; i <= 20 ; i++ )
7         cout<<c;
8 }
9
10 int main()
11 {
12     cout<<"\nfn to display a line of stars\n";
13     dispPattern( '#' );
14     dispPattern( '*' );
15     dispPattern( '@' );
16     return 0;
17 }
```

Fn with parameters & One return value

```
3  int  fnAdd( int x, int y )
4  {
5      int z;
6      z = x + y;
7      return(z);
8  }
9
10 int main()
11 {
12     int a=10, b=20, c;
13     c = fnAdd( a , b );
14     cout<<"Sum is "<< c;
15     return 0;
16 }
```

Fn with No parameters but A return value

```
3  int readNum()  
4  {  
5      int z;  
6      cin>>z;  
7      return z;  
8  }  
9  
10 int main()  
11 {  
12     int c;  
13     cout<<"\nEnter a number \n";  
14     c = readNum();  
15     cout<<"\nThe number read is "<<c;  
16     return 0;  
17 }
```


Fn that return multiple values

Will see later...

To be solved ...Functions

Write appropriate functions to

1. Find the factorial of a number 'n'.
2. Reverse a number 'n'.
3. Check whether the number 'n' is a palindrome.
4. Generate the Fibonacci series for given limit 'n'.
5. Check whether the number 'n' is prime.
6. Generate the prime series using the function written for prime check, for a given limit.

Question..

1. Write a function prototype for a function named *test()* that takes three arguments (two integer arguments, and a character argument), and returns a float value.

Ans: `float test(int , int , char);`

Questions..

2. What is the output of the following code?

```
3  int multiply(int x, y)
4  {
5      return x * y;
6  }
7  int main()
8  {
9      int a = 5, b = 6, c;
10     c = multiply( a , b );
11     return 0;
12 }
```

Error !

Functions- Parameter Passing

- Pass by value (call by value)
- Pass by reference (call by reference)

```
3 void incr( int );
4 int main()
5 {
6     int a = 10;
7     incr( a );
8     cout<<a;
9     return 0;
10 }
11 void incr( int a )
12 {
13     a++;
14 }
```

Questions..

3. What is the output of the following code?

```
3 void compute_double( int , int );
4
5 int main()
6 {
7     int p = 10, q = 15;
8     compute_double( p , q );
9     cout<<p<<"\t"<<q;
10    return 0;
11 }
12 void compute_double( int x, int y)
13 {
14     x = x * 2;
15     y = y * 2;
16 }
```

10

15

Functions- Parameter Passing

- Pass by value (call by value)
- **Pass by reference (call by reference)**

Reference variable

- *A reference variable is an alias, that is, another name for an already existing variable.*
- *Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.*

Syntax:

Datatype & *RefVariable* = *valVariable*;

Example: *int* &*r* = *i*; // *i* declared earlier

```

3  int main()
4  {
5      int a = 10;
6      int &ref_a = a;
7
8      cout<<a<<"\n";
9      cout<<ref_a<<"\n";
10
11     a *= 2;
12
13     cout<<a<<"\n";
14     cout<<ref_a<<"\n";
15
16     ref_a++;
17
18     cout<<a<<"\n";
19     cout<<ref_a<<"\n";
20 }

```

OUTPUT

10

10

20

20

21

21

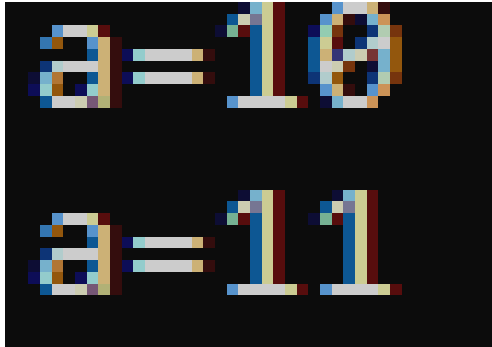
C++ References vs Pointers:

Three major differences between references and pointers are:

- ❖ *You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.*
- ❖ *Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.*
- ❖ *A reference must be initialized when it is created. Pointers can be initialized at any time.*
- ❖ *Datatype & RefVariable = valVariable;*

```
3 void increment( int& );
4 int main()
5 {
6     int a = 10;
7     increment( a );
8     cout<<a;
9     return 0;
10 }
11
12 void increment( int &x )
13 {
14     x++;
15 }
```

OUTPUT



a=10
a=11

```
3 void compute_double( int , int& );
4 int main()
5 {
6     int p = 10, q = 15;
7     compute_double( p , q );
8     cout<<p<<"\t"<<q;
9     return 0;
10 }
11 void compute_double( int x, int &y)
12 {
13     x = x * 2;
14     y = y * 2;
15 }
```

OUTPUT

10

30

Passing 1D-Array to a function

Sum = 100

```
3  int sum_array( int [ ], int );
4  int main()
5  {
6      int A[] = { 10, 20, 30, 40 }, n = 4;
7      int total = sum_array( A, n );
8      cout<<"Sum = "<<total;
9      return 0;
10 }
11
12 int sum_array( int arr[ ], int size )
13 {
14     int sum = 0;
15     for( int i = 0 ; i < size ; i++ )
16         sum += arr[i];
17     return sum;
18 }
```

```
3 void incr_array( int [ ], int );
4 int main()
5 {
6     int A[] = { 10, 20, 30, 40 }, n = 4;
7     incr_array( A, n );
8     for( int i = 0; i < n; i++ )
9         cout<<A[i]<<"\n";
10    return 0;
11 }
12
13 void incr_array( int arr[ ], int size )
14 {
15     for( int i = 0 ; i < size ; i++ )
16         arr[i]++;
17 }
```

```
11
21
31
41
```

Passing 1D-Array to Function

Rules to pass an array to a function

- *The function must be called by passing only the name of the array.*
- *In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.*
- *The function prototype must show that argument is an array.*

Passing 2D-Array to Function

```
3  int matrix_sum( int [ ][10], int, int );
4  int main()
5  {
6      int mat[][10] = {{1,2},{3,4},{5,6}};
7      int row_size=3, col_size = 2;
8
9      int total = matrix_sum(mat,row_size,col_size);
10     cout<<total;
11     return 0;
12 }

13 int matrix_sum( int M[][10], int r, int c )
14 {
15     int sum = 0;
16     for( int i = 0; i < r; i++ )
17         for( int j = 0; j < c; j++ )
18             sum += M[i][j];
19     return sum;
20 }
```

Passing 2D-Array to Function

Rules to pass a 2D- array to a function

- *The function must be called by passing only the array name.*
- *In the function definition, we must indicate that the array has two-dimensions by including two set of brackets.*
- *The size of the second dimension must be specified.*
- *The prototype declaration should be similar to function header.*

Function overloading, default arguments and inline functions

Function Overloading

- *Using the concept of function overloading - we can design a family of functions with one function name but with different argument lists.*
- *The function would perform different operations depending on the argument list in the function call.*

Example

```
void display ( int );
```

```
void display ( long );
```

```
void display ( char [ ] );
```

```
void display( int var )  
{  
    cout<<var<<endl;  
}
```

```
void display( long var )  
{  
    cout<<var<<endl;  
}
```

```
void display( char var[ ] )  
{  
    cout<<var<<endl;  
}
```

```
3 void display(int);
4 void display(long);
5 void display(char []);
6
7 int main( )
8 {
9     int i = 10;
10    long j = 600000;
11    char str[]="HELLO";
12    display( i );
13    display( j );
14    display( str );
15    return 0;
16 }
```

```
17 void display(char var[])
18 {
19     cout<<var<<endl;
20 }
21 void display(int var)
22 {
23     cout<<var<<endl;
24 }
25 void display(long var)
26 {
27     cout<<var<<endl;
28 }
```

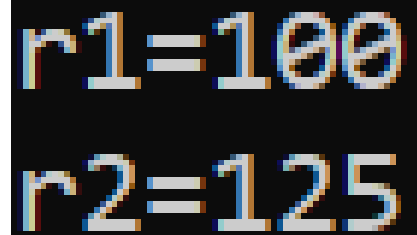
Default arguments

- ✓ *Default values are given in the function prototype declaration.*
- ✓ *Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters*
- ✓ *Only the trailing arguments can have default values and therefore we must add defaults from right to left.*
- ✓ *We cannot provide a default value to a particular argument in the middle of an argument list.*

Function with default arguments - Example

```
4 double power( double x, double n=2)
5 {
6     return pow(x,n);
7 }
8 int main( )
9 {
10     double a = 10, b = 5, c = 3;
11     double r1 = power( a );
12     double r2 = power( b , c );
13     cout<<"r1="<<r1;
14     cout<<"\nr2="<<r2;
15     return 0;
16 }
```

OUTPUT



r1=100
r2=125

Function with default arguments - Example-1

```
4 double power( double, double = 2);  
5 int main( )  
6 {  
7     double a = 10, b = 5, c = 3;  
8     double r1 = power( a );  
9     double r2 = power( b , c );  
10    cout<<"r1="<<r1;  
11    cout<<"\nr2="<<r2;  
12    return 0;  
13 }  
14  
15 double power( double x, double n)  
16 {  
17     return pow(x,n);  
18 }
```

OUTPUT

```
r1=100  
r2=125
```

Function with default arguments - Example-2

```
3  int add( int a, int b=2, int c=3 )
4  {
5      return a + b + c;
6  }
7  int main( )
8  {
9      int x = 10, y = 20, z = 30;
10     cout<< add( x ) <<"\n";
11     cout<< add( x, y ) <<"\n";
12     cout<< add( x, y, z );
13     return 0;
14 }
```

OUTPUT

```
15
33
60
```

```
int Display( int a=1, int b, int c )    // Invalid  
int Display( int a=1, int b=2, int c ) // Invalid  
int Display( int a, int b=2, int c )    // Invalid
```

Inline functions

- ❖ *An inline function is a function that is expanded in line when it is invoked(or called).*
- ❖ *The compiler replaces the function call with the corresponding function code (similar to the macros expansion).*

The inline functions are defined as follows:

```
inline function-header  
{  
    Function body  
}
```

Example:

```
inline double cube( double a )  
{  
    return a*a*a;  
}
```

The above inline function can be invoked by statements like

```
double c = cube(3.0);  
double d = cube( 2.5+1.5 );
```

- *All inline functions must be defined before they are called.*
- *It is easy to make a function as inline function. Just prefix the keyword `inline` to the function definition.*
- *The inline expansion makes a program run faster because the overhead of a function call and return is eliminated.*
- *However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called.*

```
inline float mul ( float x, float y )
{
    return x*y;
}
inline double div ( double p, double q )
{
    return p/q;
}
```

```
int main()
{
    float a = 12.5;
    float b = 10;
    cout << mul(a,b)<<"\n";
    cout << div(a,b)<<"\n";
    return 0;
}
```

Passing structure to a function

```
4 struct Emp
5 {
6     char name[50];
7     int age;
8     float salary;
9 };
10 void getData(Emp &p)
11 {
12     strcpy(p.name, "John");
13     p.age = 30;
14     p.salary = 30000;
15 }
16 void displayData( Emp &p )
17 {
18     cout<<"Name: "<<p.name<<endl;
19     cout<<"Age: "<<p.age<<endl;
20     cout<<"Salary: "<<p.salary;
21 }
```

```
22 int main()
23 {
24     Emp p;
25     getData(p);
26     displayData(p);
27     return 0;
28 }
```

```
Name: John
Age: 30
Salary: 30000
```


Passing structure to function

```
4 struct Emp
5 {
6     char name[50];
7     int age;
8     float salary;
9 };
10 void getData( Emp& );
11 void displayData( Emp& );
12
13 int main()
14 {
15     Emp p;
16     getData(p);
17     displayData(p);
18     return 0;
19 }
```

```
20 void getData(Emp &p)
21 {
22     strcpy(p.name, "John");
23     p.age = 30;
24     p.salary = 30000;
25 }
26 void displayData( Emp &p )
27 {
28     cout<<"Name: "<<p.name<<endl;
29     cout<<"Age: "<<p.age<<endl;
30     cout<<"Salary: "<<p.salary;
31 }
```