Function

* When a task is more complicated then we divide the task into smaller task, in which each task is responsible for a part of the task.

"Function is a block of code that performs a specific task. It is a self-contained unit of code that can be called from others parts of a program."

* A function has a name, a return type and a set of parameters.

* Any c++ program contain at least one function i,e main().

* Why we need function -

i) Reduce program size

ii) Error checking (debugging) and maintenance are easy.

iii) Reusability

iv) Easier to write simpler task

* There are basically two types of function -

i) Predefined function (Library function)

ii) Userdefined function

i) Predefined function -

These functions are pre-written and available to the user. We can use these function without writing the code for them. Such function are known as library function or predefined function.

→ In C++, predefined function are organised into seprate libraries. We only add the corresponding header file in which the function is defined.

→ For example, the header file 'iostream' contains I/o function and the header file 'string' contains string functions.

ii) Userdefined functions -

The user-defined functions are defined by the user according to its requirment.

To call a user-defined functions, we must first define it in our program.

Syntax

return type name of fn. (data type)

* "main ()" function is a user-defined function.

* To work with a user-defined function, we must consider three entities, these are -

i) Function declaration -

- Function declaration is also known as function prototype
- It is only the header of the function followed by a semicolon.
- It informs the compiler about three things i.e.

i) Name of the function

ii) No. and type of argument (parameter) received by the function.

iii) And the type of value returned by the argument.

Syntax -

returntype functionname (parameter list)

ii) Function definition -

→ Function definition consist of the whole description and code of the function.

→ It tells us about what function is doing, what are its input and what are its output.

→ It is made of two sections: the function header and the function body.

```
return type functionName (arguments)  ⟶ function header
{
    ----
    ----          // Function body
    ----
return value;
```

*ii) Function call -

→ When the function get called by the calling function then that is called, function call.

Syntax

Function Name (Parameter list)

* Program for representing function elements.

```
#include<iostream>
using namespace std;
int add (int a, int b);  // Function declaration
int main()
{
   int x = 2;
   int y = 3;
   int sum = add (x, y);  // Function call
   cout << "Sum of x and y is " << sum << endl;
   return 0;
}
int add (int a, int b)  // Function definition
 {
    return a+b;
 }
```

* Category of function based on argument and return.

i) Function with no argument and no return type.

Eg:-
```
#include <iostream>
using namespace std;

void sum(void);

int main()
{
    sum();
}

void sum()
{
    int a = 5, b = 7;
        s = a+b;
    cout << "sum = " << s ;
}
```

ii) Function with no argument but return value -

```
#include <iostream>
using namespace std;

int sum(void);

int main()
{
    int s;
    s = sum();
    cout << "Sum=" << s;
}

int sum()
{ int a = 10, b = 20, sum = 0;
    sum = a+b;
} return sum;
```

iii) Function with argument but no return value -

```
#include <iostream>
using namespace std;

void printSum(int a, int b);

int main()
{
    int x = 2;
    int y = 3;
    printSum(x, y);
}

void printSum(int a, int b)
{
    cout << "Sum = " << a+b;
}
```

iv) Function with argument and return value -

```
#include <iostream>
using namespace std;

int sum(int a, int b);
int main()
{
    int a = 5, b = 10, s = 0;
    s = sum(a, b);
    cout << "Sum = " << s;
}
int sum(int x, int y);
{
    return x+y;
}
```

Q. Program to find power of given base and exponent.

```
#include <iostream>
using namespace std;
int power(int, int); // Function prototype (declaration)
int main()
{
   int base = 2, exponent = 5;
    int result = 0;
    result = power(base, exponent); // Function call
       cout << result;
}
int power(int b, int e)      // Function definition
{
    int ans = 1;
    for(int i = 1; i <= e; i++)
    {
        ans = ans * b;
    }
    return ans;
}
```

→ Another way of writing this function -

```
#include <iostream>
using namespace std;
int power(int b, int e)  // Here we don't need to use a fun. prototype, because we define the function definition before we call it in the main() function.
{
   int ans = 1;
   for(int i=1; i<=e; i++)
   {
       ans = ans * b;
   }
   return ans;
}
int main()
{
   int base=2, exponent = 5, result = 0;
      result = power(base, exponent);
      cout<< result;
}
```

* Calling of Function -

i) Call by value -

→ Value of actual parameters are copied in formal parameters.

→ If any changes done to formal parameters in functi they will not modify actual parameters.

Eg:-

```
Void swap (int a, int b) // Formal parameter
{
    cout << a << " " << b << endl;   → a=10, b=20
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << a << " " << b << endl;   → a=20, b=10
}
int main()
{
    int x = 10, y = 20;
    swap (x, y); // Actual parameter we passing
    cout << x << " " << y;   → x=10, y=20
    return 0;
}
```

ii) Call by address-

→ Address of actual parameters are passed.

→ Formal parameters must be pointers.

→ Formal parameters can indirectly access actual parameter.

→ Changes done using formal parameters will reflect in actual parameters.

```
Void swap ( int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int main()
{
    int a = 10, b = 20;
    swap (&a, &b);
    cout << a << b;        → a = 20, b = 10
}
```

iii) Call by reference.

→ Actual parameters are passed as reference.

→ Formal parameters can directly access actual parameters.

→ Function call is converted into inline function, if not possible it will become call by address.

→ Reference don't take extra memory.

→ Syntax is same as call by value except formal parameters are references.

```
void swap( int &a , int &b)
  {
    int temp;
     temp = a;
     a = b;
     b = temp;
  }
int main()
 {
    int x=10, y=20;
      swap (x, y);
    cout << x << y;   →   x = 20, y=10
 }
```

* Inline function - An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instruction in memory.

* Return by reference -

→ A function can return reference.

→ It should not return reference of its local variable

→ It can return formal parameters if they are reference

```
int & fun (int &x)
 {
    return x;
 }

int main()
 {
    int a=10;
    fun(a) =25; // Here fun(a) becomes 'x' due to return by reference
    cout<<a<<; → a:25;
 }
```

* Return by address -

→ A function can return address of memory.

→ It should not return address of local variables which will be disposed after function ends.

→ It can return address of memory allocated in heap.

```
int * fun()
 {
    int *P = new int [5];
    for(int i= 0; i<5; i++)
        P[i] = 5 * i;
    cout<<p<<endl;
    return P;
 }
int main()
 {  int *q = fun();
    for(int i=0; i<5; i++)
       Cout<<q[i];
 }
```

* Function overloading -

→ Function overloading is a feature that allows multiple functions to have the same name but with different parameter list.

→ This means that we can use the same function name to perform different operations depending on the input arguments.

→ In function overloading, function name should be the same and the argument should be different.

→ Return type is not consider in overloading.

→ If two functions with same name and same parameter but different return type then they are not overloaded function.

Eg:- `int fun(int x, int y);`
`float fun(int x, int y);`

They are not overloaded fn.

Compiler will treat both as same function.

Eg.
```
int sum(int a, int b);
float sum(float a, float b);
int sum(int a, int b, int c);
```

Overloaded function

Compiler will treat every fun. differently

Eg:-

```
#include <iostream>
using name space std;
int main sum (int a, int b)
{
    return a+b;
}
float sum (float a, float b)
{
    return a+b;
}
int sum (int a, int b, int c)
{
    return a+b+c;
}
int main()
{
    cout << sum (10, 5) << endl;
    cout << sum (12.5f, 3.4f) << endl;
    cout << sum (10, 20, 3) << endl;
}
```

* Default Arguments

→ Default arguments are a feature that allows you to specify default values for function parameters.

→ If a default value is provided for a parameter, that parameter becomes optional, and can be omitted when the function is called.

→ Function with default argument can be called with various no. of argument.

→ Default values to parameter must be given from right side ~~fo~~ without skipping.

→ Default argument are much useful in constructors and defining overloaded function.

Eg:-
```
int add (int x, int y, int z = 0)   → Here 'z' is a
{                                       default Argument
    return x+y+z;
}

int main()
{
    int c = add(2, 5);
    int d = add(2, 5, 8);
    int e = add(2, 5, 0);

    cout << c << d << e;        o/p - 7, 15, 7
}
```

Eg:-
```
#include <iostream>
using namespace std;
int max(int a=0, int b=0, int c=0)
{
    return a>b && a>c ? a : b>c ? b : c ;
}
int main()
{
    cout<<max();          → 0
    cout<<max(10);        → 10
    cout<<max(10,13);     → 13
    cout<<max(10,13,15);  → 15
}
```

* Function Template -

→ Function templates are a powerful feature that allows you to create generic functions that can operate on multiple types of data.

" A function template is a function that is defined with generic parameter types, which are replaced with specific type when the function is called".

Example:

Suppose, if we need a function that find max^m of two numbers of int data type, float and double then we will have to create three different function to find maximum. But using function template we can find max^m for all data types by creating just one function.

```
template < class t >        → It will work for
t max( t x, t y)                all data type
 {
   if (x > y)
      return x;
   else
      return y;
 }
int main ()
{
   int c = max (10, 5);
   float d = max (10.5f, 6.9f);
   cout << c << d;
}
```