Function

Advantages of function

1. Code Reusability:

```
int add (int a, b)
{   int c
    return a+b;
}

main ( )
{
  c = add (10, 20);
  ---
  d = add (40, 5);
}
```

function declaration / prototype.

(2) Minimizing memory space
(3) Reduce number of lines of code
4) Code easily maintenable, Readability

Declaration, Definition and call

```
#include <iostream>
int add (int a, int b)
{
    return a+b;
}
```

/ Definition

```
int main()
{
    int c;
    c = add(10, 15); // function call.
    cout << c;
    return 0;
}
```

---

if we don't declare here function then it will give compile time error, and if we declare

```
#include <iostream>
int add(int a, int b); // Declaration
int main()
{
    int c;
    c = add(10, 15); // Function Call
    cout << c;
    return 0; // we can return any integer number
}

int add(int a, int b)
{
    return a+b; // Function Definition
}
```

structure of function

~~return function~~

optional

```
return_type function_name ( Parameter_type variable_name,... )
{
    // Logic
    return value;
}
```

Argument

→ Actual argument

→ formal argument

```
c = add(10, 15);
```

```
int add(int a, int b);
```

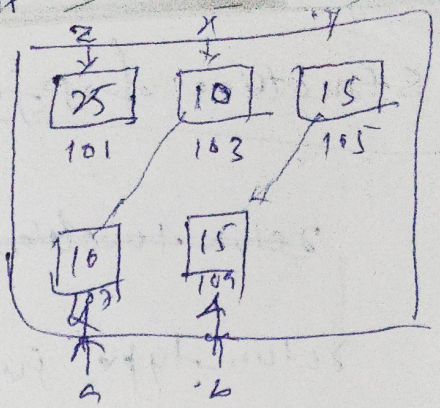
- * int, char, float, double pre-defined data type
- * class, struct, enum, these are user-defined data type.

Passing Argument to function

→ Passing by Value

```
int add(int a, int b)
{
    return a+b;
}
```

if we do a = 20 it will not change original value of x




```
int main( )
{
    int z, x=10, y=15;
    z = add(x, y) ;   // Passing by Value
    cout<<z<<endl;
    return 0;
}
```

→ Pass by Reference

// Adv. → no extra m/m allocation happens unlike in pass by value.

```
int add(int &a, int &b)
{
        return a + b ;
}
```

// return by Value

Disadv.
if a = 2x defined in fn. defn then it will change the original value.
If we will do a = 20 then it will change original value of x

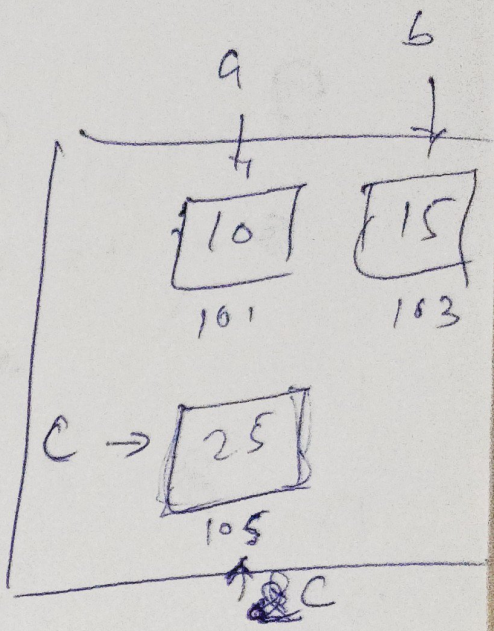
```
int main( )
{
    int z, x=10, y=15
    z = add(x, y);
    cout<<z<<endl;
    return 0;
}
```

z	x	y
25	10	15
101	103	105
	a	b

Output → 25

→ Return by Reference

```
int & add (int a, int b)
{
    int c;
    c = a+b;
    return c;
}

int main( )
{
    int &c = add(10, 15);
    cout << c << endl;   // 25
    return 0;            // &c
}
```

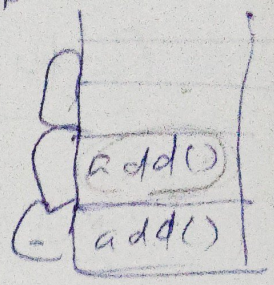


Inline function

reduces function calling overhead.

```
#include <iostream>
inline int add(int a, int b)
{
    return a+b;
}
```

Back-track memory



stack

// internally C compiler manage the stack

```
int main( )
{
    int c, x;
    c = add(10, 15);
    cout << c << endl;
    x = add(25, 30);
    Cout << x << endl;
    return 0;
}
```

Compile

```
int main( )
{
    int c, x;
    C = 10 + 15;
    cout << c << endl;
    x = 25 + 30;
    Cout << x << endl;
    return 0;
}
```

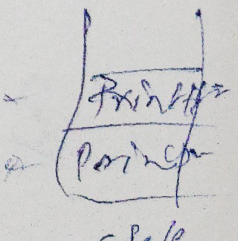
another example inline function

inline.cpp

```
#include <iostream>

void printHello( )
{
        cout << " Hello World ";
        cout << " Good bye ";
}

inline void printWellcome( )
{
        cout << " Wellcome to India " << endl;
        cout << " Good bye ";
}

main( )
{
        printHello( );
        printWellcome( );
        return 0;
        printHello( );
        printWellcome( );
        return 0;
}
```

Compile ↓p inline.out

```
#include <iostream>
void printHello ( )
{
    cout << "Hello world";
    cout << "Good Bye";
}

int main ( )
{
    PrintHello ( );
    cout << "Wellcome to India" << endl;
    cout << "Good bye";
    PrintHello ( );
    Cout << "Wellcome to India" << endl;
    cout << "Good bye";
    return 0;
}
```



// Run

Hello world
Good Byec
Wellcome to India
Good bye

Hello World
Good bye
Welcome to India
Good bye

Remember → Inline is only a request to the compiler, not a command.
The compiler can ignore the request for inlining.

Adv. of Inline Function :- Reduce function calling overhead → It may increases efficiency if it is small.

Disadvantage :- ~~Replacement~~ of compile
→ compiler replaces function with code which increases compiler overhead.
→ Executable file size will be increased means memory allocation will be more.

The compiler may not perform inlining in such circumstances:-
(i) If a function contain a loop. (for, while, do-while)
(ii) If a function is recursive
(iii) If a function contains static variables
(iv) If a function return type is other than void, and the return statement doesn't exist in a fn. body.
(v) If a fn contains a switch or goto statement

Function - Overloading in c++ → used for code reusability same m/n

It helps us to get diff behavior with the same name

↓ example of polymorphism feature in c++

→ If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.

→ Parameters conditions for function overloading follows any one or more than one of the following

(i) add (int a, int b)
add (double a, double b) } Parameters should have a different type

inline.obj
inline.cpp

Eg:-
```
void add (int a, int b)
{
cout << "sum = " << (a+b);
}
void add (double a, double b)
{
cout << endl << "sum = " << (a+b);
}
```

```
int main()
{
  add (10, 2)
  add (5.3, 6.2);
  return 0;
}
```

Output : Sum = 12
Sum = 11.5

(ii) Parameters should have a different number.

```
add (int a, int b)
add (int a, int b, int c)
```

E.g>

```
void add (int a, int b)
{
cout << "sum = " << (a+b);
}
void add (int a, int b, int c)
{
  cout << endl << "sum = " << (a+b+c);
}
int main()
{
  add(10, 2);
  add (5, 6, 4);
  return 0;
}
```

O/P : Sum = 12
sum = 15

(iii) Parameters should have a different sequence of parameters

```
add (int a, double b)
add (double a, int b)
```

e.g>

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

  int main ()
    {
       add (10, 2.5);
        add (5.5, 6);
          return 0;
    }
```

O/P:

```
Sum = 12.5
Sum = 11.5
```

Function Overloading

→ More than one function with same name but different parameter different sequence of parameter then it is called function overloading

Ex → 1) function add() with no parameter

```
int add( )
{
        int a=10 , b=15;
        return a+b;
}
```

2) Function add() with two int parameter

```
int add(int a, int b)
{
    return a+b;
}
```

3) Function add() with one int and one double Prameter

```
double add(int a, double b)
{
    return a+b;
}
```

4) Function add with one double and one int Paremter With different Sequence

```
double add(double a, int b)
{
    return a+b;
}
```

Q-> ~~Write a Program~~ and create a ~~Calculator with func~~ functionality

Q-> W.A.P. to add any two number ~~with~~ Using function overloading.

Q: WAP to add any two number using function overloading.

```
class Calculator
{
public:
    int add(int a, int b)
    {
        return a+b;
    }
    double add(double a, double b)
    {
        return a+b;
    }
    double add(int a, double b)
    {
        return a+b;
    }
    double add(double a, int b)
    {
        return a+b;
    }
};
```

```
Void main( )
{
    Calculator c;
    cout<<"10 + 5 is:"<<add(10, 5);
    Cout<<"10 + 5 is:"<<c.add(10, 5);
    Cout<<"10.5 + 5 is:"<<c.add(10.5, 5);
    Cout<<"5 + 10.5 is:"<<c.add(5, 10.5);
    Cout<<"5.6 + 6.8 is:"<<c.add(5.6+6.8);
    return 0;
}
```

~~advantage of function~~