Introduction to C++

Unit-1

- Difference between C and C++
- Evolution of C++
- Programming Paradigms
- Object Oriented Programming and its advantages
- Input/Output in C++
  - ❖ pre-defined streams
  - ❖ Stream classes
  - ❖ Scope access operator
  - ❖ Namespace
  - ❖ Memory management operators
- Functions
  - ❖ Passing arguments
  - ❖ Return by reference
  - ❖ Returning more values by reference
  - ❖ Default arguments
  - ❖ Const arguments
  - ❖ Inline functions
  - ❖ Function overloading

Difference between C and C++

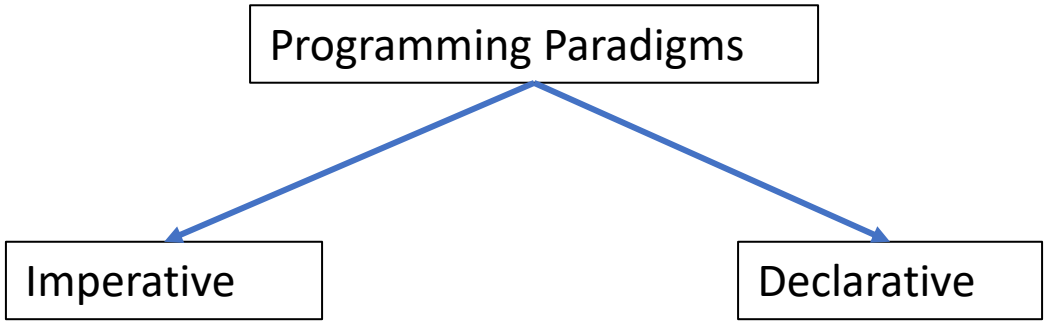
C	**C++**
It is a structured / procedure oriented language	It is an Object-oriented language
It doesn’t support Encapsulation, inheritance, polymorphism.	It supports Encapsulation, Inheritance, polymorphism.
It doesn’t support templates	It supports templates
It uses library functions for input and output	It uses input stream and output stream classes for input and output

C Program	**C++ Program**
#include<stdio.h> int main(){ printf("Hello, Have a Good Day!\n"); return 0; }	#include<iostream> *using namespace std;* int main(){ cout << "Hello, Have a Good Day!"<<endl; return 0; }
printf is an output function defined in stdio.h	**cout** is an object of **ostream** class which is defined in iostream << - extraction operator

C Program	C++ Program
#include<stdio.h> int main(){ int a, b, c; printf("Enter a and b values\n"); scanf("%d%d", &a, &b); c=a+b; printf("The sum of %d and %d is %d",a,b,c); return 0; }	#include<iostream> *using namespace std;* int main(){ int a, b, c; cout << “Enter a and b values” << endl; cin >> a >> b; c = a + b; cout<<“The sum of ”<<a<<“ and “<<b<<“ is “<<c; return 0; }
scanf is an input function defined in stdio.h printf is an output function defined in stdio.h	**cin** is an object of **istream** class which is defined in iostream **cout** is an object of **ostream** class which is defined in iostream << - extraction operator >> - Insertion operator

Programming Paradigms

- ***Programming paradigms*** are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.
- Certain paradigms are better suited for certain types of problems, so it makes sense to use different paradigms for different kinds of projects.



the programmer instructs the machine ***how to compute it***. Which changes the state.

the programmer merely ***declares*** properties of the ***desired result***, but not how to compute it.

Programming Paradigms(contd..)

- ***Imperative programming*** consists of sets of detailed instructions that are given to the computer to execute in a given order.
- Imperative programming focuses on describing *how* a program operates, step by step.

For example , you want to bake a cake

```
1- Pour flour in a bowl
2- Pour a couple eggs in the same bowl
3- Pour some milk in the same bowl
4- Mix the ingredients
5- Pour the mix in a mold
6- Cook for 35 minutes
7- Let chill
```

- ***Procedural programming*** is a derivation of imperative programming, adding to it the feature of functions (also known as "procedures" or "subroutines").
- In procedural programming, the user subdivide the program execution into functions, as a way of improving modularity and organization.

```
function pourIngredients() {
    - Pour flour in a bowl
    - Pour a couple eggs in the same bowl
    - Pour some milk in the same bowl
}
function mixAndTransferToMold() {
    - Mix the ingredients
    - Pour the mix in a mold
}
function cookAndLetChill() {
    - Cook for 35 minutes
    - Let chill
}
pourIngredients()
mixAndTransferToMold()
cookAndLetChill()
```

Programming Paradigms(contd..)

- In ***functional programming***, functions are treated as **first-class citizens**, meaning that they can be assigned to variables, passed as arguments, and returned from other functions.

- Another key concept is the idea of **pure functions**. A **pure** function is one that relies only on its inputs to generate its result. And given the same input, it will always produce the same result.

Imperative programming code

```
const nums = [1,4,3,6,7,8,9,2]
const result = [] // External variable
for (int i = 0; i < nums.length; i++) {
          if (nums[i] > 5)
            result[k++]=nums[i])
}
console.log(result) // Output: [ 6, 7, 8, 9 ]
```

functional programming code

```
const nums = [1,4,3,6,7,8,9,2]
function filterNums(){
          const result = [] // Internal variable
          for (int i = 0; i < nums.length; i++) {
              if (nums[i] > 5)
                  result[k++]=nums[i])
          }
}
console.log(filterNums()) // Output: [ 6, 7, 8, 9 ]
```

Example : code that filters numbers greater than 5

Programming Paradigms(contd..)

- ***Declarative programming*** is all about hiding away complexity and bringing programming languages closer to human language and thinking.
- The programmer specify what result is needed ,but it doesn't give instructions about *how* the computer should perform the task.

Imperative programming code

```
const nums = [1,4,3,6,7,8,9,2]
const result = [] // External variable
for (int i = 0; i < nums.length; i++) {
        if (nums[i] > 5)
           result[k++]=nums[i])
}
console.log(result) // Output: [ 6, 7, 8, 9 ]
```

Declarative programming code

```
const nums = [1,4,3,6,7,8,9,2]
console.log(nums.filter(num => num > 5)) // Output: [ 6, 7, 8, 9 ]
```

Object-Oriented Paradigm

- The core concept of OOP is to separate concerns into ***entities(known as classes)*** which are coded as objects. Each ***entity*** will group a given set of information ***(properties)*** and actions ***(methods)*** that can be performed by the entity.

- OOP makes heavy usage of classes (which are a way of creating new objects starting out from a blueprint that the programmer design). Objects that are created from a class are called instances.

Principles of Object Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism

```
// Create the two classes corresponding to each entity
class Cook {
     String name;
     constructor constructor (name) {
          this.name = name
     }
     mixAndBake() {
          - Mix the ingredients
          - Pour the mix in a mold
          - Cook for 35 minutes
     }
}
class AssistantCook {
     String name;
     constructor (name) {
          this.name = name
     }
     pourIngredients() {
          - Pour flour in a bowl
          - Pour a couple eggs in the same bowl
          - Pour some milk in the same bowl
     }
     chillTheCake() {
          - Let chill
     }
}
// Instantiate an object from each class
const Frank = new Cook('Frank')
const Anthony = new AssistantCook('Anthony')
// Call the corresponding methods from each instance
Anthony.pourIngredients()
Frank.mixAndBake()
Anthony.chillTheCake()
```

Principle	Real-world Example	C++ Language supports
Encapsulation: • Binding ***data and methods*** that can be operated on it together to form a single unit. • Provides ***security to data*** by controlled access to data.(which hides the internal state).	Human body is encapsulated with organs(data) and process(method) that work on its organs. Example : Respiration(Method) uses lungs(data) Blood purification(Method) uses kidneys(data) Blood circulation(Method) uses heart(data) etc.,	Using **Class** definition and **Object** creation
Inheritance: • ***"The process of acquiring the properties of one object into another object"*** is known as inheritance. • Inheritance provides a powerful and natural mechanism for organizing and structuring your software.	Naturally, Childrens ***inherits*** the characteristics like colour , height, facial structures etc., from their parents and ancestors.	• Using ***class A : B*** • Defining a new class by extends the existing class and adding its own properties to it known as inheritance
Polymorphism: "*One method , many actions*"	When human ***smells*** a delicious food **he** ***eats***, similarly when human *smells* a gas leakage he ***may run or turn off knob***	• Compile-time polymorphism using method overloading • Run-time polymorphism using ***inheritance*** and ***virtual functions***

Advantage of Object Oriented Paradigm

- The encapsulation feature provided by OOP languages allows programmer to define the class with many functions and characteristics and only few functions are exposed to the user.
- The technology of data hiding facilitates the programmer to design and develop safe programs that do not disturb code in other parts of the program.
- Using inheritance, we can eliminate redundant program code and continue the use of previously defined classes.
- All object oriented programming languages allows creating extended and reusable parts of programs.
- Object oriented programming changes the way of thinking of a programmer. This results in rapid development of new software in a short time.
- Objects communicate with each other and pass messages.
- Object oriented programs can be comfortably upgraded.

C++

```
#include<stdio.h>
struct student{
   int rollno;
   char name[20];
   float marks;
  void read(int n){
    for(int i=0;i<n;i++){
      scanf("%d%s%f",&s[i].rollno,&s[i].name,&s[i].marks);
   }
  void display(int n){
    for(int i=0;i<3;i++){
      printf("%d\t%s\t%f\n",s[i].rollno,s[i].name,s[i].marks);
   }
  }
};
int main(){
   struct student s[10];
   read(n);
   printf("Rollno\tName\tmarks\n");
   display(n)
   return 0;
}
```

C

```
#include<stdio.h>
struct student{
   int rollno;
   char name[20];
   float marks;
};
void read(int n){
    for(int i=0;i<n;i++){
      scanf("%d%s%f",&s[i].rollno,&s[i].name,&s[i].marks);
    }
}
void display(int n){
    for(int i=0;i<n;i++){
      printf("%d\t%s\t%f\n",s[i].rollno,s[i].name,s[i].marks);
    }
}
int main(){
   struct student s[10];
   read(n);
   printf("Rollno\tName\tmarks\n");
   display(n)
   return 0;
}
```

C++

```
#include<iostream>
class student{
   int rno; char name[20];
   float marks;
   public:
   void read(){
     cin>>rno>>name>>marks;
   }
   void display() {
       cout<<rno<<"\t"<<name<<"\t"
             <<marks<<endl;
   }
};
int main(){
  student s[10];
   int n;
   for(i=0;i<n;i++)
      s[i].read();
   cout<<rno<<"\t"<<name<<"\t"
             <<marks<<endl;
   for(i=0;i<n;i++)
      s[i].display();
   return 0;
}
```

class

Object

```
#include<iostream>
struct student{
   int rno;
   char name[20];
   float marks;
   void read(){
     cin>>rno>>name>>marks;
   }
   void display() {
       cout<<rno<<"\t"<<name<<"\t"
             <<marks<<endl;
   }
};
struct student s[10];
int main(){
   int n;
   for(i=0;i<n;i++)
      s[i].read();
cout<<rno<<"\t"<<name<<"\t"
             <<marks<<endl;
   for(i=0;i<n;i++)
      s[i].display();
   return 0;
}
```

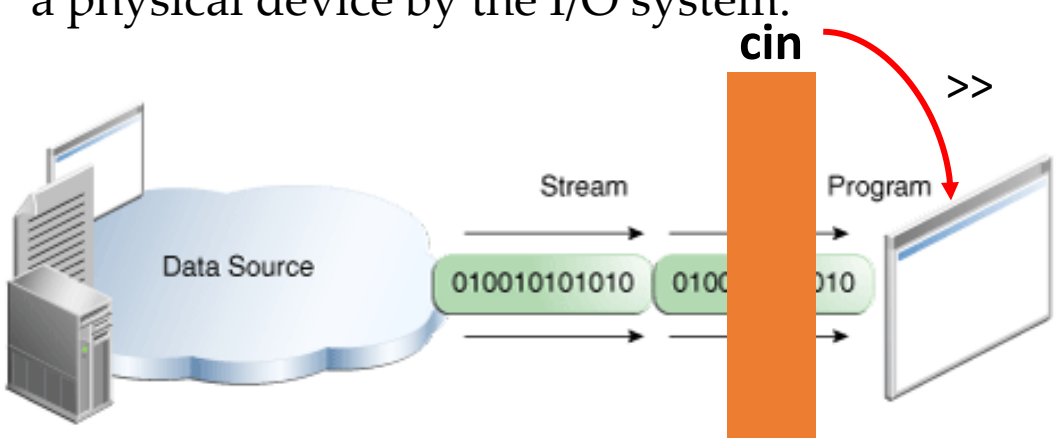
C

```
#include<stdio.h>
struct student{
   int rno;
   char name[20];
   float marks;
}s[10];
struct student read(int n){ /* ..*/}
void display(struct student) { /*…*/}
int main(){
  struct student s[10];
   int n;
   for(i=0;i<n;i++)
      s[i]=read();
   printf("rollno\tname\tmarks\n");
   for(i=0;i<n;i++)
      display(s[i]);
   return 0;
}
```

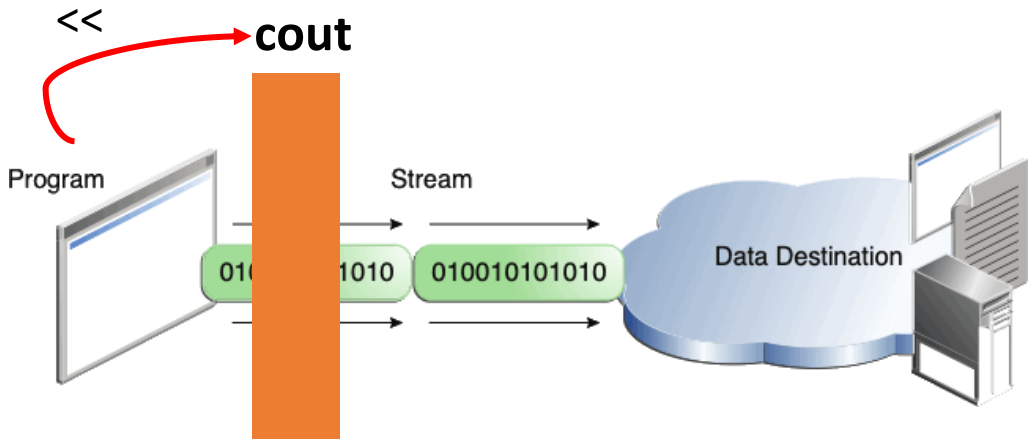
Class is similar to structures in c++, the difference is the members of structure is public (by default) ,where as in class it is private.

I/O Stream

- C++ uses two I/O systems. One is inherited from C, and another is using streams.
- C++ uses a convenient abstraction called ***streams*** to perform input and output operations in sequential media such as the screen, the keyboard or a file.
- A ***stream*** is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system.



Reading information into a program



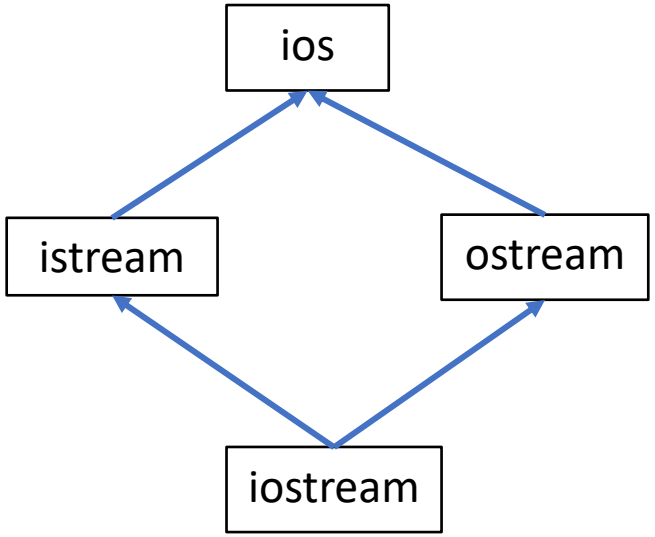
Writing information from a program.

- the << output operator is referred to as the *insertion operator* because it inserts characters into a stream
- the >> input operator is called the *extraction operator* because it extracts characters from a stream

By default cin and **cout** are connected to standard input(keyboard) and standard output(monitor) respectively

Pre-defined streams

- The **ios** class contains many member functions and variables that control or monitor the fundamental operation of a stream.
- This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O.
- ***istream*** and ***ostream*** are the derived(or sub) classes of ios class for input and output respectively.
- ***iostream*** is the subclass of both istream and ostream which is used for both input and ouput.



Objects of Stream	**Meaning**	**Default device**
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**.

Namespace and scope resolution operator

- Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose.(name collisions).
- Name collisions were increased when two or more third-party libraries were used by the same program.
- The **namespace** concept was introduced to resolve the name collisions.
- Prior to **namespace**, the entire C++ library was defined within the global namespace. After the introduction of namespace the C++ library is defined within its own namespace, called **std**.
- The **namespace** keyword allows you to partition the global namespace by creating a declarative region.
- A **namespace** defines a scope. The general form of **namespace** is shown here:

```
namespace name {
    // declarations
}
```

- Accessing members of the namespace can be done by **scope resolution *operator (::)*** or ***using*** directive.

```
#include<iostream>
using namespace std;
namespace myspace{
    int x=10;
}
int x=20;
int main(){
    int x=30;
    cout<<"namespace x :"<<myspace::x<<endl;
    cout<<"global variable x:"<<::x<<endl;
    cout<<"local variable x:"<<x<<endl;
    return 0;
}
```

std is predefined namespace

myspace is userdefined namespace

name of the namespace

Scope-resolution

name of the object

// x in user-defined namespace

// global variable x

// local variable x

Dynamic Memory management

- Allocating the memory(from heap area) to data objects at run-time.

	C++ language	**C Language**
Allocation	Uses ***new*** operator • datatype *p=new datatype(value); • datatype *p=new datatype[size]; • datatype *p=new(buf) datatype(value); • datatype *p=(type*)operator new(sizeof(datatype));	Uses ***malloc,calloc,and realloc*** functions
deallocation	Uses ***delete*** operator • delete p; • delete[] p ; • operator delete(p);	Uses ***free*** function
Syntax	Allocation int *p; p = ***new*** int[10]; No explicit and sizeof operator deallocation delete p ; // for one object delete[] p; // for array of objects	Allocation int *p; p=(int*) malloc(sizeof(int) *10); Require explicit typecast and sizeof operator Deallocation free(p)
Library support	It is a provided by language ,no library is required.	It is provided by third-party library <stdlib.h>

Dynamic Memory management (contd..)

Allocates memory for single object and initializes it.

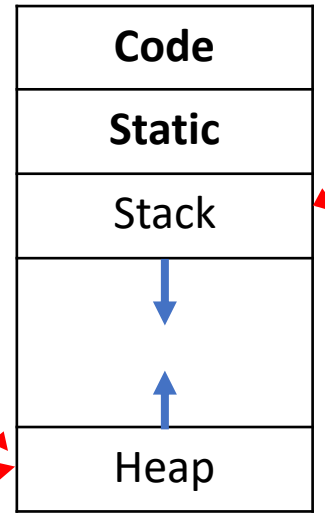
new

```
int main(){
    int *p;
    p=new int(5);
    cout<<p;
    delete p;
    return 0;
}
```

Allocates memory for an array of objects.

Array new

```
int main(){
    int *p;
    p=new int[5];
    for(int i=0;i<5;i++)
        cin>>p[i];
    for(int i=0;i<5;i++)
        cout<<p[i]<<endl;
    delete[] p;
    return 0;
}
```



Allocates memory(stack) for three object and initializes it.

buffernew

```
int main(){
    unsigned char buf[sizeof(double)*3];
    double *p=new(buf) double(10.2);
    double *q=new(buf+sizeof(double)) double(20.36);
    double *r=new(buf+(2*sizeof(double))) double(30.16);
    cout<<*p<<" is stored at "<< p<<endl;
    cout<<*q<<" is stored at "<< q<<endl;
    cout<<*r<<" is stored at "<< r<<endl;
    return 0;
}
```

Argument passing to functions

What is reference?
What is the difference between pointer and reference?

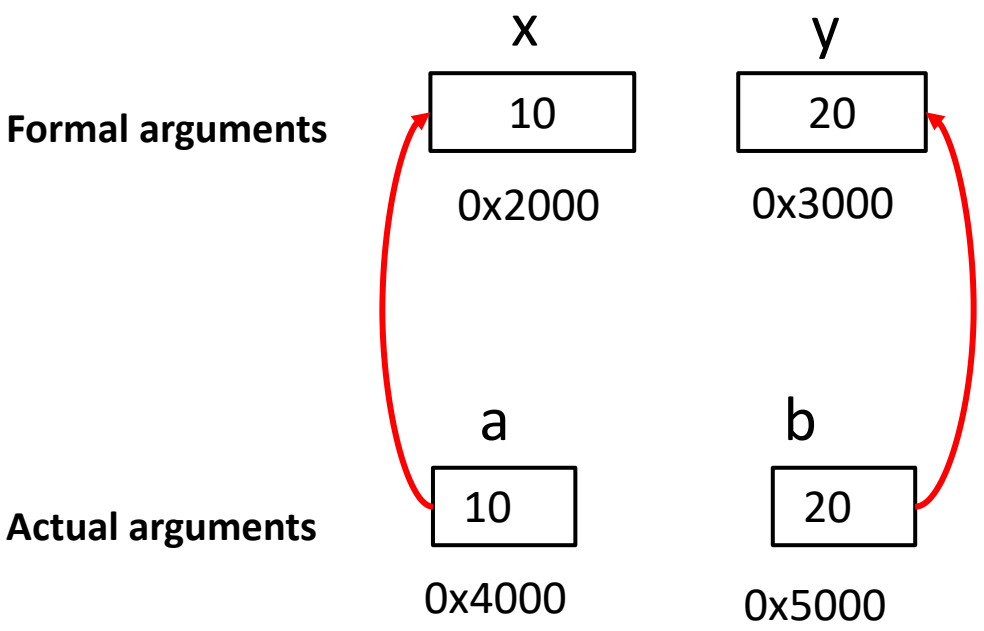
Call-by-value	**Call-by-reference**
In this formal parameters are the duplicate copy of actual parameters. so, the modifications on formal parameters **doesn't reflect** on actual parameters.	In this formal parameters are the references(aliases) of actual parameters.so, the modifications on formal parameters **reflect** on actual parameters
void swap(int,int); int main(){ int a,b; cout<<"Enter a and b values"<<endl; cin>>a>>b; cout<<" values of a and b before swaping: "<<a<<" "<<b<<endl; swap(a,b); cout<<" values of a and b after swaping: "<<a<<" "<<b<<endl; return 0; } void swap(int x , int y){ int t=x; x=y; y=t; }	void swap(int&,int&); int main(){ int a,b; cout<<"Enter a and b values"<<endl; cin>>a>>b; cout<<" values of a and b before swaping: "<<a<<" "<<b<<endl; swap(a,b); cout<<" values of a and b after swaping: "<<a<<" "<<b<<endl; return 0; } void swap(int& x,int& y){ int t=x; x=y; y=t; }

Argument passing to functions

Call-by-address	**Call-by-reference**
In this formal parameters contains the address of actural parameters.so, the modifications on formal parameters reflect on actual parameters	In this formal parameters are the references(aliases) of actual parameters.so, the modifications on formal parameters **reflect** on actual parameters.
void swap(int*,int*); int main(){ int a,b; cout<<"Enter a and b values"<<endl; cin>>a>>b; cout<<" values of a and b before swaping: "<<a<<" "<<b<<endl; swap(&a,&b); cout<<" values of a and b after swaping: "<<a<<" "<<b<<endl; return 0; } void swap(int *x , int *y){ int t=*x; *x=*y; *y=t; }	void swap(int&,int&); int main(){ int a,b; cout<<"Enter a and b values"<<endl; cin>>a>>b; cout<<" values of a and b before swaping: "<<a<<" "<<b<<endl; swap(a,b); cout<<" values of a and b after swaping: "<<a<<" "<<b<<endl; return 0; } void swap(int& x,int& y){ int t=x; x=y; y=t; }

Call-by-value

```
#include<iostream>
using namespace std;
void swap(int x,int y){
    int t;
    t=x;
    x=y;
    y=t;
}
int main(){
    int a=10,b=20;
    cout<<"Before swapping a and b values are "<<a<<" "<<b<<endl;
    swap(a,b);
    cout<<"After swapping a and b values are "<<a <<" "<<b<<endl;
    return 0;
}
```



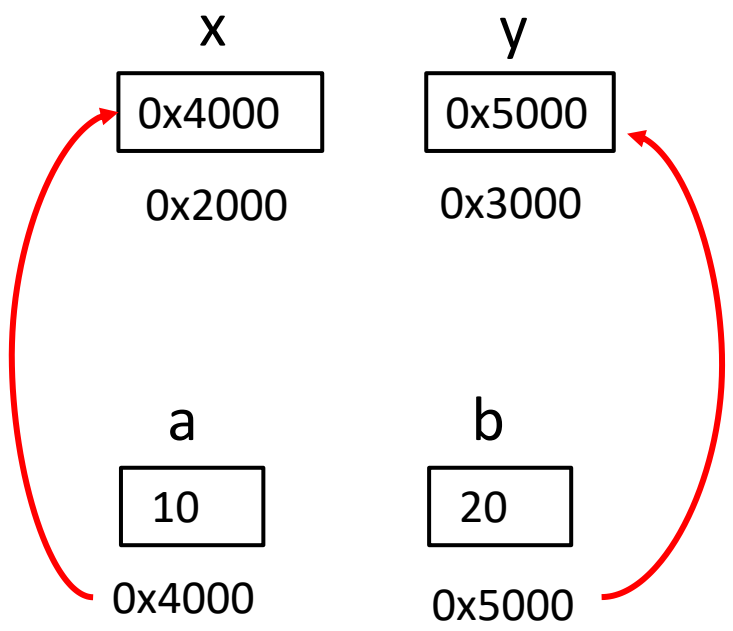
output

Before swapping a and b values are 10 20

After swapping a and b values are 10 20

Call-by-address

```
#include<iostream>
using namespace std;
void swap(int *x,int *y){
    int *t;
    t=x;          // t= 0x4000 i.e., t is pointing to a
    x=y;          // x= 0x5000 i.e., x is pointing to b
    y=t;          // y = 0x4000 i.e., y is pointing to a
}
int main(){
    int a=10,b=20;
    cout<<"Before swapping a and b values are "<<a<<" "<<b<<endl;
    swap(&a,&b);
    cout<<"After swapping a and b values are "<<a <<" "<<b<<endl;
    return 0;
}
```



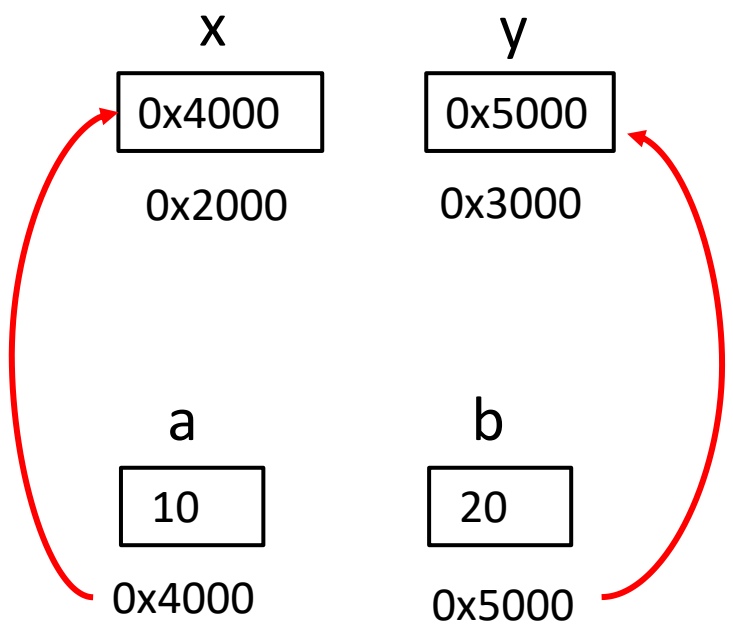
output

Before swapping a and b values are 10 20

After swapping a and b values are 10 20

Call-by-address

```
#include<iostream>
using namespace std;
void swap(int *x , int *y){
    int t;
    t=*x;          // t=*(0x4000) i.e., t=10
    *x=*y;         // *(0x4000)=*(0x5000) i.e., a=20
    *y=t;          // *(0x5000)=t i.e., b=10
}
int main(){
    int a=10,b=20;
    cout<<"Before swapping a and b values are "<<a<<" "<<b<<endl;
    swap(&a,&b);
    cout<<"After swapping a and b values are "<<a <<" "<<b<<endl;
    return 0;
}
```



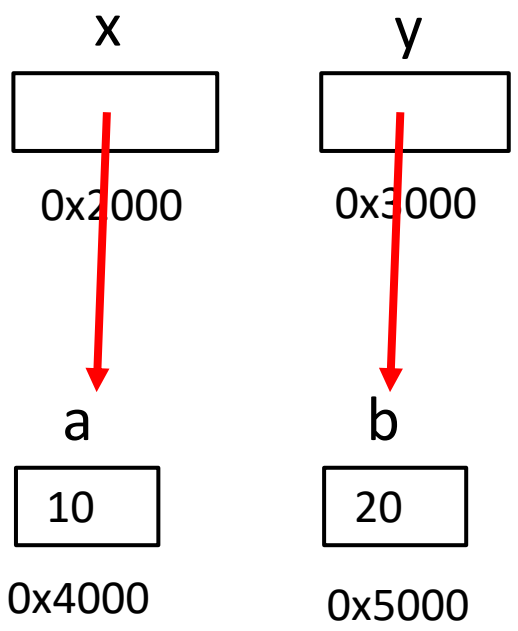
output

Before swapping a and b values are 10 20

After swapping a and b values are 20 10

Call-by-reference

```
#include<iostream>
using namespace std;
void swap(int& x , int& y){
    int t;
    t=x;
    x=y;
    y=t;
}
int main(){
    int a=10,b=20;
    cout<<"Before swapping a and b values are "<<a<<" "<<b<<endl;
    swap(a,b);
    cout<<"After swapping a and b values are "<<a <<" "<<b<<endl;
    return 0;
}
```



output

Before swapping a and b values are 10 20

After swapping a and b values are 20 10

Returning more values using references

```
void findminmax( int *q, int& n,int& min,int& max){
   for(int i=0;i<n;i++){
      if(q[i]<min) min=q[i];
      if(q[i]>max) max=q[i];
    }
 }
int main(){
   int n,min=INT_MAX,max=INT_MIN;
   cin>>n;
   int *p=new int[n];
   for(int i=0;i<n;i++)
      cin>>p[i];
   findminmax(p,n,min,max);
   cout<<"minimum ="<<min<<endl;
   cout<<"maximum="<<max<<endl;
   return 0;

}
```

- In C , using return statement we can return only one value or address(contains multiple values).
- In C++, we can pass reference arguments , the changes made in the called function will be reflected to calling function.

Example

Function returning the reference.

```
double& refMin( double& a, double& b) // Returns the reference to the minimum.
{
   return a <= b ? a : b;
}
int main()
{
   double x1 = 1.1, x2 = x1 + 0.5, y;
   y = refMin( x1, x2);                    // Assigns the minimum to y.
   cout << "x1 = " << x1 << " “
        << "x2 = " << x2 << endl;
   cout << "Minimum: " << y << endl;
   ++refMin( x1, x2);                      // ++x1, as x1 is minimal
   cout << "x1 = " << x1 << " "             // x1 = 2.1
        << "x2 = " << x2 << endl;          // x2 = 1.6
   ++refMin( x1, x2);                      // ++x2, because x2 is the minimum.
   cout << "x1 = " << x1 << " "             // x1 = 2.1
        << "x2 = " << x2 << endl;          // x2 = 2.6
   refMin( x1, x2) = 10.1;                 // x1 = 10.1, because x1 is the minimum.
   cout << "x1 = " << x1 << " "            // x1 = 10.1
        << "x2 = " << x2 << endl;          // x2 = 2.6
   refMin( x1, x2) += 5.0;                 // x2 += 5.0, because x2 is the minimum.
   cout << "x1 = " << x1 << " "            // x1 = 10.1
        << "x2 = " << x2 << endl;          // x2 = 7.6
   return 0;
}
```

const keyword

const keyword is used

- to define a variable as constant

```
const int x=5; // x is constant
x=6;           // Error

const int y;   // error . Constants must define a value at the time of definition
```

- In call-by-reference, the calling function may modify the actual data either accidentally or incidentally . So, to overcome the problem use **const** keyword to make the reference arguments ***read-only*** in the called function.

```
int modify(const int& x){
        int y=x+1;
        x++; // error
        return y;
}
int main(){
        int n=1;
        cout<<modify(n)<<endl;
        return 0;
}
```

Inline function

- The **inline** keyword tells the compiler to substitute the code within the function definition for every instance of a function call. But, the substitution is based on the compiler's decision i.e., it performs cost-benefit analysis and takes decision whether the code can be substituted or not. (one parameter is size of the function, larger code will not substituted)

- Using inline functions can make your program faster because they eliminate the overhead associated with function calls.

```
inline int max(int x , int y){
        return x>y?x:y;
}
int main(){
        int m=10,n=20;
        int maximum=max(m,n);
        cout<<“maximum= “<<maximum<<endl;
        return 0;
}
```

- A function defined in the body of a class declaration is implicitly an inline function.

```
#include <iostream>
using namespace std;
class MyClass {
    public:
        void print() {   cout << i << ‘ ‘;  } // Implicitly inline
    private:
        int i;
};
```

- Inline functions are similar to macros, because the function code is expanded at the point of the call at compile time. Inline functions are parsed by the compiler, and macros are expanded by the preprocessor

```
inline int square(int x){
        return x*x;
}
int main(){
        int n=5;
        cout<<square(n)<<endl;   // output 25
        cout<<square(n+1) <<endl; // output 36
        cout<<square(++n)<<endl;  // output 36
        return 0;
}
```

Default arguments

- In many cases, functions have arguments that are used so infrequently that a default value would suffice. To address this, the default-argument facility allows for specifying only those arguments to a function that are meaningful in a given call.
- C++ allows programmer to assign default values to the function parameters.
- Default values are specified while prototyping the function.

```
int functionIdentify(int =10);
int main(){
        int x=5,y;
        y=functionIdentify();
        cout<<"y="<<y<<endl;
        y=functionIdentify(x);
        cout<<"y="<<y<<endl;
        return 0;
}
int functionIdentify(int a){
        return a;
}
```

Default Arguments (contd…)

- **All parameters to the right of a parameter with default argument must have default arguments**

```
void f(int,double=0.0,char *);
```

- **Default arguments cannot be re-defined**

```
void g(int,double=0.0,char * = NULL);   // ok
void g(int,double=1.0,char *=NULL);   // error
```

- **All non-defaulted parameters needed in a call**

```
void g(int,double=0.0,char * = NULL);   // ok
int main() {
        int i=5;double d=1.2; char c='b';
        g();                    //error
        g(i);                   //ok
        g(I,d);                 //ok
        g(I,d,&c);              //ok
}
```

Function overloading

- Defining functions with same name but differ in number or type of arguments in the same scope is known as function overloading.

- It is used to provide static polymorphism(one name , many forms).

```
int add(int ,int);
double add(double,double);
int main(){
   int x =10,y=20;
   double a=10.5,b=21.5;
   cout<<add(x,y)<<endl; // binds to function add(int,int)
   cout<<add(a,b)<<endl; // binds to function add(double,double)
   return 0;
}
int add(int x, int y){
   return x+y;
}
double add(double x, double y){
   return x+y;
}
```

```
double area(double); // area of square
double area(double,double); //area of rectangle
int main(){
   double x =5.0,l=10.0,b=20.0;
   cout<<area(x)<<endl; // binds to function area(x)
   cout<<area(l,b)<<endl; // binds to function area(l,b)
   return 0;
}
double area(double x){
   return x*x;
}
double area(double l,double b){
   return l*b;
}
```

Contd..

- Function overloading resolution is done based on the following sequence
  - Exact match
  - promotion (standard type conversion)
  - user defined type conversion

- int g(double); //f1
- void f(); //f2
- void f(int); //f3
- double h(void); //f4
- int g(char,int); //f5
- void f(double,double=3.4); //f6
- void h(int,double); //f7
- Void f(char,char*) //f8

To match the function call ***f (5.6)***

- candidate functions : f2,f3,f6,f8
- viable functions(# of arguments) : f3,f6
- Best viable function : f6