Unit-4

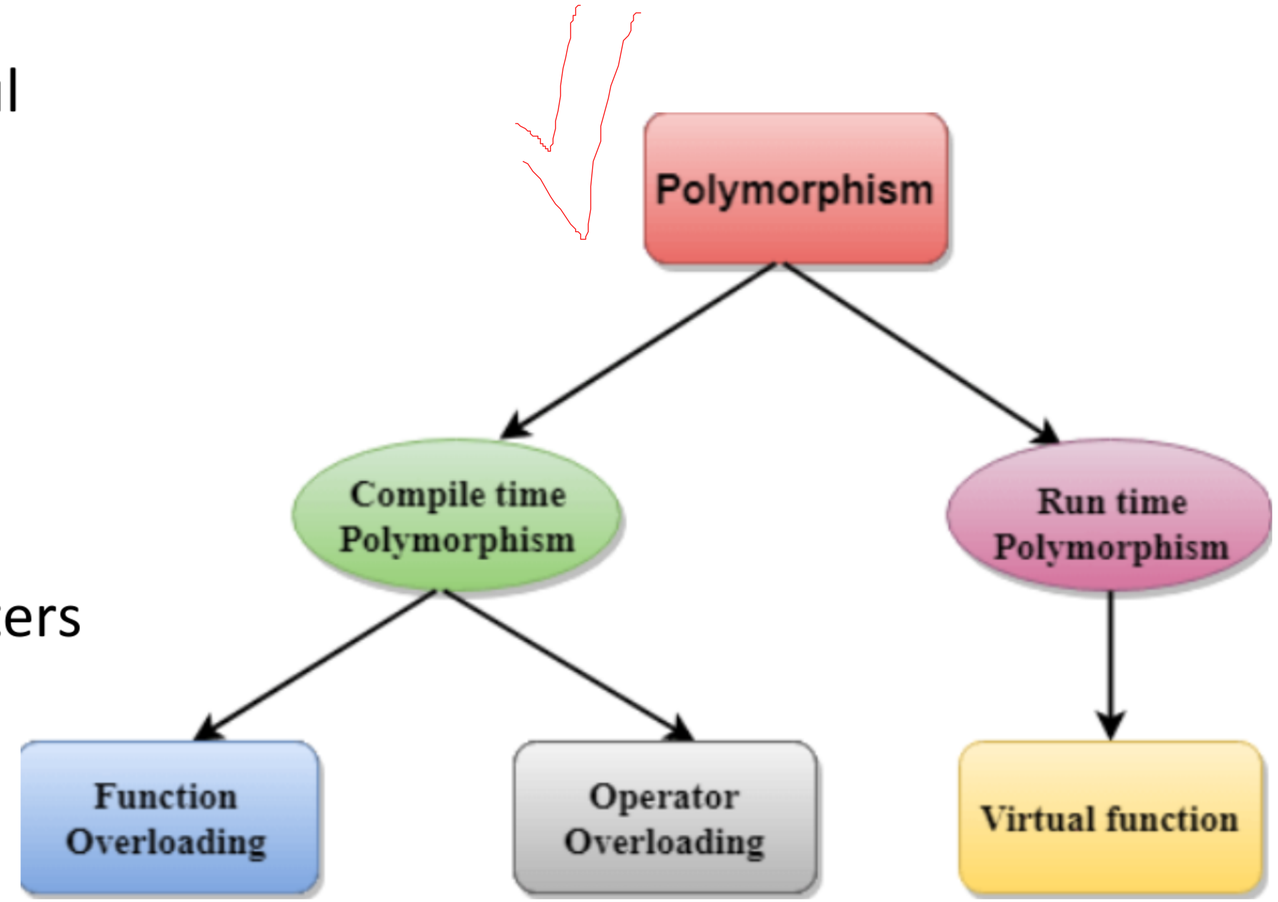
Polymorphism,

I/O and File Management

- Pointers, Pointers to Objects, pointers to members and member functions, Pointers to derived classes, virtual and pure virtual functions.

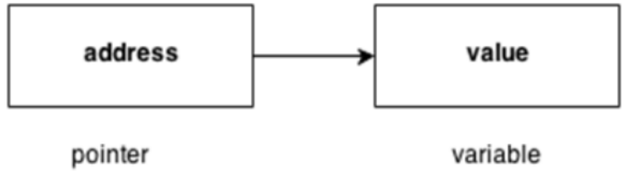
Polymorphism

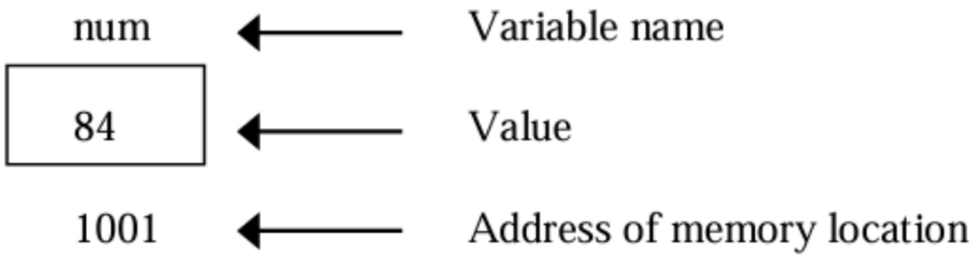
- Dynamic binding is one of the powerful features of C++
- This requires the use of pointers to objects.
- We have to learn how the object pointers and virtual functions are used to implement dynamic binding.



C++ Pointers Basics

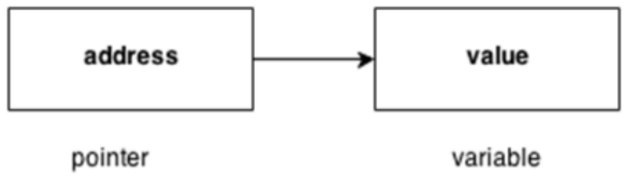
- A Pointer in C++ is variable whose value is a memory address.
- With pointers many memory locations can be referenced.
- Some data structures use pointers (e.g. linked list, tree).
- Variables are allocated at addresses in computer memory (address depends on computer/operating system)
- Name of the variable is a reference to that memory address





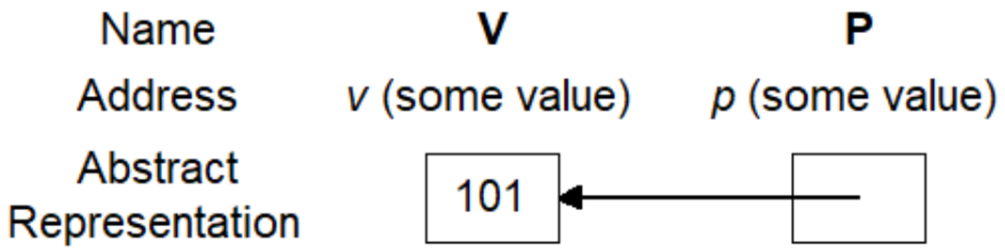
C++ Pointers Basics

- A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following):




```
int V = 101;

int *P = &V;
```



C++ Pointers Basics

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Pointer Declaration

- Pointers are declared as follows:

  <data-type> * variable_name ;

e.g.

int * xPtr; // xPtr is a pointer to data of type integer

char * cPtr; //cPtr is a pointer to data of type character

void * yPtr; // yPtr is a generic pointer,
// represents any type

Pointer Assignment

- Assignment can be applied on pointers of the same type
- If not the same type, a cast operator must be used
- Exception: pointer to **void** does not need casting to convert a pointer to **void** type
- **void** pointers cannot be dereferenced

Example

```
int *xPtr, *yPtr;  int x = 5;
    …
    xPtr = & x;   // xPtr now points to address of  x
    yPtr = xPtr;  // now yPtr and xPtr point to x
```

Address (&) Operator

- The address (&) operator can be used in front of any variable object
- the result of the operation is the location in memory of the variable

Syntax: &*Variable-Reference*

Examples:

```
int V;
int *P;
int A[5];
```

&V - memory location of integer variable V

&(A[2]) - memory location of array element 2 in array A

&P - memory location of pointer variable P

Indirection (*) Operator

- A pointer variable contains a memory address.
- To refer to the *contents* of the variable that the pointer points to, we use indirection operator

Syntax: **PointerVariable*

Example:

```
int V = 101;
int *P = &V;
/* Then *P would refer to the contents of the variable V (in this case, the integer 101)
   */
printf("%d",*P);  /* Prints 101 */
```

Pointers to Pointers

A pointer can also be made to point to a pointer variable
(but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;   /* P points to int V */
int **Q = &P; /* Q points to int pointer P */

printf("%d %d %d\n",V,*P,**Q);      /* prints 101 3 times */
```

```
#include <iostream>
using namespace std;
int main()
{
    int a=10, *ptr;
    ptr = &a;

    cout<< "Value of a : "<< *ptr<<"\n";
    *ptr=*ptr+a;

    cout<< "New Value of a : "<< a<<"\n";
    cout<<"\n";
    return 0;
}
```

```
Value of a : 10
New Value of a : 20
```

Swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=10,b=30,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
   return 0;
}
```

```
Before swap: *p1=10 *p2=30
After swap: *p1=30 *p2=10
```

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int *ptr1, **ptr2;

ptr1=&number;//stores the address of number variable
ptr2=&ptr1;

cout<<"Value of number        :"<<*ptr1<<endl;
cout<<"Address of number      :"<<&number<<endl;
cout<<"\n\n";

cout<<"Value of ptr1          :"<<ptr1<<endl;
cout<<"Address of ptr1        :"<<&ptr1<<endl;
cout<<"\n\n";

cout<<"Value of ptr2          :"<<ptr2<<endl;
cout<<"Address of ptr2        :"<<&ptr2<<endl;
cout<<"\n\n";

cout<<"Value of ptr2          :"<<*ptr2<<endl;
cout<<"Value of ptr2          :"<<**ptr2<<endl;
return 0;
}
```

Output:

```
Value of number        :30
Address of number      :0x7ffc7545e774


Value of ptr1          :0x7ffc7545e774
Address of ptr1        :0x7ffc7545e778


Value of ptr2          :0x7ffc7545e778
Address of ptr2        :0x7ffc7545e780


Value of ptr2          :0x7ffc7545e774
Value of ptr2          :30
```

Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers which are:

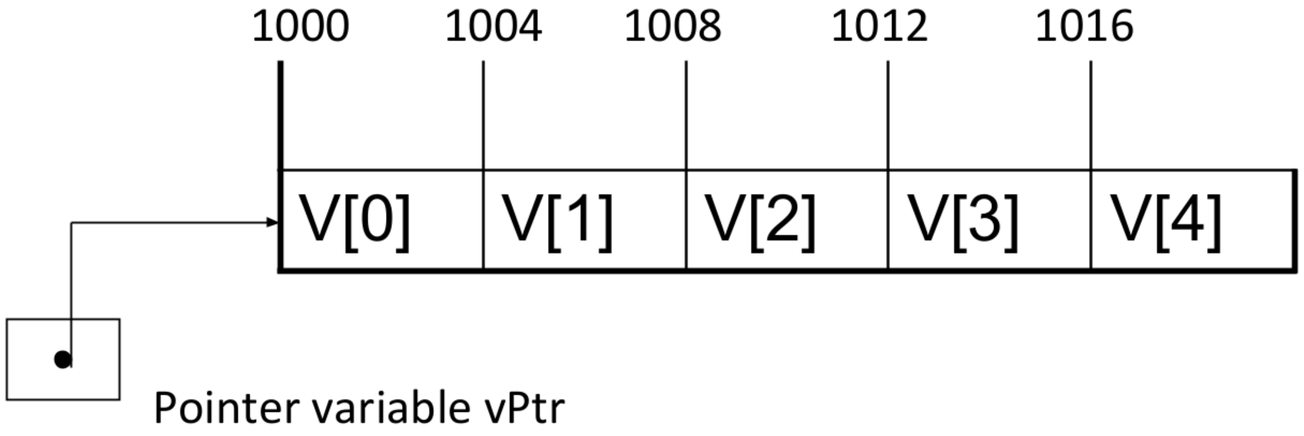
- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)
We cannot perform pointer arithmetic on variables which are not stored in contiguous memory locations.

Pointer Arithmetic (Cont.)

- **<u>Example</u>**

  Consider an integer array of 5 elements on a machine using 4 bytes for integers.



- vPtr pointes to first element V[0] (location 1000)

  i.e. vPtr = 1000

- vPtr +=2; sets vPtr to 1008

  i.e. vPtr points to V[2]

```
#include <iostream>
using namespace std;
int main()
{
   // Declare an array
   int v[3] = { 56, 100, 75 };
    // declare pointer variable
   int* ptr;
    // Assign the address of v[0] to ptr
   ptr = v;

   for (int i = 0; i < 3; i++)
   {
      cout << "Value at ptr = " << ptr << "\n";
      cout << "Value at *ptr = " << *ptr << "\n\n";

      // Increment pointer ptr by 1
      ptr++;
   }
 return 0;
}
```

```
Value at ptr = 0x7fff90be428c
Value at *ptr = 56

Value at ptr = 0x7fff90be4290
Value at *ptr = 100

Value at ptr = 0x7fff90be4294
Value at *ptr = 75
```

Subtracting pointers

- - Returns the number of elements between two addresses

e.g. if v is an array, vPtr1 and vPtr2 are pointer variables

```
vPtr1 = &v[0];
vPtr2 = &v[2];
```

then vPtr2 – vPtr1 = 2 (i.e. 2 addresses)

Subtracting pointers

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

int main() {
   // Define an array
   int v[5] = {10, 20, 30, 40, 50};
   int *vPtr1, *vPtr2;

   // Define pointers and assign addresses from array elements
   vPtr1 = &v[0]; // Pointer to the first element of the array
   vPtr2 = &v[2]; // Pointer to the third element of the array

   int difference = vPtr2 - vPtr1;
   cout << "Difference between vPtr2 and vPtr1: " << difference <<
endl;

   return 0;
}
```

Output:

Difference between vPtr2 and vPtr1: 2vPtr2

Lets Check….

```
int var=10, *ptr;

ptr=var;
// Wrong!  ptr is an address but var is not

*ptr = &var;
 //Wrong!  &var is an address, *ptr is the value stored in &var

ptr = &var;
// Correct! ptr is an address and so is &var

*ptr = var ;
// Correct!  both *ptr and var are values
```

Pointer operation

int *p1, *p2;

- p1 + 4, p2 2, p1++, p2 are valid operation

- p2 -p1 is valid and gives the no. of elements between p1 & p2, if p1 & p2 both are pointers to the same array

- Pointers can be compared --> p1 > p2, p1 ==p2, p1 != p2 etc

- p1 + p2, p1/p2 , p1 * p2, p1 / 3, p2 * 5 --> invalid operations

Using Pointers with Arrays and Strings

- We can declare the pointers to arrays as

Int *nptr;

nptr = number[0]; or nptr = number

Pointers with Arrays

Pointer to arrays

```
int main ()
{
    int arr[5] = {10,20,30,40,50};
    int *ptr;

    ptr = &arr[0];
    for ( int i = 0; i < 5; i++ )
    {
        cout <<"*(ptr+"  << i <<"):";
        cout <<*(ptr + i) << endl;
    }
    return 0;
}
```

Also, written as
`ptr = arr;`

0	10	1000	← ptr
1	20	1002	
2	30	1004	
3	40	1006	
4	50	1008	

Output:

*(ptr + 0) : 10
*(ptr + 1) : 20
*(ptr + 2) : 30
*(ptr + 3) : 40
*(ptr + 4) : 50

Pointers with Arrays

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int i, num,*ptr;

    ptr=arr;
    cout<<"enter the element to search: ";
    cin>>num;
```

```
 for(i=0;i<5;i++)
    {
        if(*ptr==num)
        {
            cout<<num<<" is present";
            break;
        }
        else if(i==5)
            cout<<"element is not present";
        ptr++;
    }

    return 0;
}
```

Array of Pointers

int *inarray[10];

It declares an array of 10 pointers, each of which points to an integer.

The address of 1st pointer is inarray[0],

Address of 2nd pointer is inarray[1] and so on.

```
Demo to show array of pointers !!!
Value of array !!
100
200
300
400
500
Value of array of pointers !!
100
200
300
400
500
```

```
#include <iostream>
using namespace std;
int main() {
    cout << "Demo to show array of pointers !!!\n";
    int  myarray[5] = {100, 200, 300, 400, 500};
    int *myptr[5];
    cout << "Value of array !!\n";
    for (int i = 0; i < 5; i+) {
        cout << myarray[i] << endl;
    }
myptr[0] = &myarray[0];
myptr[1] = &myarray[1];
myptr[2] = &myarray[2];
myptr[3] = &myarray[3];
myptr[4] = &myarray[4];
cout << "Value of array of pointers !!\n";
for (int i = 0; i < 5; i++) {
    cout << *myptr[i] << endl;   }
}
```

Pointers and Strings

- String literals are arrays of character sequences with null ends.
- The elements of a string literal are arrays of type const char (because characters in a string cannot be modified) plus a terminating null-character.

- char num[] = “one” creates an array of 4 characters with ‘\0’ at the end
- const char *numptr = “one” creates a pointer variable which points to the 1st character of the string.

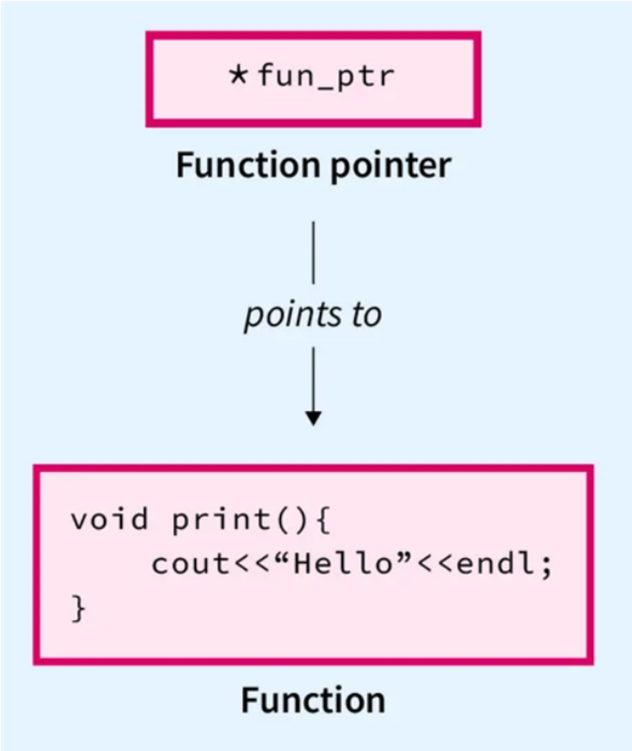
Initializing array of character pointers

```
void main(void)
{
char *song[]= { “ Humpty dumpty sat on a wall\n”,
                “ Humpty dumpty had a great fall\n”,
               “ Not all the kings horses and men\n”
};
cout<<song[2];
}
```

Not all the kings horses and men

Pointers to Functions

- Function Pointers are pointers, i.e. variables, which point to an address of a function.
- The pointer to function is known as callback function.
- It allows C++ program to select a function dynamically at run time.
- While passing a function as an argument to another function, the function is passed as a pointer.
- 2 types of function pointers:
  - that point to static member functions
  - That point to non static member functions
- Syntax: data_type (*function_name)();
- Ex: int (*num_function(int x));



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

int mult(int x, int y)
{
    return x * y;
}

int main(){

// Declaration of function pointer
    int (*funcptr)(int, int);


    funcptr = mult;


    int mul = funcptr(5, 7);

    cout << "The value of the product is: " << mul << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
typedef void (*funcptr)(int, int);

void add(int x, int y) {
    cout<< x + y;
}

void sub(int x, int y) {
    cout<< x - y;
}

int main(){
    funcptr ptr;
    ptr = &add;
    ptr(2,3);

   cout<<endl;
    ptr = &sub;
    ptr(5, 7);
    return 0;
}
```

void pointers

- In C++ special care has to be taken to handle the assignment of void pointers to other pointer types
- void pointers are very flexible because they can point to any data type.

```
void *p;
char *s;
p = s; // valid
s = p; // invalid assignment should be s = (char *) p;
```

- While you can assign a pointer of any type to a void pointer, the reverse is not true unless you specifically typecast it.

- They allow for greater flexibility in handling data where the type may vary or is not known at compile time.

Invalid pointers

- A pointer must point to a valid address, not necessarily to useful items (like for arrays).
- We refer to these as incorrect pointers.
- Additionally, incorrect pointers are uninitialized pointers.

```
int *ptr1;
int arr[10];
int *ptr2 = arr+20;
```

- It's essential for programmers to exercise caution when working with pointers, ensuring they always point to valid memory locations and are properly initialized, allocated, and deallocated.

NULL Pointers

- A null pointer is a pointer that point nowhere and not just an invalid address.
- Represent the absence of data or invalid memory addresses,
- Following are 2 methods to assign a pointer as NULL;

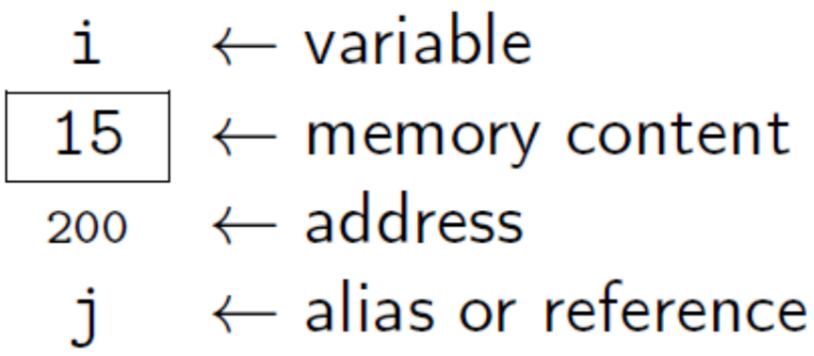
```
int *ptr1 = 0;
int *ptr2 = NULL;
```

Reference

- A reference is an alias / synonym for an existing variable

int i = 15; // i is a variable

int &j = i; // j is a reference to I



Wrong declaration	Reason	Correct declaration
`int& i;`	no variable to refer to – must be initialized	`int& i = j;`
`int& j = 5;`	no address to refer to as 5 is a constant	`const int& j = 5;`
`int& i = j + k;`	only temporary address (result of `j + k`) to refer to	`const int& i = j + k;`

Behaviour of Reference

```
#include <iostream>
using namespace std;
int main() {
int a = 10, &b = a; // b is reference of a

// a and b have the same memory
cout << "a = " << a << ", b = " << b << endl;
cout << "&a = " << &a << ", &b = " << &b << endl;

++a; // Changing a appears as change in b
cout << "a = " << a << ", b = " << b << endl;

++b; // Changing b also changes a
cout << "a = " << a << ", b = " << b << endl;
return 0;
}
```

```
a = 10, b = 10
&a = 002BF944, &b = 002BF944
a = 11, b = 11
a = 12, b = 12
```

Difference between Reference and Pointer

Pointers	References
• Refers to an address	• Refers to an address
• Pointers can point to NULL. int *p = NULL; // p is not pointing	• References cannot be NULL int &j ; //wrong

• Pointers can point to different variables at different times
 	
`int a, b, *p;`
`p = &a; // p points to a`
`...`
`p = &b // p points to b`
	• For a reference, its referent is fixed
	
	`int a, c, &b = a; // Okay`
	`.....`
	`&b = c // Error`

• NULL checking is required	• Makes code faster Does not require NULL checking
• Allows users to operate on the address – diff pointers, increment, etc.	• Does not allow users to operate on the address. All operations are interpreted for the referent
• Array of pointers can be defined	• Array of references not allowed

Pointers to objects

- Pointers to objects are useful for creating objects at run time.
- When accessing members of a class, given a pointer to an object, use the arrow (->) operator instead of the dot operator

Declaration of pointer: **className *ptr**

```
class employee {
    int code;
    char name [20] ;
    public:
        inline void getdata ( )= 0 ;
        inline void display ( )= 0 ;
};
```

The following codes is written

```
    employee *abc;
```

```
class item
{
    int code;
    float price;
public: void getdata(int a,float b) {
        code=a;
        price=b;
    }
     void show()  {
    cout<<code<<"\t"<<price<<endl;
    }
};
```

Declaration of object and pointer to class item:

```
item obj;
item *ptr=&obj;
```

The member can be accessed as follow.

a) Accessing members using dot operator

```
obj.getdata(101,77.7);
    obj.show();
```

b) using pointer

```
    ptr->getdata(101,77.7);
    ptr->show();
```

c) Using de referencing operator and dot operator

```
    (*ptr).getdata(101,77.7);
    (*ptr).show();
```

Creating array of objects using pointer:

- item *ptr=new item[10];

```
Enter number of objects to be created:2
Enter data for object1
Enter Code:11
Enter price:12345
Enter data for object2
Enter Code:12
Enter price:34556
Data in various objects are
Sno     Code    Price
1       11      12345
2       12      34556
```

```
int main() {
        int n; int cd; float pri;
        cout<<"Enter number of objects to be created:";
        cin>>n;

        item *ptr=new item[n];
        item *p; p=ptr;
        for(int i=0;i<n;i++) {
        cout<<"Enter data for object"<<i+1;
        cout<<"\nEnter Code:";
                cin>>cd;
                cout<<"Enter price:";
                cin>>pri;
                p->getdata(cd,pri);
                p++;
        }
        p=ptr;
        cout<<"Data in various objects are "<<endl;
        cout<<"Sno\tCode\tPrice\n";
        for(int i=0;i<n;i++)  {
                cout<<i+1<<"\t";
                ptr->show();
                ptr++;
        }
        return 0; }
```

C++ this pointer

- C++ language uses a unique keyword called “this” to represent an object that invokes a member function.
- this is a pointer to the object for which “this” function was called.

  Ex: The function calling A.max() set the pointer this to the address of the object A.
- The starting address is the same as the address of the first variable A in the class structure
- This unique pointer is automatically passed to a member function whenever it is called .
- The pointer this acts as an implicit argument to all the member functions.

C++ this pointer

```
class ABC
{
int a;
.....
.....
};
```

private variable ‘a’ can be used directly inside a member function like

a=145;

we can also use the following statement in below to do same activity

this->a=123;

Pointer to derived class

- Pointer to objects of the base class are type compatible with the pointers to objects of a derived class.

- a single pointer variable can be made to point to the objects belonging to different classes

Example: if B is a base class and D is a derived class from B , Then a pointer declare as a pointer to B can also be a pointer to D.

B *cptr; // here is pointer to class B type variable

B b; // base object

D d; // derived object

cptr=&b; // cptr points to object b

we can make cptr to point to the object d as follows:

cptr=&d; // cptr points to object d

```cpp
#include<iostream>
using namespace std;
class BC
{
public:
    int b;
    void show()
    {
        cout << "b=" << b << "\n";
    }
};
class DC: public BC
{
    public:
        int d;
        void show()
        {
            cout << "b=" << b << "\n";
        cout << "d=" << d << "\n";
        }
};
```

```cpp
int main()
{
BC *bptr; // base pointer
BC base;
bptr=&base; //base address
bptr->b=100; // access BC via base pointer

cout << "bptr points to base object \n";
bptr->show();

//derived class
DC derived;
bptr=&derived; // address of derived object

bptr->b=200; // access DC via base pointer

/* bptr->d=300; */ // won't work

cout << "bptr now points to derived object \n";
bptr->show(); //bptr now points to derived object
/* accessing d using a pointer of type derived
class DC*/
```

```cpp
DC *dptr; // derived type pointer
dptr=&derived;
dptr->d=300;

cout << "dptr is derived type pointer \n";
dptr->show();

cout << "using ((DC*)bptr)\n";
((DC*)bptr)->d=400;
((DC*)bptr)->show();
}
```

```
bptr points to base object
b=100
bptr now points to derived object
b=200
dptr is derived type pointer
b=200
d=300
using ((DC*)bptr)
b=200
d=400
```

Virtual functions

- Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.
- An essential requirement of the polymorphism is therefore the ability to refer to the objects without any regard to their classes .
- This necessary the use of a single pointer variable to refer to the objects of different classes.
- Here we use the pointer to base(super) class to refer to all the derived objects. but we just discovered that a base pointer, even it is made to contain the address of a derived class, always executes the function in the base class.
- The function in base class in declared as virtual using keyword virtual preceding its normal declaration when a function is made virtual, C++ language determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.
- This by making the base pointer to point to different objects, we can execute different version of virtual function

```
#include<iostream >
using namespace std;
class Base
{
public:
void display()
{
cout << "\n display
base";
}
virtual void show()
{
cout << "\n show base";
}
};
```

```
class derived: public Base
{
public:
void display()
{
cout << "\n Display Derived";
}
void show()
{
cout << "\n show derived";
}
};
```

```
int main()
{
Base B;
derived D;
Base *bptr;
cout << "\n bptr points to Base \n";
bptr=&B;
bptr->display(); // calls Base Version
bptr->show(); // calls derived version
cout << "\n \n bptr points to derived \n";
bptr=&D;
bptr->display(); // calls base version
bptr->show(); // calls derived version
}
```

output:

bptr points to Base

Display base

show base

bptr points to derived

Display base

show derived

Rules for virtual functions

1. The virtual function must be member of some class
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototype of the base class version of a virtual function and all the derived class. versions must be identical.

If two function with the same name have different prototypes, C++ consider them as overloaded functions, and the virtual function mechanism is ignored

Rules for virtual functions

7. we cannot have virtual constructors, but we can have virtual destructor.

8 while a base pointer can point to any type of the derived object, the reverse is not true.

ie, we cannot use pointer to a derived class to access an object of the base type.

9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class.

it is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases call will invoke the base function.

Pure Virtual Function

- A pure virtual function in C++ is a virtual function that has no implementation in the base class and is declared with the virtual keyword followed by = 0.
- It serves as a placeholder for a function that must be implemented by any derived class.

```
class Base {
public:
    virtual void doSomething() = 0;  // Pure virtual function
};

class Derived : public Base {
public:
    void doSomething() override {
        // Implementation of doSomething in the Derived class
    }
};
```

Unit-4

I/O and File Management

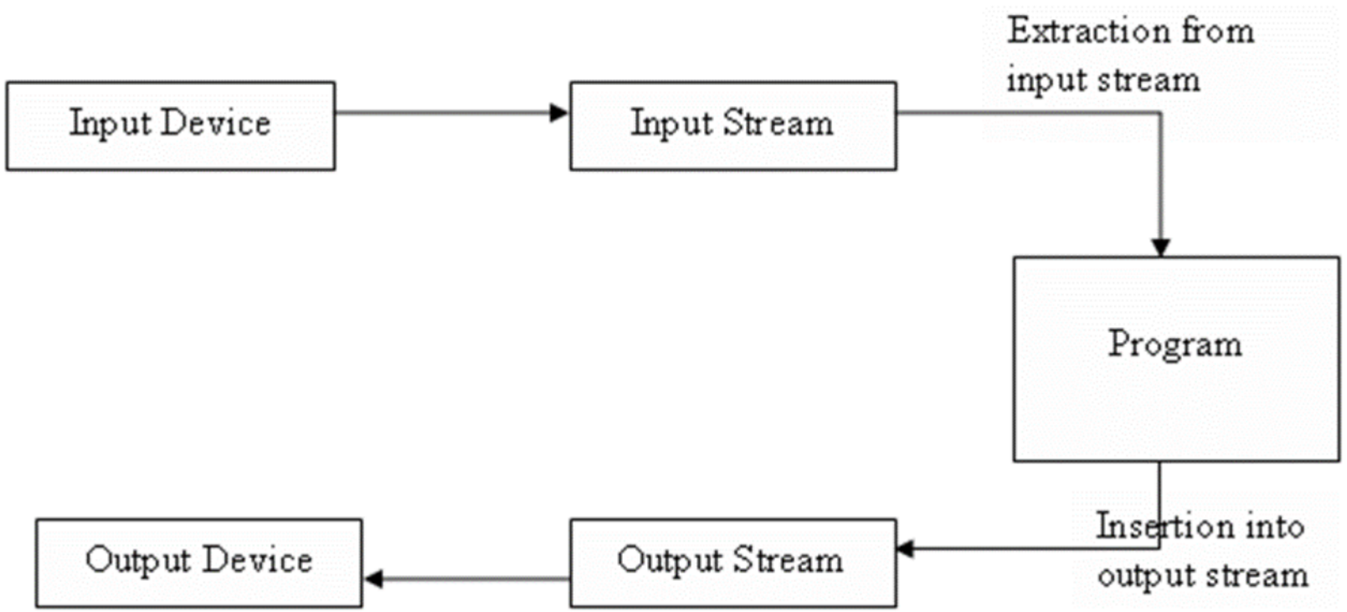
- **I/O and File Management:** Concept of streams, C++ stream classes, Unformatted and formatted I/O, manipulators, C++ File stream classes, File management functions, File modes, Binary and Random Files.

Concept of streams

- C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives.
- C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.
- **Stream is an interface** to the programmer that is independent of the actual device being accessed.
- A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

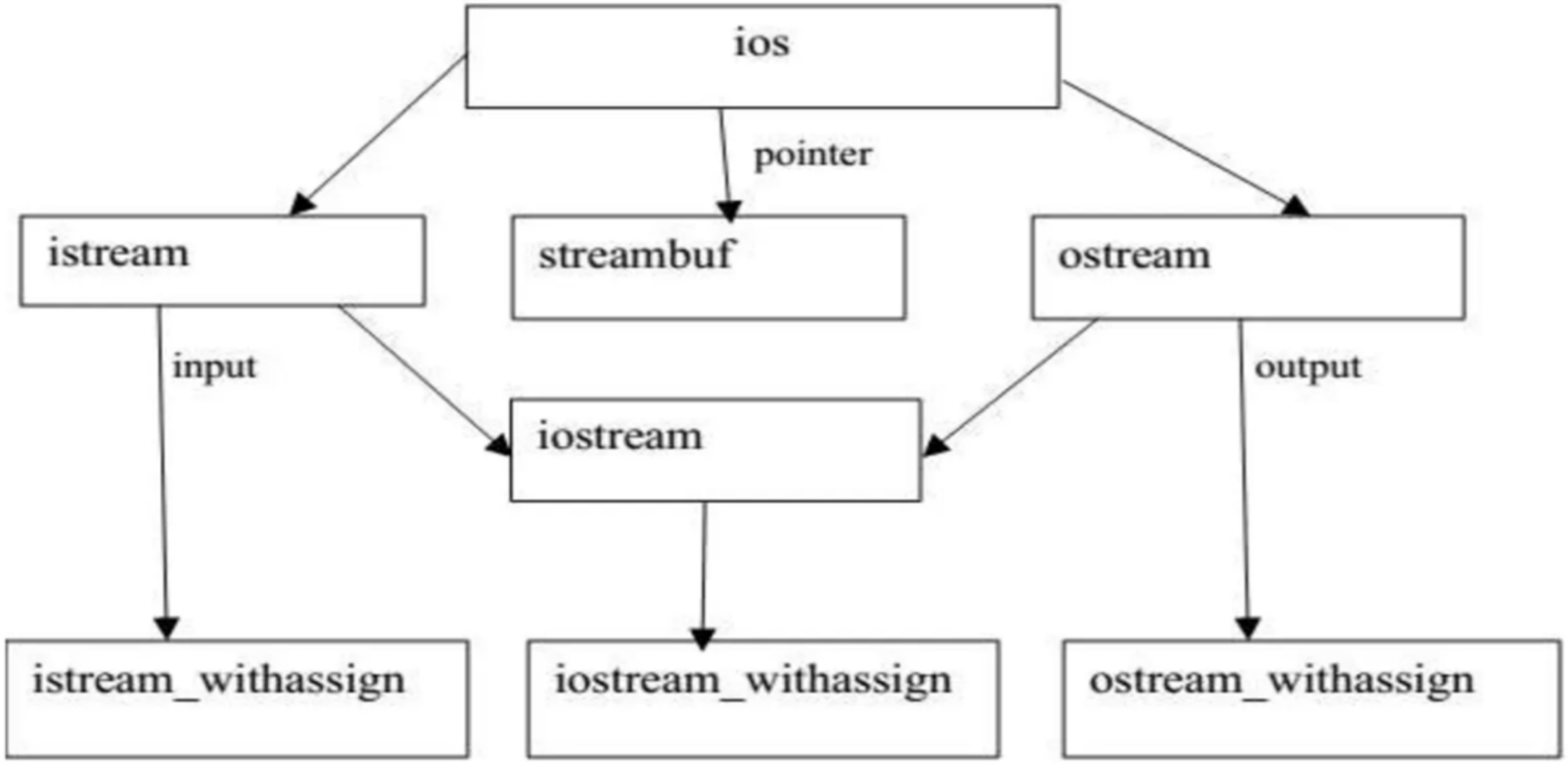
Concept of streams

- The source stream that provides data to the program is called the input stream and destination stream that receives output from the program is called the output stream.
- In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream.



Stream Classes

- These are the hierarchy of classes that are used to define various streams to deal with both the console and the disk files.



supports Assignment operators

Stream Classes

Class	Content / Functions
ios	• It is an input and output stream class. • It is used to implement a buffer. It is a pointer to the buffer, streambuf. • ios matains information about the state of the buffer.
istream	• istream provides formatted input. • It is used to handle formatted as well as unformatted conversion of character from a streambuf. • istream declares functions such as peek(), tellg(), seekg(), getline(), read() etc. • istream class overloads the “>>” operator.
ostream	• It is used for general purpose output. • ostream declares functions such as tellp(), put(), write(), seekp(), etc. • ostream overloads the “<<“ operator.
iostream	• It is used to handle both input and output streams.
istream_withassign	• It is derived from istream. • It is used for cin input.
iostream_withassign	• It is a bidirectional stream.

istream Class Functions

Function	Purpose
get(ch)	Extract one character into ch.
get(str)	Extract characters into array *str*, until '\n'.
putback(ch)	Insert last character read back in to input stream.
ignore(MAX, DELIM)	Extract and discard MAX characters until the specified delimiter DELIM.
peek(ch)	Read one character without extracting it from the stream.
gcount()	Return number of characters read by immediate *get(), getline(), or read()* function.
read(str, MAX)	Extract upto MAX characters, until EOF from a file.
seekg()	Set distance of file pointer from start of file.
seekg(pos, seek_dir)	Set distance of file prointer from specified place in a file. *seek_dir* can be *ios::beg, ios::cur,* or *ios::end*.
tellg(pos)	Return position of file pointer from start of file.

Unformatted input/output operations In C++

- The operator >> is overloaded in the istream class and operator << is overloaded in the ostream class.
- The general format for reading data from the keyboard:

```
cin >> var1 >> var2 >> …. >> var_n;
```

- Here, var1, var2, ……, varn are the variable names that are declared already.
- The input data must be separated by white space characters and the data type of user input must be similar to the data types of the variables which are declared in the program.
- The operator >> reads the data character by character and assigns it to the indicated location.
- Reading of variables terminates when white space occurs or character type occurs that does not match the destination type.
- The general format for displaying data on the screen:

```
cout << item1 << item2 << ….  << item_n;
```

put() and get() functions:

- The class istream and ostream have predefined functions get() and put(), to handle single character input and output operations.
- The function get() can be used in two ways, such as get(char*) and get(void) to fetch characters including blank spaces, newline characters, and tab.
- The function get(char*) assigns the value to a variable and get(void) to return the value of the character.

Example:

```
char c;
cin.get(c) ;            // get a character from keyboard
                // and assign it to c
while(c != '\n')
{
    cout << c;  // display the character on screen
    cin.get(c);  // get another character
}
```

get(void):

```
char c;
c=cin.get()
```

NOTE: the operator >> skip the whitespaces and newline characters

put() and get() functions:

- The function put(), a member of ostream class, can be used to output a line of text, character by character.
- For example,

  cout << put ('x'); displays the character x and

  cout << put (ch) ; displays the value of variable ch.

- The variable ch must contain a character value. We can also use a number as an argument to the function put ().
- For example,

  cout << put (68) ; displays the character D.

This statement will convert the int value 68 to a char value and display the character whose ASCII value is 68.

C++ getline() and write() Functions

- We can read and displays a line of text more efficiently using the line-oriented input/output functions getline() and write().
- The getline() function reads a whole line of text(strings) that and ends with a newline character(transmitted by RETURN key).
- This function can be invoked by the help of object cin as follows:

```
cin.getline (line, size);
```

- The reading is terminated as soon as whenever the newline character '\n' is encountered or size-1 characters are read (whichever occurs first).
- The newline character is read but it is not saved. Instead, it is replaced by the null character.
- consider the following example:

```
char name[20];
cin.getline(name,20);
```

NOTE: cin can read strings that do not contain white spaces

C++ getline() and write() Functions

- The write() function displays an entire line and has the following form:

```
cout.write (line, size)
```

- The first argument line presents the name of the string to be displayed and
- the second argument size indicates the number of characters to display.
- Note that it does not stop displaying the characters automatically when the null character is encountered.
- If the size is greater than the length of line, then it displays beyond the bounds of line.

Write a program to print the following :

P

Pr

Pro

Prog

Progr

Progra

Program

Progra

Progr

Prog

Pro

Pr

P

```
char * string2 = "Program";
int n = strlen(string2);
for (int i=1; i < n; i++)
{
cout.write(string2,i);
cout << "\n";
}
for (i=n; i>0; i--)
{
cout.write(string2, i);
cout << "\n";
}
```

Formatted console Input output operations

- C++ language also supports a number of features that could be used for formatting the output.

- These features include:

1. ios class functions and flags.
2. Manipultors.
3. User-defined output functions.

ios Class

- The ios class contain a large number of member functions that would help us to format the output in a number of ways.

- The most important ones among them are listed in below Table

Table: ios format functions	
Width ()	To specify the required fields size for displaying an output value or contents
precesion ()	To specify the number of digits to be displayed after the decimal point of a float value
fill ()	To specify a characters that are used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear flags specified

Setf(arg1,arg2)

- The setf() function can be used as follows:

  cout.setf(arg1,arg2);

- The arg1 is one of the formatting flags defined in the class ios.

- The formatting flag specifies the format action required for the output.

- Another ios constant, arg2, known as bit field specifies the group to which the formatting flag belongs.

Flags and Bit Fields

- The ios class contains several flags and bit fields to adjust or format the output displayed on the screen.
- Available bit fields and flags in ios class are as follows:

Format	Flag	Bit Field
Left justification	ios::left	ios::adjustfield
Right justification	ios::right	ios::adjustfield
Padding after sign and base	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

ios::adjustfield is used with setf() function to set the alignment of padding to left, right, or internal.

ios::floatfield is used with setf() function to set the floating point notation to scientific or fixed.

ios::basefield is used with setf() function to set the display notation to decimal, octal or hexadecimal.

Example of setf(arg1, arg2)

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout << "TABLE 1" << "\n";
```

T	A	B	L	E		1	*	*	*	*	*	*	*	*

```
cout.fill('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);
cout << -12.34567 << "\n";
```

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

Flags without Bit Fields

- Some flags does not contain any bit fields.
- Such flags are listed below:

Flag	Purpose
ios::showbase	Uses base indicator on output.
ios::showpos	Displays + preceding positive number.
ios::showpoint	Shows trailing decimal point and zeros.
ios::uppercase	Uses capital case for output.
ios::skipus	Skips white space on input.
ios::unitbuf	Flushes all streams after insertion.
ios::stdio	Adjusts the stream with C standard input and output.
ios::boolapha	Converts boolean values to text.

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

void IOS_width()
{
    cout << "-------------------------\n";
    cout << "Implementing ios::width\n";
    char c = 'A';
    cout.width(5);
    cout << c <<"\n";
    int temp = 10;

    cout<<temp;
    cout << "\n-------------------------\n";
}

void IOS_precision()
{
    cout << "\n-------------------------\n";
    cout << "Implementing ios::precision\n\n";
    cout << "Implementing ios::width";
    cout.setf(ios::fixed, ios::floatfield);
    cout.precision(2);
    cout<<3.1422;
    cout << "\n-------------------------\n";
}
```

```
void IOS_fill()
{
    cout << "\n-------------------------\n";
    cout << "Implementing ios::fill\n\n";
    char ch = 'a';
    cout.fill('*');
    cout.width(10);
    cout<<ch <<"\n";

    int i = 1;

    cout.width(5);
    cout<<i;
    cout << "\n-------------------------\n";
}

void IOS_setf()
{
    cout << "\n-------------------------\n";
    cout << "Implementing ios::setf\n\n";
    int val1=100,val2=200;
    cout.setf(ios::showpos);
    cout<<val1<<" "<<val2;
    cout << "\n-------------------------\n";
}
```

```
void IOS_unsetf()
{
    cout << "\n-------------------------\n";
    cout << "Implementing
ios::unsetf\n\n";

cout.setf(ios::showpos|ios::showpoint);
    cout.unsetf(ios::showpos);
    cout<<200.0;
    cout << "\n-------------------------\n";
}

int main()
{
    IOS_width();
    IOS_precision();
    IOS_fill();
    IOS_setf();
    IOS_unsetf();
    return 0;
}
```

Manipulators

- Manipulators are special functions that can be included in the I/O(Input output) statements to alter the format parameters of a stream.

- Two or more manipulators can be used as a chain in one statement as shown below:
  cout<<manip1<<manip2<<manip3<<item;
  cout<<manip1<<item1<<manip2<<item2;

- Table shows some important manipulator functions that are frequently used.

- To access these manipulator, the file iomanip.h should be included in the program.

Manipulators	Equivalent ios function
setw()	witdh()
setprecision()	precision()
setfill()	fill()
setiosflags()	self()
resetiosflags()	unself()
Endl	“\n”

Manipulator	Function
setw(int n)	To set the field width to *n*
setbase	To set the base of the number system
setprecision(int p)	The precision is fixed to *p*
setfill(char f)	To set the character to be filled
setiosflags(long l)	Format flag is set to *l*
resetiosflags(long l)	Removes the flags indicated by *l*
endl	Gives a new line
skipws	Omits white space in input
noskipws	Does not omit white space in the input
ends	Adds null character to close an output string
flush	Flushes the buffer stream
lock	Locks the file associated with the file handle
ws	Omits the leading white spaces present before the first field
hex, oct, dec	Displays the number in hexadecimal or octal or in decimal format

- cout<<setw(10)<<12345;
- This statement prints the value 12345 right justified in a field width of 10 characters.

- The output can be made left-justified by modifying the statement as follows:
- cout<<setw(10)<<setiosflags(ios::left)<<12345;

- One statement can be used to format output for two or more values.
- For example,

- cout<<setw(5)<<setprecision(2)<<1.2345<<setw(10)<<setprecision(4)<<sqrt(2) <<setw(15)<<setiosflags(ios::scientific) <<sqrt(3)<<endl;

- Will print all the three values in one line with the field size of 5,10,and 15 respectively.

Try this

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   float PI = 3.14;
   int num = 100;
   cout << "Entering a new line." << endl;
   cout << setw(10) << setfill('-') << "Output" << endl;
   cout << setprecision(10) << PI << endl;
   cout << setbase(16) << num << endl; //sets base to 16
}
```

Try this

- write a c++ program with precision formatting to display varying number of digits after decimal for Pi value

- There is a major difference in the way the manipulators are implemented as compared to the ios member functions.

- The ios member function return the previous format state which can be used later.

- In case, we need to save the old format states, we must use the ios member function rather than the manipulators.

```
cout.precision(2);//previous state
int p=cout.precision(4);//current state;
```

When these statements are executed, p will hold the value of 2(previous state) and the new format state will be 4.

We can restore the previous format state as follows:

```
cout.precision(p)//p=2
```

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

int main()
{
streamsize prev = cout.precision();   // Save the current precision state

cout.precision(4);            // Set precision to 4 decimal places

cout << fixed << setprecision(2) << sqrt(12) << endl; // Output square root of 12 with 2
decimal places

cout.precision(prev);            // Restore the previous precision state


cout << fixed << sqrt(12) << endl;     // Output square root of 12 with the restored precision
return 0;
}
```

User-Defined Manipulators

- Sometimes, to satisfy custom requirements, we have to create our own manipulator.
- Such manipulators which are created by the programmer or user are known as user-defined manipulators.
- Syntax for creating a user-defined manipulator is as follows:

```
ostream & m_name(ostream &os)
{
    Statement(s);
    return os;
}
```

```
ostream & newl(ostream & os)
{
    os<<"\n";
    return os;
}
```

- After creating the manipulator named newl, we can use it our program as shown below:

```
int main()
{
    cout<<"Hello"<<newl<<"World"<<newl;
    return 0;
}
```

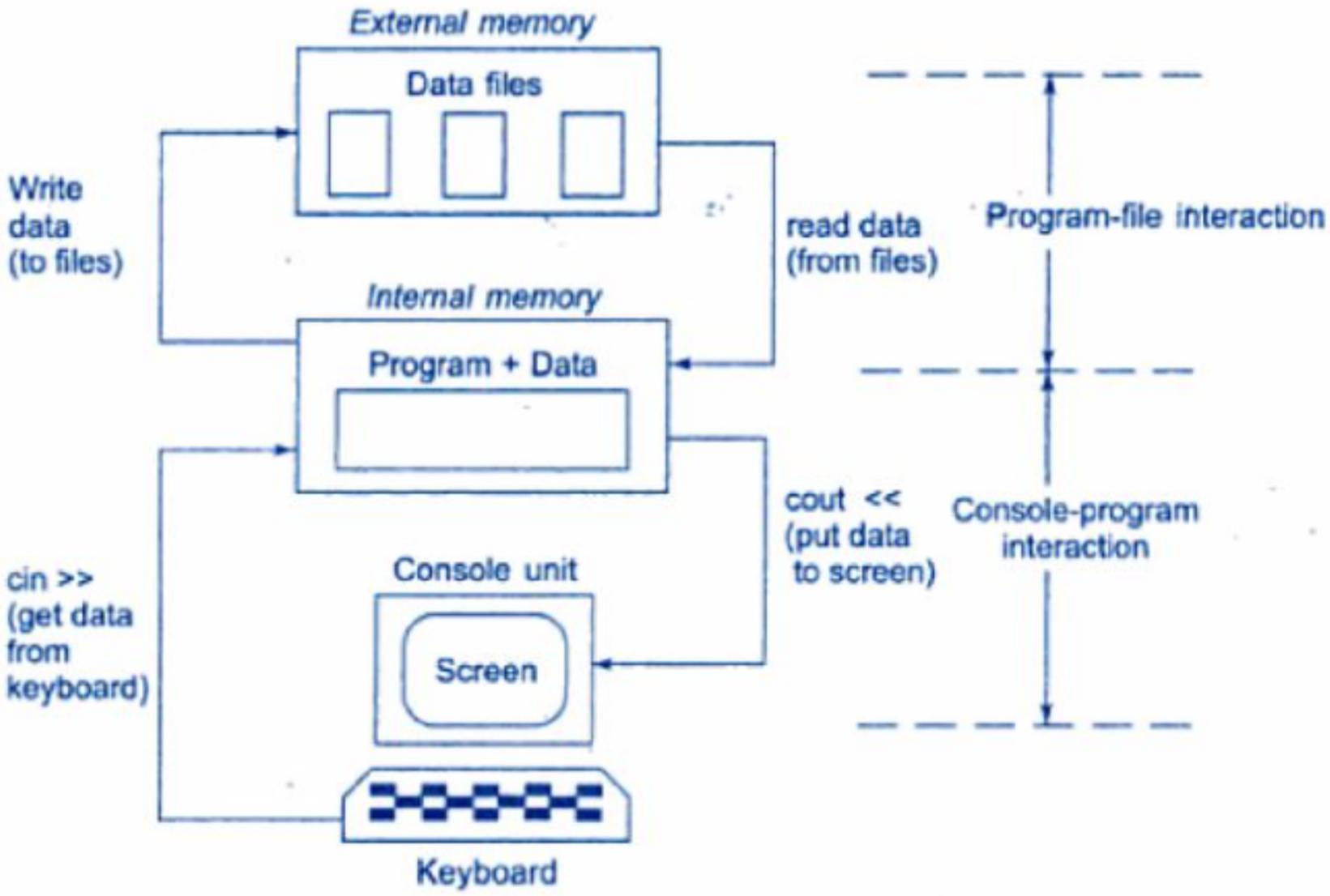
FILE HANDLING

Files

- A file is a collection of related data stored in a particular area on the disk.
- Programs can be designed to perform the read and write operations on these files
- A program involves either or both of following kinds of data communication:

1. Data transfer between the console unit and the program
2. Data transfer between the program and a disk file

Console Program File Interaction



Introduction

- Computer programs are associated to work with files as it helps in storing data & information permanently.
- File - itself a bunch of bytes stored on some storage devices.
- In C++ this is achieved through a component header file called fstream.h
- The I/O library manages two aspects-
  1. as interface and
  2. for transfer of data.
- The library predefine a set of operations for all file related handling through certain classes.

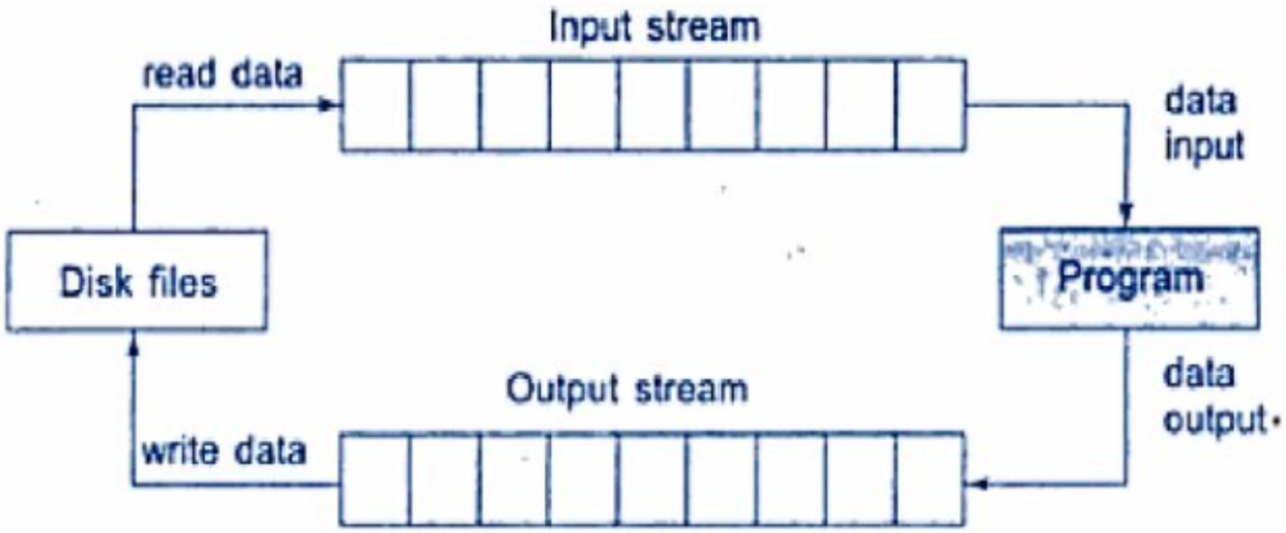
File Input and output streams

- C++ provides a new technique for handling I/O operations through mechanism known as streams.
- A stream refers to a flow of data.

- Classified in 2 categories:

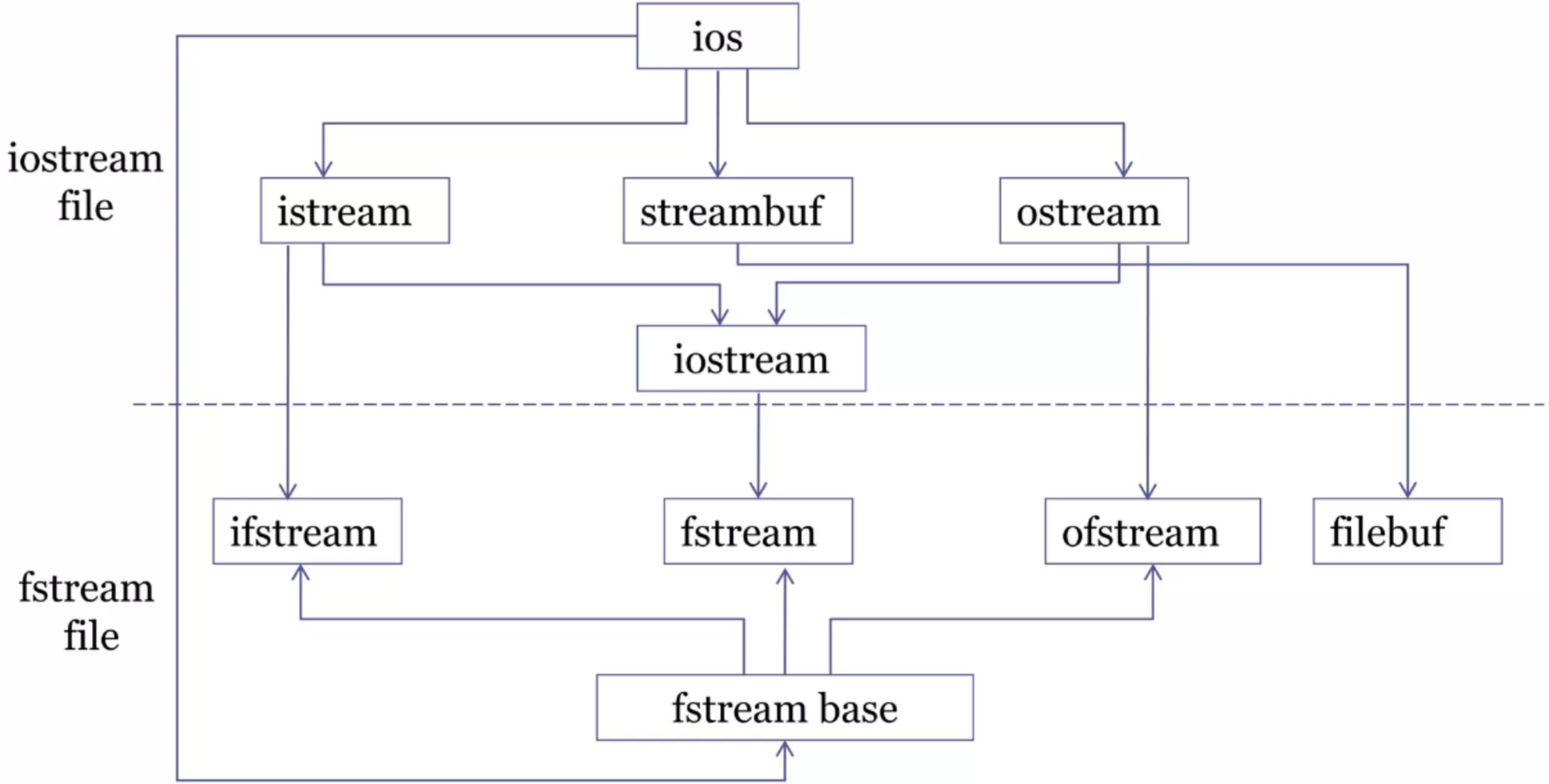
1. Output stream- the flow of data is from program to the output device.

2. Input stream- the flow of data is from input device to a program in main memory.



Classes for file stream operations

A file stream can be defined using classes ifstream, ofstream, and fstream.



Class	Contents
filebuf	it is used to set the file buffers to read and write contains **openprot** constant used in the open() of file stream classes. Also contain **close()** and **open()** as members
fstreambase	provides operations common to the file streams, serves as a base for **fstream**, **ifstream** and **ofstream** class. contains **open()** and **close()** functions
ifstream	provides input operations. contains **open()** with default input mode. Inherit the functions **get(), getline**(), **read**(),**seekg**() and **tellg**() functions from istream
ofstream	provides output operations, contains **open**() with default output mode inherits **put**(), **seekp**(),**tellp**(), and **write**() functions from ostream
fstream	provides support for simultaneous input and output operations, contains open() with default input mode. Inherits all the functions from **istream** and **ostream** classses through **iostream**

Files:

- Convenient way to deal large quantities of data.
- Store data permanently (until file is deleted).
- Avoid typing data into program multiple times.
- Share data between programs.
- We need to know:
  - how to "connect" file to program
  - how to tell the program to read data
  - how to tell the program to write data
  - error checking and handling EOF

Open and closing file

If we want to use a disk file, we necessary to decide the following things about the file and its intended use:

1. suitable name for the file
2. Data type and structure
3. purpose
4. opening method

A file can be opened in two ways:

1. **Using constructor function** - useful only when we use only one file in the stream

1. **Using member function-** useful when we want to manage multiple files using one stream

open using constructor

This involves the following steps

1. create a file stream object to manage the stream using the appropriate class.

   It is to say the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream

2. Initialize the file object with the desired filename

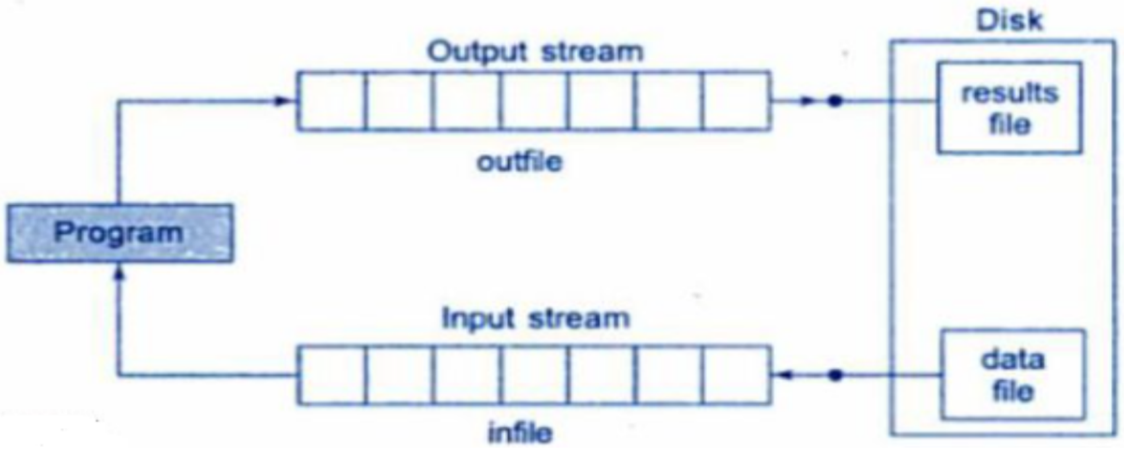
   for example the following statement open a file named “result” for output stream **outfile**

   ofstream outfile(“result”); // output only

it is creates outfile as an ofstream object that manages the output stream.

This object can be any valid in C++ programming name such as o_file, myfile or fout.

This statement also opens the file result and attaches it to the output stream outfile.



- The following statement declares infile as an ifstream object and attaches it to the file data for reading

```
ifstream infile ("data");        //input only
```

The program may contain statements like:

```
outfile<<total;
outfile<<sum;
infile>>number;
infile>>string
```

- We can also use same file for both reading and writing data

```
Program1
........
ofstream outfile("salary");  //creates outfile and connects "salary" to it
.........
Program2
..........
ifstream infile("salary");     //creates infile and connects "salary" to it
.........
```

Instead of using two program, one for writing data and another for reading data, we can use single program to do both operations on file.

```
.....
outfile.close();        //disconnect salary file from outfile
ifstream infile("salary");        // and connect to infile
.....
.....
infile.close();
```

Example

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

int main()
{
    char name[30];
    float cost;
    ofstream outf("item");

    cout<<"enter item name\n";
    cin>>name;
    outf<<name<< " ";

    cout<<"enter cost\n";
    cin>>cost;
    outf<<cost<<"\n";

    outf.close();
```

```
    ifstream inf("item");

    inf>>name;
    inf>>cost;

    cout<<"\nitem name:  "<<name;
    cout<<"\nitem cost:  "<<cost;

    inf.close();

   return 0;
}
```

Opening file using Open()

- This function is used to open multiple files that uses same stream object.

Syntax:-

file_stream_class stream_object;

stream_object.**open**("filename");

```
ofstream outfile;
outfile.open("Data1");
............
...........
outfile.close();
outfile.open("Data2");
............
............
outfile.close();
```

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ofstream fout;

    fout.open("country");
    fout << "usa\n";
    fout << "uk\n";
    fout << "india\n";
    fout.close();

    fout.open("capital");
    fout << "washinton\n";
    fout << "london\n";
    fout.close();

    const int n = 80;
    char line[n];

    ifstream fin;
```

```
fin.open("country");
  cout << "contents of country file\n";
  while(fin)
  {
     fin.getline(line, n);
     cout << line<<endl;
  }
  fin.close();

  fin.open("capital");
  cout << "contents of capital file\n";
  while(fin)
  {
     fin.getline(line, n);
     cout << line<<endl;
  }
  fin.close();
  return 0;
}
```

In C++, you cannot open a file for reading while it is still open for writing (in the same program).

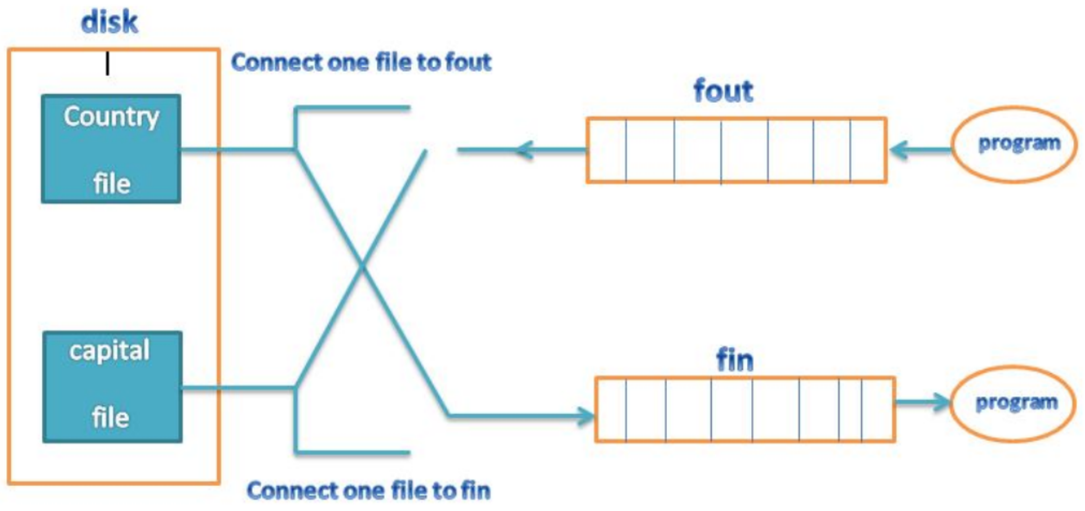


Fig: Streams working on multiple files

Detecting end-of-file

- Detecting eof condition is necessary for preventing any further attempt to read data from the file.

```
while(fin)
```

- here fin is **ifstream** object which returns a value 0 if any error occur in file including end-of-file condition.

```
if(fin.eof()!=0)
    { exit(1);}
```

- eof() is a member function of class ios. It returns non zero value if the end-of-file(eof) condition is encountered and zero otherwise.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
int main()
{
    const int SIZE = 80;
    char line[SIZE];
    std::ifstream fin1, fin2; // create two input streams

   fin1.open("country");
    fin2.open("capital");
```

```
    for (int i = 1; i <= 10; i++)
    {
        if (fin1.eof() != 0)
        {
            std::cout << "Exit from country \n";
            exit(1);
        }
        fin1.getline(line, SIZE);
        std::cout << "capital of " << line;

        if (fin2.eof() != 0)
        {
            std::cout << "\nExit from capital \n";
            exit(1);
        }
        fin2.getline(line, SIZE);
        std::cout << line << "\n";
    }
}
```

Open(): File Modes

- We can have two arguments in open(), to specify the file- mode.

```
stream-object.open("filename",mode);
```

- the second argument mode specifies the purpose for which file is opened and

- mode known as file mode parameter.

File Mode Parameters

Parameter	Meaning
ios::app	Append to end-of-file
ios::ate	Go to end of file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fail if file does not exist
ios::noreplace	Open fail if file already exist
ios::out	Open file for writing only
ios::trunc	Delete contents of file

- fstream fileout;
  fileout.open(“hello”,ios::app);

- void main()
  {
  fstream infile;
  Infile.open(“data_file”, ios::in|ios::out)

  }

note

1. Opening a file in **ios::out** mode also opens it in the **ios::trunc** mode by default.
2. Both **ios::app** and **ios::ate** take us to the end of the file when it is opened. The difference between the two parameters is that the **ios::app** allows us to add data to the end of the file only, while **ios::ate** mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.
3. The parameter **ios::app** can be used only with the files capable of output.
4. Creating a stream using **ifstream** implies input and creating a stream using **ofstream** implies output. So in these cases it is not necessary to provide the mode parameters.
5. The **fstream** class does not provide a mode by default and therefore, we must provide the mode explicitly when using an object of **fstream** class.
6. The *mode* can combine two or more parameters using the bitwise OR operator (symbol |) as shown below:

```
fout.open("data", ios::app | ios:: nocreate)
```

This opens the file in the append mode but fails to open the file if it does not exist.

File pointers and their Manipulations

Each file has two associated pointers known as the file pointers.

1. Input pointer/get pointer
2. Output pointer/put pointer

Default actions

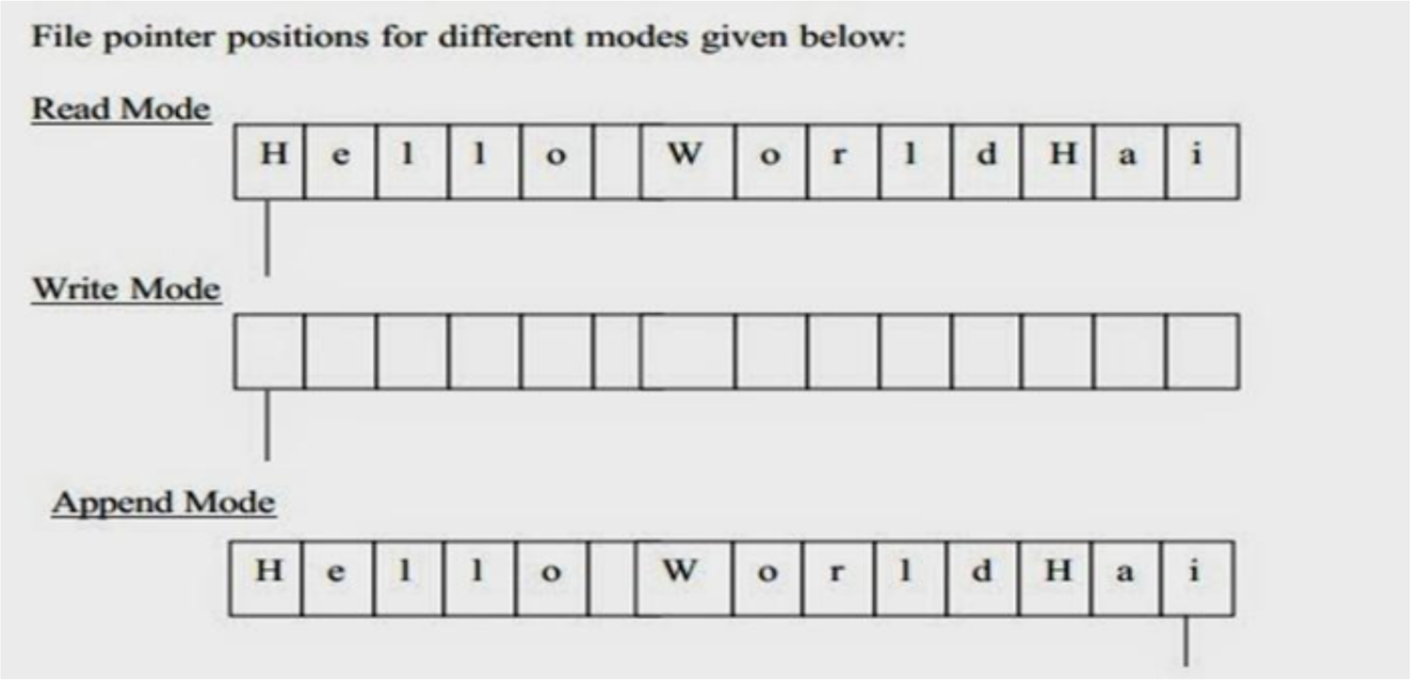
- In read only mode the Input pointer is automatically set at the beginning.
- In write only mode, the existing contents are deleted and output pointer is set at the beginning.
- In append mode, the output pointer moves to the end of file.

File opened in read mode:

- I/p pointer automatically set at beginning.

File opened in write mode:

- Existing contents deleted & O/p pointers sets at beginning.
- To add more data, file is opened in append mode.



Functions for manipulation of file pointer

- To move the file pointer to any other desired position inside the file. The file stream classes support the following to manage such situations:
- seekg()- moves get pointer to specified location
- seekp()- moves put pointer to specified location
- tellg()- gives the current position of get pointer
- tellp()- gives the current position of put pointer

Example

```
ofstream file1;
file1.open(“result”, ios::app);
int p=file1.tellp();
//The value of p will represent the number of bytes in the file.
Infile.seekg(10);
//Moves the file pointer to the byte number 10.
```

Specifying the offset

- seekg(offset, refposition);
- seekp(offset, refposition);

- Offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.

- refposition can be:

  ios::beg

  ios::cur

  ios::end

POINTER OFFSET CALLS :

Seek call	Action
fout.seekg(0,ios::beg);	-go to start
fout.seekg(0,ios::cur);	-stay at the current position
fout.seekg(0,ios::end);	-go to end of file
fout.seekg(m,ios::beg);	-move to (m+1)th byte in the file.
fout.seekg(m,ios::cur);	-go forward by m byte from the current position
fout.seekg(+m,ios::cur);	-go forward by m bytes from the current position
fout.seekg(-m,ios::cur);	-go backward by m bytes from the end

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
   long begin,end;
   ifstream myfile ("example.txt");
   begin = myfile.tellg();
   myfile.seekg (0, ios::end);
   end = myfile.tellg();
   myfile.close();
   cout << "size is: " << (end-begin) << " bytes.\n";
   }
```

Binary Files

- In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient

- There is no need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

- File streams include two member functions specifically designed to input and output binary data sequentially:

  write- is a member function of ostream inherited by ofstream

  read- is a member function of istream that is inherited by ifstream

Binary Files

- Objects of class fstream have both members.
- Their prototypes are:

  write (memory_block, size);

  read (memory_block, size);

Where,

memory_block - of type “pointer to char” (char*),

The size - an integer value

address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken

the number of characters to be read or written from/to the memory block

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

ifstream::pos_type size;
char *memblock;

int main ()
{
ifstream file("example.bin", ios::in | ios::binary | ios::ate);
if (file.is_open())
{
    size = file.tellg();
    memblock = new char[size];
    file.seekg(0, ios::beg);
    file.read(memblock, size);
    file.close();
    cout << "the complete file content is in memory";
delete[] memblock;
}
```

```
else
{
    cout << "Unable to open file";
}
return 0;
}
```