LECTURE-15

Pointer

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to "point to" y. Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable. To use pointer:

- We define a pointer variable.
- Assign the address of a variable to a pointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Example

```
#include<iostream>
using namespace std;
main()
{
	int a=10;  //normal variable
	int*p;    //declare pointer
	p = &a; //Assign the address of a variable "a" to a pointer "p"
	cout<<"a="<<*p;
}
```

Object Pointers

We have been accessing members of an object by using the dot operator. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. We can declare an object pointer just as a pointer to any other type of variable is declared. Specify its class name, and then precede the variable name with an asterisk. To obtain the address of an object, precede the object with the & operator, just as you do when taking the address of any other type of variable.

Example

```
#include<iostream>
using namespace std;
class A {
	int a;
	public:
		A(int x)
		{
			a=x;
		}
		int get()
		{
			return a;
		}
};

main( )
{
	A ob(120); //create object
	A *p; //create pointer to object
	p=&ob; //put address of ob into p
	cout <<"value using object: " <<ob.get( );
	cout <<"\n";
	cout <<"value using pointer: " <<p->get( );
```

```
}
```

Notice how the declaration: A *p; creates a pointer to an object of A. It is important to understand that creation of an object pointer does not create an object. It creates just a pointer to one. The address of ob is put into p by using the statement:

p=&ob;

Finally, the program shows how the members of an object can be accessed through a pointer.

Pointers to Derived Types

Pointers to base classes and derived classes are related in ways that other types of pointers are not. In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base. For example, assume that you have a base class called B and a class called D, which is derived from B. Any pointer declared as a pointer to B can also be used to point to an object of type D. Therefore, given

```
B *p; //pointer p to object of type B
B B_ob; //object of type B
D D_ob; //object of type D
```

Both of the following statements are perfectly valid:

```
p = &B_ob; //p points to object B
p = &D_ob; //p points to object D, which is an object derived from B
```

A base pointer can be used to access only those parts of a derived object that were

inherited from the base class. Thus, in this example, p can be used to access all elements of D_ob inherited from B_ob. However, elements specific to D_ob cannot be accessed through p.

Another point to understand is that although a base pointer can be used to point to a derived object, the reverse is not true. That is, you cannot access an object of the base type by using a derived class pointer.

Pointer To Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator:: * with the class name. For Example:

```
Class A
{
        Int m;
        Public:
        Void show()
        {
                Cout<<m;
        }
};
```

We can define a pointer to the member m as follows:

```
int A :: * ip = & A :: m
```

The ip pointer created thus acts like a class member in that it must be invoked with a class object. In the above statement. The phrase A :: * means “pointer - to - member of a class” . The phrase & A :: m means the “ Address of the m member of a class”. The following statement is not valid :

```
int   *ip=&m ; // invalid
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member. The pointer ip can now be used to access the m inside the member function (or friend function). Let us assume that “a” is an object of “ A” declared in a member function . We can
access "m" using the pointer ip as follows:

```
cout<< a . * ip;
cout<< a.m; ap=&a;
cout<< ap-> * ip;
cout<<ap->a;
```

The differencing operator ->* is used as to accept a member when we use pointers to both the object and the member. The dereferencing operator. .* is used when the object itself is used with the member pointer. Note that * ip is used like a member name. We can also design pointers to member functions which, then can be invoked using the differencing operator in the main as shown below.
(object-name.* pointer-to-member function) (pointer-to -object -> * pointer-to-member function). The precedence of ( ) is higher than that of .* and ->* , so the parenthesis are necessary.

Example

```
#include <iostream>
using namespace std;
class MyClass {
public:
    int x;
    double y;
};
main() {
    int MyClass::*ptr_x = &MyClass::x;
    double MyClass::*ptr_y = &MyClass::y;
    MyClass obj;
    obj.x = 10;
    obj.y = 3.14;
    cout << "x = " << obj.*ptr_x << endl;
    cout << "y = " << obj.*ptr_y << endl;
}
```

Example

```
#include<iostream>
using namespace std;
class A
{
       public:
           int m(int x)
              {
               return x * x;
           }
};
main()
{
       int (A::*p)(int) = &A::m; // Pointer to member function 'm' of
class 'A'
       A obj;
       int result = (obj.*p)(5); // Calling member function 'm'
through pointer 'p' with an instance 'obj'
       cout<<result;
}
```

Example

```
#include <iostream>
using namespace std;
class MyClass {
public:
    void func(int val) {
        cout << "Value: " << val << endl;
    }
};
main() {
    void (MyClass::*ptr_func)(int) = &MyClass::func;

    MyClass obj;
    (obj.*ptr_func)(10); // Calling member function func through
pointer
}
```

Example

```
#include <iostream>
using namespace std;
class MyClass {
public:
    int memberVariable;
    void memberFunction(int x) {
        cout << "Member function called with argument: " << x <<
std::endl;
    }
};
int main() {
    MyClass obj;
    // Pointer to a member variable
    int MyClass::*ptrToMemberVar = &MyClass::memberVariable;
    // Pointer to a member function
    void (MyClass::*ptrToMemberFunc)(int) = &MyClass::memberFunction;
    // Using pointer to member variable
    obj.*ptrToMemberVar = 42;
    cout << "Member variable value: " << obj.*ptrToMemberVar <<endl;
    // Using pointer to member function
    (obj.*ptrToMemberFunc)(10); // Calling member function}
```

Example

```
#include<iostream>
using namespace std;
class M
{
	int x;
	int y;
	public:
		void set_xy(int a,int b)
		{
			x=a;
			y=b;
		}
		friend int sum(M);
};
int sum (M m)
{
	int M :: *px= &M :: x; //pointer to member x
	int M :: *py= &M ::y;//pointer to y
	M *pm=&m;
	int s=m.*px + m.*py;
	return(s);
}
main ( )
{
	M n;
	void(M::*pf)(int,int)=&M::set_xy;//pointer to function set-xy
(n*pf)( 10,20);
	//invokes set-xy
	cout<<"sum=:"<<sum(n)<<endl;

	M *op=&n; //point to object n
	(op->* pf)(30,40); // invokes set-xy
	cout<<"sum="<<sum(n)<<endl;
}
```