Pointers

- Pointers are used to store the address of an data object.
- Pointers are used to avoid unnecessary copies when passing an arguments to functions.
- Pointer are used to support dynamic memory management.(allocating memory at run-time).

Syntax for defining the pointer

datatype *ptr_name;

<table>
<tr><th>Type of pointer</th><th colspan="3">Description</th></tr>
<tr><td>int *ptr ;</td><td colspan="3">A pointer which is declared but not defined is known as wild pointer</td></tr>
<tr><td>int *ptr=NULL;</td><td colspan="3">A pointer which is declared and initialized to Null is known as Null Pointer</td></tr>
<tr><td rowspan="2">void *ptr;</td><td colspan="3">A pointer whose data type is not specified is known as void pointer (or) generic pointer</td></tr>
<tr><td>int a=10
ptr=&a;
int a1=*(int*)ptr</td><td>float b=2.34f;
ptr=&b;
float b1=*(float*)ptr</td><td>char ch=‘x’;
ptr=&ch;
char ch1=*(char*)ptrr;</td></tr>
</table>

Declaration	Description
int *ptr ;	ptr is pointing to an integer variable i.e., it holds the address of integer variable
char *ptr ;	ptr is pointing to character variable

struct student{
int rno;
char name[30];
};
struct student *ptr;
	ptr is pointing to ***student*** variable which is an user-defined data types
	<u>Accessing members using pointer</u>:
	struct student s;
	ptr=&s;
	ptr->rno; (or) (*ptr).rno;
	ptr->name; (or) (*ptr).name;

class student{
int rno;
string name;
};
student *ptr;
	ptr is pointing to student objects.
	<u>Accessing members using pointer</u>:
	student s;
	ptr=&s;
	ptr->rno; (or) (*ptr).rno;
	ptr->name; (or) (*ptr).name;

Type Casting

Converting one type of object to another type of object is known as type casting. It can be done in two ways : implicit and explicit type casting.

	Implicit type casting	**Explicit type casting(using (type) operator)**
Pre-defined data type	bool -> char -> short int -> int -> unsigned int -> long -> unsigned long long -> float -> double -> long double	double a=2.34; int b=(int) a;

User-defined data type (classes)
: unrelated classes
class A {
public: int a;
};
class B {
public: int b;
};
	No Implicit type casting	A aobj;
		B bobj;
		aobj.a=10;
		Bobj.b=20;
		A *pa=&aobj;
		B *pb=&bobj;
		cout<<“ a value= “<< pa->a<<endl; //ok
		cout<<“ bvalue= “<< pb->b<<endl; //ok
		
		pa=(A*)&bobj; // forced casting
		pb=(B*)&aobj; // forced casting
		cout<<“ a value= “<< pa->a<<endl; //prints garbage
		cout<<“ bvalue= “<< pb->b<<endl; //prints garbage

	Type casting

User-defined data type (classes)
: related classes (inheritance hierarchy)
 	
class A {
   public: int a;
};
class B: public A {
   public: int b;
};
	A aobj;
	B bobj;
	
	aobj.a=10;
	bobj.a=15;
	bobj.b=20;
	
	A *pa=&aobj; // pointer to base
	B *pb=&bobj; // pointer to derived class
	
	cout << pa->a<<endl; //ok prints 10
	cout << pb->a << pb->b << endl; //ok prints 15 and 20
	
	pa=&bobj; // **upcasting**
	cout << pa->a<<endl; // prints 10
	cout<<pa->b<<endl; //error- class A doesn't know the class B members
	
	pb= (B*)&aobj; // **downcasting- forced casting**
	cout << pb->a<<endl; // ok prints 15;
	cout<< pb->b<<endl; // ok but prints garbage value

Binding

Attaching the function definition to a function call is known as Binding.

- Static binding (early binding)
- Dynamic binding(late binding)

<u>Type of a Object</u>

- The **static type** of the object is the type declared for the object while writing the code.
- The **dynamic type** of the object is determined by the type of the object to which it refers at run-time.

```
Class A { };
Class B: public A {};
int main(){
    A *p; // static type of p is A
    p= new B // dynamic type of p is B
}
```

Static and Dynamic Binding

- **Static binding(early binding)**: when a function invocation binds to the function definition based on the static type of objects.
  - Done at compile-time
  - Examples: Normal function calls, overloaded function calls, and overloaded operators.
- **Dynamic binding(late binding)** : When a function invocation binds to the function definition based on the dynamic type of objects
  - Done at run-time.
  - Examples: Function pointers , and virtual functions

	Static binding	**Dynamic Binding**
Time of Event occurred	Compile-time	Run-time
Information	All the information needed to call a function must be known at compile-time	All the information needed to call a function is known at compile-time
Advantage	Efficiency	Flexibility
Time	Fast Execution	Slow execution
Actual object	Actual object is not used for binding	Actual object is used for binding

Virtual functions

- Virtual function is a member function that can be redefined in other derived classes.
- Compiler ensures that calling of function definition is done based on the ***type of the object*** not the ***type of the pointer or reference***.
- A class that inherits a virtual function is called a polymorphic class.

```
#include <iostream>
using namespace std;
class A {
  public: void f() { cout << "Class A" << endl; }
};
class B: public A {
    public:  void f() { cout << "Class B" << endl; }
};
void g(A& arg) {
    arg.f();
}
int main() {
    B x;
    g(x);
    return 0;
}
```

Output
class A

```
#include <iostream>
using namespace std;
class A {
  public: virtual void f() { cout << "Class A" << endl; }
};
class B: public A {
    public:  void f() { cout << "Class B" << endl; }
};
void g(A& arg) {
    arg.f();
}
int main() {
    B x;
    g(x);
    return 0;
}
```

Output
class B

Example: Static vs Dynamic binding

```cpp
class B {
    public:
        void f(){
            cout<<"B::f()"<<endl;
        }
        virtual void g() {
            cout<<"B::g()"<<endl;
        }
};
class D : public B {
    public:
        void f() {
            cout<<"D::f()"<<endl;
        }
        void g() {
            cout<<"D::g()"<<endl;
        }
};
```

```cpp
int main(){
    B b;
    D d;
    B *pb=&b;
    B *pd=&d;

    B &rb=b;
    B &rd=d;

    b.f();
    b.g();
    d.f();
    d.g();

    pb->f();
    pb->g();
    pd->f();
    pd->g();

    rb.f();
    rb.g();
    rd.f();
    rd.g();
    return 0;
}
```

B *pb=&b; B *pd=&d; → Base class pointer can hold the address of derived class objects.

B &rb=b; B &rd=d; → We can create an alias to derived classes using base class reference.

- pb->f(); → Static binding
- pb->g(); → **Dynamic binding**
- pd->f(); → Static binding
- pd->g(); → **Dynamic binding**
- rb.f(); → Static binding
- rb.g(); → **Dynamic binding**
- rd.f(); → Static binding
- rd.g(); → **Dynamic binding**

```
B::f()
B::g()
D::f()
D::g()
B::f()
B::g()
B::f()
D::g()
B::f()
B::g()
B::f()
D::g()
```

Example : Overloaded functions in inheritance

```
class A {
   public:
      virtual void f() {
         cout<<“A::f()"<<endl;
      }
};
class  B: public A {
   public:
      void f(int x) {
         cout<<“B::f(int)"<<endl;
      }
};
class C: public B {
      public:
       void f() {
            cout<<“C::f()”<<endl;
       }
};
```

It is not a virtual function but, it hides A::f()

It is a virtual function

```
int main() {
         B bobj;
         C cobj;
         A* pa1=&bobj;
         A* pa2=&cobj;
      // bobj.f();
         pa1->f();
         pa2->f();
         return 0;
}
```

B::f is not virtual function, it hides A::f().So, compiler will not allow the function call bobj.f().

Abstract Class

- If a base class contains at least one pure virtual function then it is called Abstract class.
- A virtual function whose method signature is initialized to zero(=0) is known pure virtual function.

```
class A {
    public:
        void h(){….};                // non-virtual function
        virtual void f() { …..} ;  // virtual function
        virtual void g() = 0;     //  pure virtual function
};
```

- Abstract class Instantiation(creating an object) is not possible.

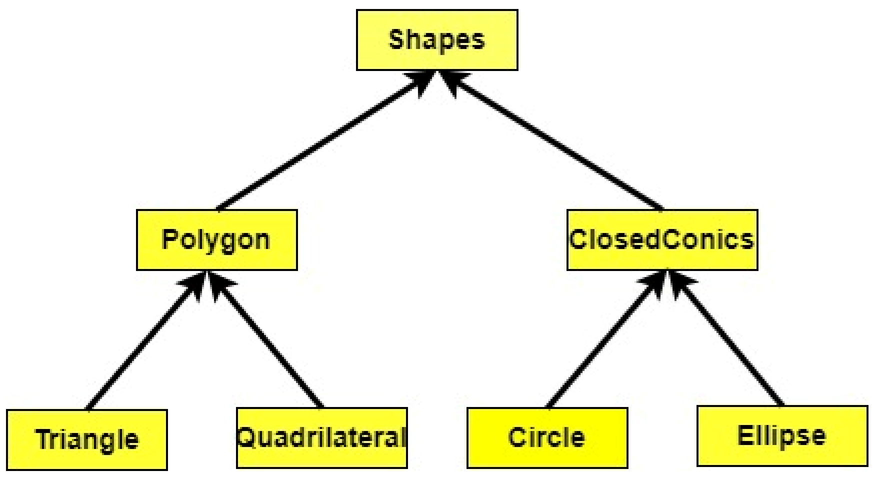
```
class A{
   public: virtual void f()=0;
};
class B:public A{
   public:
     void f() {cout<<B::f()”<<endl;}
      void g(){cout<<"B::g()"<<endl;}
 };
int main(){
   B bobj;
   bobj.f(); bobj.g();
   return 0;
}
```

Class A is Abstract class

pure virtual function must be overridded in derived class. Otherwise, derived class will become Abstract class

Example: Abstract class

```
class Shapes{
    public : virtual void draw()=0;
};
class Polygon:public Shapes{
    public : void draw() { cout<<"drawing using triangulation..."<<endl;}
};
class ClosedConics:public Shapes{
};
class Triange:public Polygon{
    public: void draw(){ cout<<"triangle: draw by lines"<<endl;}
};
class Quadrilateral: public Polygon{
    public: void draw(){ cout<<"quadrilateral: draw by lines"<<endl;};
};
class Circle:public ClosedConics{
    public: void draw(){cout<<“Circel: draw by breshenham's algorithm"<<endl;}
};
class Ellipse:public ClosedConics{
    public: void draw(){ cout<<"ellipse: draw by  ...."<<endl;}
};
```




```
int main(){
    Shapes* s[]={new Triange,
                 new Quadrilateral,
                 new Circle,
                 new Ellipse};
    for(int i=0;i<sizeof(s)/sizeof(Shapes*);i++)
        s[i]->draw();
    return 0;
}
```