Review

- Binary Search Tree
  - Binary Search Tree Property
  - Binary Search Tree Operations
    - Inorder walk
    - Preorder walk
    - Postorder walk
    - Search Tree
    - Insert a element to the Tree
    - Delete a element form the Tree

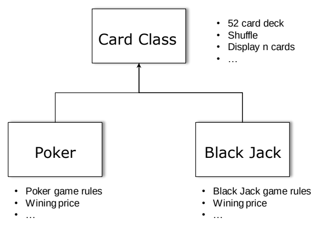
Preview

- Inheritance
  - Type of Inheritance
  - Syntax for Derived Class
  - Overriding Member Function in the Base Class
  - Using Member Functions
  - Casting Base-Class Pointers to Derived-class pointer
  - Using constructor and destructors in derived classes

Inheritance

- When creating a new class, <u>instead of writing completely new data and function members</u>, the programmer can designate that the new class is <u>to inherit data member and/or member function from the previously created class</u> –reusability !
- The <u>new class created is called a **Derived class**</u> and <u>the old class used as a base is called a **Base class**</u> in C++ inheritance terminology.

Inheritance



Inheritance

- The derived class will inherit all the features of the base class in C++ inheritance.
- But, <u>not all of them will be accessible</u> by the member functions of the derived class.
- <u>Only protected and public member</u> can be directly accessible by the members of the derived class
- The derived class can also add its own features, data etc.,

Inheritance

Some of the exceptions to be noted in C++ inheritance are as follows.

- The **constructor** and **destructor** of a base class are not inherited
- The **assignment operator** is not inherited
- the **friend functions** and **friend classes** of the base class are also not inherited

Types of Inheritances

C++ offers three kinds of inheritance

- **Public inheritance** – public member and protected member of the base class are inherited as a public member and protected member of the derived class. The private members of base class cannot be accessed by derived class members.
- **Private inheritance** – public and protected member of the base class become private member of the derived class.
- **Protected inheritance** – public and protected member of the base class become protected member of the derived class.

Syntax of Inherited Class

```
class <derived class name> : <type of inheritance> <Base class name>
{
};
```

```
// inher.cpp
#include <iostream>
using namespace std;

// Base class
class SomeBase {
private:
          int x;
public:
          void SetX(int i) {x = i;}
          SomeBase() { x =0;}
          void Hello() {cout <<"Hello" <<endl;}
          int GetX() {return x;}
};
//Derived class from class SomeBase
class SomeDerived : public SomeBase {
private:
          int x;
public:
          void SetX(int i){x = i;} //overrided function
          SomeDerived() {x =0;}
          int GetX(){return x;} // overrided function
};
```

Override Base-class Members in a Derived Class

- A derived class can <u>**override**</u> a base-class member function by supplying a new version that function with same name with same parameter list.
- If a function name is same but different parameter, it called **overloading**.

Using Member Functions

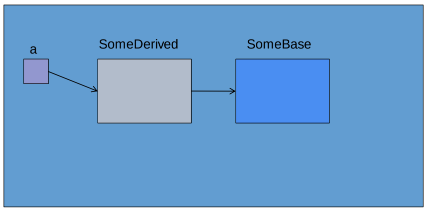
- <u>If there is no overridden function in the derived class</u>, <u>inherited functions can be accessed same way as its own member functions.</u>
- <u>If there is overridden functions in the derived class</u>, <u>need provide more information to access base class overridden functions</u>

```
// inher2.cpp
#include <iostream>
using namespace std;
// Base class
class SomeBase {
private:
          int x;
public:
          void SetX(int i) {x = i;}
          SomeBase() { x =0;}
          void Hello() {cout <<"Hello" <<endl;}
          int GetX() {return x;}
};

//Derived class from class SomeBase
class SomeDerived : public SomeBase {
private:
          int x;
public:
          void SetX(int i){x = i;} //overrided function
          SomeDerived() {x =0;}
          int GetX(){return x;} // overrided function
};

void main()
{
          SomeDerived a;
          a.SetX(5);
          a.SomeBase::SetX(6);
          cout << a.GetX()<<endl;
          cout << a.SomeBase::GetX()<<endl;
          a.Hello(); // can be used like its own function
}
```

Using Member Functions



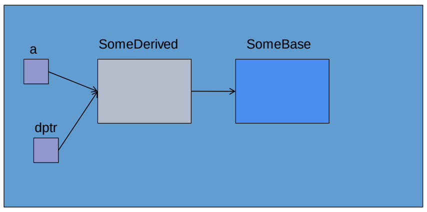
```
SomeDerived a
```

Casting Base-Class Pointers to Derived-Class Pointers

- When a derived object is created, its base class object is also created.
- By using pointer to a derived class object, base class public function can be used.
- But not from the pointer to base class object to a derived class object.

```
// inher3.cpp casting base-class pointers to derived-class pointers
#include <iostream>
using namespace std;
class SomeBase {
   int x;
public:
    int GetX() {return x;}
    void SetX(int a) {x =a;}
    SomeBase() {x=0; cout<<"Base object is created"<<endl;}
    ~SomeBase() { cout << "Base object is destroyed"<<endl;}
};
class SomeDerived : public SomeBase {
          int x;
public:
          int GetX() {return x;}
          void SetX(int a) {x = a;}
          SomeDerived() {cout<<"Derived object is created"<<endl;}
          ~SomeDerived() {cout <<"Derived object is destroyed"<<endl;}
};
void main()
{
          SomeDerived *dptr;
          SomeDerived w;
          w.SetX(3);
          dptr = &w;
          cout <<dptr->SomeBase::GetX()<<endl;
          cout<<dptr->GetX()<<endl;
          cout <<w.GetX() <<endl;
}
```

Casting Base-Class Pointers to Derived-Class Pointers



```
SomeDerived a
SomeDerived *dptr;
dptr = &a;
```

Indirect Base Class

- It might be possible to create a derived class from other derived class.
- More than two object space need to be created in this case since a base class for a derived class is derived from other class.

Indirect Base Class

```
// inher4.cpp casting base-class pointers to derived-class
// pointers  and to derived-derived class
#include <iostream>
using namespace std;
class B {
   int x;
public:
   int f() {return x;}
   void j(int a) {x =a;}
   B() {x=0; cout<<"Base object is created"<<endl;}
   ~B() { cout <<" Base object is deallocated"<<endl;}

};
class D : public B {
   int x;
public:
   D() {x=1; cout<<"Derived object is created"<<endl;}
  ~D() {cout<<"Derived object is deallocated"<<endl;}
   int f() {return x;}
   void j(int a) {x = a;}
};
```

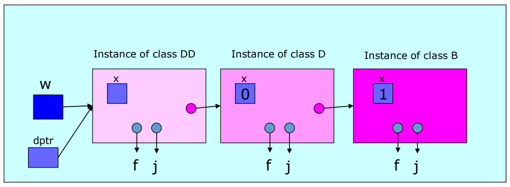
Indirect Base Class

```
class DD : public D {
   int x;
public:
   DD() {cout<<"Derived-and-derived object is created"<<endl;}
   ~DD() {cout<<"Derived-and-derived object is deallocated"<<endl;}

   int f() {return x;}
   void j(int a) {x = a;}
};
void main()
{
          DD *dptr;
          DD w;

          w.j(2);
          dptr = &w;
          cout <<dptr->B::f()<<endl; // function from base class B
          cout <<dptr->D::f()<<endl; // function from derived class D
          cout<<dptr->f()<<endl; // function from derived class DD
          cout <<w.f() <<endl; // function from derived class DD
}
```

Indirect Base Class



```
DD *dptr;
DD w;
dptr = &w;
```

Using constructors and destructors in derived classes

```
// inher5.cpp using destructor and constructor in derived class
#include <iostream>
using namespace std;
class B {
    int x;
public:
    int f() {return x;}
    void j(int a) {x =a;}
    B(int a) {x=a; cout<<"Base object is created"<<endl;}
    ~B() { cout <<"Base object is destroyed now "<<endl;}
};
class D : public B {
   int x;
public:
    int f() {return x;}
    void j(int a) {x = a;}
    // derived constructor pass a parameter to the base class constructor
    D(int a, int b):B(b) {x = a; cout<<"Derived object is created"<<endl;;}
    ~D(){cout <<"Derived class is destroyed now."<<endl;}
};
void main()
{
            D *dptr;
            D w (1,0);

            dptr = &w;
            cout <<dptr->B::f()<<endl;
            cout<<dptr->f()<<endl;
}
```

Using constructors and destructors in derived classes

```
// inher6.cpp using destructor and constructor in derived class
#include <iostream>
using namespace std;
class B {
    int x;
public:
    int f() {return x;}
    void j(int a) {x =a;}
    B(int a) {x=a; cout<<"Base object is created"<<endl;}
    ~B() { cout <<"Base object is destroyed now "<<endl;}
};

class D : public B {
    int x;
public:
    int f() {return x;}
    void j(int a) {x = a;}
    // derived constructor pass a parameter to the constructor class B
    D(int a, int b):B(b) {x = a; cout<<"Derived object is created"<<endl;;}
    ~D(){cout <<"Derived object is destroyed now."<<endl;}
};
```

Using constructors and destructors in derived classes

```
class DD : public D {
   int x;
public:
   int f() {return x;}
   void j(int a) {x = a;}
  // DD class construcor pass two parameter to the
  // construtor class D
   DD(int a, int b, int c):D(b, c)
     {x = a; cout<<"Derived-derived object is created"<<endl;;}
   ~DD()
     {cout <<"Derived-derived object is destroyed now."<<endl;}
};
void main()
{
            DD *dptr;
            DD w (2,1,0);

            dptr = &w;
            cout <<dptr->B::f()<<endl;
            cout <<dptr->D::f()<<endl;
            cout<<dptr->f()<<endl;
}
```

Using constructors and destructors in derived classes

How can you create an derived object independent from base-class parameters?

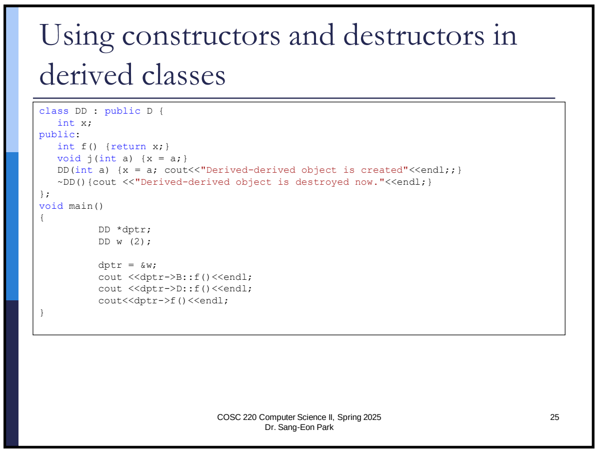
Using constructors and destructors in derived classes

```
// inher7.cpp: using default constructor for create derived object without
// concerning the parameter of  base-class destructor and
// constructor
#include <iostream>
using namespace std;
class B {
   int x;
public:
    int f() {return x;}
    void j(int a) {x =a;}
    B(int = 0);
  ~B() { cout <<"Base object is destroyed now "<<endl;}
};
B::B(int a)
{
            x=a;
            cout<<"Base object is created"<<endl;
}

class D : public B {
   int x;

public:
   int f() {return x;}
   void j(int a) {x = a;}
   D(int =1);
   ~D(){cout <<"Derived object is destroyed now."<<endl;}
};

D::D(int a)
{
            x = a;
            cout<<"Derived object is created"<<endl;
}
```



Using constructors and destructors in derived classes

```
class DD : public D {
   int x;
public:
   int f() {return x;}
   void j(int a) {x = a;}
   DD(int a) {x = a; cout<<"Derived-derived object is created"<<endl;;}
   ~DD(){cout <<"Derived-derived object is destroyed now."<<endl;}
};
void main()
{
          DD *dptr;
          DD w (2);

          dptr = &w;
          cout <<dptr->B::f()<<endl;
          cout <<dptr->D::f()<<endl;
          cout<<dptr->f()<<endl;
}
```