Inheritance：C++支持多继承和多级继承

- Inherit members (attributes and functions) from one class
  - **Base class** (parent) 基类（父类）
  - **Derived class** (child) 派生类（子类）
- C++ supports multiple inheritance and multilevel inheritance

```
class Base
{
  public:
    int a;
    int b;
};
class Derived: public Base
{
  public:
    int c;
};
```

```
class Derived: public Base1, public Base2
{
   ...
};
```

继承中的 constructors、destructors

- To instantiate a derived class object
  - Allocate memory
  - Derived constructor is invoked
    - Base object is constructed by a base constructor
    - Member initializer list initializes members
    - To execute the body of the derived constructor

```
class Derived: public Base
{
  public:
    int c;
    Derived(int c): Base(c - 2, c - 1), c(c)
    {
       ...
    }
};
```

若不写，则会调父类的默认构造函数

先初始化完父类再子类

当派生类的构造函数调用后，
①基类对象的构造函数工作
②初始化成员
③执行派生类的构造函数

- The destructor of the derived class is invoked first,
- Then the destructor of the base class.

先析构子类再父类

继承不能继承父类的友元，需要 static_cast<const Base&>(obj) 将子类转化成父类。

Access Control

public 成员能在任意地方被访问
private 成员能在类内与友元访问
protected 成员能被类内、友元与子类访问

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

- Public members
  - Accessible anywhere
- Private members
  - Only accessible to the members and friends of that class

```
class Person {
  private:
    int n; // private member
  public:
    // this->n is accessible
    Person() : n(10) {}
    // other.n is accessible
    Person(const Person& other) : n(other.n) {}
    // this->n is accessible
    void set(int n) {this->n = n;}
    // this->n and other.n are accessible
    void set(const Person& other) {this->n = other.n;}
};
```

Member Access

```
// a non-member non-friend function
void compare(Base& b, Derived& d)
{
   // b.n++; // Error
   // d.n++; // Error
}
```

- Protected members
  - Accessible to the members and friends of that class
  - Accessible to the members and friends of the derived class

```
class Base
{
  protected:
    int n;
  private:
    void foo1(Base& b)
    {
       n++; // Okay
       b.n++; // Okay
    }
};
```

```
class Derived : public Base
{
   void foo2(Base& b, Derived& d)
   {
      n++; //Okay
      this->n++; //Okay
      //b.n++; //Error.
      d.n++; //Okay
   }
};
```

Public Inheritance

- Public members of the base class
  - Still be public in the derived class
  - Accessible anywhere
- Protected members of the base class
  - Still be protected in the derived class
  - Accessible in the derived class only
- Private members of the base class
  - Not accessible in the derived class

Protected Inheritance

- **Public** members and **protected** members of the base class
  - Be **protected** in the derived class
  - Accessible in the derived class only
- Private members of the base class
  - Not accessible in the derived class

Private Inheritance

- **Public** members and **protected** members of the base class
  - Be **private** in the derived class
  - Accessible in the derived class only
- Private members of the base class
  - Not accessible in the derived class

Virtual Functions (虚函数)

```
class Person
{
  public:
    void print()
    {
      cout << "Name: " << name << endl;
    }
};
class Student: public Person
{
  public:
    void print()
    {
      cout << "Name: " << name;
      cout << ". ID: " << id << endl;
    }
};
```

```
Person * p = new Student();
p->print(); // call Person::print()
```

若想 p->print() 执行的是子类的方法，可在父类的同名函数前加 virtual.

```
virtual void print()
```

或者在子类同名方法后加 override.

```
void print() override
```

加上 virtual 后该函数能被子类重写.

- But if we define print() function as a virtual function, the output will be different.
- **Static** binding: the compiler decides which function to call
- **Dynamic** binding: the called function is decided at runtime.

- Keyword virtual makes the function virtual for the base and all derived classes.

因为若加上了 virtual，在函数前会加上一个指针，指向一个函数表，让编译器在运行时寻找该调用什么函数

虚函数是动态绑定的

Destructor 一定是虚函数

纯虚函数与抽象类.

纯虚函数是只有函数声明的虚函数，有纯虚函数的类叫抽象类，

纯虚函数必须由子类实现，抽象类不能实例化对象.

Inheritance and Dynamic Memory Allocation (继承中的动态申请内存)

需要在子类中重新定义拷贝构造和赋值操作.

```
class MyMap: pubic MyString
{
   char * keyname;
  public:
   MyMap(const char * key, const char * value)
   {
   ...
   }
   MyMap(const MyMap & mm): MyString(mm.buf_len, mm.characters)
   {
   //allocate memory for keyname
   //and hard copy from mm to *this
   }
   MyMap & operator=(const MyMap &mm)
   {
     MyMap::operator=(mm);
     //allocate memory for keyname
     //and hard copy from mm to *this
     return *this;
   }
};
```