



華中科技大学

# 第6章 继承与构造

许向阳

xuxy@hust.edu.cn





# 大纲

- 6.1 单继承类
- 6.2 继承方式
- 6.3 成员访问
- 6.4 构造与析构
- 6.5 父类和子类
- 6.6 派生类的内存布局



# 6.1 单继承类

## 类的层次性、继承和派生

- 继承是C++类型演化的重要机制
- 通过继承，新类具有原有类型的属性和行为
- 新类只需定义原有类型没有的数据成员和函数成员
- 接受成员的新类称为**派生类**
- 提供成员的原有类型称为**基类**
- C++既支持单继承又支持多继承
- 单继承只能获取一个基类的属性和行为
- 多继承可获取多个基类的属性和行为



# 6.1 单继承类

单继承的定义格式

```
class <派生类名>: 【<派生控制符>】 <基类名>
{
    <派生类新定义成员>
    <派生类重定义基类同名的数据和函数成员>
    <派生类声明恢复基类成员访问权限>
};
```

<派生控制符>指明派生类采用什么方式从基类继承成员。

**private:** 私有继承基类

**protected:** 保护继承基类

**public:** 公有继承基类



# 6.1 单继承类

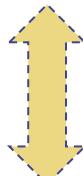
## 缺省的派生控制符

```
class <派生类名>: <基类名>
{ ..... };
```

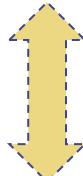
```
class <派生类名>: private <基类名>
{ ..... };
```

```
struct <派生类名>: <基类名>
{ ..... };
```

```
struct <派生类名>: public <基类名>
{ ..... };
```



等同写法



- 使用**class**定义派生类时，缺省的派生控制符为**private**；  
使用**struct**定义派生类时，缺省的派生控制符为**public**；  
与 基类是用**struct** 还是用**class**定义的无关；
- 使用 **union** 定义的类，既不能作为基类，也不能用 **union** 定义派生类。



# 6.1 单继承类

```
class A
{
protected:
    int x;
private:
    int y;
public:
    void setx(int m) { x=m; }
    void sety(int n) { y=n; }
    int getx() const { return x; }
    int gety() const { return y; }
};
```

`sizeof(A) = 8`

```
void f()
{
    A t;
    t.x=10; // error
    t.y=20; // error
    t.setx(10);
    t.sety(20);
}
```



# 6.1 单继承类

```
class B : protected A
{
    private:
        int b1,b2;
    public:
        void setBx(int m) { x=m; }
        void setBy(int n) { sety(n); }
        int getBx() const { return x; }
        int getBy() const {
            return gety();
        }
        int getsum() { return x+gety(); }
};

B b;
cout<<" x= "<< b . getBx() << endl;
```

A :    `sizeof(A) = 8`  
`protected : x`  
`private : y`  
`public: setx`  
            `sety`  
            `getx`  
            `gety`

**B继承了A的哪些成员？**  
**哪些是可访问的？**  
**可访问的权限是什么？**

`sizeof(B) = 16`





# 6.1 单继承类

A a;    B b;

B 对象的内存布局是什么样的？

监视 1

名称	值
⊕ a	{x=-858993460 y=-858993460 }
⊕ b	{b1=-858993460 b2=-858993460 }

名称	值
⊕ a	{x=-858993460 y=-858993460 }
⊕ x	-858993460
⊕ y	-858993460
⊕ b	{b1=-858993460 b2=-858993460 }
⊕ A	{x=-858993460 y=-858993460 }
⊕ x	-858993460
⊕ y	-858993460
⊕ b1	-858993460
⊕ b2	-858993460

名称	值
⊕ &b.x	0x00c3fb9c
⊕ &b.y	0x00c3fba0
⊕ &b.b1	0x00c3fba4
⊕ &b.b2	0x00c3fba8
⊕ &b	0x00c3fb9c {b1=-858993460 b2=-858993460 }

void setBy(int n) { sety(n); }  
b. setBy(10) ; 的运行过程

B \* const this = &b;  
this 作为第一参数

调用 sety 第一参数 this 指向谁?  
A\* const this = B \* const this;





## 6.2 继承方式

类定义中：

- public 成员： 可被任何类访问，公开的
- protected成员： 可被本类及其派生类访问
- private成员： 只限于本类访问

派生控制：指明基类的成员在派生类的访问方式



## 6.2 继承方式

继承类型	基类 成员特性	派生类 成员特性
公有继承	公有	公有
	保护	保护
	私有	不可访问
保护继承	公有	保护
	保护	保护
	私有	不可访问
私有继承	公有	私有
	保护	私有
	私有	不可访问

派生类，实际上继承了基类的所有成员

相当于将基类的所有成员定义拷贝到派生类中。

只是在写程序时，派生类不可访问基类的私有成员。





## 6.2 继承方式

```
class B : protected A
{ public:
    void setBx(int m) { x=m; }
    void setBy(int n)
    { y=n; //不能访问私有成员
      sety(n); }
    int getBx() const { return x; }
    int getBy() const {
        // 不能有 return y;
        return gety();
    }
    int getsum() { return x+gety(); }
};

B b;
cout<<" x=" << b . getBx() << endl;
cout<<" x=" << b . getx() << endl;
cout<<" x=" << b .A:: getx() << endl;
```

A :  
protected : x  
private : y  
public: setx  
          sety  
          getx  
          gety

B :  
protected :  
          x  
          setx  
          sety  
          getx  
          gety  
public:  
          setBx  
          setBy  
          getBx  
          getBy  
          getsum

继承的成员各有什么访问权限？

哪些语句不正确？





## 6.2 继承方式

派生类声明恢复或修改基类成员访问权限

```
Class B : private A
{
public:
    A::getx;      // getx后面没有括号
protected:
    A::x;      // 数据成员
                // 等价于 using A::x;
private:
    A::y;        // 不可访问 A类的私有成员
}cout<<" x= "<< b . getx() <<endl;
```



## 6.2 继承方式

### 派生类声明恢复或修改基类成员访问权限

- 对于基类的私有成员，在派生类中不可访问，也不能恢复或修改其访问权限；
- 对于基类的保护/公有成员，可以修改其访问权限
- 语句格式：  
**基类名::成员名 或 using 基类名::成员名**
- 对于修改或者恢复权限的成员，在派生类中不能定义同名成员。



## 6.3 成员访问

Q: 在派生类和基类有同名成员时，如何区分同名成员？

- 标识符的作用范围越小，被访问到的优先级越高。  
要访问作用范围更大的标识符，用“**类名::**”进行限定。
- 面向对象作用域关于标识符的作用范围从小到大：
  - ①作用于函数成员内（局部变量）；
  - ②作用于类或者派生类内；
  - ③作用于基类内（基类的成员要在派生类可访问）；
  - ④作用于虚基类内。



## 6.3 成员访问

```
class A {  
public: int x;  
    int getx() {cout<< "class A" << endl; return x;}  
};  
class B : public A {  
public: int x;  
    int getx() { cout<< "class B" << endl; return x; }  
    int setx(int x) {  
        this->x=x; // (*this).x=x; B::x=x;  
        A::x=2*x;      A::x = 2 * x;  
    };  
    B b;          B::A::x = 2 * x;  
    Z=b.getx();    this->A::x = 2*x;  
    Z=b.A::getx();  (*this).A::x = 2*x;  
                    ((A*) this)->x = 3 * x;
```





## 6.3 成员访问

```
class A {  
public:    int x;  
    int getx() {cout<< "class A" <<endl;  return x;}  
};  
class B : public A  
{    public:  
        int getx() { cout<< "class B" <<endl;  return x; }  
};  
B b;  
Z=b.getx();  
Z=b.A::getx(); // 为何派生控制中要有public, 此处才正确?  
  
A *p= &b;      // A 类的指针, 执向了派生类对象  
Z= p->getx(); // 显示什么?          class A
```





## 6.3 成员访问

```
class A {  
public:    int x;  
    int getx() {cout<< "class A" << endl;  return x;}  
};
```

```
class B : private A  
{    public:  
        int getx() { cout<< "class B" << endl;  return x; }  
};
```

// 在B类中，可以访问基类的保护成员

```
class C {  
private:  A  a1; // 在C类中，不能访问 A的保护成员  
          // 即不能有 a1/a2/a3. protected member  
public:   A  a2; // 在C类外，不能访问C的私有成员  
protected: A  a3;  
};
```





## 6.3 成员访问

```
class A {  
public:  
    int a;  
    A(int v) { a = v; };  
    void fa() { cout<<"A a=" << a << endl; }  
};  
class B :public A{  
public :  
    int b;  
    B(int v) :A(v) { b = 2 * v; }  
    void fb() {  
        fa();  
        this->fa();  
        A::fa();  
        cout << "B b=" << b << endl;  
    }  
};
```

派生类中访问基类函数，  
函数名前不需要加对象  
或对象指针

```
fa();  
mov     ecx, dword ptr [this]  
call    A::fa (OFE10AAh)  
this->fa();  
mov     ecx, dword ptr [this]  
call    A::fa (OFE10AAh)  
A::fa();  
mov     ecx, dword ptr [this]  
call    A::fa (OFE10AAh)
```





## 6.4 构造与析构

- ◆ 创建一个对象时，调用相应的构造函数进行初始化
- ◆ 派生类不能继承基类的构造函数、析构函数
- ◆ 派生类有自己的构造函数和析构函数

派生类构造函数的格式：

派生类名(参数表1): 基类名(参数表2), 派生类成员的构造

{ ..... 构造函数体 ..... }

- ◆ 若无“基类名(参数表2)”，执行基类中无构造函数
- ◆ 基类的构造函数最先执行
- ◆ 基类中的构造、析构函数都要有相应的访问权限，  
不能是 private



## 6.4 构造与析构

```
class Animal {  
public: int age;  
private: int weight;  
public:  
    Animal(int a,int w) {  
        cout<<" Animal with parameter"<<endl;  
        age = a;      weight =w;      }  
    Animal() {  
        cout<<" Animal without parameter"<<endl;  
        age = 1; weight =1;      }  
    ~Animal() {cout<<"destructing Animal "<<endl; }  
    void setage(int a=0)      { age=a; }  
    int   getage( )   { return age; }  
};
```



## 6.4 构造与析构

```
class Dog : public Animal {      Dog MyDog("警犬",10,100);  
private:  
    char pinzhong[10];  
public:  
    Dog(char *pz, int a, int w) : Animal( a, w )  
    {  
        strcpy(pinzhong,pz);  
        cout<<" constructing Dog "<<endl;  
    }  
    ~Dog() { cout<<"destructing Dog "<<endl; }  
    void set_dogage(int a)  
    { setage(a); }  
};
```



重要

## 6.4 构造与析构

- ◆ 单继承派生类只有一个基类或虚基类，根据4个优先级别容易确定构造顺序：
  - ① 调用派生类的虚基类的构造函数；无论列出否，总会被执行；
  - ② 调用派生类的基类的构造函数，无论列出与否，总会被执行；
  - ③ 按照派生类数据成员声明的顺序，依次初始化数据成员或调用相应构造函数，对象成员无论列出与否总会被构造；
  - ④ 最后执行派生类的构造函数体。
- ◆ 基类、对象成员不列出时自动调用无参构造函数。
- ◆ 若虚基类或基类只定义了带参数的构造函数，则派生类需要使用带参数的构造函数。

class D : virtual 派生控制 baseC； baseC 称为虚基类





## 6.4 构造与析构

构造函数与析构函数的执行顺序

```
class A{  
private: int a;  
public:  
    A():a(8) { cout<<a<<" " ; }  
    A(int x):a(x) { cout<<a<<" " ; }  
    ~A(){ cout<<a; }  
};  
class B: private A{  
private: int b, c;  
    const int d;           //有只读成员d, 故B必须定义构造函数  
    A x, y, z;  
public:  
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {  
        c=v; cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值  
    }  
    ~B(){ cout<<"D"; }  
};  
void main(void){ B u(1); }  
//输出结果: 1 2 3 8 1 1 1 C D 8 3 2 1
```

C6\_init\_order





## 6.4 构造与析构

```
class A{  
private: int a;  
public:  
    A():a(8){ cout<<a<<" " ; }  
    A(int x):a(x){ cout<<a<<" " ; }  
    ~A(){ cout<<a; }  
};  
class B: private A{  
private: int b, c;  
    const int d;           //有只读成员d, 故B必须定义构造函数  
    A x, y, z;  
public:  
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {  
        c=v; cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值  
    }  
    ~B(){ cout<<"D"; }  
};  
void main(void){ B u(1); }
```

实验

Q: ← 能否将public 改为 protected?

Q: ← 能否改为 private?

A: 对本程序而言，可改为  
protected，但不能改为 private.





## 6.4 构造与析构

```
class A{  
private: int a;  
public:  
    A():a(8) { cout<<a<<" " ; }  
    A(int x):a(x) { cout<<a<<" " ; }  
    ~A(){ cout<<a; }  
};
```

```
class B: private A{  
private: int b, c;  
    const int d;           //有只读成员d, 故B必须定义构造函数  
    A x, y, z;  
public:
```

```
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {  
        c=v; cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值  
    }  
    ~B(){ cout<<"D"; }  
};  
void main(void){ B u(1); }
```

实验

Q: ← 能否删除无参的构造函数 A() ?

A: 不能, 有语法错误  
当构造 B 中对象成员 Z 时,  
没有合适的默认构造函数可用





## 6.4 构造与析构

```
class A{
```

```
private: int a;
```

```
public:
```

```
    A():a(8) { cout<<a<<" " ; }
```

```
    A(int x):a(x) { cout<<a<<" " ; }
```

```
    ~A(){ cout<<a; }
```

```
};
```

```
class B: private A{
```

```
private: int b, c;
```

```
    const int d; //有只读成员d, 故B必须定义构造函数
```

```
    A x, y, z;
```

```
public:
```

```
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {
```

```
        c=v; cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值
```

```
}
```

```
    ~B(){ cout<<"D"; }
```

```
};
```

```
void main(void){ B u(1); }
```



实验

Q: ← 能否删除 A(v)?

A: 可以  
自动调用无参构造函数  
显示的第一个数字是 8





## 6.4 构造与析构



```
class A{  
private: int a;  
public:  
    A():a(8) { cout<<a<<" " ; }  
    A(int x):a(x) { cout<<a<<" " ; }  
    ~A(){ cout<<a; }  
};  
class B: private A{  
private: int b, c;  
    const int d;           //有只读成员d, 故B必须定义构造函数  
    A x, y, z;           Q: 若将 int b,c ; 移到 A x, y,z; 之下, 结果如何?  
public:  
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {  
        c=v;   cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值  
    }  
    ~B(){ cout<<"D"; }  
};  
void main(void){ B u(1); }
```

1 -858993459 -858993458 8 1 1  
-858993460 C D 8 -858993458  
-858993459 1





## 6.4 构造与析构



```
class A{  
private: int a;  
public:  
    A():a(8){ cout<<a<<" " ; }  
    A(int x):a(x){ cout<<a<<" " ; }  
    ~A(){ cout<<a; }  
};  
class B: private A{  
private: int b, c;  
    const int d;           //有只读成员d, 故B必须定义构造函数  
    A x, y, z;  
public:  
    B(int v):b(v), y(b+2), x(b+1), d(b), A(v) {  
        c=v; cout<<b<<c<<d; cout<<"C "; //b, c可再次赋值  
    }  
    ~B(){ cout<<"D"; }  
};  
void main(void){ B u(1); }  
//输出结果: 1 2 3 8 1 1 1 C 8 3 2 1
```

Q: 删除B的析构函数, 结果如何?





## 6.4 构造与析构

若构造函数的参数都有缺省值，能否当成一个无参的构造函数？

**class A{  
private: int a, b;  
public: A(int x=0, int y=0) { a = x; b = y; }  
};**

**A o1(10,20);  
A o2(10);  
A o3;  
A\* p = new A();**

若构造函数的参数都有缺省值，可以当成无参的构造函数！  
对象构造使用缺省值！





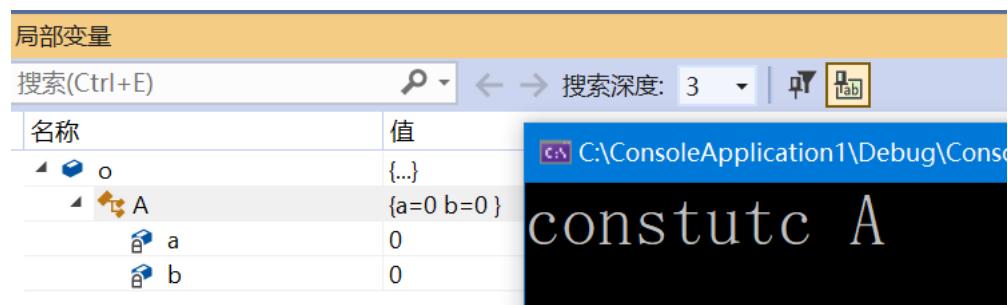
## 6.4 构造与析构

B 类中是否一定要定义构造函数？



```
class A{  
private: int a, b;  
public: A(int x=0, int y=0) { a = x; b = y;  
    cout << "constuct A" << endl; }  
};
```

```
class B : A {  
};  
int main()  
{ B o; }
```



可以不定义，会自动使用 A 的无参构造函数！



## 6.4 构造与析构

- ◆ 变量 q引用一个对象p时，由对象p完成构造和析构。

A p(...);      A &q=p;

不需要考虑q空间的释放，q本身只占4个字节，q的生命周期时，自动释放这4个字节。

- ◆ 若被q指向的对象是用new生成的，

用delete q析构对象，否则将产生内存泄漏。

A \*q = new A(10);      delete q;

- ◆ q引用的对象是用new生成的，

应使用 delete &q析构对象，否则将产生内存泄漏。

A & q = \* new A(10);      delete &q;





## 6.5 父类和子类

- 如果派生控制为public， 派生类称为基类的**子类**，  
基类则称为派生类的**父类**；
- 父类指针可以**直接**指向子类对象  
父类可以**直接**引用子类对象  
无须通过强制类型转换，  
编译时按父类说明的权限访问成员。

```
class A { ... };  
A *pa;  
pa = &pb;  
A &ra = pb;
```

直接转换

```
class B: public A {.....};  
B pb;  
pa=(A *)&pb;  
A &ra=(A &)pb;  
A &ra=*(A *)&pb;
```

强制类型转换

**非父子类**  
可通过强制类型  
转换，

- 基类指针指向  
派生类对象
- 基类引用派生  
类对象





## 6.5 父类和子类

```
class POINT{
    int x, y;
public:
    int getx( ){ return x; }    int gety( ){ return y; }
    void show( ){ cout<<"Show a point\n"; }
    POINT(int x,int y): x(x), y(y){ }
};

class CIRCLE: public POINT{//公有继承， 基类和派生类构成父子关系
    int r;
public:
    int getr( ){ return r; }
    void show( ){ cout<<"Show a circle\n"; }
    CIRCLE(int x, int y, int r):POINT(x,y){ CIRCLE::r=r; }
} c(3,7,8);

void main(void){
    POINT *p=&c; //父类指针p指向子类对象， 不用类型转换
    p=(POINT*)&c;
    cout<<c.getr()<<p->getx(); //不能用p->getr(),
                                    //编译时无POINT::getr()
    p->show(); //p指向子类对象， 但调用的是POINT::show()
}
```





## 6.5 父类和子类

### ◆ 在派生类函数成员内部 (不论 父子关系)

基类指针可以指向该派生类对象

基类被等地当作派生类的父类

### ◆ 在派生类的友元函数中

友元内部的基类指针也可以直接指向派生类对象，

即对派生类的友元， 基类被等地当作派生类的父类。

### ◆ 在上述两种情况下， 基类引用可直接引用派生类对象。



## 6.5 父类和子类

Q：为什么基类指针要指向派生类对象?  
or 为什么基类要引用子类对象?

在父类中有虚函数时，对虚函数的访问（第8章）

- 通过父类指针调用虚函数时进行晚期绑定；
- 根据对象的实际类型绑定到合适的成员函数；
- 父类指针实际指向的对象的类型不同；
- 虚函数绑定的函数的行为不同，从而产生多态。



## 6.6 派生类的内存布局

- 在派生类对象的存储空间中包含了基类的数据成员。
- 在构造派生类对象之前，首先构造或初始化基类对象的存储空间，作为派生类对象存储空间的一部分。
- 在计算派生类对象存储空间时  
    基类和派生类的**静态数据成员**都不应计算在内。



# 6.6 派生类的内存布局

```
#include <iostream>
using namespace std;
class A{
private:
    int h, i, j;
    static int k;
};
class B: A{ //等于class B: private A
    int m, n, p;
    static int q;
};
int A::k=0; //静态数据成员必须初始化
int B::q=0;
int main(void){
    cout<<"Size of int=<<sizeof(int)<<"\n";
    cout<<"Size of A=<<sizeof(A)<<"\n";
    cout<<"Size of B=<<sizeof(B)<<"\n";
    return 0;
}
```

输出

Size of int=4  
Size of A=12  
Size of B=24

int h	A	
int i		
int j		
int m	B	
int n		
int p		



## 6.7 禁止派生

- 禁止（防止）一个类产生派生类

```
class A final {  
    .....  
};
```

A 不能作为基类，不能由 A 产生派生类

```
class B : A { ..... }
```

编译器报错：不能将 final 类类型用作基类

final 的其他用途：virtual ... fun(... ) final ;

禁止虚函数 fun 被重写。



# 总结

- 单继承类
- 派生控制、派生类对象可访问的成员
- 成员访问、同名成员的优先级
- 派生类的构造与析构顺序
- 父类对象和子类对象的指针
- 派生类对象的数据存储模式



華中科技大學

