
中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II
C++类的封装
主讲：吴锋

目录
CONTENTS

类的定义

- 类是一种将数据连同函数“封装”在一起的“抽象数据类型”，该类型的变量（实例）称为对象。

```
// intstack.h: A Stack class for ints 用于定义一个类
class StackOfInt {
public:
        StackOfInt(int); //构造函数，生成对象时自动执行
        void push(int);
        int pop();
        int top() const;
        int size() const;
        ~StackOfInt();  //析构函数，撤销对象时自动执行
private:
        int *data;
        int length;
        int ptr;
};
```

类的特点

- 函数成了类的内部成员
  - 实现了封装，不能任意地增、删类内的函数和数据成员

```
class StackOfInt {
      ….;    // 数据成员
      … ;    // 成员函数
};
```

- 用作用域解析运算符 “::” 指示成员的从属关系
  - 这样函数有明确的归属

```
void StackOfInt::push(int x){
    …;
}
```

类的特点（续）

- 调用成员函数时，用成员选择符“.”指示从属关系
  - 由被动函数调用转变成主动向对象发生消息，当对象收到一个消息后，执行相应的操作（函数）

```
StackOfInt A;
A.push(80); …
```

- 对象创建时，只是分配数据成员的存储空间；成员函数的目标代码仍然只有一个拷贝

```
StackOfInt A, B, C ;    //有三个数据区
A.push(80); B.push(10); C.push(4);   // 共享同一个函数体
```

类的头文件

- 通常把类的定义放在头文件(.h，注：C++标准库的头文件没有后缀)中，而把类的成员函数的实现放在 .cpp 文件中（包含类定义的头文件）

```
// intstack.h：提供给用户，只将类的定义暴露给用户
class StackOfInt { ….; };
```

```
// intstack.cpp：可以编译成库，隐藏类函数的具体实现
#include “intstack.h”
void StackOfInt::push(int x){ …; }
```

- 为避免因多次包含头文件造成重复定义，可以：

```
#pragma once
// 一般的编译器都支持
// 否则，可用右边的宏

class StackOfInt { … };
```

```
#ifndef SIMPLE_H
#define SIMPLE_H

class StackOfInt { … };

#endif
```

类的访问控制

- 访问控制：将类的功能描述部分作为共有部分接口提供给用户，将数据的内部表示、功能的内部实现作为私有部分隐藏起来。
  - public：公有成员，其后声明的所有成员可以被所有的人访问。
  - private：私有成员，除了该类型的创建者和类的内部成员函数之外，任何人不能访问。
  - protected：保护成员，与private基本相同，区别是继承的类可以访问protected成员，但不可以访问private成员。
- 若试图访问私有成员，会产生一个编译时错误。

类的访问控制

```
class B {
public:
        int i;
        void func1();
private:
         char j;
         void func2();
};
void B::func1() {
         B tempb;
         tempb.j = ‘a’; // OK
         tempb.func2(); // OK
         i = 0; j = '0'; func2();
};
```

```
int main() {
        B b;
        b.i = 1;     // OK, public
        //!  b.j = '1';  // illegal
        //! B.func2(); // illegal
        b.func1(); // OK, public
        return 0;
}

class  X {
      int i;     // 私有成员
      int y;      // 私有成员
      void f(); // 私有成员
} // 成员默认为 private
```

◎ 类的访问控制

- 当类的 public 成员函数返回对该类 private 成员变量的引用或指针时，外界可以对私有变量进行修改，这将形成访问控制的一个“漏洞”。
- 从封装性的角度，应尽量避免这样的行为；如果在所难免，应返回const引用或指针，使得私有变量变得“只读”不可修改。

```
class X {
private:
        int i; char *str;
public:
        const int & geti() { return i; } // i 只读
        char const* getstr() { return str; } // str 只读
};
```

◎ 类的友元

- 友元：显示地声明哪些其它的函数或类可以访问一个类的私有成员，使用关键字 friend。
- 友元不是类的成员，只是一个声明；可以是外部全局函数、类或类的成员函数。
- 友元是一个“特权”函数，当类的实现发生变化后，友元也要跟着变动。
- 友员不是面向对象的特征，破坏了封装性，在面向对象编程中尽量不要使用。

类的友元

```
// 例：友元
class X {
      friend void Y::f(X*);    // 成员函数友元
      friend void g(X*, int); // 全局函数友元
      friend class Z;          // 类友元
      friend void h();         // 全局函数友元
private:
      int i;
public:
      void func();
};

void X::func() {
     i = 0;     // 类X的初始化函数
}
```

类的友元

```
// 例：友元
void Y::f(X* x) {
      x->i = 47; // 访问X的私有成员i
}

class Z {
public:
    void g(X* x) { x->I = 38; }; // 访问X的私有成员i
};

void g(X* x, int i) {
      x->i = i; // 访问X的私有成员i
}

void h() {
  X x; x.i = 100; //直接访问私有成员
}
```

嵌套类

- 一个类可以在另一个类中定义，这样的类称为嵌套类。嵌套类可以定义在外围类的公有、私有或保护去中。通过“外围类::嵌套类”的方式使用。

```
class X {
public:
        class Y { … }; // 直接定义在外围类内
        Y y;           // 外围类内可以直接使用
private:
        class Z;       // 或者在外围类内声明，外围类外定义
        Z z;           // 外围类内可以直接使用
};

class X::Z { … }       // 在外围类外定义嵌套类

X::Y y;                // 公有区的嵌套类，外界可见
X::Z z;                //  错误！私有区的嵌套类，外界不可见
```

嵌套类

- 外围类对嵌套类只起到“类域”的作用。外围类内与嵌套类同名的类（如有）被隐藏。
- 外围类不能访问嵌套类的非公有成员，除非外围类被嵌套类声明为友元。
- 嵌套类访问外围类的非静态（公有）成员，需要传入外围类的引用或指针。

```
class Y { … }

class X {
    private: // 嵌套类通常定于在私有区，为外围类专用
        class Y { friend class X; … }; // 声明外围类为友元
        Y y; // y 是嵌套类Y定义的类型，外界的Y被隐藏
};
```

局部类

- 定义在函数体内的类叫做局部类，只在定义它的局部域内可见。
- 局部类只能访问外围局部域中定义的类型名、静态变量和枚举变量，不能访问函数的动态变量。

```
int a = 0;
void func() {
    static int b = 1;

    class X {
        public:
            int g() { return a + b; }
    };

    X x;  x.g();
}
```

句柄类

- 当用户使用一个类时，需要给他提供定义类的头文件(.h)，并包含在用户的程序中，但是：
  - 用户可以清楚的看到类的定义，特别是类的私有成员，从而知道类的实现，甚至特意（为了使用方便）在类的定义中添加友元，破坏封装性和隐藏性。
  - 如果改变了类的实现，即使没有改变类的接口，因为包含的头文件改变了，也可能需要重新编译所有使用了该类的客户程序。
- 解决办法：利用“句柄类”。
  - 所谓句柄类，是一个特殊的接口类，该类中含有一个特殊的成员，即一个指向被隐藏的类的指针。

句柄类

```
// 接口：公开的部分(handle.h)
#pragma once
class Handle {
private:
        class StackOfInt;          // 类StackOfInt的声明；
        StackOfInt* stack;         //  stack 指针指向具体的实现
public:
        void initialize(int);  // 初始化
        void cleanup();     // 清理
        void push(int);     // 接口
        int pop();              // 接口
        int top();                  // 接口
        int size();                 // 接口
};
```

句柄类


```
// 实现：需要隐藏的部分(handle.cpp)
// 通常单独编译成库，再提供给用户

#include "handle.h"
#include "intstack.h"

void Handle::initialize(int size) {
    stack = new StackOfInt(size);
}

void Handle::cleanup() { delete stack; }

void Handle::push(int x) { stack->push(x); }
int Handle::pop() { return stack->pop(); }
int Handle::top() { return stack->top(); }
int Handle::size() { return stack->size(); }
```

句柄类

- 句柄类的主要优点：
  - 由于实现部分并没有暴露在.h文件中，保密性好；
  - 因为实现的细节均隐藏在句柄类的背后，因此当实现发生变化时，句柄类并没有改变，也就不必重新编译客户程序。
  - 只需重新编译隐藏的实现这一小部分代码，然后连接 (link) 就行了。减少了大量的重复编译。

◎ 小结：类的访问控制

- 访问控制的作用是对类进行模块化封装，让编译器来“监督”对面向对象封装思想的执行。
  - C++的类中，除了对外提供服务的接口函数是 public 外，其它部分都应该是 private 或 protected 的，特别是类的成员变量。
  - 通常将类的定义放在 .h 的头文件中，将成员函数的实现单独放在 .cpp 的文件中，这样既利于隐藏，也避免因实现的改变而导致的重编译。
  - 友元是C++类封装的“破坏者”，多数情况不需要用到，如果实用也应该慎用（影响最小）！
  - 嵌套类和局部类都是对类作用域的一种隐藏方法，类也可以进一步隐藏在名字空间中，避免类的重名问题。

安全性要求

- 正确地初始化和清除对象是保证程序安全性的关键，但却经常被忽略导致运行时错误。

```
void main() {
    // 使用的数组空间是动态分配的，需要初始化
    StackOfInt stack;
    stack.initialize(sizeof(int));  // 需要手动初始化
    …
    for(int i = 0; i < 100; i++){
            stack.push(i);
    }
    …
    stack.cleanup();                    // 需要手动清除
  // new分配的空间要delete清除，否则内存泄露
}
```

构造函数：确保初始化

- 类的特殊成员函数，编译器在创建对象时自动调用该函数，通常做一些初始化动作。
  - 保证同一个类的对象具有一致性。
  - 编译器不会“忘记”初始化。
- 构造函数跟类同名，可以带参数，没有返回值。
  - 跟别的成员函数没有名字冲突。
  - 编译器总能知道调用哪一个函数。

构造函数（例）

```
class StackOfInt {
public:
        StackOfInt(int); // 构造函数，跟类同名，无返回值
        …
private:
        int *data;
        int length;
        int ptr;
};
StackOfInt::StackOfInt(int size) {
    data = new int[length = size];
    ptr = 0;
}
void main() {
     StackOfInt stack(100); // 创建对象时自动调用构造函数
     StackOfInt *sptr = new StackOfInt(100);
     …
}
```

◎ 构造函数的说明

- 构造函数应声明为 public（但不是必须），否则创建对象时会发生错误。

```
class StackOfInt {
private:
    StackOfInt (int size) {…};
    …
};

void main(){
     StackOfInt stack(10);    // 错误，不能访问私有函数
}
```

◎ 构造函数的说明

- 构造函数可以设为内联函数，也可以设置默认参数值，还可以进行函数重载。

```
class StackOfInt {
public:
      StackOfInt () { length = 100; … }
      StackOfInt (int size) { length = size; … }
      …
};
void main(){
      StackOfInt  charStack;          // 调用 StackOfInt ()
      StackOfInt  intStack(2);        // 调用  StackOfInt (int sz)
      StackOfInt  errStack(10, 2);    // 错误，参数不匹配!
}
```

默认构造函数

- 如果类没有定义任何构造函数，可以创建对象但可能未被初始化，因此应定义默认构造函数（无参）。

```
class Y{ …; }     // 类定义中没有提供构造函数

Y objy;           // 可以创建对象，但可能未被正确初始化
Y objy();         // 常见错误，这事实上是声明了一个函数
```

- 如果定义了构造函数，但没有无参构造函数，则创建对象时，若不带参数，将会出错。

```
class Y{
   Y(int sz){…}  // 提供了一个带参数的构造函数
};

Y objy;           // 错误，构造函数的参数类型不匹配
Y arry[10];       // 错误，需要有不带参数的构造函数
```

非公有构造函数

- 可以将构造函数定义在非公有区，并声明友元，从而达到限制对象创建的目的。

```
class Y {
    friend class X;  // 限制只有类X才能创建对象
    friend Y getY(); // 限制只有函数getY()才能创建对象
private:
    Y(); // 定义私有的默认构造函数
};

Y& getY() {
    static Y y; // 通过静态变量，限制了Y只能有一个实例
    return y;
}
```

拷贝构造函数

- 用于一个类对象去初始化该类的另一个对象时，相当于生成该对象的一个拷贝，编译器自动调用的是拷贝构造函数。

```
class X {
public:
    X();                // 默认构造函数
    X(const X& x);      // 拷贝构造函数
    …
};

X x1;                   // 调用默认构造函数
X x2(x1);               // 调用拷贝构造函数
X x3 = x1;              // 调用拷贝构造函数
X x4;                   // 调用默认构造函数
x4 = x1;                // 调用赋值运算符 X::operator=(const X& x)
```

默认拷贝构造函数

- 如果没有自定义拷贝构造函数，编译器则默认按位拷贝（浅拷贝）的方式构建对象，可能产生错误。

```
class StackOfInt {
public:
        …                    // StackOfInt 中没有自定义的拷贝构造函数
private:
        int *data;
        int length;
        int ptr;
};

StackOfInt s1;
StackOfInt s2(s1);    // 可以执行，调用默认拷贝构造函数
StackOfInt s3 = s1;   // 可以执行，调用默认拷贝构造函数
// 位拷贝存在的问题：length 和 ptr 可以被正确拷贝
// 但data的位拷贝导致s1.data, s2.data, s3.data 指向同一空间
```

◎ 默认拷贝构造函数

- 对于位拷贝（浅拷贝）不能满足要求的情况下，应该自定义拷贝构造函数，进行深拷贝。

```
class StackOfInt {
public:
        StackOfInt(const StackOfInt& s) {
                length = s.length; ptr = s.ptr;
                // 对类成员data进行深拷贝
                data = new int[s.length];
                memcpy(data, s.data, s.length);
        }
        …
private:
        int *data;
        int length;
        int ptr;
};
```

默认拷贝构造函数

- 也可以通过声明一个私有的空拷贝构造函数，禁止类对象的拷贝，防止误操作。

```
class X {
private:
            X(const X&) {};
};

X x1;
X x2(x1);     // 错误，不能调用私有成员函数
X x3 = x1;    // 错误，不能调用私有成员函数
```

成员初始化表

- 成员初始化表：位于构造函数后，以冒号开始，逗号隔开的类成员名及其初始值的列表。

类的构造函数名(类构造函数参数列表+成员构造函数参数列表)
：成员对象1(参数列表)，成员对象2(参数列表) { …; }

```
Employee::Employee( int id,int bmonth, int bday, int byear,int hmonth,
int hday, int hyear ) : idNum( id ), // 内置类型也可以使用初始化表
    birthDate( bmonth, bday, byear ),
    hireDate( hmonth, hday, hyear ) { … }
```

成员初始化表

- 初始化表只能出现在构造函数的定义体中，不能出现在构造函数的声明中。
- 若不提供成员初始化值，则编译器隐式调用成员对象的默认构造函数（无参）。
- const 对象成员和引用对象成员必须在成员初始化表中初始化。

```
Employee::Employee(int id, Date birthdate, Date hiredate) {
        // 先调用默认构造函数生成对象，此处再调用赋值运算符函数
        idNum = id; birthDate =  birthdate; hireDate = hiredate;
}

Employee::Employee(int id, Date birthdata, Date hiredate)
        : idNum(id), birthDate(birthdate), hireDate(hiredate) {
        // 直接调用拷贝构造函数，生成对象
}
```

◎ 析构函数：确保清除

- 析构函数：类的特殊成员函数，在撤销对象时自动调用该函数。通常做一些撤销对象前的回收工作。
- 析构函数不带参数，没有返回值；不能够重载。固定形式是类名前面加上一个~
- 析构函数必须是 public 函数。

析构函数（例）

```
class StackOfInt {
public:
        …
        ~StackOfInt(); // 析构函数，无参数，无返回值
private:
        int *data;
        …
};
StackOfInt::~StackOfInt() {
    delete [] data;
}
void main() {
     StackOfInt stack(100);
     StackOfInt *sptr = new StackOfInt(100);
     …
     delete sptr; // 清除对象 sptr 时自动调用析构函数
     // 退出函数，清除对象 stack 时自动调用析构函数
}
```

显式调用析构函数

- 如果只想执行析构函数中的执行的操作，而不撤销对象，则可以显式调用析构函数。

```
//执行系统的析构函数，不释放对象的空间。

StackOfInt stack;
stack.~StackOfInt();

StackOfInt *sptr = new StackOfInt(100);
sptr->~StackOfInt();
```

构造次序

- 构造函数的执行次序：
  - 对象成员的构造函数先初始化，然后才是包含它的类的构造函数。
  - 有多个对象成员时，按照在类的定义中声明的次序初始化对象成员。

```
class Employee {
private:
    int idNum;
    Date birthDate;
    Date hireDate;
    … // 以下省略
};
```

- 先执行成员初始化：

  idNum
  birthDate.Date(…)
  hireDate.Date(…)

- 再执行类的初始化：
  e.Employee(…)

析构次序

- 析构函数的执行次序：
  - 先执行对象本身的析构函数，然后才执行对象成员的析构函数。
  - 有多个对象成员时，按照在类的定义中声明的次序相反的次序析构对象成员。

```
class Employee {
private:
    int idNum;
    Date birthDate;
    Date hireDate;
    … // 以下省略
};
```

- 先执行类的析构：
  e.~Employee(…)
- 再执行成员的析构：
  hireDate.~Date(…)
  birthDate.~Date(…)
  idNum

◎ 小结：类的构造/析构

- 构造和析构函数是保证封装后的对象能够正确的进行初始化和清除，整个过程由编译器自动的执行。
  - 当对象被创建时，自动调用构造函数；当对象被撤销时，自动调用析构函数。
  - 应该定义默认构造函数，否则按默认内存分配方式，类可能没有被正确初始化。
  - 应该定义拷贝构造函数，否则按默认的位拷贝，类可能没有被正确的拷贝。
  - 在构造函数中，尽量用成员初始化表来初始化成员变量，顺序可控且代价更低。