



# 第六章 类和使用对象

主讲教师：同济大学电子与信息工程学院 陈宇飞



# 目录

- 类和对象的基本使用（构造函数补充）
- 复制构造函数的调用时机
- 共用数据的保护
- 静态成员
- 类模板



# 6.1 类和对象的基本使用

## 6.1.1 构造函数初始值列表

- 初始化：直接初始化数据成员； **赋值**：先初始化再赋值
- 构造函数的初始值有时必不可少

```
class ConstRef {  
public:  
    ConstRef(int ii);  
private:  
    int i;  
    const int ci;  
    int &ri;  
};
```

```
// 错误：ci和ri必须被初始化  
ConstRef::ConstRef(int ii)  
{ // 赋值  
    i = ii;    // 正确  
    ci = ii;   // 错误：不能给const赋值  
    ri = i;    // 错误：ri没被初始化  
}
```



# 6.1 类和对象的基本使用

## 6.1.1 构造函数初始值列表

- **初始化**: 直接初始化数据成员; 赋值: 先初始化再赋值
- 构造函数的初始值有时必不可少

```
class ConstRef {  
public:  
    ConstRef(int ii);  
private:  
    int i;  
    const int ci;  
    int &ri;  
};
```

```
// 正确: 显式的初始化引用和const成员  
ConstRef::ConstRef(int ii): i(ii),  
ci(i), ri(ii) { }
```

结论: 如果成员是const、引用, 或者属于某种未提供默认构造函数的类类型, 必须通过构造函数初始值列表为这些成员提供初值。



# 6.1 类和对象的基本使用

## 6.1.1 构造函数初始值列表

- 成员初始化的顺序：与类定义中出现的顺序一致，跟初始化列表中的顺序无关
  - 建议1：构造函数初始值的顺序与成员声明的顺序保持一致
  - 建议2：尽量避免使用某些成员初始化其它成员

```
class X {  
    int i;  
    int j;  
public:  
    //实际上是i先被初始化!  
    X(int val): j(val), i(j) {};  
};
```

```
class X {  
    int i;  
    int j;  
public:  
    //正确  
    X(int val): i(val), j(val) {};  
};
```



# 6.1 类和对象的基本使用

## 6.1.1 构造函数初始值列表

- 默认实参和构造函数:

➤ 若构造函数为所有参数都提供了默认实参，则相当于定义了默认构造函数

```
class Sales_data{  
    Sales_data (string s = ""):bookNo(s) { }  
};
```

上例：当没有给定实参或者给定了一个string实参时，类创建的对象相同。

（因为不提供实参也能调用上述构造函数，所以该构造函数实际就为类提供了默认构造函数）



# 6.1 类和对象的基本使用

## 6.1.1 构造函数初始值列表

- 默认实参和构造函数：
  - 不能为构造函数的全部形参都提供默认实参

```
//接受string的构造函数
class Sales_data{
    Sales_data (string s = ""):bookNo(s) { }
};
//接受istream&参数的构造函数
class Sales_data{
    Sales_data (istream &is = cin) { is >> *this; }
};
```

错误

上例：不提供任何实参的创建类对象时，产生二义性



# 6.1 类和对象的基本使用

## 6.1.2 委托构造函数

- 委托构造函数使用它所属类的其他构造函数执行自己的初始化过程，或者说它把自己的一些(或全部)职责委托给了其他的构造函数
- 委托构造函数也有一个成员初始值的列表和一个函数体。在委托构造函数内，成员初始值列表只有唯一的一个入口，就是类名本身。和其他成员初始值一样，类名后面紧跟圆括号括起来的参数列表，参数列表必须与类中另外一个构造函数匹配





```
class Sales_data {  
public:
```

```
    //非委托构造函数接收三个实参，使用这些实参初始化数据成员，然后结束工作  
    Sales_data(string s,unsigned cnt,double price):  
        bookNo(s),units_sold(cnt),revenue(cnt*price) {}
```

```
    //其余构造函数全都委托给另一个构造函数
```

```
    Sales_data():Sales_data(" ", 0, 0) {}
```

定义默认构造函数令其使用三参数的构造函数完成初始化过程

```
    Sales_data(string s):Sales_data(s, 0, 0) {}
```

定义接收一个string的构造函数，同样委托给了三参数版本

```
    Sales_data(istream &is):Sales_data() { read(is, *this); }
```

定义接收istream &的构造函数，它委托给了默认构造函数，默认构造函数接着委托给三参数的构造函数。当接受委托的构造函数执行完后，接着执行istream &构造函数体的内容，即调用read函数读取给定的istream

```
private:
```

```
    string bookNo; unsigned units_sold; double revenue;
```

```
};
```



# 6.1 类和对象的基本使用

## 6.1.3 默认构造函数的作用

### ➤默认初始化发生的情况：

- 在块作用域内不使用任何初始值定义一个非静态变量或数组
- 类本身含有类类型的成员并且使用合成的默认构造函数
- 当类类型的成员没有在构造函数初始值列表中显式的初始化

### ➤值初始化发生的情况：

- 数组初始化的过程中如果提供的初始值少于数组的大小
- 不使用初始值定义一个局部静态变量
- 通过书写形如 `T()` 的表达式显式地请求值初始化时，`T`是类型名

➡ 类必须包含一个默认构造函数以便在上述情况下使用



//例：类的数据成员缺少默认构造函数

```
class NoDefault {
```

```
public:
```

```
    NoDefault(const string&) {...};
```

```
};
```

注意：使用默认构造函数：NoDefault() {}；才可以正常编译通过！！

```
struct A {
```

```
    NoDefault my_mem; //默认public
```

```
};
```

A a; //错误，不能为A合成构造函数

```
struct B {
```

```
    B() {} //错误，b_member没有初始值
```

```
    NoDefault b_member;
```

```
};
```

**合成构造函数：**如果用户定义的类中没有显式的定义任何构造函数，编译器才会自动为该类型生成默认构造函数，称为合成的构造函数



```
#include <iostream>
#include <string>
using namespace std;
class NoDefault {
public:
    NoDefault(const string&) {...};
    NoDefault() {};
};
NoDefault obj1()
{
    cout << "helloworld" << endl;
    NoDefault obj;
    return obj;
}
int main()
{
    NoDefault obj2 = obj1();
    return 0;
}
```

//注意区分:

NoDefault obj1() //定义了一个obj1函数  
NoDefault obj2 //定义了一个obj2对象



# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

➤转换构造函数 (converting constructor):

- 当一个构造函数只有一个参数，而且该参数又不是本类的const引用时，这种构造函数称为转换构造函数
- 转换构造函数的作用是将一个其他类型的数据转换成一个类的对象

注意:

转换构造函数只能有一个参数。如果有多个参数，就不是转换构造函数



```
class Sales_data {
private:
    string book_no;
    unsigned units_sold = 1;
    double revenue = 1.0;

public:
    Sales_data() = default; //不接受任何实参, 默认构造函数
    Sales_data(const string& s) : book_no(s) {} //类型转换构造函数
    Sales_data(const string& s, unsigned n, double p) :
        book_no(s), units_sold(n), revenue(p* n) {}
    Sales_data(istream&) {};

    Sales_data& combine(const Sales_data&);
    //其他成员函数...

};
```

```
string null_book = "9-999-999-9";
//构造一个临时的Sales_data对象item
item.combine(null_book);
```



# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

➤ 只允许一步类类型转换:

```
Sales_data(const string& s) : book_no(s) {}  
    //类型转换构造函数  
Sales_data& combine(const Sales_data&);
```

```
item.combine("9-999-999-9");
```

//**错误**: 需要用户定义的一种转换: "9-999-999-9"到string到Sales\_data

```
item.combine(string("9-999-999-9"));
```

//正确: 显式地转换成string, 隐式地转换成Sales\_data

```
item.combie(Sales_data("9-999-999-9"));
```

//正确: 隐式地转换成string, 显式地转换成Sales\_data



# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

➤ 类类型转换不是总有效:

`item.combine(cin);` // 隐式地将 `cin` 转换成 `Sales_data`, 这个转换执行接受一个 `istream` 的 `Sales_data` 构造函数, 该构造函数通过读取标准输入创建了一个临时的 `Sales_data` 对象, 随后将得到的对象传递给 `combine`。该对象是一个临时量, 一旦 `combine` 完成就不能再访问它了

```
...  
Sales_data(istream&) {};  
Sales_data& combine(const Sales_data&);
```





# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

➤抑制构造函数定义的隐式类型转换:

- 将构造函数声明为 **explicit** 可以阻止构造函数的隐式类型转换

```
explicit Sales_data(const std::string &s):bookNo(s) { }
```

```
explicit Sales_data(std::istream&);
```

此时:

```
item.combine(null_book);    //错误
```

```
item.combine(cin);          //错误
```



# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

```
explicit Sales_data(const std::string &s):bookNo(s) { }  
explicit Sales_data(std::istream&);
```

➤抑制构造函数定义的隐式类型转换:

- 关键字explicit **只对一个实参的构造函数**有效, 需要多个实参的构造函数不能用于隐式转换, 所以无须将这些构造函数指定为explicit
- **只能在类内声明**构造函数时使用explicit, 类外定义时不应重复
- 当使用explicit声明构造函数时, 将**只能以直接初始化**的形式使用

```
Sales_data item1(null_book); //正确, 直接初始化
```

```
Sales_data item2 = null_book;
```

//错误, 不能将explicit构造函数用于拷贝形式的初始化过程



# 6.1 类和对象的基本使用

## 6.1.4 隐式的类类型转换

➤为转换显式地使用构造函数:

```
explicit Sales_data(const std::string &s):bookNo(s) { }  
explicit Sales_data(std::istream&);  
Sales_data& combine(const Sales_data&);
```

```
item.combine(Sales_data(null_book));
```

//直接使用 Sales\_data 的构造函数, 该调用通过接受 string 的构造函数创建一个临时的 Sales\_data 对象

```
item.combine(static_cast<Sales_data>(cin));
```

//使用 \_cast 执行了显式的转换: 使用 istream 构造函数创建了一个临时static的 Sales\_data 对象



# 目录

- 类和对象的基本使用（构造函数补充）
- 复制构造函数的调用时机
- 共用数据的保护
- 静态成员
- 类模板



## 6.2 复制构造函数的调用时机

- 对象复制的基本概念（复习回顾）

- 含义：建立一个新对象，其值与某个已有对象完全相同

- 使用：

类 对象名(已有对象名)

类 对象名=已有对象名

Time t1(14, 15, 23), t2(t1), t3=t1;

- 与对象赋值的区别：

Time t1(14, 15, 23), t2, t3=t1; //复制：定义语句中

t2 = t1; //赋值：执行语句中

- 对象复制的实现：建立新对象时自动调用复制构造函数（也称为拷贝构造函数）



## 6.2 复制构造函数的调用时机

- 复制构造函数:

类名(const 类名 &引用名)

- 用一个对象的值去初始化另一个对象
- 若不定义复制构造函数，则系统自动定义一个，参数为const型引用，函数体为对应成员内存拷贝
- 若定义了复制构造函数，则系统缺省定义的定义消失
- 允许体内实现或体外实现
- 复制构造函数和普通构造函数（可能多个）的地位平等，调用其中一个后就不再调用其它构造函数



## 6.2 复制构造函数的调用时机

- 复制构造函数和普通构造函数(可能多个)的地位平等, 调用其中一个后就不再调用其它构造函数:

```
class Time {  
    ...  
public:  
    Time(int h=0);  
    Time(int h, int m, int s=0);  
    Time(const Time &t);  
};
```

```
int main()  
{  
    Time t1;  
    Time t2(10);  
    Time t3(1, 2, 3);  
    Time t4(4, 5);  
    Time t5(t2);  
    Time t6 = t4;  
}
```



```
#include <iostream>
using namespace std;
int tcount = 0; //全局变量, 计数器
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time() {cout<<"tcount="<<--tcount<<endl;}
    void display()
    {cout<<hour<<":"<<minute<<":"<<sec<<endl;}
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{
    hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    ++tcount; //计数器+1
    cout << "复制构造" << endl;
}

int main()
{
    //用对象初始化新对象
    Time t1(14, 15, 23), t2(t1);
    t2.display();
}
```

普通构造  
复制构造  
13:14:22  
tcount=1  
tcount=0





```
#include <iostream>
using namespace std;
int tcount = 0; //全局变量, 计数器
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time() {cout<<"tcount="<<--tcount<<endl;}
    void display()
    {cout<<hour<<":"<<minute<<":"<<sec<<endl;}
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{
    hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    ++tcount; //计数器+1
    cout << "复制构造" << endl;
}

void fun(Time t)
{
    //函数形参为对象
    t.display();
}

int main()
{
    Time t1(14, 15, 23);
    fun(t1);
}
```

普通构造  
复制构造  
13:14:22  
tcount=1  
tcount=0



```
#include <iostream>
using namespace std;
int tcount = 0; //全局变量，计数器
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time() {cout<<"tcount="<<--tcount<<endl;}
    void display()
    {cout<<hour<<":"<<minute<<":"<<sec<<endl;}
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    ++tcount; //计数器+1
    cout << "普通构造" << endl;
}
```

```
Time::Time(const Time &t)
{
    hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    ++tcount; //计数器+1
    cout << "复制构造" << endl;
}
```

```
Time fun() //函数返回值为对象
{
    Time t1(14, 15, 23);
    return t1; //调用复制构造函数产生
              一份拷贝t2，再释放t1
}
```

```
int main()
{
    Time t2 = fun();
    t2.display();
}
```

普通构造  
复制构造  
tcount=1  
13:14:22  
tcount=0



➤ 变量定义时赋初值与使用赋值语句赋初值的区别:

```
Time fun()
{
    Time t1(14, 15, 23);
    return t1; //调用复制构造函数产生一份拷贝t2, 再释放t1
}
int main()
{
    Time t2 = fun(); //定义时赋初值
    t2.display();
}
```

普通构造  
复制构造  
tcount=1  
13:14:22  
tcount=0

```
Time fun()
{
    Time t1(14, 15, 23);
    return t1; //调用复制构造函数产生一份临时拷贝, 赋值给t2, 再释放t1
}
int main()
{
    Time t2;
    t2 = fun(); //赋值语句赋初值
    t2.display();
}
```

普通构造  
普通构造  
复制构造  
tcount=2  
tcount=1  
13:14:22  
tcount=0



# 目录

- 类和对象的基本使用（构造函数补充）
- 复制构造函数的调用时机
- 共用数据的保护
- 静态成员
- 类模板



## 6.3 共用数据的保护

### 6.3.1 基本概念

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到**既能共享，又不会因误操作而改变**，引入共用数据保护

- 常对象
- 常对象成员



## 6.3 共用数据的保护

### 6.3.2 常对象与常对象成员

- 常对象:

`const 类名 对象名(初始化实参表)`    **或**    `类名 const 对象名(初始化实参表)`  
`const Time t1(15);`                      `Time const t2(16, 30, 0);`

- 在整个程序的执行过程中值不可再变化
- 必须在定义时进行初始化
- 不能调用普通成员函数(即使不改变数据成员的值)



//常对象

```
#define <iostream>
```

```
using namespace std;
```

```
class Time {
```

```
public:
```

```
int hour, minute, sec;
```

```
Time(int h=0, int m=0, int s=0)
```

```
{
```

```
hour = h; minute = m; sec = s;
```

```
}
```

```
void display()
```

```
{
```

```
cout << hour << minute << sec;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
const Time t1(15);
```

```
Time const t2(16, 30, 0);
```

```
t1.minute = 12; //编译报错
```

```
t2.sec = 27; //编译报错
```

```
t1.display(); //编译报错
```

```
//display函数虽然不改变值，仍报错
```

```
}
```

t1始终是15:00:00

t2始终是16:30:00



## 6.3 共用数据的保护

### 6.3.2 常对象与常对象成员

- 常对象成员：

常对象中的所有数据成员在程序执行过程中值均不可变，如果只需要限制部分成员的值在执行过程中不可变，则需要引入常对象成员的概念

- 常数据成员：该数据成员的值在执行中不可变
- 常成员函数：该函数只能引用成员的值，不能修改





//常数据成员:

```
class 类名 {  
    const 数据类型 数据成员名  
    或 数据类型 const 数据成员名  
};  
  
class Time {  
    private:  
        const int hour;  
        int const minute;  
        int sec;  
};
```

//常成员函数:

```
class 类名 {  
    返回类型 成员函数名(形参表) const;  
};  
  
class Time {  
    public:  
        void display() const;  
};
```

- 使用：常数据成员要在构造函数中初始化，使用中值不可变，在构造函数中初始化时，必须用参数初始化表形式，而不能用赋值形式



```
#define <iostream>
using namespace std;
class Time {
public:
```

```
    const int hour;
    int minute, sec;
```

```
    Time(int h=0, int m=0, int s=0) : hour(h)
    {
        minute = m;
        sec = s;
    }
```

```
};
```

```
int main()
```

```
{    Time t1;
```

```
    t1.hour = 10; //错误
```

```
}
```

//错误

```
Time(int h=0, int m=0, int s=0)
{
    hour = h;
    minute = m;
    sec = s;
}
```

//正确

- 使用：常成员函数只能引用类的数据成员 (无论是否常数据成员) 的值，而不能修改数据成员的值



```
#define <iostream>
using namespace std;
class Time {
public:
```

```
    int hour, minute, sec;
```

```
    void set(int h=0, int m=0, int s=0) const
    {
        hour = h;
        minute = m;
        sec = s;
    }
};
```

```
int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

```
void set(int h=0, int m=0, int s=0)
{
    hour = h;
    minute = m;
    sec = s;
}
```

//正确

//错误



- 使用：常成员函数写成下面形式，编译不报错但不起作用

const 返回类型 成员函数名(形参表) **或** 返回类型 const 成员函数名(形参表)

```
#define <iostream>
```

```
using namespace std;
```

```
class Time {
```

```
    public:
```

```
        int hour, minute, sec;
```

```
        const void set(int h, int m, int s)
```

```
        {    hour = h;
```

```
            minute = m;
```

```
            sec = s;
```

```
        }
```

```
void const set(int h, int m, int s)
```

```
{    hour = h;
```

```
    minute = m;
```

```
    sec = s;
```

```
}
```

```
};
```

```
int main()
```

```
{    Time t1;
```

```
    t1.set(14, 15, 23); //赋值正确，说明set不是常成员函数
```

```
}
```



- 使用：常成员函数可以调用本类的另一个常成员函数，但不能调用本类的非常成员函数（即使该非常成员函数不修改数据成员的值）

```
#define <iostream>
using namespace std;
class Time {
public:
    int hour, minute, sec;
    void display()
    { cout << hour << endl;
    }
    void fun() const
    { display(); //错误
    }
};
int main()
{
    Time t1;
    t1.fun();
}
```

```
void display() const
{ cout << hour << endl;
}
void fun() const
{ display(); //正确
}
```



- 使用：若希望常成员函数能强制修改数据成员，则要将数据成员定义为mutable

```
#include <iostream>
using namespace std;
class Time {
public:
    int mutable hour, minute, sec; // mutable int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour = h;
        minute = m;
        sec = s;
    }
};
int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```



- 使用：若定义对象为常对象，则只能调用其中的常成员函数(不能修改数据成员的值)，而不能调用其中的普通成员函数(即使该成员不修改数据成员的值)

```
class Time {  
    public:  
        int hour, minute, sec;  
        Time(int h=0, int m=0, int s=0)  
        { hour = h; minute = m; sec = s;}  
        void display()  
        { cout << hour << minute << sec << endl;}  
};  
int main()  
{ const Time t1(13, 14, 23);  
  t1.minute = 12; //编译报错  
  t1.display(); //编译报错  
}
```

```
...  
void display() const  
...  
int main()  
{ const Time t1(13, 14, 23);  
  t1.display(); //正确  
}
```



- 使用：若定义对象为常对象，则只能调用其中的常成员函数(不能修改数据成员的值), 而不能调用其中的普通成员函数(即使该成员不修改数据成员的值)

```
class Time {  
    public:  
        int hour, minute, sec;  
        Time(int h=0, int m=0, int s=0)  
        { hour = h; minute = m; sec = s;  
        }  
        void display() const  
        { cout << hour << minute << sec << endl;  
          sec++; // 错误  
        }  
};  
  
int main()  
{  
    const Time t1(13, 14, 23);  
    t1.display();  
}
```

```
mutable int sec;  
...  
void display() const  
{ cout<<hour<<minute<<sec<<endl;  
  sec++; //正确  
}
```





- 使用:

- 不能定义构造/析构函数为常成员函数
- 全局函数不能定义const

```
class Time {    //编译报错
public:
    int hour, minute, sec;
    Time(int h=0, int m=0, int s=0) const
    { hour = h; minute = m; sec = s;
    }
    ~Time() const {}
};
int main()
{    Time t1(13, 14, 23);
    ...
}
```

```
void fun() const    //编译报错
{
    ...
    return;
}
int main()
{
    fun();
    ...
}
```



# 6.3 共用数据的保护

## 6.3.3 小结

	普通 数据成员	const 数据成员	mutable 数据成员	普通 成员函数	const 成员函数
普通对象	读写	读	读写	可调用	可调用
const对象	读	读	读写	不可调用	可调用
普通成员函数	读写	读	读写	可调用	可调用
const成员函数	读	读	读写	不能调用	可调用



# 目录

- 类和对象的基本使用（构造函数补充）
- 复制构造函数的调用时机
- 共用数据的保护
- 静态成员
- 类模板



## 6.4 静态成员

### 6.4.1 基本概念

希望在同一个类的多个对象间实现数据共享

- 一个对象修改，另一个对象访问得到修改后的值
- 类似与全局变量的概念，但属于类，仅供该类的不同对象间共享数据



## 6.4 静态成员

### 6.4.2 静态数据成员

- 定义:

```
class 类名 {  
    private/public:  
        static 数据类型 成员名;  
        ...  
};
```

```
class Test{  
    private:  
        static int a; //静态数据成员  
        ...  
};
```



## 6.4 静态成员

### 6.4.2 静态数据成员

- 使用:

- 静态数据成员不属于任何一个对象，不在对象中占用空间，单独在静态数据区分配空间 (初值为0，不随对象的释放而释放)，一个静态数据成员只占有一个空间，所有对象均可共享访问
- 静态数据成员不是面向对象的概念，它破坏了数据的封装性，但方便使用，提高了运行效率



- 静态数据成员必须进行初始化，初始化位置在类定义体后，函数体外进行(此时不受类的作用域限制)    **数据类型 类名::静态数据成员名=初值;**

```
#include <iostream>
using namespace std;
class Test {
private:
    static int a; //静态数据成员
public:
    int GetA() const { return a; }
};
```

**//int Test::a; 如果这样定义不赋予初值则初值为零**  
**int Test::a = 1;**

```
int main()
{
    Test T;
    cout << T.GetA() << endl;
    return 0;
}
```



➤既可以通过类型引用，也可以通过对象名引用

```
#include <iostream>
using namespace std;
class Test {
public:
    static int a; //静态数据成员
    int GetA() const { return a; }
};
int Test::a = 1;
```

```
int main()
{
    Test T;
    T.a++;           //对象名引用
    Test::a++;       //类型引用
    cout << T.GetA() << endl; //3
    return 0;
}
```





## ➤不能通过参数初始化表进行初始化，但可以通过赋值方式初始化

```
#include <iostream>
using namespace std;
class Test {
private:
    static int a; //静态数据成员
public:
    int GetA() const { return a; }
    Test(int x) { a = x; } //正确
};
int Test::a = 1; //不可省，否则编译错
int main()
{
    Test T(5);
    cout << T.GetA() << endl; //5
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Test {
private:
    static int a; //静态数据成员
public:
    int GetA() const { return a; }
    Test(int x): a(x) {} //错误
};
int Test::a = 1;
int main()
{
    Test T(5);
    cout << T.GetA() << endl;
    return 0;
}
```



- 静态数据成员被类的所有对象共享，包括该类的派生类对象，基类对象和派生类对象共享基类的静态数据成员（后续讲继承，此处了解）

```
#include <iostream>
using namespace std;
class Base {
public:
    static int a; //静态数据成员
};
class Derived : public Base {
};
int Base::a; //可以，初值为零
```

```
int main()
{
    Base B;
    Derived D;
    B.a++;
    cout << B.a << endl;
    D.a++;
    cout << D.a << endl;
    return 0;
}
```



- 静态数据成员可以作为成员函数的默认形参，而普通数据成员则不可以
- 静态数据成员的类型可以是所属类的类型，而普通数据成员则不可以。  
普通数据成员只能声明为所属类类型的指针或引用

```
class Test {  
public:  
    static int a;    //静态数据成员  
    int b;  
    void fun_1(int i = a) {};//正确  
    void fun_2(int i = b) {};//报错  
};
```

```
class Test {  
public:  
    static Test a;//正确  
    Test b;//报错  
    Test* pTest;//正确  
    Test& m_Test;//正确  
    static Test* pStaticObject;//正确  
};
```



➤静态数据成员在const函数中可以修改，而普通的数据成员不能修改

```
class Test {  
public:  
    static int a; //静态数据成员  
    int b;  
    Test():b(0) {}  
    void test() const //不能修改当前调用该函数对象的非静态数据成员  
    {  
        a++;  
        b++; //错误  
    }  
};  
int Test::a;
```

const修饰的是当前this指针所指向的对象是const，但是静态数据成员不属于任何类的对象，它可被类的所有对象修改，this指针不修饰静态的数据成员，所以可以修改



## 6.4 静态成员

### 6.4.3 静态成员函数

- 定义:

```
class 类名 {  
    private/public:  
        static 返回类型 函数名(形参表);  
}...
```

- 调用:

类名::成员函数名(实参表);

任意对象名. 成员函数名(实参表);



➤静态成员函数没有this指针，静态成员函数不能使用修饰符(函数后面的const关键字)

```
#include <iostream>
using namespace std;
class Student
{
private:
    int num; int age; float score;
    static float sum;//静态数据成员，累计学生的总分
    static int count;//静态数据成员，累计学生的人数
public:
    Student(int n, int a, float s) :num(n), age(a), score(s) { } //构造函数
    void total();
    static float average();//声明静态成员函数，不能使用const
};
float Student::sum;
int Student::count;
```



//接上页

```
void Student::total() //定义非静态成员函数
```

```
{    sum += score;
    count++;           //有this指针, 可以写this->sum,  this->score,  this->count
}
```

```
float Student::average()
```

//定义静态成员函数, 此处不需要再次写上static, 也不能用const修饰

```
{    return( sum/count ); //没有this指针, 不可以写this->sum或this->count
}
```

```
int main()
```

```
{    Student stud[3] = { Student(1001, 18, 70), Student(1002, 19, 78), Student(1005, 20, 98) };
    for (int i = 0; i < 3; i++)
    {
        stud[i].total();    //调用非静态成员函数
    }
    cout<<"the average score of the students is "<<Student::average()<<endl;
    return 0;               //调用静态成员函数
}
```



➤静态成员函数不能调用非静态成员函数，但是反过来是可以的

...

```
void Student::total() //定义非静态成员函数
{
    sum += score;
    count++;
    average(); //可以
}

float Student::average() //定义静态成员函数
{
    total(); //不可以，报错
    return(sum / count);
}
```

...





➤在静态成员函数中不能对非静态数据成员进行直接访问，而要通过对象参数的方式(不提倡，建议静态成员函数只访问静态数据成员)

...

```
float Student::average(Student &s) //定义静态成员函数
```

```
{
```

```
    num++;    //不可以，报错
```

```
    s.num++;  //可以
```

```
    return(sum / count);
```

```
}
```

...



➤静态成员函数的地址可用普通函数指针储存，而普通成员函数地址需要用类成员函数指针来储存

...

```
float (*pf1) () = &Student::average;
```

//普通的函数指针      静态成员函数的地址

```
void (Student:: * pf2) () = &Student::total;
```

//类成员函数指针                      普通成员函数地址

...



# 目录

- 类和对象的基本使用（构造函数补充）
- 复制构造函数的调用时机
- 共用数据的保护
- 静态成员
- 类模板



## 6.5 类模板

### 6.5.1 函数模板（复习）

- 建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为**函数模板**，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

一段代码, 两个功能  
1、两个int型求max  
2、两个double型求max



```
#include <iostream>
using namespace std;
template <typename T>
T max(T x, T y)
{   cout << sizeof(x) << ' ';
    return x>y?x:y;
}
int main()
{   int a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << max(a, b) << endl;
    cout << max(f1, f2) << endl;
    return 0;
}
```



## 6.5 类模板

### 6.5.1 函数模板（复习）

- 使用：
  - 仅适用于参数个数相同、类型不同，实现过程完全相同的情况
  - typename可用class替代
  - 类型定义允许多个

```
template <typename T1, typename T2>
```

```
template <class T1, class T2>
```

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
char max(T1 x, T2 y)
{
    cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}

int main()
{
    int a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << max(a, f1) << endl;
    cout << max(f2, b) << endl;
    return 0;
}
```



# 6.5 类模板

## 6.5.2 类模板

- 使用：
  - 仅适用于参数个数**相同**、类型**不同**，实现过程**完全相同**的情况
  - 类模板可以看作是**类的抽象**，称为**参数化的类**
  - 模板的具体实现称为实例化（instantiation）或具体化（specialization）
  - 类型定义允许多个

```
template <class T1, class T2>
```



## 示例：基础stack类

```
typedef unsigned long Item;    //只能处理unsigned long类型
class Stack
{
private:
    enum {MAX = 10};          // constant specific to class
    Item items[MAX];          // holds stack items
    int top;                   // index for top stack item
public:
    Stack();
    bool isempty() const;
    bool isfull() const;

    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item);    // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);           // pop top into item
};
```

## • 定义类模板：组织在头文件里



```
// stacktp.h -- a stack template
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>
class Stack
{
private:
    enum {MAX = 10};
    Type items[MAX];
    int top;
public:
    Stack();
    bool isempty();
    bool isfull();
    bool push(const Type & item);
    bool pop(Type & item);
};

template <class Type>
Stack<Type>::Stack() { top = 0; }
```

```
template <class Type>
bool Stack<Type>::isempty() { return top == 0; }

template <class Type>
bool Stack<Type>::isfull() {return top == MAX; }

template <class Type>
bool Stack<Type>::push(const Type & item)
{   if (top < MAX) ...
    else ...
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{   if (top > 0) ...
    else ...
}

#endif
```

体外实现的类限定符不要忘记泛型名！





- 使用模板类: 包含头文件

```
// stacktem1.cpp
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using namespace std;
```

```
int main()
{
    Stack<string> st;
    string po;
    char ch;
    ...
}
```

//可使用字符串作为订单ID

```
// stacktem2.cpp
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using namespace std;
```

```
int main()
{
    Stack<int> st;
    int po;
    char ch;
    ...
}
```

//可使用int值作为订单ID

使用模板实现泛型编程:  
实现相同算法下不同类型参数的统一编程



- 深入探讨模板类：不正确地使用指针栈

//例1:

```
int main()
```

```
{
```

```
    Stack<char *> st;
```

```
    char * po;
```

```
    ...
```

```
    cin >> po;    //错误
```

```
    ...
```

```
}
```

//仅创建指针，没有创建用于保存输入字符串的空间

整个解决方案		错误 1	警告 2
	代码	说明	
	C4703	使用了可能未初始化的本地指针变量"po"	
	C6001	使用未初始化的内存"po"。	
	C26495	未初始化变量 Stack<char *>::items。始终初始化成员变量 (type.6)。	



- 深入探讨模板类：不正确地使用指针栈

//例2:

```
int main()
```

```
{
```

```
    Stack<char *> st;
```

```
    char po[40];
```

```
    ...
```

```
    st.pop(po); //错误
```

```
    ...
```

```
}
```

```
template <class Type>
bool Stack<Type>::pop(Type& item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

//引用变量item必须引用某类型的左值，而不是数组名；  
代码不能为数组名赋值

整个解决方案		错误 2	警告 2
代码	说明		
C2664	"bool Stack<char *>::pop(Type &)": 无法将参数 1 从"char [40]"转换为"Type &"		
C6001	使用未初始化的内存"po"。		
E0434	无法用"char [40]"类型的值初始化"char *&"类型的引用(非常量限定)		
C26495	未初始化变量 Stack<char *>::items。始终初始化成员变量(type.6)。		



- 深入探讨模板类：不正确地使用指针栈

//例3:

```
int main()
{
    Stack<char *> st;
    char * po = new char[40];
    ...
    st.push(po);
    ...
}
```



```
template <class Type>
bool Stack<Type>::push(const Type& item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
```

//每次执行压入操作时，加入到栈中的地址都相同

对栈执行弹出操作时，得到的地址总是指向读入的最后一个字符串



- 数组模板:

- 非类型 (non-type) 或表达式 (expression) 参数

```
template <class T, int n>    //使用模板参数提供常规数组的大小  
class ArrayTP               //详见primer书  arraytp.h  
{  
    ...  
};
```

假设有: `ArrayTP<double, 12> eggweights;`

则: 编译器定义名为`ArrayTP<double, 12>`的类, 并创建一个类型为`ArrayTP<double, 12>`的`eggweight`对象。定义类时, 编译器将使用`double`替换`T`, 使用`12`替换`n`



- 模板多功能性:

- 递归使用模板

`ArrayTP < ArrayTP<int, 5>, 10> twodee;`

- twodee是一个包含10个元素的数组，其中每个元素都是一个包含5个int元素的数组
- 等价于常规的数组声明: `int twodee[10][5]`
- 示例程序: 详见primer书

- 使用多个类型参数

- 模板可以包含多个类型参数。假设希望类可以保存两种值，则可以创建并使用Pair模板来保存两个不同的值

`Pair<string, int>("The Purpled Duck", 5) //调用构造函数`

- 标准模板库提供了类似的模板，名为pair



- 模板多功能性:

- 默认类型模板参数

```
template <class T1, class T2 = int> class Topo {...};
```

- 类模板的新特性: 为类型参数提供默认值
- 如果省略T2的值, 编译器将使用int:

```
Topo<double, double> m1; // T1 is double, T2 is double
```

```
Topo<double> m2;          // T1 is double, T2 is int
```



- 将模板用作参数:

- 模板可以包含类型参数（如typename T）和非类型参数（如int n）
- 模板还可以包含本身就是模板的参数，用于实现STL（标准模板库，不深入讨论）

```
template <template <typename T> class Thing>  
class Crab {...};
```

- 模板参数是template <typename T> class Thing
- 其中template <typename T> class 是类型，Thing是参数
- 假设有：Crab<Stack> nebula; 则Stack的声明需与Thing的声明匹配：

```
template <typename T>  
class Stack {...};      //stacktp.h中已定义
```





- 将模板用作参数示例:

```
// tempparm.cpp - templates as parameters
```

```
#include <iostream>
```

```
#include "stacktp.h"
```

```
using namespace std;
```

```
template <template <typename T> class Thing >
```

```
class Crab
```

```
{
```

```
    private:
```

```
        Thing<int> s1;
```

```
        Thing<double> s2;
```

```
    public:
```

```
        Crab() {};
```

```
        // assumes the thing class has push and pop() members
```

```
        bool push(int a, double x) { return s1.push(a) && s2.push(x); }
```

```
        bool pop(int& a, double& x) { return s1.pop(a) && s2.pop(x); }
```

```
}; //接下一页
```



// 接上一页

```
int main()
{
    Crab<Stack> nebula; // Stack must match template <typename T> class Thing
    int ni;
    double nb;
    cout << "Enter int double pairs, such as 4 3.5 (0 0 to end):\n";
    while (cin >> ni >> nb && ni > 0 && nb > 0)
    {
        if (!nebula.push(ni, nb)) break;
    }
    while (nebula.pop(ni, nb))
        cout << ni << ", " << nb << endl;
    cout << "Done.\n";
    return 0;
}
```

```
Enter int double pairs, such as 4 3.5 (0 0 to end):
50 22.48
25 33.87
60 19.12
0 0
60, 19.12
25, 33.87
50, 22.48
Done.
```



# 总结

- 类和对象的基本使用（构造函数补充：了解）
- 复制构造函数的调用时机（掌握）
- 共用数据的保护（熟悉）
- 静态成员（熟悉）
- 类模板（了解）