



# 第六章 类和对象的使用进阶

## 模块6.2：数据共享与参数化类

主讲教师：同济大学计算机科学与技术学院 陈宇飞  
同济大学计算机科学与技术学院 龚晓亮



# 目录

- 共用数据的保护
- 静态成员
- 类模板



# 目录

- 共用数据的保护

- 基本概念
- 常对象
- 常对象成员
- 共用数据小结



# 1.1 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

`const` 类名 对象名(初始化实参表)

## ➤ 常对象成员

常数据成员

常成员函数



# 1.2 常对象

- 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



# 1.3 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();
    ...
}
```





# 1.4 共用数据小结

	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

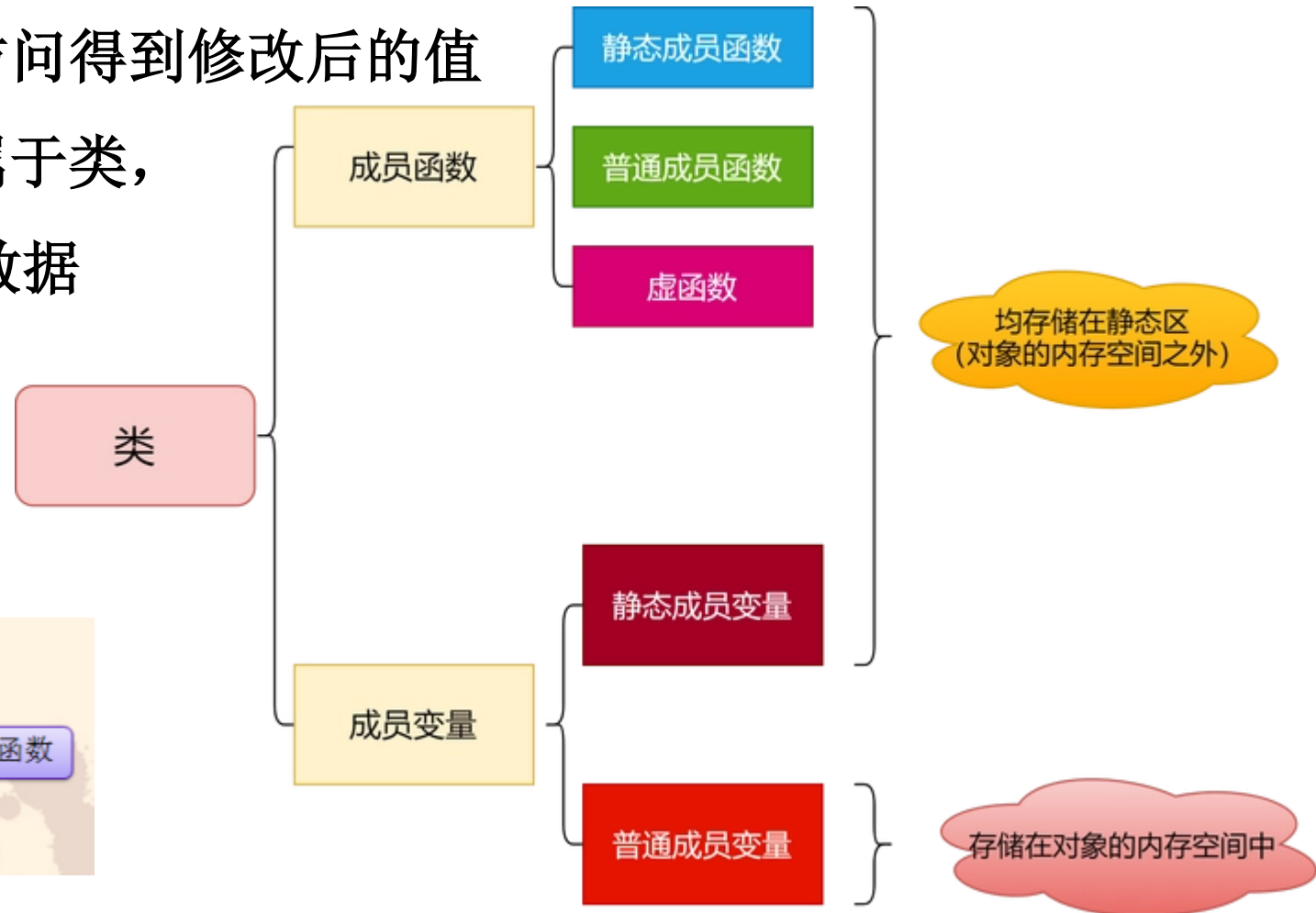
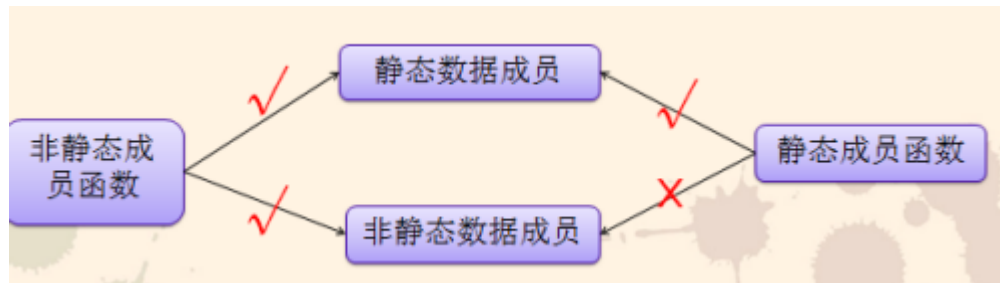
- 静态成员

- 基本概念
- 静态数据成员
- 静态成员函数

## 2.1 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





## 2.2 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  
    ...  
};
```



## 2.2 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



## 2.3 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

- 类模板
  - 函数模板（复习）
  - 类模板的使用



## 3.1 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;
}
```



## 3.1 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;
}
```



## 3.2 类模板的使用

- 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





- 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

- 共用数据的保护（熟悉）

- 基本概念
- 常对象
- 常对象成员
- 共用数据小结

- 静态成员（熟悉）

- 基本概念
- 静态数据成员
- 静态成员函数

- 类模板（了解）

- 函数模板（复习）
- 类模板的使用