第十章 模板类

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

- 智能指针模板类
- 标准模板库

目录

- 智能指针模板类
  - 基本概念
  - 使用智能指针
  - 有关智能指针的注意事项
  - unique_ptr为何优于auto_ptr
  - 选择智能指针

1.1 基本概念

- 请查找下面函数的缺陷：

```
void remodel(std::string& str)
{
    std::string * ps = new std::string(str);
    ...
    str = *ps;
    return;
}
```

- 每当调用函数时，该函数都分配堆中的内存，但从不收回，从而导致内存泄露

1.1 基本概念

- “别忘了delete ps” 有时可能记住，但有时可能在不经意间删除或注释掉

```
void remodel(std::string& str)
{
    std::string * ps = new std::string(str);
    ...
    str = *ps;
    delete ps;
    return;
}
```

1.1 基本概念

- 当出现异常时，delete将不被执行，因此也会出现内存泄露

```
void remodel(std::string& str)
{
      std::string * ps = new std::string(str);
      ...
      if (weird_thing())
          throw exception();
      str = *ps;
      delete ps;
      return;
}
```

- 现在的需求是希望remodel()这样的函数终止时（不论是正常终止，还是因异常终止），本地变量都将从栈内存中删除（包括指针ps占据的内存被释放）

1.2 使用智能指针

- 三个智能指针模板（auto_ptr、unique_ptr和shared_ptr）都定义了类似指针的对象，可以将new获得（直接或间接）的地址赋给这种对象。当智能指针过期时，其析构函数将使用delete来释放内存

```
void demo1()
{
    double* pd = new double;//#1
    *pd = 25.5;             //#2
    return;                 //#3
}
```

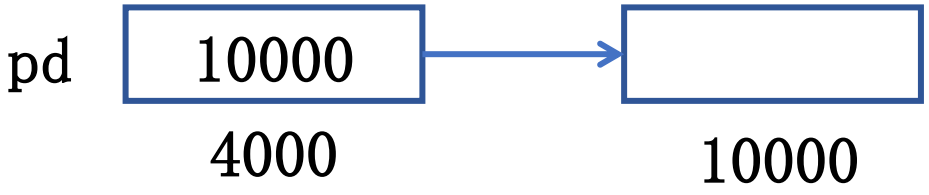
```
void demo1()
{
  auto_ptr<double> ap(new double); //#1
  *ap= 25.5;                       //#2
  return;                          //#3
}
```

- 智能指针是行为类似于指针的类对象，但这种对象还有其他功能

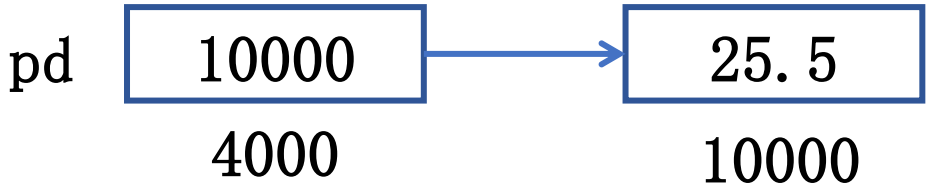
1.2 使用智能指针

```
void demo1()
{
    double* pd = new double;//#1
    *pd = 25.5;                //#2
    return;                    //#3
}
```

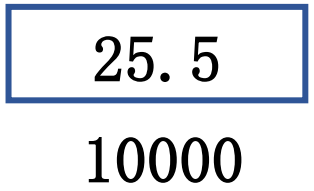
#1.为pd和double值分配存储空间，保存地址：



#2.将值赋值到动态内存中：



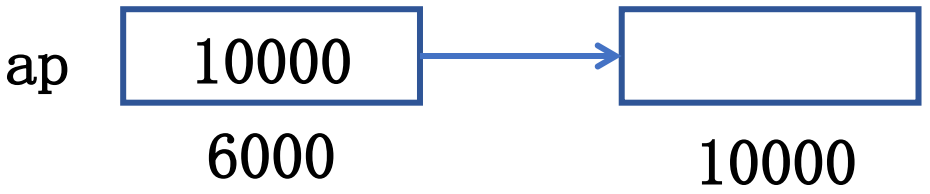
#3.删除pd，值被保留在动态内存中：



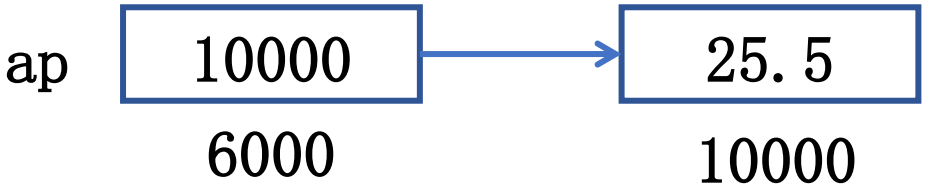
1.2 使用智能指针

```
void demo2()
{
    auto_ptr<double> ap(new double);//#1
    *ap= 25.5;                      //#2
    return;                         //#3
}
```

#1. 为ap和double值分配存储空间，保存地址：



#2. 将值赋值到动态内存中：



#3. 删除ap, ap的析构函数释放动态内存

1.2 使用智能指针

- 创建智能指针对象，必须包含头文件memory，实现该文件模板定义

```
template<class X> class auto_ptr {
public:
    explicit auto_ptr(X* p = 0) throw()   {};
   ...
};
```

- 使用通常的模板语法来实例化所需类型的指针

```
auto_ptr<double> pd(new double);
auto_ptr<string> ps(new string);
```

```
unique_ptr<double> pdu(new double);
shared_ptr<string> pss(new string);
```

1.2 使用智能指针

- 注意：

➢异常规范（包括 throw()）在 C++11 中被弃用，并在 C++17 中被完全移除。原因是它们在实际使用中增加了编译器的复杂性，并可能导致意外的程序行为（如 std::unexpected() 的调用）。C++11 引入了 noexcept 关键字作为异常规范的现代替代品

➢auto_ptr 本身由于其不安全的复制语义已经被弃用，并在 C++17 中被移除。最佳实践是使用 std::unique_ptr 或 std::shared_ptr 来管理动态分配的内存

```
auto_ptr<double> pd(new double);
auto_ptr<string> ps(new string);
```

```
unique_ptr<double> pdu(new double);
shared_ptr<string> pss(new string);
```

1.2 使用智能指针

- 要转换remodel()函数，应按3个步骤进行：

  #1.包括头文件memory；

  #2.将指向string的指针替换为指向string的智能指针对象；

  #3.删除delete语句

```
void remodel(std::string& str)
{
   std::string * ps = new std::string(str);
   ...
   if (weird_thing())
      throw exception();
   str = *ps;
   delete ps;
   return;
}
```

```
#include <memory> //# 1
void remodel(std::string& str)
{
   std::auto_ptr<std::stringbuf> ps (new
std::string(str));//# 2
   ...
   if (weird_thing())
       throw exception();
   str = *ps;
   // delete ps; NO LONGER NEEDED//# 3
   return;
}
```

```
// smrtptrs.cpp -- using three kinds of smart pointers
```

接后页

```
#include <iostream>
#include <string>
#include <memory>

class Report
{
private:
    std::string str;
public:
    Report(const std::string s):str(s) { std::cout << "Object created!\n"; }
    ~Report() { std::cout << "Object deleted!\n"; }
    void comment() const { std::cout << str << "\n"; }
};
```

续前页

```
// smrtptrs.cpp -- using three kinds of smart pointers
int main()
{
    {
        std::auto_ptr<Report> ps (new Report("using auto_ptr"));
        ps->comment();   // use -> to invoke a member function
    }
    {
        std::shared_ptr<Report> ps (new Report("using shared_ptr"));
        ps->comment();
    }
    {
        std::unique_ptr<Report> ps (new Report("using unique_ptr"));
        ps->comment();
    }

    return 0;
}
```

```
Object created!
using auto_ptr
Object deleted!
Object created!
using shared_ptr
Object deleted!
Object created!
using unique_ptr
Object deleted!
```

1.2 使用智能指针

- 所有智能指针类都有一个explicit构造函数，该构造函数将指针作为参数。因此，需要显式将指针转换为智能指针对象

```
shared_ptr<double>pd;
double* p_reg = new double;
pd = p_reg;                              // not allowed (implicit conversion)
pd =  shared_ptr<double>(p_reg>;         // allowed (explicit conversion)
shared_ptr<double> pshared = p_reg;      // not allowed (implicit conversion)
shared_ptr<double> pshared(p_reg);       // allowed (explicit conversion)
```

1.2 使用智能指针

- 由于智能指针模板类的定义方式，智能指针对象的很多方面类似于常规指针

```
shared_ptr<string> ps;
```

  - 可以对智能指针执行解引用操作（*ps）
  - 可以用它来访问结构成员（ps->pufflndex）
  - 可以将它赋给指向相同类型的常规指针
  - 还可以将智能指针对象赋给另一个同类型的智能指针对象
  - 但要注意可能引起问题（见后）

1.2 使用智能指针

- 由于智能指针模板类的定义方式，智能指针对象的很多方便类似于常规指针
  - 但要注意可能引起问题

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);// NO!
```

当pvac过期时，程序将把delete运算符用于非堆内存

```
string* vacation = new string("I wandered lonely as a cloud.");
shared_ptr<string> pvac(vacation);
```

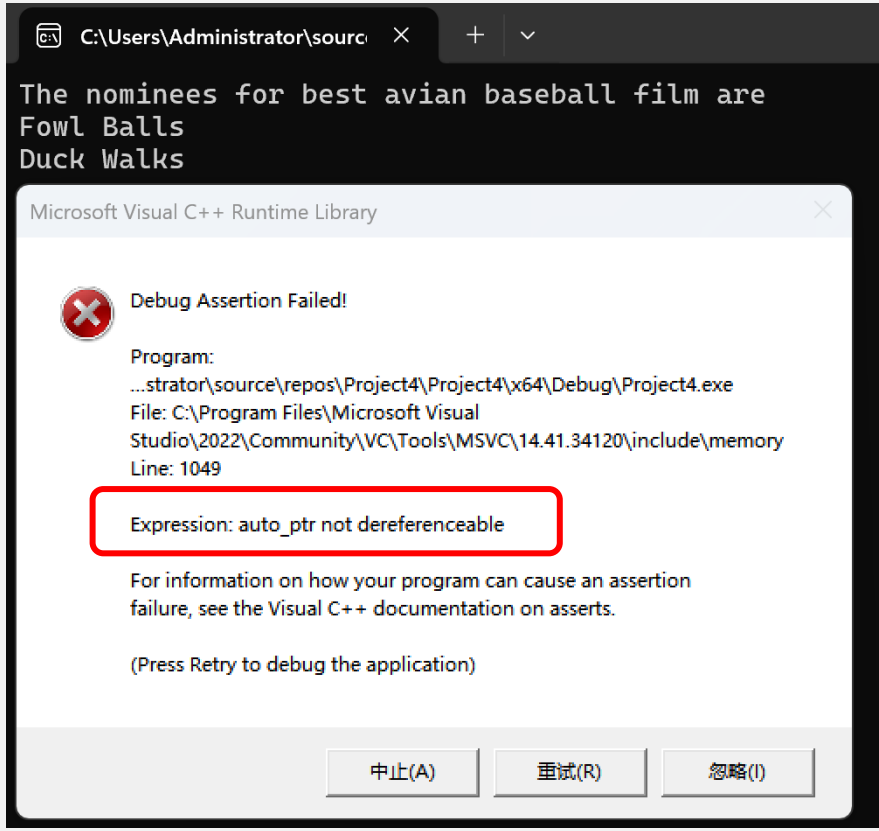
1.3 有关智能指针的注意事项

```
auto_ptr<string> ps(new string("I reigned in the year 2010"));
auto_ptr<string> vocation;
vocation = ps;//两个指针指向同一个string，程序将试图删除同一个对象两次，一次是ps过期时，一次是vocation过期时
```

避免这一问题的方法有：

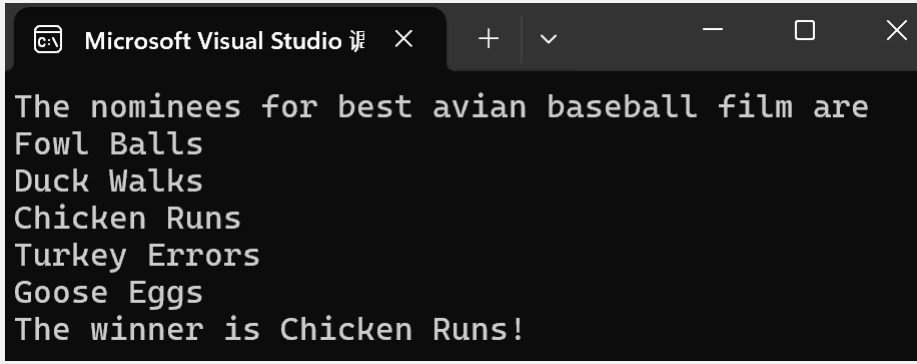
- 定义赋值运算符，使之执行深拷贝，这样两个智能指针指向不同的内存
- 建立所有权概念，对于特定的对象，只能有一个智能指针可拥用它。只有拥有对象的智能指针的构造函数会删除该对象。然后，让赋值操作转让所有权。（auto_ptr/unique_ptr的策略，unique_ptr的更严格）
- 创建智能更高的指针，跟踪引用特定对象的智能指针数（引用计数reference counting）。赋值时计数加1，指针过期计数减1，仅当最后一个指针过期时，才调用delete。（shared_ptr的策略）

```
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>
int main()
{
    using namespace std;
    auto_ptr<string> films[5] =
    {
        auto_ptr<string>(new string("Fowl Balls")),
        auto_ptr<string>(new string("Duck Walks")),
        auto_ptr<string>(new string("Chicken Runs")),
        auto_ptr<string>(new string("Turkey Errors")),
        auto_ptr<string>(new string("Goose Eggs"))
    };
    auto_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership
    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";
      return 0;}
```



- 该语句将所有权从films[2]转让给pwin，这导致films[2]不再指向该字符串
- 但程序打印films[2]指向的字符串时，发现其是一个空指针

```
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>
int main()
{
    using namespace std;
    shared_ptr<string> films[5] =
    {
        shared_ptr<string>(new string("Fowl Balls")),
        shared_ptr<string>(new string("Duck Walks")),
        shared_ptr<string>(new string("Chicken Runs")),
        shared_ptr<string>(new string("Turkey Errors")),
        shared_ptr<string>(new string("Goose Eggs"))
    };
    shared_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership
    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";
      return 0;}
```



- 该语句使得两个两个指针指向同一个对象，因此引用计数从1增加到2
- 后声明的pwin首先调用其析构函数，使得引用计数降到1
- 然后，shared_ptr数组的成员被释放，films[2]调用析构函数时，引用计数降低到0，并释放以前分配的空间

```cpp
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>

int main()
{
    using namespace std;
    unique_ptr<string> films[5] =
    {
        unique_ptr<string>(new string("Fowl Balls")),
        unique_ptr<string>(new string("Duck Walks")),
        unique_ptr<string>(new string("Chicken Runs")),
        unique_ptr<string>(new string("Turkey Errors")),
        unique_ptr<string>(new string("Goose Eggs"))
    };
    unique_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";
    // cin.get();
    return 0;
}
```

- 如果使用unique_ptr时，程序不会等到运行阶段崩溃，而是在编译器下就报错

整个解决方案 | 错误 2 | 警告 0 | 消息 0 | 生成 + IntelliSense

	代码	说明	项目
abc	E1776	无法引用 函数 "std::unique_ptr<_Ty, _Dx>::operator=(const std::unique_ptr<_Ty, _Dx> &) [其中 _Ty=std::string, _Dx=std::default_delete<std::string>]" (已声明 所在行数:3452，所属文件:"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.41.34120\include\memory") -- 它是已删除的函数	Project4
⊗	C2280	"std::unique_ptr<std::string,std::default_delete<std::string>> &std::unique_ptr<std::string,std::default_delete<std::string>>::operator =(const std::unique_ptr<std::string,std::default_delete<std::string>> &)": 尝试引用已删除的函数	Project4

1.4 unique_ptr为何优于auto_ptr

```
auto_ptr<string> p1(new string("auto")); //#1
auto_ptr<string> p2;                     //#2
p2 = p1;                                 //#3
```

- 在语句#3，p2接管string对象的所有权后，p1的所有权将被剥夺，成为悬挂指针，不再指向有效的数据

```
unique_ptr<string> p3(new string("auto")); //#4
unique_ptr<string> p4;                     //#5
p4 = p3;                                   //#6
```

- 编译器认为语句#6非法，避免了p3不再指向有效数据的问题。因此unique_ptr比auto_ptr更安全（编译阶段错误比潜在的程序崩溃更安全）

1.4 unique_ptr为何优于auto_ptr

```
unique_ptr<string> demo(const char* s)
{
      unique_ptr<string> temp(new string(s));
      return temp;
}
unique_ptr<string> ps;
ps = demo("Uniquely special");
```

- demo()返回一个临时unique_ptr，ps接管了原本归返回的unique_ptr所有的string对象，而返回的unique_ptr被销毁。这样的方式没有出现指向无效数据的悬挂指针
- 如果unique_ptr是临时右值，这种赋值是编译器允许的；但如果源unique_ptr将存在一段时间，编译器将禁止

1.4 unique_ptr为何优于auto_ptr

```
using namespace std;
unique_ptr<string> pu1(new string("Hi ho!"));
unique_ptr<sting> pu2;
pu2 = pu1;  // #1 illegal

unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string("Yo!"));// #2 legal
```

- 语句#1将留下悬挂的unique_ptr(pu1)，导致危害，因此不允许
- 语句#2调用unique_ptr构造函数，创建的临时对象将所有权转让给pu3后就会被销毁，因此是允许
- 因此，建议在容器对象中禁止使用auto_ptr，但允许使用unique_ptr

1.4 unique_ptr为何优于auto_ptr

- C++中有一个标准库函数std::move(),可以将一个unique_ptr赋给另一个unique_ptr
- C++新增的移动构造函数和右值引用，来确保unique_ptr的安全使用

```
using namespace std;
unique_ptr<string> ps1, ps2;
ps1 = demo("Uniquely special");
ps2 = move(ps1); // transfer ownership
ps1 = demo(" and more");
cout << *ps2 << *ps1 << endl;
```

- unique_ptr有一个可用于数组的变体

1.4 unique_ptr为何优于auto_ptr

	指向栈内变量	指向堆内（动态内存）变量 new/delete	指向堆内（动态内存）变量数组 new[]/delete[]
auto_ptr	×	√	×
shared_ptr	×	√	×
unique_ptr	×	√	√

```
std::unique_ptr<double[]>pda(new doubel(5)); // will use delete[] to release memory
```

1.5 选择智能指针

	选择智能指针的情况	
shared_ptr	如果程序需要使用多个指向同一个对象的指针时，选择shared_ptr	共享所有权。一个shared_ptr指向的资源可以被多个shared_ptr实例共享。这是通过内部的引用计数机制来实现的。当最后一个拥有资源的shared_ptr 被销毁时，资源才会被释放
unique_ptr	如果程序不需要多个指向同一个对象的指针时，可使用unique_ptr	独占所有权。一个unique_ptr 指向的资源只能被一个 unique_ptr 实例拥有。当尝试将一个unique_ptr赋值给另一个 unique_ptr 或者传递给函数时，默认的行为是转移所有权，这要求源unique_ptr 被销毁（即不再拥有资源），并且目标 unique_ptr 开始拥有这个资源。这种转移所有权的操作不允许复制，只允许移动

1.5 选择智能指针

- 如果程序需要使用多个指向同一个对象的指针时，选择shared_ptr

具体指：

- 有一个指针数组，并使用一些辅助指针来标识特定的元素，如最大元素和最小元素
- 两个对象包含指向第三个对象的指针
- STL容器包含指针

1.5 选择智能指针

- 如果程序不需要多个指向同一个对象的指针时，可使用unique_ptr

具体指：

- 如果函数使用new分配内存，并返回指向该内存的指针，将其返回类型声明为unique_ptr是不错的选择。当所有权转让给接受返回值的unique_ptr，该智能指针将负责调用delete
- 可以将unique_ptr存储到STL容器中，只要不调用将一个unique_ptr复制或赋值给另一个的方法或算法（如sort()）

1.5 选择智能指针

```
unique_ptr<int> make_int(int n)
{
        return unique_ptr<int>(new int(n));
}
void show(unique_ptr<int> &p)        // pass by reference
{
        cout << *a << ' ';
}
int main()
{       ...
        vector<unique_ptr<int>> vp(size);
        for (int i = 0; i < vp.size(); i++)
                vp[i] = make_int(rand()%1000;      //copy temporary unique_ptr
        vp.push_back(make_int(rand() % 1000));  //ok because arg is temporary
        for_each(vp.begin(), vp.end(), show);  //use for_each() to show all elements
        ...
}
```

1.5 选择智能指针

```
unique_ptr<int> pup(make_int(rand() % 1000));//ok
```

- 创建了一个 unique_ptr<int> 实例 pup，它拥有通过 make_int返回的 unique_ptr<int> 指针

```
shared_ptr<int> spp(pup);// not allowed, pup an lvalue (left value)
shared_ptr<int> spp(move(pup)); // ok, moves ownership from pup to spp
```

- 试图将一个unique_ptr直接转换为shared_ptr，这是不允许的。因为unique_ptr不能被复制，它只能被移动。如果想要将unique_ptr 转换为shared_ptr，需要使用 move 来转移所有权

```
shared_ptr<int> spr(make_int(rand() % 1000));//ok
```

- 直接创建了一个shared_ptr，它拥有一个 int 对象，因为 make_int 返回的是一个右值unique_ptr<int> ，可以被用来初始化shared_ptr（通过shared_ptr的一个显式构造函数进行转换）

目录

- 智能指针模板类
- 标准模板库

目录

- 标准模板库
  - 基本概念
  - 模板类vector
  - 可对矢量执行的操作
  - 对矢量可执行的其他操作
  - 基于范围的for循环

2.1 基本概念

- STL提供了一组表示容器、迭代器、函数对象和算法的模板
  - 容器是一个与数组类似的单元，可以存储若干个值。STL容器是同质的，即存储的值的类型相同
  - 迭代器能够用来遍历容器的对象，与能够遍历数组的指针，是广义的指针
  - 函数对象可以是类对象或函数指针（包括函数名，函数名被用作指针）
  - 算法是完成特定任务的方法
- STL能够构造各种容器（包括数组、队列和链表）和执行各种操作（包括搜索、排序和随机排列）

2.2 模板类vector

- 模板类vector类似于string类，是一种动态数组。它是使用new创建动态数组的替代品。本质上，vector类使用new和delete来管理内存，但这些工作是自动完成的
  - 要使用vector对象，必须要声明头文件vector
  - vector包含在名称空间std中
  - 模板使用不同的语法来指出它存储的类型
  - vector类使用不同的语法来指定元素数

2.2 模板类vector

- 声明创建一个名为vt的vector对象，它可存储n_elem个类型为typeName的元素，n_elem可以是整型常量，也可以是整型变量

```
vector <typeName> vt (n_elem);
```

```
#include<vector>
...
using namespace std;
vector<int> vi;//create a zero-size array of int
int n;
cin >> n;
vector<double> vd(n);//create an array of n doubles
```

2.2 模板类vector

- vector类的功能比数组强大，但代价是效率稍低。如果需要固定长度的数组，使用数组是更佳的选择，但代价是不那么方便和安全
- C++11新增了模板类array，它也位于名称空间std中
- 和数组一样，array对象的长度也固定，也使用栈（静态内存分配），而不是堆（自由存储区），因此效率与数组相同，但更方便，更安全
- 需要包含头文件array，array对象的创建语法与vector稍有不同

2.2 模板类vector

- 与创建vector对象不同，n_elem不能是变量

```
array <typeName, n_elem> arr;
```

```
#include<array>
...
using namespace std;
array <int, 5> ai; // create array object of 5 integers
array <double, 4> ad={1.2, 2.1, 3.43, 4.3} ;
```

```
// choices.cpp -- array variations                                    //后页续
#include <iostream>
#include <vector>   // STL C++98
#include <array>    // C++0x
int main()
{
    using namespace std;
// C, original C++
    double a1[4] = {1.2, 2.4, 3.6, 4.8};
// C++98 STL
    vector<double> a2(4);   // create vector with 4 elements
// no simple way to initialize in C98
    a2[0] = 1.0/3.0;
    a2[1] = 1.0/5.0;
    a2[2] = 1.0/7.0;
    a2[3] = 1.0/9.0;
// C++0x -- create and initialize array object
    array<double, 4> a3 = {3.14, 2.72, 1.62, 1.41};
    array<double, 4> a4;
    a4 = a3;     // valid for array objects of same size
```

```
// choices.cpp -- array variations                                    //续前页
// use array notation
    cout << "a1[2]: " << a1[2] << " at " << &a1[2] << endl;
    cout << "a2[2]: " << a2[2] << " at " << &a2[2] << endl;
    cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
    cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;
// misdeed
    a1[-2] = 20.2;
    cout << "a1[-2]: " << a1[-2] <<" at " << &a1[-2] << endl;
    cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
    cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

    return 0;
}
```

```
a1[2]: 3.6 at 000000192651F458
a2[2]: 0.142857 at 000001DCE6545C60
a3[2]: 1.62 at 000000192651F4D8
a4[2]: 1.62 at 000000192651F518
a1[-2]: 20.2 at 000000192651F438
a3[2]: 1.62 at 000000192651F4D8
a4[2]: 1.62 at 000000192651F518
```

- 数组、vector对象还是array对象，都可以使用标准数组表示来访问各个元素（相同点）（注意：运算符[]被重载的原因）
- 从地址可知，array对象和数组对象存储在相同的内存区域（即栈）中，而vector对象存储在另一个区域（自由存储区域或堆中）（区别）
- 可以将一个array对象赋给另一个array对象；而对于数组，必须逐个元素复制（区别）
- vector和array对象，通过成员函数at(),begin(),end()等，来确定边界，以免无意超界（区别）

```
// vect1.cpp -- introducing the vector template                    //后页续

#include <iostream>
#include <string>
#include <vector>

const int NUM = 5;
int main()
{
    using std::vector;
    using std::string;
    using std::cin;
    using std::cout;
    using std::endl;

    vector<int> ratings(NUM);
    vector<string> titles(NUM);
    cout << "You will do exactly as told. You will enter\n"
         << NUM << " book titles and your ratings (0-10).\n";
```

```
// vect1.cpp -- introducing the vector template
    int i;
    for (i = 0; i < NUM; i++)
    {
        cout << "Enter title #" << i + 1 << ": ";
        getline(cin, titles[i]);
        cout << "Enter your rating (0-10): ";
        cin >> ratings[i];
        cin.get();
    }
    cout << "Thank you. You entered the following:\n"
         << "Rating\tBook\n";
    for (i = 0; i < NUM; i++)
    {
        cout << ratings[i] << "\t" << titles[i] << endl;
    }
    return 0;
}
```

//续前页

```
You will do exactly as told. You will enter
5 book titles and your ratings (0-10).
Enter title #1: The Cat Who Knew C++
Enter your rating (0-10): 6
Enter title #2: Felonious Felines
Enter your rating (0-10): 4
Enter title #3: Warlords of Wonk
Enter your rating (0-10): 3
Enter title #4: Don't Tounch That Metaphor
Enter your rating (0-10): 5
Enter title #5: Panic Oriented Programming
Enter your rating (0-10): 8
Thank you. You entered the following:
Rating  Book
6       The Cat Who Knew C++
4       Felonious Felines
3       Warlords of Wonk
5       Don't Tounch That Metaphor
8       Panic Oriented Programming
```

- 该程序创建两个vector对象--一个是int规范，另一个是string规范，都包含5个元素
- 该程序使用vector模板只是为方便创建动态分配的数组

2.3 可对矢量执行的操作

- 除分配存储空间外，vector模板还可以完成的操作（所有STL容器提供的一些基本方法）
  - size():返回容器中元素数目
  - swap():交换两个容器的内容
  - begin():返回一个指向容器中第一个元素的迭代器
  - end():返回一个表示超过容器尾的迭代器

迭代器，是广义的指针，它可以是指针，也可以是可对其执行类似指针的操作，例如：

解引用（operator*()）和
递增（operator++()）的对象

2.3 可对矢量执行的操作

- 每个容器类都定义了一个合适的迭代器，该迭代器的类型是一个名为iterator的typedef，其作用域为整个类

```
vector<double>::iterator pd; // pd is an iterator for type double

vector<double> scores; //scores是一个vector<double>对象
pd = scores.begin(); // set pd point to the first element of scores
*pd = 22.3; // set the value of the first element of scores to 22.3
pd++; // increment pd to point to the next element of scores
```

- 结合C++11自动类型推断

```
vector<double>::iterator pd = scores.begin();
auto pd = scores.begin(); //C++11 automatic type deduction (与前句等价)
```

2.3 可对矢量执行的操作

- 超过结尾（past-the-end）迭代器，指向容器最后一个元素后面那个元素。end()成员函数标识超过结尾的位置
- vector模板类中只有某些STL容器才有的方法：
  - push_back()：将元素添加到矢量末尾，负责内存管理，增加矢量的长度
  - erase()：删除矢量中给定区间的元素，它接受两个迭代器参数，定义要删除的区间
  - insert()：接受三个迭代器参数，第一个参数指定新元素的插入位置，第二和第三个迭代器参数定义了被插入区间，该区间通常是另一个容器对象的一部分

2.3 可对矢量执行的操作

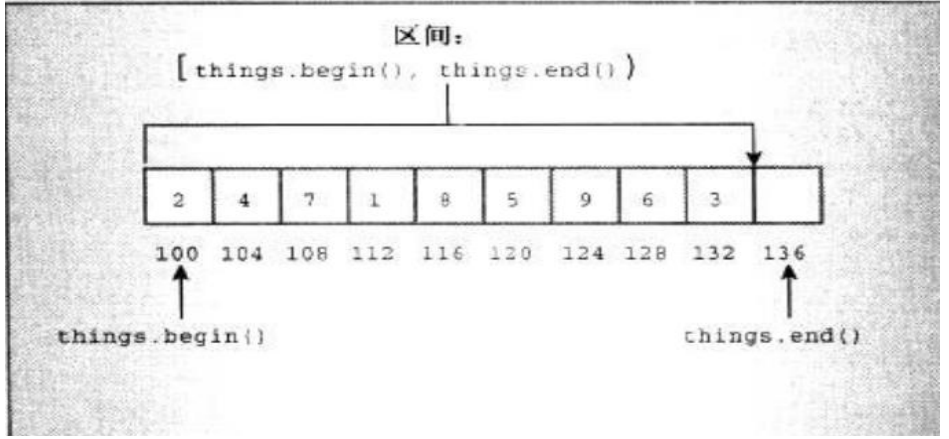
```
for(pd = scores.begin(); pd != scores.end(); pd++)//遍历整个容器内容
      cout << *pd << endl;
```

```
//每次循环都给scores对象增加一个元素，无需了解元素数目
vector<double> scores; // create an empty vector
double temp;
while (cin >> temp && temp >= 0)
      scores.push_back(temp);
cout << "You entered " << scores.size() << " scores\n";
```

```
//删除begin()和begin()+1指向的元素
scores.erase(scores.begin(), scores.begin() + 2);
```

注意：迭代器指定的begin()到begin()+2，其范围为begin()到begin()+2（但不包括begin()+2）

2.3 可对矢量执行的操作



STL的区间概念

```
//将矢量new_v中除第一个元素外的所有元素插入到old_v矢量的第一个元素前面:
vector<int> old_v;
vector<int> new_v;
...
old_v.insert(old_v.begin(), new_v.begin()+1, new_v.end());

//将矢量new_v中除第一个元素外的所有元素插入到old_v矢量的最后一个元素后面:
old_v.insert(old_v.end(), new_v.begin() + 1, new_v.end());
```

2.4 对矢量可执行的其他操作

- STL定义了非成员（non-memner）函数来执行如搜索、排序、随机排序等操作
- 有时即使有执行相同任务的非成员函数，STL也会定义一个成员函数。因为，成员函数的效率比通用非成员函数高
  - 如vector的成员函数swap()的效率比非成员函数swap()高，但非成员函数能交换两个不同类型的容器的内容
- 具有代表性的STL函数：for_each()、random_shuffle()和sort()

2.4 对矢量可执行的其他操作

- for_each()：可用于多容器类，接受3个参数。前两个定义容器中区间的迭代器，最后一个指向函数的指针（函数对象）。该函数将被指向的函数应用于容器区间中的各个元素，被指向函数不能修改容器元素值，可用来替代for循环

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
      ShowReview(*pr);
```

等价

```
for_each(books.begin(), books.end(), ShowReview);
```

2.4 对矢量可执行的其他操作

- random_shuffle()：接受2个指定区域的迭代器参数，并随机排列该区域中的元素。该函数要求容器类允许随机访问，vector类满足

```
random_shuffle(books.begin(), books.end());
```

- sort()：也要求容器支持随机访问
  - 接受两个定义区间迭代器参数，并使用为存储在容器中的类型元素定义的比较运算符operator<，对区间中的元素进行操作
  - 接受3个参数版本，前两参数也是指定区间的迭代器，最后一个参数是指向要使用的函数的指针（函数对象），不使用比较运算符operator<

2.4 对矢量可执行的其他操作

```
//sort()的2参数版本

vector<int>coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());

bool operator<(const Review& r1, const Review& r2)
{
      if (r1.title < r2.title)
            return true;
      else if (r1.title == r2.title && r1.rating < r2.rating)
            return true;
      else
            return false;
}
```

如果title成员都相同，则再按rating排序比较

```
struct Review {
    std::string title;
    int rating;
};
```

2.4 对矢量可执行的其他操作

```
//sort()的3参数版本

bool WorseThan(const Review& r1, const Review& r2)
{
      if (r1.rating < r2.rating)
            return true;
      else
            return false;
}
sort(books.begin(), books.end(), WorseThan);
```

注意：与operator<()相比，WorseThan()函数执行的对Review对象进行排序的工作不那么完整

全排序（total ordering），是相同

完整弱排序（strict weak ordering），是等价

2.5 基于范围的for循环（C++11）

```
double prices[5] = { 4.99, 10.99, 6.87, 7.99, 8.49 };
for (double x : prices)
      cout << x << endl;
```

```
for_each(books.begin(), books.edn(), ShowReview));
```

```
for (auto x : books) ShowReview(x);
```

```
void InflateReview(Review & r) { r.rating++; }
```

```
for (auto x : books) InflateReview(x);
```

总结

- 智能指针模板类（了解）
  - 基本概念
  - 使用智能指针
  - 有关智能指针的注意事项
  - unique_ptr为何优于auto_ptr
  - 选择智能指针
- 标准模板库（掌握）
  - 基本概念
  - 模板类vector
  - 可对矢量执行的操作
  - 对矢量可执行的其他操作
  - 基于范围的for循环