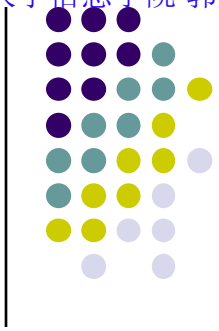
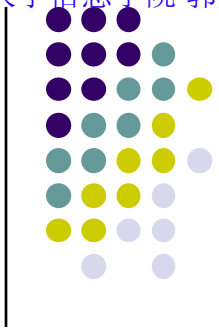
新标准C++程序设计

北京大学信息学院　郭 炜

GWPL@PKU.EDU.CN



标准模板库(一)



标准模板库 STL

C++ 语言的核心优势之一就是便于软件的重用

C++中有两个方面体现重用：

1. 面向对象的思想：继承和多态，标准类库

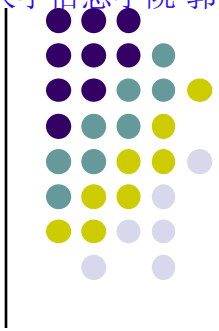
2. 泛型程序设计(generic programming) 的思想：模板机制，以及标准模板库 STL

泛型程序设计，简单地说就是使用模板的程序设计法。

将一些常用的数据结构（比如链表，数组，二叉树）和算法（比如排序，查找）写成模板，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。

标准模板库（Standard Template Library）就是一些常用数据结构和算法的模板的集合。主要由 Alex Stepanov 开发，于1998年被添加进C++标准

有了STL，不必再写大多的标准数据结构和算法，并且可获得非常高的性能。



STL中有几个基本的概念：

容器：可容纳各种数据类型的数据结构。

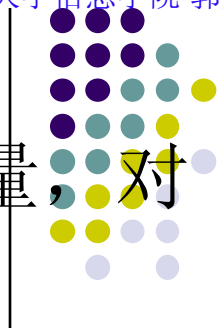
迭代器：可依次存取容器中元素的东西

算法：用来操作容器中的元素的函数模板。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象。函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

比如，对于 int array[100];

这个数组就是个容器，而 int * 类型的指针变量就可以作为迭代器，可以为这个容器编写一个排序的算法

1 容器概述



可以用于存放各种类型的数据（基本类型的变量，对象等）的数据结构。

容器分为三大类：

1) 顺序容器

vector, deque, list

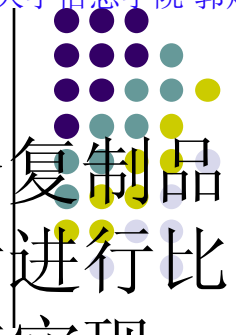
2) 关联容器

set, multiset, map, multimap

前2者合称为第一类容器

3) 容器适配器

stack, queue, priority_queue



对象被插入容器中时，被插入的是对象的一个复制品。许多算法，比如排序，查找，要求对容器中的元素进行比较，所以，放入容器的对象所属的类，往往还应该实现 == 和 < 运算符。

1.1 顺序容器简介

1) vector 头文件 <vector>

实际上就是个动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能。

2) deque 头文件 <deque>

也是个动态数组，随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能。

3) list 头文件 <list>

双向链表，在任何位置增删元素都能在常数时间完成。不支持随机存取。

上述三种容器称为顺序容器，是因为元素的插入位置同元素的值无关。

1.2 关联容器简介

关联式容器内的元素是排序的，插入任何元素，都按相应的排序准则来确定其位置。关联式容器的特点是在查找时具有非常好的性能。

1) set/multiset: 头文件 <set>

set 即集合。set中不允许相同元素，multiset中允许存在相同的元素。

2) map/multimap: 头文件 <map>

map与set的不同在于map中存放的是成对的key/value。

并根据key对元素进行排序，可快速地根据key来检索元素。map同multimap的不同在于是否允许相同key的元素。

上述4中容器通常以平衡二叉树方式实现，插入和检索的时间都是 O(logN)

1.3 容器适配器简介

1）stack ：头文件〈stack〉

栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项。即按照后进先出的原则

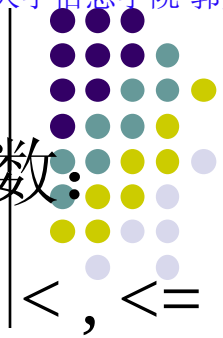
2） queue ：头文件〈queue〉

队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。按照先进先出的原则。

3）priority_queue ：头文件〈queue〉

优先级队列。最高优先级元素总是第一个出列

1.4 容器的共有成员函数：



1) 顺序容器、关联容器和stack,queue有以下成员函数：

相当于按词典顺序比较两个容器的运算符： =, < , <= , > , >=, == , !=

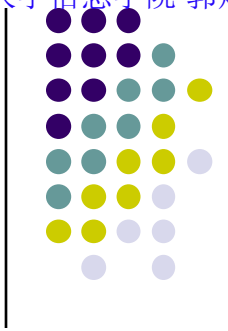
empty : 判断容器中是否有元素

size: 容器中元素个数

除了容器适配器外，其他容器还都支持：

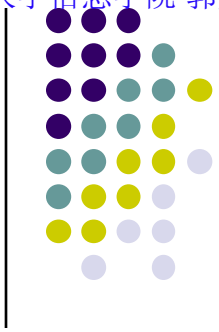
swap: 交换两个容器的内容

max_size: 容器中最多能装多少元素



比较两个容器的例子：

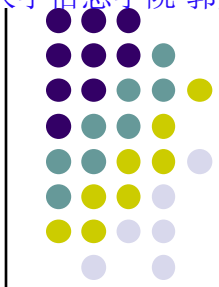
```
#include <vector>
#include <iostream>
class A {
private:
        int n;
public:
        friend bool operator < (const A &,  const  A &);
        A(int n_ ) { n = n_ ; }
};
bool operator < (const A & o1,  const  A & o2)  {
        return o1.n < o2.n;
}
```



```
int main()
{
        std::vector<A> v1;    std::vector<A> v2;
        v1.push_back (A(5)); v1.push_back (A(1));
        v2.push_back (A(1)); v2.push_back (A(2));
        v2.push_back (A(3));
        std::cout << (v1 < v2);
        return 0;
}
```

输出：

0



2）只在第一类容器中的函数：

begin 返回指向容器中第一个元素的迭代器

end 返回指向容器中最后一个元素后面的位置的迭代器

rbegin 返回指向容器中最后一个元素的迭代器

rend 返回指向容器中第一个元素前面的位置的迭代器

erase 从容器中删除一个或几个元素

clear 从容器中删除所有元素

2 迭代器

用于指向第一类容器中的元素。有const 和非 const两种。

通过迭代器可以读取它指向的元素，通过非const迭代器还能修改其指向的元素。迭代器用法和指针类似。

定义一个容器类的迭代器的方法可以是：

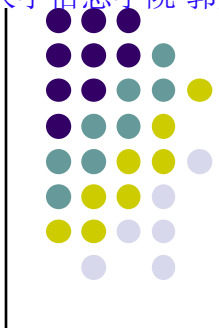
容器类名::iterator 变量名;

或：

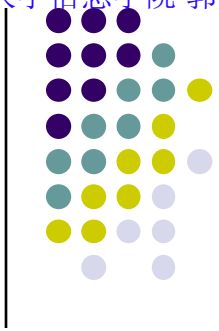
容器类名::const_iterator 变量名;

访问一个迭代器指向的元素：

* 迭代器变量名

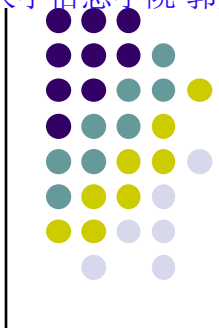


迭代器上可以执行 ++ 操作, 以指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，则迭代器变成past-the-end值。使用一个past-the-end值的迭代器来访问对象是非法的，就好像使用NULL或未初始化的指针一样。



例如：

```
#include <vector>
#include <iostream>
using namespace std;
int main()  {
    vector<int> v; //一个存放int元素的向量，一开始里面没有元素
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    vector<int>::const_iterator i;   //常量迭代器
    for( i = v.begin();i != v.end();i ++ )
        cout << * i << ",";
    cout << endl;
```



```
    vector<int>::reverse_iterator r;  //反向迭代器
    for( r = v.rbegin();r != v.rend();r++ )
        cout << * r << ",";
    cout << endl;
    vector<int>::iterator j;   //非常量迭代器
    for( j = v.begin();j != v.end();j ++ )
        * j =  100;
    for( i = v.begin();i != v.end();i++ )
        cout << * i << ",";
}
```

输出结果：

1,2,3,4,

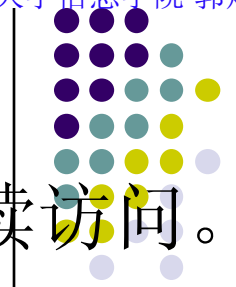
4,3,2,1,

100,100,100,100,

不同容器上支持的迭代器功能强弱有所不同。

容器的迭代器的功能强弱，决定了该容器是否支持STL中的某种算法。

比如，只有第一类容器能用迭代器遍历。排序算法需要通过随机迭代器来访问容器中的原素，那么有的容器就不支持排序算法。



STL 中的迭代器按功能由弱到强分为5种：

1. 输入：Input iterators 提供对数据的只读访问。

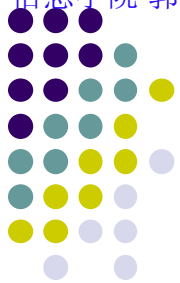
1. 输出：Output iterators 提供对数据的只写访问

2. 正向：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器。

3. 双向：Bidirectional iterators提供读写操作，并能一次一个地向前和向后移动。

4. 随机访问：Random access iterators提供读写操作，并能在数据中随机移动。

编号大的迭代器拥有编号小的迭代器的所有功能，能当作编号小的迭代器使用。



不同迭代器所能进行的操作(功能)：

所有迭代器： ++p, p ++

输入迭代器： * p, p = p1, p == p1 , p!= p1

输出迭代器： * p, p = p1

正向迭代器： 上面全部

双向迭代器： 上面全部，--p, p --,

随机访问迭代器： 上面全部，以及：

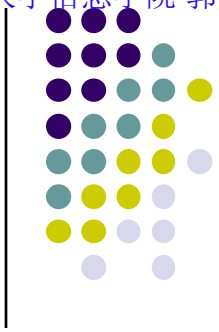
p+= i, p -= i,

p + i： 返回指向 p 后面的第i个元素的迭代器

p - i： 返回指向 p 前面的第i个元素的迭代器

p[i]： p 后面的第i个元素的引用

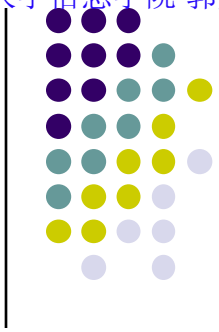
p < p1, p <= p1, p > p1, p>= p1



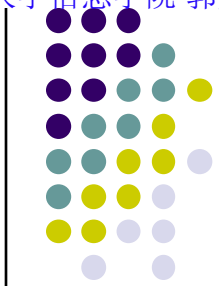
容器	迭代器类别
vector	随机
deque	随机
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

例如，vector的迭代器是随机迭代器，所以遍历 vector 可以有以下几种做法：

```
vector<int> v(100);

vector<int>::value_type i; //等效于写 int i;

for(i = 0;i < v.size() ; i ++)

      cout << v[i];

vector<int>::const_iterator ii;

for( ii = v.begin(); ii != v.end ();ii ++ )

      cout << * ii;

for( ii = v.begin(); ii < v.end ();ii ++ )

      cout << * ii;
```



```
//间隔一个输出：
ii = v.begin();
while( ii < v.end()) {
        cout << * ii;
        ii = ii + 2;
}
```



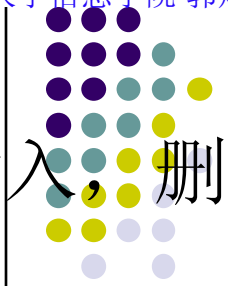
而 list 的迭代器是双向迭代器，所以以下代码可以：

```
list<int> v;
list<int>::const_iterator ii;
for( ii = v.begin(); ii != v.end ();ii ++ )
        cout << * ii;
```

以下代码则不行：

```
for( ii = v.begin(); ii < v.end ();ii ++ )
        cout << * ii;
//双向迭代器不支持 <
for(int i = 0;i < v.size() ; i ++)
        cout << v[i];
```

3 算法简介



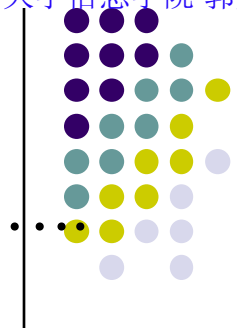
STL中提供能在各种容器中通用的算法，比如插入，删除，查找，排序等。大约有70种标准算法。

算法就是一个个函数模板。

算法通过迭代器来操纵容器中的元素。许多算法需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器。比如，排序和查找

有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器。

算法可以处理容器，也可以处理C语言的数组



1） 变化序列算法：

copy ,remove,fill,replace,random_shuffle,swap, ……

会改变容器

2) 非变化序列算法：

adjacent-find, equal, mismatch,find ,count, search, count_if, for_each, search_n

以上函数模板都在<algorithm> 中定义

此外还有其他算法，比如<numeric>中的算法

算法示例：find()

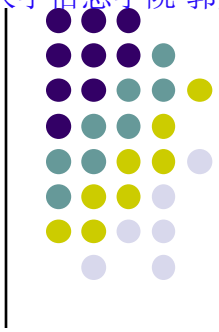
```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点。

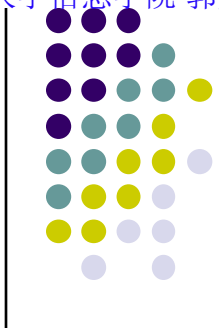
这个区间是个左闭右开的区间，即区间的起点是位于查找范围之中的，而终点不是

val参数是要查找的元素的值。用 == 判断相等

函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。如果找不到，则该迭代器指向查找区间终点。



```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()  {
        int array[10] = {10,20,30,40};
        vector<int> v;
        v.push_back(1);         v.push_back(2);
        v.push_back(3);         v.push_back(4);
        vector<int>::iterator p;
        p = find(v.begin(),v.end(),3);
        if( p != v.end())
                cout << * p << endl;
```

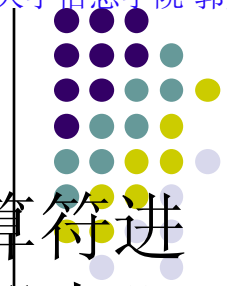


```
        p = find(v.begin(),v.end(),9);
        if( p == v.end())
                cout << "not found " << endl;
        p = find(v.begin()+1,v.end()-2,1);
        if( p != v.end())
                cout << * p << endl;
        int * pp = find( array,array+4,20);
        cout << * pp << endl;
}
```

输出：

```
3
not found
3
20
```

4 STL中“大”“小”“相等”的概念

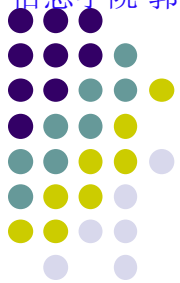


在STL中，缺省的情况下，比较大小是用“<”运算符进行的，和“>”运算符无关。在STL中提到“大”“小”的概念时，以下三个说法是等价的：

1) x 比y 小
2) 表达式“x<y”为真
3) y比x大

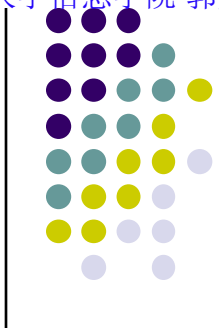
与“>”无关，“>”可以没定义

4 STL中“大”“小”“相等”的概念



在STL中，“x和y相等”也往往不等价于“x==y为真”。

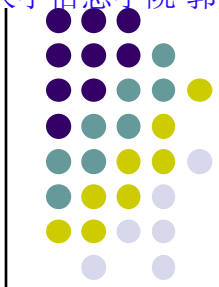
对于在未排序的区间上进行的算法，比如顺序查找find，查找过程中比较两个元素是否相等，用的是“==”运算符；但是对于在排好序的区间上进行查找、合并等操作的算法(比如折半查找算法binary_search, 关联容器自身的成员函数find)来说，“x和y相等”，是与“x<y和y<x同时为假”等价的，与“==”运算符无关。



```
#include <iostream>
#include <algorithm>
using namespace std;
class A
{
            int v;
           public:
            A(int n):v(n) { }
            bool operator < ( const A & a2) const  {return false;}   };

int main()
{
 A a [] = { A(1),A(2),A(3),A(4) };
 cout << binary_search(a,a+4,A(9));
 return 0;
}
```

输出： 1

4 顺序容器



除前述共同操作外，顺序容器还有以下共同操作：

front() :返回容器中第一个元素的引用

back() : 返回容器中最后一个元素的引用

push_back(): 在容器末尾增加新元素

pop_back(): 删除容器末尾的元素

比如，查 list::front 的help,得到的定义是：

```
reference front();
const_reference front() const;
```

list有两个front函数

reference　　和 const_reference 是typedef的类型，

对于 list<double> ，

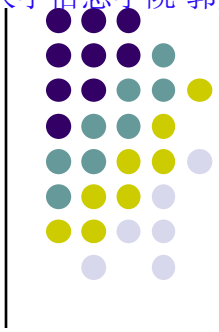
list<double>::refrence 实际上就是 double &

list<double>::const_refreence 实际上就是 const double &

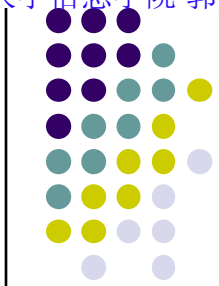
对于 list<int> ，

list<int>::refrence 实际上就是 int &

list<int>::const_refreence 实际上就是 const int &



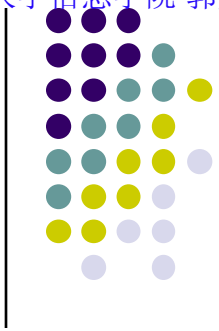
```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
template <class T>
class A
{
      public:
      T a;
      typedef T &refrence;
};
int main()
{
    A<int>::refrence n;
    n = 5;
    cout << n;
    return 0;
}
```



4.1 vector

支持随机访问迭代器，所有STL算法都能对vector操作。

随机访问时间为常数。在尾部添加速度很快，在中间插入慢。实际上就是动态数组。

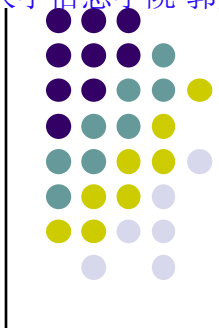


例1：

```
int main()  {    int i;
        int a[5] = {1,2,3,4,5 };    vector<int>  v(5);
        cout << v.end()  - v.begin() << endl;
        for( i = 0;i < v.size();i ++ )  v[i] = i;
        v.at(4) = 100;
        for( i = 0;i < v.size();i ++ )
                cout << v[i] << "," ;
        cout << endl;
        vector<int> v2(a,a+5);  //构造函数
        v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置插入 13
        for( i = 0;i < v2.size();i ++ )
                cout << v2[i] << "," ;    }
```

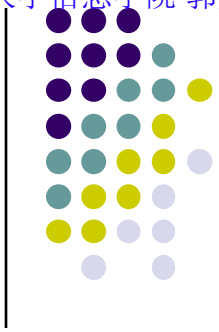
输出：

```
5
0,1,2,3,100,
1,2,13,3,4,5,
```

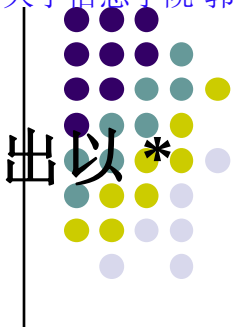


例2：

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <iterator>

int main()  {
        const int SIZE = 5;
        int a[SIZE] = {1,2,3,4,5 };
        vector<int> v (a,a+5);  //构造函数
        try {
                v.at(100) = 7;
        }
        catch( out_of_range e) {
                cout << e.what() << endl;
        }
        cout << v.front() << “,” << v.back() << endl;
```



```
        v.erase(v.begin());
        ostream_iterator<int> output(cout ,“*");
        copy (v.begin(),v.end(),output);
        v.erase( v.begin(),v.end());  //等效于 v.clear();
        if( v.empty ())
                cout << "empty" << endl;
        v.insert (v.begin(),a,a+SIZE);
        copy (v.begin(),v.end(),output);
} // 输出：
invalid vector<T> subscript
1,5
2*3*4*5*empty
1*2*3*4*5*
```

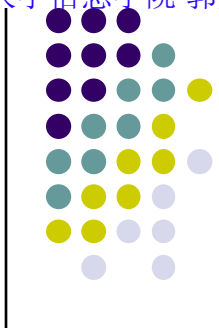


ostream_iterator<int> output(cout ,“*");

定义了一个 ostream_iterator<int> 对象，可以通过cout输出以 * 分隔的一个个整数

copy (v.begin(),v.end(),output);

导致v的内容在 cout上输出



copy 函数模板(算法) :

```
template<class InIt, class OutIt>
OutIt copy(InIt first, InIt last, OutIt x);
```

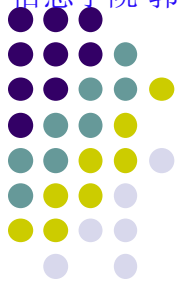
本函数对每个在区间[0, last - first)中的N执行一次 *(x+N) = *(first + N) ， 返回 x + N

对于

copy (v.begin(),v.end(),output);

first 和 last 的类型是 vector<int>::const_iterator

output 的类型是 ostream_iterator<int>

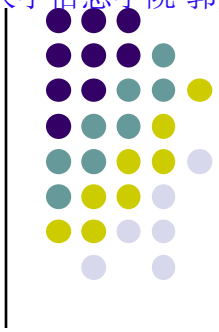


copy 的源代码：

```
template<class _II, class _OI>
 inline _OI copy(_II _F, _II _L, _OI _X)
{
        for (; _F != _L; ++_X, ++_F)
                *_X = *_F;
        return (_X);
}
```

关于 ostream_iterator, istream_iterator的例子

```
int main()  {
        istream_iterator<int> inputInt(cin);
        int n1,n2;
        n1 = * inputInt; //读入 n1
        inputInt ++;
        n2 = * inputInt; //读入 n2
        cout << n1 << "," << n2 << endl;
        ostream_iterator<int> outputInt(cout);
        * outputInt = n1 + n2;          cout << endl;
        int a[5] = { 1,2,3,4,5};
        copy(a,a+5,outputInt);  //输出整个数组
        return 0;  }
```

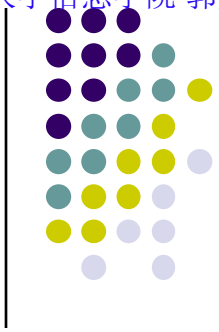


程序运行后输入 78 90敲回车，则输出结果为：

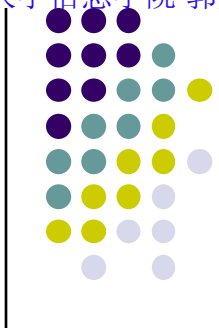
78,90

168

12345



```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
	int a[4] = { 1,2,3,4 };
	My_ostream_iterator<int> oit(cout,"*");
	copy( a,a+4,oit); //输出 1*2*3*4*
	ofstream oFile("test.txt",ios::out);
	My_ostream_iterator<int> oitf(oFile,"*");
	copy(a,a+4,oitf);//向test.txt文件中写入 1*2*3*4*
	oFile.close();
	return 0;
} // 如何编写 My_ostream_iterator?
```

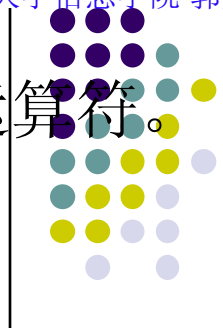


copy 的源代码：

```
template<class _II, class _OI>
 inline _OI copy(_II _F, _II _L, _OI _X)
{
        for (; _F != _L; ++_X, ++_F)
                *_X = *_F;
        return (_X);
}
```

上面程序中调用语句 “copy(a,a+4,oit)” 实例化后得到copy如下：

```
My_ostream_iterator<int> copy(int * _F, int * _L,
   My_ostream_iterator<int> _X)
{
        for (; _F != _L; ++_X, ++_F)
                        *_X = *_F;
        return (_X);
}
```



因此，My_ostream_iterator类应该重载“++” 和“*”运算符。“=”也应该被重载。

```
template<class T>
class My_ostream_iterator
{
 private:
          string sep; //分隔符
          ostream & os;
 public:
         My_ostream_iterator(ostream & o, string s):sep(s),os(o){ }
         void operator ++() { }; // ++只需要有定义即可，不需要做什么
         My_ostream_iterator & operator * ()
         {          return * this; }
          My_ostream_iterator & operator = ( const T & val)
         {         os << val << sep;    return * this;   }
};
```