



面向对象程序设计与实践

主讲：赵培海

上周内容回顾

- ① 虚函数 和 抽象类
- ② 泛型程序设计
- ③ 模板的概述
- ④ 函数模板

基类指针引用派生类指针

- 用基类指针引用一个派生类对象，由于派生类对象也是基类的对象，所以这种引用是安全的。
- 但是基类指针只能引用基类成员。若试图通过基类指针引用那些只在派生类中才有的成员，编译器会报告语法错误。(该问题的答案是虚函数和多态性)。

抽象类

- **抽象类**：带有纯虚函数的类。
- 抽象类的主要**作用**：
 - 通过它为一个类族建立一个公共的接口，使它们能够更有效地发挥多态特性。
- **抽象类**刻画了一组子类的公共操作接口的通用语义，这些接口的语义也传给子类。
- 一般而言：**抽象类只描述这组子类共同操作接口，而完整的实现留给子类。**

泛型程序设计

➤ C++语言的核心优势之一：便于软件的重用。

➤ C++有两个方面体现重用：

- 面向对象的思想：继承和多态，标准类库
- 泛型程序设计(generic programming)的思想：
 - 模板机制
 - STL(标准模板库)

泛型程序设计

➤ 泛型程序设计：使用模板的程序设计方法。

- 将一些常用的数据结构(如链表、二叉树等)和算法(如排序、查找等)写成模板；
- 不论数据结构里放的是什麼对象，算法针对什麼样的对象，都不必要重新实现数据结构，重新编写算法。

➤ STL(标准模板库)：

- 一些常用的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。
- 优点：有了STL，不必再从头编写大多数的标准数据结构和算法，并且可以获得非常高的性能。

示例：函数模板重载

```
#include<iostream>
using namespace std;
int mymax(int a, int b)
{cout<<"Normal mymax called: "<<(b<a? a:b)<<endl;
  return b<a? a:b;}

template<typename T>
T mymax(T a, T b)
{cout<<"Template mymax called: "<<(b<a? a:b)<<endl;
  return b<a? a:b;}
int main( )
{
    mymax(7,42);
    mymax(2.0,31.0);
    mymax('a','b');
    mymax<>(7,42);
    mymax('c',41);
    mymax('a',41);}
```

```
Normal mymax called: 42
Template mymax called: 31
Template mymax called: b
Template mymax called: 42
Normal mymax called: 99
Normal mymax called: 97
Program ended with exit code: 0
```

面向对象程序设计

模板 PART2

主要内容

① 类模板

类模板

- **类模板**：在声明一个类时，能够将实现这个类所需要的某些数据类型(包括类中**数据成员的类型**、**成员函数的参数的类型**或**其返回值的类型**)参数化，使之成为一个可以处理多种类型数据的**通用类**。
- 如果一个类中数据成员的数据类型不能确定，或者是某个成员函数的参数或返回值的类型不能确定，就必须将此类声明为**模板**，它的存在不是代表一个具体的、实际的类，而是代表着一类类。

类模板的声明格式

声明格式：

`template` < 模板参数表 >

`class` < 类名 >;



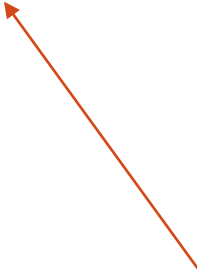
`template`是定义类模板的关键字。

类模板的声明格式

声明格式：

template < 模板参数表 >

class < 类名 >;



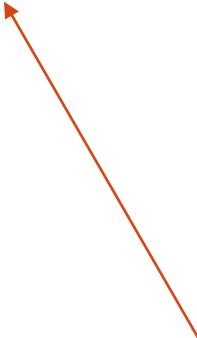
template后面是用逗号分隔的模板参数表，用一对尖括号括起来。该列表是模板参数表，不能为空。

类模板的声明格式

声明格式：

template < 模板参数表 >

class < 类名 >;



模板参数可以是一个模板类型参数，代表一种类型；也可以是一个模板非类型参数，代表一个常量表达式。

类模板的声明格式

声明格式：

template < 模板参数表 >

class < 类名 >;

例：

```
template <class T>  
class QueueItem;
```

模板类型参数由关键字 `class` 或 `typename` 后加一个标识符构成。在类的模板参数表中，这两个关键字的意义相同。它们表示后面的参数名代表一个潜在的内置或用户定义的类型。模板参数名由程序员选择。

类模板的声明格式

声明格式：

```
template < 模板参数表 >
```

```
class < 类名 >;
```

例：

```
template <class T>  
class QueueItem;
```

当模板被实例化时，实际的内置或用户定义类型将替换模板的类型参数。类型int、double、char*等都是有效的模板实参类型。

类模板的声明格式

声明格式：

```
template < 模板参数表 >
```

```
class < 类名 >;
```

模板非类型参数由一个普通的参数声明构成。模板非类型参数表示该参数名代表了一个潜在的值，而该值代表了模板定义中的一个常量。

例：

```
template <class Type, int size>  
class Buffer;
```


类模板的声明格式


声明格式：

```
template < 模板参数表 >
```

```
class < 类名 >;
```

例：

```
template <class Type, int size>  
class Buffer;
```



一个 Buffer 类模板可以有一个类型参数来表示它所包含的元素类型，和一个非类型参数来表示其大小的常量值。

类模板的声明

➤ 一个类模板可以有多个类型参数：

◦ `template < class T1, class T2, class T3 >`

`class < 类名 >;`

➤ 一旦声明了类型参数，那么在类模板定义的余下部分中，它就可以被用作类型指示符。它在类模板中的使用方式与内置的或用户定义的类型在非模板类定义中的用法一样。

➤ 例如：类型参数可以被用来声明数据成员、成员函数和嵌套类的成员等等。

类模板的声明

➤ 一个类模板可以有多个类型参数：

◦ `template < class T1, class T2, class T3 >`

`class < 类名 > {`

➤ 每个模板类型参数的前面都必须有关键字 `class` 或 `typename`。定义的余下部分中，它就可以被用作类型指示符。它在类模板中的使用方式与内置的或用户定义的类型在非模板类定义中的用法一样。

➤ 例如：类型参数可以被用来声明数据成员、成员函数和嵌套类的成员等等。

类模板的定义

定义的语法格式如下：

◦ `template < class T>`

`class < 类名 >`

`{`

`类成员声明`

`};`

类成员声明紧跟在模板参数表后面除了模板参数外类模板的定义看起来和非模板类相同。

注意：类模板的定义

- 模板参数的名字在它被声明为模板参数后，一直到模板声明或定义的结束都可以被使用。如果在全局域中声明了与模板参数同名的变量，则该变量被隐藏掉。

```
typedef double Type;

template <class Type>
class QueueItem {
public:
    // ...
private:
    Type item;
    QueueItem *next;
};
```

item 的类型不是double，
它的类型是模板参数的
类型。

注意：类模板的定义

- 模板参数名不能被用作在类模板定义中声明的类成员的名字。

```
template <class Type>
class QueueItem {
public:
    // ...
private:
    typedef double Type;
    Type item;
    QueueItem *next;
};
```

错误：成员名不能与模板参数 Type 同名。

注意：类模板的定义

- 模板参数的名字在模板参数表中只能被引入一次。

```
template <class Type, class Type>
class QueueItem {
public:
    // ...
private:
    Type item;
    QueueItem *next;
};
```


编译错误：重复使用名为
Type 的模板参数。

注意：类模板的定义

- 在不同的类模板声明或定义之间，模板参数的名字可以被重复使用。

```
template <class Type>  
class QueueItem;
```

```
template <class Type>  
class Queue;
```



正确：名字 ‘Type’ 在不同模板之间可以被重复使用。

注意：类模板的定义

- 在类模板的前向声明和类模板定义中，模板参数的名字可以不同。

```
// 所有三个 QueueItem 声明都引用同一个类模板
```

```
// 模板的声明
```

```
template <class T> class QueueItem;
```

```
template <class U> class QueueItem;
```

```
// 模板的真正定义
```

```
template <class Type>
```

```
class QueueItem { ... };
```

类模板的成员函数

- 在类定义体外定义成员函数时，若此成员函数中有模板参数存在，则除了需要和一般类的体外定义成员函数一样的定义外，还需在函数体外进行模板声明：

`template < class T >`

类型名类名<T>::成员函数名(形参表)

例：

```
template<class T>
void Test<T>::print( ){
    cout<<"n="<<n<<endl;
    cout<<"i="<<i<<endl;
    cout<<"cnt="<<cnt<<endl;
}
```

总结：类模板的定义

- 定义类模板时使用关键字template。
- 定义类模板时至少要确定一个模板参数，多个模板参数用逗号分隔。
- 类模板中的成员函数可以是函数模板。在类体外定义函数模板时，应在模板名前用类模板名限定(<类模板名>::)来表示该函数模板的所属。

示例：类模板的定义

➤ 定义一个模板类Student：

- **学号**：设计成参数化类型，可以实例化成字符串、整型等；
- **成绩**：设计成参数化类型，可以实例化成整型、浮点型、字符型(用来表示等级分)等。

示例：类模板的定义

```
// TN0, TScore 为参数化类型
template <class TN0, class TScore, int num>
class Student
{
private:
    TN0 StudentID[num];           //参数化类型数组，存储姓名
    TScore score[num];           //参数化类型数组，存储分数
public:
    TN0 TopStudent( );
    int BelowNum(TScore ascore);
    void sort( );
};
```

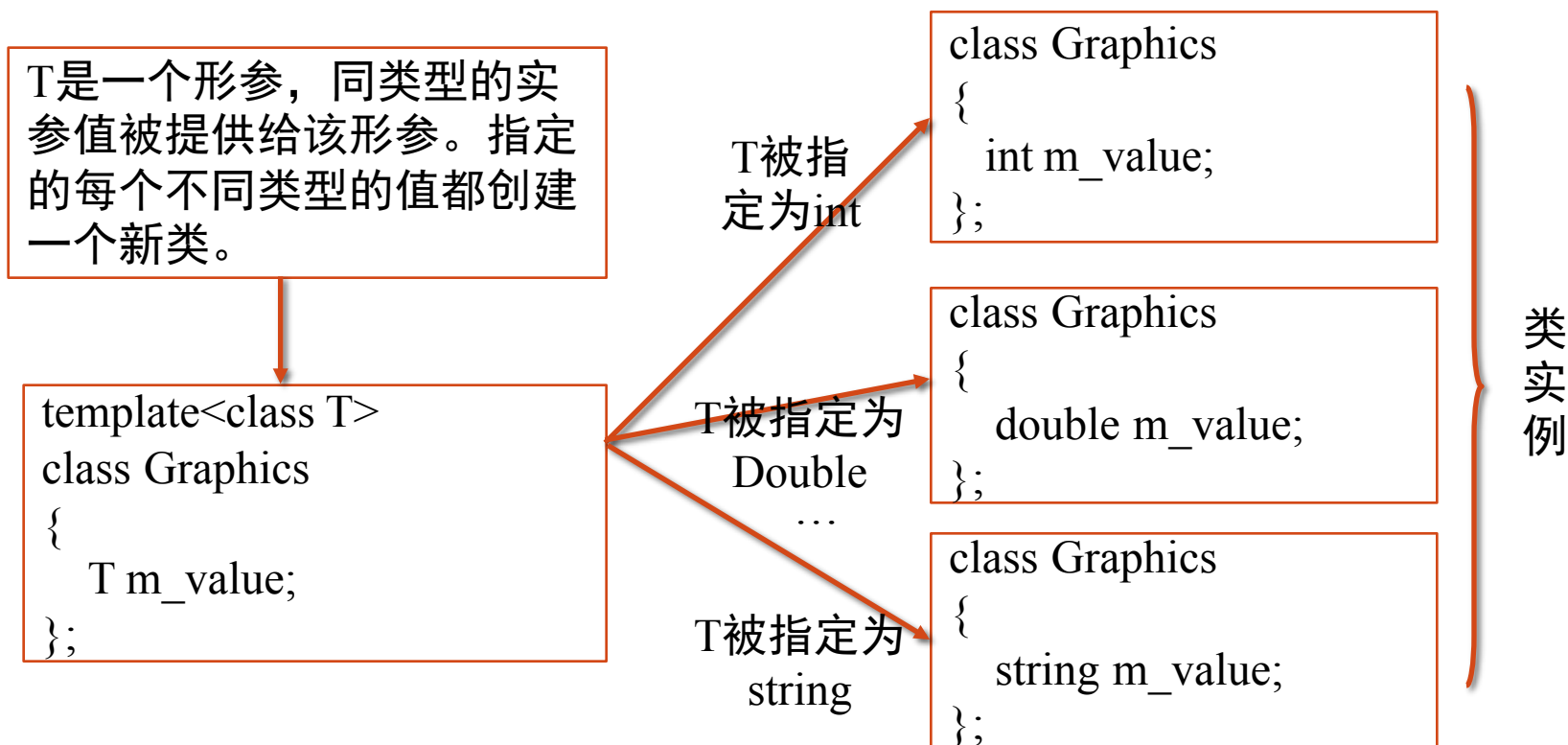
示例：类模板的定义

模板类Student的成员函数在类外实现如下：

```
template <class TN0, class TScore, int num>
int Student<TN0, TScore, num>::BelowNum(TScore ascore)
{
    return 0;
}
template <class TN0, class TScore, int num>
void Student<TN0, TScore, num>::sort( )
{
}
template <class TN0, class TScore, int num>
TN0 Student<TN0, TScore, num>::TopStudent( )
{
    return StudentID[0];
}
```

类模板的实例化

➤ **类模板实例化**：从通用的类模板定义中生成类(模板类)的过程。



类模板的实例化

- 一个类模板是具体类的抽象，在使用类模板建立对象时，才根据给定的模板参数值实例化(专门化)成具体的类，然后由类建立对象。
- 与函数模板不同，类模板实例化只能采用显式方式：
类模板名 <模板参数值表> 对象1, 对象2, ..., 对象n;

类模板的实例化

```
template <class TN0, class
TScore, int num>
class Student
{
private:
    TN0 StudentID[num];
    TScore score[num];
public:
    TN0 TopStudent( );
    int BelowNum(TScore
ascore);
    void sort( );
};
```

将类Student实例化成：

```
class Student
{
private:
    String StudentID[100];
    float score[100];
public:
    String TopStudent( );
    int BelowNum(float
ascore);
    void sort( );
};
```

默认模板参数

➤ 类模板的实例化过程与函数调用的实参与形参结合的过程相似，函数的形参可以采用默认值，类模板的类型参数也可以采用默认值，这样避免每次实例化时都显式给出实参。

类模板的实例化

```
template <class TNO, class
TScore=int, int num=10>
class Student
{
private:
    TNO StudentID[num];
    TScore score[num];
public:
    TNO TopStudent( );
    int BelowNum(TScore
ascore);
    void sort( );
};
```

TScore, num分别给出默认值int, 10。

用以下方式实例化：

```
Student<char *> S1;
```

```
class Student
{
private:
    char * StudentID[10];
    int score[10];
public:
    char * TopStudent( );
    int BelowNum(int ascore);
    void sort( );
};
```

实例：类模板的定义及实例化

➤教材例8-2：将栈设计成一个类模板，在栈中存放任意的数据。

➤分析：

- 使用：动态分配数组；
- 使用：指针top指向栈顶元素；
- 使用：成员函数push()、pop()、IsEmpty()、IsFull()分别进行压栈、出栈、判空、判满。

➤Project组成：TStack、Main

程序名： TStack 功能： 栈类模板

```
#include <iostream>
using namespace std;
template<class T>
class Stack{
private:
    int size, top;
    T *space;
public:
    Stack(int = 10);
    ~Stack( ){
        delete [ ] space;
    }
    bool push(const T&);
    T pop( );
    bool IsEmpty( ) const{
        return top == size;
    }
    bool IsFull( ) const{
        return top == 0;
    }
};
```

```
template<class T>
Stack <T>::Stack(int size){
    this->size = size;
    space = new T[size];
    top = size;
}
```

```
template <class T>
bool Stack <T>::push(const T&
element)
{
    if( ! IsFull( ) ){
        space[--top] = element;
        return true;
    }
    return false;
}

template <class T>
T Stack <T>::pop( ){
    return space[top++];
}
```

程序名：Main 功能：使用栈类模板

利用上面定义的Stack类模板，使用int型堆栈：

```
#include <iostream>
using namespace std;
int main( ){
    Stack <int> intStack(5);
    int i = 1;
    cout<< "Pushing elements onto intStack \n ";
    while (intStack.push(i)){
        cout<<i<<" " ;
        i++;
    }
    cout<<"\n Stack is full.\n ";
    while (!intStack.IsEmpty())
        cout <<intStack.pop()<< " ";
    cout<<"\n Stack is empty. Cannot pop \n";
    return 0;
}
```

程序名：Main 功能：使用栈类模板

利用上面定义的Stack类模板，使用int型堆栈：

```
#include <iostream>
using namespace std;
int main( ){
    Stack <int> intStack(5);
    int i = 1;
    cout<< "Pushing elements onto intStack \n ";
    while (intStack.push(i)){
        cout<<i<<" " ;
        i++;
    }
    cout<<"\n Stack is full\n ";
    while (!intStack.IsEmpty()){
        cout <<intStack.pop()<<" ";
    }
    cout<<"\n Stack is empty\n ";
    return 0;
}
```

```
Pushing elements onto intStack
1 2 3 4 5
Stack is full.
5 4 3 2 1
Stack is empty. Cannot pop
Program ended with exit code: 0
```

模板对象作为函数模板的形参

- 类模板的实例化对象模板类同普通类的使用相同，可以从模板类中实例化模板对象进行操作。
- 模板对象还可以作为函数模板的参数进行使用。
- 下面的实例利用模板对象作为函数模板的形参，完成和上个实例同样的功能。

程序名： Main.cpp 功能： 使用栈类模板

```
#include <iostream>
using namespace std;
template <class T>
void testStack(Stack<T> & theStack, T value, T
increment, const char* stackName){
    cout<<"\nPushing elements onto "<<stackName<<"\n";
    while(theStack.push(value)){
        cout<<value<<" ";
        value+=increment; }
    cout<<"\nStack is full. Cannot push \n";
    while( ! theStack.IsEmpty( ) )
        cout<<theStack.pop( )<<endl;
    cout<<"\nStack is empty. Cannot pop\n";
}

int main( ){
    Stack <int > intStack (5);
    testStack( intStack, 1, 1, "intStack");
    return 0; }
```

类模板的派生

- 类模板作为一种特殊的类，也可以有自己的派生类。
- 类模板的派生类也有公有派生和私有派生。
- 派生类不能访问基类的私有成员，派生的语法格式也是相似的。

```
template < class Type >
class 基类名 {
    // 基类模板定义
    ...
}
template < class Type >
class 派生类名: public 基类名< Type > {
    // 派生类模板定义
    ...
}
```

类模板的派生

- 类模板作为一种特殊的类，也可以有自己的派生类。
- 类模板的派生类也有公有派生和私有派生。
- 派生类不能访问基类的私有成员，派生的语法格式也是相似的。

```
template < class Type >
```

```
class 基类名 {  
    // 基类模板定义
```

```
    ...
```

```
}
```

```
template < class Type >
```

```
class 派生类名: public 基类名 < Type > {  
    // 派生类模板定义
```

```
    ...
```

```
}
```

派生类模板的参数表中应包含基类模板的参数。

类模板的派生

```
template <class T >
class C
{
public:
    void fun1 (T m ){
        cout<< m << endl ;
    }
};
```

```
template <class T1, class T2>
class D: public C<T2>
{
public:
    void fun2 (T1 n ){
        cout<<n<<endl ;
    }
};
```

如果派生类没有类型参数，或者说它的类型参数与基类的类型参数相同时，派生类的模板参数只要包含基类的模板参数就可以。

类模板的派生

```
template <class T >
class C
{
public:
    void fun1 (T m ){
        cout<< m << endl ;
    }
};
```

```
template <class T1, class T2>
class D: public C<T2>
{
public:
    void fun2 (T1 n ){
        cout<<n<<endl ;
    }
};
```

T1是它的类型参数，T2
是它的基类C的类型参数。

如果派生类没有类型参数，或者说它的类型参数与基类的类型参数相同时，派生类的模板参数只要包含基类的模板参数就可以。

类模板的派生

➤在实例化派生类时，其类型参数也传递给了基类，因而也实例化了基类模板。

➤例：

```
D < char , int > dci;
```

此定义将派生类D实例化成D < char >模板类时，也把基类C实例化成了C < int >模板类。然后可以使用基类和派生类中定义的公有成员函数。

```
dci.fun1(20); //显示整数20，调用基类C<int>::fun1()
```

```
dci.fun2('h'); //显示字符'h',调用派生类D<char>::fun2()
```

类模板的派生

➤在实例化派生类时，其类型参数也传递给了基类，因而也实例化了基类模板。

➤例：

```
template <class T >
class C
{
public:
    void fun1 (T m ){
        cout<< m << endl ;
    }
};
```

```
template <class T1 , class T2>
class D: public C<T2>
{
public:
    void fun2 (T1 n ){
        cout<<n<<endl ;
    }
};
```

dci.fun1(20); //显示整数20, 调用基类C<int>::fun1()

dci.fun2('h'); //显示字符'h',调用派生类D<char>::fun2()

实例：类模板的派生

```
1  #include <iostream>
2  using namespace std;
3  template <class T>
4  class Test{
5  public:
6      T m;
7  };
8  template <class T>
9  class subTest: private Test < T >{
10 public:
11     void set_m( T m ){ this->m = m; }
12     T get_m( ){ return this->m; }
13     void disp( )
14     {cout<<"Show this subTest: "<< this->m <<endl;}
15 };
16 int main( ){
17     subTest <int > A ;
18     A.set_m( 25 );
19     A.disp( );
20     return 0;
21 }
```


实例：类模板的派生

```
1  #include <iostream>
2  using namespace std;
3  template <class T>
4  class Test{
5  public:
6      T m;
7  };
8  template <class T>
9  class subTest: private Test < T >{
10 public:
11     void set_m( T m ){ this->m = m; }
12     T get_m( ){ return this->m; }
13     void disp( )
14     {cout<<"Show this subTest: "<< this->m <<endl;}
15 };
16 int main( ){
17     subTest <int > A ;
18     A.set_m( 25 );
19     A.disp( );
20     return 0;
21 }
```

Show this subTest: 25
Program ended with exit code: 0

面向对象程序设计

STL 编程

主要内容

- ① STL概述
- ② STL容器
- ③ STL迭代器
- ④ STL顺序容器
- ⑤ STL String

STL(Standard Template Library)

- **STL**: 标准模板库, 是一个高效的C++程序库。
- ANSI/ISO C++ **标准函数库**的一个子集,
- 提供了大量可扩展的类模板, 包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。

STL编程的基本设计思想

- 从**逻辑结构**和**存储结构**来看，基本数据结构的数量是有限的。对于其中的数据结构，用户可能需要反复的编写一些类似的代码，只是为了适应不同数据的类型变化而在细节上有所出入。
- 如果能够将这些经典的数据结构，采用**类型参数**的形式，设计为**通用的类模板和函数模板**的形式，允许用户**重复利用已有的数据结构**构造自己特定类型下的、符合实际需要的数据结构，无疑将简化程序开发，提高软件的开发效率。

STL编程的基本设计思想：逻辑层次

- STL中体现了泛型化程序设计的思想，它提倡使用现有的模板程序代码开发应用程序，是一种代码的重用技术(reusability)。
- 代码重用可以提高软件开发人员的劳动生产率和目标系统质量，是软件工程追求的重要目标。许多程序设计语言通过提供标准库来实现代码重用的机制。
- STL是一个通用组件库，它的目标是将常用的数据结构和算法标准化、通用化，这样用户可以直接套用而不用重复开发它们，从而提高程序设计的效率。

STL编程的基本设计思想：实现层次

- **STL**是一种类型参数化(type parameterized)的程序设计方法，是一个基于模板的标准类库，称之为**容器类**。
- 每种容器都是一种**已经建立完成的标准数据结构**。
- 在**容器**中，放入任何类型的数据，很容易建立一个存储该类型(或类)的数据结构。

STL的组成

➤ STL主要由六个部分组成：

- **容器**(container)：包括各种基本数据结构的类模板。
 - STL 容器部分主要由头文件 `<vector>`、`<list>`、`<deque>`、`<set>`、`<map>`、`<stack>`和`<queue>`组成。
- **迭代器**(iterator)：是面向对象版本的指针，如同指针可以指向内存中的一个地址，迭代器可以指向容器中的一个位置。
 - STL的每一个**容器类**模板中，都定义了一组对应的**迭代器类**，用以存取**容器**中的元素。
 - 在STL中，**迭代器**将**算法**和**容器**联系起来：通过**迭代器**，**算法**函数可以访问容器中指定位置的元素，而无需关心元素的具体类型。
 - **迭代器**是通过重载一元的`*`和`->`来从容器中间接地返回一个值。
 - STL**迭代器**部分主要由头文件`<utility>`和`<iterator>`组成。

STL的组成

➤ STL主要由六个部分组成：

- **适配器**(adaptor)：简单地说就是一种**接口类**，专门用来修改现有类的接口，提供一种新的接口；或调用现有的函数来实现所需要的功能。
 - 主要包括3种适配器**Container Adaptors**、**Iterator Adaptors**与**Function Adaptors**。
 - 其中：**迭代器适配器**的定义在头文件<iterator>中，**函数适配器**的定义在头文件<functional>中。
- **算法**(algorithm)：包括各种基本算法，如比较、交换、查找、排序、遍历操作、复制、修改、移除、反转、合并等等。
 - STL**算法**部分主要由头文件<algorithm>和<numeric>组成。

STL的组成

➤ STL主要由六个部分组成：

- **函数对象**(function object)：一种行为类似于函数的class，实现技术上是一个改写了 **call operator()** 的class。
- STL 提供 15 个预定义的 Function objects。头文件<**functional**>中定义了一些类模板，用以声明函数对象。
- **内存适配器**(allocator)：为STL提供空间配置的系统。
- 头文件<**memory**>中的主要部分是模板类**allocator**，它负责产生所有容器中的默认分配器。
- 容器使用allocator完成对内存的操作，allocator提供内存原语以对内存进行统一的存取。

STL六大组件的交互关系

- 容器通过内存适配器取得数据存储空间；
- 算法通过迭代器存取容器的内容；
- 函数对象用在和算法的结合中，协助算法完成不同的策略变化，从而扩展算法的效用；
- 适配器可以对容器和迭代器的接口进行转换、修饰或套接函数对象。

STL的主要特点

- STL最主要的一个特点：**数据结构**和**算法**的分离，
 - **容器**是像链表，向量、栈、队列之类的数据结构，并按类模板方式提供；
 - **算法**是函数模板，用于操作容器中的数据。
- 由于**STL**以模板为基础，所以能用于任何数据类型和结构。
- 实际上，可以认为**STL**是以容器和迭代器为基础的一种泛型算法(Generic Algorithms)库。

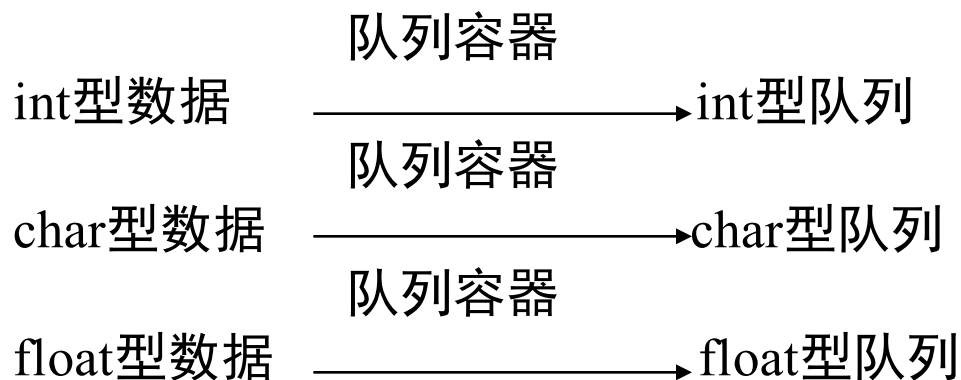
STL的主要特点

- **算法是泛型的**：不与任何特定的数据结构或对象类型系在一起；
- **容器是泛型的**：它可以是数组、向量、链表、集合、映射、队列、栈、字符串等等。**容器**中包含的元素对象可以是任意数据类型，**容器**提供**迭代器**用来定址其所包含的元素；
- **迭代器是泛型的**：泛型的指针，是一种指向其他对象的对象，**迭代器**能够遍历由对象所形成的序列区间(Range)。**迭代器**将**容器**与作用其上的**算法**分离，大多数的**算法**自身并不直接操作于**容器**上，而是操作于**迭代器**所形成的区间中。

STL容器(container)概述

- **STL 容器**：可容纳各种数据类型(基本类型的变量、对象等)的多元素数据结构。
- C++中的**容器**存在一定的**数据加工能力**：如同一个对数据对象进行加工的模具，可以把不同类型的数据放到这个模具中进行加工处理，形成具有一定共同特性的数据结构。

例如：



STL容器分类

➤ STL容器分为两大类：

- **顺序容器**(sequence container)：组织成对象的有限线性集合，所有对象都是同一类型。元素的插入位置同元素的值无关。
- **关联容器**：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**。

STL容器分类

➤ STL容器分为两大类：

- **顺序容器**：组织成对象的有限线性集合，所有对象都是同一类型。
 - STL中三种基本顺序容器是：**向量(vector)**、**线性表(list)**、**双向队列(deque)**。
- **关联容器**：提供了基于Key的数据的快速检索能力。元素被排好序，检索数据时可以二分搜索。
 - STL有四种关联容器：**集合(set)**、**多元集合(multiset)**、**映射(map)**、**多元映射(multimap)**。
 - 当一个Key对应一个Value时，可以使用集合和映射；若对应同一Key有多个元素被存储时，可以使用多元集合和多元映射。

顺序容器简介

- **顺序容器**：以**逻辑线性排列方式**存储一个元素序列，在这些容器类型中的对象在**逻辑**上被认为是在**连续的存储空间中存储**的。
- 在**顺序容器**中的对象有**相对于容器的逻辑位置**，如在容器的起始、末尾等。
- **顺序容器**可以用于**存储线性表类型的数据结构**。

顺序容器分类

➤ 顺序容器分为三大类：

- **向量**(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。
- **线性表**(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。
- **双向队列**(deque)：随机存取任何元素都能在常数时间完成(但性能次于vector)。在两端增删元素具有较佳的性能。

顺序容器分类

容器类名	特性	何时使用	头文件
vector (向量)	在内存中 占有一块连续的空间 ，存储一个元素序列。可以看作一个可自动扩充的动态数组，而且提供越界检查。可用[]运算符直接存取数据。	需要快速查找，不在意插入/删除的速度快慢。能使用数组的地方都能使用向量。	<vector>
list (列表)	双向链接列表，每个节点包含一个元素。列表中的每个元素均有指针指向前一个元素和下一个元素。	需要快速的插入/删除，不在意查找的速度慢，就可以使用列表。	<list>
deque (双端队列)	在内存中 不占有一块连续 的空间，介于向量和列表之间，更接近向量，适用于由两端存取数据。可用[]运算符直接存取数据。	可以提供快速的元素存取。在序列中插入/删的速度除较慢。一般不需要使用双端队列，可以转而使用vector或list。	<deque>

关联容器简介

- **关联容器**：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。
- **特点**：提供了基于Key的数据的快速检索能力。元素被排好序，检索数据时可以二分搜索。在查找时具有非常好的性能。

关联容器分类

➤ 关联容器分为四大类：

- 集合(set)/多元集合(multiset): 头文件 <set>
 - 集合set中不允许存在相同元素,
 - multiset中允许存在相同的元素。
- 映射(map)/多元映射(multimap): 头文件 <map>
 - map与set的不同在于map中存放的是成对的key/value。
 - 根据key对元素进行排序,可快速地根据key来检索元素。

➤ 当一个Key对应一个Value时,可以使用集合和映射;若对应同一Key有多个元素被存储时,可以使用多元集合和多元映射。

➤ 关联容器: 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。

容器适配器简介

➤ **栈**(stack): 头文件 <stack>

- 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。

➤ **队列**(queue): 头文件 <queue>

- 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。

➤ **优先级队列**(priority_queue): 头文件 <queue>

- 最高优先级元素总是第一个出列。

STL容器类

标准库容器类	说明
顺序容器 vector(矢量) list(列表) deque(双端队列)	从后面快速插入与删除，直接访问任何元素 从任何地方快速插入与删除，双链表 从前或后面快速插入与删除，直接访问任何元素
关联容器 set(集合) multiset(多重集合) map(映射) multimap(多重映射)	快速查找，不允许重复值 快速查找，允许重复值 一对一映射，基于关键字快速查找，不许重复值 一对多映射，基于关键字快速查找，允许重复值
容器适配器 stack(栈) queue(队列) priority_queue	后进先出(LIFO) 先进先出(FIFO) 最高优先级元素总是第一个出列

STL容器的成员函数

➤所有STL容器的共有成员函数：

- 相当于按词典顺序比较两个容器大小的运算符：

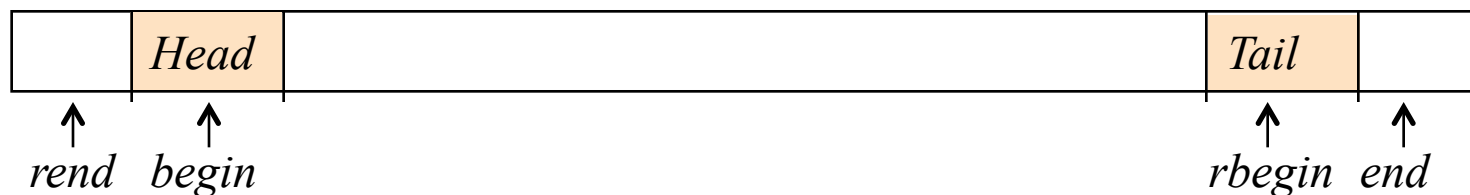
=, <, <=, >, >=, ==, !=

- **empty**：判断容器中是否有元素
- **max_size**：容器中最多能装多少元素
- **size**：容器中元素个数
- **swap**：交换两个容器的内容

STL容器的成员函数

➤ 顺序容器和关联容器的共有成员函数：

- **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。



STL迭代器

➤ 定义：

- 提供一种方法访问一个容器(container)对象中各个元素，而又**不需暴露该对象的内部细节**。
- 用于指向顺序容器和关联容器中的元素。有const 和非 const两种。

➤ 迭代器用法和指针类似。

- 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。

➤ 定义一个容器类的迭代器的方法：

容器类名::iterator 变量名;

或：

容器类名::const_iterator 变量名;

STL迭代器

➤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。

STL迭代器的功能

➤ 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。

STL迭代器的功能

➤ 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。

STL迭代器的功能

STL迭代器类型	说明
输入 InputIterator	从容器中读取元素。输入迭代器只能一次一个元素地向前移动(即从容器开头到容器末尾)。要重读必须从头开始。
输出 OutputIterator	向容器写入元素。 输出迭代器只能一次一个元素地向前移动。 输出迭代子要重写，必须从头开始。
正向 ForwardIterator	组合输入迭代器和输出迭代器的功能，并保留在容器中的位置(作为状态信息)，所以重新读写不必从头开始。
双向 BidirectionalIterator	组合正向迭代器功能与逆向移动功能(即从容器序列末尾到容器序列开头)
随机访问 RandomAccessIterator	组合双向迭代器的功能，并能直接访问容器中的任意元素，即可向前或向后调任意个元素。

不同迭代器所能完成的功能

- 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
 - $p < p1, p \leq p1, p > p1, p \geq p1$

容器所支持的迭代器类别

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：容器所支持的迭代器类别

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

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

for(int i = 0; i < v.size( ); i ++ )
    cout << v[i];

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

例：容器所支持的迭代器类别

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];           //双向迭代器不支持 []
```

STL的三种迭代器

- 除了标准的迭代器外，STL中还有三种迭代器：
 - `reverse_iterator`：如果想用向后的方向而不是向前的方向的迭代器来遍历除vector之外的容器中的元素，可以使用 `reverse_iterator` 来反转遍历的方向，也可以用 `rbegin()` 来代替 `begin()`，用 `rend()` 代替 `end()`，而此时的 `++` 操作符会朝向后的方向遍历。
 - `const_iterator`：一个向前方向的迭代器，它返回一个常数值。可以使用这种类型的游标来指向一个只读的值。
 - `const_reverse_iterator`：一个朝反方向遍历的迭代器，它返回一个常数值。

示例： STL迭代器

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main( ) {
5      //一个存放int元素的向量，一开始里面没有元素
6      vector<int> v;
7      v.push_back(1); v.push_back(2);
8      v.push_back(3); v.push_back(4);
9      //常量迭代器
10     vector<int>::const_iterator i;
11     for( i = v.begin( ); i != v.end( ); i ++ )
12         cout << * i << ",";
13     cout << endl;
14     //反向迭代器
15     vector<int>::reverse_iterator r;
16     for( r = v.rbegin( ); r != v.rend( ); r++ )
17         cout << * r << ",";
18     cout << endl;
19     //非常量迭代器
20     vector<int>::iterator j;
21     for( j = v.begin( ); j != v.end( ); j ++ )
22         * j = 100;
23     for( i = v.begin( ); i != v.end( ); i++ )
24         cout << * i << ",";
25 }
```

示例： STL迭代器

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main( ) {
5      //一个存放int元素的向量，一开始里面没有元素
6      vector<int> v;
7      v.push_back(1); v.push_back(2);
8      v.push_back(3); v.push_back(4);
9      //常量迭代器
10     vector<int>::const_iterator i;
11     for( i = v.begin( ); i != v.end( ); i ++ )
12         cout << * i << ",";
13     cout << endl;
14     //反向迭代器
15     vector<int>::reverse_iterator r;
16     for( r = v.rbegin( ); r != v.rend( ); r++ )
17         cout << * r << ",";
18     cout << endl;
19     //非常量迭代器
20     vector<int>::iterator j;
21     for( j = v.begin( ); j != v.end( ); j ++ )
22         * j = 100;
23     for( i = v.begin( ); i != v.end( ); i++ )
24         cout << * i << ",";
25 }
```

1,2,3,4,
4,3,2,1,
100,100,100,100,

示例： STL迭代器

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main( ) {
5     //一个存放int元素的向量，一开始里面没有元素
6     vector<int> v;
7     v.push_back(1);
8     v.push_back(2);
9     v.push_back(3);
10    v.push_back(4);
11
12    const vector<int>::iterator i;
13    i++;
14
15 }
```

! Cannot increment value of type 'const vector<int>::iterator' (aka 'const __wrap_iter<pointer>')

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main( ) {
5     //一个存放int元素的向量，一开始里面没有元素
6     vector<int> v;
7     v.push_back(1);
8     v.push_back(2);
9     v.push_back(3);
10    v.push_back(4);
11
12    const vector<int>::iterator i=v.begin();
13    cout<<*i<<endl;
14
15 }
```

1
Program ended with exit code: 0

示例： STL迭代器

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main( ) {
5     //一个存放int元素的向量，一开始里面没有元素
6     vector<int> v;
7     v.push_back(1);
8     v.push_back(2);
9     v.push_back(3);
10    v.push_back(4);
11
12    const vector<int>::iterator i=v.begin();
13    cout<<*i<<endl;
14    *i = 100;
15    cout<<*i<<endl;
16 }
```

```
1
100
Program ended with exit code: 0
```

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main( ) {
5     //一个存放int元素的向量，一开始里面没有元素
6     vector<int> v;
7     v.push_back(1);
8     v.push_back(2);
9     v.push_back(3);
10    v.push_back(4);
11
12    vector<int>::const_iterator i=v.begin();
13    *i = 100;
14    cout<<*i<<endl;
15 }
```

! Read-only variable is not assignable

向量容器(vector)

- **向量容器**：线性顺序结构，相当于数组。
- 用于容纳不定长线性序列的容器，提供对序列的**快速随机访问** (也称**直接访问**)。
 - 其大小可以不预先指定，并且可**自动扩展**。
 - 数据元素的存储采用**连续存储空间**，当需要增加数据元素时，直接从vector容器**尾端**插入，如果vector容器发现此时存储空间不足，整个容器中的数据元素将会被搬到一个新的、更大的内存空间中，并将所有数据元素复制到新的内存空间中。

向量容器与数组的比较

➤ 向量容器的数据安排以及操作方式，与C++中的数组非常相似。

- 其中的元素在内存是连续存储的，可以直接访问。

➤ 区别在于：

- C++中的数组是固定大小的，分配给C++数组的空间数量在数组的生存期内是不会改变的。
- 向量是动态结构，它的大小不固定，可以在程序运行时增加或减少。

向量容器(vector)

➤ vector重新分配空间的过程：

- 首先，vector会申请一块更大的内存块；
- 然后，将原来的数据拷贝到新的内存块中；
- 其次，销毁掉原内存块中的对象(调用对象的析构函数)；
- 最后，将原来的内存空间释放掉。

➤ 结论：如果需要快速的存取元素，但不打算经常增加或删除元素，应当在程序中使用vector容器。

vector的特点

- 指定一块如同数组一样的连续存储空间：空间可以动态扩展，即它可以像数组一样操作，并且可以进行动态操作；
- 随机访问方便：像数组一样被访问；
- 节省空间：因为是连续存储，在存储数据的区域都是没有被浪费的。但是要明确一点vector 大多情况下并不是满存的，在未存储的区域实际是浪费的；

vector的特点

- 在内部进行插入、删除操作效率非常低：这样的操作基本上是被禁止的。vector 被设计成只能在后端进行追加和删除操作，其原因是vector 内部的实现是按照顺序表的原理。
- 要vector 达到最优的性能，最好在创建vector 时就指定其空间大小：当动态添加的数据超过vector 默认分配的大小时要进行内存的重新分配、拷贝与释放，这个操作非常消耗性能。

vector的使用

- 包含 **vector** 类的头文件是 `<vector>`。如果要在程序里使用向量容器，就要在程序中包含下面语句：

```
#include <vector>
```

- 在定义 **vector** 类型对象时，必须指定该对象的类型。

```
//将intVec声明为一个元素类型为int的向量容器对象。
```

```
vector <int>    intVec;
```

```
//将stringVec声明为一个元素类型为string的向量容器对象。
```

```
vector <string>    stringVec;
```

vector较为重要的成员函数

函数名	功能说明
push_back	在容器后端增加元素
pop_back	在容器后端删除元素
insert	在容器中间插入元素
erase	删除容器中间的元素
clear	清除容器内的元素
front	返回容器前端元素的引用
back	返回容器末端元素的引用
begin	返回容器前端的迭代器
end	返回容器末端的迭代器
rbegin	返回容器前端的倒转迭代器
rend	返回容器末端的倒转迭代器
max_size	返回容器可存储元素的最大个数
size	返回当前容器中的元素个数
empty	若容器为空（无元素）则返回true，否则返回false
capacity	返回当前容器可以存储的最大元素个数
at(n)	返回第n个元素的引用
swap(x)	与容器x（vector容器）互换元素
operator[]	利用[]运算符取出容器中的元素

vector的定义和初始化

➤ **vector类**有4种构造函数：

第一种构造函数：

```
//默认构造函数，构造一个空的vector，其大小为零  
vector();
```

例如：

```
//使用默认构造函数创建一个没有任何元素的空向  
//量vecList。vecList.size()=0。  
vector <elementType> vecList;
```

vector的定义和初始化

第二种构造函数：

```
//构造一个初始放入n个值为value的元素的vector。第1个参数  
//为vector初始化的大小，第2个参数是vector中每个对象的初  
//始值，默认为T()构造的对象
```

```
vector<size_type n, const T& value=T( )>;
```

例如：

```
//创建一个大小为size的向量vecList，并使用elementType类  
//的默认构造函数初始化该向量。
```

```
vector<elementType> vecList(size);
```

```
//创建一个大小为n的向量vecList，该向量中所有的n个元素  
//都初始化为elem
```

```
vector<elementType> vecList(n, elem);
```


vector的定义和初始化

第三种构造函数：

//复制构造函数，用另一个向量x来初始化此向量

```
vector(const vector& x);
```

例如：

//创建一个向量vecList，并使用向量otherVecList中

//的元素初始化该向量。向量vecList与向量

//otherVecList的类型相同

```
vector <elementType> vecList(otherVecList);
```

vector的定义和初始化

第四种构造函数：

```
//从另一个支持const_iterator的容器中选取一部分  
//([start,end)区间的元素)来构造一个新的vector  
vector(const_iterator start , const_iterator end );
```

例如：

```
//创建一个向量vecList，并初始化该向量[begin,end)  
//中的元素，即从begin到end-1之间的所有元素  
vector <elementType> vecList(begin,end);
```

示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector <int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector <int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector <int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector <int> v4(v1.begin( ), v1.end( ));
16     vector <int> v5(&num[0],&num[10]);
17
18     // 将向量v2中的10个元素值均修改为5
19     for (i=0;i<10;i++)
20         v2[i]=5;
21     //五个vector对象是相等的
22     //可用operator==来判断
23     //v1==v2
24     if (v1==v2) cout<<"v1==v2"<<endl;
25     else cout<<"v1!=v2"<<endl;
26     //v1==v3
27     if (v1==v3) cout<<"v1==v3"<<endl;
28     else cout<<"v1!=v3"<<endl;
29     //v1==v4
30     if (v1==v4) cout<<"v1==v4"<<endl;
31     else cout<<"v1!=v4"<<endl;
32     //v1==v5
33     if (v1==v5) cout<<"v1==v5"<<endl;
34     else cout<<"v1!=v5"<<endl;
35     return 0;
}
```

示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector<int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector<int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector<int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector<int> v4(v1.begin( ), v1.end( ));
16     vector<int> v5(&num[0],&num[10]);
17
18     // 将向量v2中的10个元素值均修改为5
19     for (i=0;i<10;i++)
20         v2[i]=5;
21     //五个vector对象是相等的
22     //可用operator==来判断
23     //v1==v2
24     if (v1==v2) cout<<"v1==v2"<<endl;
25     else cout<<"v1!=v2"<<endl;
26     //v1==v3
27     if (v1==v3) cout<<"v1==v3"<<endl;
28     else cout<<"v1!=v3"<<endl;
29     //v1==v4
30     if (v1==v4) cout<<"v1==v4"<<endl;
31     else cout<<"v1!=v4"<<endl;
32     //v1==v5
33     if (v1==v5) cout<<"v1==v5"<<endl;
34     else cout<<"v1!=v5"<<endl;
35     return 0;
}
```

示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector <int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector <int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector <int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector <int> v4(v1.begin( ), v1.end( ));
16     vector <int> v5(&num[0],&num[10]);
17
18     // 将向量v2中的10个元素值均修改为5
19     for (i=0;i<10;i++)
20         v2[i]=5;
21     //五个vector对象是相等的
22     //可用operator==来判断
23     //v1==v2
24     if (v1==v2) cout<<"v1==v2"<<endl;
25     else cout<<"v1!=v2"<<endl;
26     //v1==v3
27     if (v1==v3) cout<<"v1==v3"<<endl;
28     else cout<<"v1!=v3"<<endl;
29     //v1==v4
30     if (v1==v4) cout<<"v1==v4"<<endl;
31     else cout<<"v1!=v4"<<endl;
32     //v1==v5
33     if (v1==v5) cout<<"v1==v5"<<endl;
34     else cout<<"v1!=v5"<<endl;
35     return 0;
}
```

示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector <int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector <int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector <int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector <int> v4(v1.begin( ), v1.end( ));
16     vector <int> v5(&num[0],&num[10]);
17
18     // 将向量v2中的10个元素值均修改为5
19     for (i=0;i<10;i++)
20         v2[i]=5;
21     //五个vector对象是相等的
22     //可用operator==来判断
23     //v1==v2
24     if (v1==v2) cout<<"v1==v2"<<endl;
25     else cout<<"v1!=v2"<<endl;
26     //v1==v3
27     if (v1==v3) cout<<"v1==v3"<<endl;
28     else cout<<"v1!=v3"<<endl;
29     //v1==v4
30     if (v1==v4) cout<<"v1==v4"<<endl;
31     else cout<<"v1!=v4"<<endl;
32     //v1==v5
33     if (v1==v5) cout<<"v1==v5"<<endl;
34     else cout<<"v1!=v5"<<endl;
35     return 0;
}
```

示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector <int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector <int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector <int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector <int> v4(v1.begin( ), v1.end( ));
16     vector <int> v5(&num[0],&num[10]);
17
18     // 将向量v2中的10个元素值均修改为5
19     for (i=0;i<10;i++)
20         v2[i]=5;
21     //五个vector对象是相等的
22     //可用operator==来判断
23     //v1==v2
24     if (v1==v2) cout<<"v1==v2"<<endl;
25     else cout<<"v1!=v2"<<endl;
26     //v1==v3
27     if (v1==v3) cout<<"v1==v3"<<endl;
28     else cout<<"v1!=v3"<<endl;
29     //v1==v4
30     if (v1==v4) cout<<"v1==v4"<<endl;
31     else cout<<"v1!=v4"<<endl;
32     //v1==v5
33     if (v1==v5) cout<<"v1==v5"<<endl;
34     else cout<<"v1!=v5"<<endl;
35     return 0;
}
```


示例： vector定义并初始化方法

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int num[10] = {5,5,5,5,5,5,5,5,5,5};
7      int i;
8      // 初始化有10个元素、其值都是5的向量v1
9      vector <int> v1(10,5);
10     // 初始化size为10、元素值为默认值0的向量v2
11     vector <int> v2(10);
12     //复制构造函数，用另一个向量v1来初始化此向量v3
13     vector <int> v3(v1);
14     // 指针可以作为迭代器来使用
15     vector <int> v4(v1.begin( ), v1.end( ));
16     vector <int> v5(&num[0],&num[10]);

        v1==v2
        v1==v3
        v1==v4
        v1==v5
        Program ended with exit code: 0

17     // 将向量v2中的10个元素值均修改为5
18     for (i=0;i<10;i++)
19     |     v2[i]=5;
20     //五个vector对象是相等的
21     //可用operator==来判断
22     //v1==v2
23     if (v1==v2) cout<<"v1==v2"<<endl;
24     else cout<<"v1!=v2"<<endl;
25     //v1==v3
26     if (v1==v3) cout<<"v1==v3"<<endl;
27     else cout<<"v1!=v3"<<endl;
28     //v1==v4
29     if (v1==v4) cout<<"v1==v4"<<endl;
30     else cout<<"v1!=v4"<<endl;
31     //v1==v5
32     if (v1==v5) cout<<"v1==v5"<<endl;
33     else cout<<"v1!=v5"<<endl;
34     return 0;
35 }
```


示例： 比较两个vector

```
#include <vector>
#include <iostream>
using namespace std;
int main( )
{
    vector<int> v1;
    vector<int> v2;
    v1.push_back (5);
    v1.push_back (1);
    v2.push_back (1);
    v2.push_back (2);
    v2.push_back (3);
    cout << (v1 < v2)<<endl;
    return 0;
}
```

示例： 比较两个vector

```
#include <vector>
#include <iostream>
using namespace std;
int main( )
{
    vector<int> v1;
    vector<int> v2;
    v1.push_back (5);
    v1.push_back (1);
    v2.push_back (1);
    v2.push_back (2);
    v2.push_back (3);
    cout << (v1 < v2)<<endl;
    return 0;
}
```

输出结果：

0

示例： 比较两个vector

```
#include <vector>
#include <iostream>
using namespace std;
int main( )
{
    vector<int> v1;
    vector<int> v2;
    v1.push_back (5);
    v1.push_back (1);
    v2.push_back (6);
    v2.push_back (2);
    v2.push_back (3);
    cout << (v1 < v2)<<endl;
    return 0;
}
```

输出结果：

1

访问vector的信息

➤ 向量容器 **vector** 有4个成员函数可以返回向量的信息，函数原型如下：

1.

// 返回当前vector所容纳元素的数目

size_type size();

2.

// 返回当前vector在重新进行内存分配以前所能

// 容纳的元素数量

size_type capacity();

访问vector的信息

➤ 向量容器 **vector** 有4个成员函数可以返回向量的信息，函数原型如下：

3.

// 返回当前vector所能容纳元素数量的最大值
//(包括可重新分配内存)

size_type max_size();

4.

// 如果当前vector没有容纳任何元素，则
//empty()函数返回true，否则返回false

bool empty();

示例：访问vector的信息

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main( )
5  {
6      int i;
7      vector<int> ivec(2,10);
8      cout << "Max_size=" << ivec.max_size( )<<endl;
9      cout << "size=" << ivec.size( ) << endl;           // size=2
10     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=2
11     ivec.push_back(1);
12     cout << "size=" << ivec.size( ) << endl;           // size=3
13     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=4
14     ivec.push_back(2);
15     cout << "size=" << ivec.size( ) << endl;           // size=4
16     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=4
17     ivec.push_back(3);
```

示例：访问vector的信息

```
18     cout << "size=" << ivec.size( ) << endl;           // size=5
19     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
20     ivec.push_back(4);
21     cout << "size=" << ivec.size( ) << endl;           // size=6
22     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
23     for(i=0; i<ivec.size( ); i++)
24     |     cout << ivec[i] << ' ';                       // 10 10 1 2 3 4
25     cout << endl;
26     ivec.pop_back( );
27     ivec.pop_back( );
28     cout << "size=" << ivec.size( ) << endl;           // size=4
29     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
30     ivec.clear( );
31     cout << "size=" << ivec.size( ) << endl;           // size=0
32     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
33     return 0;
34 }
```

示例：访问vector的信息

```
18     cout << "size=" << ivec.size( ) << endl;           // size=5
19     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
20     ivec.push_back(4);
21     cout << "size=" << ivec.size( ) << endl;           // size=6
22     cout << "capacity=" << ivec.capacity( ) << endl;    // capacity=8
23     for(i=0; i<ivec.size( ); i++)
24     |     cout << ivec[i] << ' ';
25     cout << endl;
26     ivec.pop_back( );
27     ivec.pop_back( );
28     cout << "size=" << ivec.size( ) <<
29     cout << "capacity=" << ivec.capacity( ) <<
30     ivec.clear( );
31     cout << "size=" << ivec.size( ) <<
32     cout << "capacity=" << ivec.capacity( ) <<
33     return 0;
34 }
```

```
Max_size=4611686018427387903
size=2
capacity=2
size=3
capacity=4
size=4
capacity=4
size=5
capacity=8
size=6
capacity=8
10 10 1 2 3 4
size=4
capacity=8
size=0
capacity=8
Program ended with exit code: 0
```


vector元素的访问和遍历

- 向量类似于数组，向量类模板中对下标运算符`[]`进行了重载。可以按照访问数组元素的方式来访问向量中的元素。

例如：

设`vecList`是一个向量对象，则`vecList[index]`就代表由`index`指定的位置上的元素。

vector元素的访问和遍历

- 向量类中还提供了3个成员函数来访问向量中的元素。
设vecList是一个向量对象，函数原型如下：

1.

//返回vecList容器中指定位置index的元素的引用：

vecListp[]

2.

//返回第一个元素的引用(不检查容器是否为空)。

//如果容器非空，则vecList.front()和 vecList[0]相同

vecList.front()

vector元素的访问和遍历

➤ 可以使用迭代器来操作向量容器中的数据：

例如，语句：

```
vector<int>::iterator intVecIter;
```

将intVecIter声明为元素为int类型的向量容器的迭代器。

因为iterator是一个定义在vector类中的typedef，所以必须使用容器名(vector)、容器元素类型和作用域符来使用iterator。

vector元素的访问和遍历

➤ 声明了迭代器后，执行语句：

//将使迭代器intVecIter指向容器intVec中第一个元素。

intVecIter = intVec.begin();

// 将迭代器intVecIter加 1，使其指向容器intVec中的下一个元素。

表达式 ++ intVecIter

// 将返回当前迭代器位置上的元素(即intVec[1]或 intVec.at(1))。

表达式 * intVecIter

//将返回当前迭代器位置之后两个位置上的元素(即intVec[3]或 intVec.at(3))

表达式 * (intVecIter + 2)

vector容器中有成员函数begin和end。

函数begin()： 返回容器中第一个元素位置的迭代器，

函数end()： 返回容器中最后一个元素下一个位置的迭代器。

vector元素的访问和遍历

➤ 遍历容器intVec中所有元素并输出到标准输出设备上的代码为：

所谓反向迭代器就是反向移动的迭代器。

例如，语句：

```
vector<int>::reverse_iterator intVecRIter;
```

将intVecRIter声明为元素为int类型的向量容器的反向迭代器。

下面的for循环将逆序遍历容器intVec中所有元素并输出到标准输出设备上：

```
for (intVecRIter = intVec.rbegin( ); intVecRIter != intVec.rend( ); intVecRIter++)  
    cout<<* intVecRIter <<“ ”;
```

vector元素的访问和遍历

➤ vector容器中另有成员函数rbegin和rend：

- 函数rbegin()：返回指向当前vector末尾的反向迭代器，
- 函数rend()：返回指向当前vector起始位置的反向迭代器。

```
for (intVecIter = intVec.begin(); intVecIter != intVec.end(); intVecIter++)  
    cout<<* intVecIter <<"  ";
```

在这个for循环中，每次进行循环条件判断均需去重复计算end，而vector的end()函数不是常数时间的，因此可以先计算end并缓存下来，这样能提高效率。

例如，可将上面程序段改写为：

```
vector<int>:: iterator intVecIter, itEnd; itEnd= intVec.end();  
for (intVecIter = intVec.begin(); intVecIter !=itEnd ;intVecIter++)  
    cout<<* intVecIter <<"  ";
```

vector元素的插入和删除

➤ 向量vector类中的成员函数：

- `push_back()`：可以将一个元素添加到容器的末尾，
- `pop_back()`：可以删除容器最后一个元素，
- `clear()`：可以删除容器中的所有元素。
- 还可以使用插入函数`insert()`和删除函数`erase()`。

vector元素的插入和删除

➤ insert() 函数有以下三种用法：

函数原型： `iterator insert(iterator loc, const TYPE &val);`

功能：在指定位置loc前插入值为val的元素，返回指向这个元素的迭代器。

函数原型： `void insert(iterator loc, size_type num, const TYPE &val);`

功能：在指定位置loc前插入num个值为val的元素。

函数原型： `void insert(iterator loc, input_iterator start, input_iterator end);`

功能：在指定位置loc前插入区间[start, end)的所有元素。

vector元素的插入和删除

➤ erase() 函数有以下两种用法：

函数原型： `iterator erase(iterator loc);`

功能：删除指定位置loc的元素，返回值是指向删除元素的下一位置的迭代器。

函数原型： `iterator erase(iterator start, iterator end);`

功能：删除区间[start, end)的所有元素，返回值是指向删除的最后一个元素的下一位置的迭代器。

示例： vector元素的插入和删除

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  void allPrint(vector<int> ivec);
5  int main( )
6  {   vector<int> v1;
7      v1.push_back(20);
8      v1.push_back(30);
9      v1.push_back(40);
10     allPrint(v1);    // 20 30 40
11     v1.insert(v1.begin( ), 10);
12     allPrint(v1);    // 10 20 30 40
13     v1.insert( v1.begin( ) + 1, 3, 15);
14     allPrint(v1);    // 10 15 15 15 20 30 40
15     v1.insert( v1.end( ), v1.begin( )+1, v1.begin( )+5 );
16     allPrint(v1);    // 10 15 15 15 20 30 40 15 15 15 20
17     v1.erase(v1.begin( )+3);
18     allPrint(v1);    // 10 15 15 20 30 40 15 15 15 20
19     v1.erase(v1.begin( )+4,v1.end( ));
20     allPrint(v1);    // 10 15 15 20
21     return 0;}
```

```
22 void allPrint(vector<int> ivec)
23 {
24     vector<int>::iterator Iter,itLast;
25     itLast=ivec.end( );
26     for ( Iter = ivec.begin( );
27           Iter !=itLast  ; Iter++ )
28         cout << " " << *Iter;
29     cout << endl;
30 }
```

示例： vector元素的插入和删除

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  void allPrint(vector<int> ivec);
5  int main( )
6  {   vector<int> v1;
7      v1.push_back(20);
8      v1.push_back(30);
9      v1.push_back(40);
10     allPrint(v1);    // 20 30 40
11     v1.insert(v1.begin( ), 10);
12     allPrint(v1);    // 10 20 30 40
13     v1.insert( v1.begin( ) + 1, 3, 15);
14     allPrint(v1);    // 10 15 15 15 20 30 40
15     v1.insert( v1.end( ), v1.begin( )+1, v1.begin( )+5 );
16     allPrint(v1);    // 10 15 15 15 20 30 40 15 15 15 20
17     v1.erase(v1.begin( )+3);
18     allPrint(v1);    // 10 15 15 20 30 40 15 15 15 20
19     v1.erase(v1.begin( )+4,v1.end( ));
20     allPrint(v1);    // 10 15 15 20
21     return 0;}
```

```
22 void allPrint(vector<int> ivec)
23 {
24     vector<int>::iterator Iter,itLast;
25     itLast=ivec.end( );
26     for ( Iter = ivec.begin( );
27           Iter !=itLast  ; Iter++ )
28         cout << " " << *Iter;
29     cout << endl;
30 }
```

20 30 40
10 20 30 40
10 15 15 15 20 30 40
10 15 15 15 20 30 40 15 15 15 20
10 15 15 20 30 40 15 15 15 20
10 15 15 20
Program ended with exit code: 0

示例：vector元素的插入和删除

➤ 在执行插入和删除操作后，应注意迭代器的变化情况：

例如：

设有vector<int> v1; v1容器中有8个元素1、2、3、4、5、3、7、8，现在要求删除该容器中所有值为3的元素，写出如下的代码：

```
for(vector<int>::iterator iter=v1.begin(); iter!=v1.end(); iter++)  
{  
    if( *iter == 3)  
        iter = v1.erase(iter);  
}
```

示例：vector元素的插入和删除

➤ 在执行插入和删除操作后，应注意迭代器的变化情况：

例如：

设有vector <int> v1; v1容器中有8个元素1、2、3、4、5、3、7、8，现在要求删除该容器中所有值为3的元素，写出如下的代码：

由于erase()函数的返回值是指向删除元素的下一位置的迭代器，因此这段代码是错误的：

1) 无法删除两个连续的“3”。例如，如果v1中的元素为1、2、3、3、5、3、7、8，执行后，v1中元素为1、2、3、5、7、8，有一个3没有删除。

2) 当3位于vector最后位置的时候，程序在执行时会出错（在v1.end()上执行++操作）。

示例：vector元素的插入和删除

➤ 在执行插入和删除操作后，应注意迭代器的变化情况：

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     vector<int> v1;
7     v1.push_back (1);v1.push_back (2);v1.push_back (3);
8     v1.push_back (3);v1.push_back (5);v1.push_back (3);
9     v1.push_back (7);v1.push_back (8);
10    for(vector<int>::iterator iter=v1.begin( ); iter!=v1.end( ); iter++)
11    {
12        if( *iter == 3)
13            iter = v1.erase(iter);
14    }
15    for(vector<int>::iterator iter=v1.begin( ); iter!=v1.end( ); iter++)
16    {
17        cout<<*iter<<endl;
18    }
19 }
20
```

1
2
3
5
7
8

Program ended with exit code: 0

示例：vector元素的插入和删除

➤ 解决方法：

在删除操作后弃用原来的迭代器，使用返回值；没有删除操作时，仍使用原来的迭代器即可。

正确的代码可写为：

```
for(vector<int>::iterator iter=v1.begin(); iter!=v1.end(); )
{
    if( *iter == 3)
        iter = v1.erase(iter);
    else
        iter ++ ;
}
```

STL String

举例：字符串

- 用双引号括起来的字符序列；
- 结束标志： ‘\0’ ；
- 字符串长度：有效字符的个数

例如，“abc” 的长度为3

- 用字符型数组存放字符串时，在有效字符后自动加 ‘\0’。

STL String

定义格式：

char 字符数组名 [最大字符数+1] = "字符串";

例1: char c [7] = "MONDAY";

或 char c [7] = { "MONDAY"};

例2: char c[7];

c[0] = 'M'; c[1] = 'O'; c[2] = 'N'; c[3] = 'D';

c[4] = 'A'; c[5] = 'Y'; c[6] = '\0';;

字符数组的元素是单个字符数据。

STL String

已知： `char name[20]`

输入：

格式1： `cin>>name;`

格式2： `cin.get(name, n);` //读入n-1个字符

输出：

格式： `cout<<name;`

STL String

```
1 #include <iostream>
2 using namespace std;
3 int main( )
4 {
5     char name1[20], name2[20];
6     cin.get(name1, 20);
7     cout<<name1<<endl;
8     cin>>name2;
9     cout<<name2<<endl;
10 }
```

```
zhangsan
zhangsan
lisi
lisi
Program ended with exit code: 0
```

STL String

➤再举例来说：

如果文本格式是：

用户名 电话号码，文件名name.txt

Tom 23245332

Jenny 22231231

Henry 22183942

Tom 23245332

...

现在需要对用户名排序，且只输出不同的姓名。

STL String

➤ C/C++ 字符数组的处理步骤：

- (1) 先打开文件，检测文件是否打开，如果失败，则退出。
- (2) 声明一个足够大得二维字符数组或者一个字符指针数组。
- (3) 读入一行到字符空间，然后分析一行的结构，找到空格，存入字符数组中。
- (4) 关闭文件，写一个排序函数，或者写一个比较函数，使用qsort排序。
- (5) 遍历数组，比较是否有相同的，如果有，则要删除。
- (6) 输出信息。

STL String的处理方式

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <fstream>
using namespace std;
int main( )
{
    ifstream in("name.txt");
    string strtmp;
    vector<string> vect;
    while(getline(in, strtmp, '\n'))
        vect.push_back(strtmp.substr(0, strtmp.find(' ')));
    sort(vect.begin( ), vect.end( ));
    vector<string>::iterator it=unique(vect.begin( ),
vect.end( ));
    copy(vect.begin( ), it, ostream_iterator<string>(cout,
"\n"));
    return 0;
}
```

String分析

- 这里 `string` 的作用不只是可以 **存储字符串**，还可以提供 **字符串的比较，查找** 等。在 `sort` 和 `unique` 函数中就默认使用了 `less` 和 `equal_to` 函数。
- 上面的一段代码，其实使用了 `string` 的以下功能：
 - **存储功能**：在 `getline()` 函数中
 - **查找功能**：在 `find()` 函数中
 - **子串功能**：在 `substr()` 函数中
 - **`string operator <`**：默认在 `sort()` 函数中调用
 - **`string operator ==`**：默认在 `unique()` 函数中调用

String分析

- string并不是一个单独的容器，只是basic_string 模板类的一个typedef而已，相对应的还有wstring。
- 在string 头文件中会发现下面的代码：

```
extern "C++"  
{  
    typedef basic_string <char> string;  
    typedef basic_string <wchar_t> wstring;  
} // extern "C++"
```


String本质

- `string`其实相当于一个保存字符的序列容器，因此除了有字符串的一些常用操作以外，还有包含了所有的序列容器的操作。
- 字符串的常用操作包括：
 - 增加
 - 删除
 - 修改
 - 查找比较
 - 链接
 - 输入
 - 输出等

String类的构造函数

函数原型： `string(const char *s);`

功能：用const字符串s初始化。

函数原型： `string(int n, char c);`

功能：用n个字符c初始化。

此外，string类还支持默认构造函数和复制构造函数。

如： `string s1; string s2 = "Hello";` 等都是正确的写法。

当构造的string太长而无法表达时，会抛出length_error异常。

示例：String类的建立和初始化

```
#include <string>
#include <iostream>
using namespace std;
int main( )
{
    const char * S1="12345";
    string S2;           //建立长度为0的字符串
    string S3("abcde");  //用字符串初始化新串
    string S4(S3);       //利用已存在的串S3初始化新串
    string S5(S3,0,3);    //利用已存在的串S3初始化新串
    string S6(S1,3);      //利用已存在的字符数组初始化新串
    string S7(6,'A');
    cout<<"S2="<<S2<<endl;
    cout<<"S3="<<S3<<endl;
    cout<<"S4="<<S4<<endl;
    cout<<"S5="<<S5<<endl;
    cout<<"S6="<<S6<<endl;
    cout<<"S7="<<S7<<endl;
    return 0;
}
```

```
S2=
S3=abcde
S4=abcde
S5=abc
S6=123
S7=AAAAAA
Program ended with exit code: 0
```

String类的字符操作

函数原型：
`const char & operator[] (int n);`
`const char & at (int n);`
`char & operator[] (int n);`
`char & at (int n);`

`operator[]`和`at()`均返回当前字符串中第`n`个字符的位置，但`at`函数提供范围检查。

当越界时会抛出`out_of_range`异常，下标运算符`[]`不提供检查访问。

String类的特性描述

函数原型： `int capacity() ;`

功能： 返回当前容量(即string中不必增加内存即可存放的元素个数)。

函数原型： `int max_size() ;`

功能： 返回string对象中可存放的最大字符串的长度。

函数原型： `int size() ;`

功能： 返回当前字符串的大小。

函数原型： `int length() ;`

功能： 返回当前字符串的长度。

函数原型： `bool empty() ;`

功能： 当前字符串是否为空。

函数原型： `void resize(int len, char c) ;`

功能： 把字符串当前大小置为len，并用字符c填充不足的部分。

String类的输入输出操作

string类重载运算符 `operator>>` 用于输入,

同样重载运算符 `operator<<` 用于输出操作。

函数原型: `getline(istream &in,string &s);`

功能: 用于从输入流in中读取字符串到s中, 以换行符 ‘\n’ 分开。

String类的赋值操作

函数原型: `string &operator=(const string &s);`

功能: 把字符串s赋给当前字符串。

函数原型: `string &assign(const char *s);`

功能: 用c类型字符串s赋值。

函数原型: `string &assign(const char *s,int n);`

功能: 用c字符串s开始的n个字符赋值。

函数原型: `string &assign(const string &s);`

功能: 把字符串s赋给当前字符串。

函数原型: `string &assign(int n,char c);`

功能: 用n个字符c赋值给当前字符串。

函数原型: `string &assign(const string &s,int start,int n);`

功能: 把字符串s中从start开始的n个字符赋给当前字符串。

函数原型: `string &assign(const_iterator first,const_iterator last);`

功能: 把first和last迭代器之间的部分赋给字符串。

String类的连接操作

函数原型: `string &operator += (const string &s);`

功能: 把字符串s连接到当前字符串的结尾。

函数原型: `string &append(const char *s);`

功能: 把c类型字符串s连接到当前字符串结尾。

函数原型: `string &append(const char *s,int n);`

功能: 把c类型字符串s的前n个字符连接到当前字符串结尾。

函数原型: `string &append(const string &s);`

功能: 同`operator+=()`。

函数原型: `string &append(const string &s,int pos,int n);`

功能: 把字符串s中从pos开始的n个字符连接到当前字符串的结尾。

函数原型: `string &append(int n,char c);`

功能: 在当前字符串结尾添加n个字符c。

函数原型: `string &append(const_iterator first,const_iterator last);`

功能: 把迭代器first和last之间的部分连接到当前字符串的结尾。

String类的比较操作

函数原型: `bool operator==(const string &s1,const string &s2);`

功能: 比较两个字符串是否相等。

运算符"`>`", "`<`", "`>=`", "`<=`", "`!=`"均被重载用于字符串的比较

函数原型: `int compare(const string &s);`

功能: 比较当前字符串和s的大小。

函数原型: `int compare(int pos, int n,const string &s);`

功能: 比较当前字符串从pos开始的n个字符组成的字符串与s的大小。

函数原型: `int compare(int pos, int n,const string &s,int pos2,int n2);`

功能: 比较当前字符串从pos开始的n个字符组成的字符串与s中pos2开始的n2个字符组成的字符串的大小。

函数原型: `int compare(const char *s);`

`int compare(int pos, int n,const char *s) const;`

`int compare(int pos, int n,const char *s, int pos2) ;`

功能: `compare`函数在`>`时返回1, `<`时返回-1, `==`时返回0 。

String类常用操作符

string类重载了多个操作符用于对字符串进行操作。

操作符	示 例	功 能
+	S+T	将S与T连成一个新串
=	T=S	以S更新T
+=	T+=S	T=T+S
==, !=, <, <=, >, >=	T==S, T!=S ,T<S, T<=S, T>S, T>=S	将T串与S串进行比较
[]	S[i]	存取串中第i个元素
<<	cout<<S	将串S插入出流对象
>>	cin>>S	从输入流对象提取串给S

String类的子串和交换

函数原型: `string substr(int pos = 0, int n = npos);`

功能: `string`的子串, 返回`pos`开始的`n`个字符组成的字符串。

函数原型: `void swap(string &s2);`

功能: `string`的交换, 交换当前字符串与`s2`的值。

String类的查找操作

函数原型： `int find(char c, int pos = 0);`

功能：从pos开始查找字符c在当前字符串的位置。

函数原型： `int find(const char *s, int pos = 0);`

功能：从pos开始查找字符串s在当前串中的位置。

函数原型： `int find(const char *s, int pos, int n);`

功能：从pos开始查找字符串s中前n个字符在当前串中的位置。

函数原型： `int find(const string &s, int pos = 0);`

功能：从pos开始查找字符串s在当前串中的位置。

查找成功时返回所在位置，失败返回`string::npos`的值。

String类的查找操作

函数原型: `int rfind(char c, int pos = npos);`

功能: 从pos开始从后向前查找字符c在当前串中的位置。

函数原型: `int rfind(const char *s, int pos = npos);`
`int rfind(const char *s, int pos, int n = npos);`
`int rfind(const string &s, int pos = npos);`

功能: 从pos开始从后向前查找字符串s中前n个字符组成的字符串在当前串中的位置, 成功返回所在位置, 失败时返回string::npos的值。

函数原型: `int find_first_of(char c, int pos = 0);`

功能: 从pos开始查找字符c第一次出现的位置。

函数原型: `int find_first_of(const char *s, int pos = 0);`
`int find_first_of(const char *s, int pos, int n);`
`int find_first_of(const string &s, int pos = 0);`

功能: 从pos开始查找当前串中第一个在s的前n个字符组成的数组里的字符的位置。查找失败返回string::npos。

String类的查找操作

函数原型: `int find_first_not_of(char c, int pos = 0);`

`int find_first_not_of(const char *s, int pos = 0);`

`int find_first_not_of(const char *s, int pos, int n);`

`int find_first_not_of(const string &s, int pos = 0);`

功能: 从当前串中查找第一个不在串s中的字符出现的位置, 失败返回`string::npos`。

函数原型: `int find_last_of(char c, int pos = npos);`

`int find_last_of(const char *s, int pos = npos);`

`int find_last_of(const char *s, int pos, int n = npos);`

`int find_last_of(const string &s, int pos = npos);`

`int find_last_not_of(char c, int pos = npos);`

`int find_last_not_of(const char *s, int pos = npos);`

`int find_last_not_of(const char *s, int pos, int n);`

`int find_last_not_of(const string &s, int pos = npos);`

功能: `find_last_of`和`find_last_not_of`与`find_first_of`和`find_first_not_of`相似, 只不过是后向前查找。

String类的替换操作

函数原型: `string &replace(int p0, int n0, const char *s);`

功能: 删除从p0开始的n0个字符, 然后在p0处插入串s。

函数原型: `string &replace(int p0, int n0, const char *s, int n);`

功能: 删除p0开始的n0个字符, 然后在p0处插入字符串s的前n个字符。

函数原型: `string &replace(int p0, int n0, const string &s);`

功能: 删除从p0开始的n0个字符, 然后在p0处插入串s。

函数原型: `string &replace(int p0, int n0, const string &s, int pos, int n);`

功能: 删除p0开始的n0个字符, 然后在p0处插入串s中从pos开始的n个字符。

函数原型: `string &replace(int p0, int n0, int n, char c);`

功能: 删除p0开始的n0个字符, 然后在p0处插入n个字符c。

String类的替换操作

函数原型: `string &replace(iterator first0, iterator last0, const char *s);`

功能: 把[first0, last0)之间的部分替换为字符串s。

函数原型: `string &replace(iterator first0, iterator last0, const char *s, int n);`

功能: 把[first0, last0)之间的部分替换为s的前n个字符。

函数原型: `string &replace(iterator first0, iterator last0, const string &s);`

功能: 把[first0, last0)之间的部分替换为串s。

函数原型: `string &replace(iterator first0, iterator last0, int n, char c);`

功能: 把[first0, last0)之间的部分替换为n个字符c。

函数原型: `string &replace(iterator first0, iterator last0, const_iterator first, const_iterator last);`

功能: 把[first0, last0)之间的部分替换成[first, last)之间的字符串。

String类的插入操作

函数原型：
`string &insert(int p0, const char *s);`
`string &insert(int p0, const char *s, int n);`
`string &insert(int p0, const string &s);`
`string &insert(int p0, const string &s, int pos, int n);`
功能：这4个函数在p0位置插入字符串s中pos开始的前n个字符。

函数原型：
`string &insert(int p0, int n, char c);`
功能：此函数在p0处插入n个字符c。

函数原型：
`iterator insert(iterator it, char c);`
功能：在it处插入字符c，返回插入后迭代器的位置。

函数原型：
`void insert(iterator it, const_iterator first, const_iterator last);`
功能：在it处插入[first, last)之间的字符。

函数原型：
`void insert(iterator it, int n, char c);`
功能：在it处插入n个字符c。

String类的删除操作

函数原型： `iterator erase(iterator first, iterator last);`

功能：删除[first, last)之间的所有字符，返回删除后迭代器的位置。

函数原型： `iterator erase(iterator it);`

功能：删除it指向的字符，返回删除后迭代器的位置。

函数原型： `string &erase(int pos = 0, int n = npos);`

功能：删除pos开始的n个字符，返回修改后的字符串。

示例：String运算符使用

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main( ){
5      string strinfo="Please input your name:"; cout << strinfo ;
6      cin >> strinfo;
7      if( strinfo == "winter" )
8          cout << "you are winter!"<<endl;
9      else if( strinfo != "wende" )
10         cout << "you are not wende!"<<endl;
11     else if( strinfo < "winter")
12         cout << "your name should be ahead of winter"<<endl;
13     else cout << "your name should be after of winter"<<endl;
14     strinfo += " , Welcome to China!\n";
15     cout << strinfo<<endl;
16     cout <<"Your name is :"<<endl;
17     string strtmp = "How are you? " + strinfo;
18     for(int i = 0 ; i < strtmp.size( ); i ++ )
19         cout<<strtmp[i];
20     return 0;
21 }
```

示例：String运算符使用

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main( ){
5      string strinfo="Please input your name";
6      cin >> strinfo;
7      if( strinfo == "winter" )
8          cout << "you are winter!"<<endl;
9      else if( strinfo != "wende" )
10         cout << "you are not wende!"<<endl;
11      else if( strinfo < "winter")
12         cout << "your name should be ahead of winter"<<endl;
13      else cout << "your name should be after of winter"<<endl;
14      strinfo += " , Welcome to China!\n";
15      cout << strinfo<<endl;
16      cout <<"Your name is :"<<endl;
17      string strtmp = "How are you? " + strinfo;
18      for(int i = 0 ; i < strtmp.size( ); i ++ )
19          cout<<strtmp[i];
20      return 0;
21 }
```

Please input your name:ZhangSan
you are not wende!
ZhangSan , Welcome to China!

Your name is :
How are you? ZhangSan , Welcome to China!
Program ended with exit code: 0

示例：String运算符使用

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main( ){
5      string  str1("C++");
6      string  str2("is");
7      string  str3,str4;
8      string  str5("powerful");
9      cin>>str3;    //字符串输入
10     cout<<str1<<" "<<str2<<" "<<str3<<"!"<<endl; //string对象输出
11     str4=str1+" "+str2+" "+str3+"!"; //字符串连接
12     cout<<str4<<endl;
13     if(str3<str5)           //字符串比较
14     |     cout<<"str3<str5"<<" str5:"<<str5<<endl;
15     string  str6(str5); //字符串赋值
16     str6=str1+" "+str2+" "+str6+"!";
17     cout<<str6<<endl;
18     return 0;
19 }
```

示例：String运算符使用

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main( ){
5      string  str1("C++");
6      string  str2("is");
7      string  str3, str4;
8      string  str5("powerful");
9      cin>>str3;    //字符串输入
10     cout<<str1<<" "<<str2<<" "<<str3<<"!"<<endl; //string对象输出
11     str4=str1+" "+str2+" "+str3+"!"; //字符串连接
12     cout<<str4<<endl;
13     if(str3<str5)    //字符串比较
14     |     cout<<"str3<str5"<<" str5:"<<str5<<endl;
15     string str6(str5); //字符串赋值    beautiful
16     str6=str1+" "+str2+" "+str6+"!";    C++ is beautiful!
17     cout<<str6<<endl;    C++ is beautiful!
18     return 0;    str3<str5 str5:powerful
19 }    C++ is powerful!
    Program ended with exit code: 0
```

示例2：字符串查找和替换

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4  int main( )
5  {
6      string text("I like c++, I use c++ programming."); //文本
7      string newstr;                                     //新串
8      string sstr;                                       //待查串
9      int pos;                                           //存放查找到串的位置
10     cout<<"Input string and new string:";
11     cin>>sstr>>newstr;
12     if((pos=text.find(sstr))==string::npos) //未查找到
13     |     cout<<sstr<<" not found in \""<<text<<"\"<<endl;
14     else {
15         cout<<"old string: "<<text<<endl;
16         text.replace(pos,sstr.length(),newstr);
17         cout<<"new string: "<<text<<endl;
18     }
19 }
```

Input string and new string:c++ java
old string: I like c++, I use c++ programming.
new string: I like java, I use c++ programming.
Program ended with exit code: 0

示例3：string类的操作符将字符串排序

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4  int main( )
5  {
6      const int n=6;
7      string line[ ]={"A","above","a","an","about","1234"};
8      for( int i=0; i<n-1; i++ )
9          for ( int j=i+1; j<n; j++ )
10             if (line[i]>line[j]) //如果前面字符串比后面大
11                 { // line[i].swap(line[j]); 交换
12                     string temps;
13                     temps=line[i];
14                     line[i]=line[j];
15                     line[j]=temps;
16                 }
17     for(int i=0;i<n;i++)
18         cout<<line[i]<<endl;
19     return 0;
20 }
```

1234
A
a
about
above
an
Program ended with exit code: 0

String类的迭代器处理

- string类提供了向前和向后遍历的迭代器`iterator`，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。
- 用`string::iterator`或`string::const_iterator`声明迭代器变量。
- `const_iterator`不允许改变迭代的内容。

String类的常用迭代器函数

函数原型: `const_iterator begin();`

`iterator begin();`

功能: 返回string的起始位置。

函数原型: `const_iterator end();`

`iterator end();`

功能: 返回string的最后一个字符后面的位置。

函数原型: `const_iterator rbegin();`

`iterator rbegin();`

功能: 返回string的最后一个字符的位置。

函数原型: `const_iterator rend();`

`iterator rend();`

功能: 返回string第一个字符位置的前面。

`rbegin`和`rend`用于从后向前的迭代访问, 通过设置迭代器
`string::reverse_iterator`, `string::const_reverse_iterator`实现。

String类的字符串流处理

- 通过定义ostringstream和istringstream变量实现，需包含<sstream>头文件。

例如：

```
#include<iostream>
#include<sstream>
using namespace std;
int main( ){
    string input("hello,this is a test");
    istringstream is(input);
    string s1,s2,s3,s4;
    is>>s1>>s2>>s3>>s4;
    s1="hello,this",s2="is",s3="a",s4="test";
    ostringstream os;
    os<<s1<<s2<<s3<<s4<<endl;
    cout<<os.str( );
    return 0; }
```

String类的字符串流处理

- 通过定义ostringstream和istringstream变量实现，需包含<sstream>头文件。

例如：

```
#include<iostream>
#include<sstream>
using namespace std;
int main( ){
    string input("hello,this is a test");
    istringstream is(input);
    string s1,s2,s3,s4;
    is>>s1>>s2>>s3>>s4;
    s1="hello,this",s2="is",s3="a",s4="test";
    ostringstream os;
    os<<s1<<s2<<s3<<s4<<endl;
    cout<<os.str( );
    return 0; }
```

```
hello,thisisatest
Program ended with exit code: 0
```