# STL与泛型编程

## Part1 C++模板

### C++模板简介1

* **模板**（Templates）是c++的一种特性，允许函数或类（对象）通过泛型（generic types）的形式表现或运行。
* 模板可以使得函数或类在对应不同的**型别**（types）的时候正常工作，而无需为**每个类型**都写一份代码
* 一个简单的例子：
  + 如果要写一个取两个数中较大值函数Max，在不使用模板的情况下，我们不得不针对不同的类型（比如int，long，char）提供每一种类型的重载。

|  |  |  |
| --- | --- | --- |
| int Max(int a, int b)  {  return ( a > b ) ? a : b  } | int Max(long a, long b)  {  return ( a > b ) ? a : b  } | int Max(char a, char b)  {  return ( a > b ) ? a : b  } |

### C++模板简介2

* 一个简单的例子（续）
  + 如果使用模板，则可以省去一堆冗余代码。

|  |
| --- |
| template <typename **T**> **T** Max(**T** a,**T** b)  {  return ( a > b ) ? a : b  } |

* C++主要有两种类型的模板
  + 类模板（Class Template）：使用泛型参数的类（class with generic parameters）
  + 函数模板（Function Template）：使用泛型参数的函数（functions with generic parameters）

### C++模板简介3

* 模板的实例化

模板的声明（declaration）其实并未给出一个函数或类的完全定义（definition），只是提供了一个函数或类的语法框架（syntactical skeleton）

* 实例化是指从模板构造出一个真正的类或者函数的过程，比如：template <typename T> struct Object{...};

可以用来构造诸如 Object<int>, Object<char>, Object<int \*> 等等不同型别的具体实例

* + 实例化有两种类型
    - **显式** 实例化-在代码中明确指定要针对哪种型别进行实例化
    - **隐式** 实例化-在首次使用时具体的情况使用那一种合适的型别进行实例化

### C++函数模板

#### C++函数模板1

* + 函数模板的定义：
    - **函数模板** 是参数化的一族函数
    - 通过函数模板可以定义出一系列的函数，这些函数都是基于同一套代码，但是可以作用在不同的型别的参数上

|  |  |
| --- | --- |
| **template** <**typename** T>  inline T Max(const T &a, const T &b)  {  return (a>b)?a:b;  } | * + 定义一个函数模板 * 定义一个函数模板，返回两个数中较大值，该函数有两个参数（a,b） * 参数型别未定，以模板参数T表示 * 模板参数由关键字typename引入 |

#### C++函数模板2

* + 定义函数模板（续）
    - 也可以使用class替代typename来定义型别参数

|  |
| --- |
| template <**class** T> inline T Max (const T &a, const T &b){...} |

* 从语法上来说class和typename没有任何的却别
* 但从语义上来说，class可能会导致误区，即只有类才能作为型别参数；而事实上所表达的意思不仅仅针对类，任何型别都可以
* class可以取代typename，但struct却不可以，一下语法上是错误的。

|  |
| --- |
| **//this is wrong!!!**  template <struct T> inline T Max(const T &a, const T &b) {...} |

* **请尽量使用typename**

#### C++函数模板3

* + 函数模板的使用
    - 调用Max， 用int， float，以及std::wstring作为模板参数替代**T**
    - 对于不同的型别，都从模板实例化出不同的函数实体
    - 但是不可以使用两个不同的型别来实例化Max，因为编译器在编译时已经知道了Max函数需要传递的型别

|  |  |
| --- | --- |
| int i=7, j=30;  \_tprintf(TEXT(“Max(i,j)=%d\n”), Max(i,j));  double e=7, f=30;  \_tprintf(TEXT(“Max(e,f)=%d\n”), Max(e,f));  std::wstring s1=TEXT(“mathmatics”), s=TEXT(“math”);  \_tprintf(TEXT(“Max(s1,s2)=%d\n”), Max(s1,s2).c\_str());  **Max(i, f); // compile error:template parameter ‘T’ is ambiguous** |  |

#### C++函数模板4

* + 模板实例化
    - 用具体的型别替代模板参数**T**的过程叫做实例化；从而产生了一个**模板实例**
    - 一旦使用函数模板，这种实例化过程便由编译器自动触发的，不需要额外去请求模板实例化
    - 如果实例化一种型别，而该型别内部并不支持函数所使用的操作，那么就会导致一个编译错误，如下所示：
      * std::complex 并没有重载”>”，也就是说该型别并不支持使用”>”比较大小，而Max函数使用”>”来判断C1，C2的大小，所以无法通过Max(c1,c2)得到预期的结果

|  |
| --- |
| std::complex<int> c1(1,2), c2(15,16); **//编译错误** |

#### C++函数模板5

* + 结论：模板被编译了两次
    - 没有实例化之前，检查模板本身是否有语法错误
    - 实例化期间，检查对模板代码的调用是否合法

#### C++函数模板6

* + 参数推导
    - 模板参数是由传递给模板函数的实参决定的
    - 不允许自动型别的转换：每个T必须严格匹配！

|  |
| --- |
| Max(1,2) //OK:两个实参的型别都是int  Max(1, 2.0) **//ERROR:第一个参数是int，第二个参数是double** |

* 一般有两种处理这种错误的方法：

1. 用static\_cast或强制类型转换

|  |
| --- |
| Max(**static\_cast**(double)(1), 20) |

1. 显示指定T的型别

|  |
| --- |
| Max<**double**> (1, 20) |

#### C++函数模板7

* 函数模板的重载
  + 函数模板也可以像普通的函数一样重载
  + 非模板函数可以和**同名**的模板函数存在
  + 编辑器通过函数模板**参数推导**来决定调用哪个重载

|  |
| --- |
| //普通函数  Inline int const &Max (const int &a, const int &b) ----❶  template <typename T>  Inline T const &Max (const T &a, const T &b) ----❷  template <typename T>  Inline T const &Max (const T &a, const T &b, const T &c, const T &d) ----❸ |

#### C++函数模板8

* 函数模板重载（续）
  + Max(7, 43, 68);调用----❸
  + Max(7.0, 42.0);调用Max<double>(参数推导) ----❷
  + Max(‘a’, ‘b’);调用Max<char>(参数推导) ----❷
  + Max(7, 42);调用非模板函数，参数类型为int ----❶，其他因素都相同的情况下，重载裁决过程调用**非模板**函数，而不是从模板实例化
  + Max<>(7, 42);调用Max<int>(参数推导) ----❷ 允许**空模板参数列表**
  + Max<double>(7,32);调用Max<double> 无需参数推导 ----❷
  + Max(‘a’, 42.7);调用**非模板**函数，参数类型为int ----❶ 对于型别不同的参数只能调用非模板函数（char 和double都可以转为int）

#### C++函数模板9

* 总结
  + 对于不同的实参型别，模板函数定义了一族函数
  + 函数模板可以根据传递的模板实参，进行参数推导过程
  + 可以显式指定模板的实参型别
  + 函数模板可以重载
  + 当重载函数模板时，将改变限制在：显式指定模板参数
  + 所有的重载版本的声明必须位于它们被调用的位置之前

### C++类模板

#### C++类模板1

* 与函数模板相似，类也可以通过参数泛化，从而可以构建出一族不同型别的类实例（对象）
* 类模板实参可以是某一性别或常量（仅限int和enum）

#### C++类模板2

* 一个类模板的例子：Stack<T>

|  |  |
| --- | --- |
| const std::size\_t DefaultStackSize=1024;  template <typename T, std::size\_t n= DefaultStackSize >  class Stack {  public:  void Push(const T const &element);  int Pop(T &element);  int Top(T &element) const;  private:  std::vector<T> m\_members;  std::size\_t m\_maxsize=n;  }; | * T可以是任意型别 * 模板实参也可以是一个int或enum型别的常量（此处是size\_t，本质是int型别） * N是编译时定义的常量 * N可以有默认值 * size\_t 型别的成员变量可以用n初始化 |

#### C++类模板3

* 类模板声明
  + 声明类模板与声明函数模板类似
  + 关键字class和typename都可以用，但还是倾向于typename

|  |
| --- |
| template <typename T, std::size\_t n> class Stack {...};  template <class T, std::size\_t n> class Stack {...}; |

* 在类模板内部，T可以像其他型别一样（比如int，char等）定义成员变量和成员函数

|  |
| --- |
| void Push(const T const &element);  int Pop(T &element);  int Top(T &element) const;  std::vector<T> m\_members; |

#### C++类模板4

* 类模板的声明（续）
  + 除了Copy constructor之外，如果在类模板中需要使用到这个类本身，比如定义operator =，那么应该使用期完成的定义（Stack<T>），而不是省略型别T。如下面的例子：

|  |
| --- |
| template <typename T, std::size\_t n> class Stack  {  public:  ...  Stack(const Stack<T, n> &); //copy constructor  Statck & operator= (const Stack<T, n> &); // assignment operator  ...  }; |

#### C++类模板5

* 类模板的实现
  + 要定义一个类模板的成员函数，则要指明其是一个模板函数，例如，Push函数的定义应当如下：

|  |
| --- |
| template <typename T, std::size\_t nMaxSize>  void Stack<T, nMaxSize>::Push(const T const &element)  {  If (m\_members.size() >= mMaxSize) {  return;  }  m\_members.push\_back(element);  } |

#### C++类模板6

* 类模板的实现（续）
  + Pop函数：从Stack中弹出顶部元素

|  |
| --- |
| template <typename T, std::size\_t nMaxSize>  int Stack<T, nMaxSize>::Pop(T &element)  {  If (m\_members.empty()) {  return 0;  }  element = m\_members.back(); //we have to first store the back element;  m\_member.pop\_back(); //because pop\_back of a vector removes;  return 1; //the last elment but doesn’t return it!  } |

#### C++类模板7

* 类模板的实现（续）
  + GetTop函数：获取Stack顶部元素，但没有Pop出该元素

|  |
| --- |
| template <typename T, std::size\_t nMaxSize>  int Stack<T, nMaxSize>::GetTop (T & element) const  {  If (m\_members.empty()) {  return 0;  }  element = m\_members.back();  return 1;  } |

#### C++类模板8

* 使用类模板
  + Stack<int> stack：定义了一个型别为int的Stack，大小为默认值
  + Stack<int, 100>stack:定义了一个型别为int， 大小为100的Stack
  + 将100个元素Push到Stack中

|  |
| --- |
| for (int i=0; i<100; ++i) {  stack.Push(i);  } |

* Pop出Stack顶部元素

|  |
| --- |
| I nt element;  stack.Pop(element); |

* + 获取Stack顶部元素

|  |
| --- |
| stack.GetTop(element); |

* + Stack的Stack定义：

|  |
| --- |
| **Stack<Stack <int>** ss **>**intStackStack (旧版本的编译器必须有个空格，否则认为>>操作符) |

#### C++类模板 9

* 类模板特化（speicalizations）
  + 允许对一个类模板的某些模板参数型别做特化
  + 特化的作用或好处在于：
    - 对于某种特殊的型别，可能可以做些特别的优化或提供不同的实现
    - 避免在实例化类的时候引起可能产生的诡异行为
* 特化一个类模板的时候也意味着需要特化其所有参数化的成员函数
* 如果要特化一个类，那么做法是：
  + 声明一个带template <>的类，即空参数列表
  + 在类名称后面紧跟的尖括号中显示指明型别，例如：
    - 在类名称后面紧跟的尖括号中显式指明型别，例如：

|  |
| --- |
| template<>  class Stack<std::wstring> {  ...  }; |

#### C++类模板 10

* 类模板特化（specializations）(续)
  + 特化后的具体实现可以和主模板的实现不一样，比如一下的特化增加了一个成员函数，并采用list作为元素存取的实现

|  |
| --- |
| template<> class Stack <std::wstring> {  public:  void SetStackSize(const std::size\_t n) {m\_mMaxSize=n;} //添加了一个新的成员函数  std:size\_t CurrentSize() const {return m\_member.size(); }  void Push(const std::wstring const &element);  int Pop(std::wstring &element);  int GetTop(std::wstring &element) const;  private:  std::size\_t m\_mMaxSize;  std::list<std::wstring> m\_members; //采用list作为的内部实现，替换了原来的vector  }; |

#### C++类模板 11

* 偏特化（parital specilizations）
  + 类模板也可以被偏特化，比如主模板如果定义为：

|  |
| --- |
| template <typename T1, typename T2> class MyClass {...}; //Primary -❶ |

* + 可能产生以下几种对于主模板的偏特化：
    - 将模板参数偏特化为同样型别：

|  |
| --- |
| template <typename T> class MyClass <T, T> {...}; // -❷ |

* 将第二个模板参数偏特化为int型别，不再是泛型的T

|  |
| --- |
| template <typename T> class MyClass <T, int> {...}; //-❸ |

* 将两个型别偏特化为指针：

|  |
| --- |
| template <typename T1, typename T2> class MyClass <T1 \*, T2 \*> {...}; //-❹ |

#### C++类模板12

* 偏特化（Partial specialization） (续)
  + 使用实例：

|  |  |
| --- | --- |
| 使用 | 原型 |
| MyClass<int, float> obj; | MyClass <T1, T2> - ❶ |
| MyClass< float, float> obj; | MyClass <T, T> -❷ |
| MyClass< float, int> obj; | MyClass<T, int> - ❸ |
| MyClass< int \*, float \*> obj; | MyClass<T \*, T\*> -❹ |

* 如果有不止一个偏特化同等程度地能够匹配某一个调用，那么该调用具有二义性，编译器不会通过编译

|  |  |
| --- | --- |
| 使用 | 原型 |
| MyClass <int, int> obj; | ERROR:mathes MyClass<T, T> and MyClass <T, int> |
| MyClass<int \*, int \*> obj; | ERROR:mathes MyClass<T, T> and MyClass <T 1 \*, T2 \*> |

#### C++类模板 13

* 默认模板实参
  + 类似函数的默认参数，对于类模板而言也可以定义其模板参数的默认值，这些值就叫做默认模板实参

|  |
| --- |
| template <typename T, typename **Tcontainer=std::vector<T>** ss > **//默认实参**  class Stack {  private:  Tcontainer m\_container;  }; |

* + Stack<int> intStack;使用默认的vector作为实参
  + Stack<std::wstring, std::list<std::wstring> > wstrStack; 指定list作为容器而非默认的vector

#### C++类模板 14

* 总结
  + 模板类的性质是，有一个或多个型别未被指定
  + 要使用一个模板类，就传入具体的型别作为实参；编译器会基于该型别来实例化类模板
  + 对于类模板而言，只有被调用到的成员函数才会被实例化
  + 类模板可以特定的型别特化（specializations）
  + 类模板也可以用特定的型别偏特化（parital specializations）
  + 类模板参数可有**默认值**

### C++操作符重载（Operator Overloading）

#### C++操作符重载1

* 关键字operator定义了一种特殊的函数，该函数的行为是将操作符应用于某一特定的型别，使之能够通过该操作符进行操作。比如：如果定义了string型别的operator+，那么连接两个字符串a和b的行为就可以用a+b进行操作
* 操作符重载给出了操作符不同的含义
* 编译器通过具体型别来识别某个操作符在该型别上的意义
* 本质上operator重载就是函数，即如果定义了string型别的Append函数，那么string型别的a+b和a.Append(b)是等价的
* 大多数内置的操作符支持重载，比如：

|  |
| --- |
| !,!=,%,%=,&,&=,&&, ||, (), \*, \*=,+,+=, -, -=, /, /=, ^, ^=, |, |=, <, <=, > , >=, =, ==, << ,<<=, >>, >>=, ~, [], new, delete |

#### C++操作符重载2

* 操作符重载的一般规则
  + 不可以用operator定义一种新的操作符， 比如\*\*
  + 对于内置型别（built-in type）, 不能再用operator重载
  + 操作符重载的两种情况：
    - 非静态成员函数
    - 静态全局函数（如果该全局函数需要访问的类的private或protected成员，则声明为friend成员）

|  |
| --- |
| class ComplexType {  public:  //non-static member  ComplexType operator <(const ComplexType &);  //global functions  Friend ComplexType operator+(int, ComplexType &);  }; |

#### C++操作符重载3

* 操作符重载的一般规则（续）
  + 一元操作符（Unary operators）如果声明为成员函数，则没有参数；如果声明为全局函数则有一个参数
  + 二元操作符（Binary operators）如果声明为成员函数，则有一个参数；如果声明为全局函数则有两个参数
  + 如果一个操作符既能用作一元，也能用作二元操作（如：&, \*, +, -），则可以分别重载
  + 操作符重载不能带有默认实参值
  + **除了operator=，所有其他的操作符重载可以被子类继承**

## Part 2 泛型编程

### 泛型编程-概观

* **泛型编程（Generic Programming）**是一种编程方法**，**这种方法将型别（type）以一种to-be-specified-later的方式给出，等到调用的时候，再以参数方式，通过具体的、特定的型别实例化（instantiate）一个具体的方法或对象
* 泛型编程作为一种编程的想法或思维，不依赖于具体的语言
* 大多数面向对象的语言（OO languages）都支持泛型编程，比如：c++, c#, java...
* C++里面的泛型是通过**模板**以及相关性质表现的

#### 关联特性（Traits）

##### 关联特性（Traits）1

* 什么是traits以及为什么使用traits？
  + 假设给定一个数组，计算数组中的所有元素的和：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A[0] | A[1] | ... | A[n] |  |

* 我们可以直接写出如下的计算函数

|  |  |
| --- | --- |
| template <typename T> inline T Sigma(const T const \*start, const T const \* end)  {  T total = T(); //suppose T() actually creates a zero value  while () {  total += \*start++;  }   |  | | --- | | 需要考虑的一点：如果构建型别T的初始值0。此处姑且使用T()， 对于内置的型别，比如int,float等，该初始值是0 |   return total;  } |

* 当我们使用char型别调用模板函数时，问题来了：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| char szNames[]=”abc”;  std::size\_t nLength=strlen(szNames);  char \*p = szNames;  char \*q=szNames+nLength;  printf(“Sigma(szNames)=%d\n”, **Sigma(p, q)**);  294=0x0126   |  |  |  |  | | --- | --- | --- | --- | | 0000 | 0001 | 0010 | 0110 |  |  | | --- | | **该bit溢出**  **(overflowed)** |     char型别能hold住的  最大值是OxFF=255 | * 调用Sigma(szNames)的结果是38(=0x26)!而并非期盼的值(97+98+99=294) * 原因是显而易见的：char型别无法存下294这个值！ * 如果要得到正确的结果，我们不得不强制使用int型别：int s = Sigma<int>(p, q); * 但是这种不必要的转换是完全可以避免的！ |

##### 关联特性（Traits）2

* 什么是traits以及为什么使用traits?（续）
  + 解决的办法是：为每个Sigma函数的参数型别T创建一种关联(association)，关联的型别就是用来存储Sigma结果的型别
  + 这种关联可以看作是型别T的一种特性(characteristic of the typeT)，因此Sigma函数返回值的型别叫做T的trait
  + T与其trait的关系推演如下：

T 🡪 association🡪 characteristic of T🡪 another type🡪 **trait**!

* + **Traits**可以实现为模板类，而关联(association)则是针对每个具体型别T的特化。在这个例子里我们将traits命名为SigmaTraits，做traits模板(traits template)

##### 关联特性（Traits）3

* **Traits**实现

|  |  |
| --- | --- |
| template <typename T> class **SigmaTraits** { };  template <> class **SigmaTraits**<**char**> {  public: typedef int ReturnType;  };  template < > class **SigmaTraits**<**short**> {  public: typedef int ReturnType;  };  template < > class **SigmaTraits**<**int**> {  public: typedef long ReturnType;  };  template < > class **SigmaTraits**<**unsigned int**> {  public: typedef unsigned long ReturnType;  };  template < > class **SigmaTraits**<**float**> {  public: typedef double ReturnType;  }; | C:\Users\qt\AppData\Roaming\Tencent\Users\714049438\QQ\WinTemp\RichOle\08Z4J3R4MB73GSK{CEHPMOK.png |

##### 关联特性（Traits）4

* **Traits**实现（续）
  + 模板类SigmaTraits叫做traits template，它含有其参数型别T的一个特性（trait），即ReturnType
  + 现在Sigma函数可以改写如下：

|  |
| --- |
| template <typename T>  inline typename SigmaTraits<T>::ReturnType Sigma (const T const\* start,const T const\* end)  {  typedef typename SigmaTraits <T> ::ReturnType Retu rnType;  ReturnType s = ReturnType0;  while (start != end) {  s +=\*start++;  }  return s;  } |

##### 关联特性（Traits）5

* Traits实现（续）
* 现在如果我们以char为型别调用Sigma将得到预想中的结果：

|  |
| --- |
| char szNames[]="abc";  std::size\_t nLength=strlen(szNames);  char\*p= szNames;  char\*q= szNames+nLength;  printf("Sigma(szNames)=%d\n”,Sigma(p, q)); |

* 虽然传入参数T的型别是char，但是返回的型别却是int，原因就在template<>class SigmaTraits<char>特化将型别char的返回值变成int(通过typedef int ReturnType)

##### 关联特性（Traits）6

* Traits实现（续）
  + 现在如果我们以char为型别调用Sigma将得到预想中的结果：

|  |
| --- |
| char szNames[]="abc";  std::size\_t nLength=strlen(szNames);  char\*p= szNames;  char\*q= szNames+nLength;  printf("Sigma(szNames)=%d＼n“,**Sigma(p, q)**); |

* 虽然传入参数T的型别是char，但是返回的型别却是int，原因就在于template<>class SigmaTraits<char>特化将型别char的返回值变成了int(通过typedef int ReturnType)

#### 迭代器（Iterators）

##### 迭代器（1）

* 什么是迭代器？
* 迭代器是指针的泛化(generalization of pointers)
* 迭代器本身是一个对象，指向另外一个（可以被迭代的）对象
* 用来迭代一组对象，即如果迭代器指向一组对象中的某个元素，则通过increment以后它就可以指向下这组对象中的下一个元素
  + 在STL中迭代器是容器与算法之间的**接口**
* 算法通常以迭代器作为输入参数
* 容器只要提供一种方式，可以让迭代器访问容器中的元素即可

##### 迭代器（2）

* 迭代器的基本思想
  + 在STL中，迭代器最重要的思想就是分离算法和容器，使之不需要相互依赖
  + 迭代器将算法和容器粘合可以运用到多种不同的容受一对迭代器，分别指向(stick)在一起从而使得一种算法的实现器上，如下面的例子所示，find算法接容器的开始位置和最终位置

|  |
| --- |
| template < class \_lnlt, class \_Ty>  inline \_lnlt find(\_lnlt \_First, \_lnlt \_Last, const \_Ty& \_Val) {  // find first matching \_Val  for (; \_First != \_Last; ++\_First)  if (\*\_First = = \_Val)  break;  return (\_First);  } |

##### 迭代器（3）

* 迭代器的基本思想（续）
  + find算法对于不同的容器，比如vector, list，deque均适用：

|  |
| --- |
| std::vector<int> v(...);  std::list<int> l(...);  std::deque<int> d(...);  std::vector<int>::iterator itv = std::find(v.begin0, v.end0, elementToFind)  std ::list<int> ::iterator itl = std ::find (l.begin (), l.end (), elementTo Find)  std::deque <int> ::iterator it3 = std ::find(d.begin (), d.end0, elementToFind) |

* 每种容器都有其对应的迭代器

## Part 3 容器（Containers）

|  |
| --- |
| Vector |
| Deque |
| List |
| Stack |
| Queue |
| Map and Multimap |
| Set and Multiset |

#### Vector

##### Vector（1）

* 概述
  + Vector是一个能够存放任意型别的动态数组
  + Vector的数据结构和操作与数组(array)类似，在内存中的表现形式是一段地址连续的空间
  + Vector与数组的区别在于，数组大小往往是定义是就固定的(比如：char buffer[256]);Vector支持动态空间大小调整，随着元素的加入，vector内部会自动扩充内存空间。
  + 为了使用vector，必须用include指令包含如下文件，并通过std命名

|  |
| --- |
| #include <vector>  int main() {  std::vector v;  } |

##### Vector（2）

* 创建Vector

|  |  |
| --- | --- |
| 常用方式 | 代码 |
| 创建一个T型别的空vector | std::vector<T> v; |
| 创建一个容量是n的T型别的vector | std::vector<T> v(n); |
| 创建一个容量是n的T型别的vector，并且都初始化为i | std::vector<T> v(n, i); |
| 创建一个已有v的拷贝 | std::vector<T> copyOfV(v); |
| 通过一个数组创建一个vector | int array[ ] = {1, 2, 3, 4, 5, 6J 7, 8, 9, 10 };  std::vector<int> v(array, array + 10); |

##### Vector（3）

* 向Vector添加元素
  + 向vector添加元素的方法为调用其push\_back函数，表示将元素添加至其尾部：

|  |
| --- |
| std::vector<std::wstring> v3;  for (std::size\_t i = O; i < 10; i++)  {  std::wstringstream wss;  WSS << TEXT(”String[") << i << TEXT(”]”);  v3.push\_back(wss.str());  } |

##### Vector（4）

* 判断vector是否为空、获取vector大小
  + 如果要判断vector是否为空则调用empty()函数
  + 如果要获取vector大小则调用size()函数

|  |
| --- |
| std::vector<std::wstring> v3;  bool isEmpty = v3.empty();  int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  std::vector<int> v(array, array + 10);  std::size vSize = v.size(); |

##### Vector（5）

* 访问vector中元素
  + 要访问vector中的元素，有两种方法：
    - 调用vector::at();
    - 调用vector::operator[]
* 两者的区别在于：
  + operator[]提供了类似数组的存取方式，但不做边界检查，可能越界，但访问效率更高
  + at()进行边界检查，如果访问越界则抛出exception，但访问效率不如operator[]

##### Vector（6）

* 访问vector中元素（续）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| std::vector<std::wstring> v;  v.reserve(10);   |  |  |  |  | | --- | --- | --- | --- | | String[0] | String[1] | String[2] | ????? |   for (std::size\_t i = 0; i < 3; i++) {  std::wstringstream wss;  WSS << TEXT(" String[") << i << TEXT("] ");  v.push\_back(wss.str0);   |  | | --- | | 预留了10个wstring空间，  push\_back了3个，v[5]访问越界 |   }  try {  std::wstring wszl = vl[5] ; // not bounds checked - will not throw  std::wstring wsz2 = v.at(5); //11 bounds checked - will throw if out of range  }  catch (const std::exception& ex) {  Console::WriteLine(ex.wha());  } |

##### Vector（7）

* 删除-vector中元素
  + clear:清除整个vector
  + pop\_back:弹出vector尾部元素
  + erase:删除vector中某一位置的元素
    - 用法一：指定iterator出删除某一元素

|  |
| --- |
| std::vector<int>::const\_iterator it = v.begin();  v.erase(it + 1); 11 erase the second element in the vector |

* 用法二：通过某一条件函数找到vector中需要删除的元素。所谓条件函数是一个按照用户定义的条件返回true/false的函数对象。我们以remove\_if为例说明：

##### Vector（8）

* 删除vector中元素（续）
  + 假设一个vector由下列元素构成，我们的目标是要删除vector中所有含有C++的字符串的元素：

|  |
| --- |
| std::vector<std::wstring> v;  v.push\_back(TEXT("Standard Template Library”));  v.push\_back(TEXT("The C++ Programming Languate");  v.push\_back(TEXT("Windows Internals”));  v.push\_back(TEXT("Prog ramming Applications for Windows”));  v.push\_back(TEXT("Design Patterns“));  v.push\_back(TEXT(" Effective C++");  v.push\_back(TEXT("More Effective C++ "); |

##### Vector （9）

* 删除vector中元素（续）
  + remove.if函数定义在algorithm中，故需include <algorithm>
  + 定义筛选器：一个一元函数对象(unary\_fu nction)，关键在于重载operator()

|  |
| --- |
| struct ContainsString : public std::unary\_function<std::wstring, bool>  {  ContainsString(const std::wstring& wszMatch) : m\_wszMatch(wszMatch) { }  bool operator ()(const std::wstring &wszStringToMatch) const  {  return (wszStringToMatch.find(m\_wszMatch) != -1);  }  std::wstring m\_wszMatch;  } |

##### Vector（10）

* 删除vector中元素（续）
  + 在erase函数中调用remove\_if执行删除：

|  |  |
| --- | --- |
| v.erase(std::remove\_if(  v.begin (),  v.end ),  ContainsString(L¨C++“)  ), v.end()); |  |

* remove\_if是不是真正remove了vector中的元素呢？

##### Vector（11）

* 删除vector中元素（续）
  + remove.if其实真正的做的是针对ContainsString条件对给出了erase函数需要操作的iterator位置，如下图所示：

|  |  |
| --- | --- |
| 0 | Standard Template Library |
| 1 | The C++ Programming Languate |
| 2 | Windows Internals |
| 3 | Programming Applications for Windows |
| 4 | Design Patterns |
| 5 | Effective C++ |
| 6 | More Effective C++ |

|  |  |
| --- | --- |
| 0 | Standard Template Library |
| 1 | Windows Internals |
| 2 | Programming Applications for Windows |
| 3 | Design Patterns |
| 4 | ???????? |
| 5 | ???????? |
| 6 | ???????? |

remove\_if

|  |  |
| --- | --- |
| before | after |

#### Deque

##### Deque（1）

* 概述
  + Deque是一个能够存放任意型别的**双向队列**
  + Deque提供的函数与vector类似，新增了两个函数：
    - push\_front:在头部插入一个元素
    - pop\_front:在头部弹出一个元素
  + Deque采用了与vector不同内存管理方法：大块分配内存
  + 为了使用deque，必须用include指令包含如下文件，并通过std命名空间去访问：

|  |
| --- |
| #include <deque>  int main() {  std::deque dq;  } |

#### List

##### list（1）

* 概述
  + List是一个能够存放任意型别的双向链表(doubly Iinked list)

|  |
| --- |
|  |

* 可以向List中接入一个子链表(sub-list)
* -为了使用List，必须用include指令包含如下文件，并通过std命名空间去访问：

|  |
| --- |
| #include <list>  int main() {  std::list templist;  } |

##### list（2）

* Iist的优势
  + Iist的优势在于其弹性(scala bility)，可随意插入和删除元素，所需之操作仅是改变下一节点中的前项(Previous)和后项(Next)的链接
  + 对于插入、删除和替换等需要重排序列的操作，效率极高
  + 对于移动元素到另一个list、把一个排好序的list合并到另一个Iist之操作，

实际上之改变list节点间的链接，没有发生元素复制

* Iist的劣势
  + 只能以连续的方式存取Iist中的元素一查找任意元素的平均时间和Iist的长度成线型比例
  + 对于查找、随机存取等元素定位操作，效率低在每个元素节点上增加一些较为严重的开销：即每个节点的前向和后向指针

##### list（3）

* 创建List

|  |  |
| --- | --- |
| 常用方式 | 代码 |
| 创建一个T型别的空list | std::list<T> l; |
| 创建一个容量是n的T型别的I．st | std::list<T> l(n); |
| 创建一个容量是n的T型别的I．st，并且都初始化为x | std::list <T> l(nl x); |
| 创建一个已有I．st的拷贝 | std::list<T> copyOfList(l); |
| 通过一个数组创建一个I．st | std::wstring array[ ] = {  TEXT("Str-1"), TEXT("Str-2"), TEXT("Str-3")  };  std ::list < std ::wstring > l(a rray, array + 3); |

##### list（4）

* 向list添加元素
  + 向list添加元素的方法为调用其push\_back、push\_front函数，表示将元素添加至其尾部或头部：

|  |
| --- |
| std::list<std::wstring> l;  l. push\_back(TEXT("Some text pushed at back"));  l.push\_front(TEXT("Some text pushed at front")); |

##### list（5）

* 判断Iist是否为空、获取vector大小
  + 如果要判断list是否为空则调用empty()函数
  + 如果要获取Iist大小则调用size()函数

|  |
| --- |
| std::list< std::wstring > l;  std::wstring array[] = {TEXT("Str-l"), TEXT("Str-2"), TEXT("Str-3") };  std::list< std ::wstring > l(array, array+3);  std::size listSize = l.size(); |

##### list（6）

* 删除list中元素
  + clear:清除整个list，内部是调用erase(begin(), end())
  + pop\_back:弹出list尾部元素

|  |
| --- |
| std::wstring array[]=(TEXT("Str-1"), TEXT("Str-2"), TEXT("Str-3"));  std::list<std::wstring>l(array, array+3)j  l.pop\_back() //”Str-3" is removed |

* + pop\_front:弹出list头部元素

|  |
| --- |
| std::wstring array[]={TEXT("Str-1"), TEXT("Str-2"), TEXT("Str-3"));  std::list<std::wstring>l(array, array+3);  l.pop\_front();//”Str-l" is removed |

* + remove:删除list中指定的元素

|  |
| --- |
| std::wstring array[] = { TEXT("Str-1"), TEXT("Str-2"), TEXT("Str-3") };  std::list< std ::wstring > l(array, array+3);  l.remove (TEXT("Str- 2 ")); // "Str-2" is removed |

##### list（7）

* 删除list中元素（续）
  + remove\_if:通过某一条件函数找到list中需要删除的元素。例如，假设一个list由下列元素构成，我们的目标是要删除list中所有含有”C++”的字符串的元素：

|  |
| --- |
| std::list< std::wstring > l;  l.push\_back(TEXT("Standard Template Library”))；  l.push\_back(TEXT("The C++ Programming Languate”));  l.push\_back(TEXT("Windows Internals”));  l.push\_back(TEXT("Programming Applications for Windows”));  l.push\_back(TEXT("Design Patterns”));  l.push\_back(TEXT(" Effective C++ "));  l.push\_back(TEXT("More Effective C++")); |

##### list（8）

* 删除list中元素（续）
  + 我们还需要定义条件函数对象ContainsString：

|  |
| --- |
| struct ContainsString : public std::unary\_function<std::wstring, bool> {  ContainsString(const std::wstring& wszMatch) : m\_wszMatch(wszMatch) { }  bool operator()(const std::wstring& wszStringToMatch) const {  return (wszStringToMatch.find(m\_wszMatch) 1= -1);  }  std::wstring m\_wszMatch;  }; |

##### list（9）

* 删除list中元素（续）
  + 调用remove\_if:

|  |
| --- |
| // remove string that contains "C++"  l.remove\_if(ContainsString(TEXT(“C++”))); |

* erase:删除list中某一位置的元素
  + 用法一：指定iterator出删除某一元素

|  |
| --- |
| std ::list< std::wstring > ::const\_iterator it = l.begin();  l.erase(it); // erase the front element in the list |

* 用法二：通过某一条件函数找到list中需要删除的元素。所谓条件函数是一个按照用户定义的条件返回true/false的函数对象。用法与remove\_if类似：

|  |
| --- |
| l.erase(std ::remove\_if(l.begin(),l.end(),ContainsString(L”C++")); |

##### list（10）

* 向Iist中插入元素
  + lnsert:list中某一位置插入的元素

|  |
| --- |
| std::list<std ::wstring>::const\_iterator it = l.begin ();  l.insert(it, anotherList.begin(), anotherList.end()); |

##### list（11）

* 粘接list
  + splice实现list粘接的功能，即将一个list的部分元素或全部元素删除，拼插入到另一个list

|  |  |
| --- | --- |
| std::list< std::wstring > list2 | |
| 0 | [Str-1] |
| 1 | [Str-2] |
| 2 | [Str-3] |

* + 假设现在有两个list:

|  |  |
| --- | --- |
| std::list<std::wstring > list1 | |
| 0 | Standard Template Library |
| 1 | The C++ Programming Language |
| 2 | Windows Internals |
| 3 | Programming Applications for Windows |
| 4 | Design Patterns |
| 5 | Effective C++ |
| 6 | More Effective C++ |

##### list（12）

* 粘接list（续）
  + 将list2粘接到list1头部，同时list2被清空：

|  |  |
| --- | --- |
| std::list<std::wstring>::const\_iterator itl = listl.begin();  listl.splice(itl, list2); |  |

|  |  |
| --- | --- |
| std::list<std::wstring > list1 | |
| 0 | Standard Template Library |
| 1 | The C++ Programming Language |
| 2 | Windows Internals |
| 3 | Programming Applications for Windows |
| 4 | Design Patterns |
| 5 | Effective C++ |
| 6 | More Effective C++ |

|  |
| --- |
| [Str-1] |
| [Str-2] |
| [Str-3] |

|  |
| --- |
| *it 1* |

|  |
| --- |
| std::list< std::wstring > list2 |
| <empty> |

##### list（13）

* 粘接list（续）
  + 将it1指向” The C++ Programming Language”，将此字符串粘接到Iist2:

|  |
| --- |
| it1++; //move the iterator forward or: std::advance(itl, 1);  list2.splice(list2.begin(),list1, it1); |

字符串”The C++Programming Language”添加到list2头部，同时从list1中it1指向的位置删除

|  |
| --- |
| std::list<std::wstring > list1 |
| [Str-1] |
| [Str-2] |
| [Str-3] |
| Standard Template Library |
| The C++ Programming Language |
| Windows Internals |
| Programming Applications for Windows |
| Design Patterns |
| Effective C++ |
| More Effective C++ |

|  |
| --- |
| *it 1* |

|  |
| --- |
| *it 1++* |

|  |
| --- |
| std::list<std::wstring > list2 |
| The C++ Programming Language |

##### list（14）

* 粘接list（续）
  + 把list1开头的三个字符”[Str-1]”、 ”[Str-2]”、 ”[Str-3]”粘到其余字符串之后：

|  |
| --- |
| itl=list1.begin();  std::advance(it1,3);  list1.splice(listl.begin(), list1, it1, list1.end()); |

|  |  |
| --- | --- |
|  | it1以及list1.end()框定了需要splice的数据的范围listl.end() |

|  |
| --- |
| std::list<std::wstring > list1 |
| [Str-1] |
| [Str-2] |
| [Str-3] |
| Standard Template Library |
| The C++ Programming Language |
| Windows Internals |
| Programming Applications for Windows |
| Design Patterns |
| Effective C++ |
| More Effective C++ |

|  |
| --- |
| *it 1* |

|  |
| --- |
| *list.begin()* |

|  |
| --- |
| *advance(it1,3)* |

|  |
| --- |
| *list.end()* |

#### Stack

##### Stack（1）

* 概述
  + Stack是一种先进后出(First In Last Out)的数据结构，只有一个出口
  + Stack支持的操作：增加元素(push)、移除元素(pop)、获取最顶端元素(top)
  + 只能访问Stack的顶层元素，不允许遍历
  + 欲使用Stack，必须包含<stack>头文件

|  |  |
| --- | --- |
|  | #include <stack>  int main() {  std::stack s;  } |

##### Stack(2)

* Stack的底层数据结构
  + 杏看Stack定义可知，STL Stack是以deque作为默认的底层结构：

|  |
| --- |
| template <class \_Ty, class \_Container=deque<\_Ty> >  class stack {  ...  } |

* 可以看做是对deque的“包装”或者“适配”，它修改了deque的接口，并提供top()、pop()、push()接口
* 因为stack不允许遍历，故其没有iterator

##### Stack(3)

* 以List作为Stack的底层数据结构
  + 除了deque，还可以list作为stack的底层结构：

|  |
| --- |
| std::stack<int, std::list<int> > s;  s.push(l);  s.push(2);  std::wcout << s.size() << std::endl;  std::wcout << s.top() << std::endl;  std::wcout << s.top() << std::endl; |

### Queue

##### Queue(1)

* 概述
  + Queue是一种先进先出(First In First Out)的数据结构，有两个出口
  + Queue支持的操作：增加元素(push)、移除元素(pop)、获取最前端元素(front)、获取最后的元素(back)
  + 只能访问queue的最前或最后元素，不允许遍历
  + 欲使用queue，必须包含<queue>头文件

|  |  |
| --- | --- |
|  | #include <queue>  int main() {  std::queue q;  } |

##### Queue(2)

* Queue的底层数据结构
  + 查看queue定义可知，STL queue是以deque作为默认的底层结构：

|  |
| --- |
| template<class \_Ty, class \_Container=deque<\_Ty> >  class queue {  ...  } |

* 可以看做是对deque的“包装”或者“适配”，它修改了deque的接口，并提供front0、back()、pop()、push()接口
* 因为queue不允许遍历，故其没有iterator

##### Queue(3)

* 以list作为Stack的底层数据结构
  + 除了deque，还可以list作为queue的底层结构：

|  |
| --- |
| std::queue<int, std::list<int> > q;  q.push(i);  q.push(2);  std::wcout << s.size() << std::endl;  std::wcout << s.front() << std::endl;  std::wcout << s.front() << std::endl;  s.pop();  std::wcout << s.front() << std::endl; |

#### Map and Multimap

##### Map(1)

* 概述
  + map是一种关联容器，存储的对象是Key/Value pair
  + 不允许有重复的Key
  + map存储的对象必须是具备可排序性的

|  |
| --- |
| template<class \_Kty, class \_Ty, class \_Pr=less<\_Kty>,  class \_Alloc=allocator<pair<const \_Kty, \_Ty> > >  class map{…) |

* 默认采用less定义排序行为
* 可以自定义排序行为（通过仿函数）
* 为了使用map，必须用include指令包含<map>文件，并通过std命名空间去访问

##### Map(2)

* map初始化

|  |  |
| --- | --- |
| struct Employee {  Employee0 { }  Employee(const std::wstring& wszName) : Name(wszName) { }  std::wstring Name;  };  struct Reverseld :public std::binary\_function<int, int, bool> {  **bool operator() (const int& keyl, const int& key2) const {**  **return (key1 <= key2) ? false : true;**  **}**   |  | | --- | | 意味着将通过Reverseld对象作为排序行为，  具体调用operator()实现 |   }  const int size = 3;  const std::pair<int, Employee> items[size] = {  std::make\_pair(1, Employee(L”Tom”));  std::make\_pair(2, Employee(L”Jerry”));  std::make\_pair(1, Employee(L”Alice”));  }  std::map<int, Employee, **ReverseId**> map1(items, items+3); |

##### Map(3)

* 插入元素

|  |
| --- |
| mapl.insert(std::make\_pair(4, Employee(L'Brown”)));  mapl[5] = Employee(L'Fisher”); |

* 删除元素

|  |
| --- |
| std::map<Person, PersonldComparer>::iterator it = mapl.begin();  mapl.erase(it); |

* operator[]存取元素

|  |
| --- |
| Employee& e = mapl[2]  e.setName(L”...”); |

##### Multimap

* 类似map的关联容器
* 允许Key重复

|  |  |
| --- | --- |
| std::multimap<int, Employee, Reverseld> mm(items, items + 3);  mm.insert(std::make\_pair(1,Employee(L”Peter"));  printf("key(1) count: %d\n", mm.count(1)); |  |

#### Set and Multiset

##### Set(1)

* 概述
* set是一种关联容器，存储的对象本身既是Key又是Value
* 不允许有重复的Key
* set存储的对象必须是具备可排序性的

|  |
| --- |
| template<class \_Kty, class \_Pr=less<\_Kty>, class \_Alloc=allocator<\_Kty>>  class set{  ) |

* 默认采用less定义排序行为，存储对象必须具备operator<行为
* 可以自定义排序行为（通过仿函数）
* 为了使用set，必须用include指令包含<set>文件，并通过std命名空间去访问

##### Set(2)

* set初始化

|  |
| --- |
| class Person {  public:  Person(const std::wstring& wszName, const std::size\_t nld) {...}  const std::wstring& GetNameo const { ... }  const std::size\_t Getldo const { ... }  };  struct PersonldComparer : public std::binary\_function<Person, Person, bool> {  bool operator()(const Person& pl, const Person& p2) const {  return (pl.Getld() < p2.Getld()) ? true : false;  }  struct PersonNameComparer : public std::binary\_function< Person, Person, bool> {  const bool operator()(const Person& pl, const Person& p2) const {  return (pl.GetName() < p2.GetName()) ? true : false;  )  const std : :size\_t nSize = 3;  const Person personArray[nSize] = {  Person(L"Tom", 1),  Person ( L"Jason", 2),  Person(L"Alice", 3)  };  std::set<Person, PersonIdComparer> ps1(personArray, personArray+nSize); |

##### Set(3)

* 插入元素

|  |
| --- |
| psl.insert(Person(L"Bill", 4)); |

* 删除元素

|  |
| --- |
| std::set< Person, PersonldComparer>:;iterator it = psl.begin();  std::advance(it, 1);  psl.erase(it); |

##### Set(4)

* set相关算法
  + set\_union

|  |
| --- |
| std::set< Person, PersonldComparer> dest;  std::insert\_iterator<std::set<Person, PersonldComparer> ii(dest, dest.begin());  std::set\_union(psl.begin(), psl.end(),ps2.begin(),ps2.end(),ii, PersonldComparer(); |
|  |

##### Set(5)

* set相关算法（续）
  + set\_intersection

|  |
| --- |
| std::set< Person, PersonldComparer> dest;  std::insert\_iterator<std::set<Person, PersonldComparer) ii(dest, dest.begin();  std::set\_intersection(psl.begin(), psl.end(), ps3.begin(), ps3.end(), ii, PersonldComparer()); |
|  |

##### Set(6)

* set相关算法（续）
  + set\_difference

|  |
| --- |
| std::set< Person, PersonldComparer> dest;  std::insert\_iterator<std::set< Person, PersonldComparer>> ii(dest, dest.begin());  std::set\_difference(psl.begin(),psl.end(), ps3.begin(), ps3.end (), ii, PersonldComparer()); |
| 注意set\_difference的取法是：包含在(first1，last1)中而不包含在(first2，last2)中的不同的元素 |
|  |

##### Set(7)

* set需特别注意的问题
  + 用于排序的成员（比如Person对象的ld，也就是“**真正的Key**”）不可改变
  + 除了“真正的Key”，其他成员可以改变但需要特殊手段
    - 比如，要将ps1中名称为”BiII"、ld为”4"的Person对象的名称重新设置为”BiII Gates"：

|  |
| --- |
| std::set<Person, PersonldComparer> ::iterator it = psl.find(Person(L”Bill”, 4));  if(it != psl.end())  it->SetName(L"Bill Gates");  **//上述代码无法通过编译！set的实现方式不允许通过迭代器改变对象成员！** |

##### Set(8)

* set需特别注意的问题（续）
  + 要改变对象的成员(非真正的Key)，需通过以下手段实现：

|  |
| --- |
| std::set< Person, PersonldComparer>::iterator it = psl.find(Person(L”Bill”, 4));  if(it != psl.end())  const\_cast< Person &> (\*it).SetName(L"Bill Gates"); //**顺利通过编译** |

* 注意：一定要cast为对象的引用而不是对象本身，也就是说以下两种方式虽然都可以通过编译，但无法改变对象的成员：

|  |
| --- |
| static\_cast<Person>(\*it) .SetName( L" Bill Gates " ) ;  (Person) (\*it).SetName(L"Bill Gates"); |

##### Set(9)

* set需特别注意的问题（续）
  + 无法改变对象的成员的原因在于，上述代码皆等价于：

|  |
| --- |
| Person tempCopy(\*it);  tempCopy.SetName(L“Bill Gates”); |