

GeekBand 极客班

互联网人才 + 油站!

STL与泛型编程

www.geekband.com

GeekBand 极客班 互联网人才+油站：

极客班携手网易云课堂，针对热门IT互联网岗位，联合业内专家大牛，紧贴企业实际需求，量身打造精品实战课程。

专业课程

+ 项目碾压

+ 习题&辅导

- 顶尖大牛亲授
- 贴合企业实际需求
- 找对重点深挖学习

- 紧贴课程内容
- 全程实战操练
- 作品就是最好的PASS卡

- 学前导读
- 周末直播答疑
- 定期作业点评
- 多项专题辅导



Understanding the C++ Standard Template Library

— 张文杰



课程内容

Part 1 C++ 模板简介(An Introduction to C++ Template)

Part 2 泛型编程(Generic Programming)

Part 3 容器(Containers)

Part 4 一些进阶问题(Some Advanced Topics)

Part 1 C++ 模板简介

- C++ 模板概观(Overview)
- C++ 函数模板(Function Template)
- C++ 类模板(Class Template)
- C++ 操作符重载(Operator Overloading)

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，而无需为每一个型别都写一份代码
- 一个简单的例子：
 - 如果要写一个取两个数中较大值的函数Max，在不使用模板的情况下，我们不得不针对不同的型别(比如int, long, char)提供每一种型别的重载：

```
int Max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```
long Max(long a, long b)
{
    return (a > b) ? a : b;
}
```

```
char 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，从而将函数原型缩减到非常简洁的表达：

```
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(classes 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建出一个真正的函数或类的过程，比如：

```
template <typename T> struct Object { ... };
```

可以用来构建诸如Object<int>, Object<char>, Object<int*>, Object<MyClass*>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-在首次使用时根据具体情况使用一种合适的型别进行实例化

Part 1 C++ 模板简介

- C++ 模板概观(Overview)
- C++ 函数模板(Function Template)
- C++ 类模板(Class Template)
- C++ 操作符重载(Operator Overloading)

C++ 函数模板(1)

■ 什么是函数模板?

- 函数模板是参数化的一族函数(a **family** of functions)
- 通过函数模板, 可以定义一系列函数, 这些函数都基于同一套代码, 但是可以作用在不同型别的参数上

```
template <typename T>
inline T Max(
    const T& a,
    const T& b)
{
    return ( a > b ) ? a : b;
}
```

- 定义函数模板
 - 定义一个函数模板, 返回两数中较大的那个, 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)
{
    ...
}
```

- 从语法上讲使用**class**和使用**typename**没有区别
- 但从语义上，**class**可能会导致误区，即只有类才能作为型别参数；而事实上**T**所表达的意思不仅仅只针对类，任何型别都可以
- 请尽量使用**typename**！
- **class**可以取代**typename**，但**struct**却不可以，以下写法语法上是错误的：

// **this is wrong!!!**

```
template <struct T> inline T Max(const T& a, const T& b) { ... }
```

C++ 函数模板(3)

■ 模板函数的使用

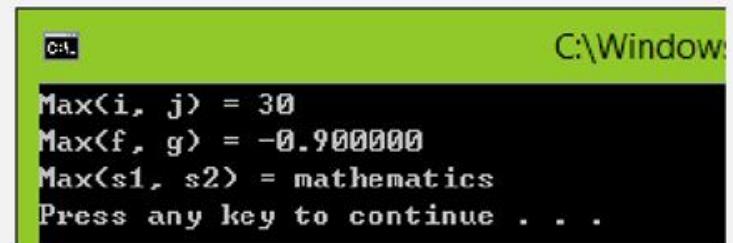
- 调用 Max，用 int, float, 以及 std::wstring 作为模板参数替换 T
- 对于不同的型别，都从模板实例化出不同的函数实体
- 但是不可以使用不同型别的参数来调用 Max，因为编译器在编译时已经知道 Max 函数需要传递的型别

```
int i = 7, j = 30;
_tprintf(TEXT("Max(i, j) = %d\n"), Max(i, j));

double f = -1.8, g = -0.9;
_tprintf(TEXT("Max(f, g) = %f\n"), Max(f, g));

std::wstring s1 = TEXT("mathematics"), s2 = TEXT("math");
_tprintf(TEXT("Max(s1, s2) = %s\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
 - 用具体型别替代模板参数T的过程叫做实例化(instantiation);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

- 1) 没有实例化之前，检查模板代码本身是否有语法错误
- 2) 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

- 模板参数是有传递给模板函数的实参决定的
- 不允许自动型别转换: 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: 第一个参数型别是int, 第二个参数型别是double
```

- 一般有两种处理这种错误的方法:

1) 用 `static_cast` 或强制转换参数型别以使两者匹配

```
Max(static_cast<double>(1), 2.0)
```

2) 显式指定T的型别

```
Max<double>(1, 2.0)
```

C++ 函数模板(7)

■ 函数模板重载

- 函数模板也可以像普通函数一样被重载
- 非模板函数可以和同名的模板函数共存
- 编译器通过函数模板参数推导来决定使用调用哪个重载

```
// 普通函数
```

```
inline int const& Max(const int const& a, const int const& b) - ①
```

```
template <typename T>
```

```
inline T const& Max(const T const& a, const T const& b) - ②
```

```
template <typename T>
```

```
inline T const& Max(const T const& a, const T const& b, const T const& c) - ③
```

C++ 函数模板(8)

■ 函数模板重载(续)

- Max(7, 42, 68): 调用接受三个参数的模板 - ③
- Max(7.0, 42.0): 调用 **Max<double>** (参数推导) - ②
- Max('a', 'b'): 调用 **Max<char>** (参数推导) - ②
- Max(7, 42): 调用 **非模板** 函数, 参数型别为 **int** - ① 其他因素都相同的情况下, 重载裁决过程调用 **非模板函数**, 而不是从模板产生实例
- Max<>(7, 42): 调用 **Max<int>** (参数推导) - ② 允许 **空模板参数列表**
- Max<double>(7, 42): 调用 **Max<double>** (无需参数推导) - ②
- Max('a', 42.7): 调用 **非模板** 函数, 参数型别为 **int** - ① 对于型别不同的参数只能调用非模板函数(**char**型别'a'和**double**型别42.7都将转化为**int**型别)

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
- 当传递模板实参的时候，函数模板依据实参的型别进行实例化
- 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

Part 1 C++ 模板简介

- C++ 模板概观(Overview)
- C++ 函数模板(Function Template)
- **C++ 类模板(Class Template)**
- C++ 操作符重载(Operator Overloading)

C++类模板(1)

- 与函数模板类似，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型别或常量(仅限int或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_nMaxSize = 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)

- 类模板的声明
 - 声明类模板与声明函数模板类似
 - 关键字**class**和**typename**都可以用，但还是倾向于使用**typename**
- 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=，那么应该使用其完整的定义(Stack<T>)，而不是省略型别T。如下面的例子所示：

```
template <typename T, std::size_t n> class Stack
{
public:
    ...
    Stack (Stack<T, n> const&);                                // copy constructor
    Stack<T>& operator= (Stack<T, n> const&); // 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() >= m_nMaxSize) {
        // error handing ...
        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_Members.pop_back();      // because pop_back of a vector removes
                               // the last element but doesn't return it!
    return 1;
}
```

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: `int element;`

```
stack.Pop(element);
```

- 获取Stack顶部元素: `stack.GetTop(element);`

注意: 此处必须有一个空格，否则编译器会认为是`>>`操作符

- Stack的Stack定义:

```
Stack<Stack<int>> intStackStack
```

```
Stack<Stack<int>> intStackStack; // ERROR: >> is not allowed
```

C++类模板(9)

■ 类模板特化(specializations)

- 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一些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，比如以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_nMaxSize = n; }
    std::size_t CurrentSize() const { return m_Members.size(); }

    void Push(const std::wstring& element);
    int Pop(std::wstring& element);
    int GetTop(std::wstring& element) const;

private:
    std::size_t m_nMaxSize;
    std::list<std::wstring> m_Members;
};
```

添加了一个新的成员函数

采用list作为Stack的内部实现，取代了主模板中用vector实现的方式

C++ 类模板(11)

■ 偏特化(Partial specialization)

- 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 - ①
```

- 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> { ... }; - ②
```

- 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> { ... }; - ③
```

- 将两个型别偏特化为指针：

```
template <typename T1, typename T2> class MyClass<T1*, T2*> { ... }; - ④
```



C++ 类模板(12)

■ 偏特化(Partial specialization)(续)

■ 使用示例：

使用	圆型
<code>MyClass<int, float> obj;</code>	<code>MyClass<T1, T2></code> - ①
<code>MyClass<float, float></code>	<code>MyClass<T, T></code> - ②
<code>MyClass<float, int></code>	<code>MyClass<T, int></code> - ③
<code>MyClass<int*, float*></code>	<code>MyClass<T1*, T2*></code> - ④

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

Usage	Prototype
<code>MyClass<int, int> obj;</code>	ERROR: matches <code>MyClass<T, T></code> and <code>MyClass<T, int></code>
<code>MyClass<int*, int*> obj;</code>	ERROR: matches <code>MyClass<T, T></code> and <code>MyClass<T1*, T2*></code>



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> >
class Stack {
    private: TContainer m_Container;
    ...
};
```

使用std::vector<>作为默认实参

- `Stack<int> intStack`: 使用默认的vector作为实参
- `Stack<std::wstring, std::list<std::wstring> > wstrStack`: 指定使用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
- 类模板可以用特定的型别特化(specialization)
- 类模板也可以用特定的型别偏特化(partial specialization)
- 类模板参数可以有默认值



Part 1 C++ 模板简介

- C++ 模板概观(Overview)
- C++ 函数模板(Function Template)
- C++ 类模板(Class Template)
- 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的不同含义
- 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，或
 - 静态全局函数(如果该全局函数需要访问类的private或protected成员，则须声明为friend成员)

```
class ComplexType {  
public:  
    // non-static member  
    ComplexType operator<(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够用作一元操作，又能够用作二元操作(比如：**&**, *****, **+**, **-**)，则可以分别被重载
 - 操作符重载不能带有默认参数值
 - 除了 **operator=**，所有其他操作符重载均可以被子类继承

Part 2 泛型编程(Generic Programming)

- 概观(Overview)
- 关联特性(Traits)
- 迭代器(Iterators)

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，等到需要调用的时候，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, Ada, ...
- C++里面的泛型是通过模板以及相关性质表现的

Part 2 泛型编程(Generic Programming)

- 概观(Overview)
- 关联特性(Traits)
- 迭代器(Iterators)

特性(Traits)(1)

■ 什么是traits以及为什么使用traits?

- 假设给定一个数组，计算数组中所有元素的和：

A[0] | A[1] | ... | A[n] $\sum_{k=0}^n A[k]$

- 我们可以很直接地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 (start != end) {  
        total += *start++;  
    }  
    return total;  
}
```

需要考虑的一点：如何构建型别为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

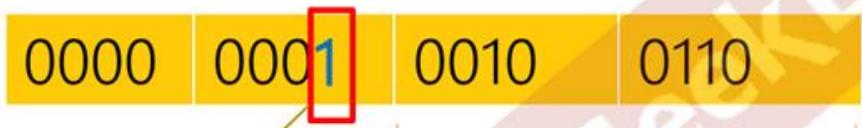
特性(Traits)(2)

■ 什么是traits以及为什么使用traits?(续)

- 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));
```

$$294 = 0x0126$$



该bit溢出
(overflowed)

Char型别能hold住的最大值是 $0xFF = 255$

- 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！

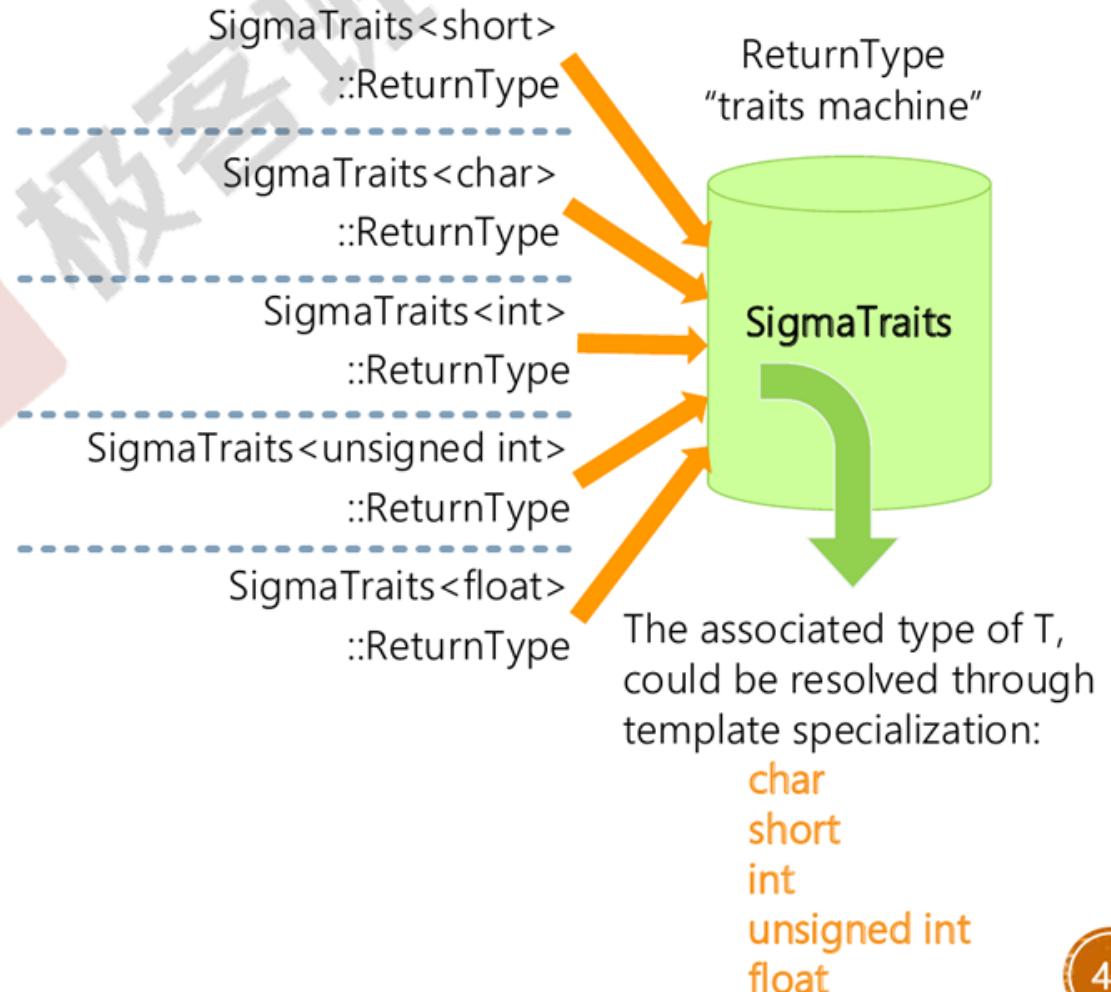
特性(Traits)(3)

- 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 type T)，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:
 $T \rightarrow \text{association} \rightarrow \text{characteristic of } T \rightarrow \text{another type} \rightarrow \text{trait!}$
 - **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**，叫做traits模板(**traits template**)

特性(Traits)(4)

■ 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;
};
```



特性(Traits)(5)

■ 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 ReturnType;
    ReturnType s = ReturnType();
    while (start != end)
        s += *start++;
    return s;
}
```

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)

Part 2 泛型编程(Generic Programming)

- 概观(Overview)
- 关联特性(Traits)
- 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(stick)在一起从而使得一种算法的实现可以运用到多种不同的容器上，如下面的例子所示，**find**算法接受一对迭代器，分别指向容器的开始位置和最终位置：

```
template<class _InIt, class _Ty>
inline _InIt find(_InIt _First, _InIt _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.begin(), v.end(), elementToFind)  
std::list<int>::iterator itl = std::find(l.begin(), l.end(), elementToFind)  
std::deque<int>::iterator it3 = std::find(d.begin(), d.end(), 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(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命名空间去访问：

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

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 = 0; 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)

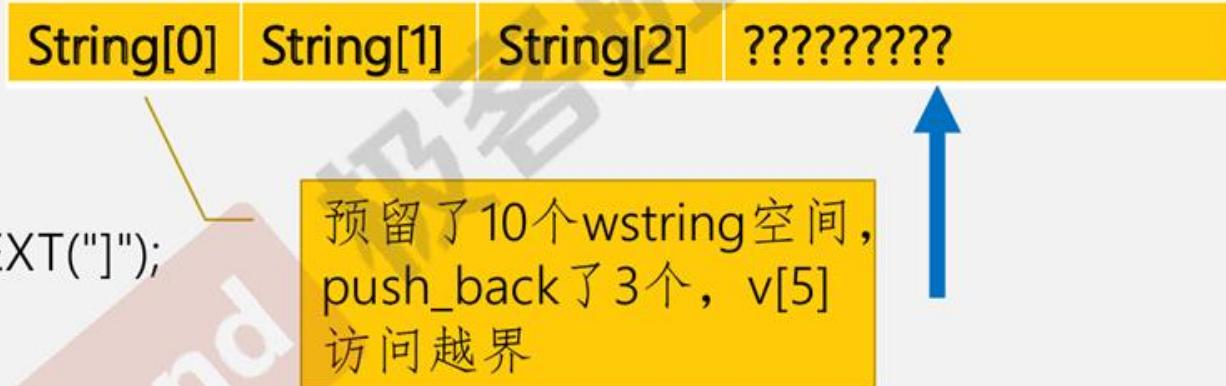
- 访问vector中元素
 - 要访问vector中的元素，有两种方法：
 - 调用vector::at()
 - 调用vector::operator[]
 - 两者的区别在于：
 - operator[]提供了类似数组的存取方式，但不做边界检查，可能越界，但访问效率更高
 - at()进行边界检查，如果访问越界则抛出exception，但访问效率不如operator[]

Vector(6)

■ 访问vector中元素(续)

```
std::vector<std::wstring> v;  
v.reserve(10);  
for (std::size_t i = 0; i < 3; i++) {  
    std::wstringstream wss;  
    wss << TEXT("String[") << i << TEXT("]");  
    v.push_back(wss.str());  
}
```

```
try {  
    std::wstring wsz1 = v[5]; // not bounds checked - will not throw  
    std::wstring wsz2 = v.at(5); // bounds checked - will throw if out of range  
}  
catch (const std::exception& ex) {  
    Console::WriteLine(ex.what());  
}
```



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); // 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("Programming 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_function)，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());
```

Standard Template Library
The C++ Programming Language
Windows Internals
Programming Applications for Windows
Design Patterns
Effective C++
More Effective C++

after **erase(remove_if):**

Standard Template Library
Windows Internals
Programming Applications for Windows
Design Patterns

- remove_if 是 不 是 真 正 remove 了 vector 中 的 元 素 呢 ?

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());
```

Standard Template Library
The C++ Programming Language
Windows Internals
Programming Applications for Windows
Design Patterns
Effective C++
More Effective C++

after `erase(remove_if):`

Standard Template Library
Windows Internals
Programming Applications for Windows
Design Patterns

- `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++

Before

remove_if()

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

After

Part 3 容器(Containers)

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

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;
}
```

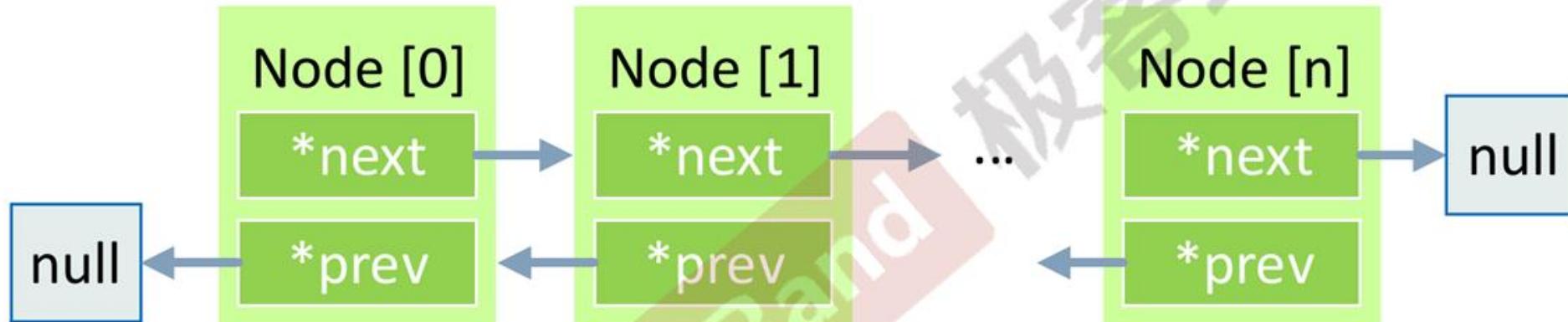
Part 3 容器(Containers)

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

List(1)

■ 概述

- List是一个能够存放任意型别的双向链表(doubly linked 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 l;
}
```

List(2)

■ List的优势

- List的优势在于其弹性(scalability)，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list之操作，实际上之改变list节点间的链接，没有发生元素复制

■ List的劣势

- 只能以连续的方式存取List中的元素-查找任意元素的平均时间和List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	<code>std::list<T> l;</code>
创建一个容量是n的T型别的list	<code>std::list<T> l(n);</code>
创建一个容量是n的T型别的list， 并且都初始化为x	<code>std::list<T> l(n, x);</code>
创建一个已有list的拷贝	<code>std::list<T> copyOfList(l);</code>
通过一个数组创建一个list	<code>std::wstring array[] = { TEXT("Str-1"), TEXT("Str-2"), TEXT("Str-3") }; std::list<std::wstring> l(array, array + 3);</code>

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list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list大小则调用size()函数

```
std::list<std::wstring> l;  
bool isEmpty = l.empty();
```

```
std::wstring array[] = {TEXT("Str-1"), 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);
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-1" 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);  
    }  
  
    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++")));
```

```
Standard Template Library  
The C++ Programming Languate  
Windows Internals  
Programming Applications for Windows  
Design Patterns  
Effective C++  
More Effective C++  
  
after remove_if():  
Standard Template Library  
Windows Internals  
Programming Applications for Windows  
Design Patterns
```

list(9)

■ 删 除 list 中 元 素 (续)

■ 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++")),
        l.end());
```

list(10)

- 向list中插入元素
 - insert: 向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**实现了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
- 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++

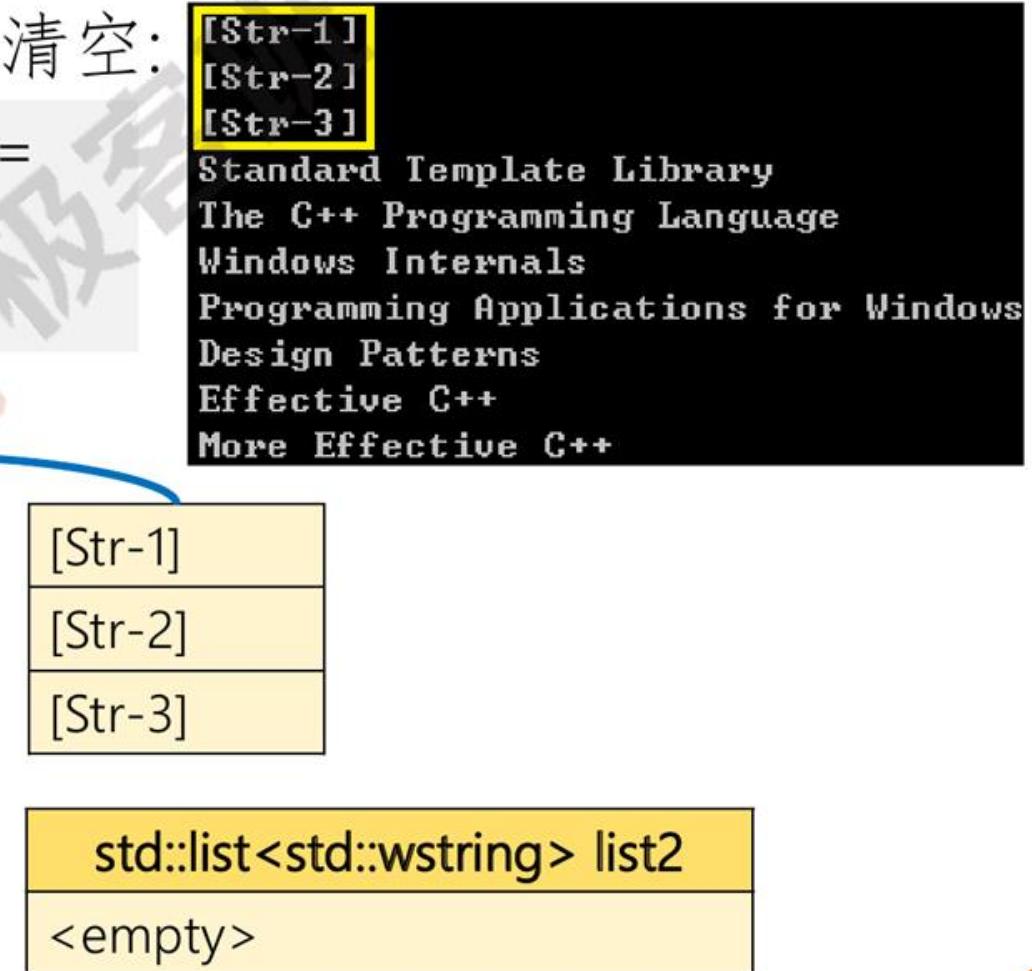
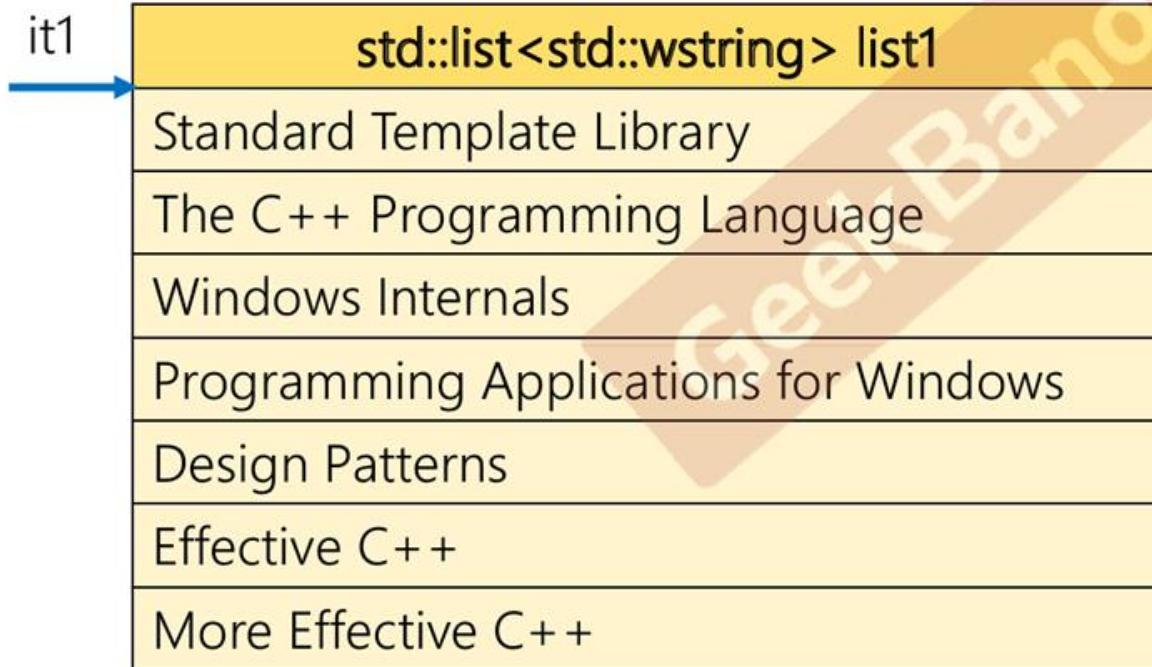
std::list<std::wstring> list2	
0	[Str-1]
1	[Str-2]
2	[Str-3]

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 it1 =  
    list1.begin();  
list1.splice(it1, list2);
```



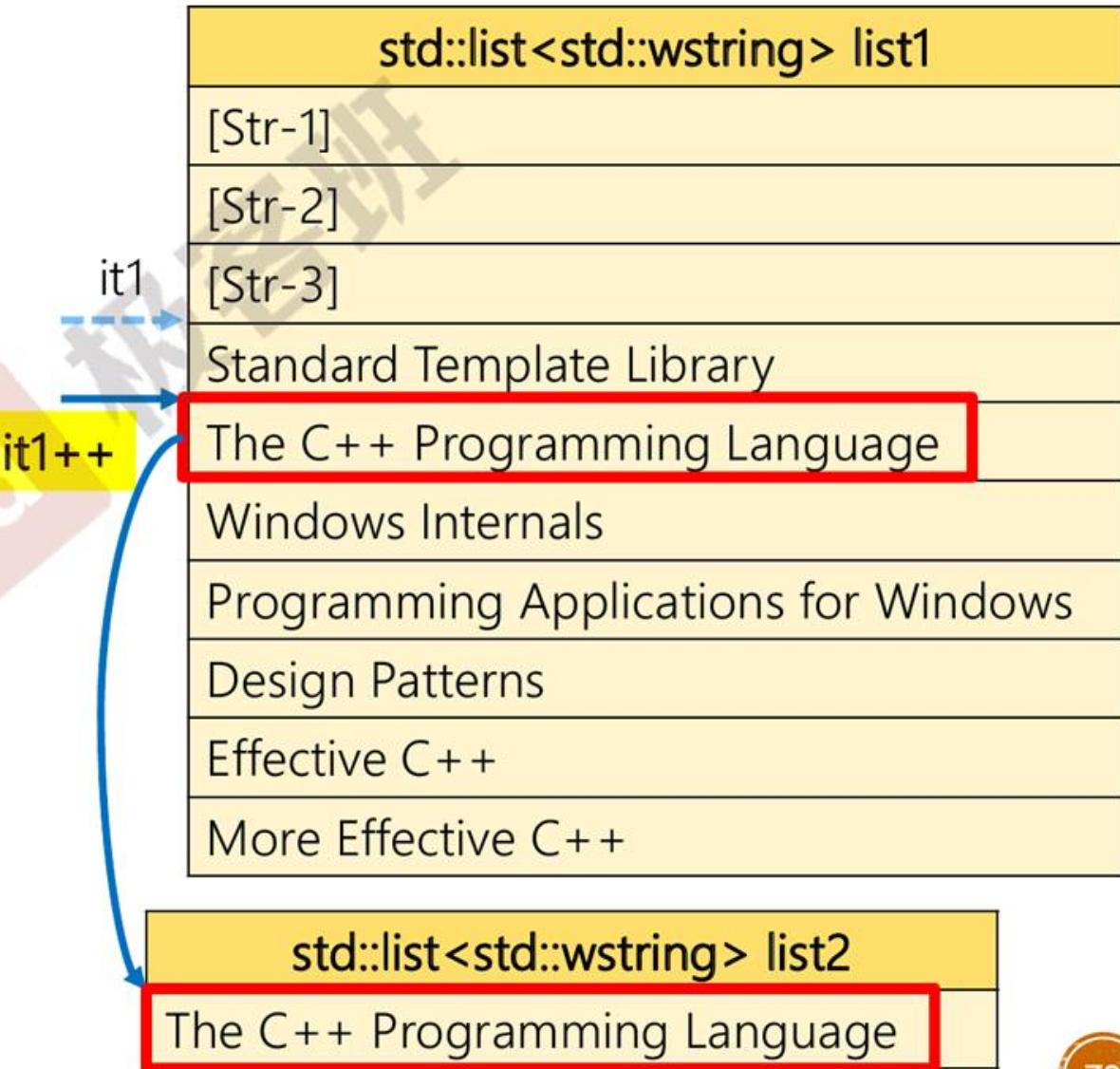
list(13)

■ 粘接list(续)

- 将it1指向"The C++ Programming Language"，并将其字符串粘接到list2：

```
// move the iterator forward  
it1++; // or: std::advance(it1, 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

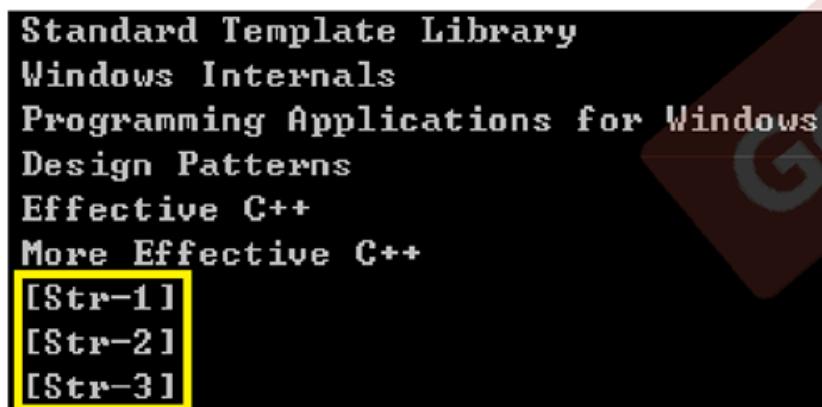


list(14)

■ 粘接list(续)

- 把list1开头的三个字符串 "[Str-1]"、"[Str-2]"、"[Str-3]" 粘到其余字符串之后：

```
it1 = list1.begin();
std::advance(it1, 3);
list1.splice(list1.begin(), list1, it1, list1.end());
```



advance(it1, 3)

list.begin()

it1

It1以及
list1.end()框
定了需要
splice的数据
的范围

list1.end()

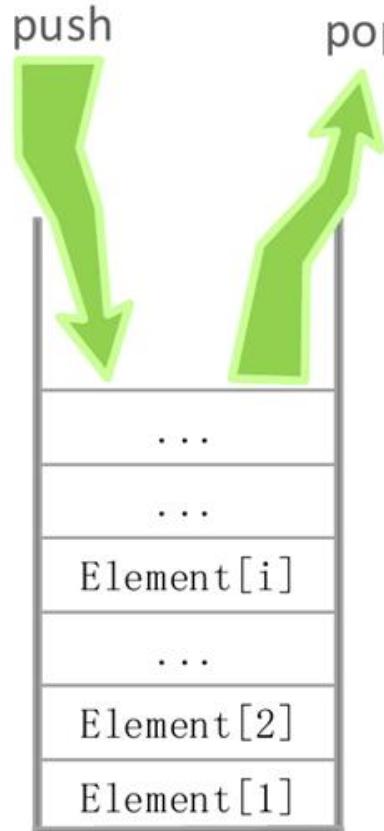
Part 3 容器(Containers)

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

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
 - 查看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(1);  
s.push(2);  
  
std::wcout << s.size() << std::endl;  
std::wcout << s.top() << std::endl;  
  
s.pop();  
std::wcout << s.top() << std::endl;
```

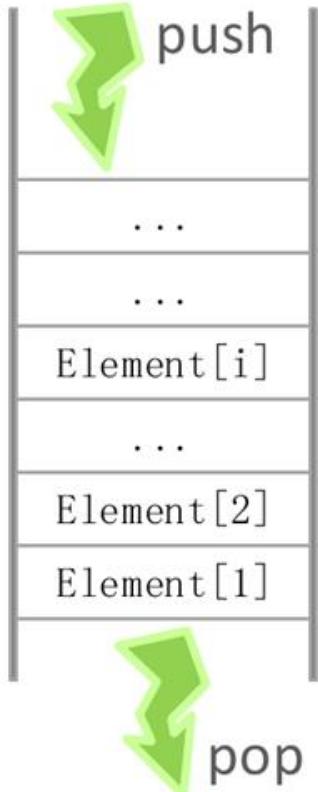
Part 3 容器(Containers)

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

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



- Stack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front()、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(1);
```

```
q.push(2);
```

```
std::wcout << s.size() << std::endl;
```

```
std::wcout << s.front() << std::endl;
```

```
s.pop();
```

```
std::wcout << s.front() << std::endl;
```

Part 3 容器(Containers)

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

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 {  
    Employee() {}  
    Employee(const std::wstring& wszName) : Name(wszName) {}  
    std::wstring Name;  
};  
  
struct ReverseId : public std::binary_function<int, int, bool> {  
    bool operator()(const int& key1, const int& key2) const {  
        return (key1 <= key2) ? false : true;  
    }  
};  
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(3, Employee(L"Alice")),  
};  
std::map<int, Employee, ReverseId> map1(items, items + 3);
```

意味着将通过
ReverseId对象作为
排序行为，具体调
用operator()实现

Map(3)

- 插入元素

```
map1.insert(std::make_pair(4, Employee(L"Brown")));
map1[5] = Employee(L"Fisher");
```

- 删 除 元 素

```
std::map<Person, PersonIdComparer>::iterator it = map1.begin();
map1.erase(it);
```

- operator[]存取元素

```
Employee& e = map1[2];
e.SetName(L"...");
...
```

Multimap

- 类似map的关联容器
- 允许Key重复

```
std::multimap<int, Employee, ReverseId> mm(items, items + 3);  
mm.insert(std::make_pair(1, Employee(L"Peter")));
```

```
printf("key(1) count: %d\n", mm.count(1));
```

key(1) count: 2

Part 3 容器(Containers)

- Vector
- Deque
- List
- Stack
- Queue
- Map and Multimap
- 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 nId) {...}  
    const std::wstring& GetName() const { ... }  
    const std::size_t GetId() const { ... }  
    ...  
};
```

```
struct PersonIdComparer : public std::binary_function<Person, Person, bool> {  
    bool operator()(const Person& p1, const Person& p2) const {  
        return (p1.GetId() < p2.GetId()) ? true : false;  
    }  
};
```

```
struct PersonNameComparer : public std::binary_function<Person, Person, bool> {  
    const bool operator()(const Person& p1, const Person& p2) const {  
        return (p1.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);
```

意味着将通过
PersonIdComparer
对象作为排序行为，
具体调用operator()
实现

Set(3)

- 插入元素

```
ps1.insert(Person(L"Bill", 4));
```

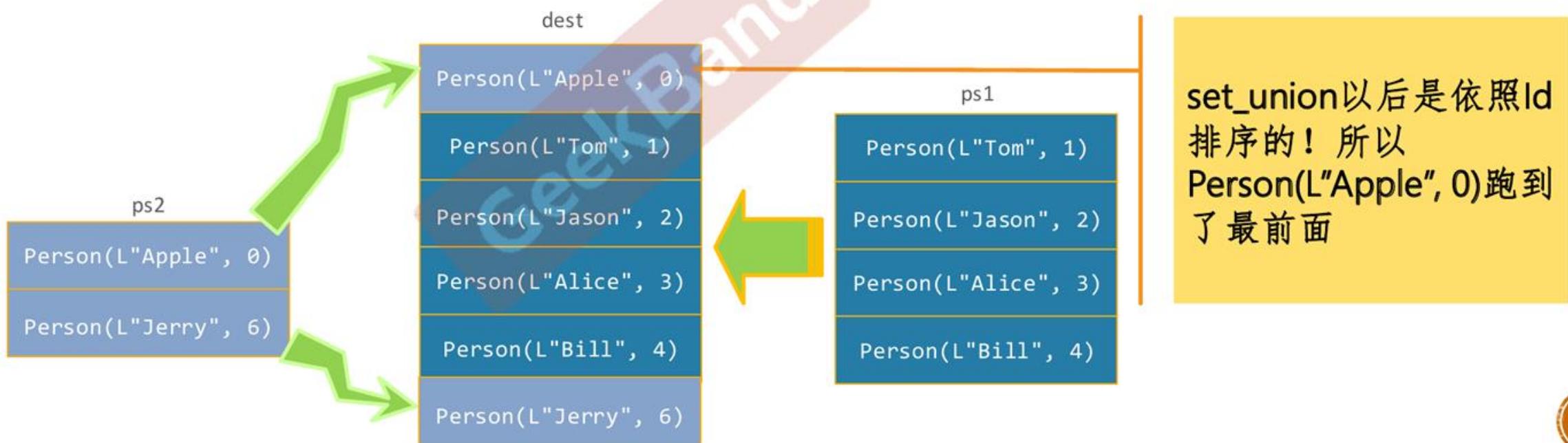
- 删除元素

```
std::set<Person, PersonIdComparer>::iterator it = ps1.begin();
std::advance(it, 1);
ps1.erase(it);
```

Set(4)

- set相关算法
 - set_union

```
std::set<Person, PersonIdComparer> dest;  
std::insert_iterator<std::set<Person, PersonIdComparer>> ii(dest, dest.begin());  
  
std::set_union(ps1.begin(), ps1.end(), ps2.begin(), ps2.end(), ii, PersonIdComparer());
```



Set(5)

- set相关算法(续)
 - set_intersection

```
std::set<Person, PersonIdComparer> dest;  
std::insert_iterator<std::set<Person, PersonIdComparer>> ii(dest, dest.begin());
```

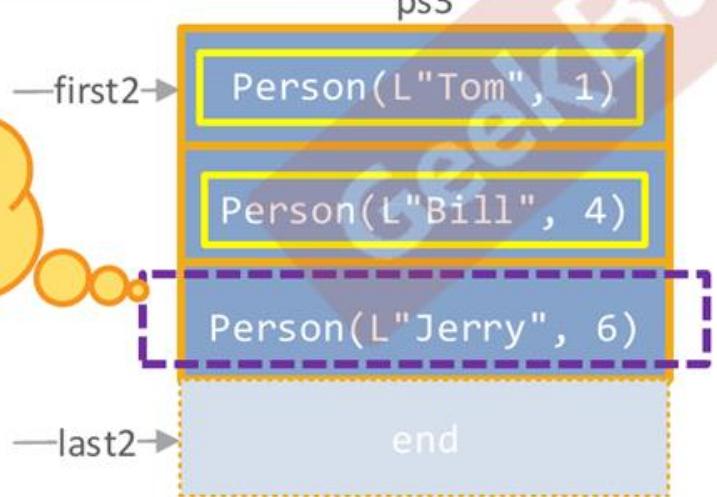
```
std::set_intersection(ps1.begin(), ps1.end(), ps3.begin(), ps3.end(), ii, PersonIdComparer());
```



Set(5)

- set相关算法(续)
 - set_difference

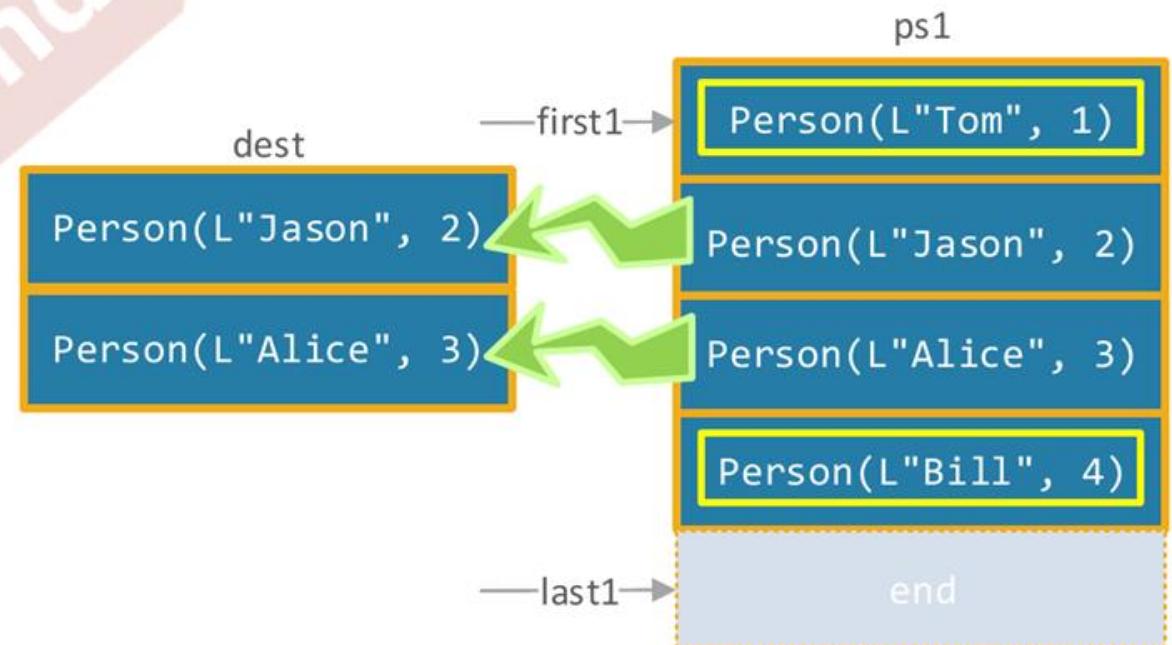
注意set_difference的取法是：包含在[first1, last1]中而不包含在[first2, last2]中的不同的元素



```
std::set<Person, PersonIdComparer> dest;
std::insert_iterator<std::set<Person, PersonIdComparer>> ii(dest,
dest.begin());
```



```
std::set_difference(ps1.begin(), ps1.end(), ps3.begin(), ps3.end(),
ii, PersonIdComparer());
```



Set(6)

■ set需特别注意的问题

- 用于排序的成员(比如Person对象的Id, 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“Bill”、Id为“4”的Person对象的名称重新设置为“Bill Gates”：

```
std::set<Person, PersonIdComparer>::iterator it = ps1.find(Person(L"Bill", 4));  
if(it != ps1.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(7)

- 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, PersonIdComparer>::iterator it = ps1.find(Person(L"Bill", 4));
if(it != ps1.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(8)

- 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：

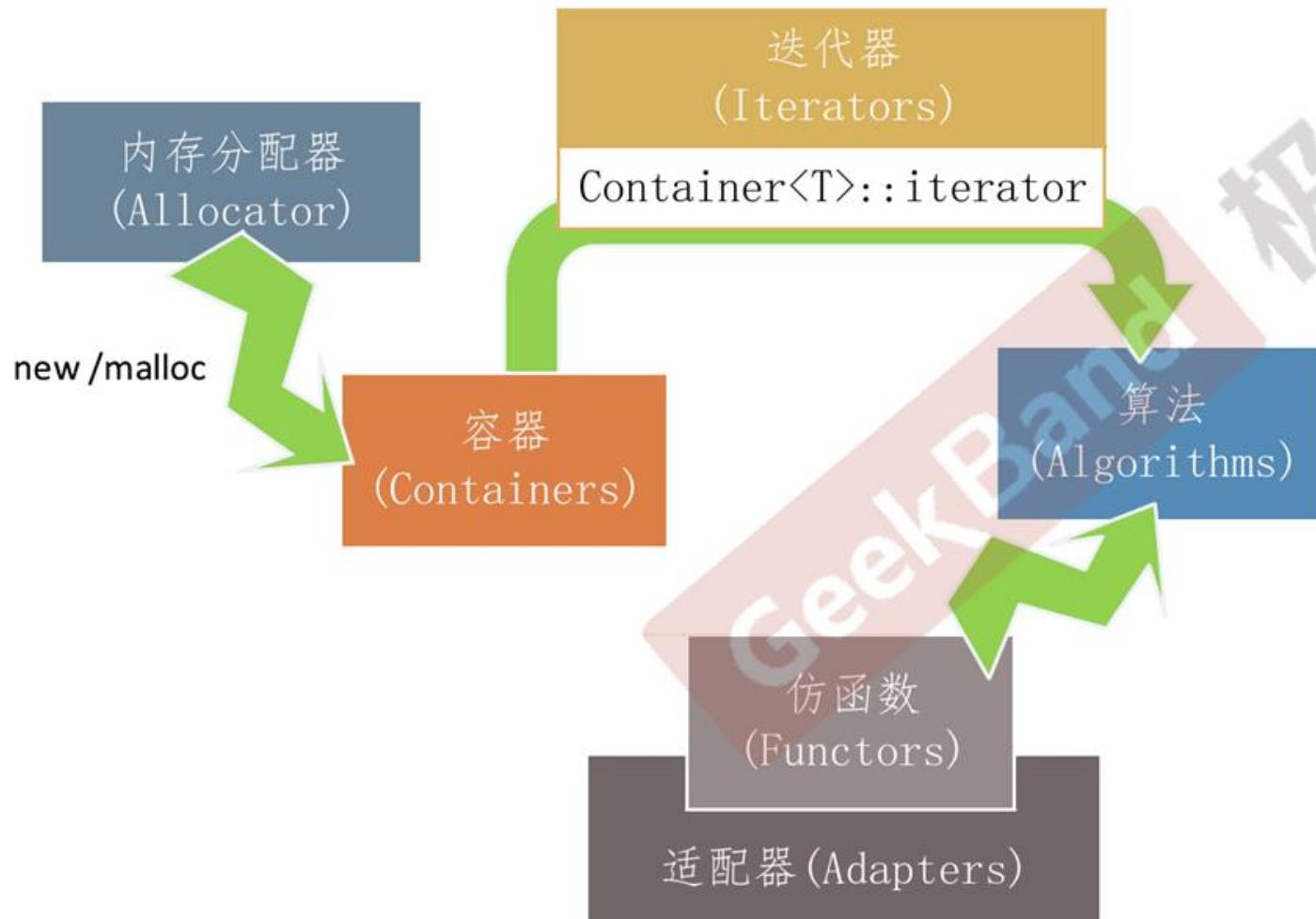
```
Person tempCopy(*it);
tempCopy.SetName(L"Bill Gates");
```

Part 4 一些进阶问题(Advanced Topics)

- STL整体结构
- 仿函数(Functor)
- 适配器(Adapters)
- 内存分配器(Allocator)
- 其他值得注意的问题

STL整体结构

■ STL组件之间的关系



- 容器通过内存分配器分配空间
- 容器和算法分离
- 算法通过迭代器访问容器
- 仿函数协助算法完成不同的策略变化
- 适配器套接仿函数



仿函数(Functors)

- 仿函数又称为函数对象(Function Object)，其作用相当于一个函数指针
- 回顾之前的remove_if，输入参数ContainsString即为仿函数

```
std::remove_if(v.begin(), v.end(), ContainsString(L"C++"))
```

- 要将某种“行为”作为算法的参数(此处的行为是：判断字符串中是否存在“C++”这几个字符)，就需要将该“行为”函数指针作为算法的参数
- STL中将这种行为函数指针定义为所谓的仿函数，其实现是一个class，再以该仿函数产生一个对象作为算法参数



仿函数(2)

■ 仿函数与算法之间的关系

```
Algorithm(Iterator first, Iterator last, ... , Functor func)
```

```
{
```

```
...
```

```
func(...)
```

```
...
```

```
}
```

STL内建仿函数，或用户自定义仿函数

```
template <typename T> class Functor
```

```
{
```

```
    void operator()(...) { }
```

```
}
```



仿函数(3)

- 仿函数的类别定义中必须重载函数调用(function call)运算子operator(), 从而使得仿函数对象可以像函数一样被调用
 - 例如STL内置的仿函数std::greater<T>:

```
#include <functional>
```

```
std::greater<int> g;
std::cout << std::boolalpha << g(10, 3) << std::endl;      // output: true
std::cout << std::greater<int>()(100, 300) << std::endl;  // output: false
```

std::boolalpha是一种所谓的流控制器(iostream manipulator), 意思是从该标志之后对bool值的输出用"true"、"false"表达, 而不是默认的1、0



仿函数(4)

- 自定义的仿函数必须重载operator()
- 回顾ContainsString对象：

```
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;
};
```



仿函数(5)

- 自定义的仿函数的另一个例子

- 打印容器中的所有元素

```
int buffer[BufferSize] = { 0, 100, -98, 31, 7 };
std::vector<int> v(buffer, buffer + BufferSize);
std::for_each(v.begin(), v.end(), PrintContainer<int>(std::cout));
```

```
template<typename T> struct PrintContainer
{
    PrintContainer(std::ostream& out) : os(out) { }
    void operator() (const T& x) { os << x << ' '; }
    std::ostream& os;
};
```



仿函数(6)

- 为什么要用仿函数而不是普通函数指针作为算法的行为参数?
 - 普通函数指针不能满足STL的抽象要求
 - 怎么定义该函数指针呢? 参数和返回性别如何指定?
 - 函数指针无法和STL其他组件交互
- 仿函数可作为模板实参用于定义对象的某种默认行为
 - 以某种顺序对元素进行排序的容器, 排序规则就是一个模板实参,比如STL标准容器std::set:

```
template<typename _Kty, typename _Pr = less<_Kty>, ...>
class set {
    ...
};
```



仿函数(7)

- Set默认以less作为元素的排序行为，如果两个set具有不同的排序规则，那么对他们的进行赋值或==判断会导致错误：

```
class Person {  
...  
};
```

```
std::set<Person, std::less<Person>> set1, set2; // operator<做为排序行为  
std::set<Person, std::greater<Person>> set3, set4; // operator>作为排序行为
```

```
if (set1 == set2) ... // 正确，相同的型别  
if (set1 == set3) ... // 错误，不同的型别！
```

```
set1 = set2 // 正确，相同的型别  
set4 = set1 // 错误，不同的型别！
```



仿函数(8)

- 如果在Person类中重载operator<，那么less函数对象将以此行为排序Person对象：

```
class Person {  
public:  
    Person(const std::wstring& wszName, const std::size_t nId) { ... }  
    const bool operator<(const Person& p) const { return (this->m_nId < p.m_nId); }  
private:  
    std::wstring m_wszName;  
    std::size_t m_nId;  
};  
  
std::set<Person, std::less<Person>> set1;  
set1.insert(Person(L"Tom", 0));  
set1.insert(Person(L"Alice", 1));  
set1.insert(Person(L"Jack", 2));  
std::for_each(set1.begin(), set1.end(), PrintContainer<Person>(std::wcout, L' '));
```

- 将按照Person的Id依次输出“Tom”、“Alice”、“Jack”

```
Tom Alice Jack  
Press any key to continue . . .
```



仿函数(9)

- 如果要改变set默认的排序行为，并提供自定义的排序行为，那么可采用自定义仿函数方式，并重载operator():
 - 如下所示PersonComparer对象将以Person的Name来进行排序

```
class PersonComparer {  
public:  
    const bool operator()(const Person& p1, const Person& p2) const {  
        return (p1.GetName() < p2.GetName()) ? true : false;  
    }  
};
```

```
std::set<Person, PersonComparer> set2;  
set2.insert(Person(L"Tom", 0));  
set2.insert(Person(L"Alice", 1));  
set2.insert(Person(L"Jack", 2));  
std::for_each(set2.begin(), set2.end(), PrintContainer<Person>(std::wcout, L' '));
```

- 将按照Person的Name依次输出“Alice”、“Jack”、“Tom”

```
Alice Jack Tom  
Press any key to continue . . .
```

仿函数适配器(Functor Adapters)

- 仿函数适配器(Functor adapter), 目的在于将无法匹配的仿函数“套接”成可以匹配的型别
- 我们以两个适配器加以说明:
 - binder1st/binder2nd
 - mem_fun/mem_fun_ref



仿函数适配器(Functor Adapters)

- binder1st/binder2nd
- mem_fun/mem_fun_ref

GeekBand



仿函数适配器(binder1st/binder2nd)

- 给定一个vector，其元素为：[0, 0, 0, 0, 0, 0, 0, 0, 10, 0]，如何通过std::not_equal_to匹配到第一个非零元素？
- 想象中，或许可以这样：

```
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(),
    std::not_equal_to<int>(0))
```

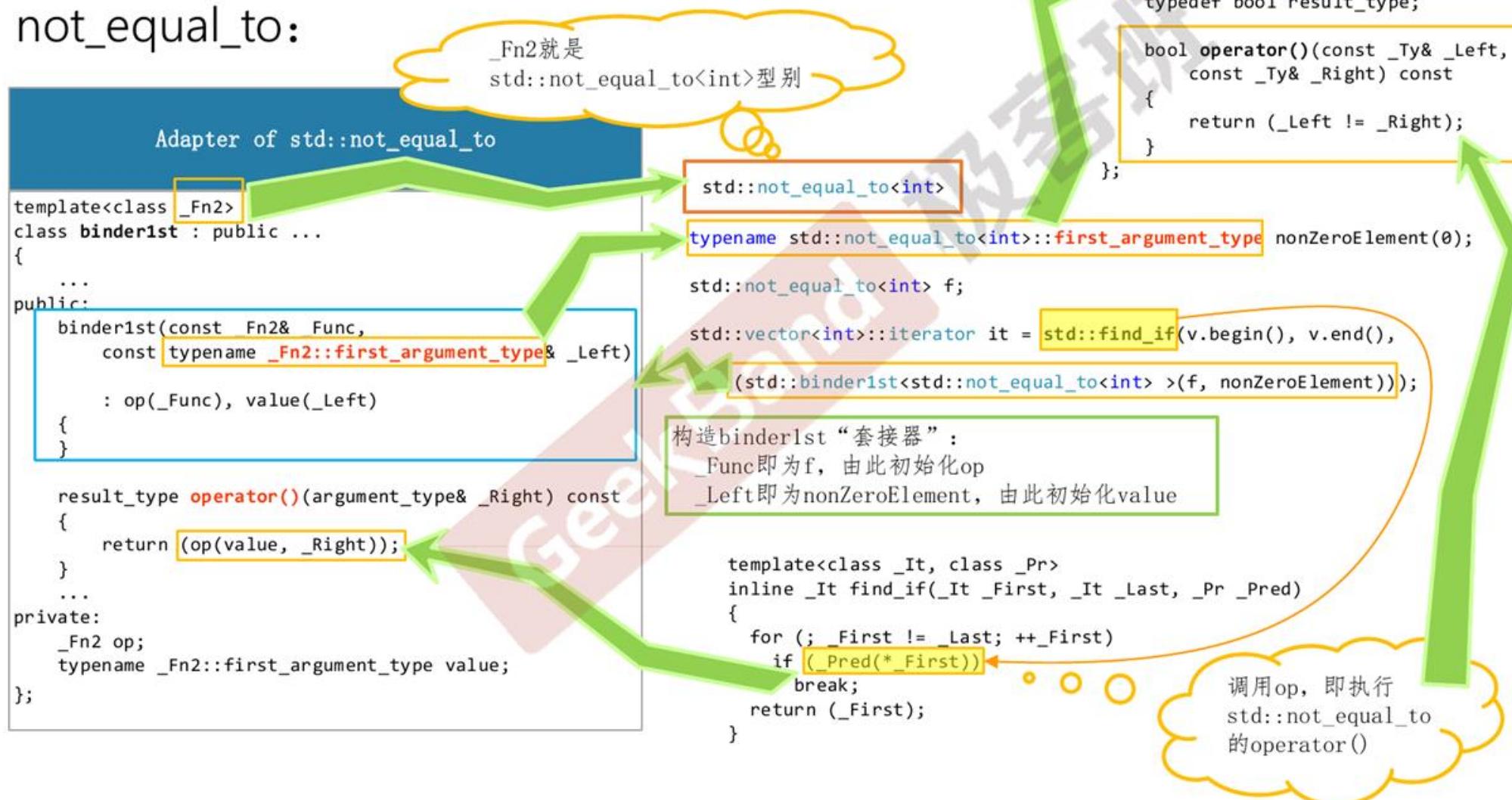
- 事实上行不通，因为：
 - not_equal_to构造函数不接受参数“0”
 - not_equal_to的operator()不接受一个参数，需要两个参数！

```
bool operator()(const _Ty& _Left, const _Ty& _Right) const
```



仿函数适配器(binder1st/binder2nd)

- 解决的办法是通过binder1st套接
not_equal_to:



仿函数适配器(binder1st/binder2nd)

- 实际使用时，应当使用更简洁的写法：

```
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(),
    std::bind1st(std::not_equal_to<int>(), 0));
```

- bind1st封装了binder1st的调用复杂性

```
template<class _Fn2, class _Ty>
inline binder1st<_Fn2> bind1st(const _Fn2& _Func, const _Ty& _Left)
{
    typename _Fn2::first_argument_type _Val(_Left);
    return (binder1st<_Fn2>(_Func, _Val));
}
```



仿函数适配器(binder1st/binder2nd)

- 类似地有bind2nd适配器，区别在于：
 - _Func操作是作用在左值还是右值上

```
template<class _Fn2, class _Ty>
inline binder1st<_Fn2> bind1st(
    const _Fn2& _Func, const _Ty& _Left) {
    typename _Fn2::first_argument_type _Val(_Left);
    return (binder1st<_Fn2>(_Func, _Val));
}
```

```
template<class _Fn2, class _Ty>
inline binder2nd<_Fn2> bind2nd(
    const _Fn2& _Func, const _Ty& _Right) {
    typename _Fn2::second_argument_type _Val(_Right);
    return (binder2nd<_Fn2>(_Func, _Val));
}
```

- 对于vector: [-10, -300, 1, 0, 0, 0, 0, 0, 10, 0]

```
std::vector<int>::iterator it =
std::find_if(
    v.begin(), v.end(),
    std::bind1st(std::less<int>(), 0));
```

两者等价

```
std::vector<int>::iterator it =
std::find_if(
    v.begin(), v.end(),
    std::bind2nd(std::greater<int>(), 0));
```



仿函数适配器(Functor Adapters)

- binder1st/binder2nd
- mem_fun/mem_fun_ref

GeekBand



仿函数适配器(mem_fun/mem_fun_ref)

- mem_fun 和 mem_fun_ref:
 - 用来适配对象的成员函数

```
class Person {  
public:  
    ...  
    void Print() {  
        std::wcout << m_nld << L": "  
            << m_wszName << std::endl;  
    }  
    ...  
private:  
    std::wstring m_wszName;  
    std::size_t m_nld;  
};
```

如果Print函数是一个全局函数，这样的方式才可行：

```
void PrintPerson (const Person* p) {  
    ...  
}
```

```
std::vector<Person*> v;  
v.push_back(new Person(L"Tom", 1));  
v.push_back(new Person(L"Jerry", 2));  
v.push_back(new Person(L" Micheal", 3));  
v.push_back(new Person(L"Jason", 4));  
v.push_back(new Person(L"Bill", 5));
```

```
std::for_each(v.begin(), v.end(), &Person::Print);  
// 遗憾的是，无法通过编译！！！
```

```
std::for_each(v.begin(), v.end(), &PrintPerson); // OK
```

仿函数适配器(mem_fun/mem_fun_ref)

- 对于函数f以及对象obj，在obj上调用f的形式可以有3中：
 - (1): f(obj) // f是全局函数(非obj成员函数)
 - (2): obj.f() // f是obj的成员函数，obj非指针
 - (3): obj->f() // f是obj的成员函数，obj是指针
- 然而，在for_each的定义中，只接受形如(1)的调用：

```
template<class _It, class _Fn1>
inline void for_each(_It _First, _It _Last, _Fn1& _Func)
{
    for (; _First != _Last; ++_First)
        _Func(*_First);
}
```



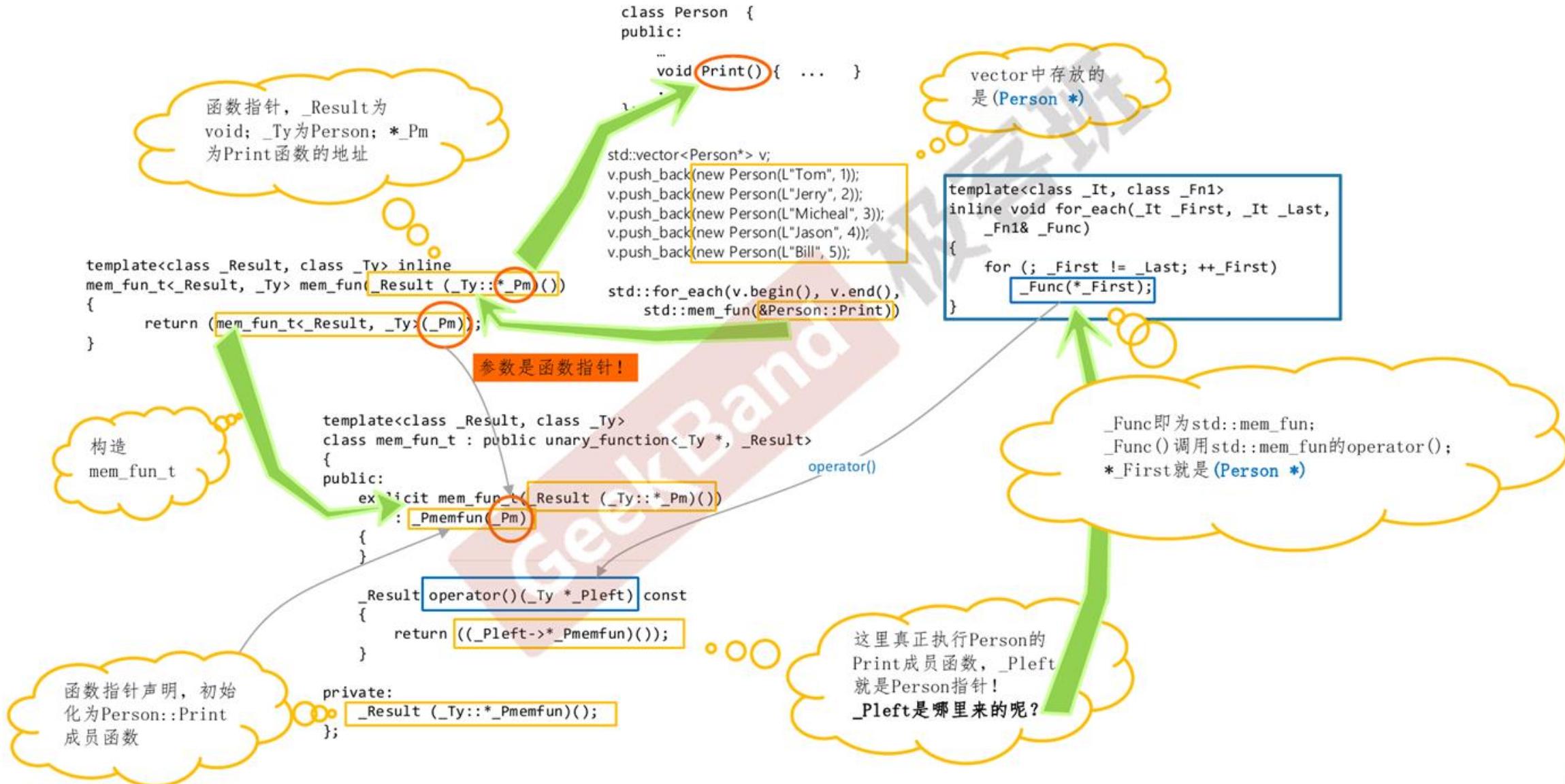
仿函数适配器(mem_fun/mem_fun_ref)

- mem_fun和mem_fun_ref存在，是要让obj的成员函数f可以被调用，并且以形如(1)的方式被调用！

```
std::vector<Person*> v;
v.push_back(new Person(L"Tom", 1));
v.push_back(new Person(L"Jerry", 2));
v.push_back(new Person(L" Micheal", 3));
v.push_back(new Person(L"Jason", 4));
v.push_back(new Person(L"Bill", 5));

std::for_each(v.begin(), v.end(), std::mem_fun(&Person::Print));
```

仿函数适配器(mem_fun/mem_fun_ref)



仿函数适配器(mem_fun/mem_fun_ref)

- 如果vector中存放的Person对象不是指针，那么可以用mem_fun_ref适配：

```
std::vector<Person> v;
v.push_back(Person(L"Tom", 1));
v.push_back(Person(L"Jerry", 2));
v.push_back(Person(L"Michéal", 3));
v.push_back(Person(L"Jason", 4));
v.push_back(Person(L"Bill", 5));

std::for_each(v.begin(), v.end(), std::mem_fun_ref(&Person::Print));
```



其他值得注意的问题

(1) std::string/std::wstring 与 vector<char>/vector<wchar_t>

- 单线程情况下涉及对字符串的操作，首选 std::string/std::wstring
- 多线程情况下需注意 string 是否带引用计数 (reference count)
 - 原本 reference count 的目的是避免不必要的内存分配和字符拷贝
 - 在多线程环境下，避免分配和拷贝所节省下的开销转嫁到了并发控制上
 - 可考虑使用 vector<char>/vector<wchar_t> 替代，而 vector 的实现是不带引用计数的

其他值得注意的问题

(2) 当new出对象并放入容器时，要在销毁容器前delete那些对象

```
std::vector<Person*> v;
v.push_back(new Person(L"Tom", 1));
v.push_back(new Person(L"Jerry", 2));
v.push_back(new Person(L"Michael", 3));
v.push_back(new Person(L"Jason", 4));
v.push_back(new Person(L"Bill", 5));
...
for (vector<Person*>::iterator it = v.begin(); it != v.end(), it++)
    delete (*it);
v.clear();
```



其他值得注意的问题

(3) 尽量用算法调用代替手写循环

- 对于(2)中的对象delete操作，使用for_each而不是for循环
- 为此需定义DeleteElement对象并重载operator()

```
struct DeleteElement {  
    template <typename TElement>  
    void operator() (const TElement* p) const {  
        delete p;  
    }  
};
```

- 使用for_each销毁容器中的对象

```
std::for_each(v.begin(), v.end(), DeleteElement());
```



其他值得注意的问题

(4) 通过swap为容器“缩水”

- 容器的size(大小)和capacity(容量)的区别

```
int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
std::vector<int> v(array, array + 10);
v.reserve(100000); // 使得容器的容量预留出100000个可以存放int的空间
::_tprintf(TEXT("vector size: %d\n"), v.size());           // size = 10
::_tprintf(TEXT("vector capacity: %d\n"), v.capacity()); // capacity = 100000
```

- 使用swap进行缩水

```
std::vector<int>(v).swap(v);
::_tprintf(TEXT("vector size: %d\n"), v.size());           // size = 10
::_tprintf(TEXT("vector capacity: %d\n"), v.capacity()); // capacity = 10
```

```
std::vector<int>().swap(v); // 清除v并最小化其容量 → size = 0, capacity = 0
```



其他值得注意的问题

(5) 在有对象继承情况下，建立指针的容器而不是对象的容器

- STL容器装入的对象是原始对象的一个拷贝

```
std::vector<Object> v;  
Object obj(...);  
v.push_back(obj); // push进去的不是原始obj, 二是obj的一个拷贝
```

- 如果对象很大，拷贝需要大量性能开销
- 由于继承的存在，拷贝会发生slicing，即：如果以基类对象建立一个容器而插入派生类对象，那么当对象通过基类的拷贝构造函数拷入容器的时候对象的派生部分会被切割

```
class SubObject : public Object { ... } // 继承Object的对象  
std::vector<Object> v;  
SubObject subObj;  
v.push_back(subObj); // slicing! subObj被当作基类拷贝进vector
```



其他值得注意的问题

(5) 在有对象继承情况下，建立指针的容器而不是对象的容器(续)

- 一个好的做法是建立指针的容器而不是对象的容器
 - 拷贝指针总是很快，开销小
 - 不会产生slicing问题

```
class SubObject : public Object { ... } // 继承Object的对象
std::vector<Object*> v;
Object* ptrSubObj = new SubObject();
v.push_back(ptrSubObj);
```



Thanks😊
Zhang wenjie
wjzhangb@163.com

