



Notes for CS106L

Lecture 4 Associative Contationers & Iterators

据说是最重要的一节课

回顾：

使用Vec[i]，访问一个越界的位置会发生什么？

- 1 会导致undefined behavior, 也就是造成了非常大的问题，并且这个问题是不可预测的。比如在Mac上很可能会继续运行，仿佛无事发生，然而在其他系统上可能会直接停止运行。

双端队列deque

- 1 基本上和vector功能一致，但是可以很快的在队头插入元素，不过访问中间元素更慢。
- 2 一般来说使用vector，但如果要在队头/尾进行很多操作，那么使用deque

Container Adaptors

<https://oi-wiki.org/lang/csl/container-adapter/>

eg: stack queue

stack和queue可以用sequence container实现吗？

- 1 C++ 的栈和队列在底层实际就是向量和双端队列，只不过是功能受限

既然stack和queue的本质是由vector和deque实现的，那么两者的效率相比是如何的？

- ▼
 - 1 stack和queue使用的是vector和deque中常数时间量级的函数，所以实际上它们比vector和deque更快，因为它们只使用了vector和deque中最快的函数

Associative Containers

<https://oi-wiki.org/lang/csl/associative-container/>

无法按照索引访问，数据按照键值对的形式存在。但实际上在底层set和map会按照key的某种特性排序(可以重载 `<` 来定义排序顺序)，而unordered_map/set不会。

eg: map/set/unordered_map/unordered_set

Iterators

<https://oi-wiki.org/lang/csl/iterator/>

Iterators允许在任何一个容器上迭代，无论它是有序的还是无序的

如何迭代一个map/set? (毕竟它们不能通过index进行迭代)

- ▼
 - 1 借助二叉树进行遍历
 - 2 (然而我们现在不需要关心这件事)

Iterator 的类型是?

类型就是 `Iterator` ,并且iterator取决于正在使用的集合的类型

Iterators 的四个常用功能:

Iterators provide a guaranteed interface!

Four essential iterator operations:

Create iterator

```
std::set<int>::iterator iter = mySet.begin();
```

Dereference iterator to read value currently pointed to

```
int val = *iter;
```

Advance iterator

```
iter++; or ++iter;
```

Compare against another iterator (especially .end() iterator

```
if (iter == mySet.end()) return;
```

Lecture 5 Advanced Containers

Multimap

和普通的map有什么区别?

- 1 普通的map中一个value不会对应多个key，而multimap允许相同的key指向不同的values

Map Iterators

它和普通的iterators略有不同，因为map包含key和value，所以map iterators 是一个pair，类似这种 `std::pair<string, int>`

可以通过 `(*i).first` 和 `(*i).second` 的方式分别访问所遍历map的key和value

Further Iterator Usages

<https://oi-wiki.org/lang/csl/algorithms/>

- `find` : 顺序查找。`find(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值。
- `count(x)` 返回 `set/map` 内键为 x 的元素数量。

* `find` 其实比 `count` 更快，因为 `count` 的底层是依照 `find` 实现的

- `sort`：排序。`sort(v.begin(), v.end(), cmp)` 或 `sort(a + begin, a + end, cmp)`，其中 `end` 是排序的数组最后一个元素的后一位，`cmp` 为自定义的比较函数。
- `lower_bound`：在一个有序序列中进行二分查找，返回指向第一个 大于等于 `x` 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。`lower_bound(v.begin(), v.end(), x)`。
- `upper_bound`：在一个有序序列中进行二分查找，返回指向第一个 大于 `x` 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。`upper_bound(v.begin(), v.end(), x)`。

Ranges

	[a,b]	[a,b)	(a,b]	(a,b)
begin	<code>lower_bound(a)</code>	<code>lower_bound(a)</code>	<code>upper_bound(a)</code>	<code>upper_bound(a)</code>
end	<code>upper_bound(b)</code>	<code>lower_bound(b)</code>	<code>upper_bound(b)</code>	<code>lower_bound(b)</code>

Iterator Type

所有iterator都可以完成：

- 创建
- 自增++
- 比较 == 或 !=

然而有一些iterator有着更加强大的功能：

input/output iterators

用于顺序单通道(sequential single-pass)数据结构，iterators只能递增一，并且input iterator是只读不写的；output iterator是只写的。

eg: `find`, `count`

forward iterators

可读可写，可以实现multiple passes

eg: `replace`

bidirectional iterators

可以实现递减 -- 操作

eg: reverse

random access iterators

功能最强大的iterator!

可读可写，可以任意递增或递减，比如 `+3/-2` 这样，而不用局限于 `++/--`

Lecture 6&7 Templates

Templates

或许运用template就如它的名字一样，也就是给一堆有共性的事物提取出来一个模板

比如，现在要比较两者之间的大小，返回时需要小的在前面，大的在后面。比较的对象的类型可以是int double string 等，然而对于这些类型来说，比较的方式是完全相同的，因此可以抽象成下面这样：

```
1 template <typename T>
2 pair<T, T> my_minmax(T a, T b){
3     if(a < b) return {a, b};
4     else return {b, a};
5 }
```

instantiation

最清晰的指定上文中的 `T` 的方式就是显式实例化(explicit instantiation)

eg: 调用 `my_minmax` 函数的时候指明 `T` 代表了什么(整个过程发生在编译时)

```
1 my_minmax<double>
2 my_minmax<string>
3 my_minmax<vector<int>>
4 ...
```

Generic Programming and Concept Lifting

泛型函数让我们可以编写出可以在多种不同上下文中使用的函数，它非常灵活。

概念提升意味着我们将从一个函数开始，然后审视我们对于函数的参数假设，质疑它们是否真的有必要？利用这些假设，我们将尝试给函数添加模版。

eg: 统计一个整数在向量中的出现次数

```
1 template<>
2 int countOccurrences(const vector<int>& vec, int val){
3     int count = 0;
4     for(size_t i = 0; i < vec.size(); ++i){
5         if (vec[i] == val) ++count;
6     }
7     return count;
8 }
```

我们可以发现，统计出现次数这件事未必需要发生在整数和向量之间，也可以是 [type][val] 在 vector of [type] 中出现的次数，修改后如下：

```
1 template <typename DataType>
2 int countOccurrences(const vector<DataType>& vec, DataType val){
3     int count = 0;
4     for(size_t i = 0; i < vec.size(); ++i){
5         if (vec[i] == val) ++count;
6     }
7     return count;
8 }
```

然而 vector 也不是必须的！我们可以统计 [type][val] 在 [collection] of [type] 中出现的次数，修改后如下：

```
1 template <typename Collection, typename DataType>
2 int countOccurrences(const Collection<DataType>& list, DataType val){
3     int count = 0;
4     for(size_t i = 0; i < list.size(); ++i){
5         if (list[i] == val) ++count;
6     }
7     return count;
8 }
```

*如果这里的 Collection 是 set/map 那么代码将不起作用，因为 set/map 无法索引

但可以用 iterator，它提供了一个可以遍历任何容器的统一接口。

```
1 template <typename Collection, typename DataType>
```

```
2 int countOccurrences(const Collection<DataType>& list, DataType val){  
3     int count = 0;  
4     for(auto iter = list.begin(); iter != list.end(); ++iter){  
5         if (*iter == val) ++count;  
6     }  
7     return count;  
8 }
```

最后，👉的函数假设我们需要遍历整个 `Collection`，我们也可以只遍历一个范围，只需要传入一个两个iterator，一个用来开始，另一个用来结束：

```
1 template <typename InputIterator, typename DataType>  
2 int countOccurrences(InputIterator begin, InputIterator end,  
3                      DataType val)  
4 {  
5     int count = 0;  
6     for(auto iter = begin; iter != end; ++iter)  
7     {  
8         if (*iter == val) ++count;  
9     }  
10    return count;  
11 }
```

*这里写的是 `InputIterator`，如果是 `random access iterators` 其实会限制可以使用函数的 `Collection`。这里的 `iterator` 只需要有读的功能就可以了，不需要其他的功能了。

Implicit Interfaces and Concepts

在做concept lifting的时候需要十分小心，因为这就是隐式接口(implicit interfaces)出现的地方。我们没有显示的说明一些要求是什么，但是代码本身会强制这些要求。

- 1 The compiler literally replaces each template parameter with whatever you instantiate it with.

当你声明模板时，你可以直接尝试将任何类型插入模版中，编译器会试图推断类型是什么，然后将其直接放入代码中，有时这样做是有效的，有时是无效的。

模板函数定义了每个模板参数必须满足的隐式接口，eg:

```
1 template <typename InputIterator, typename DataType>  
2 int countOccurrences(InputIterator begin, InputIterator end,
```

```

3             DataType val)
4 {
5     int count = 0;
6     for(auto iter = begin; iter != end; ++iter)
7     {
8         if (*iter == val) ++count;
9     }
10    return count;
11 }

```

- begin必须是可以复制的(copyable), eg: `stream` 是不可复制的
- iter必须可以和end比较
- iter必须是可以递增的

eg:

```

1 template <typename Collection, typename DataType>
2 int countOccurrences(const Collection& list, DataType val){
3     int count = 0;
4     for(size_t i = 0;i < list.size(); ++i){
5         if (list[i] == val) ++count;
6     }
7     return count;
8 }

```

- list必须有size()方法，并可以通过size()返回一个整数
- list必须可以索引，并且支持 `[]`
- list中的元素需要能和 `DataType` 所表示的类型可以比较

Functions and Lambdas

如果想要对之前统计某元素出现数量的函数做进一步的提升，可以采用统计有多少元素满足某一条件的方法，其中的某一条件可以是判断两元素是否相等，也可以是元素是否小于/大于某一个值等。

我们可以通过一个函数来实现判断元素是否满足条件这件事。我们将这个 `function` 称为 `predicate`

```

1 template <typename InputIterator, typename UnaryPredicate>
2 int countOccurrences(InputIterator begin, InputIterator end,
3                      UnaryPredicate predicate)
4 {

```

```

5   int count = 0;
6   for(auto iter = begin; iter != end; ++iter)
7   {
8     if (predicate(*iter)) ++count;
9   }
10  return count;
11 }

```

eg :

```

1 bool isLessThan5(int val){
2   return val < 5;
3 }
4 int main()
5 {
6   vector<int> vec{1, 3, 5, 7, 9};
7   countOccurrences(vec.begin(), vec.end(), isLessThan5);
8   //returns 2
9   return 0
10 }

```

* **predicates** 必须返回一个布尔值, **predicates** 用于调用某些东西并看它是否返回 **true/false**

我们称这种方法为 **function pointers** , 我们本质上在传递另一个函数。必须编写函数, 并且必须知道调用哪个函数。这种方法有两个主要问题:

- 我们必须为功能相似的任务编写不同的函数, 比如判断是否小于x这件事, 随着x值的不同, 要写非常多几乎相似的函数来实现对应的功能。
- 从上面的例子中可以发现, 当调用predicate的时候, 只传递了一个参数! 如果尝试声明另一个有多个参数的predicate, 那么就会报错, 代码将会是无法编译的。

解决方法:

- function objects (functors)
- *lambda functions*
 - 你可以创建一个非常轻量的函数, 这个函数是一个对象, 但它表现得像一个函数。

eg:

```

1 int main()
2 {
3   int limit = getInteger("Minimum for A?");
4   vector<int> grades = readStudentGrades();

```

```
5 auto func = [limit](auto val) {return val >= limit;};
6 countOccurrences(grades.begin(), grades.end(), func);
7 }
```

lambda函数的格式如下：

```
1 auto func = [capture-clause](parameters) -> return-value{
2 //body
3 };
4 //lambda函数的返回值一般是非常明显的，通常对于predicate它们返回boolean
5 //所以其实return-value那里可以不写
6 auto func = [capture-clause](parameters) {
7 //body
8 };
9 //capture-clause的部分用于捕捉这个作用域中的变量并使其在func中可以使用
10 //parameters的部分可以使用auto，body的部分和函数是一样的
```

`capture-clause` 的部分可以选择使用 `reference` 的方式来捕捉，也可以不用。

eg:

```
1 set<string> teas{"black", "green", "oolong"};
2 string banned = "boba";
3 auto likedByAvery = [&teas, banned](auto type){
4     return teas.count(type) && type != banned;
5 };
```

Algorithm

可参阅文档：

<https://www.apiref.com/cpp-zh/cpp/algorithm.html>

<https://oi-wiki.org/lang/csl/algorithm/>

- 上文中提到的 `countOccurrences` 其实在STL中有更好的实现方法，即 `count/count_if`。`count` 是没有谓词的，你只需要提供数据值，然后它会检查有多少个实例。`count_if` 需要谓词。
- `sort`：<https://www.apiref.com/cpp-zh/cpp/algorithm/sort.html>

排序。`sort(v.begin(), v.end(), cmp)` 或 `sort(a + begin, a + end, cmp)`，其中 `end` 是排序的数组最后一个元素的后一位，`cmp` 为自定义的比较函数。

- `stable_sort`：稳定排序，用法同 `sort()`。

- `nth_element` : 按指定范围进行分类, 即找出序列中第 n大的元素, 使其左边均为小于它的数, 右边均为大于它的数。 `nth_element(v.begin(), v.begin() + mid, v.end(), cmp)` 或 `nth_element(a + begin, a + begin + mid, a + end, cmp)`。
- `copy_if` : <https://www.apiref.com/cpp-zh/cpp/algorith/copy.html>

查看某一个范围, 并将所有满足predicate的元素复制到目标位置。

```
std::copy(iterator source_first, iterator source_end, iterator target_start);
```

其中

- `iterator source_first, iterator source_end` - 是源容器的迭代器位置。
- `iterator target_start` - 是目标容器的开始迭代器。

返回值: `iterator` - 它是指向已复制元素的目标范围末尾的迭代器。

eg: 这种情况下是无效的:

```
1 std::copy_if(courses.begin(), courses.end(), cscourses, isDep);
```

vector会有一个默认的大小, 但是这些algorithm并不是vector本身的成员函数, 所以它们其实不能改变那个vector的大小。所以它们其实是尝试向vector中写入, 然而会超出end iterator, 然后继续写入东西。

解决方法: 有一种特殊的迭代器, 可以改表container的大小, 即 `back_inserter`

```
1 std::copy(course.begin(), course.end(), back_inserter(cscourses), isDep);
```

- `remove_if` : <https://www.apiref.com/cpp-zh/cpp/algorith/remove.html>

删除。从向量中移除所有满足predicate返回true的项。但是注意, 它删除后不会改变 container的大小, 即使用remove后, container有效部分的后面会有一堆垃圾数据。

解决方法: `erase-remove`

```
1 v.erase(
2   std::remove_if(v.begin(), v.end(), pred),
3   v.end()
4 )
```

这样就删掉了尾部的垃圾数据。

Lecture 8 STL & Lecture 9 Classes and Const Correctness

Abstraction in the STL

在我们学习c++的过程中最早接触到的就是c++中的 **basic types**，然而在编程的过程中我们往往需要用到这些basic types的collection，这些collection就是STL中的 **container**；再进一步抽象，**iterator** 允许我们对container进行操作，不管这个container的类型是什么；最后 **algorithm** 允许我们对任何类型的容器使用任何类型的函数，它不仅允许我们运算，还允许我们传递predicate以找到满足条件的元素。

classes 的内容在CS106B中有更加详细的说明

Const

const的两种用法：

- 通过引用传递的常量参数
- 将const用作函数，const函数不能修改任何类变量或传递给它的任何变量

使用const对保护我们程序的安全性有很大的帮助。

const允许我们推断我们传递给某个函数的变量是否会被修改。当我们传递const参数时，const方法保证不会改变我们传递的值。

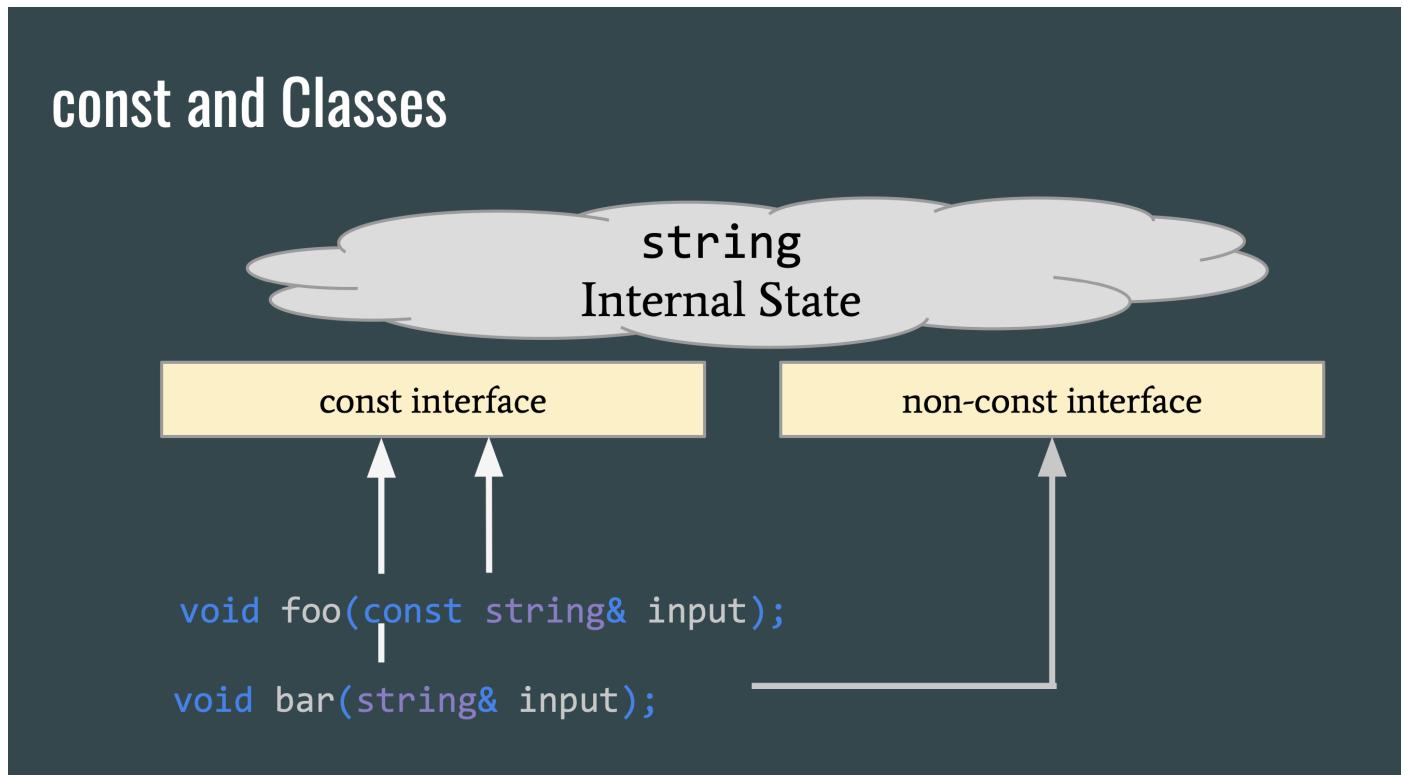
eg:

```
1 void f(int& x){  
2 // the value of x here  
3     aConstMethod(x);  
4     anotherConstMethod(x);  
5 // is the same value of x here  
6 }
```

const & classes

实际上，我们不仅可以使参数变为const，我们还可以使函数本身成为const，也就是 **const member functions**。现在我们意识到，实际上函数可以分为两类，一类是 **const member**

`functions` 另一类是 `non-const member functions`，前者能提供`const`接口，处理`const`和`non-const`变量，而后者仅可以处理`non-const`变量。



const pointer

`const` pointer所指向的内容是可以更改的(eg: `(*p)++`是可以的)，但是这个指针本身是不能更改的(eg:`p++`是不可以的)。

`const`可以修饰指针，也可以修饰常量，还可以既修饰指针又修饰常量！

- `const`修饰指针(常量指针) `const int* p`：指针的指向可以修改，但指针指向的值不能修改
- `const`修饰常量(指针常量) `int * const p`：指针的指向不可以改，但是指针指向的值可以改
- `const`既修饰指针又修饰常量 `const int* const p`：指针的指向和指针指向的值都不能变

A Const Pointer

- Using pointers with const is a little tricky
 - When in doubt, read right to left

```
//constant pointer to a non-constant int
int * const p;

//non-constant pointer to a constant int
const int* p;
int const* p;

//constant pointer to a constant int
const int* const p;
int const* const p;
```

const iterators

可以改变iter指向的内容，但不能改变它本身

```
1 vector v{1, 123}
2 const vector<int>::iterator itr = v.begin();
3 ++ iter; // doesn't compile
4 *iter = 15; // compiles
5
6
7 vector<int>::const_iterator itr = v.begin();
8 *iter = 5; // bad!!! can't change value of iter
9 ++ iter; // ok
10 int value = *iter; // ok
```

Final Notes

Final Notes

`const` on objects:

Guarantees that the object won't change by allowing you to call only `const` functions and treating all public members as if they were `const`. This helps the programmer write safe code, and also gives the compiler more information to use to optimize.

`const` on functions:

Guarantees that the function won't call anything but `const` functions, and won't modify any non-static, non-mutable members.

Lecture 10 Operators

operator overloading:

<https://en.cppreference.com/w/cpp/language/operators>

可重载的运算符：

There are 40 (+4) operators you can overload!

Arithmetic	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>		
	<code>+=</code>	<code>=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>		
Bitwise		<code>&</code>	<code> </code>	<code>~</code>	<code>!</code>		
Relational	<code>==</code>	<code>!=</code>	<code><</code>	<code>></code>	<code><=</code>	<code>>=</code>	<code><=></code>
Stream	<code><<</code>	<code>>></code>	<code><<=</code>	<code>>>=</code>			
Logical	<code>&&</code>	<code> </code>	<code>^</code>	<code>&=</code>	<code> =</code>	<code>^=</code>	
Increment	<code>++</code>	<code>--</code>					
Memory	<code>-></code>	<code>->*</code>	<code>new</code>	<code>new []</code>	<code>delete</code>	<code>delete []</code>	
Misc	<code>()</code>	<code>[]</code>	<code>,</code>	<code>=</code>			<code>co_await</code>

在重载运算符的时候需要注意，有些运算符必须用 **member function** 实现，有些必须用 **non-member function** 实现：

General rule of thumb: member vs. non-member

1. Some operators must be implemented as members (eg. [], (), ->, =) due to C++ semantics.
2. Some must be implemented as non-members (eg. << if you are writing class for rhs, not lhs).
3. If unary operator (eg. ++), implement as member.

General rule of thumb: member vs. non-member

4. If binary operator and treats both operands equally (eg. both unchanged) implement as non-member (maybe friend). Examples: +, <.
5. If binary operator and not both equally (changes lhs), implement as member (allows easy access to lhs private members). Examples: +=



要点：

Key Takeaways

1. Always think about const-ness of parameters. Here, we are modifying the stream, not the Fraction struct.
2. Return reference to support chaining << calls.
3. Here we are overloading << so our class works as the rhs...but we can't change the class of lhs (stream library).

31 October 2019



34

Lecture 11 Special Member Functions

special member functions之所以 **special** 是因为如果你不在类中声明这些函数，编译器会为你创建它们。但问题是，有时候编译器为你创建的函数可能不是你实际需要的。所以需要知道如何声明它们，以备不时之需。

copy operations

由编译器生成的special member functions通常意味着有很多例外，但通常是指以下四个：

- Default construction: 创建对象时没有参数
- Copy construction: 在使用现有对象创建新对象时调用，只是创建一个现有对象的副本。
- Copy assignment: 有两个对象的时候，一个正在赋值给另一个，赋值的那个对象必须首先清除那里的一切，并用另一个vector的内容替换它。
- Destruction: 越界时将对象destory

The copy operations must perform the following tasks.

Copy Constructor



- Use initializer list to copy members where assignment does the correct thing.
 - int, other objects, etc.
- Deep copy all members where assignment does not work.
 - pointers to heap memory

Copy Assignment

- Clean up any resources in the existing object about to be overwritten.
- Copy members using initializer list when assignment works.
- Deep copy members where assignment does not work.

5 November 2019

41

eg:

```
1 StringVector function(StringVector vec0){ //vec0:Copy constructor
2     StringVector vec1; //Default constructor
3     StringVector vec2{"ding", "han", "fei"}; //Normal constructor
4     StringVector vec3(); //declares a function
5     StringVector vec4(vec2); //Copy constructor
6     StringVector vec5{}; //Default constructor
7     StringVector vec6{vec3 + vec4}; //Copy constructor
8     StringVector vec7 = vec4; //Copy constructor
9     vec7 = vec2; //Copy assignment
10    return vec7; // Copy constructor creates a copy of local variable
11 }
```

在函数return的时候会发生 **copy construction**，创建一个局部变量的拷贝，以便在其他地方也能访问这个被拷贝的变量。

rule of three

When do you need to write your own special member functions?

When the default one generated by the compiler does not work.

Most common reason: ownership issues
A member is a handle on a resource outside of the class.
(eg: pointers, mutexes, filestreams.)

Rule of Three



If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three.

The fact that you defined one of these means one of your members has ownership issues that need to be resolved.

rule of zero

Rule of Zero



If the default operations work, then don't define your own custom ones.

emplace_back

https://www.apiref.com/cpp-zh/cpp/container/vector/emplace_back.html

emplace_back和push_back有什么区别?

- 1 当需要添加一个新元素的时候, push_back会先创建一个临时对象, 然后再移动/拷贝到vector中。然而与其创建这个临时的对象, 不如直接在vector尾部创建这个元素, 这样就省去了移动/拷贝的过程。

Lecture 12 Move Semantics

lvalues and rvalues

this is a simplification of a complicated topic!

l-value : 有 `name(identity)`

- 可以用 `&` 来获取它的地址
- 可以出现在 `=` 的左右两边

r-value : 没有 `name(identity)`

- 临时的值
- 不能用 `&` 来获取它的地址
- 只能出现在 `=` 的右边

eg: `=` 的左都是 **l-value**

```
1 int val = 2; //2是r-value, val是l-value
2 int* ptr = 0x1235532; //ptr是l-value, 0x1235532是l-value
3 vector<int> v1{1, 2, 3};
4
5 auto v4 = v1 + v2; //v1, v2是l-value, 但v1+v2是r-value,
6 v1 += v4; // v4: l-value
7 size_t size = v.size(); //size: l-value, v.size(): r-value
8 val = static_cast<int>(size); //static_cast<int>(size): r-value
9 v1[1] = 4 * i; //4 * i: r-value, v1[1]: l-value, 因为v[]这种形式是通过引用
  (reference)返回的
10 ptr = &val; //&val:r-value
```

```
11 v1[2] = *ptr; /*ptr: l-value
```

lvalues reference and rvalues reference

lvalues reference就是对lvalue的reference

```
1 int val = 2;
2 int* ptr = 0x1235532;
3 vector<int> v1{1, 2, 3};
4
5 auto& ptr2 = ptr; //相当于创建一个对原值的引用，对ptr2的任何更改都在改变ptr
6 auto&& v4 = v1 + v2; //v4是一个rvalue的reference，相当于延长了临时值的生命周期
7 //任何对v4进行的更改都会改变其对应的临时值
8 //表示这是一个r-value的reference
9 auto& ptr3 = &val; //error!不能将lvalue reference绑定到rvalue
10
11 auto&& val2 = val; //error!
12 //不能将rvalue reference绑定到lvalue
13
14 const auto& ptr3 = ptr + 5; //可以将一个常量左值引用绑定到右值
```

Examples: what the value categories of each expression?

```
int val = 2;
int* ptr = 0x02248837;
vector<int> v1{1, 2, 3};

auto& ptr2 = ptr;           // ptr2 is an l-value reference
auto&& v4 = v1 + v2;       // v4 is an r-value reference
auto& ptr3 = &val;         [ ] // ERROR: can't bind l-val ref
                           //      to r-value
auto&& val2 = val;         // ERROR: can't bind r-val ref
                           //      to l-value
const auto& ptr3 = ptr + 5; // OKAY: CAN bind const l-val ref
                           //      to r-value (WHY?)
```

Fun C++ errors: "Invalid non-const ref of type X& from r-value of type X"

```
void nocos_Lref(vector<int>& v);
void const_Lref(const vector<int>& v);
void nocos_Rref(vector<int>&& v);
// BTW: no one uses const_Rref

vector<int> v1 = v2 + v3;           // v1 is l-value
nocos_Lref(v1);                  // OKAY: l-val ref binds to l-v
nocos_Rref(v1);                   // ERROR: r-val ref NO bind to r-v
nocos_Lref(v2 + v3);             // ERROR: l-val ref NO bind to l-v
const_Rref(v2 + v3);             // OKAY: const l-ref binds to r-v
nocos_Rref(v2 + v3);             // OKAY: r-val ref binds to r-v
```

6 November 2019

20

感觉第三四行注释最后面的r-v和l-v好像写反了?

move operations

r-value 是临时的值，它很快就会消失，因此可以窃取(**steal**)它的资源。

- **l-value** 是不可丢弃(disposable)的
- **r-value** 是可丢弃的

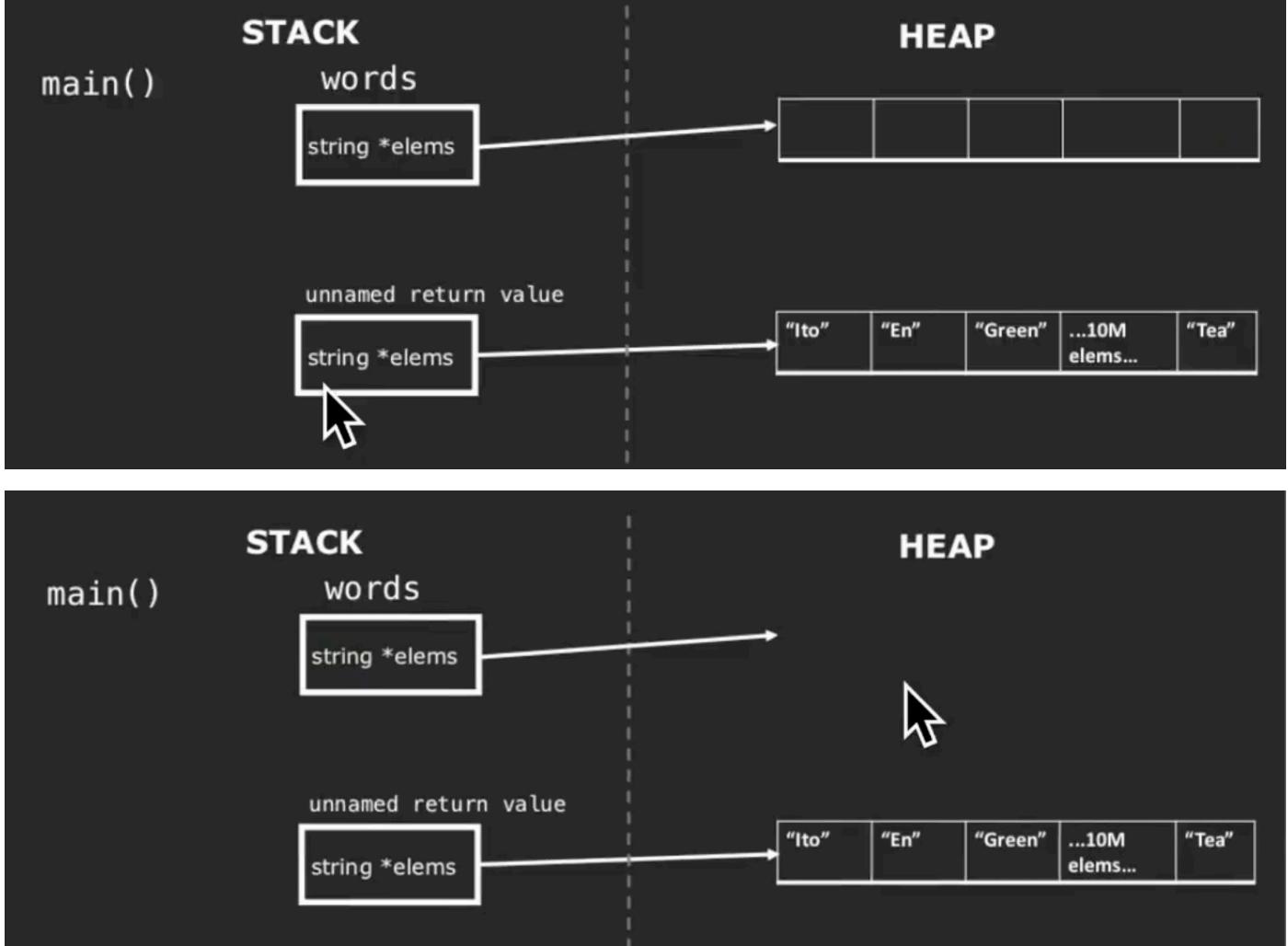
2 new special member functions!

Move constructor：从现有的r-value创建一个新对象。直接将内容移动到新的对象那里。

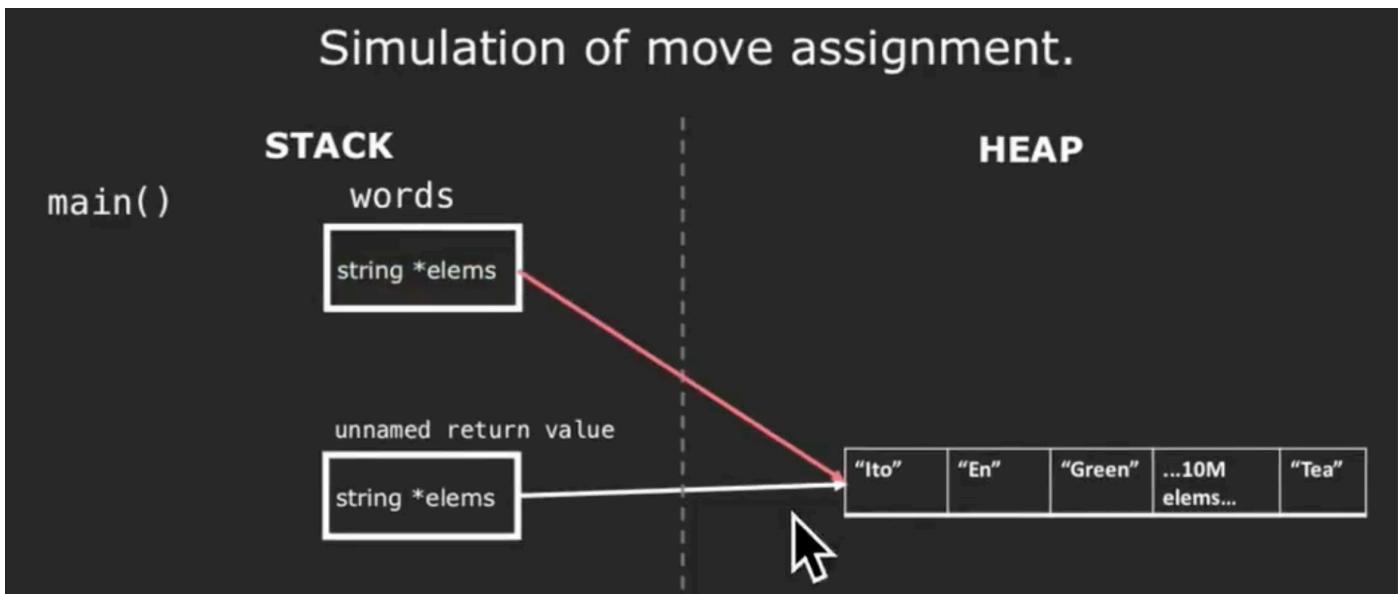
Move assignment：使用一个现有的r-value覆盖另一个已经存在的对象。因为r-value是临时的，可以被steal，所以在清除对象的原有内容之后直接将内容移动过去就可以了。

step1:删除

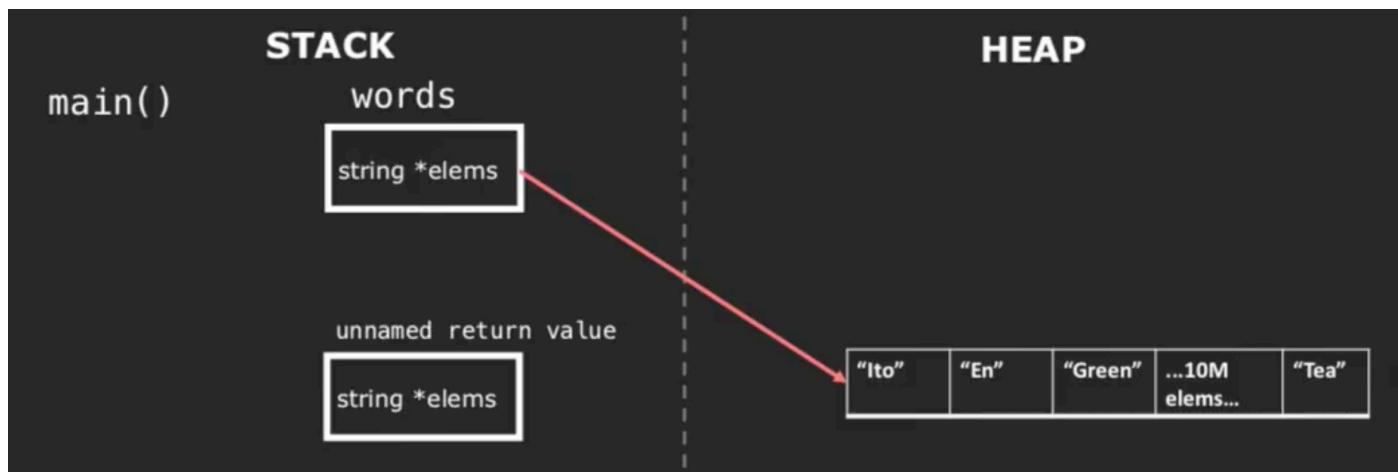
Simulation of move assignment.



step2: 移动



step3: 窃取



在实际情况中，如果 `=` 左边是一个l-value，那么我们所做的操作是copy而不是move，eg:

```

1 class axess{
2     public:
3     axess()
4         //other special member functions
5     axess(axess&& other) : student(other.students){}
6     axess& operator=(axess&& rhs){
7         students = rhs.students;
8     }
9     private:
10    vector<student> students;
11 }
```

`students = rhs.students` 这里的 `rhs.students` 实际上是一个l-value，因此我们在操作是 copy，如果这是一个右值，那么才是move操作。

使用 `std::move` 可以将它变成move操作: `students = std::move(rhs.students)`

`std::move` 无条件将变量强制转换为r-value。这也意味着你对某个东西调用了 `move` 之后，不应该再使用它。

eg: 使用move的方法写swap()函数

```

1 template <typename T>
2 void swap(T& a,T& b)
3 {
4     T temp = std::move(a);
5     a = std::move(b);
6     b = std::move(temp);
```

rule of 5

因为多了两个move，所以现在是rule of 5:

Rule of Five

If you explicitly define (or delete) a copy constructor, copy assignment, move constructor, move assignment, or destructor, you should define (or delete) all five.

The fact that you defined one of these means one of your members has ownership issues that  need to be resolved.

Lecture 13 Namespaces&Inheritance

Namespaces

它是一种方法，这种方法可以让你将代码行或你声明的类分组到一个单一的 `namesapce` 中，以便以后引用。

标准库中有非常多的函数，在我们编写程序的过程中很可能会出现和标准库中函数重名的情况。
`namesapce` 将全局作用域划分成不同的部分，从而不同 `namesapce` 中的标识符可以重名而不发生冲突。

假设在编写程序的过程中定义了一个 `int` 型的变量 `count`，而此时我们又想使用 `algorithm` 中的 `count` 函数，因此就需要有一种方式来告诉编译器我们使用的是哪个 `count`。引入 `namesapce` 可以更好的控制标识符的作用域。本质上，命名空间就是定义了一个范围。

Scope Resolution

作用域解析的运算符是两个冒号写在一起 `::`。用于访问命名空间中的成员，指明要使用哪个命名空间下的标识符。

总结：

- **命名空间**: 用于组织代码，避免名称冲突。
- **作用域解析运算符 `::`**: 用于访问命名空间中的成员。

eg:

```
1 #include <iostream>
2 .
3 // 定义命名空间 MyMath
4 namespace MyMath {
5     int add(int a, int b) {
6         return a + b;
7     }
8
9     int subtract(int a, int b) {
10        return a - b;
11    }
12 }
13
14 // 定义另一个命名空间 MyString
15 namespace MyString {
16     void printHello() {
17         std::cout << "Hello, World!" << std::endl;
18     }
19 }
20
21 int main() {
22     // 使用作用域解析运算符访问 MyMath 命名空间中的函数
23     int sum = MyMath::add(5, 3);
24     int difference = MyMath::subtract(5, 3);
25
26     std::cout << "Sum: " << sum << std::endl;           // 输出: Sum: 8
27     std::cout << "Difference: " << difference << std::endl; // 输出:
28     Difference: 2
29
30     // 使用作用域解析运算符访问 MyString 命名空间中的函数
31     MyString::printHello(); // 输出: Hello, World!
32
33     return 0;
34 }
```

Inheritance

interfaces

C++中没有interface(接口)的关键字

- 如果想成为一个 `interface`，需要确保 `class` 中只包含 `pure virtual functions`
- 如果想要实现一个 `interface`，那么那个 `class` 必须定义所有的 `pure virtual functions`，不然它将无法编译。

如果我们想要在 `class` 中定义一些函数呢？

- 1 如果一个 `class` 中包含至少一个``pure virtual functions``，那么这个 `class` 被称作是一个 `abstract class`(抽象类)。
- 2
- 3 `abstract class` 不能被实例化，只有当一个 `class` 中不含``pure virtual functions`` 的时候才能被实例化。

public & protected & private

`public`: 公有属性，凡是在它下面声明的变量和函数，都可以在类的内部和外部访问。

`protected`: 保护属性，凡是在它下面声明的变量和函数，只能在类的内部以及派生类（子类）中访问。

`private`: 私有属性，凡是在它下面声明的变量和函数，只能在类的内部访问。可以使用公有成员函数，用于获取私有变量的值。

有三种继承方式：

- **public继承**: 基类 `public` 成员，`protected` 成员，`private` 成员的访问属性在派生类中分别变成：`public`, `protected`, `private`
- **protected继承**: 基类 `public` 成员，`protected` 成员，`private` 成员的访问属性在派生类中分别变成：`protected`, `protected`, `private`
- **private继承**: 基类 `public` 成员，`protected` 成员，`private` 成员的访问属性在派生类中分别变成：`private`, `private`, `private`

结构体（`struct`）的继承在 C++ 中与类（`class`）的继承基本相同。不同之处在于默认的访问控制修饰符：

- 在 `struct` 中，默认的继承方式是 `public`。

- 在 `class` 中，默认的继承方式是 `private`。

Inherited Member

Inherited Member 指的是基类中定义的属性（成员变量）和方法（成员函数），这些成员在派生类中可以直接访问和使用。通过继承，派生类能够获得基类的功能，进而实现代码重用和扩展。

当派生类中定义了一个与基类同名的成员（无论是成员变量还是成员函数），派生类中的成员会覆盖基类中的同名成员，使得基类的同名成员无法通过派生类的对象直接访问。

eg:

名称隐藏：派生类中的 `display(int)` 方法隐藏了基类中的 `display()` 方法，因此在派生类对象中无法直接访问基类的 `display()` 方法。

访问隐藏的方法：可以通过作用域解析运算符（`Base::`）访问基类的同名成员。

```
1 #include <iostream>
2 using namespace std;
3
4 class Base {
5 public:
6     void display() {
7         cout << "Display from Base" << endl;
8     }
9 };
10
11 class Derived : public Base {
12 public:
13     void display(int) { // 重新定义了 display 方法
14         cout << "Display from Derived" << endl;
15     }
16 };
17
18 int main() {
19     Derived obj;
20     obj.display(10); // 调用 Derived 的 display 方法
21
22     // obj.display(); // 错误：无法调用 Base 的 display 方法，因为它被隐藏了
23     obj.Base::display(); // 可以通过作用域解析运算符调用基类的 display 方法
24
25     return 0;
26 }
```

Abstract Classes

如果一个 `class` 中包含至少一个 `pure virtual functions`，那么这个 `class` 被称作是一个 `abstract class` (抽象类)。

`Interfaces` 是 `abstract class` 的子集。

eg:

```
1 class Base{
2     public:
3         virtual void foo() = 0; // pure virtual function
4         virtual void foo2(); //non-pure virtual function
5         void bar() = {return 42;}; // regular function
6 }
```

概括来说，如果你想要你的函数可以被 `overwrite`，你需要声明它为 `virtual` (虚函数)，如果你想强制它被 `overwrite`，可以通过第三行代码那样添加0来声明它是 `pure virtual functions`。

Base class & Derived class

基类是一个被其他类继承的类。它通常包含一些通用的属性和方法，这些属性和方法可以被派生类继承和重用。

- 特点

- 可以包含数据成员和成员函数。
- 可以是抽象类 (包含纯虚函数) 或具体类。

派生类是从基类继承而来的类。它可以扩展基类的功能，添加新的属性和方法，或重写基类的方法。

- 特点

- 继承基类的成员。
- 可以重写基类中的虚函数，以提供特定实现。
- 可以添加新的成员变量和方法。

Destructors

如果你想使你的类是可继承的，需要将析构函数设为虚拟的，否则可能会出现内存泄露的情况。

Lecture 14 Template Classes and Concepts

Templates vs. Derived Classes

模板是隐式接口，派生类是显式接口。

- 模板的多态性是通过类型参数实现的，也称为静态多态性。编译器在编译时根据传入的类型生成相应的代码。
- 派生类的多态性是通过虚函数实现的，称为动态多态性。运行时根据对象的实际类型调用相应的函数。

区别总结：

多态的实现方式：

- **模板**：静态多态性，编译时决定类型。
- **派生类**：动态多态性，运行时决定调用哪个函数。

应用场景：

- **模板**：
 - 用于编写通用代码，如数据结构（如 `std::vector`）。
 - 运行时速度/效率很重要
 - 没有共同的基类
- **派生类**：
 - 用于需要共享接口和实现的场景，如基于类层次结构的设计。
 - 编译时速度/效率很重要
 - 不想出现 `code bloat`，含有template的代码在编译的时候可能会有多个模板版本，int版，double版...

类型安全性：

- 模板在编译时检查类型，确保类型安全。
- 派生类使用虚函数时，运行时根据对象的实际类型进行调用。

Casting

补充知识点

```
1 int a = (int)b;
2 int a = int(b);
3 int a = static_cast<int>(b); //best practice
```

Template Classes

一个 `class template` 描述了如何构建一堆看上去相似的类。通过使用模板，可以实现代码的重用和类型独立性。

类模板的定义使用关键字 `template`，后面跟着模板参数。在类模板中，可以使用这些参数作为数据成员的类型或成员函数的参数类型。

eg:

```
1 // 定义类模板
2 template <typename T>
3 class Box {
4 private:
5     T value; // 数据成员，类型为 T
6 public:
7     Box(T val) : value(val) {} // 构造函数
8     // 成员函数，显示值
9     void display() {
10         cout << "Value: " << value << endl;
11     }
12     // 成员函数，返回值
13     T getValue() {
14         return value;
15     }
16 };
17
18 int main() {
19     // 创建 Box<int> 对象
20     Box<int> intBox(123);
21     intBox.display(); // 输出: Value: 123
22
23     // 创建 Box<double> 对象
24     Box<double> doubleBox(45.67);
25     doubleBox.display(); // 输出: Value: 45.67
26
27     // 使用 getValue
28     cout << "Integer Box Value: " << intBox.getValue() << endl; // 输出: 123
29     cout << "Double Box Value: " << doubleBox.getValue() << endl; // 输出:
45.67
```

```
30
31     return 0;
32 }
```

Concept & Constraints

concept: named set of constraints.

概念（Concepts）是用于指定模板参数要求的一种机制。允许我们将template中那些隐式接口转换为显式要求。concepts可以帮助我们定义一个模板所需的特性，从而提高代码的可读性和可维护性。

约束（Constraints）是应用概念的一种方式，限制模板参数的类型。通过约束，编译器可以确保只有满足特定条件的类型才能用于模板实例化。

eg:

```
1 // 定义一个概念，要求类型 T 必须是可迭代的
2 template <typename T>
3 concept Iterable = requires(T a) {
4     { begin(a) } -> convertible_to<typename T::iterator>;
5     { end(a) } -> convertible_to<typename T::iterator>;
6 };
7
8 // 定义一个函数，计算可迭代容器的总和
9 template <Iterable T>
10 auto sum(const T& container) {
11     typename T::value_type total = 0; // 使用容器的值类型
12     for (const auto& item : container) {
13         total += item;
14     }
15     return total;
16 }
17
18 int main() {
19     vector<int> intVec = {1, 2, 3, 4, 5};
20     list<double> doubleList = {1.1, 2.2, 3.3};
21
22     cout << "Sum of intVec: " << sum(intVec) << endl;
23     // 输出: Sum of intVec: 15
24     cout << "Sum of doubleList: " << sum(doubleList) << endl;
25     // 输出: Sum of doubleList: 6.6
26
27     return 0;
```

特点：

- **提高可读性**: 使用概念可以使模板代码更易于理解。
- **早期错误检测**: 编译器可以在模板实例化时检查约束，提供更早的错误反馈。
 - eg: 第20行

```

1 // 定义一个概念，要求类型 T 必须是可比较的
2 template <typename T>
3 concept Comparable = requires(T a, T b) {
4     { a < b } -> convertible_to<bool>;
5 };
6
7 // 使用概念约束模板参数
8 template <Comparable T>
9 void compare(T a, T b) {
10     if (a < b) {
11         cout << a << " is less than " << b << endl;
12     } else {
13         cout << a << " is not less than " << b << endl;
14     }
15 }
16
17 int main() {
18     compare(5, 10);          // 整数可以比较
19     compare(3.14, 2.71);    // 浮点数可以比较
20     // compare("hello", "world"); // 编译错误，因为字符串不可比较
21     return 0;
22 }
23

```

- **灵活性**: 概念允许组合多个类型要求，增加了模板编程的灵活性。

Lecture15 RALL & Smart Pointers

RALL

Resource Acquisition Is Initialization

- 1 "The best example of why I shouldn't be in marketing"
- 2 "I didn't have a good day when I named that"

我们如何保证一个类释放资源？不管是否有异常发生。

Aside: Enforcing exception safety

Functions can have four levels of exception safety:

- **Nothrow exception guarantee**
 - absolutely does not throw exceptions: destructors, swaps, move constructors, etc.
- **Strong exception guarantee**
 - rolled back to the state before function call
- **Basic exception guarantee**
 - program is in valid state after exception
- **No exception guarantee**
 - resource leaks, memory corruption, bad...

在获取后需要被释放的资源：

Resources	Acquire	Release
Heap memory	new	delete
Files	open	close
Locks	try_lock	unlock
Sockets	socket	close

RALL有一个更好的名字是SBRM(Scope Based Memory Management) 作用域基础内存管理，使用作用域的思想来自动释放内存。

也可以叫做CADRE(Constructor Acquires, Destructor Releases) 构造函数获取，析构函数释放。可以把释放资源的代码放在析构函数中，这样无论发生什么，只要你退出了函数，资源就会被释放。此外，如果你获得了一个资源，你应该总是在构造函数中获取！

Smart Pointer

eg:

```
1 std::unique_ptr  
2 std::shared_ptr  
3 std::weak_ptr
```

std::unique_ptr

单一的拥有某种资源，并在对象被销毁的时候，它将在析构函数中删除该资源。它是不可被复制的！如果我们试图去复制它，那么它就无法单一的拥有某种资源。

我们如何说明一个class不允许复制？

```
1 delete copy constructor and copy assignment
```

std::shared_ptr

资源可以被任意数量的指针指向，当不再有指针指向这个资源的时候，这个资源将被释放。

声明共享指针的方式：

```
1 在创建第一个指针之后，使用copy constructor声明所有后续的指针。
```

std::weak_ptr

作用类似于共享指针，但不会增加引用计数。

Lecture 16 Mutithreading

<https://www.runoob.com/cplusplus/cpp-multithreading.html>

线程是程序中的轻量级执行单元，允许程序同时执行多个任务。但实际上同一时刻只有一个在执行，但从一段时间上来看，确实有多个任务被执行了。（上课时举的拿粉笔写数字的例子）

Things to Take Away:

- Use atomic types if doing multithreading!
- `std::lock_guard` vs. `std::unique_lock`
- 3 types of “locks”/mutexes: normal, timed, recursive
- Condition variables allow cross-thread communication
 - see CS 110
- `std::async` is one way to use multithreading

C++11 及以后的标准提供了多线程支持，核心组件包括：

- `std::thread`：用于创建和管理线程。
- `std::mutex`：用于线程之间的互斥，防止多个线程同时访问共享资源。
- `std::lock_guard` 和 `std::unique_lock`：用于管理锁的获取和释放。
- `std::condition_variable`：用于线程间的条件变量，协调线程间的等待和通知。
- `std::future` 和 `std::promise`：用于实现线程间的值传递和任务同步。

eg:

```
1 #include <iostream>
2 #include <thread>
3 void printMessage(int count) {
4     for (int i = 0; i < count; ++i) {
5         std::cout << "Hello from thread (function pointer)!\n";
6     }
7 }
8 int main() {
9     std::thread t1(printMessage, 5); // 创建线程，传递函数指针和参数
10    t1.join(); // 等待线程完成
11    return 0;
12 }
```

`join()` 用于等待线程完成执行。如果不调用 `join()` 或 `detach()` 而直接销毁线程对象，会导致程序崩溃。上述例子中，只有 `t1` 线程 rejoin 主线程之后，主线程才会继续执行。

where to go?

The screenshot shows a presentation slide with the following content:

Where to go?

Use C++!

Further C++ reading:

Accelerated C++	<i>Andrew Koenig</i>
Effective C++	<i>Scott Meyers</i>
Effective Modern C++	<i>Scott Meyers</i>
Exceptional C++	<i>Herb Sutter</i>
Modern C++ Design	<i>Andrei Alexandrescu</i>
C++ Template Metaprogramming	<i>Abrahams and Gurtovoy</i>
C++ Concurrency in Action	<i>Anthony Williams</i>

At the bottom, there is a navigation bar with icons for back, forward, search, and other presentation controls, along with a video camera icon and the number '6'.