模板与STL初步

(OOP)

刘知远

liuzy@tsinghua.edu.cn

http://nlp.csai.tsinghua.edu.cn/~lzy/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 纯虚函数与抽象类
- 向下类型转换
- 多重继承的虚函数表，多重继承的利弊
- 多态
- 函数模板和类模板

本讲内容提要

- 类模板与函数模板特化
- 命名空间
- STL初步——容器与迭代器

回顾：类模板

- 在定义类时也可以将一些类型信息抽取出来，用模板参数来替换，从而使类更具通用性。这种类被称为“类模板”。例如：

```
#include <iostream>
using namespace std;

template <typename T> class A {
    T data;
public:
    void print() { cout << data << endl; }
};
int main() {
    A<int> a;
    a.print();
    return 0;
}
```

回顾：类模板

- 类模板除了可以用于指定成员变量的类型，还可以约束成员函数的返回值类型和参数类型。例如：

```
#include <iostream>
using namespace std;

template <typename T> class A {
    int data;
public:
    T sum(T a, T b) { return a + b; }
};
int main() {
    A<int> a;
    cout<<a.sum(1, 2)<<endl;
    return 0;
}
```

回顾：函数模板

- 有些算法实现与类型无关，所以可以将函数的参数类型也定义为一种特殊的“参数”，这样就得到了“函数模板”。
- 定义函数模板的方法

```
template <typename T> ReturnType Func(Args);
```

- 如：任意类型两个变量相加的“函数模板”

```
template <typename T>
T sum(T a, T b) { return a + b; }
```

函数模板特化

- 有时，有些类型并不合适，则需要对模板在某种情况下的具体类型进行特殊处理，这称为“模板特化”。
- 对于如下模板进行特化的两种方法：

```
template <typename T> T sum(T a, T b)
```

  - 在函数名后用<>括号括起具体类型

```
template<> char* sum<char*>(char* a, char* b)
```

  - 由编译器推导出具体类型，函数名为普通形式

```
template<> char* sum(char* a, char* b)
```

函数模板特化

```
#include <iostream>
using namespace std;

template<class T>
T div2(const T& val)
{
    cout << "using template" << endl;
    return val / 2;
}

template<>
int div2(const int& val) //函数模板特化
{
    cout << "better solution!" << endl;
    return val >> 1; //右移取代除以2
}
```

```
int main() {
    cout<<div2(1.5) << endl;
    cout<<div2(2) << endl;
    return 0;
}
```

运行结果

```
using template
0.75
better solution!
1
```

函数模板特化

- 注意：对于函数模板，如果有多个模板参数，则特化时必须提供所有参数的特例类型，不能部分特化。
- 但可以用重载来替代部分特化。

函数模板特化

```
#include <iostream>
using namespace std;

template<class T, class A>
T sum(const A& val1, const A& val2)
{
    cout << "using template" << endl;
    return T(val1 + val2);
}

template<class A>
int sum(const A& val1, const A& val2)
{   //不是部分特化，而是重载函数
    cout << "overload" << endl;
    return int(val1 + val2);
}
```

```
int main()
{
    float y = sum<float, float>
                (1.4, 2.4);
    cout << y << endl;
    int x = sum(1, 2);
    cout << x << endl;
    return 0;
}
```

运行结果

```
using template
3.8
overload
3
```

函数模板特化

- 函数模板重载解析顺序：

类型匹配的普通函数→基础函数模板→全特化函数模板

  - 如果有普通函数且类型匹配，则直接选中，重载解析结束
  - 如果没有类型匹配的普通函数，则选择最合适的基础模板
  - 如果选中的基础模板有全特化版本且类型匹配，则选择全特化版本，否则使用基础模板

函数模板特化

```
#include <iostream>
using namespace std;
template<class T> void f(T) {
  //func1为基础模板
   cout<< “full template” <<endl;};
template<class T> void f(T*) {
  //func2为func1的重载，仍是基础模板
   cout<< “full template -> overload template” <<endl;};
template<> void f(char*) {
  //func3为func2的特化版本(T特化为char)
   cout<< “overload template -> specialized” <<endl;};
```

```
int main() {
    char *p;
    f(p);
    return 0;
}
```

- 主函数调用的是哪一个版本？func3
- 优先匹配特化版本，前提是被特化的对应基础函数模板被匹配到。

函数模板特化

```
#include <iostream>
using namespace std;
template<class T> void f(T) {
  //func1为基础模板
   cout<< “full template” <<endl;};
template<> void f(char*) {
  //func3为func1的特化版本(T特化为char*)
   cout<< “full template -> specialized” <<endl;};
template<class T> void f(T*) {
  //func2为func1的重载,仍是基础模板
   cout<< “full template -> overload template” <<endl;};
```

```
int main() {
    char *p;
    f(p);
    return 0;
}
```

- 主函数调用的是哪一个版本？func2
- 先从基础模板func1和func2中选择更匹配的模板实例，func2参数类型更匹配，因此优先选中。
- 函数模板func2无特化版本，因此直接调用模板func2。

类模板特化

- 对于类模板，也可以进行特化
- 对于以下模板

```
template<typename T1, typename T2> class A
{ ... };
```

- 与函数模板类似，可以进行全部特化：

```
template<> class A<int, int> { ... };
```

类模板特化：全部特化

- 示例：计算a+b的类模板

```
#include <iostream>
using namespace std;
template<typename T1, typename T2> class Sum { //类模板
public:
Sum(T1 a, T2 b) {cout << "Sum general: " << a+b << endl;}
};
template<> class Sum<int, int> { //类模板全部特化
public:
Sum(int a, int b) {cout << "Sum specific: " << a+b << endl;}
};
int main(){
        Sum<int, int> s1(1, 2);
        Sum<int, double> s2(1, 2.5);
        return 0;
}
```

运行结果

```
Sum specific: 3
Sum general: 3.5
```

类模板特化

- 对于类模板，还允许部分特化，即只部分限制模板的通用性，如通用模板为：

```
template<typename T1, typename T2> class A
{ ... };
```

- 部分特化：第二个类型指定为int

```
template<typename T1> class A<T1, int> {...};
```

- 对比全部特化：指定所有类型

```
template<> class A<int, int> { ... };
```

类模板特化：部分特化

- 示例：计算a+b的类模板

```
#include <iostream>
using namespace std;
template<typename T1, typename T2> class Sum { //类模板
public:
Sum(T1 a, T2 b) {cout << "Sum general: " << a+b << endl;}
};
template< typename T1 > class Sum<T1, int> { //类模板部分特化
public:
Sum(T1 a, int b) {cout << "Sum specific: " << a+b << endl;}
};
int main(){
        Sum<double, int> s1(1.5, 2);
        Sum<double, double> s2(1.5, 2.5);
        return 0;
}
```

运行结果

```
Sum specific: 3.5
Sum general: 4
```

模板特化总结

- 类模板可以部分特化或者全部特化，编译器会根据调用时的类型参数自动选择合适的模板类。
- 函数模板只能全部特化，但可以通过重载代替部分特化的实现。编译器在编译阶段决定使用特化函数或者标准模板函数。
- 函数模板的全特化版本的匹配优先级可能低于重载的非特化基础函数模板，因此最好不要使用全特化函数模板而直接使用重载函数。

命名空间

命名空间（1）

- 为了避免在大规模程序的设计中，以及在程序员使用各种各样的C++库时，标识符的命名发生冲突，标准C++引入了关键字**namespace**（命名空间），可以更好地控制标识符的作用域。
- 标准C++库（不包括标准C库）中所包含的所有内容（包括常量、变量、结构、类和函数等）都被定义在命名空间**std**（**standard**标准）中。
  - cout、cin
  - vector、set、map

命名空间（2）

- 定义命名空间：

```
namespace A {
    int x, y;
}
```

- 使用命名空间：

```
A::x = 3;
A::y = 6;
```

命名空间（3）

- 使用using声明简化命名空间使用
- 使用整个命名空间：所有成员都直接可用

```
using namespace A;
x = 3; y = 6;
```

- 使用部分成员：所选成员可直接使用

```
using A::x;
x = 3; A::y = 6;
```

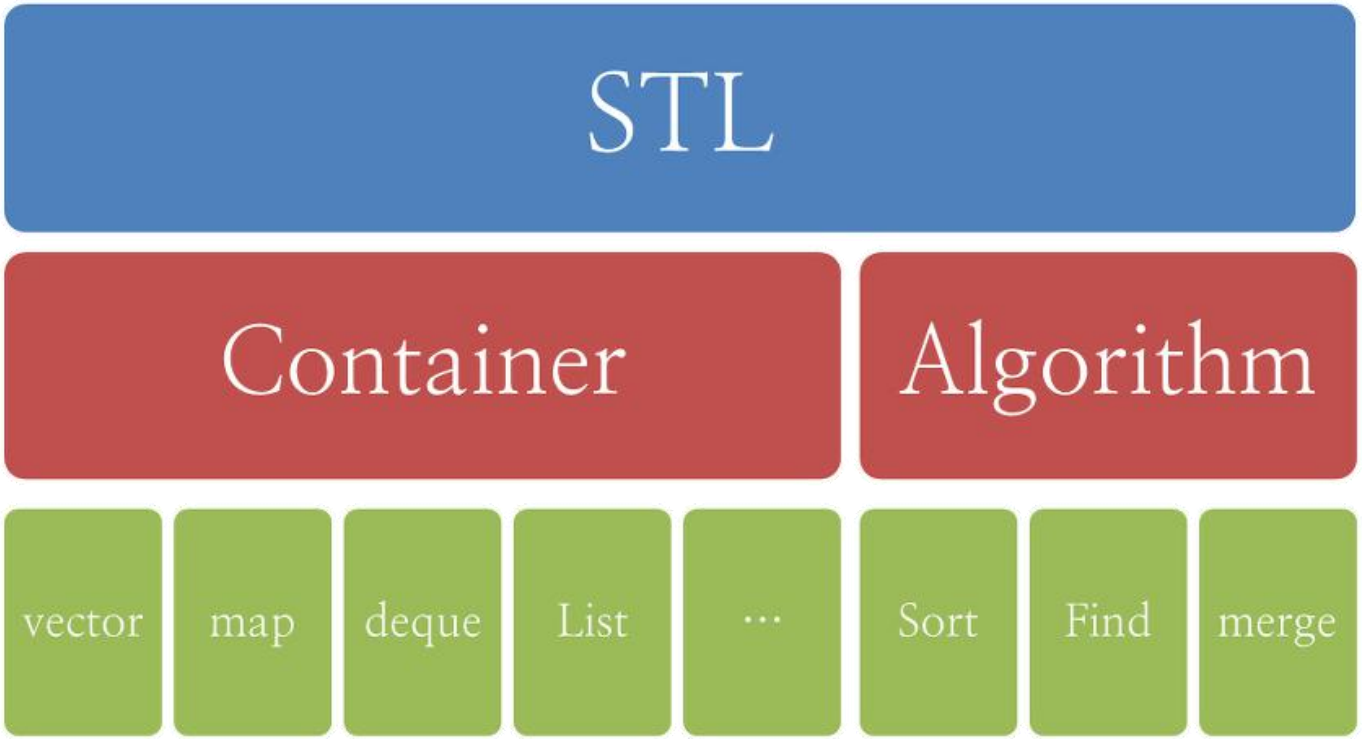
- 任何情况下，都不应出现命名冲突

STL初步

STL简介

- 标准模板库（英文：Standard Template Library，缩写：STL），是一个高效的C++软件库，它被容纳于C++ 标准程序库C++ Standard Library中。其中包含4个组件，分别为算法、容器、函数、迭代器。
- 基于模板编写。
- 关键理念：将“在数据上执行的操作”与“要执行操作的数据”分离。

STL简介


STL
Container
Algorithm
vector
map
deque
List
…
Sort
Find
merge

STL简介

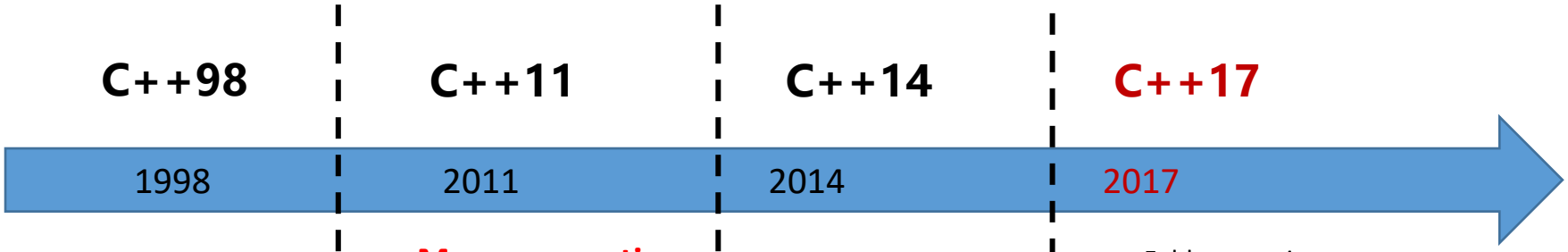
- STL的命名空间是std
  - 一般使用std::name来使用STL的函数或对象
  - 也可以使用using namespace std来引入STL的命名空间（不推荐在大型工程中使用，容易污染命名空间）

- 关于STL的文档和例子可以在以下网址查询

  http://www.cplusplus.com/

  多写多查多用，是学习STL库的最好方法

STL简介



- **containers**
- **algorithms**
- **Strings**
- **I/O Streams**

- **Move semantic**
- Unified initialization
- **auto and decltype**
- **Lambda functions**
- Multithreading
- **Regular expressions**
- **Smart pointers**
- Hash tables
- std::array

- Reader-writer locks
- Generalized lambdas

- Fold expressions
- constexpr if
- Initializers in if and switch statements
- Structured blinding declarations
- Template deduction of constructors
- Guarantees copy elision
- auto_ptr and trigraphs removes
- string_view
- Parallel algorithm of the STL
- The filesystem library
- std::any
- std::optional
- std::variant

STL容器

- 容器是包含、放置数据的工具。通常为数据结构。
- 包括
  - 简单容器（simple container）
  - 序列容器（sequence container）
  - 关系容器（associative container）

STL容器：pair

- 最简单的容器，由两个单独数据组成。

```
template<class T1, class T2> struct pair{
    T1 first;
    T2 second;
    //若干其它函数
};
```

进一步阅读：http://hahaya.github.io/study-std-pair/

- 通过first、second两个成员变量获取数据。

```
std::pair<int, int> t;
t.first = 4; t.second = 5;
```

STL容器：pair

- 创建：使用函数make_pair。

```
auto t = std::make_pair("abc", 7.8);
```

  - 优势：自动推导成员类型。
- 支持小于、等于等比较运算符。
  - 先比较first，后比较second。
  - 要求成员类型支持比较(实现比较运算符重载)。

STL容器：pair举例

- pair使用举例:

```
#include <string>
int main(){
  std::pair<std::string, double> p1("Alice", 90.5);
  std::pair<std::string, double> p2;

  p2.first = "Bob";
  p2.second = 85.0;

  auto p3 = std::make_pair("David", "95.0");
  return 0;
}
```

STL容器：tuple

- C++11新增，pair的扩展，由若干成员组成的元组类型。

```
template< class ... Types > class tuple;
```

- 通过std::get函数获取数据。

```
v0 = std::get<0>(tuple1);
v1 = std::get<1>(tuple2);
```

- 其下标需要在编译时确定：不能设定运行时可变的长度，不能当做数组

STL容器：tuple

- 创建：make_tuple函数

```
auto t = std::make_tuple(“abc”, 7.8, 123, ‘3’);
```

- 创建：tie函数—返回左值引用的元组

```
std::string x; double y; int z;
std::tie(x, y, z) = std::make_tuple(“abc”, 7.8, 123);
//等价于 x = "abc"; y = 7.8; z = 123
```

STL容器：tuple举例

- 用于函数多返回值的传递：

```
#include <tuple>
std::tuple<int, double> f(int x){
    return std::make_tuple(x, double(x)/2);
}
int main() {
    int xval;
    double half_x;
    std::tie(xval, half_x) = f(7);
    return 0;
}
```

- 作为tuple的特例，pair可用于两个返回值的传递
- 除此之外，pair在map中大量使用。

多选题 1分

设置

下面关于pair和tuple描述正确的是：

A tuple的长度可在运行时改变

B pair的两个成员的类型必须相同

C pair之间的大小比较优先级为first>second

D 使用make_pair初始化pair可自动推导成员类型

提交

STL容器：vector

- 会自动扩展容量的数组，以循序(Sequential)的方式维护变量集合。

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

- STL中最基本的序列容器，提供有效、安全的数组以替代C语言中原生数组。
- 允许直接以下标访问。（高速）

STL容器：vector

- 创建：`std::vector<int> x;`
- 当前数组长度：`x.size();`
- 清空：`x.clear();`
- 在末尾添加/删除：（高速）

```
x.push_back(1); x.pop_back();
```

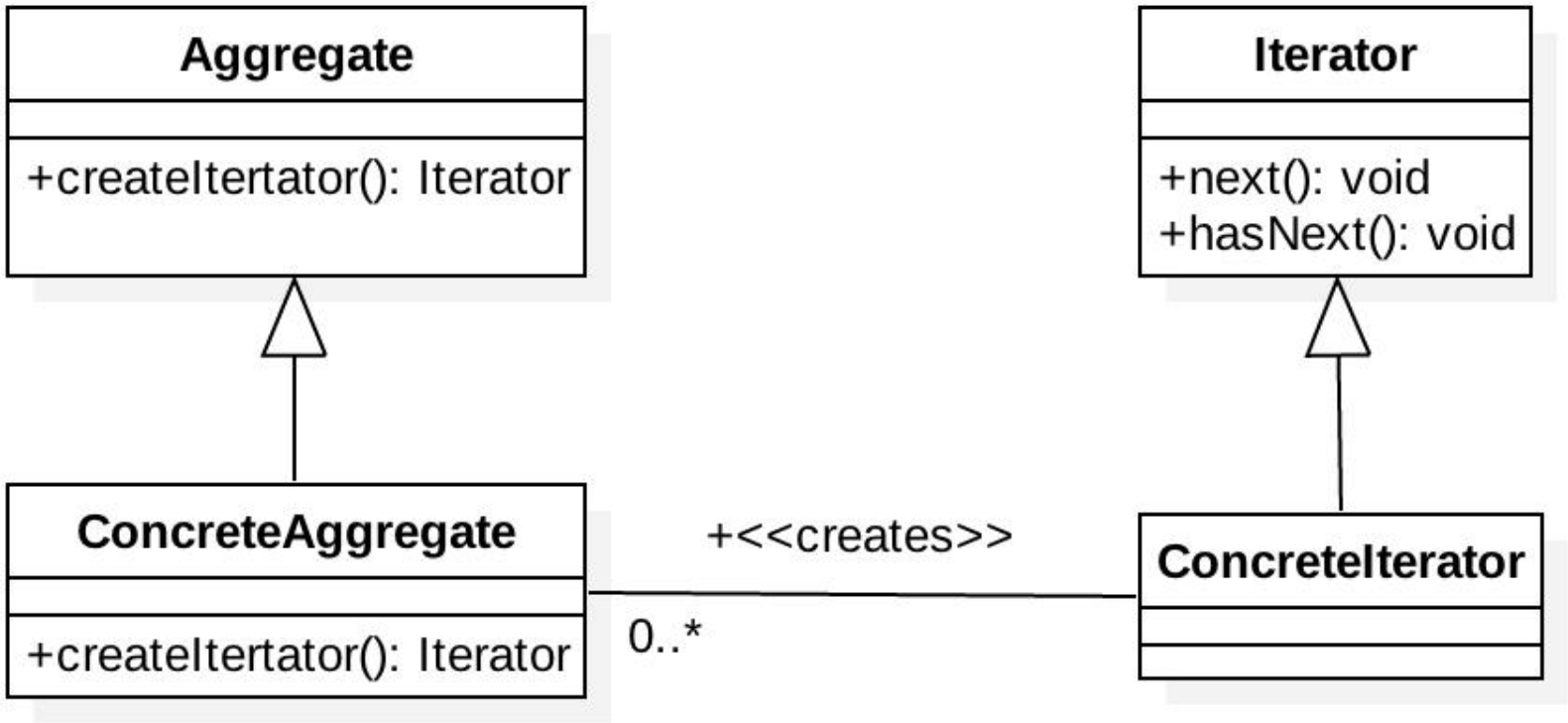
- 在中间添加/删除：（使用迭代器，低速）

```
x.insert(x.begin()+1, 5);
x.erase(x.begin()+1);
```

迭代器

- 一种检查容器内元素并遍历元素的数据类型。
- 提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。
- 为遍历不同的聚合结构（需拥有相同的基类）提供一个统一的接口。
- 使用上类似指针。

迭代器：设计模式


Aggregate
+createItertator(): Iterator
Iterator
+next(): void
+hasNext(): void
ConcreteAggregate
+createItertator(): Iterator
+<<creates>>
0..*
ConcreteIterator

迭代器：以vector为例

- 定义：

```
template<class T, class Allocator =
std::allocator<T>>
class vector {
     class iterator {
          ...
     }
};
```

迭代器：以vector为例

- vector<int>::iterator iter;
- 定义了一个名为iter的变量，它的数据类型是由vector<int>定义的iterator类型。
- begin函数：x.begin()，返回vector中第一个元素的迭代器。
- end函数：x.end()，返回vector中最后一个元素之后的位置的迭代器。
- begin和end函数构成所有元素的**左闭右开**区间。

迭代器：以vector为例

- 下一个元素：`++iter`
- 上一个元素：`--iter`
- 下n个元素：`iter += n`
- 上n个元素：`iter -= n`
- 访问元素值——解引用运算符 `*`

```
*iter = 5;
```

- 解引用运算符返回的是左值引用

迭代器：以vector为例

- 迭代器移动：与整数作加法

```
iter += 5;
```

- 元素位置差：迭代器相减

```
int dist = iter1 - iter2;
```

- 其本质都是重定义运算符

迭代器：以vector为例

- 遍历vector

```
for(vector<int>::iterator it = vec.begin();
    it != vec.end(); ++it) //use *it
```

- C++11中常使用auto替代vector<int>::iterator，以简化代码

```
for(auto it = vec.begin(); it != vec.end(); ++it)
                                    //use *it
```

迭代器：以vector为例

- 完整示例：

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec = {1,2,3,4,5};
    for(auto it = vec.begin(); it != vec.end(); ++it)
        cout << *it << endl;
    return 0;
}
```

运行结果

```
1
2
3
4
5
```

迭代器：以vector为例

- C++11中按范围遍历vector：

```
for(auto x : vec)
 //直接利用vec中元素x
```

- 与以下代码等价：

```
for(vector<int>::iterator it =
vec.begin(); it != vec.end(); ++it)
 //使用 *it，即it是指向元素的指针
```

迭代器：以vector为例

- `auto it = vec.begin();`
- `vec.erase(it);`
- 继续使用it迭代器?

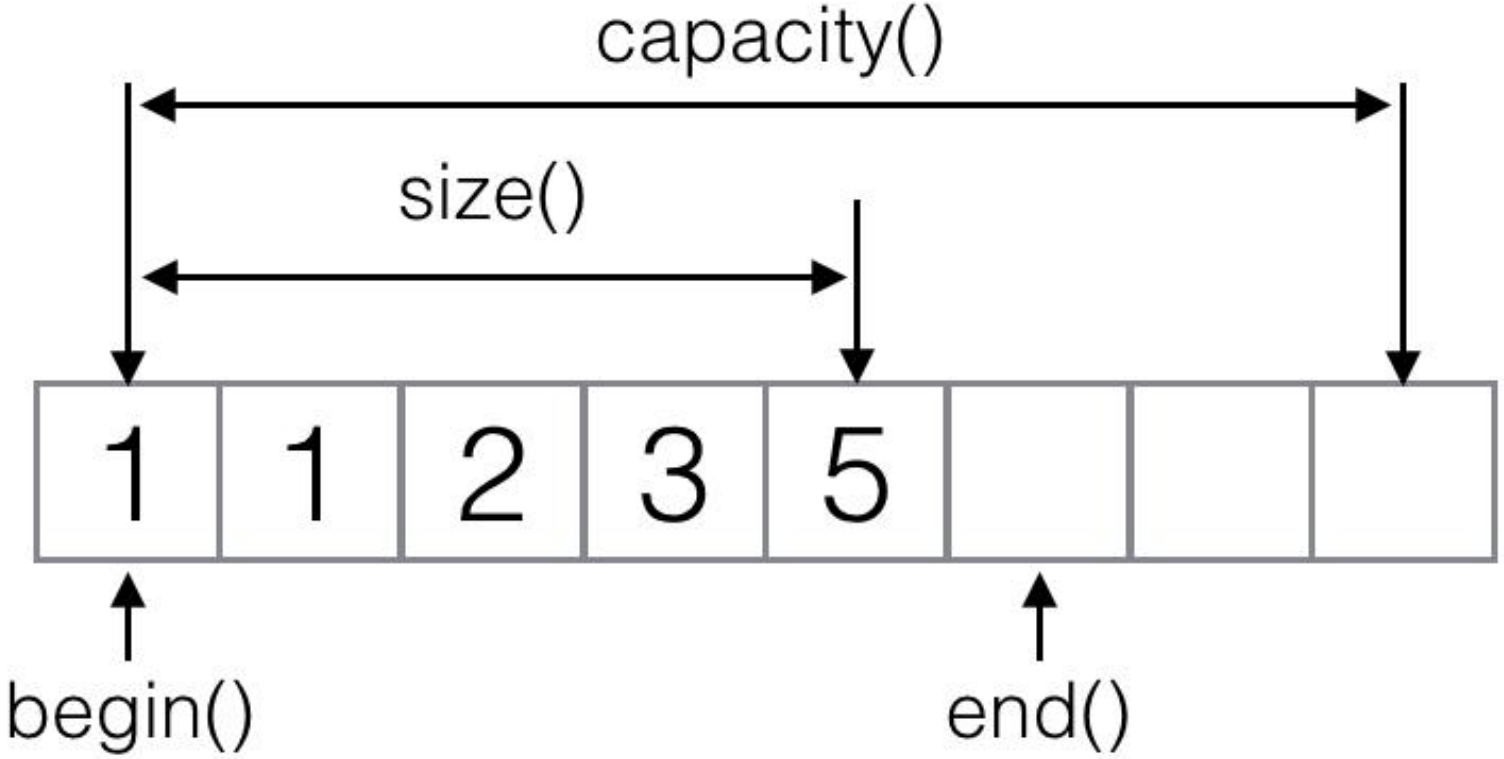
迭代器：失效

- 当迭代器不再指向本应指向的元素时，称此迭代器失效。
- vector中什么情况下会发生迭代器失效?
- 看作纯粹的指针
  - 调用insert/erase后，所修改位置之后的所有迭代器失效。（原先的内存空间存储的元素被改变）
  - 调用push_back等修改vector大小的方法时，可能会使所有迭代器失效（为什么？）

STL容器：vector原理

- vector是会自动扩展容量的数组
- 除了size，另保存capacity：最大容量限制。
- 如果size达到了capacity，则另申请一片capacity*2的空间，并整体迁移vector内容。
- 其时间复杂度为均摊O(1)。
- 整体迁移过程使所有迭代器失效。

STL容器：vector原理

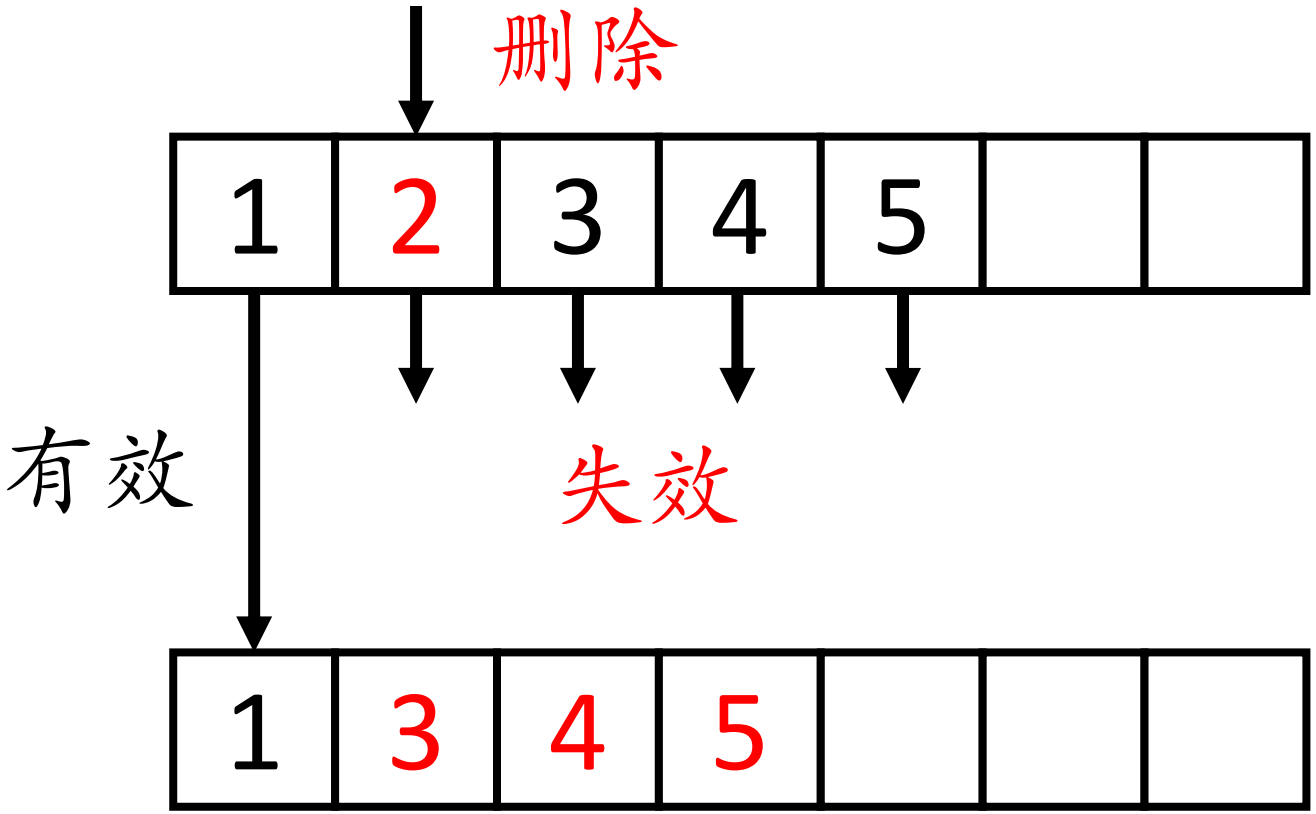


迭代器：失效

- 在遍历的时候增加元素，可能会导致迭代器失效

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
vector<int> vec = {1,2,3,4,5};
   for(auto it = vec.begin(); it != vec.end(); ++it)
      vec.push_back(*it); //Error
   return 0;
}
```

STL容器：vector原理



迭代器：失效

- 使用erase删除元素，被删除元素及之后的所有元素均会失效

```
vector<int> vec = {1,2,3,4,5};
auto first = vec.begin();
auto second = vec.begin() + 1;
auto third = vec.begin() + 2;
auto ret = vec.erase(second);
//first指向1，second和third失效
//ret指向3
```

迭代器：失效

- 迭代器是否会失效，和实现容器的数据结构有关
- 在文档中，容器的修改操作有一项Iterator validity，表示该操作是否会引发迭代器失效
- 一个绝对安全的准则：

  在修改过容器后，不使用之前的迭代器
- 若一定要使用，查文档确定迭代器是否有效

例如：查询push_back对迭代器是否失效的影响
http://cplusplus.com/reference/vector/vector/push_back/

多选题 1分

设置

下面关于vector的相关描述正确的是

A Vector的push_back操作复杂度总为O(1)

B vector 在大小发生改变时，可能致使所有迭代器失效

C 定义vector<int> vec {1,2,3}; 则*vec.end()的值为3

D 当 vector 的 size 达到 capacity，则将 vector 的内容迁移到另外申请的 2*capacity 的空间

提交

STL容器：list

- 链表容器（底层实现是双向链表）

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

- 定义：

```
std::list<int> l;
```

STL容器：list

- 插入前端：

```
l.push_front(1);
```

- 插入末端：

```
l.push_back(2);
```

- 查询：

```
std::find(l.begin(), l.end(), 2); //返回迭代器
```

- 插入指定位置：

```
l.insert(it, 4); //it为迭代器
```

STL容器：list

- 不支持下标等随机访问
- 支持高速的在任意位置插入/删除数据
- 其访问主要依赖迭代器
- 操作不会导致迭代器失效（除指向被删除的元素的迭代器外）

STL容器：set

- 不重复元素构成的无序集合

```
template< class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

- 内部按大小顺序排列，比较器由函数对象Compare完成。
- 注意：无序是指不保持插入顺序，容器内部排列顺序是根据元素大小排列。
- 定义：

```
std::set<int> s;
```

STL容器：set

- 插入：

```
s.insert(1);
```

- 查询：

```
s.find(1);    //返回迭代器
```

- 删除：

```
s.erase(s.find(1));    //导致被删除元素的迭代器失效
```

- 统计：

```
s.count(1);    //1的个数，总是0或1
```

STL容器：map

- 关联数组
- 每个元素由两个数据项组成，map将一个数据项映射到另一个数据项中。

```
template<class Key,
  class T,
  class Compare = std::less<Key>,
  class Allocator =
     std::allocator<std::pair<const Key, T> >
> class map;
```

STL容器：map

- 其值类型为pair<Key, T>。
- map中的元素key互不相同，需要key存在比较器。
- 可以通过下标访问（即使key不是整数）。下标访问时如果元素不存在，则创建对应元素。
- 也可使用insert函数进行插入。

```
#include <string>
#include <map>
int main() {
    std::map<std::string, int> s;
    s["oop"] = 1;
    s.insert(std::make_pair(std::string("oop"), 1));
    return 0;
}
```

STL容器：map

- 查询：find函数，仅需要提供key值，返回迭代器。
- 统计：count函数，仅需要提供key值，返回0或1。
- 删除：erase函数，使用迭代器，导致被删除元素的迭代器失效。
- 以上部分与set类似。

STL容器：map举例

- map常用作过大的稀疏数组或以字符串为下标的数组。

```
#include <string>
#include <map>

int main() {
    std::map<std::string, std::string> M;
    M["fp"] = "c";
    M["oop"] = M["fp"] + "++"; // M["oop"] = "c++"
    return 0;
}
```

STL容器：关联容器原理

- Set和Map所用到的数据结构都是红黑树（一种二叉平衡树）
- 其几乎所有操作复杂度均为O(logn)
- 相关内容将在数据结构课程中学习

STL容器：总结

- 序列容器：vector、list
- 关联容器：set、map
- 序列容器与关联容器的区别：

  序列容器中的元素有顺序，可以按顺序访问。

  关联容器中的元素无顺序，可以按数值（大小）访问。

  vector中插入删除操作会使操作位置之后全部的迭代器失效。

  其他容器中只有被删除元素的迭代器失效。

多选题 1分

设置

下列关于 STL 的说法正确的是：

A vector 在大小发生改变时，可能致使所有迭代器失效

B 为了计算效率，list 的访问主要依赖下标

C 通过下标访问 map<int,int>时，如果元素不存在，程序会抛出异常

D 关联容器(如 map、set)可以使用迭代器访问

提交

单选题 1分

设置

下列从**运行效率**和**使用方式**的角度上考虑，关于容器的选择合理的是：

A 使用下标访问容器中的元素选择 list

B 使用键值对方式访问元素选择 map

C 对容器的中间位置进行插入/删除选择 vector

D 在容器的首尾插入元素选择 vector

提交

课后阅读

- 《C++编程思想》
  - 模板介绍，p400-p435
- 强烈推荐《STL源码剖析》

结束