C++ 11/14/17(Session 6)

Gong xuan Jun.2021

课程内容

- 泛型编程介绍
- STL
  - 容器
  - 算法
  - 迭代器
  - 迭代器适配器
  - Optional
  - variant

泛型编程(Generic Programming)

泛型编程

起 源

- 1989年首次由David Musser和Alexander A. Stepanov提出:

泛型编程是一种编程风格，其中算法以尽可能抽象的方式编写，而不依赖于将在其上执行这些算法的数据形式

- 2011年，Alexander A. Stepanov等出版的<<From Mathematics to Generic Programming>>对泛型编程进行更为精确的定义:

泛型编程是一种专注于对算法及其数据结构进行设计的编程方式，它使得这些算法即数据结构能够在不损失效率的前提下，运用到最为通用的环境中

泛型编程

概 念

- 泛型编程旨在编写独立于数据类型的代码，C++中，完成通用程序的工具是模板(templete)；
- 面向对象编程关注的是编程的数据方面，而泛型编程关注的是算法。它们之间的共同点是抽象和创建可重用代码，但它们的理念完全不同；
- 泛型编程使用某种方式（比如查找函数）来处理数组、链表或任何其他容器类型，即函数不仅独立于容器中存储的数据类型，而且独立于容器本身的数据结构。

与特定数据结构关联的查找函数（示例）

- 模板使得算法独立于存储的数据类型，而迭代器使算法独立于使用的容器类型，查找函数的演进示例：

```
double * findArr(doulbe* ar,int n,const double& val) {//此函数与一种特定的数组关联
    for(int i=0;i<n;i++){
      if(ar[i] == val) return &arr[i];
     return 0;
     }
}
 Node * findList(Node* head,const double& val) {//此函数与一种特定的链表关联
   Node *start;
    for(start=head;start!=0;start=start->pNext{
      if(start->item==val) return start
     return 0;
     }
}
```

通过迭代器改进查找函数（示例）

- 通过迭代器改进查找函数，迭代器应具备哪些特征?

```
typedef double *iterator;
 iterator findArr(iterator begin, iterator end, const double& val) {
   iterator ar;
   for(ar=begin; ar!=end; ar++){
     if(*ar == val) return ar;
    return end;
    }
}
 Iterator findList(iterator* head,const double& val) {//iterator是一个迭代器类
  iterator *start;
   for(start=head;start!=0;++start{
     if(*start==val) return start
    return 0;
    }
}
```

STL(Standard template library)

顺序容器

- STL提供了顺序式容器类(sequence containers)的基本选择集合，这些容器将它们的对象组织到一个严格的线性布局中（类似数组）。主要的顺序式容器有:
  - vector：动态数组，但更安全，能自动增长
  - list：双向链表，不支持随机访问
  - deque: 可从头，尾两端操作
  - forward_list(C++11):是一种单向链表，除最后一个节点外，每个节点都链接到下一个节点
  - array(C++11)： array容器一旦声明，其长度就是固定的，使用静态内存，其受到的限制不比vector多，但效率更好

- 顺序容器类定义示例:

```
template<
   class T, //要保存的元素类型
   class Allocator = std::allocator<T> // 分配器类型
> class vector;
```

关联容器

- STL 提供的关联容器（Associative containers），关联容器不采用线性方式保存，其将键映射到值上，关联容器包括:
  - set
  - multiset
  - map
  - multimap

- STL 关联容器定义示例(map):

```
template<
  class Key, //键类型
  class T,  //值类型
  class Compare = std::less<Key>, //比较类型
class Allocator = std::allocator<std::pair<const Key, T> > //分配器类型
  > class map;
```

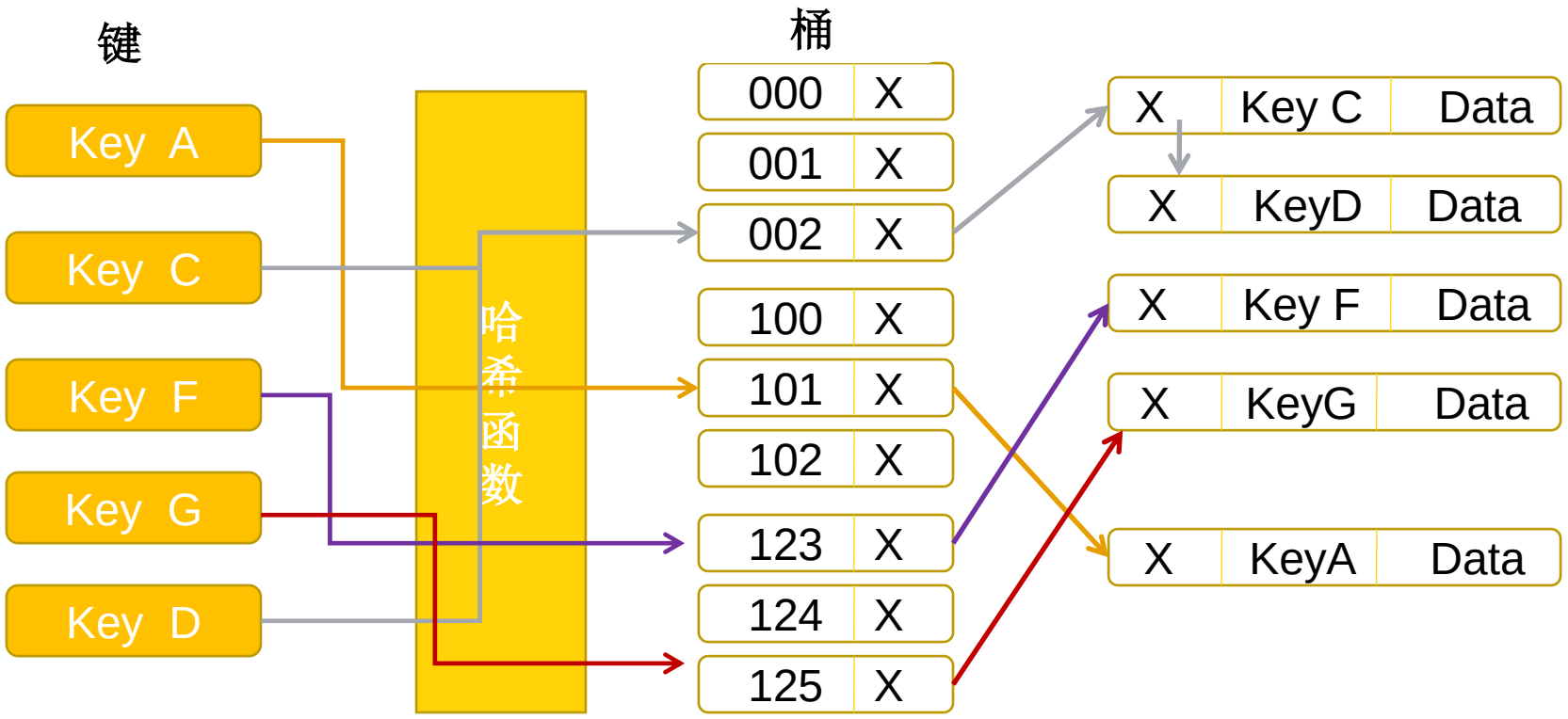
无序关联容器

- 无序关联容器，能够使用键快速检索数据，关联容器使用的底层数据结构为树，而无序关联容器使用的是哈希表;关联容器对元素进行排序，而无序关联容器不会对元素排序;
- 无序关联容器包括:
  - unordered_map(C++11)
  - unordered_multimap(C++11)
  - unordered_set(C++11)
  - unordered_multiset(C++11)
- STL无序关联容器定义示例(unordered_map):

```
template<
class Key,
class T,
class Hash = std::hash<Key>,
class KeyEqual = std::equal_to<Key>,
class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

哈希函数

- 无序关联容器也称为哈希表，因为他们使用了哈希函数;
- 哈希表通常会使用某种形式的数组来实现，数组中的每个元素称为桶(bucket);
- 哈希冲突和查找效率问题;



C++标准提供的哈希函数

- C++标准为指针和所有基本数据类型（bool,char,int,float,double等）提供了哈希函数，例如：

```
template<> struct hash<bool>;
template<> struct hash<char>;
```

也为下面的库类型提供了哈希函数error_code、bitset、unique_ptr、shared_ptr、type_index、string、vector<bool>和thread，例如：

std::hash<std::vector<bool>>

std::hash<std::unique_ptr>

也可编写自定义的哈希函数（如果要使用的键类型没有可用的标准哈希函数）

无序关联容器-underorder_map

- unordered_map定义在头文件<underorder_map>
- 相对于map而言，下表中的操作仅存在于unordered_map中：

unordered_map	
Bucket()	指定键的桶索引
Bucket_count()	容器中桶的数目
Bucket_size()	指定桶的元素数量
Load_factor()	返回每一个桶的平均元素数，以反映冲突次数
Local_iterator/const_local_iterator	遍历单个桶中的元素，但不能用于遍历多个桶
Max_bucket_count()	最大桶的数量
Max_load_factor()	每个桶的最大平均元素数量

STL中的算法概述（一）

- STL中的算法不仅独立于底层元素的类型，还独立于操作的容器类型；
- 算法仅使用迭代器接口执行操作：

- 在使用STL的算法时，关键的问题是使用迭代器指定范围，每个操作接收两个迭代器的开始和末尾，它们表示容器中必须应用这个操作范围.
- 大部分算法都接受回调，回调可以是一个函数指针，内嵌的lambda表达式，函数对象(仿函数functor)

STL中的算法概述(二)

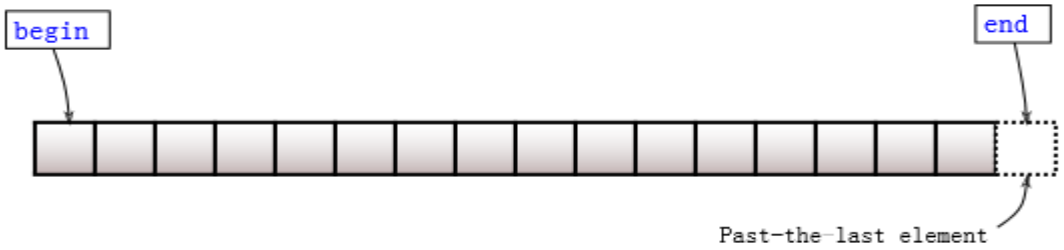
- STL算法库中提供了许多基本的算法（前三组是在头文件algorithm中，第四组在 numeric）：
  - 不修改容器的算法，例如，for_each,find,find_if 等
  - 修改容器的算法，例如， copy,transform等
  - 排序算法，例如， sort等
  - 通用数字运算,主要完成区间的内容累计，计算两个容器的内部乘积等。

迭代器

- 对于STL的每个容器类(vector,list,deque)等定义了相应的迭代器类型，而且迭代器都提供了基本的操作，如*,++等等；
- 每个容器类都有一个超尾标记；
- 使用容器类，无需知道其迭代器是如何实现的，也无需知道超尾是如何实现的，只需知道它有迭代器；
- STL通过为每个类定义适当的迭代器，并以统一的风格设计类，能够对内部表示决然不同的容器，编写相同的代码；
- 所有不同容器的迭代器都遵循C++标准中定义的特定接口，也就是说即使容器提供了不同的功能，访问容器元素的代码也可以使用迭代器的统一接口

迭代器基本算法

- 迭代器重载了特定的运算符，能够进行+、－、++、--、+=、－=、= = 、!=等运算，例如 operator * operator ->的重载；
- 根据迭代器的特点，迭代器又称循环子,其迭代器前闭后开区间 [first, last) 如下图：



迭代器使用示例

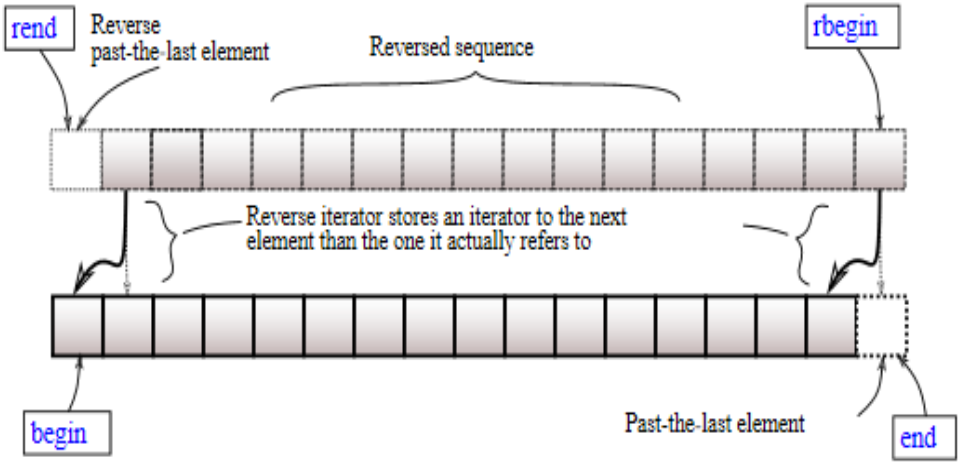
```
int main(void){
  vector<int> vec;
  int element;

 while(cin >> element) {
  vec.push_back(element);
 }

 for(vector<int>::iterator iter=vec.begin();iter !=vec.end();++iter) {
   cout<<"vector iterator"<<*iter<<endl;
 }
 return 0;
}
```

迭代器适配器

- C++标准库提供了4个基于其他迭代器构建的特殊迭代器-迭代器适配器，四个迭代器均在头文件<iterator>中定义：
  - 反向迭代器(reverse_iterator)
  - 流迭代器(stream iterator)
  - 插入迭代器(insert_iterator)
  - 移动迭代器(move_iterator)

反向迭代器(reverse_iterator)

- 反向迭代器以相反的方向遍历双向迭代器或随机访问迭代器
  - STL中所有可反向迭代的容器都提供了一个typedef reverse_iterator以及rbegin()和rend()方法
  - 对reverse_iteratro应用operator++运算符，会对底层容器迭代器调用 operator—运算符



请比较下面两端代码:

```
//从头至尾遍历
 for(auto iter=begin(Arr);iter != end(Arr);++iter){}
//从尾到头遍历
for(auto iter=rbegin(Arr);iter != rend(Arr);++iter){}
```

流迭代器(stream_iterator)

- STL提供的一些适配器允许把输入和输出流用作输入和输出迭代器

流迭代器可以对输入和输出流进行适配：

- 输出流迭代器（ostream_iterator），通过operator<<运算符写入元素
- 输入流迭代器(istream_iterator)，通过operator>>运算符读取元素

插入迭代器(insert_iterator)

- STL提供了三个插入迭代适配器，真正将元素插入容器：
  - insert_iterator 调用容器的insert(position,element)方法
  - back_insert_iterator 调用push_back(element)方法
  - front_insert_iterator 调用push_front(element)方法
- 插入迭代器根据容器类型模板化，在构造函数中接受实际的容器引用；
- 通过提供必要的迭代器接口，这些适配器可用作 copy这类算法的目标迭代器

移动迭代器(move_iterator)

- 移动迭代器(move_iterator)通过引用运算符会自动将值转换为rvalue引用（这个值可以移动到新的目的地，不会有复制开销）
- 使用移动语义前，需要确保对象支持移动语义

  代码示例

迭代器使用示例

- 下面的代码段对输入的数子做排序

```
vector<int> vec;
int inputElement;
while(cin>> inputElement)
vec.push_back(inputElement);
sort(vec.begin(),vec.end());
for(int i=0;i<vec.size();i++)
  cout<<vec[i]<<"\n";
```

- 利用迭代器改进代码：

```
typedef vector<int>  int_vector;
typedef istream_iterator<int> istream_itr;
typedef ostream_iterator<int> ostream_itr;
typedef back_insert_iterator<int_vector>
back_ins_itr;
int_vector vec;

copy(istream_itr(cin),istream_itr(),back_ins_itr(ve
c));
sort(vec.begin(),vec.end());
copy(vec.begin(),vec.end(),ostream_itr(cout,"\n"));
```

Std::optional(C++17)

- 在编程中，函数可能返回/传递/使用某种类型的对象，且对象可能有某个类型的值，也可能没有任何值。两种常见情况：

Case1 字符串转换函数：

{ "42", " 077", "hello", "0x33" } ?


```
int  convertStringToInt(const std::string& s)
```

Q 1：函数对于字符串会有几种转换结果?

Q 2：传统的方式如何解决?

Q 3：有更好的解决方式?

Case 2：作为参数和数据成员：

```
class Employee{
//…
 private:
      std::string employeeName;
      int employeeAge;
      double employeeIncome;
}
Employee  jack{"jack",20,1000};
```

Q 1：要求构造对象时employeeIncome可选，传统的方式如何实现?

Q 2：基于传统的方式如何获取employeeIncome?

Q 3：有更好的解决方式?

代码示例

Std::variant(C++17)

- 传统union存在的问题?
- variant 是一种类型安全的union（union是能够保存可能类型列表之一的对象）；
- 对于每个构建的对象，通常采用默认构造函数初始化第一个类型,下面例子中没有默认构造，编译报错:

```
template <typename T, typename Container>
class Grid{
public:
        Grid(size_t width, size_t height );
//…
};
 std::variant<Grid<int, vector<int>>,std::string> var;
```

Q&A

/