

C++编程实例与数据结构基础

我们已经学完了 C++语言中的重要语法结构。通过练习和作业我们也掌握了具体的编程方法和技巧。本章，我们将使用一个实际例子来看看如何使用以对象为基础的程序设计方法。本章中的例子也可以是对所介绍的 C++的内容的复习和拓展学习。

本章中的例子同时是一个过渡到数据结构课程的例子。简单地说数据结构指的主要是一些常用容器类，数据结构这门课程是介绍各种数据结构的使用、特性、和实现的课程。正确了解和熟练使用各种数据结构大大简化绝大多数的程序的编写。数据结构与面向对象程序课程有基础与应用的关系。本章还将通过例子简单地介绍两种常用的数据结构：集合和关联表。作为这个学期的最后一章，本章起着承前启后和的融汇贯通作用。

1. 面向对象的程序设计方法

我们在最开始就讲过，面向对象的程序设计方法能帮助我们更好地设计程序，尤其是具有一定规模的程序。面向对象的程序设计方法并不是能写出所有程序来解决所有问题的方法：实际上有很多的问题是当前无解的，另外很多问题是很困难的，它们由研究人员通过长时间的探索才找到答案的。这

里我们介绍的程序设计方法指的是在知道程序的算法的情况下，编写出结构清晰，不容易包含错误的程序。

下面我们回忆一下所讨论过的各个 C++ 的语法成分的在程序设计中的作用：

1. 对象 **object**：类（或复合数据类型）的变量，它使的我们可以把每个现实世界中的物体表现为程序里面的一个复合的数据。
2. 封装 **encapsulation**：使每个功能模块的使用方法简单明了，并且消除由用户的错误操作而引起程序的错误。
3. 模版类 **template class**：使得容器可以存放不同类型的元素，避免为不同类型的元素重写相同的容器类而造成程序的冗余。
4. 多态性 **polymorphism**：具有同一个程序接口的多种数据类型可以通过这个接口做相同的处理。避免程序的冗余。
5. 异常处理 **exception handling**：让错误处理更加简单。把错误的报告和错误的处理分离，使得每个错误都可以在任何（最适合处理的）地方处理。
6. 名字空间 **namespace**，静态变量 / 函数 **static variable/function**：避免命名冲突，封装非对象变量 / 函数。

我们曾经通过简单的例子来讨论这些语法结构的用法和技巧。这章中我们使用一个比较长的例子来包含以上内容。这个例子中我们写一个程序，程序的输入使多个事件和它们的先行事件，程序的输出给出一个这些事件的可行发生顺序。输入由文件给出，文件中的每一行描述一个事件。每行中的第一个单词是一个事件，后面各个是这个事件的先行事件。

例子 1：排列课程。

Algorithm	Programming	DiscreteMaths	#		
OperatingSystem	Programming	DataStructure	Algorithm	#	
Programming	#				
ComputerNetwork	Programming	OperatingSystem	#		
DiscreteMaths	#				
DataStructure	Programming	DiscreteMaths	#		
DataBase	Programming	OperatingSystem	##		

每行描述一个课程和它的先修课程。如第一行表示 Algorithm 的先修课程是 Programming 和 DiscreteMaths。每行的#表示本行结束，最后一行中的##表示整个文件的结束。第 3 行中的 Programming 和第 5 行中的 DiscreteMaths 不需要任何先修课程。程序的输出应该是一个课程的排序列表，每个课程的先修课程必须出现在列表的前面。为了简化例子，我们只要求输出其中的一种可行的排序。例如，Programming 和 DicreteMaths 都没有先修课程，输出的时候那个在前面都可以。

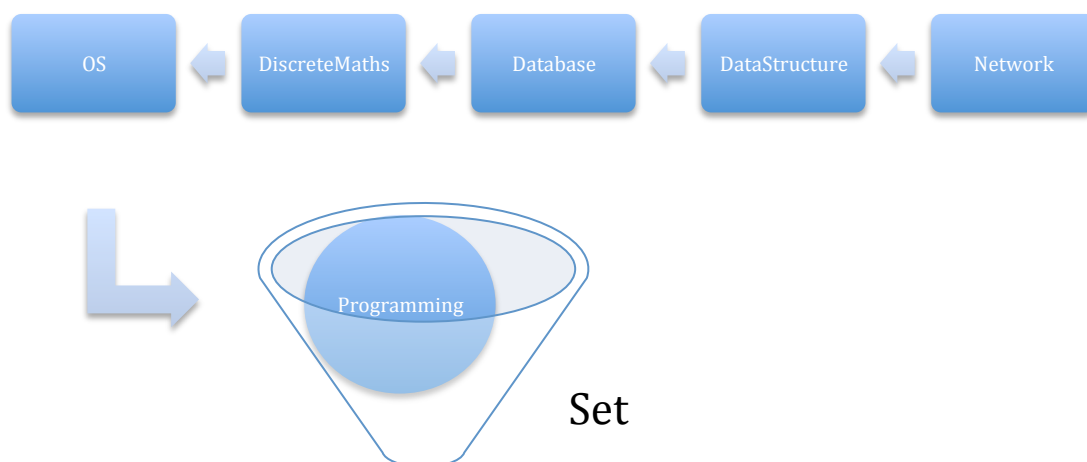
例子 2：学生的生活安排。

basket-ball	submit-maths-hw	submit-C++-hw	#
prepare-C++	review-C++	complete-C++-hw	#
prepare-maths	review-maths	complete-maths-hw	#
submit-maths-hw	register-email	complete-maths-hw	#
submit-C++-hw	complete-C++-hw	#	
review-C++	#		
review-maths	#		
complete-C++-hw	review-C++	#	
complete-maths-hw	review-maths	#	
register-email	#		

例子 2：学生的生活安排。学生给出自己一天当中需要做的事情的优先顺序，程序给出一个可能的先后安排顺序。

在使用面向对象程序(OOP)的方式设计程序之前，我们看看这个程序的算法。我们以排列课程为例。首先，我们把所有课程排成一个队伍。开始的时候，课程在队伍中的顺序可以是随意的，我们就按照文件中的顺序把课程添加到队伍中。另外我们还需要一个记录已经修过了的课程容器。

Queue



计算机擅长执行简单的重复性任务，我们的算法描述如下：

- 1 把所有课程添加到队伍中
- 2 当队伍中有课程的时候
- 3 取出队伍中的第一个课程
- 4 如果这个课程的先修科都以修完
- 5 修这门课
- 6 否则
- 7 把这门课添加到伍中最后

以上的算法模拟一个实际修课的过程：不断尝试每一门课程，如果一门课的先修课都修完那么我们就修这门课，否则把这门课放到队伍最后。要让程序记住那些课已经修过：我们在修没门课的时候，出了把这门课输出到控制台，还把它加入到已经修课程的容器里；以后便可以通过查看这个容器中是否有一门课程来知道该课程是否已修过。下面是更具体的算法：

- 1 把所有课程添加到队伍中
- 2 把已修课程的容器初始化为空
- 3 当队伍中有课程的时候
- 4 取出队伍中的第一个课程
- 5 如果已修课程的容器包含这门课程
- 6 把这门课程添加到已修课程的容器
- 7 在控制台输出这门课程
- 8 否则
- 9 把这门课添加到伍中最后

有了算法，我们开始使用 OOP 的方式设计程序。设计的任务的看看程序需要中使用那些对象，然后看看这些对象之

间的程序接口是什么。了解了算法的整个过程，我们采取自上而下的设计步骤。

首先每个课程的名字用一个字符串对象来表示。这个程序需要使用每个课程的先修课程，我们用一个容器来存储每个课程的先修课程，类型为 `vector<string>`。我们用一个对象表示一个课程的信息，它包括一个课程的名字和这个课程的先修课。课程信息的对象类型如下：

```
1 #include <string>
2 #include <vector>
3 using namespace std;
4
5 class CourseInfo
6 {
7 public:
8     string course;
9     vector<string> prerequisites;
10 };
```

我们还需要一个队伍，在数据结构课程中，有一个重要的数据结构叫做队列 `queue`，它可以作为使用在我们的例子中。我们需要使用的队列应具有以下的接口。

```
1 class Queue
2 {
3 public:
4     CourseInfo getFirst();
5     void addLast(CourseInfo &);
6     bool isEmpty() const;
7 };
```

队列中的元素的类型是 `CourseInfo`。队列需要具有取出

第一个元素的 `getFirst` 函数, 在队列最后添加元素的 `addLast` 函数, 和判断队列是否为空的 `isEmpty` 函数。为了以后可以更好地重用我们的代码, 把队列写成模版类如下:

```
1 template <typename E>
2 class Queue
3 {
4 public:
5     E getFirst();
6     void addLast(const E &);
7     bool isEmpty() const;
8 };
```

我们还需要一个记录已修课程的容器。容器的元素为课程的名字, 我们用字符串对象表示。这个容器能加入元素, 和能查询元素是否已加入该容器。这种容器在数据结构课程中叫做集合 set。集合中只关心元素是否存在, 而不关心原属在容器中的顺序。集合中的相同元素不必重复存储。程序中所需的集合对象具有以下接口:

```
1 class Set
2 {
3 public:
4     void add(const string &);
5     bool contains(const string &) const;
6 };
```

集合中元素的类型是 `string`。集合中添加元素的函数是 `add`。查询给定元素是否在集合中的函数是 `contains`。把集合写成容器类如下:

```

1 template <typename E>
2 class Set
3 {
4 public:
5     void add(const E &);
6     bool contains(const E &) const;
7 };

```

我们还需要一个读文件的对象，把课程信息从文件中读取出来，并添加到队列中。这对象的接口如下：

```

1 class FileReader
2 {
3 public:
4     Queue<CourseInfo> readFile(const char filename[]);
5 };

```

分析完程序中所需要的对象与它们的程序接口后，前面的伪代码可以使用以下的程序实现：

```

1 int main() {
2     FileReader filereader;
3     Queue<CourseInfo> courseQueue =
4         filereader.read("1.txt");
5     Set<string> coursesTaken;
6
7     while (! courseQueue.isEmpty()) {
8         CourseInfo courseInfo = courseQueue.getFirst();
9         if (containsAll(coursesTaken,
10             courseInfo.prerequisites)) {
11             coursesTaken.add(courseInfo.course);
12             cout << courseInfo.course << endl;
13         }
14         else {
15             courseQueue.addLast(courseInfo);
16         }
17     }
18 }

```

第 8-9 行我们需要判断已修的课程 `coursesTaken` 是否包

含给定的课程 `courseInfo` 中的所有先修课程 `courseInfo.prerequisites`。我们需要写一个函数，这个函数逐个判断 `courseInfo.prerequisites` 中的每个课程是否在已修课程 `coursesTaken` 当中。程序如下：

```
1  /*
2   Return true if every elements in prerequisites
3   is also in courseTaken.
4   */
5  bool containsAll(Set<string> & coursesTaken,
6                  vector<string> prerequisites)
7  {
8      for (int i = 0; i < prerequisites.size(); ++ i) {
9          string prerequisite = prerequisites[i];
10         if (! coursesTaken.contains(prerequisite)) {
11             return false;
12         }
13     }
14     return true;
15 }
```

下面，我们我们看看下层程序的类如何实现，它们包括 `Queue`、`Set` 和 `FileReader`。首先我们讨论队列 `Queue` 的实现。在 C++ 中标准类库当中提供一个队列的实现，名为 `queue`。C++ 标准类库中的 `queue` 与我们的 `Queue` 的接口是不同的。我们可以修改 `main` 函数，把关于 `Queue` 的语句改为对应 `queue` 的使用的语句。我们选择通过封装 `queue` 来实现 `Queue`，以换取更好的程序可读性。我们的 `Queue` 实现如下：

```

1  #include <queue>
2  using namespace std;
3
4  template <typename E>
5  class Queue
6  {
7  private:
8      queue<E> queueImpl;
9  public:
10
11      class EmptyQueueException {};
12
13      bool isEmpty() const {
14          return queueImpl.empty();
15      }
16
17      E getFirst() {
18          if (isEmpty()) {
19              throw EmptyQueueException();
20          }
21          E element = queueImpl.front();
22          queueImpl.pop();
23          return element;
24      }
25
26      void addLast(const E & element) {
27          queueImpl.push(element);
28      }
29
30 };

```

C++标准类库中的 `queue` 具有函数 `push` 和 `pop`，它们分别把元素添加到队列最后和把容器从队列前面取出来。但是 `pop` 本身并不返回队列前面元素的值，我们需要使用函数 `front` 来获得它。

对于数据结构 `Set`，C++的标准库中没有提供有效的实现方法。C++的标准库中有一个也叫做的类 `set`，但是这个类只

提供元素遍历的功能，不提供我们所需要的查询功能。实际上有多种非常有效的实现方法，我们在下一小节将会介绍一种名叫哈希集合 `HashSet` 的实现方法。在这里我们提供一种直观的实现方法如下：

```
1 template <typename E>
2 class Set
3 {
4 private:
5     vector<E> elements;
6
7     int find(const E & element) const {
8         for (int i = 0; i < elements.size(); ++ i) {
9             if (elements[i] == element) {
10                 return i;
11             }
12         }
13         return -1;
14     }
15
16 public:
17     void add(const E & element) {
18         if (find(element) == -1) {
19             elements.push_back(element);
20         }
21     }
22
23     bool contains(const E & element) const {
24         return (find(element) != -1);
25     }
26 };
```

最后，我们给出 `FileReader` 的实现。

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 class FileReader
6 {
```

```

7 public:
8     class FileException {};
9
10    Queue<CourseInfo> read(const char filename[]) {
11        ifstream input(filename);
12        if (input.fail()) {
13            throw FileException();
14        }
15
16        Queue<CourseInfo> courseQueue;
17        while (true) {
18            CourseInfo courseInfo;
19            string word;
20            input >> word;
21            courseInfo.course = word;
22            while (true) {
23                input >> word;
24                if (word == "#" || word == "##") break;
25                courseInfo.prerequisites.push_back(word);
26            }
27            courseQueue.addLast(courseInfo);
28            if (word == "##") break;
29        }
30        input.close();
31        return courseQueue;
32    }
33 };

```

2. 数据结构基础

数据结构课程将介绍常用的数据结构和它们的高效的实现方法。掌握了这些常用的数据结构和它们的运行效率，并知道何时应该使用何种数据结构，我们写的程序就可以更简洁和高效。下面我们将介绍两个常用数据结构的实现。

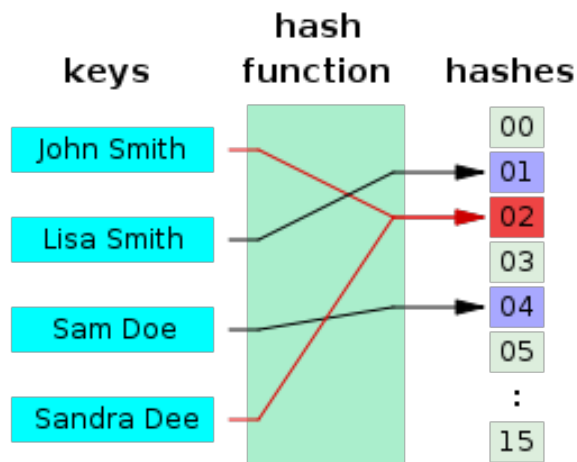
2.1 集合 Set

```
1 template <typename E>
2 class Set
3 {
4 public:
5     virtual bool add(const E &) = 0;
6     virtual bool contains(const E &) const = 0;
7     virtual bool remove(const E &) = 0;
8     virtual void clear() = 0;
9     virtual int size() const = 0;
10    virtual vector<E> toVector() const = 0;
11 };
```

以上是一个更完整的 set 类的接口。这个 set 的接口中多了 **remove** 函数和 **size** 函数。**remove** 函数从集合中取出一个元素。**size** 函数返回集合中的元素的个数。**clear** 函数清除集合中的所有元素。**toVector** 函数返回一个存储集合中所有元素的 **vector** 对象,元素在这个 **vector** 对象中的顺序是不确定的。

我们要介绍的 set 的高效实现方法叫做哈希集合 hash-set。为什么我们要引入 hash-set 而不使用之前使用 **vector** 的集合的实现呢？这是因为，hash-set 的查找速度非常快。那么如何评价一个算法的快慢呢？我们一般使用最坏情况 worst-case下的速度，也就是当 set 中的元素非常多的时候查找的数度。因为如果数据量较少的情况下，所有算法都很快。如果之前的使用 **vector** 实现的集合中具有 10^8 个元素的时候，那么查找一个存在的元素平均要读取 5×10^7 个元素，查找一个不存在的元素要读取所有的 10^8 个元素。这是一般程序不能接受的。

而我们将要介绍的 `hash-set` 查找一个元素平均只需要读取两个元素。



在介绍 `hash-set` 之前，我们先介绍 哈希函数 `hash-function`。Hash 是打乱的意思，`hash-function` 把对象映射为具有固定长度的整数。`Hash-function` 被广泛应用于加速数据的精确或精确查找，是当前一个热门的研究方向。`hash-set` 使用的 `hash-function` 比较简单，只需要把对象映射为整数就可以了，相同的对象必须映射到相同的整数。

对于 OOP，设计程序的时候，`everything is an object`，函数也不例外。注意这里的函数是指外部世界的数学函数而不是 C++ 本身的函数。以下是 `hash-function` 的接口。

```
1 template <typename E>
2 class HashFunction
3 {
4 public:
5     virtual int getHashCode(const E & e) const = 0;
6 };
```

对于 *hash-set*，好的 *hash-function* 能把集合中的元素均匀地映射到某个整数范围以内。这种函数通常不难找。我们直接给出一个把 *string* 对象映射为整数的 *hash-function* 的实现。具体方法在学习数据结构的时候会详细介绍。

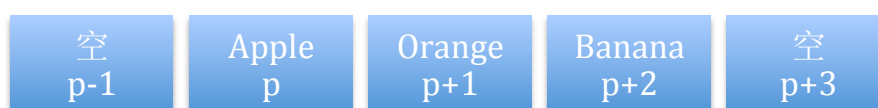
```
1 class StringHashFunction : public HashFunction<string>
2 {
3 public:
4     int getHashCode(const string & text) const {
5         int code = 0;
6         for (int i = 0; i < text.size(); ++ i) {
7             int code1 = text[i];
8             code1 = code1 << (i * 8 % 24);
9             code = code ^ code1; // exclusive or
10        }
11        return code % 181325909; // % a large prime
12    }
13 };
```

有了把元素映射为整数的 *hash-function* 后，我们来介绍 *hash-set*。*hash-set* 是一种以浪费空间换取快的速度的方法，*hash-set* 中，容器中的大部分位置必须是空的。*hash-set* 查找的基本思路如下：假设容器的存储空间大小为 N ，*hash-function* 应用在元素 "apple" 的时候返回 h ，那么我们计算出一个位置 $p=h\%N$ 。如果这个位置没有，那么，我们把元素 "apple" 放在容器中的位置 p 。当以后查找元素 "apple" 的时候，我们用同样的方法计算 p ，然后我们就可以马上找到这个元素了。如果这个位置为空，表示元素 "apple" 不在集合中。

但是，问题没有这么简单，因为不同的元素可能被映射

到同一个 p ，这种情况叫做冲突。如果容器中元素的总个数为 K ，两个不同的元素冲突的概率是 K/N 。解决的冲突方法是把元素往后挪：当加入一个元素的时候，如果位置 p 已经有别的元素，那么尝试把元素放到位置 $p+1$ ，如果位置有别的元素那么继续尝试位置 $p+2$ ，直到找到一个没有存放元素的位置 $p+k$ 。当元素的总个数 $K=N/2$ 的时候，平均情况下插入一个元素要尝试 2 次。

当查找一个元素的时候，如果位置 p 有元素但不是要找的元素，那么尝试查找位置 $p+1$ ，直到在位置 $p+k$ 找到该元素或则发现位置 $p+k$ 没有元素。在发现位置 $p+k$ 没有元素的时候，我们直到该元素不在 **set** 中。当元素的总个数 $K=N/2$ 的时候，平均情况下查找一个元素要尝试 2 次。



最后是删除元素的操作。首先要查找待删除元素在容器中的位置 p' ，如果该元素不存在则无需删除。如果存在则把位置 p' 标志为空。但是这可能导致错误。如上如所示，如果 3 个元素都被映射到位置 p ，根据按其加入顺序，它们的存放情况可能如上图所示。如果我们在删除 **Orange** 的时候仅仅把它对应的位置设为空，那么以后查找 **Banana** 的时候就会错误地认为它不存在。解决的思路是通过把后面的元素向前挪，把

这些不正确的空的位置填补掉。当元素的总个数 $K=N/2$ 的时候，平均情况下需要填补少于 2 次。

从上面我们看到 **hash-set** 的各种操作都非常快，而且与容器中元素的个数无关。但是，我们要注意，上面提到的所有的操作访问元素只需平均 2 次的一个前提条件是：元素的个数必须是容器存储空间大小的一半 $K=N/2$ 。为了能使 **hash-set** 保持这样高的速度，我们在插入元素后可能要扩大容器存储空间。申请到新的存储空间后，并不能把元素拷贝到原来的位置。这是因为在新的存储空间内 N 变了，所以各个元素的位置 p 也变了。方法是首先把新的存储空间全部置空，然后把元素按照 **hash-set** 加入元素的方法一个一个加入到容器中。当然，需要重新分配空间会带来额外的开销。但我们知道，这是很多容器，如 **vector**，都不可避免的问题。而且，我们从 **vector** 的学习中知道：当新的空间只能成倍地增长的时候，平局的拷贝开销其实并不大。

关于 **hash-set** 的理论我们讨论到这里，下面我们看看它的实现细节。首先，**hash-set** 容器中通过动态分配存储空间来存放元素。这时，动态分配的存储空间中每个位置的类型不再直接是元素的类型。这是因为我们需要知道每个位置是空的还是已被使用。我们使用类 **Entry** 表示动态分配的存储空间中每个位置的类型。看下面的程序的开始部分。

```

1 template <typename E, typename H = HashFunction<E> >
2 class HashSet : public Set<E>
3 {
4 private:
5
6     const H hashFunction;
7
8     class Entry
9     {
10    public:
11        E element;
12        bool isInUse;
13
14        Entry() {
15            isInUse = false;
16        }
17    };
18
19    Entry * entries;

```

容器类是一个模版类，该类有两个参数：第一个是容器中元素的类型，第二个参数表示 `HashSet` 中所使用的 `HashFunction` 的类型。我们看到，模版类的类型参数也可以有默认参数，如本例中的第二个类型参数。第二个类型参数的默认值 `HashFunction<E>` 是一个类型 `E` 的哈希函数，由于类这个型参数的默认值 `HashFunction<E>` 是一个抽象类，实际使用的时候还是不能缺少这个类型参数的。这中情况下，实际的类型参数一般需要是这个类型参数的一个非抽象的子类。如上面所给出的 `StringHashFunction`。

在第 2 行中，我们定义了类型 `HashSet`，并把它制定为 `Set` 的子类。注意，在 `HashSet` 的内部，C++ 规定要省略它的所有类型参数，否则应该写为 `HashSet<E, H>`。而对于父类 `Set`

类型参数是不能省略的，而且元素的类型与 `HashSet` 同为 `E`，所以写为 `Set<E>`。第 6 行中定义了一个 hash-function 的对象，用来产生对象的 hash-code。第 8-17 行定义了动态分配的存储空间中每个位置的类型 `Entry`。每个 `Entry` 的对象有 2 个成员变量，类型为 `E` 的成员变量用来存储放在这个位置中的元素的值，另一个 `bool` 类型的成员变量表示这个位置是空的还是正在被使用。在 19 行中定义的 `Entry *`的地址变量用来存储动态分配的存储空间的地址。

```
21     int capacity;
22     int count;
23
24     void initialize(int capacity2) {
25         count = 0;
26         capacity = capacity2;
27         entries = new Entry[capacity];
28     }
29
30     void assign(const HashSet & set2) {
31         count = set2.count;
32         capacity = set2.capacity;
33         entries = new Entry[capacity];
34         for (int i = 0; i < capacity; ++ i) {
35             entries[i] = set2.entries[i];
36         }
37     }
```

第 21-22 行定义的成员变量，分别用来存储动态分配的存储空间的大小和实际存储的元素的大小。函数 `initialize` 初始化一个给定存储大小的空的存储空间。函数 `assign` 拷贝另一个 `HashSet` 的对象。

```

39 public:
40
41     HashSet() {
42         initialize(2);
43     }
44
45     ~HashSet() {
46         delete [] entries;
47     }
48
49     HashSet(const HashSet & set2) {
50         assign(set2);
51     }
52
53     HashSet & operator = (const HashSet & set2) {
54         delete [] entries;
55         assign(set2);
56         return (*this);
57     }
58
59     void clear() {
60         delete [] entries;
61         initialize(2);
62     }

```

41-57 行定义了我们收悉的“四大”函数。函数 `clear` 的作用的晴空 `hash-set` 里面的所有数据。它首先把原来的存储空间释放掉，然后调用函数 `initialize` 初始化新的空间。

```

64 private:
65
66     int hashIndex(const E & e) const {
67         int hashCode = hashFunction.getHashCode(e);
68         return hashCode % capacity;
69     }

```

66 行的函数返回元素的通过 `hash-function` 计算的容器

中的位置。67 行通过调用 hash-function 对象 `hashFunction` 的函数 `getHashCode` 获得元素 `e` 的 hash-code。但是这个 hash-code 是一个任意大的整数，它的范围可能超出容器的容量。因此 68 行把它对容器的大小求模（求余），得到一个容器中的位置。`hashIndex` 的返回值就是前面讨论过的 $p=h\%N$ 。

```
71     int find(const E & e) const {
72         int index = hashIndex(e);
73         while (true) {
74             if (! entries[index].isInUse) {
75                 return index;
76             }
77             if (entries[index].element == e) {
78                 return index;
79             }
80             index = (index + 1) % capacity;
81         }
82     }
```

函数 `find` 返回函数 `hashIndex` 返回的位置后面的第一个为空的或则存储着元素 `e` 的位置。对这个位置的寻找是在容器中循环的：再 80 行中可以看到，如果当前的位置是容器的最后一个位置，那么下一个寻找的位置将是容器的第一个位置。

```
84     void resize(int capacity2) {
85         Entry * oldEntries = entries;
86         int oldCapacity = capacity;
87         initialize(capacity2);
88         for (int i = 0; i < oldCapacity; ++ i) {
89             if (oldEntries[i].isInUse) {
90                 add(oldEntries[i].element);
91             }
92         }
93         delete [] oldEntries;
94     }
```

函数 **resize** 改变容器的存储空间,它会在后面的函数 **add** 中用来保证 $K < N/2$, 在函数 **remove** 中在的 $K < N/2$ 前提下减少存储空间。这个函数的首先原来的存储空间的地址, 然后初始化一个新的大小的存储空间, 最后把原来存储空间中的元素通过调用函数 **add** 添加回新的存储空间中。

```
96 public:
97
98     bool add(const E & e) {
99         int index = find(e);
100         if (entries[index].isInUse) {
101             return false;
102         }
103         entries[index].element = e;
104         entries[index].isInUse = true;
105         ++ count;
106         if (count > capacity / 2) {
107             resize(capacity * 2);
108         }
109         return true;
110     }
```

函数 **add** 把元素 **e** 添加到离 hash-function 指定的位置最近的可能位置。实际上这个位置已经由函数 **find** 找到。函数 **find** 返回的位置要么是 hash-function 指定的位置后面第一个空的位置, 要么是一个存储元素 **e** 的位置。当函数 **find** 返回的位置为非空的时候, 表示容器中已有元素 **e**, 且这个位置必定存储元素 **e**。这种情况下, 函数返回 **false** 表示添加失败, 如 100-102 行所示。103-105 行把返回的位置设为元素 **e**, 把这个位置设定为已使用, 并增加容器元素个数的计数器 **count**。106-108 行判断容器的剩余空间是否足够, 以保证以后的操作

能具有较高的速度；并在不够时调用函数 `resize` 以调整空间的大小。

```
112     bool contains(const E & e) const {
113         int index = find(e);
114         return (entries[index].isInUse);
115     }
116
117     int size() const {
118         return count;
119     }
```

函数返回容器中是否包含给定的元素。这个函数调用函数 `find` 实现：函数 `find` 返回一个空的位置表示容器中不包含元素 `e`。函数 `size` 返回容器中实际存储的元素的个数。

```
145 public:
146     bool remove(const E & e) {
147         int index = find(e);
148         if (! entries[index].isInUse)
149             return false;
150         fillNotInUseEntry(index);
151         -- count;
152         if (count < capacity / 4) {
153             resize(capacity / 2);
154         }
155         return true;
156     }
```

函数 `remove` 删除容器中的给定元素 `e`。如果元素不存在则返回 `false`，如第 149 行所示。然而，删除一个元素并不能直接把元素所在的位置置空，这可能使得容器丢失：函数 `find` 的算法时搜索到空的位置则停止。函数调用一个我们下面会讨论的函数 `fillNotInUseEntry` 来删除元素。当删除后如果容

器中的空位太多，则调用函数 `resize` 调整容器的大小。

我们知道当一个元素的 `hash-function`（求模后的）指定的位置由于冲突被别的元素占用了的时候，这个元素必须向后移位。后移的元素也会占用别的位置，造成其它的冲突和位移。当一个在位置 `index` 元素要被删除的时候，要是把位置 `index` 直接置空，那么由位置 `index` 后移的元素将不能被函数 `find` 找到。我们可以把放在位置 `index` 后面的、并且从位置 `index` 前面移位到 `index` 后面的元素补充 `index` 的空位。如果 `index` 后面没有这样的元素，则可以直接置空位置 `index`。

```
121 private:
122     void fillNotInUseEntry(int index) {
123         int next = index;
124         while (true) {
125             next = (next + 1) % capacity;
126             if (! entries[next].isInUse) {
127                 entries[index].isInUse = false;
128                 return;
129             }
130             int index0 = hashIndex(entries[next].
131                                     element);
132             if (index < next) {
133                 if (index0 > index &&
134                     index0 <= next) continue;
135             }
136             else {
137                 if (index0 > index ||
138                     index0 <= next) continue;
139             }
140             entries[index] = entries[next];
141             index = next;
142         }
143     }
```


函数 `fillNotInUseEntry` 的算法是从位置 `index` 开始扫描至下一个空位（这个空位可在第 127-129 行被发现）。如果扫描到位置 `next` 并发现这里的元素的 `hash-function`（求模后的）指定的位置是 `index0`（第 130-131 行），并且 `index0 ≤ index`（第 121-139 行考虑更复杂的情况：函数 `find` 的查找会在碰到容器末尾的时候回到位置 0 继续查找），那么就把位置 `next` 中的元素挪到位置 `index` 以补充空位（第 140 行）。这时我们产生了一个新的空位 `next`。我们使用同样的方法去处这个新的空位（第 141 行）。

```
158     vector<E> toVector() const {
159         vector<E> vec;
160         for (int i = 0; i < capacity; ++ i) {
161             if (entries[i].isInUse) {
162                 vec.push_back(entries[i].element);
163             }
164         }
165         return vec;
166     }
```

最后函数 `toVector` 返回一个存储所有元素的向量。

2.1 关联表 Associative array / dictionary / map

我们将介绍一个最常用的数据结构：关联表 associative map。它具有很多个同义的名字，如字典 dictionary、映射 map。我们使用当前比较流行的一个名字：`map`，并且将介绍一种很高效的实现方法：哈希表 hash-map (hash-table)。`Hash-map` 的实现与 `hash-set` 的实现几乎相同。`Map` 最直观的

应用就是字典，它能通过一个键 key（如一个英文单词）访问一个值 value（这个英文单词所对应的中文解释）的数据结构。

Map 的接口如下：

```
1 template <typename K, typename V>
2 class Map
3 {
4 public:
5     virtual void put(const K &, const V &) = 0;
6     virtual bool containsKey(const K &) const = 0;
7     virtual V get(const K &) const = 0;
8     virtual bool remove(const K &) = 0;
9     virtual void clear() = 0;
10    virtual int size() const = 0;
11    virtual vector<K> getKeys() const = 0;
12 };
```

模版类接口 Map 有两个类型参数，k 是键 key 的类型，v 是值 value 的类型。函数 **put** 把一个键-值对 key-value pair 添加到容器中。函数 **containsKey** 检查容器中是否有给定的 key 值的 key-value pair。函数 **get** 在容器中存在给定的 key 值的 key-value pair 的时候返回 value 的值，如果容器中不存在给定的 key 值的 key-value pair 则抛出异常。函数 **remove** 删除容器中给定的 key 值的 key-value pair。函数 **clear** 清空容器中的所有元素。函数 **size** 返回容器中元素的个数。最后函数 **getKeys** 返回一个存储所有 key 值的向量。

首先，我们看看 **Map** 的使用。在下面的例子中我们定义一个 **Map** 用它来表示一个子字典 **dictionary** 的对象。这个对象的 key 和 value 的类型都是 **string**。由于我们还没有给出

Map 的实现，我们在第 2 行中只能定义它的引用，然后假设这个应用能存储一个对象。接着 5-7 行通过调用函数 **put** 在这个表示字典的容器中添加了 3 对中英文。最后 10-15 行循环地读入英文单词，然后在字典中去处对应的中文解释。当然，如果要使这个字典有用，我们可以把 5-7 行替换为从某个字典文件中读入大量的中英文对的程序。

```
1 int main() {
2     Map<string, string> & dictionary = ...
3
4     // 插入字典中的中英文单词对
5     dictionary.put("apple", "苹果");
6     dictionary.put("orange", "橙子");
7     dictionary.put("banana", "香蕉");
8
9     // 通过英文查中文
10    while (true) {
11        string english;
12        cin >> english;
13        string chinese = dictionary.get(english);
14        cout << chinese << endl;
15    }
16 }
```

Map 的实现的一种直观的方法是把所有的 **key** 存放在一个 **vector** 中，把所有的 **value** 存放在另一个 **vector** 中的对应位置。当我们向通过一个 **key** 的值查找对应的 **value** 的值的时候，我们可以先搜索这个 **key** 在第一个 **vector** 中的位置，然后返回第二个 **vector** 中对应位置中的 **value** 的值。如果使用一种比较高效的存储方式 **heap**（**heap** 本身是一种数据结构，本章中不讨论）在 **vector** 中存放 **key**，那么插入和删除需要

读取或移动 $\log(K)$ 个数据，其中 K 是容器中元素的个数。实际上标准 C++ 中的 `map` 的实现使用一种叫做平衡二叉树的实现方法，它的效率与上述实现方法相同。我们将要介绍的 Hash-map 实现方法具有更高的效率：所有的操作都只需要 2 个数据数据的读取或移动。而且 hash-map 的实现与 hash-set 基本相同，因此非常容易理解。主要的不同是我们把 hash-set 中每个位置 `Entry` 中原来的一个元素 `element` 改为一对的 `key-value`。

```
1 class NoSuchKeyException {};  
2  
3 template <typename K, typename V,  
4           typename H = HashFunction<K> >  
5 class HashMap : public Map<K, V>  
6 {  
7 private:  
8  
9     const H hashFunction;  
10  
11     class Entry  
12     {  
13     public:  
14         K key;  
15         V value;  
16         bool isInUse;  
17  
18         Entry() {  
19             isInUse = false;  
20         }  
21     };
```

我们把改动了的地方用高亮显示。`HashMap` 的类型参数比 `HashSet` 的类型参数对了 一个，与 `Map` 的类型参数相比对了一个 `hash-function` 的类型参数。在 11-21 行的类 `Entry` 与

HashSet::Entry 相比, 把 Element 改为了 K key 和 V value。

即容器里每个位置从一个元素替换为一个 key-value pair。

```
23     Entry * entries;
24
25     int capacity;
26     int count;
27
28     void initialize(int capacity2) {
29         count = 0;
30         capacity = capacity2;
31         entries = new Entry[capacity];
32     }
33
34     void assign(const HashMap & map2) {
35         count = map2.count;
36         capacity = map2.capacity;
37         entries = new Entry[capacity];
38         for (int i = 0; i < capacity; ++ i) {
39             entries[i] = map2.entries[i];
40         }
41     }
42
43 public:
44
45     HashMap() {
46         initialize(2);
47     }
48
49     ~HashMap() {
50         delete [] entries;
51     }
52
53     HashMap(const HashMap & map2) {
54         assign(map2);
55     }
56
57     HashMap & operator = (const HashMap & map2) {
58         delete [] entries;
59         assign(map2);
60         return (*this);
61     }
```

HashMap 中的构造函数基本与 HashSet 中的相同。

```
63     void clear() {
64         delete [] entries;
65         initialize(2);
66     }
67
68 private:
69
70     int hashIndex(const K & key) const {
71         int hashCode = hashFunction.getHashCode(key);
72         return hashCode % capacity;
73     }
74
75     int find(const K & key) const {
76         int index = hashIndex(key);
77         while (true) {
78             if (! entries[index].isInUse) {
79                 return index;
80             }
81             if (entries[index].key == key) {
82                 return index;
83             }
84             index = (index + 1) % capacity;
85         }
86     }
87
88     void resize(int capacity2) {
89         Entry * entries0 = entries;
90         int capacity0 = capacity;
91         initialize(capacity2);
92         for (int i = 0; i < capacity0; ++ i) {
93             if (entries0[i].isInUse) {
94                 put(entries0[i].key, entries0[i].value);
95             }
96         }
97         delete [] entries0;
98     }
```

对于函数 `find`, 从 `HashMap` 中的查找 `element` 的位置改为了 `HashSet` 中的对指定 `key` 值的 `key-value pair` 的位置的查找。

函数 `resize` 中从对 `HashSet::add` 的调用改为对 `HashMap::put` 调用把 key-value pairs 加回到新的存储空间内。

```
100 public:
101
102     void put(const K & key, const V & value) {
103         int index = find(key);
104         entries[index].value = value;
105         if (entries[index].isInUse) return;
106
107         entries[index].isInUse = true;
108         entries[index].key = key;
109
110         ++ count;
111         if (count > capacity / 2) {
112             resize(capacity * 2);
113         }
114     }
115
116     V get(const K & key) const {
117         int index = find(key);
118         if (! entries[index].isInUse) {
119             throw NoSuchKeyException();
120         }
121         return entries[index].value;
122     }
123
124     bool remove(const K & key) {
125         int index = find(key);
126         if (! entries[index].isInUse) return false;
127         fillNotInUseEntry(index);
128
129         -- count;
130         if (count < capacity / 4) {
131             resize(capacity / 2);
132         }
133         return true;
134     }
```

`HashMap` 中使用函数 `put` 代替 `HashSet` 中的函数 `add`。函数 `put` 把一个 key-value pair 加入到容器中。函数 `put` 把总是会

成功的，因此无需返回一个值表示添加成功。当容器中存在给定的 **key** 的 key-value pair 的时候，函数 **put** 只是更新 key-value pair 中的 **value** 的值然后就返回（第 104-105 行）。否则，函数 **put** 插入新的 key-value pair（第 107-113 行）。

函数 **get** 是 **HashMap** 中的新函数。如果容器中存在给定的 **key** 的 key-value pair，那么函数 **get** 返回这个 key-value pair 中的 **value** 的值。否则函数 **get** 无法返回一个有效的值，只能抛出 **NoSuchKeyException**。函数删除给定 **key** 的 key-value pair。

```
136 private:
137
138     void fillNotInUseEntry(int index) {
139         int next = index;
140         while (true) {
141             next = (next + 1) % capacity;
142             if (! entries[next].isInUse) {
143                 entries[index].isInUse = false;
144                 return;
145             }
146             int index0 = hashIndex(entries[next].key);
147             if (index < next) {
148                 if (index0 > index &&
149                     index0 <= next) continue;
150             }
151             else {
152                 if (index0 > index ||
153                     index0 <= next) continue;
154             }
155             entries[index] = entries[next];
156             index = next;
157         }
158     }
```

函数 **fillNotInUseEntry** 的作用与 **HashSet** 中的对应函数

完全相同。

```
160 public:
161
162     bool containsKey(const K & key) const {
163         int index = find(key);
164         return (entries[index].isInUse);
165     }
166
167     int size() const {
168         return count;
169     }
170
171     vector<K> getKeys() const {
172         vector<K> vec;
173         for (int i = 0; i < capacity; ++ i) {
174             if (entries[i].isInUse) {
175                 vec.push_back(entries[i].key);
176             }
177         }
178         return vec;
179     }
180
181 };
```

函数 `containsKey`、`size`、`getKeys` 分别对应 `HashSet` 中的 `contains`、`size`、`toVector`。

习题

1. 哈希集合 hash-set 的另一种实现

`Hash-set` 和 `hash-map` 的实现十分相似。为了避免几乎重复的代码，在某些程序设计语言中，如 `Java`，使用 `hash-map` 来简单地实现 `hash-set`。假设模版类接口 `Set`、`HashFunction` 和 `HashMap` 已经编写好并放在 "`set.cpp`"、"`hashmap.cpp`" 和

"hashfunction.cpp"中，请实现类 HashSet。

```
1 #include "set.cpp"
2 #include "hashmap.cpp"
3 #include "hashfunction.cpp"
4
5 template <typename E, typename H = HashFunction<E> >
6 class HashSet : public Set<E>
7 {
8 private:
9
10     HashMap<E, int, H> map;
11
12 public:
13
14     bool add(const E & e);
15
16     bool contains(const E & e) const;
17
18     bool remove(const E & e);
19
20     void clear();
21
22     int size() const;
23
24     vector<E> toVector() const;
25
26 };
```

2. 包含 hash-map 对象的稀疏矩阵

实现所缺的 2 个函数函数。

```
1 #include "hashmap.cpp"
2
3 class Sparse
4 {
5 public:
6
7     class NoSuchDimensionException {};
8     class IndexOutOfBoundsException {};
9 }
```

```

10     class IntHash : public HashFunction<int>
11     {
12     public:
13         int getHashCode(const int & value) const {
14             return value;
15         }
16     };
17
18 private:
19
20     int rows;
21     int columns;
22     HashMap<int, double, IntHash> entries;
23
24     void checkBounds(int row, int column) const {
25         if (row < 1 || row > rows) {
26             throw IndexOutOfBoundsException();
27         }
28         if (column < 1 || column > columns) {
29             throw IndexOutOfBoundsException();
30         }
31     }
32
33 public:
34
35     int size(int dimension) const {
36         switch (dimension) {
37             case 1: return rows;
38             case 2: return columns;
39         }
40         throw NoSuchDimensionException();
41     }
42
43     void set(int row, int column, double value);
44
45     double get(int row, int column) const {
46         checkBounds(row, column);
47         int index = (row - 1) * columns + column - 1;
48         if (! entries.containsKey(index)) {
49             return 0;
50         }
51         else {
52             return entries.get(index);
53         }

```

```
54     }  
55  
56 public:  
57  
58     Sparse(int rows, int columns) {  
59         this->rows = rows;  
60         this->columns = columns;  
61     }  
62  
63     friend ostream & operator << (ostream & out,  
64                                     const Sparse & matrix);  
65  
66 };
```