知识笔记 | C++

- 第一部分：面向对象的程序设计方法
  - 抽象
    - 所有的程序设计语言都是抽象的。**人们能够解决的问题的复杂性直接与抽象的类型和质量有关。**汇编语言是对机器底层的小幅度抽象，C语言是对汇编语言的抽象，而C++，可以说是对C语言的进一步抽象。
    - 汇编语言和C语言，要求程序员按着计算机结构，而不是要解决的问题的机构去思考，因此程序员必须要在机器模型（解空间，即建模该问题的空间，也就是计算机中）和实际上要解决的问题的模型（问题空间，即问题存在的空间中）建立联系。程序员必须在这两者时间建立对应，这就导致了程序设计之外的任务非常重。
    - 面向对象的程序设计，为程序员**提供了在问题空间表示各种事物元素的工具**，这种表示方法是**通用**的，并不限定程序员处理特定类型的问题。**我们把问题空间中的事物和它们在解空间中的表示成为“对象”，其思想是允许程序通过添加新的对象类型，而使程序本身能够根据问题的实际情况进行调整，这样当我们读描述解决方案的代码时，也就是在读表达该问题的问字。**这是比以前更为灵活和强大的语言抽象。因此，面向对象允许程序员使用问题本身的术语来描述问题，而不是要运用解决方案的计算机术语来描述问题。
    - 面向对象的五个特点（主要说的是smalltalk）
      - 1）万物皆对象。
      - 2）程序就是一组对象，对象之间通过发送消息互相通知做什么。
      - 3）每个对象都有它自己的由其它对象构成的存储区。
      - 4）每个对象都有一个类型。
      - 5）一个特定类型的所有对象都能接手相同的消息。
  - 程序分析和设计
    - 面向对象是一种新的（相对于过程语言），不同的编程方式，很多人一开始学习如何处理OOP项目都会感到困哪。这里我们处理OOP项目的方法称之为方法论，这是一系列的过程和探索。当我们设计程序时，**我们应时刻记住我们正在努力寻找什么：**

      只要我们知道了对象和接口，就可以编写程序了。
      - 1）有哪些对象？（如何将项目分成多个组成部分？）
      - 2）他们的接口是什么？（需要向每个对象发送什么信息？）
    - 第0阶段：制定计划
      - 最好首先决定在此过程中应当有哪些步骤。也最好在进程中设立一些里程碑可以帮助集中我们的注意力，激发我们的热情，而不是只注意“完成项目”这个单一目标。另外，还必须**做一个任务陈述，它设定了项目的基调，不必一开始就让它正确，但是要不停地努力直至它越来越正确。**
    - 第1阶段：我们在做什么
      - 这一阶段，我们必须把注意力始终放在核心问题上：确定这个系统要做什么。为此最有价值的工具是一组所谓的“use case”，用来指明系统中的关键特性，它将展现一些我们使用的基本的类。实际上可以通过回答一下问题进行梳理：
        - 1）谁将使用这个系统？

- 2）执行者用这个系统做什么？
- 3）执行者如何用这个系统工作？
- 4）如果其他人也做这件事，或者同一个执行者有不同的目标，该怎么办？（揭示变化）
- 5）当使用这个系统时，会发生什么问题（揭示异常）？

- 第2阶段：我们将如何建立对象
  - 卡片法设计对象的方法

    每个卡片描述一个类，并在上面书写：
    - 1）类的名字。
    - 2）类的职责：它应当做什么。
    - 3）类的协同：它与其他类有哪些交互？
  - 对象设计的五个阶段

    对象设计的声明周期不仅仅限于写程序的时间。对对象做什么和它应该像什么的理解，会随着实践的推移而呈现。特殊类型程序的模式是通过一次又一次的求解问题而形成的。
    - 1）对象发现。
    - 2）对象装配。
    - 3）系统构造。
    - 4）系统扩充。
    - 5）对象重用。
  - 对象开发准则
    - 1）让特定问题生成一个类，然后在解决其他问题期间让这个类生长和成熟。
    - 2）记住，发现所需要的类（和它们的接口），是设计系统的主要内容。
    - 3）不要强迫自己在一开始就知道每一件事情，应当不断地学习。
    - 4）开始编程，让一部分能够运行，这样就能证明或者否定已生成的类。
    - 5）尽量保持简单。具有明显用图的不太清楚的对象要比很复杂的接口好。
- 第3阶段：创建核心
  - 这是从粗线条设计向编译和执行可执行代码体的最初转换阶段，特别是，它将证明或者否定我们的体系结构。这不是一遍的过程，而是反复地建立系统的一系列步骤的开始。
  - 建立这个系统的一部分工作时实际检查，就是对好需求分析和系统规范说明与测试结果。确保我们的测试结果与需求和用例相符。当系统核心稳定之后，就可以向下进行和增加更多而功能了。
- 第4阶段：迭代用例
- 第5阶段：进化

- 小结：
  - 对我来说，第一部分需要掌握两个要点。第一，C++，面向对象的这种程序设计语言是比C语言更高一层的抽象，它将问题的求解空间由解空间迁移到了问题空间。这就要求我们使用C++时，必须从问题的描述去着手思考（VS C语言的解的描述）。第二，掌握比较成熟的面向对象的程序设计与分析套路（步骤）。平时自己写程序哪里有什么严格的步骤啊，就是想到哪里写到哪里，不出问题还好，问题稍微复杂一点就招架不住了。另外，学习这个套路还有一个好处，就是几乎

> 所有对象的类应该几乎都是这个思路设计的，因此在学习别人的程序时，可以用这个方法抓住要点快速切入。

- 第二部分：C,C++通用技术
  - [1]指定存储空间分配

    创建一个变量时，我们拥有指定变量生存周期的很多选择，指定怎样给变量分配存储空间，以及指定编译器怎样处理这些变量。
    - **全局变量**：定义在所有函数体的外部，程序的所有部分可用，不受作用域的影响；在另一个文件中使用时，必须用extern关键字来声明。
    - **局部变量**：定义在一个作用域内，局限于一个函数中。局部变量经常被称为自动变量，因为它们在进入作用域时自动生成，离开作用域时自动消失。局部变量默认为auto，所以没有必要声明为auto。
      - **寄存器变量**，是局部变量的一种，关键字register告诉编译器尽可能快地访问这个变量。一般情况下，最好不用这个关键字。
    - **静态变量**：使用关键字static来修饰变量。函数的局部变量在函数作用域结束时消失，当再次调用时，会重新创建该变量的存储空间，其值会重新被初始化。但如果想使局部变量的值在整个生命期里都存在，则可以**将局部变量定义为static，并给它一个初值。初始化只在函数第一次调用时执行，函数调用之间变量的值保持不变。**

      注意：
      1. 之所以不用全局变量，是因为，**static变量的优点在函数范围之外它是不可用的，所以不能被轻易改变，这会使错误局部化。**
      2. 被static修饰的全局变量，也有自己的作用域，不可以再通过extern 在别处调用。
    - **外部变量：extern。**
      - 连接：在一个执行程序中，标识符代表存放比变量或者被编译过的函数体的存储空间。连接用连接器所见的方式描述存储空间。连接有内部连接和外部连接两种。
      - 内部连接意味着只对正被编译的文件创建存储空间。
    - **常量**：在C语言中，使用预处理可以定义 # define PI 3.14159，将PI定义为常量，现在C++中也可以使用。但该PI就只是常量，没有指针和引用。C++中引入了命名常量的概念，命名常量就像变量一样，只是它的值不能改变。在C++中，const常量必须有初始值。
    - **volatile变量。**
    - 小结：这里我要注意两点，第一，局部变量想要具有某种“记忆”性的话，可以通过static关键字获得；第二，在C++中，常量可以使用const定义，这样定义的常量具有变量的特性，也即可以传地址和引用。
- 第三部分：面向对象的基本特征（进阶技术）
  - **[1] 继承与组合（C++的面向对象的基本特征之一）**

    C++重要特征之一是代码重用，使用但是不修改已存的代码，这是通过继承来实现的。
    - **继承**
      - 与多重继承：在类的左括号前面，加一个冒号和基类的名字（如果是多重继承，要给出多个基类名，他们之间用逗号分开），就会自动地得到基类中所用数据成员和成员函数。
    - **组合**
      - 就是将已有的类的对象放在新类中，当做数据成员。
    - **重写与重定义**

      如果继承一个类，并对它的成员函数进行重定义，则有两种情况

- 第一，重写。在派生类的定义中明确地定义操作和返回类型，称之为重写（redefinig）。
- 第二，重定义。如果基类的成员函数被定义成虚函数，再在派生类中定义该函数的行为称之为重定义（overriding）。
- 新知1：任何时候重新定义了基类中的一个重载函数，在新类之中所有其他的版本都被自动地隐藏了，这里指的是**非虚函数的重写**。非虚函数在派生类中重写，会覆盖其他版本的函数。@Q

- **三个关键字：public，private，protected**
  - **访问权限**：public 公共接口 ： 向外界开放，可通过对象或类名访问；protected 受保护的访问：只向子类开放访问权限 ，不可通过对象访问；private私有 ： 只有类成员可以访问，外界不可以通过任何形式访问。
  - **继承关系**：
    - 1）public继承：父类成员的访问权限在子类中不变，仍为自己原来的权限。
    - 2）protected继承：父类的public成员访问权限在子类中变为protected；父类的protected成员和private成员在子类权限维持原来的权限不变。
    - 3）private继承：父类的所有成员访问权限在子类中变为private。其中，public继承是is-a的关系，可以用父类的指针或者引用指向子类的对象；protected和private继承没有is-a的关系，只表示“组合”或者“拥有”的关系，不可以用父类的指针或引用指向子类对象。C++对象模型中，子类对象的内存空间中包含父类的部分，当用父类指针指向一个子类的对象实时，这个指针可以访问的是相应的父类那部分的内存；在protected和private继承的情况下，父类的内存部分是私有的，不对外开放的，所以，protected和private继承时，不可以用父类的指针或引用指向子类对象。
  - 通过对象访问成员时，只能访问到共有（Public）成员。需要禁止类以构造形式实例化类的时候，可以将类的构造函数声明为private 和protected 的形式。构造函数声明protected 的形式的类不能被直接实例化，但可以通过被继承，子类可以在实例化的时候调用父类的protected构造函数。构造函数声明为private的类的实例化不能依赖构造函数，可以提供public 的Instance（）的方法，在Instance（）定义中调用构造函数，返回类的实例。单例模式就是利用这个原理。@Q

- **[2] 多态性与虚函数（C++的面向对象的基本特征之二）**

  多态性将what（实现）与how（接口）分离开来，为程序创建了可扩展性。多态性是指类的多态，表达的是同一大类中多个相似的派生类之间的区别，并且这种区别可以随着派生无限扩展。【类似于重载表达的是函数的一语多义，多态表达的是同一类相似对象的不同形态】
  - 新知1：**实现多态性的机制**。C++中多态性是通过**虚函数**来实现的。多态性允许许多具有相似性的类被看做是一个类。虚函数允许一个类型表达自己与另一个相似类型之间的区别，只要这两个类型是从同一个基类派生出来的。**多态性通过虚函数机制，保证了可以通过基类的地址根据具体对象（多个相似对象的某个个体）来动态地调用类的具体实现。**

    意思是指，函数的形参可以是指向基类的指针或者引用，而实参可以是其不同的继承类的对象，这样函数调用时就可以根据具体继承类的对象的不同，动态调用相应的类/子类的“特殊”函数。
  - 新知2：**向上类型转换**。用自己的话理解，可以**通过基类的地址（指针或者引用）来操作所有他自己的对象，或者所有由他派生出去的类的对象**。并且，在操作过程中，本着由下到上的就近原则进行虚函数的匹配，较近层的派生找不到相应的函数时，转到上一层寻找，直至找到。

- 新知3：多态性是通过虚函数来实现的，虚函数实现函数多态调用的机制称为向上类型转换。
- 新知4：早捆绑与晚捆绑（动态捆绑）。所谓早捆绑就是指在程序的编译和链接过程中，将函数体与函数调用相联系。所谓动态捆绑，是指在程序运行时才根据构建的具体（实际）对象调用合适的成员函数。
- 新知5：动态绑定的机制----虚函数（Virtual）。一个类的基类中某个对象被声明为虚函数时，编译器会对每个包含虚函数的类创建一个表（Vtable），并制定一个指针（Vptr）来指向这个表。在这个表中，**编译器存放了这个类中或者他的基类中所有已经声明为virtual的函数地址（当这个类中没有重写虚函数时，存放的是上一层基类的虚函数地址）**。通过基类的地址做虚函数调用（也即多态调用）时，编译器会插入能够能够取得这个Vptr并在Vtable表中查找函数地址的代码，这样就能保证晚捆绑的发生。

- **[3] 数据抽象（C++的面向对象的基本特征之三）**

  组合特性和行为来生成新的数据类型。
  - 的

- 第四部分：面向对象的基础技术
  - **[1] 内联函数（C++的非面向对象特征）**
    - **问题。**C++来源于C，C中使用宏来提高效率，因此C++也使用宏，但存在两个问题，第一，宏看起来像个函数调用，但并不总是这样，会带来隐藏的错误；第二，宏是作为预处理进行的，C++中不允许预处理器访问类的成员数据，因此就和类脱离了。
    - **C++中的宏：内联函数。**内联函数和普通函数一样，是真正的函数，可以实现所有函数能够实现的功能，唯一的不同之处在于，**内联函数在适当的地方像宏一样展开，所以不需要函数调用的开销。**
    - 内联函数的来源。第一，在类中定义的函数，自动成为内联函数。第二，在非类函数前面加上inline使之成为内联函数，但是应该把内联函数的定义放在头文件中。
    - 为什么要用内联函数。主要是为了减少函数调用的开销。但是如果函数较大，会导致代码膨胀，在速度方面获得的好处就会减小，因此任何效率的提高都是相对的。
  - **[2] 函数重载与参数默认（C++的非面向对象的特征）**

    程序中的一词多义。**一词，指的是名字相同，多义，通过不同的参数列表识别。**用统一的对外接口来完成不同的功能。
    - **新知1**：**程序中名字的内涵。**我们创建一个对象（一个变量），其实质是在划分一块存储区出来，并为其取一个名字。函数名也是一块内存的名字。通过编制各种名字来描述身边的系统，就可以产生易于人们理解和修改的程序。因此，**能使名字方便使用，是任何程序设计语言的重要特征。**
    - **新知2：重载的来源。**如何把人类自然语言中有细微差别的概念映射到程序设计语言中呢？通常，自然语言中同一个词语可以有多重不同的含义，具体含义依赖上下文来确定。这就是所谓的**一词多义**，也即**重载（overload）。**
    - **新知3**：**程序中的一词多义。**一词，指的是名字相同，多义，通过不同的参数列表识别。编译器会修饰这些名字、范围和参数来产生内部名以供它和连接器使用。

      绝不可以使用返回值的差别来进行函数的重载，因为函数的返回值不是必须的，有时候可以省略，那么这时候就会有歧义产生。
    - **新知4**：**参数默认。**参数默认是在函数声明时就指定一个值，如果在调用函数时没有指定这个值，则编译器自动调用默认值。编译器能够看到默认参数值。

      Stash（int size，int initQuantiy = 0）；//声明了一个默认参数
      Stash A（100），B（100,0）//两者的初始化方式相同。

- 两条规则：第一，只有参数列表的后部参数才是可默认的，第二，一旦在一个函数调用中开始使用默认参数，那么这个参数后面的所有参数都必须是默认的。
- undefined

- **[3] 静态成员函数**
  - 为什么要用静态成员函数？有时候需要为某个类的所有对象分配一个单独的存储空间，也就是让这个类的所有对象共享这块存储空间。类似于全局函数。
  - undefined
- **[4] 构造函数的初始化表达式表**
  - 构造函数的初始化表达式表，是用来**初始化新类中的数据成员、对象成员，以及改变基类初始化默认参数的。**默认情况下，编译器在继承类中会默认调用基类的构造函数进行基类对象的初始化，这在类体的左括号之前就已经完成了，但当我们需要改变默认构造函数时，就会用到这个表，有三种类型，语法一致：
  - 第一类，在继承类中向基类的构造函数传递非默认参数值。

    ```
    MyType::MyType(int i)：Bar(i) {//....
    ```

    - 在这个例子中，MyType继承于Bar，在建立对象MyType时要向基类传递一个参数i。
  - 第二类，在继承类中初始化对象成员，向其传递非默认参数值。

    ```
    MyType2::MyType2 (int i)：Bar(i)，m(i+1) {//....
    ```

    - 在这例子中，MyType2包含一个称为m的对象成员，需要向其传递一个新的初始化参数(i+1)。
  - 第三类，在类的构造函数中，初始化它自己的数据成员（又叫做，类的内建类型）。

    ```
    class X {
    int i;
    float f;
    char c;
    char *s;
    public:
     X( ) : i (7), f (1.4), c ('x'), s ("howdy") { //构造函数体}
    ```

    - 在这个例子中，构造函数的初始化表达式表，对它自己的数据成员进行了初始化操作。

- 第五部分：面向对象的高阶技术
  - **[1] C++模板**
    - **【9.1】概述：**
      - 模板是C++支持**参数化多态**的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数，返回值取得任意类型。模板是一种**对类型进行参数化**的工具。通常有两种形式：函数模板和类模板。函数模板针对仅仅**参数类型不同**的函数；类模板仅仅针对**数据成员和成员函数类型不同**的类。
      - **使用模板的目的就是能够让程序员编写与类型无关的代码。**比如编写了一个交换两个整型int的函数swap函数，这个函数就只能实现int 型，对double，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个swap函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个swap模板函数，即可以实现int 型，又可以实现double型的交换。
      - 【注意】模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。

- **【2】函数模板**
  - 函数模板的格式：
    > template<class 形参名，class 形参名，....>返回类型 函数名（参数列表）{函数体}
  - 其中，template和class是关键字，class可以用typename关键字替代，这里typename和class没有区别，<>括号中的参数叫做**模板形参**，模板形参和函数形参很相像，**模板形参不能为空。一旦声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，即可以在该函数中使用内置类型的地方都可以使用模板形参名。**模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际模板参数类型就称他实例化了函数模板的一个实例。比如swap的模板函数形式为：
    > template<class T> voide swap (T& a, T&b) { 函数体}
  - 当调用这样的模板函数时，类型T就会被调用时的类型所替代，比如swap(a, b)，其中a和b是int型，这时模板函数swap中的形参T就会被int所替代，模板函数就变为了swap(int &a, int &b)。而当swap(c,d)其中c和d是double类型时，模板函数会被替换为swap(double &a, double &b)，这样就实现了函数的实现与类型无关的代码。
  - 【注意】对于函数模板而言不存在h(int, int)这样的调用，不能在函数调用的参数中指定模板形参的类型，对于函数模板的调用应该使用实参推演来进行，即智能进行h（2,3）这样的调用，或者int a，b；h（a，b）
- **【3】类模板**
  - **类模板的格式**：
    > template<class 形参名，class 形参名，...> class 类名{ 类体};
  - 类模板和函数模板都是以template开始后接模板形参列表组成，模板形参不能为空，一旦声明了类模板就可以使用类模板的形参名声明类中的成员变量和成员函数，即可以在类中使用内置类型的地方都可以使用模板形参名来声明。比如：
    ```
    template<class T> class A
    { public: T a; T b;
    T hy(T c，T &d）；
    };
    ```
  - 在类A中，声明了两个类型为T的成员变量a和b，还声明了一个返回类型为T，带两个参数类型为T的函数hy。
  - **类模板对象的创建：**
  - 比如一个模板类A，则使用类模板创建对象的方法为A<int> m; 在类A后面跟上一个<>尖括号并在括号里面填上相应的类型，这样的话，类A中凡是用到模板形参的地方都会被int所替代。当类模板有两个模板形参时，创建对象的方法为A<int, double> m；类型间用逗号隔开。
  - 【注意1】对于类模板，模板形参的类型必须在类名后的尖括号中明确指定。例如A<2> m;这种方法把模板形参设置为int的做法是错误的，**类模板形参不存在实参推演的问题。**也就是说不能把整型值2推演为int型传递给模板形参。要把类模板形参调置为int型必须这样指定A<int>m。
  - **【注意2】在类模板外部定义成员函数的方法为：**
    > template<模板形参列表> 函数返回类型 类名<模板形参名>:: 函数名（参数列表）{函数体},
    > 比如有两个模板形参T1，T2的类A中含有一个void h（）函数，则定义该函数的语法为：
    > template<class T1, class T2> void A<T1, T2>::h() {};
  - 【注意3】当在类外面定义类的成员时template后面的模板形参应与要定义的类的模板形参一致。

- 【注意4】模板的声明或者定义只能在全局，命名空间或类范围内进行。既不能在局部范围内进行，函数内进行，比如，不能在main函数中声明或定义一个模板。

- **【4】模板的形参**

  有三种类型的模板形参：类型形参，非类型形参，模板形参。

  - **类型形参**
    - 类型模板形参：**类型形参由关键字class或者typename后接说明构成**，如template<class T> void h(T a) {};其中T就是一个类型形参，类型形参的名字由用户自己确定。**模板形参标识的是一个未知的类型。**模板类型形参可作为类型说明符用在模板中的任何地方，与内置类型说明符或者类类型说明符的使用方式相同，既可以用于指定返回类型，变量声明等。
  - **非类型形参**
    - 非类型模板形参：**模板的非类型形参也就是内置类型形参**，如template<class T, int a>class B {};其中int a就是非类型的模板形参。
    - 非类型形参在模板定义的内部是常量值，也就是说非类型形参在模板的内部是常量。
    - **非类型模板的形参只能是整型，指针和引用**，像double，String, String **这样的类型是不允许的。但是double &，double *，对象的引用或指针是正确的。
    - **调用非类型模板形参的实参必须是一个常量表达式**，即他必须能在编译时计算出结果。
    - 【注意】：**任何局部对象，局部变量，局部对象的地址，局部变量的地址都不是一个常量表达式，都不能用作非类型模板形参的实参。**全局指针类型，全局变量，全局对象也不是一个常量表达式，不能用作非类型模板形参的实参。
    - **全局变量的地址或引用，全局对象的地址或引用const类型变量是常量表达式，可以用作非类型模板形参的实参。**
    - sizeof表达式的结果是一个常量表达式，也能用作非类型模板形参的实参。
    - 当模板的形参是整型时调用该模板时的实参必须是整型的，且在编译期间是常量，比如template <class T, int a> class A{};如果有int b，这时A<int, b> m;将出错，因为b不是常量，如果const int b，这时A<int, b> m;就是正确的，因为这时b是常量。
    - **非类型形参一般不应用于函数模板中**，比如有函数模板template<class T, int a> void h(T b){}，若使用h(2)调用会出现无法为非类型形参a推演出参数的错误，对这种模板函数可以用显示模板实参来解决，如用h<int, 3>(2)这样就把非类型形参a设置为整数3。显示模板实参在后面介绍。
    - **非类型模板形参的形参和实参间所允许的转换**
      - 1、允许从数组到指针，从函数到指针的转换。如：template <int *a> class A{}; int b[1]; A<b> m;即数组到指针的转换
      - 2、const修饰符的转换。如：template<const int *a> class A{}; int b; A<&b> m; 即从int *到const int *的转换。
      - 3、提升转换。如：template<int a> class A{}; const short b=2; A<b> m; 即从short到int 的提升转换
      - 4、整值转换。如：template<unsigned int a> class A{}; A<3> m; 即从int 到unsigned int的转换。

        - 5、常规转换。
    - **【5】类模板的默认模板类型参数**
        - 可以为类模板的类型形参提供默认值，但不能为函数模板的类型形参提供默认值。函数模板和类模板都可以为模板的非类型形参提供默认值。
        - 类模板的类型形参默认值形式为：template<class T1, class T2=int> class A{};为第二个模板类型形参T2提供int型的默认值。
        - 类模板类型形参默认值和函数的默认参数一样，如果有多个类型形参则从第一个形参设定了默认值之后的所有模板形参都要设定默认值，比如template<class T1=int, class T2>class A{};就是错误的，因为T1给出了默认值，而T2没有设定。
        - 在类模板的外部定义类中的成员时template 后的形参表应省略默认的形参类型。比如template<class T1, class T2=int> class A{public: void h();}; 定义方法为template<class T1,class T2> void A<T1,T2>::h(){}。
- **[2]** C++**容器**
    - **【8.1】什么是容器?**

      什么是容器？实质上容器就是一组相同类型对象的集合，但它又不仅仅像数组那样简单，它实现了比数组更复杂的数据结构，同时也具有了比数组更强大的功能。C++标准模板库里提供了10种通用的容器类，它基本上可以解决程序中遇到的大多数问题。**容器进一步解析。**容器的概念是基于数据结构的基本知识，实际上这些容器就是对数据结构提炼的产物，或者说每一个容器都是对某一种数据结构的实例化。容器是由数据结构而来，如果不了解数据结构就难以理解容器的本质。
        - 在C++中容器被定义为：**在数据存储上，有一种对象类型，它可以持有其他对象或者指向其他对象的指针，这类对象类型就叫做容器。**很简单，容器就是保存其它对象的对象，这种“对象”还包含了一系列处理“其他对象”的方法，因为这些方法在程序设计上经常会被用到，所以容器也体现了一个好处，就是“**容器类是一种对特定代码重用问题的良好的解决方案。**”容器还有另一个特点是**容器可以自行扩展。**在解决问题时我们常常不知道我们需要存储多少个对象，也就是说我们不知道应该创建多大的内存空间来保存我们的对象。显然，数组在这一方面也力不从心。容器的优势就在这里，它不需要你预先告诉它你要存储多少对象，只要你创建一个容器对象，并合理的调用它所提供的方法，所有的处理细节将由容器来自身完成。它可以为你申请内存或释放内存，并且用最优的算法来执行您的命令。容器是随着面向对象语言的诞生而提出的，容器类在面向对象语言中特别重要，甚至它被认为是早期面向对象语言的基础。在现在**几乎所有的面向对象的语言中也都伴随着一个容器集，在C++ 中，就是标准模板库（STL ）。**和其它语言不一样，C++ 中处理容器是采用基于模板的方式。标准C++ 库中的容器提供了多种数据结构，这些数据结构可以与标准算法一起很好的工作，这为我们的软件开发提供了良好的支持!
    - **【8.2】通用容器的分类**

      STL 对定义的通用容器分三类：顺序性容器、关联式容器和容器适配器。
        - **顺序性容器** 是一种各元素之间有顺序关系的线性表，是一种线性结构的可序群集。顺序性容器中的每个元素均有固定的位置，除非用删除或插入的操作改变这个位置。这个位置和元素本身无关，而和操作的时间和地点有关，顺序性容器不会根据元素的特点排序而是直接保存了元素操作时的逻辑顺序。比如我们一次性对一个顺序性容器追加三个元素，这三个元素在容器中的相对位置和追加时的逻辑次序是一致的。
        - **关联式容器** 和顺序性容器不一样，关联式容器是非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。但是关联式容器提供了另

一种根据元素特点排序的功能，这样迭代器就能根据元素的特点“顺序地”获取元素。关联式容器另一个显著的特点是它是**以键值的方式来保存数据，就是说它能把关键字和值关联起来保存**，而顺序性容器只能保存一种（可以认为它只保存关键字，也可以认为它只保存值）。这在下面具体的容器类中可以说明这一点。

- **容器适配器** 是一个比较抽象的概念， C++ 的解释是：适配器是使一事物的行为类似于另一事物的行为的一种机制。容器适配器是让一种已存在的容器类型采用另一种不同的抽象类型的工作方式来实现的一种机制。其实仅是发生了接口转换。那么你可以把它理解为容器的容器，它实质还是一个容器，只是他不依赖于具体的标准容器类型，可以理解是容器的模版。或者把它理解为容器的接口，而适配器具体采用哪种容器类型去实现，在定义适配器的时候可以由你决定。

- **【8.3】STL中的具体容器**

  STL中定义的三类容器所包含的具体容器类。

  - **顺序容器**

    线性数据结构

    - **向量 vector**

      从后面快速的插入与删除，直接访问任何元素；

      - 是一个线性顺序结构。相当于数组，但其大小可以不预先指定，并且自动扩展。它可以像数组一样被操作，由于它的特性我们完全可以将vector 看作**动态数组**。
      - 在创建一个vector 后，它会自动在内存中分配一块连续的内存空间进行数据存储，初始的空间大小可以预先指定也可以由vector 默认指定，这个大小即capacity （）函数的返回值。当存储的数据超过分配的空间时vector 会重新分配一块内存块，但这样的分配是很耗时的，在重新分配空间时它会做这样的动作：首先，vector 会申请一块更大的内存块；然后，将原来的数据拷贝到新的内存块中；其次，销毁掉原内存块中的对象（调用对象的析构函数）；最后，将原来的内存空间释放掉。如果vector 保存的数据量很大时，这样的操作一定会导致糟糕的性能（这也是vector 被设计成比较容易拷贝的值类型的原因）。所以说vector 不是在什么情况下性能都好，只有在预先知道它大小的情况下vector 的性能才是最优的。
      - vector 的特点：(1) 指定一块如同数组一样的连续存储，但空间可以动态扩展。即它可以像数组一样操作，并且可以进行动态操作。通常体现在push_back() pop_back() 。(2) 随机访问方便，它像数组一样被访问，即支持[ ] 操作符 和 vector. at() (3) 节省空间，因为它是连续存储，在存储数据的区域都是没有被浪费的，但是要明确一点vector 大多情况下并不是满存的，在未存储的区域实际是浪费的。(4) 在内部进行插入、删除操作效率非常低，这样的操作基本上是被禁止的。Vector 被设计成只能在后端进行追加和删除操作，其原因是vector 内部的实现是按照顺序表的原理。(5) 只能在 vector 的最后进行push 和pop ， 不能在vector 的头进行push 和 pop 。(6) 当动态添加的数据超过vector 默认分配的大小时要进行内存的重新分配、拷贝与释放，这个操作非常消耗性能。 所以要 vector 达到最优的性能，最好在创建vector 时就指定其空间大小。

    - **双端队列deque**

      从前面或后面快速的插入与删除，直接访问任何元素；

      - 是一种优化了的、对序列两端元素进行添加和删除操作的基本序列容器。它允许较为快速地随机访问，但它不像vector 把所有的对象保存在一块连续的内存块，而是采用多个连续的存储块，并且在一

个映射结构中保存对这些块及其顺序的跟踪。向deque 两端添加或删除元素的开销很小。它不需要重新分配空间，所以向末端增加元素比vector更有效。实际上，deque 是对vector 和list 优缺点的结合，它是处于两者之间的一种容器。

- deque 的特点：(1) 随机访问方便，即支持[ ] 操作符和vector.at() ，但性能没有vector 好；(2) 可以在内部进行插入和删除操作，但性能不及list ； (3) 可以在两端进行push 、pop ；

- **双向链表list**

  ，双链表，从任何地方快速插入与删除；

  - 是一个线性链表结构，它的数据由若干个节点构成，每一个节点都包括一个信息块（即实际存储的数据）、一个前驱指针和一个后驱指针。它无需分配指定的内存大小且可以任意伸缩，这是因为它存储在非连续的内存空间中，并且由指针将有序的元素链接起来。
  - 由于其结构的原因，list 随机检索的性能非常的不好，因为它不像vector 那样直接找到元素的地址，而是要从头一个一个的顺序查找，这样目标元素越靠后，它的检索时间就越长。检索时间与目标元素的位置成正比。
  - 虽然随机检索的速度不够快，但是它可以迅速地在任何节点进行插入和删除操作。因为list 的每个节点保存着它在链表中的位置，插入或删除一个元素仅对最多三个元素有所影响，不像vector 会对操作点之后的所有元素的存储地址都有所影响，这一点是vector 不可比拟的。
  - list 的特点：(1) 不使用连续的内存空间这样可以随意地进行动态操作；(2) 可以在内部任何位置快速地插入或删除，当然也可以在两端进行push 和pop 。(3) 不能进行内部的随机访问，即不支持[ ] 操作符和vector.at() ； (4) 相对于verctor 占用更多的内存。

- **三者之间的区别**

  - vector 是一段连续的内存块，而deque 是多个连续的内存块， list 是所有数据元素分开保存，可以是任何两个元素没有连续。
  - vector 的查询性能最好，并且在末端增加数据也很好，除非它重新申请内存段；适合高效地随机存储。
  - list 是一个链表，任何一个元素都可以是不连续的，但它都有两个指向上一元素和下一元素的指针。所以它对插入、删除元素性能是最好的，而查询性能非常差；适合 大量地插入和删除操作而不关心随机存取的需求。
  - deque 是介于两者之间，它兼顾了数组和链表的优点，它是分块的链表和多个数组的联合。所以它有被list 好的查询性能，有被vector 好的插入、删除性能。 如果你需要随即存取又关心两端数据的插入和删除，那么deque 是最佳之选。

- **关联容器**

  是一种非线性的树结构，具体的说采用的是一种比较高效的特殊的平衡检索二叉树。因为关联容器的这四种容器类都使用同一原理，所以他们核心的算法是一致的，但是它们在应用上又有一些差别，先描述一下它们之间的差别。

  - **set**

    快速查找，不允许重复值；

    - set ，又称集合，实际上就是一组元素的集合，但其中所包含的元素的值是唯一的，且是按一定顺序排列的，集合中的每个元素被称

作集合中的实例。因为其内部是通过链表的方式来组织，所以在插入的时候比vector 快，但在查找和末尾添加上被vector 慢。

- **multiset**

  ，快速查找，允许重复值；

  - multiset ，是多重集合，其实现方式和set 是相似的，只是它不要求集合中的元素是唯一的，也就是说集合中的同一个元素可以出现多次。

- **map**

  对多映射，基于关键字快速查找，不允许重复值；

  - map ，提供一种“键- 值”关系的一对一的数据存储能力。
    其“键”在容器中不可重复，且按一定顺序排列（其实我们可以将set 也看成是一种键- 值关系的存储，只是它只有键没有值。它是map 的一种特殊形式）。由于其是按链表的方式存储，它也继承了链表的优缺点。

- **multimap**

  一对多映射，基于关键字快速查找，允许重复值；

  - multimap ， 和map 的原理基本相似，它允许“键”在容器中可以不唯一。

- **关联容器的特点**

  - 关联容器的特点是明显的，相对于顺序容器，有以下几个主要特点：1， 其内部实现是采用非线性的二叉树结构，具体的说是红黑树的结构原理实现的；2， set 和map 保证了元素的唯一性，mulset 和mulmap 扩展了这一属性，可以允许元素不唯一；
    3， 元素是有序的集合，默认在插入的时候按升序排列。
  - 基于以上特点，1， 关联容器对元素的插入和删除操作比vector 要快，因为vector 是顺序存储，而关联容器是链式存储；比list 要慢，是因为即使它们同是链式结构，但list 是线性的，而关联容器是二叉树结构，其改变一个元素涉及到其它元素的变动比list 要多，并且它是排序的，每次插入和删除都需要对元素重新排序；
    2， 关联容器对元素的检索操作比vector 慢，但是比list 要快很多。vector 是顺序的连续存储，当然是比不上的，但相对链式的list 要快很多是因为list 是逐个搜索，它搜索的时间是跟容器的大小成正比，而关联容器 查找的复杂度基本是Log(N) ，比如如果有1000 个记录，最多查找10 次，1,000,000 个记录，最多查找20 次。容器越大，关联容器相对list 的优越性就越能体现；3， 在使用上set 区别于vector,deque,list 的最大特点就是set 是内部排序的，这在查询上虽然逊色于vector ，但是却大大的强于list 。
    4， 在使用上map 的功能是不可取代的，它保存了“键- 值”关系的数据，而这种键值关系采用了类数组的方式。数组是用数字类型的下标来索引元素的位置，而map 是用字符型关键字来索引元素的位置。在使用上map 也提供了一种类数组操作的方式，即它可以通过下标来检索数据，这是其他容器做不到的，当然也包括set 。
    （STL 中只有vector 和map 可以通过类数组的方式操作元素，即如同ele[1] 方式）

- **容器适配器**

  STL 中包含三种适配器：栈stack 、队列queue 和优先级priority_queue 。

  - **stack，后进先出；**
  - **queue，先进先出；**

- **priority_queue，最高优先级元素总是第一个出列；**
- **容器适配器的特点**
  - 适配器是容器的接口，它本身不能直接保存元素，它保存元素的机制是调用另一种顺序容器去实现，即可以把适配器看作“它保存一个容器，这个容器再保存所有元素”。
  - STL 中提供的三种适配器可以由某一种顺序容器去实现。默认下stack 和queue 基于deque 容器实现，priority_queue 则基于vector 容器实现。当然在创建一个适配器时也可以指定具体的实现容器，创建适配器时在第二个参数上指定具体的顺序容器可以覆盖适配器的默认实现。
  - 由于适配器的特点，一个适配器不是可以由任一个顺序容器都可以实现的。
  - 栈stack 的特点是后进先出，所以它关联的基本容器可以是任意一种顺序容器，因为这些容器类型结构都可以提供栈的操作有求，它们都提供了push_back 、pop_back 和back 操作；
  - 队列queue 的特点是先进先出，适配器要求其关联的基础容器必须提供pop_front 操作，因此其不能建立在vector容器上；
  - 优先级队列priority_queue 适配器要求提供随机访问功能，因此不能建立在list 容器上。
- **比较**

[3] C++迭代器

C++ STL有六大组件分别为：容器，迭代器，算法，仿函数，迭代适配器，空间适配器。

- **【1】概述（http://www.cnblogs.com/lhuan/p/5706654.html）**
  - STL--迭代器（iterator），C++中的标准算法库之一。所有容器有含有其各自的迭代器型别（iterator types），所以当你使用一般的容器迭代器时，并不需要含入专门的头文件。不过有几种特别的迭代器，例如逆向迭代器，被定义于<iterator>中。
  - 代器共分为五种，分别为: **Input iterator**、**Output iterator**、**Forward iterator**、**Bidirectional iterator**、**Random access iterator**。
  - 迭代器iterator 提供了一种**一般化的方法对顺序或关联容器类型中的每个元素进行连续访问。**例如，假设iter为任意容器类型的一个iterator，则++iter表示向前移动迭代器使其指向容器的下一个元素，而*iter 返回iterator 指向元素的值，每种容器类型都提供一个begin()和一个end()成员函数。begin()返回一个iterator 它指向容器的第一个元素。end()返回一个iterator 它指向容器的末元素的下一个位置
  - 迭代器支持的操作
    -
  - 迭代器之间的关系
    -
  - Iterator（迭代器）模式又称Cursor（游标）模式，用于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。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。迭代器的作用：能够让迭代器与算法不干扰的相互发展，最后又能无间隙的粘合起来，重载了*，++，==，！=，=运算符。用以操作复杂的数据结构，容器提供迭代器，算法使用迭代器；

- **【2】Input（输入）迭代器**
  - 只能一次一个向前读取元素，按此顺序一个个传回元素值。下表列出了Input迭代器的各种操作行为。Input迭代器只能读取元素一次，如果你复制Input迭代器，并使原Input迭代器与新产生的副本都向前读取，可能会遍历到不同的值。纯粹Input迭代器的一个典型例子就是“从标准输入装置（通常为键盘）读取数据”的迭代器。

    表达式 功能表述
    *iter 读取实际元素
    iter->member 读取实际元素的成员（如果有的话）
    ++iter 向前步进（传回新位置）
    iter++ 向前步进（传回旧位置）
    iter1 == iter2 判断两个迭代器是否相同
    iter1 != iter2 判断两个迭代器是否不相等
    TYPE(iter) 复制迭代器（copy 构造函数）
- **【3】Output（输出）迭代器**
  - Output迭代器和Input迭代器相反，其作用是将元素值一个个写入。下表列出Output迭代器的有效操作。operator*只有在赋值语句的左手边才有效。Output迭代器无需比较（comparison）操作。你无法检验Output迭代器是否有效，或“写入动作”是否成功。你唯一可以做的就是写入、写入、再写入。

    表达式 功能表述
    *iter = value 将元素写入到迭代器所指位置
    ++iter 向前步进（传回新位置）
    iter++ 向前步进（传回旧位置）
    TYPE(iter) 复制迭代器（copy 构造函数）
- **【4】Forward（前向）迭代器**
  - Forward迭代器是Input迭代器与Output迭代器的结合，具有Input迭代器的全部功能和Output迭代器的大部分功能。下表总结了Forward迭代器的所有操作。Forward迭代器能多次指向同一群集中的同一元素，并能多次处理同一元素

    表达式 功能表述
    *iter 存取实际元素
    iter->member 存取实际元素的成员
    ++iter 向前步进（传回新位置）
    iter++ 向前步进（传回旧位置）
    iter1 == iter2 判断两个迭代器是否相同
    iter1 != iter2 判断两个迭代器是否不相等
    TYPE() 产生迭代器（default构造函数）
    TYPE(iter) 复制迭代器（copy构造函数）
    iter1 == iter2 复制
- **【5】Bidirectional（双向）迭代器**
  - Bidirectional（双向）迭代器在Forward迭代器的基础上增加了回头遍历的能力。换言之，它支持递减操作符，用以一步一步的后退操作。
- **【6】Random Access（随机存取）迭代器**
  - Random Access迭代器在Bidirectional迭代器的基础上再增加随机存取能力。因此它必须提供“迭代器算数运算”（和一般指针“指针算术运算”相当）。也就是说，它能加减某个偏移量、能处理距离（differences）问题，并运用诸如<和>的相互关系操作符进行比较。以下对象和型别支持Random Access迭代器：

    可随机存取的容器（vector, deque）
    strings（字符串，string，wstring）
    一般array（指针）

- **【7】迭代器相关辅助函数**
  - advance() 令迭代器前进
  - distance() 处理迭代器之间的距离
  - iter_swap() 交换两个迭代器所指内容
- **【8】迭代器配接器**
  - **Reverse（逆向迭代器）**
    - 逆向迭代器重新定义递增运算和递减运算，使其行为正好倒置。成员函数rbegin()和rend()各传回一个Reverse迭代器，和begin()和end()类似，共同定义一个半开区间。用正向迭代器可以直接构造一个逆向迭代器，但是构造之后会出现“错位”现象。原因在逆向迭代器要保证半开区间不会越界，可调用逆向迭代器的base()函数，保证转换值的正确性（迭代器移了一位）。
  - **Insert（安插型）迭代器**
    - Insert迭代器，也称为inserters，用来将“赋值新值”操作转换为“安插新值”操作。通过这种迭代器，算法可以执行安插（insert）行为而非覆盖（overwrite）行为。所有Insert迭代器都隶属于Output迭代器类型。所以它只提供赋值（assign）新值的能力。下表列出Insert迭代器的所有操作函数

      表达式 功能表述
      *iter 无实际操作（传回iter）
      iter = value 安插value
      ++iter 无实际操作（传回iter）
      iter++ 无实际操作（传回iter）
    - C++标准程序库提供三种Insert迭代器：back inserters, front inserters, general inserters。它们的区别在于插入位置。事实上它们各自调用所属容器中不同的成员函数。所以Insert迭代器初始化时要清楚知道自己所属的容器是哪一种。下表列出Insert迭代器的种类。

      名称 Class 其所调用的函数 生成函数
      Back inserter back_inserter_iterator push_back(value) back_inserter(cont)
      Front inserter front_insert_iterator push_front(value) front_inserter(cont)
      General inserter insert_iterator insert(pos, value) inserter(cont, pos)
  - **Stream（流）迭代器**
    - Stream迭代器是一种迭代器配接器，通过它，你可以把stream当成算法的原点和终点。更明确的说，一个istream迭代器可以用来从input stream中读元素，而一个ostream迭代器可以用来对output stream写入元素。
    - Stream迭代器的一种特殊形式是所谓的stream缓冲区迭代器，用来对stream缓冲区进行直接读取和写入操作。
    - **Ostream迭代器**
      - ostream迭代器 可以被赋予的值写入output stream中。下表列出ostream迭代器的各项操作

        算式 功能表述
        ostream_iterator<T>(ostream) 为ostream产生一个ostream迭代器
        ostream_iterator<T>(ostream, delim) 为ostream产生一个ostream迭代器，各元素间以delim为分隔符（请注意，delim的型别是const char*）

*iter	无实际操作（传回iter）
iter = value	将value写到ostream，像这样：ostream<<value。其后再输出一个delim（分隔符；如有定义的话）
++iter	无实际操作（传回iter）
iter++	无实际操作（传回iter）

- **Istream迭代器**
  - istream迭代器是ostream迭代器的拍档，用来从input stream读取元素。透过istream迭代器，算法可以从stream中直接读取数据。istream迭代器的各项操作。

算式	功能表述
istream_iterator<T>()	产生一个end-of-stream迭代器
istream_iterator<T>(istream)	为istream产生的一个迭代器（可能立即去读第一个元素）
*iter	传回先前读取的值（如果构造函数并未立刻读取第一个元素值，则本式执行读取任务）
iter->member	传回先前读取的元素的成员（如果有的话）
++iter	读取下一个元素，并传回其位置
iter++	读取下一个元素，并传回迭代器指向前一个元素
iter1 == iter2	检查iter1和iter2是否相等
iter1 != iter2	检查iter1和iter2是否不相等

- **【9】实例**
  - 实例

```
1、vector
#include <iostream>
#include <vector>
int main()
{
   std::vector<char> charVector;
   int x;
   for (x=0; x<10; ++x)
      charVector.push_back(65 + x);
   int size = charVector.size();
   for (x=0; x<size; ++x)
   {
      std::vector<char>::iterator start =
         charVector.begin();
      charVector.erase(start);
      std::vector<char>::iterator iter;
      for (iter = charVector.begin();
            iter != charVector.end(); iter++)
      {
         std::cout << *iter;
      }
      std::cout << std::endl;
   }
   return 0;
}
2、deque
#include <iostream>
#include <deque>
int main()
{
   std::deque<char> charDeque;
```

```
    int x;
    for (x=0; x<10; ++x)
        charDeque.push_front(65 + x);
    int size = charDeque.size();
    for (x=0; x<size; ++x)
    {
        std::deque<char>::iterator start =
            charDeque.begin();
        charDeque.erase(start);
        std::deque<char>::iterator iter;
        for (iter = charDeque.begin();
                iter != charDeque.end(); iter++)
        {
            std::cout << *iter;
        }
        std::cout << std::endl;
    }
    return 0;
}
```

3、list

```
#include <iostream>
#include <list>
int main()
{
    // Create and populate the list.
    int x;
    std::list<char> charList;
    for (x=0; x<10; ++x)
        charList.push_front(65 + x);
    // Display contents of list.
    std::cout << "Original list: ";
    std::list<char>::iterator iter;
    for (iter = charList.begin();
            iter != charList.end(); iter++)
    {
        std::cout << *iter;
        //char ch = *iter;
        //std::cout << ch;
    }
    std::cout << std::endl;

    // Insert five Xs into the list.
    std::list<char>::iterator start = charList.begin();
    charList.insert(++start, 5, 'X');
    // Display the result.
    std::cout << "Resultant list: ";
    for (iter = charList.begin();
    iter != charList.end(); iter++)
    {
        std::cout << *iter;
        //char ch = *iter;
        //std::cout << ch;
    }

    return 0;
}
```

4、set

```
#include <iostream>
```

```
#include <set>
int main()
{
   // Create the set object.
   std::set<char> charSet;
   // Populate the set with values.
   charSet.insert('E');
   charSet.insert('D');
   charSet.insert('C');
   charSet.insert('B');
   charSet.insert('A');
   // Display the contents of the set.
   std::cout << "Contents of set: " << std::endl;
   std::set<char>::iterator iter;
   for (iter = charSet.begin(); iter != charSet.end(); iter++)
      std::cout << *iter << std::endl;
   std::cout << std::endl;
   // Find the D.
   iter = charSet.find('D');
   if (iter == charSet.end())
      std::cout << "Element not found.";
   else
      std::cout << "Element found: " << *iter;
   return 0;
}
```

5、multiset

```
#include <iostream>
#include <set>
int main()
{
   // Create the first set object.
   std::multiset<char> charMultiset1;
   // Populate the multiset with values.
   charMultiset1.insert('E');
   charMultiset1.insert('D');
   charMultiset1.insert('C');
   charMultiset1.insert('B');
   charMultiset1.insert('A');
   charMultiset1.insert('B');
   charMultiset1.insert('D');
   // Display the contents of the first multiset.
   std::cout << "Contents of first multiset: " << std::endl;
   std::multiset<char>::iterator iter;
   for (iter = charMultiset1.begin();
         iter != charMultiset1.end(); iter++)
      std::cout << *iter << std::endl;
   std::cout << std::endl;
   // Create the second multiset object.
   std::multiset<char> charMultiset2;
   // Populate the multiset with values.
   charMultiset2.insert('J');
   charMultiset2.insert('I');
   charMultiset2.insert('H');
   charMultiset2.insert('G');
   charMultiset2.insert('F');
   charMultiset2.insert('G');
   charMultiset2.insert('I');
```

```
    // Display the contents of the second multiset.
    std::cout << "Contents of second multiset: "
        << std::endl;
    for (iter = charMultiset2.begin();
    iter != charMultiset2.end(); iter++)
        std::cout << *iter << std::endl;
    std::cout << std::endl;

    // Compare the sets.
    if (charMultiset1 == charMultiset2)
        std::cout << "set1 == set2";
    else if (charMultiset1 < charMultiset2)
        std::cout << "set1 < set2";
    else if (charMultiset1 > charMultiset2)
        std::cout << "set1 > set2";

    return 0;
}
```

6、map

```
#include <iostream>
#include <map>
typedef std::map<int, char> MYMAP;
int main()
{
    // Create the first map object.
    MYMAP charMap1;
    // Populate the first map with values.
    charMap1[1] = 'A';
    charMap1[4] = 'D';
    charMap1[2] = 'B';
    charMap1[5] = 'E';
    charMap1[3] = 'C';
    // Display the contents of the first map.
    std::cout << "Contents of first map: " << std::endl;
    MYMAP::iterator iter;
    for (iter = charMap1.begin();
          iter != charMap1.end(); iter++)
    {
        std::cout << (*iter).first << " --> ";
        std::cout << (*iter).second << std::endl;
    }
    std::cout << std::endl;
    // Create the second map object.
    MYMAP charMap2;
    // Populate the first map with values.
    charMap2[1] = 'F';
    charMap2[4] = 'I';
    charMap2[2] = 'G';
    charMap2[5] = 'J';
    charMap2[3] = 'H';
    // Display the contents of the second map.
    std::cout << "Contents of second map: " << std::endl;
    for (iter = charMap2.begin();
          iter != charMap2.end(); iter++)
    {
        std::cout << (*iter).first << " --> ";
        std::cout << (*iter).second << std::endl;
    }
```

```
    std::cout << std::endl;
    // Compare the maps.
    if (charMap1 == charMap2)
       std::cout << "map1 == map2";
    else if (charMap1 < charMap2)
       std::cout << "map1 < map2";
    else if (charMap1 > charMap2)
       std::cout << "map1 > map2";

    return 0;
}
```

7、multimap

```
#include <iostream>
#include <map>
typedef std::multimap<int, char> MYMAP;
int main()
{
    // Create the first multimap object.
    MYMAP charMultimap;
    // Populate the multimap with values.
    charMultimap.insert(MYMAP::value_type(1,'A'));
    charMultimap.insert(MYMAP::value_type(4,'C'));
    charMultimap.insert(MYMAP::value_type(2,'B'));
    charMultimap.insert(MYMAP::value_type(7,'E'));
    charMultimap.insert(MYMAP::value_type(5,'D'));
    charMultimap.insert(MYMAP::value_type(3,'B'));
    charMultimap.insert(MYMAP::value_type(6,'D'));
    // Display the contents of the first multimap.
    std::cout << "Contents of first multimap: " << std::endl;
    MYMAP::iterator iter;
    for (iter = charMultimap.begin();
          iter != charMultimap.end(); iter++)
    {
       std::cout << (*iter).first << " --> ";
       std::cout << (*iter).second << std::endl;
    }
    std::cout << std::endl;
    // Create the second multimap object.
    MYMAP charMultimap2;
    // Populate the second multimap with values.
    charMultimap2.insert(MYMAP::value_type(1,'C'));
    charMultimap2.insert(MYMAP::value_type(4,'F'));
    charMultimap2.insert(MYMAP::value_type(2,'D'));
    charMultimap2.insert(MYMAP::value_type(7,'E'));
    charMultimap2.insert(MYMAP::value_type(5,'F'));
    charMultimap2.insert(MYMAP::value_type(3,'E'));
    charMultimap2.insert(MYMAP::value_type(6,'G'));
    // Display the contents of the second multimap.
    std::cout << "Contents of second multimap: " << std::endl;
    for (iter = charMultimap2.begin();
          iter != charMultimap2.end(); iter++)
    {
       std::cout << (*iter).first << " --> ";
       std::cout << (*iter).second << std::endl;
    }
    std::cout << std::endl;
    // Compare the multimaps.
    if (charMultimap == charMultimap2)
```

```
        std::cout << "multimap1 == multimap2";
    else if (charMultimap < charMultimap2)
        std::cout << "multimap1 < multimap2";
    else if (charMultimap > charMultimap2)
        std::cout << "multimap1 > multimap2";

    return 0;
}
```

8、stack

```
#include <iostream>
#include <list>
#include <stack>
int main()
{
    std::stack<int, std::list<int> > intStack;
    int x;
    std::cout << "Values pushed onto stack:"
              << std::endl;
    for (x=1; x<11; ++x)
    {
        intStack.push(x*100);
        std::cout << x*100 << std::endl;
    }
    std::cout << "Values popped from stack:"
              << std::endl;
    int size = intStack.size();
    for (x=0; x<size; ++x)
    {
        std::cout << intStack.top() << std::endl;
        intStack.pop();
    }
    return 0;
}
```

9、queue

```
#include <iostream>
#include <list>
#include <queue>
int main()
{
    std::queue<int, std::list<int> > intQueue;
    int x;
    std::cout << "Values pushed onto queue:"
              << std::endl;
    for (x=1; x<11; ++x)
    {
        intQueue.push(x*100);
        std::cout << x*100 << std::endl;
    }
    std::cout << "Values removed from queue:"
              << std::endl;
    int size = intQueue.size();
    for (x=0; x<size; ++x)
    {
        std::cout << intQueue.front() << std::endl;
        intQueue.pop();
    }
    return 0;
}
```

10、priority_queue

```
#include <iostream>
#include <list>
#include <queue>
int main()
{
   std::priority_queue<int, std::vector<int>,std::greater<int> > intPQueue;
   int x;
   intPQueue.push(400);
   intPQueue.push(100);
   intPQueue.push(500);
   intPQueue.push(300);
   intPQueue.push(200);
   std::cout << "Values removed from priority queue:"
             << std::endl;
   int size = intPQueue.size();
   for (x=0; x<size; ++x)
   {
      std::cout << intPQueue.top() << std::endl;
      intPQueue.pop();
   }
   return 0;
}
```

- **【10】迭代器与容器**
  - 只有顺序容器和关联容器支持迭代器遍历，各容器支持的迭代器的类别如下：
- **【11】额外的理解**
  - 【1】**迭代器就是对指针的一层封装，提供了统一的接口。**使用迭代器的好处有：第一，访问数据内容，同时不暴露其内部结构，降低耦合性；第二，支持multiple traversal（同时有多个遍历发生）；第三，提供统一的访问接口和多态遍历（该多态为静态多态，发生在编译期间）。
  - 【2】迭代器与指针
    - 因为迭代器实际上是指针的抽象，很多功能概念都是从指针上“扒下”来的，所以它的语义根指针是一致的。
    - 这意味着可以传入指针作为迭代器，因为指针上的操作（递增，递减，算术运算等）都是迭代器的超集，模板定义对迭代器所做出的操作要求放在指针上完全使用。这在操作内置数组的时候，可以省去不少麻烦（不用再去多敲多余的代码来生成迭代器）：

      ```
      int numbers[ ] = {1,2,3,4};
      std :: find (numbers, numbers+4, 2);
      ```
  - 【3】迭代器基础设施

[4] 具体容器使用

- **【1】vector**
  - 基本知识
    - vector(向量): C++中的一种数据结构,确切的说是一个类.它相当于一个动态的数组,当程序员无法知道自己需要的数组的规模多大时,用其来解决问题可以达到最大节约空间的目的.
    - 文件包含:首先在程序开头处加上#include<vector>以包含所需要的类文件vector。还有一定要加上using namespace std;

- 变量声明: 例:声明一个int向量以替代一维的数组:vector <int> a;(等于声明了一个int数组a[],大小没有指定,可以动态的向里面添加删除)。例:用vector代替二维数组.其实只要声明一个一维数组向量即可,而一个数组的名字其实代表的是它的首地址,所以只要声明一个地址的向量即可,即:vector <int *> a. 同理想用向量代替三维数组也是一样,vector <int**>a;再往上面依此类推.

- 具体的用法以及函数调用:
  - 如何得到向量中的元素?其用法和数组一样:

    例如:

    ```
    vector <int *> a
    int b = 5;
    a.push_back(b);//该函数下面有详解
    cout<<a[0];      //输出结果为5
    ```

  - 方法:

    1.push_back 在数组的最后添加一个数据
    2.pop_back 去掉数组的最后一个数据
    3.at 得到编号位置的数据
    4.begin 得到数组头的指针
    5.end 得到数组的最后一个单元+1的指针
    6．front 得到数组头的引用
    7.back 得到数组的最后一个单元的引用
    8.max_size 得到vector最大可以是多大
    9.capacity 当前vector分配的大小
    10.size 当前使用数据的大小
    11.resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值
    12.reserve 改变当前vecotr所分配空间的大小
    13.erase 删除指针指向的数据项
    14.clear 清空当前的vector
    15.rbegin 将vector反转后的开始指针返回(其实就是原来的end-1)
    16.rend 将vector反转构的结束指针返回(其实就是原来的begin-1)
    17.empty 判断vector是否为空
    18.swap 与另一个vector交换数据

  - 详细的函数实现功能：其中vector<int> c.

    c.clear() 移除容器中所有数据。
    c.empty() 判断容器是否为空。
    c.erase(pos) 删除pos位置的数据
    c.erase(beg,end) 删除[beg,end)区间的数据
    c.front() 传回第一个数据。
    c.insert(pos,elem) 在pos位置插入一个elem拷贝
    c.pop_back() 删除最后一个数据。
    c.push_back(elem) 在尾部加入一个数据。
    c.resize(num) 重新设置该容器的大小
    c.size() 回容器中实际数据的个数。
    c.begin() 返回指向容器第一个元素的迭代器
    c.end() 返回指向容器最后一个元素的迭代器

- vector对象初始化
  - 第一类

    ```
    //vector<T> v(n)形式，v包含n 个元素
    vector<int> ivec1(10);
    //vector<T> v(n,i)形式，v包含n 个值为 i 的元素
    vector<int> ivec(10,0);
    for(int_ite=ivec.begin ();int_ite!=ivec.end ();int_ite++)
    cout<<"ivec: "<<*int_ite<<endl;
    ```

- 第二类：拷贝一份已有的vector

```
//vector<T> v(v1)形式，v是v1 的一个副本
vector<int> ivec1(ivec);
for(int_ite=ivec1.begin ();int_ite!=ivec1.end ();int_ite++)
cout<<"ivec1: "<<*int_ite<<endl;
```

- 第三类：用int/string数组初始化vector数组

```
//数组初始化vector
int iarray[]={1,2,3,4,5,6,7,8,9,0};
//count： iarray数组个数
size_t count=sizeof(iarray)/sizeof(int);
//int数组初始化 ivec3
vector<int> ivec3(iarray,iarray+count);
for(int_ite=ivec3.begin ();int_ite!=ivec3.end ();int_ite++)
cout<<"ivec3: "<<*int_ite<<endl;
//string数组初始化 svec1
string word[]={"ab","bc","cd","de","ef","fe"};
//s_count： word数组个数
size_t s_count=sizeof(word)/sizeof(string);
//string数组初始化 svec1
vector<string> svec1(word,word+s_count);
for(string_ite=svec1.begin ();string_ite!=svec1.end ();string_ite++)
cout<<"svec1: "<<*string_ite<<endl;
```

- 第四类：使用back_inserter函数初始化vector

```
//用 back_inserter 函数
vector<int> ivec4; //空对象
fill_n(back_inserter(ivec4),10,3); //10个3 填充ivec4.
for(int_ite=ivec4.begin ();int_ite!=ivec4.end ();int_ite++)
cout<<"ivec4: "<<*int_ite<<endl;
}
```

- **【2】deque**
-
-
-
- **【2】for_each**
  - for_each用于逐个遍历容器元素，它对迭代器区间[first，last)所指的每一个元素，执行由单参数函数对象f所定义的操作。

```
template<class InputIterator, class Function>
  Function for_each(
    InputIterator _First,
    InputIterator _Last,
    Function _Func
    );
```

  - for_each 算法范围 [first, last) 中的每个元素调用函数 F，并返回输入的参数 f。此函数不会修改序列中的任何元素。

```
void print_element(int n) {
   cout << n ;

int a[] = { 1, 3, 2, 3, 4, 5 };
vector<int> v1(a,a+6);
   for_each(v1.begin(), v1.end(), print_element);
   cout << endl;
```

- **【3】remove**

- 第六部分：概念比较
  - **[1] 多态与重载的区别**
    - 相同之处：多态是基于对抽象方法的覆盖来实现的，用统一的对外接口来完成不同的功能。重载也是用统一的对外接口来完成不同的功能。
    - 不同之处：**重载**，是指允许存在多个同名方法，而这些方法的参数不同。重载的实现是：编译器根据方法不同的参数表，对同名方法的名称做修饰。对于编译器而言，这些同名方法就成了不同的方法。它们的调用地址在编译期就绑定了。**多态**：是指子类重新定义父类的虚方法（virtual,abstract）。当子类重新定义了父类的虚方法后，父类根据赋给它的不同的子类，动态调用属于子类的该方法，这样的方法调用在编译期间是无法确定的。不难看出，**两者的区别在于编译器何时去寻找所要调用的具体方法，对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为“早绑定”或“静态绑定”；而对于多态，只有等到方法调用的那一刻，编译器才会确定所要调用的具体方法，这称为“晚绑定”或“动态绑定”。**
    - **个人理解：重载是从函数角度来理解“一个函数对应多种功能”，参数列表作为区分；多态是从类的角度来理解“一个大类的不同子类之间的细微差别”，通过虚函数来作区分。**
  - **[2] 关于C++ STL标准模板库**
    - **【1】概述**
      - STL（Standard Template Library），即标准模板库，是一个具有工业强度的，高效的C++程序库。它被容纳于C++标准程序库（C++ Standard Library）中，是ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。为广大C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。
      - 从逻辑层次来看，在STL中体现了泛型化程序设计的思想（generic programming），引入了诸多新的名词，比如像需求（requirements），概念（concept），模型（model），容器（container），算法（algorithmn），迭代子（iterator）等。与OOP（object-oriented programming）中的多态（polymorphism）一样，泛型也是一种软件的复用技术；
      - 从实现层次看，整个STL是以一种类型参数化（type parameterized）的方式实现的，这种方式基于一个在早先C++标准中没有出现的语言特性--模板（template）。如果查阅任何一个版本的STL源代码，你就会发现，模板作为构成整个STL的基石是一件千真万确的事情。除此之外，还有许多C++的新特性为STL的实现提供了方便；
    - **【2】STL六大组件**
      - **容器**（Container），是一种数据结构，如list，vector，和deques，以模板类的方法提供。为了访问容器中的数据，可以使用由容器类输出的迭代器；
      - **迭代器**（Iterator），提供了访问容器中对象的方法。例如，可以使用一对迭代器指定list或vector中的一定范围的对象。迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。但是，迭代器也可以是那些定义了operator*()以及其他类似于指针的操作符地方法的类对象；
      - **算法**（Algorithm），是用来操作容器中的数据的模板函数。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用；

- **仿函数**（Function object，仿函数(functor)又称之为函数对象（function object），其实就是重载了()操作符的struct，没有什么特别的地方
- **迭代适配器**（Adaptor）
- **空间配制器**（allocator）其中主要工作包括两部分1.对象的创建与销毁　2.内存的获取与释放

- **【3】STL算法**
  - **概述：**
    - STL算法部分主要由头文件<algorithm>,<numeric>,<functional>组成。要使用 STL中的算法函数必须包含头文件<algorithm>，对于数值算法须包含<numeric>，<functional>中则定义了一些模板类，用来声明函数对象。
    - STL中算法大致分为四类：
    - 1）、非可变序列算法：指不直接修改其所操作的容器内容的算法。
    - 2）、可变序列算法：指可以修改它们所操作的容器内容的算法。
    - 3）、排序算法：包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作。
    - 4）、数值算法：对容器内容进行数值计算。
  - **查找算法**(13个)
    判断容器中是否包含某个值
    - **adjacent_find**:　　在iterator对标识元素范围内，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的ForwardIterator。否则返回last。重载版本使用输入的二元操作符代替相等的判断。
    - **binary_search**:　　在有序序列中查找value，找到返回true。重载的版本实用指定的比较函数对象或函数指针来判断相等。
    - **count**:　　利用等于操作符，把标志范围内的元素与输入值比较，返回相等元素个数。
    - **count_if**:　　利用输入的操作符，对标志范围内的元素进行操作，返回结果为true的个数。
    - **equal_range**:　　功能类似equal，返回一对iterator，第一个表示lower_bound，第二个表示upper_bound。
    - **find**:　　利用底层元素的等于操作符，对指定范围内的元素与输入值进行比较。当匹配时，结束搜索，返回该元素的一个InputIterator。
    - **find_end**:　　在指定范围内查找"由输入的另外一对iterator标志的第二个序列"的最后一次出现。找到则返回最后一对的第一个ForwardIterator，否则返回输入的"另外一对"的第一个ForwardIterator。重载版本使用用户输入的操作符代替等于操作。
    - **find_first_of**:　　在指定范围内查找"由输入的另外一对iterator标志的第二个序列"中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符。
    - **find_if**:　　使用输入的函数代替等于操作符执行find。
    - **lower_bound**:　　返回一个ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置。重载函数使用自定义比较操作。
    - **upper_bound**:　　返回一个ForwardIterator，指向在有序序列范围内插入value而不破坏容器顺序的最后一个位置，该位置标志一个大于value的值。重载函数使用自定义比较操作。

- **search**: 给出两个范围，返回一个ForwardIterator，查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向last1。重载版本使用自定义的比较操作。
- **search_n**: 在指定范围内查找val出现n次的子序列。重载版本使用自定义的比较操作。

- **排序和通用算法**(14个)

  提供元素排序策略

  - **inplace_merge**: 合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序。
  - **merge**: 合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较。
  - **nth_element**: 将范围内的序列重新排序，使所有小于第n个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作。
  - **partial_sort**: 对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作。
  - **partial_sort_copy**: 与partial_sort类似，不过将经过排序的序列复制到另一个容器。
  - **partition**: 对指定范围内元素重新排序，使用输入的函数，把结果为true的元素放在结果为false的元素之前。
  - **random_shuffle**: 对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作。
  - **reverse**: 将指定范围内元素重新反序排序。
  - **reverse_copy**: 与reverse类似，不过将结果写入另一个容器。
  - **rotate**: 将指定范围内元素移到容器末尾，由middle指向的元素成为容器第一个元素。
  - **rotate_copy**: 与rotate类似，不过将结果写入另一个容器。
  - **sort**: 以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作。
  - **stable_sort**: 与sort类似，不过保留相等元素之间的顺序关系。
  - **stable_partition**: 与partition类似，不过不保证保留容器中的相对顺序。

- **删除和替换算法**(15个)

  - **copy**: 复制序列
  - **copy_backward**: 与copy相同，不过元素是以相反顺序被拷贝。
  - **iter_swap**: 交换两个ForwardIterator的值。
  - **remove**: 删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用remove和remove_if函数。
  - **remove_copy**: 将所有不匹配元素复制到一个制定容器，返回OutputIterator指向被拷贝的末元素的下一个位置。
  - remove_if: 删除指定范围内输入操作结果为true的所有元素。
  - **remove_copy_if**: 将所有不匹配元素拷贝到一个指定容器。

- **replace**: 将指定范围内所有等于vold的元素都用vnew代替。
- **replace_copy**: 与replace类似，不过将结果写入另一个容器。
- **replace_if**: 将指定范围内所有操作结果为true的元素用新值代替。
- **replace_copy_if**: 与replace_if，不过将结果写入另一个容器。
- **swap**: 交换存储在两个对象中的值。
- **swap_range**: 将指定范围内的元素与另一个序列元素值进行交换。
- **unique**: 清除序列中重复元素，和remove类似，它也不能真正删除元素。重载版本使用自定义比较操作。
- **unique_copy**: 与unique类似，不过把结果输出到另一个容器。

- **排列组合算法**(2个)

  提供计算给定集合按一定顺序的所有可能排列组合

  - **next_permutation**: 取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作。
  - **prev_permutation**: 取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回false。重载版本使用自定义的比较操作。

- **算术算法**(4个)
  - **accumulate**: iterator对标识的序列段元素之和，加到一个由val指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上。
  - **partial_sum**: 创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法。
  - **inner_product**: 对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。重载版本使用用户定义的操作。
  - **adjacent_difference**: 创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差。

- **生成和异变算法**(6个)
  - **fill**: 将输入值赋给标志范围内的所有元素。
  - **fill_n**: 将输入值赋给first到first+n范围内的所有元素。
  - **for_each**: 用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素。
  - **generate**: 连续调用输入的函数来填充指定的范围。
  - **generate_n**: 与generate函数类似，填充从指定iterator开始的n个元素。
  - **transform**: 将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器。

- **关系算法**(8个)
  - **equal**: 如果两个序列在标志范围内元素都相等，返回true。重载版本使用输入的操作符代替默认的等于操作符。

- **includes**: 判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的<操作符，成功返回true。重载版本使用用户输入的函数。
- **lexicographical_compare**: 比较两个序列。重载版本使用用户自定义比较操作。
- **max**: 返回两个元素中较大一个。重载版本使用自定义比较操作。
- **max_element**: 返回一个ForwardIterator，指出序列中最大的元素。重载版本使用自定义比较操作。
- **min**: 返回两个元素中较小一个。重载版本使用自定义比较操作。
- **min_element**: 返回一个ForwardIterator，指出序列中最小的元素。重载版本使用自定义比较操作。
- **mismatch**: 并行比较两个序列，指出第一个不匹配的位置，返回一对iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的last。重载版本使用自定义的比较操作。

- **集合算法**(4个)
  - **set_union**: 构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作。
  - **set_intersection**: 构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作。
  - **set_difference**: 构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作。
  - **set_symmetric_difference**: 构造一个有序序列，该序列取两个序列的对称差集(并集-交集)。
- **堆算法**(4个)
  - **make_heap**: 把指定范围内的元素生成一个堆。重载版本使用自定义比较操作。
  - **pop_heap**: 并不真正把最大元素从堆中弹出，而是重新排序堆。它把first和last-1交换，然后重新生成一个堆。可使用容器的back来访问被"弹出"的元素或者使用pop_back进行真正的删除。重载版本使用自定义的比较操作。
  - **push_heap**: 假设first到last-1是一个有效堆，要被加入到堆的元素存放在位置last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作。
  - **sort_heap**: 对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作。

- **【4】适配器**
  - **概述：**
    - STL提供了三个容器适配器：queue、priority_queue、stack。这些适配器都是包装了vector、list、deque中某个顺序容器的包装器。注意：**适配器没有提供迭代器，也不能同时插入或删除多个元素。下面对各个适配器进行概括总结。**
  - **stack用法**
  - **queue用法**
  - **priority_queue用法**

幕布 - 思维概要整理工具