基础篇

1. C++和C相比最大的特点
   1）面向对象：封装，继承，多态。
   2）引入引用代替指针。
   3）const/inline/template替代宏常量。
   4）namespace解决重名的问题。
   5）STL提供高效的数据结构和算法

- C++其实是以C为基础的，比如语句、预处理器、内置数据类型、指针、数组等等都来自于c。c更注重于数据结构和算法，但是c的缺陷在于没有模板、没有异常、没有重载。
- c++也是c with classes， c++通过类（构造函数、析构函数）、继承、封装、多态、virtual函数（动态绑定）等等实现面向对象设计。
- c++还支持template泛型编程。
- c++用STL template程序库，包含容器、迭代器和算法、函数对象的规约。

1. 类
   c++提供的自定义数据类型的机制，可包含数据、函数和类型成员。一个类定义了一种新的数据结构和一个新的作用域。

2. 继承
   通过一个基类来定义多个派生类。派生类可以获得基类中的大部分特性，减少一定的代码量。

3. 封装
   分离类的实现和接口，隐藏类的实现细节。

   - 接口
     类型提供的一些公有的操作，通常情况下不包含数据成员

4. 多态
   多态就是一种事物的多种表现形态，也可以说是“一个接口的多种实现”

   - 分类
     静态多态：编译时的多态,包括函数的重载和泛型编程，程序调用函数，由编译器来决定使用哪个可执行代码块。
     动态多态：运行时的多态，通过继承和虚函数的实现。通过指向派生类的基类指针或引用来访问派生类同名的覆盖成员函数。
   - 多态的实现。
     简而言之编译器根据虚函数表找到恰当的虚函数。对于一个父类的对象指针类型变量，如果给他赋父类对象的指针，那么他就调用父类中的函数，如果给他赋子类对象的指针，他就调用子类中的函数。函数执行之前查表。

5. 虚函数相关的问题

   - 虚函数是什么
     实现多态所必须，父类类型的指针指向子类的实例，执行的时候会执行子类中定义的函数。
     在基类中用virtual关键字声明，并且在一个或者多个派生类中被重新定义的成员函数。
   - 虚函数的实现
     对象携带相应的信息，在运行期时用来决定要调用哪个版本的虚函数，一般由vptr指针指出。vptr指向一个由函数指针组成的数组vtbl，每一个带虚函数的类都对应一个vtbl，当调用某一virtual函数时，实际被调用的函数取决于该对象的vptr所指向的vtbl，编译器在其中查找对应的函数指针，并执行。
   - 析构函数可以是虚函数吗？
     如果有子类的话，析构函数必须是虚函数。否则析构子类类型的指针时，析构函数有可能不会被调用到。造成内存泄漏。
   - 虚函数表是针对类还是针对对象的?
     虚函数表是针对类的，一个类的所有对象的虚函数表都一样。
   - 纯虚函数和虚函数有什么区别
     纯虚函数就是定义了一个虚函数但并没有实现，原型后面加"=0"。包含纯虚函数的类都是抽象类，不能生成实例。
   - 构造函数可以是虚函数吗？

每个对象的虚函数表指针是在构造函数中初始化的，因为构造函数没执行完，所以虚函数表指针还没初始化好，构造函数的虚函数不起作用。

- 构造函数中可以调用虚函数吗？
  就算调用虚函数也不起作用，调用虚函数同调用一般的成员函数一样。在派生类还没有构成的时候，编译器和运行期的类型判断都会解析为基类对象。
- 析构函数中可以调用虚函数吗？
  析构函数中调用虚函数也不起作用，调用虚函数同调用一般的成员函数一样。析构函数的顺序是先派生类后基类，有可能内容已经被析构没了，所以虚函数不起作用。
- 虚继承和虚基类？
  虚继承是多重继承的一种，为了解决多重继承出现菱形继承时出现的问题。例如：类B、C分别继承了类A。类D多重继承类B和C的时候，类A中的数据就会在类D中存在多份。通过声明继承关系的时候加上virtual关键字可以实现虚继承。
  虚继承的目的是令某个类作出声明，承诺愿意共享他的基类，共享的基类子对象叫做虚基类。在这种机制下，无论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。注意：虚基类是由最底层的派生类初始化的。也就是说创建D对象时，由D来负责初始化共享的A基类部分。顺序是：先构造A, 在构造B,C，最后构造D；如果一个类有多个虚基类，那么先构造虚基类，在安装声明顺序逐一构造其他非虚基类。

6. 预处理、const和sizeof

- const 有什么用途
  主要有三点：
  1：定义只读变量，即常量
  2：修饰函数的参数和函数的返回值
  3： 修饰函数的定义体，这里的函数为类的成员函数，被const修饰的成员函数代表不修改成员变量的值
- const与宏的区别
  const常量有数据类型，而宏常量没有数据类型。前者可以进行类型安全检查，后者只能进行字符替换。
  有一些集成化的工具可以对const常量进行调试，但不能对宏变量进行调试。const常量完全可以替换宏常量。
- 宏与内联函数的差别是什么
  宏是由预处理器对宏进行替换，不会写入符号表
  内联是由编译器来控制实现的，它是真正的函数，使用时展开代码取消了函数参数的压栈出栈，减少调用的消耗；内联函数定义的代码放入符号表，使用时直接替换，没有调用的消耗，效率很高。内联可以作为类的成员函数，可以访问该类的保护成员和私有成员
  内联优于宏，能够进行参数类型检查，保证调用正确
- sizeof与strlen的区别
  sizeof是运算符，strlen是函数
  sizeof是求一个类型所占内存的大小，而strlen是求字符串的长度
  数组传递给sizeof不会退化为指针，而传递给strlen会退化为指针
  sizeof可以用类型来做参数，strlen只能用char*来做参数，且必须以'\0'为结尾。'\0'记作一位
  char str[20] = '0123456789';
  strlen(str) = 11; sizeof(str)=20;
- 内存对齐问题

7. 指针和引用

- 指针和引用的区别
  1：引用是变量的一个别名，内部实现是只读指针
  2：引用只能在初始化时被赋值，其他时候值不能被改变，指针的值可以在任何时候被改变
  3：引用不能为NULL，指针可以为NULL
  4：引用变量内存单元保存的是被引用变量的地址
  5：“sizeof 引用" = 指向变量的大小， "sizeof 指针"= 指针本身的大小
  6：引用可以取地址操作，返回的是被引用变量本身所在的内存单元地址
  7：引用使用在源代码级相当于普通的变量一样使用，做函数参数时，内部传递的实际是变量地址

- 成员变量可以是引用吗
  可以。引用类型的成员变量必须在构造函数的初始化列表中进行初始化，因此正确的写法是：
  `class Test { private: int &a; public: Test(int &b) : a(b)//注意写法 { } };`

- 传递动态内存

- 函数指针

- 指针数组和数组指针
  指针数组：int (* ptr)[], 数组里存储的都是指针地址。如果ptr+1，加的是整个数组的大小
  数组指针：int * (ptr[])， 指针指向一个数组。

- 指针加减法

- 迷途指针
  迷途指针是在程序员对一个指针进行delete操作后，释放它所指的内存，但是并没有将它设置为空时产生的。
  因为这个指针仍然会指向原来的内存区域，但是编译器已经将这块内存区域分配给其他的数据，如果再次使用时会引起程序崩溃。
  安全的做法是：删除一个指针后，将它设置为空。

- this指针
  this指针本质上是一个函数参数，只是编译器隐藏起形式的，语法层面上的参数。只能在成员函数中使用，全局函数、静态函数都不能使用this。
  this指针是在成员函数之前构造，在成员函数结束后清除。编译器会对this指针进行优化，传递效率提高。
  this指针不占用对象的空间。
  this指针可能存放在栈、堆或者寄存器。在实际应用中，this应该是寄存器参数，大多数编译器通过ecx寄存器传递this指针。
  this指针只有在成员函数中才有定义，因此可在成员函数中通过`&this`来获取this指针的位置。

- C++中有了malloc/free , 为什么还需要 new /delete
  1：malloc与free是C++/C语言的标准库函数，new /delete是C++的运算符。它们都可用于申请动态内存和释放内存。
  2：对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。
  对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。
  由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。
  3：因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以一个能完成清理与释放内存工作的运算符delete。注意new /delete不是库函数.
  4: new 建立一个对象，new 出来的对象可以通过成员函数访问，并且是带类型的信息。而malloc分配的是一块内存，在内存区域中可以移动指针，并且malloc返回的是void指针。

8. 友元类
   友元类向外部提供了其非公有成员访问权限的一种机制。友元的访问权限与成员函数一样。友元可以是类，也可以是函数。

9. 一个由c/C++编译的程序占用的内存
   一个由c/C++编译的程序占用的内存分为以下几个部分
   1：栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
   2：堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收。
   3：全局区（静态区）（static）— 全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。
   4：文字常量区 — 常量字符串就是放在这里的。程序结束后由系统释放。
   5：程序代码区 — 存放函数体的二进制代码。

区别	堆	栈
申请方式	由程序员申请	由系统自动分配
申请后系统响应	遍历一个记录空闲内存地址的链表	只要栈的剩余空间大于申请的空间，就分配内存
申请大小的限制	向高地址拓展的数据结构，不连续的内存区域	向低地址拓展的数据结构，连续的内存区域
申请效率	速度慢	速度快

11. 关键字static的作用

该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值

- 在c语言中，static主要定义全局静态变量和局部静态变量以及静态函数
  内存分配：在程序的全局数据区分配。
  如果没有初始化的话，取其默认值0.
  全局的话，变量从文件定义开始到文件结束始终可见。局部的话作用于为局部作用域，定义他的函数或语句块结束时，其作用域也随之结束。静态函数只能在本源文件中使用。
- 在c++语言中，static新增了定义静态成员函数和静态数据成员。
  内存分配：在程序的全局数据区分配。
  静态数据成员定义时要分配空间，不能在类声明中定义。
  静态成员函数不能访问非静态数据成员。主要用于对静态数据成员的操作。

11. 在c++程序中调用被C编译器编译后的函数，为什么要加extern“C”
    C++语言支持函数重载，C语言不支持函数重载，函数被C++编译器编译后在库中的名字与C语言的不同，
    假设某个函数原型为：

```
void foo(int x, inty);
```

该函数被C编译器编译后在库中的名字为: `_foo`
而C++编译器则会产生像: `_foo_int_int` 之类的名字。
为了解决此类名字匹配的问题，C++提供了C链接交换指定符号 extern "C"。

13. 头文件种的ifndef/define/endif 是干什么用的
    是条件编译的一种，处理防止头文件被重复引用，还可以防止重复定义(变量、宏或者结构)。防止头文件被重复包含

14. 重载、重写、重定义

    - 重载
      - 概念：函数有同样的名称，但是参数列表不相同的情形。这样的同名不同参数的函数之间，互称为重载函数
      - 条件：函数名相同；函数参数必须不同（参数类型、参数个数）；返回值可以相同，可以不同
      - 注意：只能通过不同参数样式进行重载，不能通过访问权限（返回类型、抛出异常）进行重载，重载函数应该在相同的作用域中
    - 重写
      - 概念：也叫覆盖，子类重新定义父类中具有相同名称和参数的虚函数，主要在继承关系中出现
      - 条件：重写函数和被重写函数必须为virtual函数，重写和被重写函数的函数名和参数必须一致；重写函数和被重写函数的返回值一致；访问修饰可以不同（private、protected、public）；抛出来的异常必须相同
    - 重定义
      - 概念：也叫隐藏，子类重新定义父类中的非虚函数，屏蔽了父类的同名函数
      - 条件：被隐藏的函数之间作用域不相同

自己实现一个String类

```
Class String{
    private:
        char * data_;
    public:
        String();
        String(const char* );
        String(const String& rhs);
        String& operator=(const String& rhs);
        ~String();
```

```
}

String::String(const String& rhs) {
    if (&rhs!=this) {
        delete [] data_;
        data_ = new char[rhs.size() + 1];
        memcpy(data_, rhs.c_str(), rhs.size());
    }
    return *this;
}

String::~String() {
    delete [] data_;

String::String& operator=(const String& rhs){
    if (&rhs!=this) {
        String tmp(rhs);
        char * str = rhs.data_;
        rhs.data_ = this.data_;
        this.data_ = str;
    }
    return *this;
}
```

13. 说说源代码到最后的可执行文件经历的过程，动态链接和静态链接的区别，优缺点，怎么让程序使用动态，静态链接
    - c++从编译到运行经过哪几步
      预处理-编译-汇编-链接
    i. 预处理：hello.c源文件与相关的头文件被预处理器预编译成一个.i文件
      将#define删掉，展开所有宏定义；
      处理条件预编译ifdef命令；
      处理#include预编译指令，把包含的文件插入到预编译指令的位置；
      删除注释；
      添加行号、文件名标识
    ii. 编译：把预处理完的文件进行一系列词法分析、语法分析、语义分析和优化后生成相应的汇编代码文件。将.i文件转为.s文件
    iii. 汇编：汇编器将汇编代码转为机器可以执行的指令，每个汇编语言都会对应一条机器指令。将.c文件转为.o文件
    iv. 得到.out文件。当一个程序很复杂的时候，我们把每个源代码模块独立的编译，然后按照要求进行组装，这个组装的过程就是链接。主要内容是将每个模块相互引用的部分都处理好，使得每个模块之间能够正确地衔接。
      .o文件+libray=.out文件
      a. 地址和空间分配 b. 符号决议 c. 重定位
    - 静态链接
      - 定义
        静态链接库：将(lib)文件用到的函数代码直接链接(拷贝)到目标程序，程序运行的时候不需要其他库文件
      - 优点
        代码装载速度快，执行速度比动态链接快
        只需要保证开发者的计算机有正确的.lib文件
      - 缺点
        生成的可执行代码比较大，可能会含有相同的重复的公共代码
    - 动态链接
      - 定义
        动态链接库：把调用的函数所在的文件模块(dll)和调用函数所在文件的位置等信息链接进目标程序，在程序运行时再从dll中寻找相应的函数代码，因此需要相应的dll文件支持。
      - 优点
        节省内存，并且减少内存交换
        DLL文件与exe文件独立，只要保证接口不变，就可以更换dll文件，提高维护性和可拓展性
        对于不同的语言可以用同一个dll
        适合于大规模的软件开发，使开发过程独立，耦合度小，便于不同开发者和开发组织之间进行开发和测试。
      - 不足
        如果程序以来的dll不存在，载入时进行动态链接，程序会终止并给出错误提示；如果是运行时动态链接，程序加载失败。速度慢
    - 区别：

i. 在静态链接中，lib的指令会被包含到最终生成的exe文件中，在动态链接中，dll不必包含在最终的exe文件中，exe文件可以随时动态的引用和卸载这个与exe独立的dll文件。
ii. 在静态链接库中不能再包含其他动态链接库或者静态库；而在静态链接库中可以包含其他动态链接库和静态库。
iii. 如果多处对dll文件的同一个函数进行调用，那么只会留下一处拷贝；而如果对lib文件同一个函数调用，每一处调用会多一处拷贝，会有多份拷贝

STL各种问题

1. 容器的分类。
   序列式容器
   array(c++内建)
   vector：维护一段连续线性空间，支持随机读取
   heap(算法实现): priority_queue(优先队列)
   list：可以在两头插入，双向链表
   slist
   deque
   适配器(adapter，以deque为底层实现)
   queue：先进先出
   stack：后进先出
   关联式容器
   以RB-Tree为底层实现的：set、map
   以hash_table为底层实现的：hash-set、hash-map

2. vector，list，deque的实现。
   vector是一块连续内存，当空间不足了会再分配(两倍原则)。
   list是双向链表。
   deque是双端队列可在头和尾部插入、删除元素。

3. hashmap和map有什么区别。
   底层实现不同。一个是基于hash table实现，一个是基于红黑树实现。
   rb-tree就是一个树状的结构，二叉搜索平衡树，hash table是有vector和linked list来实现的。对于vector中的每个值，我们称之为bucket，也叫桶，如果出现冲突的时候，用链式法来解决冲突。

4. 红黑树有什么特性
   红黑树是平衡二叉树，具有以下特性:
   1) 每个节点不是红色就是黑色
   2) 根节点是黑色
   3) 如果节点是红色，那么他的子节点一定是黑色
   4) 任何一个节点到NULL(树尾端)的任何路径，所包含的黑节点数目必须相同
   5) 每个叶子节点都是黑色的空节点（NIL节点）

5. 配接器 allocator

- SGI标准的空间配置器是allocator, 只是基层内存配置/释放行为(new /delete)的一层薄薄的封装，没有考虑到效率上的强化。
- SGI特殊的空间配置器是alloc，
  - 空间的配置和释放
    SGI以malloc()和free()完成内存的配置和释放。考虑到小型区块可能造成内存破碎问题，SGI设计了双层级配置器。第一层直接用malloc和free，第二级配置器视情况采用不同策略。如果配置区别大于128bytes，选用第一级配置器；当小于128bytes时，采用复杂的memory pool的整理方式。是否同时开放第二级配置器，取决于__USE_MALLOC是否被定义。
  - STL里面空间分配是怎么样的？
    STL中用allocator类来实现空间分配，有一级配置器二级配置器,根据一个环境组态来确定使用哪一级的配置器。SGI STL将alloc设置为第二级配置器。
  - 如果用让你写一个STL的空间配置器，这个配置器需要经常分配大量的小内存，但大量的分配小内存会造成内存碎片，你会怎么解决这个问题？那如果用你实现的配置器分配的空间是怎么释放的？（扯一些二级配置器是怎么实现的，可以参照STL特有的空间配置器alloc来想）
    a. 针对于小内存的问题，可以像第二级配置器一样来实现，用内存池来管理。每次配置一大块内存，然后维护16个自

由链表free-lists。每个链表会维护若干个小额区块，每个小额区块的大小是8的倍数。

b. 如果自由链表里的小额区块不够了，利用chunk_alloc(size,num)，从内存池中取空间给free-list使用（参考上面的内存池内容）

c. 如果客端申请需求量小于128byte，会将申请的内存需求量上调至8的倍数，然后再查找对应的自由链表中的小额区块。如果大于128byte，就用第一级配置器的allocate()函数，直接用malloc来配置内存。

- 内存池

  chunk_alloc的工作：从内存池中取空间给free-list使用。

  `if(内存池水量足够) 直接调出20个区块给free list else if(内存池水量还足够提供至少1个区块) 调出实际能够供应的区块 else 利用malloc()向堆heap中配置内存，为内存池注入源头活水以应付需求。一般申请为需求量的两倍，再加上随着配置次数增加而越来越大的附加量 if(heap的空间也不够) malloc()失败，调用第一级配置器中的out of memory处理机制,或许有机会释放其内存拿来此处使用。如果可以就成功，否则发出bad_alloc异常。`

a. 释放内存时，如果释放量小于128byte，配置器会将它归还到对应的自由链表中。如果大于128byte，就用第一级配置器的deallocate()函数，直接用free来释放内存。

1. STL仿函数。
   - STL仿函数

     也叫函数对象，是一种具有函数性质的对象。
   - 为什么要有仿函数

     可以看到STL的算法一般会提供两个版本，一个版本是已经定义好的常用的某种算法，另一个版本是表现出泛化的演算流程。允许用户“以template参数来指定所要采取的策略”。比方说sort这个函数，版本一是默认用Operator<作为排序的依据，而版本2允许用户自定义比较的依据操作，使得排序后两两之间的比较会使这个操作为true。
   - 方式——将整组操作当做算法

     将这样的操作设计为一个函数，再将函数指针作为算法的一个参数，或者将这个操作设计为一个仿函数，再以这个仿函数生成一个对象，以这个对象作为算法的一个参数
   - 为什么还要用仿函数，函数指针不行么

     函数指针确实可以做到将整组操作当做算法的参数，但是不能满足STL对抽象性的要求，不能满足软件积木的要求——函数指针无法与STL其他组件(比如adapter)搭配，产生更灵活的配置。
   - 分类

     以操作数的个数划分：一元、二元仿函数(定义了两个class)

     以功能划分：算术类仿函数、逻辑运算类仿函数、关系运算类仿函数

```
template<class Arg1, class Arg2, class Result>
struct binary_function{
    typedef Arg1 first_argument_type;//第一参数型别
    typedef Arg2 first_argument_type;//第二参数型别
    typedef Result result_type;//回返值型别

};
//以下仿函数继承了binary_function，用户可以得到该仿函数的各种相应型别
template<class T>
struct plus: public binary_function<T,T,T>{
    T operator()(const T& x, const T&y) const{return x+y;}
};

//产生仿函数实体
plus<int> plusobj;
//运用上述对象，履行函数功能
cout<<plusobj(3,5)<<endl;
//直接以仿函数的临时对象履行函数功能
cout<<plus<int>()(3,5)<<endl;
//搭配STL算法
accumulate(iv.begin(), iv.end(), 1, multiplies<int>());
```

用一个场景来说明为什么函数指针不能替换仿函数

```
bool my_count(int num)
{
    return (num < 5);
}
```

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> v_a(a, a+10);
cout << "count: " << std::count_if(v_a.begin(), v_a.end(), my_count);
在上面我们传递进去了一个函数指针作为count_if的比较条件。但是现在根据新的需求，不再统计容器中小于5的变量个数，改为了8或者3。那么最直接的
bool my_count(int num， int threshold)
{
    return (num < threshold));
}
但是这样的写法STL中是不能使用的，而且当容器中的元素类型发生变化的时候就不能使用了，更要命的是不能使用模板函数。
那么，既然多传递传递参数不能使用，那就把需要传递进来的那个参数设置为全局的变量，那样确实能够实现当前情况下对阈值条件的修改，但是修改起来
template<typename T> struct my_count1
{
    my_count1(T a)
    {
        threshold = a;
    }
    T threshold;
    bool operator()(T num)
    {
        return (num < threshold);
    }
};

int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> v_a(a, a+10);

cout << "count: " << std::count_if(v_a.begin(), v_a.end(), my_count1<int>(8));
```

7. STL配接器
   将一个class的接口转换为另一个class的接口，使原本因接口不兼容而不能合作的classes可以一起运作。

- 应用于容器(stack,queue)
- 应用于迭代器(insert iterator, reverse iterator, iostream iterators)
- 应用于仿函数(很灵活，可以配接，配接，再配接，配接的操作包括bind，negate，compose，以及对一般函数或成员函数的修饰使其成为仿函数)，通过绑定、组合和修饰能力能够无限制地组合不同的表达式

```
f(x) = x*3; g(y) = y+2
compose1(bind2nd(multiplies<int>(),3),bind2nd(plus<int>(),2))
```

8. 线程安全问题
   在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。
   可访问的全局变量和堆数据随时可能被其他线程改变，因此多线程在并发的时候要保证数据的一致性
9. STL是线程安全的吗
   有些操作是线程安全的，有些操作是线程不安全的。
   线程安全：同一个容器，多个读者；或者是不同的容器，多个写者
   线程不安全：同一个容器(线程)，既有读者又有写者
10. 如何实现线程安全(无锁队列，读写锁) 手动控制多线程控制的方方面面 由程序员来控制，加入读写锁、临界区来做同步
    i. 每次调用容器的成员函数的时候要锁定
    ii. 每次返回容器迭代器（例如通过调用begin或end）的生命周期要锁定
    iii. 每个容器在调用算法的执行期需要锁定

```
vector<int> v;
...
getMutexFor(v);

vector<int>::iterator first5(find(v.begin(), v.end(), 5));

if (first5 != v.end()) {                        // 这里现在安全了

    *first5 = 0;                        // 这里也是
```

```
}
releaseMutexFor(v);
```

一个更面向对象的解决方案是创建一个Lock类，在它的构造函数里获得互斥量并在它的析构函数里释放它，这样使getMutexFor和releaseMutexFor的调用不匹配的机会减到最小。这样的一个类（其实是一个类模板）基本是这样的：

```
template<typename Container>                // 获取和释放容器的互斥量
class Lock {                          // 的类的模板核心；
public:                              // 忽略了很多细节
    Lock(const Containers container): c(container)
    {
        getMutexFor(c);// 在构造函数获取互斥量
    }
    ~Lock()
    {
        releaseMutexFor(c);// 在析构函数里释放它
    }
private:
    const Container& c;
};
```

使用一个类（像Lock）来管理资源的生存期（例如互斥量）的办法通常称为资源获得即初始化，你应该能在任何全面的C++教材里读到它。记住上述Lock是最基本的实现。一个工业强度的版本需要很多改进，但是那样的扩充与STL无关。而且这个最小化的Lock已经足够看出我们可以怎么把它用于我们一直考虑的例子：

```
vector<int> v;
...
{// 建立新块；
    Lock<vector<int> > lock(v);// 获取互斥量
    vector<int>::iterator first5(find(v.begin(), v.end(), 5));
    if (first5 != v.end()) {
        *first5 = 0;
    }
}                                 // 关闭块，自动
                                 // 释放互斥量
```

10. 栈和队列的区别
    底层都是deque实现，不属于容器的一种，都属于适配器 adapter(一种修饰容器或仿函数或迭代器接口的东西)
    栈：先进先出，操作上push(),pop(),top()
    队列：先进后出,操作上push(),pop(),front()
    相同点：线性结构，都是在表尾部插入，插入与删除的时间效率为O(1)，都可以通过线性结构和链式结构实现
11. list和vector的区别
    vector维护的是一段线性空间，动态扩容，可以随机访问，高效随机存取，插入与删除的时间复杂度是o(n)，查找的时间复杂度是O(1)，内存空间是连续的。当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝
    list底层是双向链表，不支持随机访问，插入与删除的时间复杂度是o(1)，查找的时间复杂度是O(n)，高效插入和删除，内存空间是不连续的
12. vector与deque的区别
    vector是单向开口的连续线性空间，deque是双向开口的连续线性空间，可以在头尾两端分别做插入和删除的操作。（vector也可以在头部插入啦，只是效率很差）
    差异1：deque允许于常数时间内对头端进行元素的插入和删除操作
    差异2：deque没有容量的概念，它是以分段的连续空间组合而成，随时可以增加一段新的空间并链接起来。(不会像vector那样，空间不足时需要申请一块更大的内存，然后复制过去，然后析构对象，然后释放内存)
    deque的实现用了中控器(map,每一个节点指向不同的缓冲区)、缓冲区(buffer)与迭代器(cur, first, last, node)
13. vector动态增长的过程
    a. 申请一块更大的内存空间以存储数据(两倍)
    b. 将数据从旧内存中拷贝到新内存空间
    c. 析构旧内存空间的对象
    d. 释放旧内存空间

智能指针的原理和实现

- 内存泄漏

  - 什么是内存泄漏？
    内存泄漏(memory leak)是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

  - 原因
    1：分配完内存之后忘了回收；
    2：某些API函数操作不正确，造成内存泄漏
    3：程序Code有问题，造成没有办法回收；

    ```
    Temp1 = new BYTE[100];
    Temp2 = new BYTE[100];
    Temp2 = Temp1;//Temp2的内存地址就丢掉了，而且永远都找不回了，这个时候Temp2的内存空间想回收都没有办法。
    ```

  - 如何查看是否发生了内存泄漏
    一个是程序当掉，一个是系统内存不足。 还有一种就是比较介于中间的结果程序不会当，但是系统的反映时间明显降低，需要定时的Reboot才会正常。
    有一个很简单的办法来检查一个程序是否有内存泄漏。就是是用Windows的任务管理器(Task Manager)。运行程序，然后在任务管理器里面查看 “内存使用”和”虚拟内存大小”两项，当程序请求了它所需要的内存之后，如果虚拟内存还是持续的增长的话，就说明了这个程序有内存泄漏问题。 当然如果内存泄漏的数目非常的小，用这种方法可能要过很长时间才能看的出来。
    ⑴ 使用工具软件BoundsChecker，BoundsChecker是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误。
    ⑵ 调试运行DEBUG版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境OUTPUT窗口)，综合分析内存泄漏的原因，排除内存泄漏。

- 智能指针

  - 为什么要用智能指针（管理内存防止内存泄漏）
    动态内存管理容易出现问题，要保证在正确的时间里面释放内存是很困难的，如果忘了释放内存，会导致内存泄漏；如果尚有指针引用内存的情况下就释放，会产生引用非法内存的指针
  - 智能指针的原理
    智能指针是一个类，这个类的构造函数中传入一个普通指针，析构函数中释放传入的指针。智能指针的类都是栈上的对象，所以当函数（或程序）结束时会自动被释放，
    智能指针的行为类似于常规指针，只是不需要手动释放指针，而是通过智能指针自己管理内存的释放，重要的区别在于他负责自动释放所指向的对象。使用智能指针就是为了更容易地使用动态内存。
  - 分类
    shared_ptr允许多个指针指向同一个对象
    unique_ptr独占某个对象,
    weak_ptr是一个弱引用，指向shared_ptr所管理的对象
  - 最常用的智能指针
    1）std::auto_ptr，有很多问题,已经被启用了。 不支持复制（拷贝构造函数）和赋值（operator =），但复制或赋值的时候不会提示出错。因为不能被复制，所以不能被放入容器中。
    2) C++11引入的unique_ptr， 也不支持复制和赋值，但比auto_ptr好，直接赋值会编译出错。实在想赋值的话，需要使用：std::move。
    例如：
    std::unique_ptr p1(new int(5));
    std::unique_ptr p2 = p1; // 编译会出错
    std::unique_ptr p3 = std::move(p1); // 转移所有权, 现在那块内存归p3所有, p1成为无效的指针.
    3) C++11或boost的shared_ptr，基于引用计数的智能指针。可随意赋值，直到内存的引用计数为0的时候这个内存会被释放。由析构函数来完成。
    4）C++11或boost的weak_ptr，弱引用。 引用计数有一个问题就是互相引用形成环，这样两个指针指向的内存都无法释

放。需要手动打破循环引用或使用weak_ptr。顾名思义，weak_ptr是一个弱引用，只引用，不计数。如果一块内存被shared_ptr和weak_ptr同时引用，当所有shared_ptr析构了之后，不管还有没有weak_ptr引用该内存，内存也会被释放。所以weak_ptr不保证它指向的内存一定是有效的，在使用之前需要检查weak_ptr是否为空指针。

- 智能指针用在多线程会有什么问题，效率相比不用智能指针会如何
  Shared_ptr可以让你通过多个指针来共享资源，这些指针自然可以用于多线程。有些人想当然地认为用一个shared_ptr来指向一个对象就一定是线程安全的，这是错误的。你仍然有责任使用一些同步原语来保证被shared_ptr管理的共享对象是线程安全的。
  建议– 如果你没有打算在多个线程之间来共享资源的话，那么就请使用unique_ptr。
- 智能指针的实现
  下面是一个基于引用计数的智能指针的实现，需要实现构造，析构，拷贝构造，=操作符重载，重载*-和>操作符。

```
template <typename T>
class SmartPointer {
public:
    //构造函数
    SmartPointer(T* p=0): _ptr(p), _reference_count(new size_t){
        if(p)
            *_reference_count = 1;
        else
            *_reference_count = 0;
    }
    //拷贝构造函数
    SmartPointer(const SmartPointer& src) {
        if(this!=&src) {
            _ptr = src._ptr;
            _reference_count = src._reference_count;
            (*_reference_count)++;
        }
    }
    //重载赋值操作符
    SmartPointer& operator=(const SmartPointer& src) {
        if(_ptr==src._ptr) {
            return *this;
        }
        releaseCount();
        _ptr = src._ptr;
        _reference_count = src._reference_count;
        (*_reference_count)++;
        return *this;
    }

   //重载操作符
   T&amp;amp; operator*() {
       if(ptr) {
           return *_ptr;
       }
       //throw exception
   }
   //重载操作符
   T* operator-&amp;gt;() {
       if(ptr) {
           return _ptr;
       }
       //throw exception
   }
   //析构函数
   ~SmartPointer() {
       if (--(*_reference_count) == 0) {
           delete _ptr;
           delete _reference_count;
       }
   }


private:
    T *_ptr;
        size_t *_reference_count;
        void releaseCount() {
        if(_ptr) {
            (*_reference_count)--;
                if((*_reference_count)==0) {
                    delete _ptr;
                    delete _reference_count;
```

```
                }
        }
        }
};

int main()
{
    SmartPointer<char> cp1(new char('a'));
    SmartPointer<char> cp2(cp1);
    SmartPointer<char> cp3;
    cp3 = cp2;
    cp3 = cp1;
    cp3 = cp3;
    SmartPointer<char> cp4(new char('b'));
    cp3 = cp4;
}
```

- 使用智能指针的十大建议
  错误#1：当唯一指针够用时却使用了共享指针
  建议 – 默认情况下，你应该使用unique_ptr。如果接下来有共享这个对象所有权的需求，你依然可以把它变成一个shared_ptr。
  错误#2：没有保证shared_ptr共享的资源/对象的线程安全性！
  建议– 如果你没有打算在多个线程之间来共享资源的话，那么就请使用unique_ptr。
  错误#3：使用auto_ptr！
  建议 – unique_ptr可以实现auto_ptr的所有功能。你应该搜索你的代码库，然后找到其中所有使用auto_ptr的地方，将其替换成unique_ptr。最后别忘了重新测试一下你的代码！
  错误#4：没有使用make_shared来初始化shared_ptr！
  建议- 使用make_shared而不是裸指针来初始化共享指针。
  错误#5：在创建一个对象（裸指针）时没有立即把它赋给shared_ptr。
  建议– 如果不使用make_shared创建shared_ptr，至少应该像下面这段代码一样创建使用智能指针管理的对象
  错误#6：删掉被shared_ptr使用的裸指针！
  建议 – 在你从共享指针中获取对应的裸指针之前请仔细考虑清楚。你永远不知道别人什么时候会调用delete来删除这个裸指针，到那个时候你的共享指针(shared_ptr)就会出现Access Violate（非法访问）的错误。
  错误#7：当使用一个shared_ptr指向指针数组时没有使用自定义的删除方法！
  建议 – 保证在使用shared_ptr管理一组对象时总是传递给它一个自定义的删除方法。下面这段代码就修复了这个问题：
  错误#8：在使用共享指针时使用循环引用！
  建议 – 在设计类的时候，当不需要资源的所有权，而且你不想指定这个对象的生命周期时，可以考虑使用weak_ptr代替shared_ptr。
  错误#9：没有删除通过unique_ptr.release()返回的裸指针！
  建议 – 无论何时，在对unique_ptr使用Release()方法后，记得一定要删除对应的裸指针。如果你是想要删掉unique_ptr指向的对象，可以使用unique_ptr.reset()方法。
  错误#10：在调用weak_ptr.lock()的时候没检查它的有效性！
  建议 – 一定要检查weak_ptr是否有效 — 其实就是在使用共享指针之前，检查lock()函数的返回值是否为空。