现代C++详解(98, 11, 14, 17)

## Part1：课程介绍

#### 课程的总体概述

**一：为什么要出这门课？**

我发现市面上的C++课程基本都停留在C++98阶段，而且基础概念反复重复。

讲的东西对于专业的C++开发不过是入门，还差的很远很远。这就给了很多新手我已经对C++有了基本使用能力的错觉，我已经入门了。

C++可以说是一个大杂烩，它包含c语言面向过程的全部知识，又包括面向对象部分，还包括模板与泛型编程。**很多新手在用c++的语法写c，这是非常不正确的，相当于抛弃了面向对象与模板，这些都是C++的精髓。面向对象与模板才是C++的核心，面向过程更多的是对C语言的一个妥协。**

要学好这三部分，就要把面向过程，面向对象，模板与泛型编程全部学好。

1. 学好面向过程部分，也就是c语言，需要从编译器的角度出发，每一行代码都要明白内存的变化。
2. 面向对象部分需要在一定程度上理解内存与编译器，但核心已经不再是内存层面了。
3. 模板与泛型不怎么需要了解底层，但也是一个非常复杂的模块。可**以说，模板是C++特有的精髓部分，C++几乎所有的库，框架都离不开模板。**

很多新手对于C++的理解太过单一，而且学的太浅了。

所以我出了这门课，讲的是企业级别的C++，会有一些难度，但这基本就是现代C++的全部知识点了。应该比市面上几乎所有的教程都全，而且很有深度的。

**二：课程内容介绍：企业中，C++11是必学的，后面的会了更好，不会也无所谓。**

这门课总共包含10个部分和三个附录，大致介绍一下：（重要的课程前面都加了\*）

1. Part1: 课程介绍：没什么实际知识点，就是一个介绍
2. Part2: C++的基本特性，属于C++的基础知识，难以被划分到任意一个模块
3. Part3 类的特性，C++类的全部常用特性应该就这些，比较全了
4. Part4：智能指针，不仅要学习基础知识，使用场景更是核心，很多人根本知道什么时候 该用智能指针。
5. Part5：模板与泛型编程。
   1. **模板很重要，很多人对模板的重视程度不够。**
   2. 模板也是泛型编程的基础，泛型编程是一个常用的编程方式，新手对泛型编程可能比较迷糊，觉得不好理解，但它确实功能强大，是C++的核心之一。
6. Part6：stl库：有了泛型编程的基础，就可以学习stl的实现原理。学习stl不仅要学习使 用方法，实现原理更加重要。可以说，对于stl，懂实现原理和只会使用完全是两个阶段。
7. Part7：io库：C++的io功能也很强大，很常用，必会

**这个有点像一个规范,他是对用户输入的一种规范;**

1. Part8：多线程：现代程序基本都是多线程程序，而C++自带的多线程又是现代C++程序实现多线程的主流方式，所以非常重要。
2. Part9：异常处理：能否合理的处理异常是衡量一个人水平的重要指标，对于大型程序非常重要。
3. Part10：各种难以归类但很重要的知识点。
4. 附录一：STL全部的算法
5. 附录二：STL标准库提供的仿函数
6. 附录三：STL各种容器的操作

综上所述：13个部分，除了Part1的课程介绍和三个附录几乎全程重点。所以做了个目录，方便随时查看。

**三：对新手讲一下我推荐的学习C++的路线**

1. C++语法学到差不多，就去学后面的东西，比如后端开发方向，就可以去学linux开发，网络开发，等有了应用场景，C++的语法就很好学了。比如，异常处理的很多语法单讲非常空洞，但放到具体环境中就很好理解了。我将异常处理放到最后也是因为只有有足够多的代码才可以体现出异常处理的意义。
2. 一些理论性的东西建议往后放一放，比如操作系统，网络通信的理论部分。对于想做开发的朋友，推荐先实践，后理论。而数据结构，放到最后，它和开发的关系不怎么紧密，**数据结构是体系成型的人用来提高的。**
3. C++一大特点就是没有固定的生态，其他语言，比如java都是有自己对应生态的，直接学对应生态即可，导致了C++的以下特点。
   1. C++和底层关系紧密，因为没有固定生态，难以大规模分装，而且C++涉及很多偏向底层的大型项目（比如windows，游戏开发这类特别要求效率的项目），无法牺牲运行效率来换取开发效率。**所以C++非常重视计算机网络和操作系统的理论知识。这也导致C++在很多环境下都是不可替代的。**
   2. 语言特性繁多，需要兼容各种使用场景，有很大的历史包袱。

**四：这门课就不做ppt了**

有图片就直接粘贴在文档中。ppt更多的是用来展示的，界面华丽，但信息密度太低了，对于这门课，ppt要做需要500页往上，不论我做起来，还是你们看起来，都费劲。

**五：直接开始**

其实时间是很宝贵的，工作的人对此体会应该更加深刻。所以我选择直接把文档写出来，而不是边讲边写文档，这样效率更高，节省时间。

#### （\*）开发环境介绍

使用的开发环境就是vs2019，当然vs2017，vs2022也可以，再早就不推荐了，可能有些功能不支持。

接下来介绍一下vs的基础知识，老手完全可以跳过这一节。

##### vs的介绍：

vs是一个功能极其强大的集成开发环境，说最强也没有问题，主要的缺点就是不能跨平台，只能在windows上运行。

而现在C++的主流开发平台是linux，所以很多人就说vs让开发者缺乏对C++执行过程的了解，linux上没有像vs那样强大的集成开发环境，比较分散，开发人员需要自己调度各种资源。

##### vs的安装与卸载：

* 1. vs安装：直接去microsoft官网下载就可以了，傻瓜版，我这里演示一下。注意只安装需要的模块就可以了，如果全安装要上百G。

**大致演示一下：**

如果没有vs，可以暂停，官网下载一下vs。

* 1. **（\*）vs的卸载**：这里特别注意，vs由微软官方推出，是**直接关系操作系统内核**的，如果乱弄是会玩坏操作系统的，最后就只能重装系统，所以强调一下如何卸载vs。

**演示一下：**

**其实卸载很简单，主要是不要乱弄，其它任何方式都可能会导致各种遗留问题，最后只能重装系统了。**

##### vs项目的创建与vs的程序编译过程

* 1. vs项目的创建，我用vs2019来演示，C++项目大致分为两种
     1. 控制台项目：程序在黑窗口运行
     2. windows桌面应用程序：程序有图形窗口

两种程序的区**别就是使用的链接库不同**，C++默认是不支持图形功能的，要实现图形功能，必须调用系统api，所以两种项目必须有两个链接库。

当然，两种库是可以随时替换的，不一定创建的控制台程序，就一定要当做控制台程序来编写。

我们这门课用的是**空项目**，能直接体现出C++的语法，没有其它因素的干扰。用的是控制台的链接库。

* 1. vs程序的编译过程。很多人说vs的编译过程不利于新人成长，这个说法其实有一定道理，这就和vs的项目目录结构有关了。
     1. vs的项目目录是一个解决方案下包括多个项目，vs是以文件组成的项目为最小编译单元的。也就是说vs无法单独编译一个项目中的一个cpp文件，会给新手一定的困扰。

这些在老家伙中理所当然，常常被忽略的东西，经常给萌新带来困惑。

##### vs的debug功能简介：

暂时我们只要会断点功能就可以了，需要知道的是，为了支持调试功能，编译和链 接必须添加很多额外的东西，所以使用debug功能，必须以调试的方式进行编译。

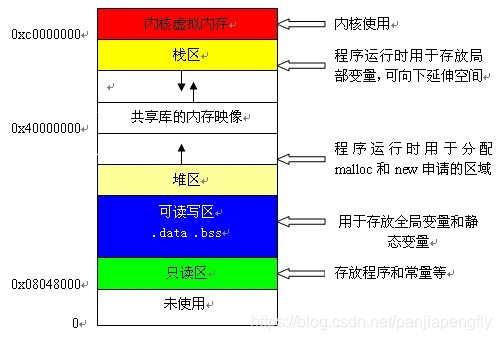
## Part2：C++的基本特性

#### （\*）程序的执行过程

程序被执行后就被称为一个进程，一个进程可以被划分为很多区域，这门课我们只需要理解以下的四个区就可以了。

1. **代码区与常量区**：进程按照代码区的代码执行，真正的常量也存储在这里，比如“abc”字符串，“1”，“88”等数字。这些是真正的常量。再看一下const关键字。const只不过是让编译器将变量视为常量罢了，和真正的常量有本质上的区别。
2. **栈区**：函数的执行所需的空间，注意，当函数执行完毕，函数对应的栈内存全部销毁。
3. **堆区**：进程用来灵活分配内存的地方，只有手动释放时才会销毁内存。
4. **静态变量区**：用来存储静态变量与全局变量的区域
   1. 静态变量：我们常常需要一些局部作用范围，生命周期却很长的变量
   2. 全局变量：重要性就不必说了，在c语言程序中经常用到，但在C++中不推荐使用，因为会**破坏封装性**。

具体的存储方式如图所示：



接下来用代码演示一下这几个区域。

堆区和栈区，是程序运行的主要地方。我们用一个最简单的程序来显示栈的用途。

**代码演示：**

至于堆区，主要意义在于灵活的生命周期，同样是刚才那个例子。

如果需要创建的对象有几十M，每次调用函数都需要创建一个这么大的对象，再复制到对应的容器中，那就太过耗费内存了。而且栈内存非常的小，通常不超过8M。

而使用堆内存，每调用一次函数就可以在堆内存中创建一个对象，容器中只要存储指针就可以了，极大的提高了程序效率。

**代码演示：**

而静态变量区：

有很多情况下，我们需要作用范围局限在函数之内，但生命周期却很长的变量，比如统计一个函数被调用的次数。

总结：栈区是函数执行的区域，堆区是函数内灵活分配内存的地方，二者缺一不可。

有很多人问为什么不能只在栈上运行程序，因为当函数运行结束时，栈是要销毁的，其上分配的内存全部失效。

也不能只在堆上运行程序，因为堆的唯一寻址方式就是指针，如果没有栈，根本无法使用堆。

提一下：栈区远远小于堆区，一般不超过8M，所以主要的内容都在对堆区上。堆区很大，虚拟内存剩下的都是堆区。

**注意：这节课对内存的划分比较粗糙，但新手理解到这里完全没有问题，这门课也够用了。**

程序执行的细节，还有剩下的区域是干什么的，我打算出一门叫做C++内存详解的课程，里面会详细讲解。包括pe文件头，内核区等。

#### （\*）new关键字及内存泄露

1. new关键字是C++用来动态分配内存的主要方式。

**代码演示：**

new可以直接分配单个变量的内存，也可以分配数组。

在分配单个对象的内存时。

当对象是普通变量时，可以分配对应的内存

当对象是类对象时，会调用构造函数，如果没有对应的构造函数，就会报错。

在分配数组对象内存时：

对于普通变量：可以使用“（）”将所有对象全部初始化为0。

对于类对象，有没有“（）”都一样，均使用默认构造函数，如果没有默认构造函数就会报错。初次之外,new的时候如果new数组,类一定要有默认构造(无参的),否则没办法申请，就比如**auto stringArr=new std::string[100](“HHAHA”); 是不行的!**

1. 内存泄露介绍：内存泄露是经常出现的常见bug。

**代码演示：**

内存泄露会导致堆内存的逐渐被占用，最终内存用完程序崩溃。常见的情况就是项目测试没问题，上线几天就炸了。然后就会非常麻烦，排查困难，损失很大。

**内存泄露是最严重的错误之一，程序不怕报错，就怕一开始运行的好好的，突然就出现了莫名其妙的错误。**

这句话也引出了后面的两个部分。

**Part4的智能指针**可以非常好的避免内存泄露的问题。

**Part9的异常处理**部分可以恰当的处理程序出现的异常，让程序有错误就立马处理，或直接终止进程，或忽略，不要让异常莫名其妙。这是程序设计的重要理念。

#### 命名空间

C++经常需要多个团队合作来完成大型项目。多个团队就常常出现起名重复的问题，C++就提供了命名空间来解决这个问题。

比如团队A和团队B都需要定义一个叫做Test的类。

**这里用代码简单演示：**

顺便提两点：  
 命名空间的实现原理，C++最后都要转化为C来执行程序。在namespace A中定义的Test类，其实全名是A::Test。

C++所有特有的库（指c没有的库）,都使用了std的命名空间。比如最常用的iostream。

using关键字设计的目的之一就是为了简化命名空间的。using关键字在命名空间方面主要有两种用法。

1. using 命名空间：：变量名。这样以后使用此变量时只要使用变量名就可以了。举个例子。
2. using namspce 命名空间。这样，每一个变量都会在该命名空间中寻找。举个例子。

**（\*）所以，头文件中一定不能使用using关键字。会导致命名空间的污染。**

**还是用刚才的代码演示。**

#### （\*）C++的标准输入输出简介

输入输出简单来说就是数据在输入设备，内存，硬盘，输出设备之间移动的过程。

c语言设定了很多不相关的函数还实现这些过程。

比如printf就是让数据从内存到显示屏（显示屏就是输出设备）。scanf就是让数据从键盘（键盘是输入设备）到内存。此外还有从内存到磁盘的文件操作函数。

c语言的函数虽然简单方便，但彼此之间没有关联。C++有了继承功能，可以让子类与父类之间有关联性，极大的提高各种输入输出功能之间的耦合性。

于是C++用继承功能重写了输入输出功能，这就是io库，io库引入了“流”的概念，数据从一个地方到另一个地方，原本地方的数据就没了，叫做流很贴切。

io库是一个很大的部分，但现阶段我们只要会使用输入输出流，cout和cin就可以了。

cout可以让数据从内存流到输出设备，cin可以让数据从输入设备流到内存。

**代码演示：**

**等到Part7，io库会详细讲解。**

#### 4.5. const关键字介绍

**首先说一下：这一课是后面加的，原先的目录上没有这一课，所以就叫做4.5课了。后面觉得讲一下const关键字还是很有意义的。很多人对const修饰的变量和真正的常量分不清。**

const关键字介绍：const是让编译期将变量视为常量，用const修饰的变量和真正的常量有本质的区别。

代码演示：

1. 真正的常量存储在常量区或代码区，比如“abcdefg”这个字符串就存储在常量区，而“3”，“100”这些数字就存储在代码区中，这些都是真正的常量，无法用任何方式修改。
2. const修饰的变量仍然存储在堆区或栈区中，从内存分布的角度讲，和普通变量没有区别。const修饰的变量并非不可更改的，C++本身就提供了mutable关键字（这个关键字在Part3就会讲的）用来修改const修饰的变量，从汇编的角度讲，const修饰的变量也是可以修改的。

#### （\*\*） auto关键字的使用

auto是C++11新加入的关键字，就是为了简化一些写法。

**代码演示：**

使用auto推断类型确实简单方便，但有个基本要求，就是在使用auto时清楚的知道编译器会给auto推断出什么类型。

为了学习auto的类型推断，我使用一个boost库来确定变量的具体类型。至于boost是什么，这里就不介绍了，大家可以去百度一下。

boost库的类型推断更加灵活方便，直观。而且Part10讲万能引用，完美转发时boost也是必须使用的，所以这里就提前使用boost了。其实这里采用vs默认提供的类型提示功能也可以。

首先下载，安装boost库，就直接视频演示了，不在文档中描述了。

这两个命令执行完毕，boost库也就编译好了。boost库很大，可以选择编译自己想要的模块，我就直接全部编译了。boost是很复杂的，不是几句话能说清楚，要深入理解可以去官网学习。

两个库，每一个编译都需要十几分钟，所以视频就暂停了。

这两个命令我放在一个boostCommand.txt的文件中，这个文件已经放在最终的文件包中。

auto有几个点需要注意：

**有些不好理解，可以多看几遍，或者带着问题学习下面的课程，Part2的所有知识都是反复用到的。**

1. auto只能推断出类型，引用不是类型，所以auto无法推断出引用，要使用引用只能自己加引用符号。

**代码演示：**

1. auto关键字在推断引用的类型时：会直接将引用替换为引用指向的对象。其实引用一直是这样的，引用不是对象，任何使用引用的地方都可以直接替换成引用指向的对象。

**代码演示：**

1. auto关键字在推断类型时，如果没有引用符号，会忽略值类型的const修饰，而保留修饰指向对象的const，典型的就是指针。**可能有些不好理解，看看代码就好说了。3和4的主要作用对象就是指针。**

**代码演示：**

const int\* const pi = &i;

auto pi2 = pi;

这个auto的来的pi2类型就是`const int\*`

1. auto关键字在推断类型时，如果有了引用符号，那么值类型的const和修饰指向对象的const都会保留。
2. const int\* const pi = &i;
4. auto pi2 = pi;
5. //pi = &i; ×
6. //pi2 = &i; √

这里科普下两个const 第一个const是指针指向内存的,代表指针指向的内容不能修改(指向对象的const),第二个const则修饰指针这个值不能修改(值类型的const);

**代码演示：**

**其实3，4为什么会出现这种情况，因为在传递值时，修改这个值并不会对原有的值造成影响。而传递引用时，修改这个值会直接对原有的值造成影响。**

确实不太好理解，尤其是基础不扎实的人。不懂了可以多问问我。

1. 当然，我们可以在前面加上const，这样永远都有const的含义。

**代码演示：**

1. auto不会影响编译速度，甚至会加快编译速度。因为编译器在处理XX a = b时，当XX是传统类型时，编译期需要检查b的类型是否可以转化为XX。当XX为auto时，编译期可以按照b的类型直接给定变量a的类型，所以效率相差不大，甚至反而还有提升。
2. （\*）最重要的一点，就是auto不要滥用，对于一些自己不明确的地方不要乱用auto，否则很可能出现事与愿违的结果，使用类型应该安全为先。
3. （\*）auto主要用在与模板相关的代码中，一些简单的变量使用模板常常导致可读性下降，经验不足还会导致安全性问题。

**注意：auto的用法这里大致了解就可以了，Part10会有一节详细的讲auto的类型推断的，这节课是没有涉及右值引用的，那里会将这一部分。而且现在没有讲模板，难以讲出auto关键字的主要用法。**

#### （\*）静态变量，指针和引用

变量的存储位置有三种，分别是静态变量区，栈区，堆区。

静态变量区在编译时就已经确定地址，存储全局变量与静态变量。

**代码演示：**

指针都是存储在栈上或堆上，不管在栈上还是堆上，都一定有一个地址。

本质上说，指针和普通变量没有区别。

在32位系统中，int变量和指针都是32位。指针必须和“&”，“\*”这两个符号一起使用才有意义。

&a代表的a这个变量的地址，a代表的a对应地址存储的值，\*a代表对应地址存储的值作为地址对应的值，这句话可能不好理解。

**代码演示：**

所以指针才可以灵活的操作内存，但这也带来了严重的副作用，比如指针加加减减就可以操作内存，所以引用被发明了，**引用就是作用阉割的指针（可以视为“类型\*const”，所以引用必须上来就赋初值，不能设置为空）**，编译器不将其视作对象，操作引用相当于操作引用指向的对象。**也就从根本是杜绝了引用篡改内存的能力。**

**新手如果不懂内存，就直接将引用视为指向对象的别名就可以了。**

**要真正理解指针，引用是需要学习c语言对应汇编的，只要懂了汇编，一目了然。不懂汇编就只能这样理解了。**

#### （\*\*）左值，右值，左值引用，右值引用

**首先说一点：在学这节课时不要去想左值，右值，左值引用，右值引用有什么意义。以后会反复使用的，这些概念都很重要。**

##### 左值和右值

左值右值从C++11开始就是一个很重要的概念了，但想要真正理解左值，右值不是一件容易的事。

尤其新人要彻底理解左值右值就更加困难了，所以我推荐新手将这些概念死死记住，带着疑惑学习下面的课程，积累的多了，自然就明白了。后面左值，右值的概念会被反复提及。

**C++任何一个对象要么是左值，要么是右值。**

**比如int i = 10，i和10都是对象**

**左值：拥有地址属性的对象就叫左值，**左值来源于c语言的说法，能放在“=”左面的就是左值，注意，左值也可以放在“=”右面。

**右值：不是左值的对象就是右值。或者说无法操作地址的对象就叫做右值。**一般来说，判断一个对象是否为右值，就看它是不是左值，有没有地址属性，不是左值，那就是右值。

**比如临时对象，就都是右值，临时对象的地址属性无法使用。**

**注意：左值也可以放在“=”右面，但右值绝对不可以放在等号左面**

**接下来就是大量举例了，说明那些是左值，哪些是右值。**

**代码演示。**

比如++i和i++;前者是左值,后者是右值;因为前者有地址属性;

##### 引用的分类

* 1. 普通左值引用：就是一个对象的别名，只能绑定左值，无法绑定常量对象。
  2. **const左值引用：可以对常量起别名，可以绑定左值和右值。**
  3. **右值引用就是int&& ref这种**（暂时不要去管右值引用有什么用，只要记住语法就可以了，实际用途下一课就会讲到）：只能绑定【右值的引用。
  4. 万能引用：这节课不讲，等到part10涉及模板时再讲，这是一个很重要，但需要模板等基础的概念。

**代码演示：**

#### （\*\*）move函数，临时对象

**首先说一点：这节课是第7课的继续，是对右值基础的补充，右值的具体应用要等到Part3的移动语义那里才能完全体现。**

##### move函数：

* 1. 右值看重对象的值而不考虑地址，move函数可以对一个左值使用，使操作系统不再在意其地址属性，将其完全视作一个右值。
  2. move函数让操作的对象失去了地址属性，**所以我们有义务保证以后不再使用该变量的地址属性，简单来说就是不再使用该变量，因为左值对象的地址是其使用时无法绕过的属性。**

1. int i = 0;
2. int&& rRef = *std*::*move*(i);
3. **就是以后都不要再用i了;!**

**代码演示：**

**move函数的具体意义现阶段无需在意，Part3讲移动语义时会体现move函数意义的。**

##### 临时对象：

**右值都是不体现地址的对象。那么，还有什么能比临时对象更加没有地址属性呢？右值引用主要负责处理的就是临时对象。**

程序执行时生成的中间对象就是临时对象，注意，所有的临时对象都是右值对象，因为临时对象产生后很快就可能被销毁，使用的是它的值属性。

**代码演示：**

##### 总结：

右值和右值引用这里只介绍语法，等到Part3的第11课，会学习移动构造，右值引用会真正体现出提高程序性效率的功能。

#### （\*\*）可调用对象

如果一个对象可以使用调用运算符“()”，()里面可以放参数，这个对象就是可调用对象。

（\*）**注意：可调用对象的概念新手只要记住就可以了，后面会反复用到，这个概念很重要。**

可调用对象的分类：

##### 函数：

函数自然可以调用()运算符，是最典型的可调用对象。

using pfType = void(\*)(int);

void test(pfType func) {

func(111);

}

int main() {

const auto& ref = 1;

auto func = pfType(11);

test(func);

}

##### 仿函数：

具有operator()函数的类对象（**知道有这么个东西就可以了，具体实现过程Part3会讲**），此时类对象可以当做函数使用，因此称为仿函数。

class Test {

public:

void operator()(int i) {

*std*::*cout* << "hhahha" << *std*::*endl*;

}

};

void test(pfType func) {

func(111);

}

int main() {

Test t;

t(1);

return 0;

}

就是类(实例化之后的对象 )可以用()直接调用

##### lambda表达式：

就是匿名函数，普通的函数在使用前需要找个地方将这个函数定义，于是C++提供了lambda表达式，需要函数时直接在需要的地方写一个lambda表达式，省去了定义函数的过程，增加开发效率。

**注意：lambda表达式很重要，现代C++程序中，lambda表达式是大量使用的。**

**lambda表达式的格式：**最少是“[] {}”，完整的格式为“[] () ->ret {}”。

代码演示：

lambda各个组件介绍

1. **[]代表捕获列表：表示lambda表达式可以访问前文的哪些变量。**
   1. []表示不捕获任何变量。
   2. [=]：表示按值捕获所有变量。
   3. [&]：表示按照引用捕获所有变量。

=，&也可以混合使用，比如

* 1. [=, &i]：表示变量i用引用传递，除i的所有变量用值传递。
  2. [&, i]：表示变量i用值传递，除i的所有变量用引用传递。

当然，也可以捕获单独的变量

* 1. [i]：表示以值传递的形式捕获i
  2. [&i]：表示以引用传递的方式捕获i

1. ()代表lambda表达式的参数，函数有参数，lambda自然也有 用于不是上下文的。
2. ->ret表示指定lambda的返回值，如果不指定，lambda表达式也会推断出一个返回值的。
3. {}就是函数体了，和普通函数的函数体功能完全相同。

l**ambda后面会广泛使用，现在只要理解基础就可以了。**

**Lmmbda最常见的用法就是给普通的函数当参数用!就是相当于可调用对象;比如下面的写法**

using pfType = void(\*)(int);

void test(pfType func) {

func(111);

}

int main() {

test([](int item) ->void {

*std*::*cout* << "lambda" << *std*::*endl*;

*std*::*cout* << item << *std*::*endl*;

return;

});

return 0;

}

可以直接在里面写函数;

C++的可调用对象主要就这三个，当然，这三个也可以衍生出很多写法。

最常见的就是函数指针，函数指针的本质就是利用指针调用函数，本质还是函数。

函数指针要细分也可以分为指向类成员函数的指针，指向普通函数的指针。

这些在这里就不演示了。

**此外,还有一个重要的点就是,lambda如果作为函数指针进行传递,一个很大的缺陷就是无法使用捕获列表**

int catchVar;

test([&catchVar](int item) ->void {

*std*::*cout* << catchVar << *std*::*endl*;

*std*::*cout* << "lambda" << *std*::*endl*;

*std*::*cout* << item << *std*::*endl*;

return;

});

这样就不行,可以使用std库的function;

using funcType = *std*::*function*<void(int)>;

void test(funcType func) {

func(111);

}

int main() {

int catchVar;

test([&catchVar](int item) ->void {

*std*::*cout* << catchVar << *std*::*endl*;

*std*::*cout* << "lambda" << *std*::*endl*;

*std*::*cout* << item << *std*::*endl*;

return;

});

return 0;

}

## Part3：类

**首先提一下：类的权限修饰就不讲了，相信大家对于这个已经很了解了，如果不了解，就百度一下吧，非常简单，易懂，直接问我也可以。**

#### （\*）类介绍，构造函数，析构函数

1. 类介绍：
   1. 面试的时候经常会听到一个问题，谈一下对面向对象和面向过程的理解。我来说一下我对这两个概念的理解
      1. 面向对象和面向过程是一个相对的概念。
      2. 面向过程是按照计算机的工作逻辑来编码的方式，最典型的面向过程的语言就是c语言了，c语言直接对应汇编，汇编又对应电路。
      3. 面向对象则是按照人类的思维来编码的一种方式，C++就完全支持面向对象功能，可以按照人类的思维来处理问题。
      4. 举个例子，要把大象装冰箱，按照人类的思路自然是分三步，打开冰箱，将大象装进去，关上冰箱。

要实现这三步，我们就要首先有人，冰箱这两个对象。人有给冰箱发指令的能 力，冰箱有能够接受指令并打开或关闭门的能力。

但是从计算机的角度讲，计算机只能定义一个叫做人和冰箱的结构体。人有手这个部位，冰箱有门这个部位。然后从天而降一个函数，是这个函数让手打开了冰箱，又是另一个函数让大象进去，再是另一个函数让冰箱门关上。

从开发者的角度讲，面向对象显然更利于程序设计。用面色过程的开发方式，程序一旦大了，各种从天而降的函数会非常繁琐，一些用纯c写的大型程序，实际上也是模拟了面向对象的方式。

那么，如何用面向过程的c语言模拟出面向对象的能力呢？类就诞生了，在类中可以定义专属于类的函数，让类有了自己的动作。回到那个例子，人的类有了让冰箱开门的能力，冰箱有了让人打开的能力，不再需要天降神秘力量了。

**总结：到现在，大家应该可以理解类的重要性了吧，这是面向对象的基石，也可以说是所有现代程序的基石。**

1. 构造函数：  
    类再怎么吹，它也是通过面向过程的机器实现的，类相当于定义了一个新类型，该类型生成在堆或栈上的对象时内存排布和c语言相同。但是c++规定，C++有在类对象创建时就在对应**内存将数据初始化的能力，这就是构造函数**。

**用excel表格演示一下：**

**构造函数**有以下类型。

1. **普通构造函数**：写法代码演示
2. **复制构造函数**：用另一个对象来初始化对象对应的内存，代码演示
3. **移动构造函数**：也是用另一个对象来初始化对象，具体内容会在Part3第13节详细讲解。
4. **默认构造函数**：当类没有任何构造函数时，编译期会为该类生成一个默认的的构造函数，在最普通的类中，默认构造函数什么都没做，对象对应的内存没有被初始化。

**代码演示。**

总结：构造函数就是C++提供的**必须有的**在对象创建时初始化对象的方法，（默认的什么都不做也是一种初始化的方式）

1. 析构函数：

析构函数介绍：当类对象被销毁时，就会调用析构函数。栈上对象的销毁时机就是函数栈销毁时，代码演示。堆上的对象销毁时机就是该堆内存被手动释放时，如果用new申请的这块堆内存，那调用delete销毁这块内存时就会调用析构函数。

**代码演示：**

总结，当类对象销毁时有一些我们必须手动操作的步骤时，析构函数就派上了用场。所以，几乎所有的类我们都要写构造函数，析构函数却未必需要。

#### （\*）this，常成员函数与常对象

1. this关键字：
   1. this是什么：
      1. 编译器将this解释为指向函数所作用的对象的指针，这句话新手有些不好理解，用代码演示一下就好说了。C++类的本质就是C语言的结构体外加几个类外的函数，C++最后都要转化为C语言来实现，类外的函数就是通过this来指向这个类的。

**代码演示：**

* + 1. 当然，这么说并非完全准确，this是一个关键字，只是我们将它当做指针理解罢了。

this有很多功能是单纯的指针无法满足的。比如每个类函数的参数根本没有名 叫this的指针。这不过是编译器赋予的功能罢了。

1. 常成员函数和常对象

**首先说一下：常成员函数和常对象很多人并不在意，确实，都写普通变量也可以。但是，我还是要提一点，在大型程序中，尽量加上const关键字可以减少很多不必要的错误。**

**这一点，开发过大型程序的人应该深有体会，没开发过大型程序的人也不必在意，记住多用const，这是一个很好的习惯。**

* 1. const关键字含义：普通的const在Part2的第4.5节就已经讲完了。所以这里说一下**常成员函数和常对象。**

常成员函数就是无法修改成员变量的函数。可以理解为将this指针指向对象用const修饰的函数。

常对象就是用const修饰的对象，定义好之后就再也不需要更改成员变量的值了。常对象在大型程序中还是很有意义的。

**代码演示：**

* 1. 常成员函数注意事项：

因为类的成员函数已经将this指针省略了，只能在函数后面加const关键字来实现无法修改类成员变量的功能了

* + 1. 注意：常函数无法调用了普通函数，否则常函数的这个“常”字还有什么意义。
    2. **成员函数能写作常成员函数就尽量写作常成员函数，可以减少出错几率。**
    3. 同名的常成员函数和普通成员函数是可以重载的，常量对象会优先调用常成员函数，普通对象会优先调用普通成员函数

**代码演示。**

* 1. 常对象注意事项：
     1. 常对象不能调用普通函数，这一点之前其实已经讲过了。
     2. 常函数在大型程序中真的很重要，很多时候我们都需要创建好就不再改变的对象。
  2. 总结：**再说一遍，毕竟重要的话说三遍吗！常成员函数和常对象要多用，这真的是一个非常好的习惯，写大项目可以少出很多bug，**

#### inline，mutable，default，delete

**这一节是Part3少有的不带“\*”的课程。**

**inline和mutable只要知道有这么个关键字就可以了。**

**default和delete关键字是需要掌握的，但是比较简单，也就放在这里了。**

1. inline关键字
   1. inline关键字的有什么作用：
      1. 在函数声明或定义中函数返回类型前加上关键字inline就可以把函数指定为内联函数。关键字inline必须与函数定义放在一起才能使函数成为内联，仅仅将inline放在函数声明前不起任何作用。
      2. 内联函数的作用，普通函数在调用时需要给函数分配栈空间以供函数执行，压栈等操作会影响成员运行效率，于是C++提供了内联函数将函数体放到需要调用函数的地方，用空间换效率。
   2. i**nline关键字的注意事项：inline关键字只是一个建议，开发者建议编译器将成员函数当做内联函数，一般适合搞内联的情况编译器都会采纳建议。**
   3. Inline关键字的总结。使用inline关键字就是一种提高效率，但加大编译后文件大小的方式，**现在随着硬件性能的提高，inline关键字用的越来越少了**。
2. mutable关键字
   1. mutable关键字的作用：
      1. Mutable意为可变的，与const相对，被mutable修饰的成员变量，永远处于可变的状态，即便处于一个常函数中，该变量也可以被更改。

**代码演示：**

**这个关键字在现代C++中使用情况并不多，一般来说只有在统计函数调用次数时才会用到。**

* 1. mutable关键字的注意事项
     1. mutable是一种万不得已的写法，一个程序不得不使用mutable关键字时，可以认为这部分程序是一个糟糕的设计。
     2. mutable不能修饰静态成员变量和常成员变量。
  2. 总结：mutable关键字是一种没有办法的办法，设计时应该尽量避免，只有在统计函数调用次数这类情况下才推荐使用。这个关键字也称不上是重点。

1. default关键字
   1. default关键字的作用：default关键字的作用很简单。
      1. 在编译时不会生成默认构造函数时便于书写。就是指定了类的构造函数,顺便弄一下构造函数;
      2. 也可以对**默认复制构造函数，默认的赋值运算符和默认的析构函数**使用，表示使用的是系统默认提供的函数，这样可以使代码更加明显。
      3. **现代C++中，哪怕没有构造函数，也推荐将构造函数用default关键字标记，可以让代码看起来更加直观，方便。**
2. TTest() = default;
3. TTest(const TTest&) = default;
4. ~TTest() = default;

**代码演示：**

**总结：default关键字还是推荐使用的，在现代C++代码中，如果需要使用一些默认的函数，推荐用default标记出来。**

1. delete关键字
   1. Delete关键字的作用：C++会为程序生成默认构造函数，默认复制构造函数，默认重载赋值运算符（重载部分会详细讲解）。

在很多情况下，我们并不希望这些默认的函数被生成，在C++11以前，**只能有将此 函数声明为私有函数或是将函数只声明不定义两种方式。**

C++11于是提供了delete关键字，只要在函数最后加上“=delete”就可以明确告诉 编译期不要默认生成该函数。

**代码演示：**

TTest() = delete;

TTest(const TTest&) = delete;

~TTest() = delete;

**总结：delete关键字还是推荐使用的，在现代C++代码中，如果不希望一些函数默认生成，就用delete表示，这个功能还是很有用的，比如在单例模式中，**

#### 友元类与友元函数

**抱歉：我在录视频时看错课数了，这应该是第4课的，所以这节课就又当第4课，又当第5课了，下一课还是第六课：**

1. 友元的介绍：友元就是可以让另一个类或函数访问私有成员的简单写法。

**代码演示：**

1. 注意：
   1. 友元会破坏封装性，一般不推荐使用，所带来的方便写几个接口函数就解决了。
   2. **(\*)某些运算符的重载必须用到友元的功能，这才是友元的真正用途，具体怎么重载下一课就会讲。比如位运算运算符**

oid FriendFunc() {

auto f = new Friend;

f->name = 1;

}

class Friend {

friend class TTest;

friend void FriendFunc();

private:

*std*::*string* name;

unsigned int old;

1. };
2. 总结：**友元平常并不推荐使用，新手不要再纠结友元的语法了，只要可以用友元写出必须用友元的重载运算符就可以了，重载运算符下一课就会讲。**

#### （\*\*）重载运算符

重载运算符在整个C++中拥有非常重要的地位，这一节非常重要。

1. 重载运算符的作用：
   1. 很多时候我们想让类对象也能像基础类型的对象一样进行作基础操作，比如“+”，“-”，“\*”，“\”，也可以使用某些运算符“=”，“()”，“[]”,“<<”，“>>”。但是一般的类即使编译器可以识别这些运算符，类对象也无法对这些运算符做出应对，我们必须对类对象定义处理这些运算符的方式。
   2. C++提供了定义这些行为的方式，就是“operator 运算符”来定义运算符的行为，operator是一个关键字，告诉编译器我要重载运算符了。
2. 注意：
   1. 我们只能重载C++已有的运算符，所有无法将“\*\*”这个运算符定义为指数的形式，因为C++根本没有“\*\*”这个运算符。
   2. C++重载运算符不能改变运算符的元数，“元数”这个概念就是指一个运算符对应的对象数量，比如“+”必须为“a + b”，也就是说“+”必须有两个对象，那么“+”就是二元运算符。比如“++”运算符，必须写为“a++”，也就是一元运算符。
3. 重载运算符举例

**以下全部用代码演示：**

* 1. 一元运算符重载
     1. “++”，“--”,
     2. “[]”
     3. “()”
     4. “<<”，“>>”
  2. 二元运算符重载
     1. “+”，“-”，“\*”，“/”
     2. “=”，
     3. “>”，“<”，“==”

1. class Test {
2. friend *std*::*ostream*& operator<<(*std*::*ostream*& os, const Test& test);
3. public:
4. void operator++() {
5. ++count;
6. }
7. void operator--() {
8. --count;
9. }
10. int operator[](unsigned index) {
11. return this->\_Vec[index];
12. }
13. private:
14. unsigned count = 0;
15. *std*::*vector*<int> \_Vec = { 1,2,3,4,5 };
16. };
17. *std*::*ostream*& operator<<(*std*::*ostream*& os, const Test& a) {
18. return os << a.count;
19. }

至于唯一的三元运算符“?:”，不能重载

* 1. 类类型转化运算符：“operator 类型”
  2. 特殊的运算符：new，delete，new[]，delete[]

**注意：“=”类会默认进行重载，如果不需要可以用“delete关键字进行修饰”。**

**总结：重载运算符非常重要，C++类中几乎都要定义各种各种的重载运算符。**

**可以重载成员函数,也可以重载全局函数;**

#### （\*）普通继承及其实现原理

**C++面向对象的三大特性：封装，继承，多态。分装就是类的权限管理，很简单，就不讲了。继承这节课讲，继承很重要，有些地方也是需要重点理解的。**

1. C++继承介绍：C++非继承的类相互是没有关联性的，假设现在需要设计医生，教师，公务员三个类，需要定义很多重复的内容而且相互没有关联，调用也没有规律。如果这还算好，那一个游戏有几千件物品，调用时也要写几千个函数。这太要命了。于是继承能力就应运而生了。

**代码演示：**

1. C++继承原理：C++的继承可以理解为在创建子类成员变量之前先创建父类的成员变量，实际上，C语言就是这么模仿出继承功能的。

**用excel表格演示一下：**

要注意的是继承的类基类

class gun {

protected:/\*private\*/

*std*::*string* name;

*std*::*string* icon;

};

class iceGun :public gun {

iceGun() {

*std*::*cout* << name;

}

};

1. C++继承的注意事项。
   1. **C++子类对象的构造过程。先调用父类的构造函数，再调用子类的构造函数，也就是说先初始化父类的成员，再初始化子类的成员。**
   2. 若父类没有默认的构造函数，子类的构造函数又未调用父类的构造函数，则无法编译。
   3. C++子类对象的析构过程。先调用父类的析构函数，再调用子类的析构函数。

**演示一下：**

**把构造函数理解为建房子就行**

总结：面向对象三大特性的继承就这么简单，很多人觉得类继承很复杂，其实完全不是这样的，只要明白子类在内存上其实就相当于把父类的成员变量放在子类的成员变量前面罢了。构造和析构过程也是为了这个机制而设计的。

#### （\*\*）虚函数及其实现原理，override关键字

1. 虚函数介绍：
   1. 虚函数就是面向对象的第三大特点：多态。多态非常的重要，它完美解决了上一课设计游戏装备类的问题，我们可以只设计一个函数，函数参数是基类指针，就可以调用子类的功能。比如射击游戏，所有的枪都继承自一个枪的基类，人类只要有一个开枪的函数就可以实现所有枪打出不同的子弹。
   2. 父类指针可以指向子类对象，这个是自然而然的，因为子类对象的内存前面就是父类成员，类型完全匹配。（**不要死记硬背，尽量理解原理**）
   3. 当父类指针指向子类对象，且子类重写父类某一函数时。父类指针调用该函数，就会产生以下的可能
      1. **该函数为虚函数：父类指针调用的是子类的成员函数。**
      2. **该函数不是虚函数：父类指针调用的是父类的成员函数。**

**代码演示**：

1. 虚函数的注意事项：
   1. 子父类的虚函数必须完全相同，为了防止开发人员一不小心将函数写错，于是C++11添加了override关键字。

**代码演示：**

* 1. **(\*) 父类的析构函数必须为虚函数：这一点很重要，当父类对象指向子类对象时，容易使独属于子类的内存泄露。会造成内存泄露的严重问题。**

**代码演示**

**这里说的情况是设计多态然后delete的时候!**

void onFire(const gun\* \_gun) {

\_gun->fire();

delete \_gun;

}

int main() {

onFire(new iceGun);

//delete icegun;

return 0;

}

1. overide关键字的作用：前面已经说过了，为了防止开发人员将函数名写错了，加入了override关键字。
2. **虚函数实现多态的原理介绍**
   1. 动态绑定和静态绑定：
      1. 静态绑定：程序在编译时就已经确定了函数的地址，比如非虚函数就是静态绑定。
      2. 动态绑定：程序在编译时确定的是程序寻找函数地址的方法，只有在程序运行时才可以真正确定程序的地址，比如虚函数就是动态绑定。
   2. 虚函数是如何实现动态绑定的呢？
      1. 每个有虚函数的类都会有一个虚函数表，对象其实就是指向虚函数表的指针，编译时编译器只告诉了程序会在运行时查找虚函数表的对应函数。每个类都会有自己的虚函数表，所以当父类指针引用的是子类虚函数表时，自然调用的就是子类的函数。

**代码演示：**

1. 总结：虚函数是C++类的重要特性之一，很简单，但使用频率非常高，至于如何实现的也要掌握。

#### 静态成员变量与静态函数

1. 静态成员变量：
   1. Part2的第六节课就讲过**C语言的静态成员变量，在编译期就已经在静态变量区明确了地址，所以生命周期为程序从开始运行到结束，作用范围为与普通的成员变量相同。这些对于类的静态成员变量同样适用。**

**代码演示：**

* 1. 类的静态成员变量因为创建在静态变量区，所以直接属于类，也就是我们可以直接通过类名来调用，当然通过对象调用也可以。

**代码演示：**

class StaticVar {

public:

static unsigned int a;

};

unsigned int StaticVar::a = 55;

1. 静态成员变量的注意项：
   1. 静态成员变量必须在**类外进行初始化**，否则会报未定义的错误，不能用构造函数进行初始化。因为静态成员变量在静态变量区，只有一份，而且静态成员变量在编译期就要被创建，成员函数那都是运行期的事情了
2. **静态成员函数的特点：静态成员函数就是为静态成员变量设计的，就是为了维持封装性**。
3. class StaticVar {
4. public:
5. static unsigned int get() {
6. return a;
7. };
8. private:
9. static unsigned int a;
10. };
11. unsigned int StaticVar::a = 55;

**代码演示：**

#### （\*）纯虚函数

1. 纯虚函数介绍：
   1. 还是那个枪械射击的例子，基础的枪类有对应的对象吗？没有。它唯一的作用就是被子类继承。
   2. 基类的openfire函数实现过程有意义吗？没有。它就是用来被重写的。
   3. 所以纯虚函数的语法诞生了，只要将一个虚函数写为纯虚函数，那么该类将被认为无实际意义的类，无法产生对象。纯虚函数也不用去写实际部分。写了编译期也会自动忽略。

**代码演示：**

1. 纯虚函数的注意事项：
   1. 没什么注意事项，这个语法非常简单。
2. 总结：纯虚函数的特点就是语法简单，却经常使用，必会。

#### RTTI：

**RTTI使用频率不是很高，但仍然有一定的意义，应当掌握。**

1. RTTI介绍：
   1. RTTI（Run Time Type Identification）即通过运行时类型识别，程序能够通过基类的指针或引用来检查这些指针或引用所指向的对象的实际派生类。
   2. C++为了支持多态，C++的指针或引用的类型可能与它实际指向对象的类型不相同，这时就需要rtti去判断类的实际类型了，**rtti是C++判断指针或引用实际类型的唯一方式。**
2. RTTI的使用场景：**可能有很多人会疑惑RTTI的作用，所以单独拿出来说一下。**
   1. 异常处理：这是RTTI最主要的使用场景，具体作用在异常处理章节会详细讲解。
   2. IO操作：具体作用等到IO章节会详细讲解。
3. RTTI的使用方式：RTTI的使用过程就两个函数
   1. typeid函数：typeid函数返回的一个叫做type\_info的结构体，该结构体包括了所指向对象的实际信息，其中name()函数就可以返回函数的真实名称。type\_info结构体其他函数没什么用.

代码演示：

gun\* p = new iceGun;

*std*::*cout* << typeid(\*p).name() << *std*::*endl*;

用法很简单,只需要解引用以下就行;

* 1. dynamic\_cast函数：C++提供的将父类指针转化为子类指针的函数。

1. gun\* p = new iceGun;
2. *std*::*cout* << typeid(\*p).name() << *std*::*endl*;
4. if (*std*::*string*(typeid(\*p).name()) == "iceGun") {
5. auto pT = dynamic\_cast<iceGun\*>(p);
6. *std*::*cout* << typeid(pT).name() << *std*::*endl*;
7. }

代码演示：

1. RTTI的注意事项：
   1. **当使用typeid函数时，父类和子类必须有虚函数（父类有了虚函数，子类自然会有虚函数），否则类型判断会出错。**
2. RTTI总结：就是C++在运行阶段判断对象实际类型的唯一方式。

#### 多继承

**首先提一下：多继承了解一下就可以了。**

1. 多继承的概念
   1. 就是一个类同时继承多个类，在内存上，该类对象前面依次为第一个继承的类，第二个继承的类，依次类推。

**代码演示：**

1. 多继承的注意点：
   1. 多继承最需要注意的点就是重复继承的问题，这个问题下一个将会详细讲解。
   2. 多继承会使整个程序的设计更加复杂，平常不推荐使用。C++语言中用到多继承的地方主要就是借口模式。相较于C++，java直接取消了多继承的功能，添加了借口。
2. 多继承的总结：多**继承这个语法虽然在某些情况下使代码写起来更加简洁，但会使程序更加复杂难懂，一般来说除了借口模式不推荐使用。**

#### 虚继承及其实现原理

1. 虚继承的概念：虚继承就是为了避免多重继承时产生的二义性问题。虚继承的问题用语言不好描述，但用代码非常简单，所以直接写代码了。

**代码演示：**

1. 虚继承的实现原理介绍：
   1. 使用了虚继承的类会有一个虚继承表，表中存放了父类所有成员变量相对于类的偏移地址。
   2. 按照刚才的代码，B1，B2类同时有一个虚继承表，当C类同时继承B1和B2类时，每继承一个就会用虚继承表进行比对，发现该变量在虚继承表中偏移地址相同，就只会继承一份。
2. 虚继承的注意点：没什么需要注意的，语法简单。
3. 虚继承的总结：这个语法就是典型的语法简单，但在游戏开发领域经常使用的语法，其它领域使用频率会低很多。

#### （\*\*）移动构造函数与移动赋值运算符

1. 对象移动的概念：
   1. 对一个体积比较大的类进行大量的拷贝操作是非常消耗性能的，因此C++11中加入了“对象移动”的操作
   2. 所谓的对象移动，其实就是把该对象占据的**内存空间的访问权限转移给另一个对象。**比如一块内存原本属于A，在进行“移动语义”后，这块内存就属于B了。
2. 移动语义为什么可以提高程序运行效率。因为我们的各种操作经常会进行大量的“复制构造”，“赋值运算”操作。这两个操作非常耗费时间。移动构造是直接转移权限，这是不是就快多了。

**注意：在进行转移操作后，被转移的对象就不能继续使用了，所以对象移动一般都是对临时对象进行操作（因为临时对象很快就要销毁了）。**

**只是一个概念;**

**代码演示：**

注意这里的右值引用不能是const的，因为你用右值引用函数参数就算为了让其绑定到一个右值上去的！就是说这个右值引用是一定要变的，但是你一旦加了const就没法改变该右值引用了。

1. **默认移动构造函数和默认移动赋值运算符**

会默认生成移动构造函数和移动赋值运算符的条件：

**只有一个类没有定义任何自己版本的拷贝操作（拷贝构造，拷贝赋值运算符），且类的每个非静态成员都可以移动，系统才能为我们合成。**

**可以移动的意思就是可以就行移动构造，移动赋值。所有的基础类型都是可以移动的，有移动语义的类也是可以移动的。**

class movClass {

movClass(movClass&& rRef) {

//XXX

}

movClass& operator=(movClass&& rRef) {

//移动拷贝

this->str = rRef.str;

rRef.str = 0;

return \*this;

}

movClass& operator=(const movClass& ref) {

if (&ref == this) {

return \*this;

}

*strcpy\_s*(this->str,100,ref.str);

}

private:

char\* str = nullptr;

};

**这就是移动拷贝,就靠右值引用来写拷贝函数,右值引用决定了了传入的对象一定是可移动临时的!**

## Part4：智能指针

#### （\*）智能指针概述

1. 为什么要有智能指针：在Part2的第二节课已经讲过，直接使用new和delete运算符极其容易导致内存泄露，而且非常难以避免。于是人们发明了智能指针这种可以自动回收内存的工具。
2. 智能指针一共就三种：普通的指针可以单独一个指针占用一块内存，也可以多个指针共享一块内存。
   1. 共享型智能指针：shared\_ptr，同一块堆内存可以被多个shared\_ptr共享。
   2. 独享型智能指针：unique\_ptr，同一块堆内存只能被一个unique\_ptr拥有。
   3. 弱引用智能指针：weak\_ptr，也是一种共享型智能指针，可以视为对共享型智能指针的一种补充
3. **（\*）智能指针注意事项：**

**智能指针和裸指针不要混用，接下来的几节课会反复强调这一点。（这一点太重要了，所以上来就提了）**

如果一个对象的生命周期完全由一个作用域管理，那么在这个作用域内直接创建对象可能是更好的选择。

#### （\*）shared\_ptr

1. **shared\_ptr的工作原理**
   1. 我们在动态分配内存时，堆上的内存必须通过栈上的内存来寻址。也就是说栈上的指针（堆上的指针也可以指向堆内存，但终究是要通过栈来寻址的）是寻找堆内存的唯一方式。
   2. 所以我们可以给堆内存添加一个引用计数，有几个指针指向它，它的引用计数就是几，当引用计数为0是，操作系统会自动释放这块堆内存。
2. **Shared\_ptr的常用操作**
   1. **shared\_ptr的初始化**
      1. **使用new运算符初始化**

**代码演示：**

一般来说不推荐使用new进行初始化，因为C++标准提供了专门创建shared\_ptr的函数“make\_shared”，该函数是经过优化的，效率更高。

* + 1. **使用make\_shared函数进行初始化：**

**代码演示：**

**注意：千万不要用裸指针初始化shared\_ptr，容易出现内存泄露的问题。**

* + 1. **当然使用复制构造函数初始化也是没有问题的。**

代码演示:

* 1. **shared\_ptr的引用计数：**

智能指针就是通过引用计数来判断释放堆内存时机的。

use\_count()函数可以得到shared\_ptr对象的引用计数。

**代码演示：**

1. **智能指针可以像普通指针那样使用，”share\_ptr”早已对各种操作进行了重载，就当它是普通指针就可以了。**

**代码演示：**

1. **Shared\_ptr的常用函数**
   1. unique函数：判断该shared\_ptr对象是否独占若独占，返回true。否则返回false。

代码演示：

* 1. reset函数：
     1. 当reset函数有参数时，改变此shared\_ptr对象指向的内存。
     2. 当reset函数无参数时，将此shared\_ptr对象置空，也就是将对象内存的指针设置为nullptr。

代码演示：

* 1. **get函数，强烈不推荐使用：**

代码演示：

**如果一定要使用，那么一定不能delete返回的指针。**

* 1. swap函数：交换两个智能指针所指向的内存
     1. std命名空间中全局的swap函数
     2. shared\_ptr类提供的swap函数

1. **关于智能指针创建数组的问题。**

可能就是这个比较有用;

**代码演示：**

1. **用智能指针作为参数传递时直接值传递就可以了。**shared\_ptr的大小为固定的8或16字节（也就是两倍指针的的大小，32位系统指针为4个字节，64位系统指针为8个字节，shared\_ptr中就两个指针），所以直接值传递就可以了。

代码演示：

1. **shared\_ptr总结：**在现代程序中，当想要共享一块堆内存时，优先使用shared\_ptr，可以极大的减少内存泄露的问题。

#### （\*）weak\_ptr

1. **weak\_ptr介绍：**
   1. 这个智能指针是在C++11的时候引入的标准库，它的出现完全是为了弥补shared\_ptr天生有缺陷的问题，其实shared\_ptr可以说近乎完美。
   2. 只是通过引用计数实现的方式也引来了引用成环的问题，这种问题靠它自己是没办法解决的，所以在C++11的时候将shared\_ptr和weak\_ptr一起引入了标准库，用来解决循环引用的问题。
2. **shared\_ptr的循环引用问题：**

class B;

class A {

public:

*std*::*shared\_ptr*<B> sharedB;

};

class B {

public:

*std*::*shared\_ptr*<A> sharedA;

};

int main() {

auto sharedA = *std*::*make\_shared*<A>();

auto sharedB = *std*::*make\_shared*<B>();

sharedA->sharedB = sharedB;

sharedB->sharedA = sharedA;

}  
上面的代码就是会出现内存泄露;

本质上就是因为引用计数是2,结束 A和B的指针都是1,不会被销毁;

1. **weak\_ptr的作用原理：**weak\_ptr的对象需要绑定到shared\_ptr对象上，作用原理是weak\_ptr不会改变shared\_ptr对象的引用计数。**只要shared\_ptr对象的引用计数为0，就会释放内存，weak\_ptr的对象不会影响释放内存的过程**。

重新回到刚才的代码：

1. weak\_ptr的总结：**weak\_ptr使用较少，就是为了处理shared\_ptr循环引用问题而设计的。**

#### （\*）unique\_ptr

1. **uniqe\_ptr介绍：**独占式智能指针，在使用智能指针时，我们一般优先考虑独占式智能指针，因为消耗更小。如果发现内存需要共享，那么再去使用“shared\_ptr”。
2. **unique\_ptr的初始化**：和shared\_ptr完全类似
   1. 使用new运算符进行初始化

代码演示：

* 1. 使用make\_unique函数进行初始化

代码演示：

1. **unique\_ptr的常用操作**
   1. unque\_ptr禁止复制构造函数，也禁止赋值运算符的重载。否则独占便毫无意义。、

**代码演示：**

* 1. unqiue\_ptr允许移动构造，移动赋值。移动语义代表之前的对象已经失去了意义，移动操作自然不影响独占的特性。

**代码演示：**

* 1. reset函数：
     1. 不带参数的情况下：释放智能指针的对象，并将智能指针置空。
     2. 带参数的情况下：释放智能指针的对象，并将智能指针指向新的对象。

**代码演示：**

1. **将unque\_ptr的对象转化为shared\_ptr对象，**当unique\_ptr的对象为一个右值时，就可以将该对象转化为shared\_ptr的对象。

**这个使用的并不多，需要将独占式指针转化为共享式指针常常是因为先前设计失误。**

**注意：shared\_ptr对象无法转化为unique\_ptr对象。**

**代码演示：**

#### （\*\*）智能指针的使用范围

这节课一共就几句话，但仍然是两个（\*），足以说明如何使用智能指针的重要性。

1. **能使用智能指针就尽量使用智能指针，那么哪些情况属于不能使用智能指针的情况 呢？**

有些函数必须使用C语言的指针，这些函数又没有替代，这种情况下，才使用普通的指针，其它情况一律使用智能指针。

必须使用C语言指针的情况包括：

1. **网络传输函数**，比如windows下的send，recv函数，只能使用c语言指针，无法替代.
2. **c语言的文件操作部分**。这方面C++已经有了替代品，C++的文件操作完全支持智能指针，**所以在做大型项目时，推荐使用C++的文件操作功能（Part7会详细讲解）。**

**除了以上两种情况，剩下的均推荐使用智能指针。**

1. **我们应该使用哪个智能指针呢？**
2. **优先使用unique\_ptr，内存需要共享时再使用shared\_ptr。**
3. **当使用shared\_ptr时，如果出现了循环引用的情况，再去考虑使用weak\_ptr。**
4. **总结：智能指针部分就这样了，东西真的不多，但都非常重要，很常用的。**

**下面是uniquePtr常见的用法,放到全局变量中**

1. int\* gPtr = nullptr;
2. void init() {
3. auto p= *std*::*make\_unique*<int>(1);
4. gPtr = p.release();
5. }
6. void release() {
7. *std*::*unique\_ptr*<int> uP;
8. uP.*reset*(gPtr);
9. }

## Part5：模板与泛型编程

#### （\*）模板介绍，类模板与模板实现原理

1. **模板的重要性**：**模板是C++最重要的模块之一，很多人对模板的重视不够，这一章一定要好好学，所有课时都是重点。**

**C++的三大模块，面向过程，面向对象，模板与泛型。面向过程就是C语言，面向对象就是类，现在轮到模板与泛型了。**

1. **模板的介绍：**
   1. 模板能够实现一些其他语法难以实现的功能，但是理解起来会更加困难，容易导致新手摸不着头脑。
   2. 模板分为类模板和函数模板，函数模板又分为普通函数模板和成员函数模板。
2. **类模板基础：**

这节课讲一下类模板，函数模板下一课再讲

* 1. 类模板的写法与使用十分固定

**代码演示：注意，这段代码非常有代表性，在下一课补完后，一定要掌握，多看几遍。**

#pragma once

template<typename T>

class MyArray {

using iterator = T\*;

using const\_iterator = const T\*;

public:

//MyArray(size\_t count) {

// if (count) {

// \_Data = new T[count];

// }

// else \_Data = nullptr;

//}

//在外边定义

MyArray(*size\_t* count);

~MyArray() {

if (\_Data) {

delete[] \_Data;

}

}

//这个是在里面写的

iterator begin()const;

//这个代表就是返回是const p；

const\_iterator cbegin()const {

return \_Data;

}

private:

T\* \_Data;

};

//最多只能这样写 不能cpp和h文件分开

template<typename T>

MyArray<T>::MyArray(*size\_t* count) {

if (count) {

\_Data = new T[count];

}

else \_Data = nullptr;

}

template<typename T>

typename MyArray<T>::iterator MyArray<T>::begin()const {

return \_Data;

}  
**可以看到,类模板一定是不能hpp和cpp分开实现的**

**还用一个重要的点**template<typename T>

typename MyArray<T>::iterator MyArray<T>::begin()const {

return \_Data;

}

**如果想使用类模板里面定义好的类型,必须加上typename MyArray<T>::iterator**

1. **模板的实现原理：**

模板需要编译两次，在第一次编译时仅仅检查最基本的语法，比如括号是否匹配。等函数真正被调用时，才会真正生成需要的类或函数。

所以这直接导致了一个结果，就是不论是模板类还是模板函数，声明与实现都必须放在同一个文件中。因为在程序在编译期就必须知道函数的具体实现过程。如果实现和声明分文件编写，需要在链接时才可以看到函数的具体实现过程，这当然会报错。

于是人们发明了.hpp文件来存放模板这种声明与实现在同一文件的情况。

#### （\*）initializer\_list与typename

1. **initializer\_list的用法**
   1. initializer\_list介绍：initializer\_list其实就是初始化列表，我们可以用初始化列表初始化各种容器，比如“vector”，“数组”。

代码演示：

*std*::*initializer\_list*<int> iList{1, 2, 3, 4}; 可以

*std*::*vector*<int> test = { 1,2,3,4 };可以

MyArray a{ 1,2,3,4 };不行

原因是我们没有initializer\_list,所以在构造函数中加入initializer\_list就行

* 1. 这节课的主要任务是在上一课的代码中加入initializer\_list。

**代码演示：**

#pragma once

#include <memory>

#include <type\_traits>//模板萃取

template <typename T>

struct get\_type {

using type = T;

};

template <typename T>

//模板特化

struct get\_type<T\*> {

using type = T;

};

template<typename T>

class MyArray {

using iterator = T\*;

using const\_iterator = const T\*;

public:

//MyArray(size\_t count) {

// if (count) {

// \_Data = new T[count];

// }

// else \_Data = nullptr;

//}

//在外边定义

MyArray(*size\_t* count);

//左值版本

MyArray(*std*::*initializer\_list*<T>& list);

//右值

MyArray(*std*::*initializer\_list*<T>&& list);

~MyArray() {

if (\_Data) {

delete[] \_Data;

}

}

//这个是在里面写的

iterator begin()const;

iterator end() const;

//这个代表就是返回是const p；

const\_iterator cbegin()const {

return \_Data;

}

T& operator[](unsigned index) const { return \_Data[index]; }

private:

T\* \_Data;

unsigned \_Count=0;

};

template<typename T>

typename MyArray<T>::iterator MyArray<T>::end()const {

return \_Data + \_Count;

}

//最多只能这样写 不能cpp和h文件分开

template<typename T>

MyArray<T>::MyArray(*size\_t* count) {

if (count) {

//智能指针不好分配arry

\_Data = new T[count]();

\_Count = count;

}

else \_Data = nullptr;

}

template<typename T>

typename MyArray<T>::iterator MyArray<T>::begin()const {

return \_Data;

}

template<typename T>

MyArray<T>::MyArray(*std*::*initializer\_list*<T>& list) {

if (list.*size*()) {

\_Data = new T[list.*size*()]();

unsigned count = 0;

if (*std*::*is\_pointer*<T>::*value*) {

//这里是指针

for (auto elem : list) {

//这里用类模板特化萃取出T的类型

//不能直接\*T \*后面

\_Data[count++] = new typename get\_type<T>::type(\*elem);

}

}

else {

for (const auto& elem : list) {

//这里有可能是指针 导致浅复制

//解决这个问题 1.萃取 2.模板特化

\_Data[count++] = elem;

}

}

\_Count = list.*size*();

}

else \_Data = nullptr;

}

//对于右值引用 不用考虑深浅拷贝问题

template<typename T>

MyArray<T>::MyArray(*std*::*initializer\_list*<T>&& list) {

if (list.*size*()) {

unsigned count = 0;

\_Data = new T[list.*size*()];

for (auto elem : list) {

\_Data[count++] = elem;

}

\_Count = list.*size*();

}

else \_Data = nullptr;

}

**此外,需要注意的是深浅拷贝问题(如果是指针为T\*,要写个模板特化萃取出T的类型)**

1. typename的用法
   1. 在定义模板时表示这个一个待定的类型

**代码演示：**

* 1. 在类外表明自定义类型时使用 比如typename MyArray<T>::iterator

**代码演示：**

**在C++的早期版本，为了减少关键字数量，用class来表示模板的参数，但是后来因为第二个原因，不得不引入typename关键字。**

#### （\*）函数模板，成员函数模板

1. 普通函数模板的写法与类模板类似

**代码演示：**

**在现代C++中，函数模板一直普遍使用，一定要掌握。**

比如我们写一个`for\_each`

namespace mystd {

template<typename iter\_type,typename func\_type>

void for\_each(iter\_type first, iter\_type last, func\_type func) {

for (auto iter = first;iter!=last;++iter) {

func(\*iter);

}

}

}

*std*::*vector*<int> ivec = { 0, 2030, 403210, 2041 };

mystd::for\_each(ivec.*begin*(), ivec.*end*(), [](int& elem) {

*std*::*cout* << elem << *std*::*endl*;

});

加上lambda就可以遍历每个成员了;

这里没有写函数模板的两个成员,主要是因为他自动给填上了

1. 成员函数模板

**代码演示：**

成员函数模板使用情况也不少，需要掌握的

template<typename T>

class MyVector {

public:

template<typename T2>

void outPut(const T2& elem);

};

template<typename T>

template<typename T2>

void MyVector<T>::outPut (const T2& elem) {

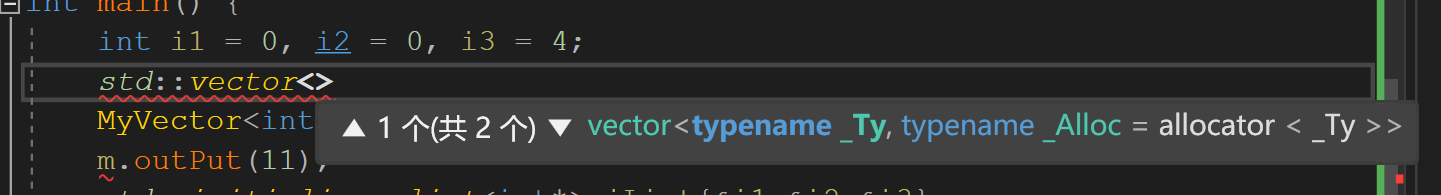
*std*::*cout* << elem << *std*::*endl*;

}

#### （\*）默认模板参数

默认模板参数：

1. 默认模板参数是一个经常使用的特性，比如在定义vector对象时，我们就可以使用 默认分配器。



1. 模板参数就和普通函数的默认参数一样，一旦一个参数有了默认参数，它之后的参 数都必须有默认参数
   1. **函数模板使用默认模板参数**

**代码演示：**

namespace mystd {

using func\_type = *std*::*function*<void(int)>;

template<typename iter\_type,typename func\_type=func\_type>

void for\_each(iter\_type first, iter\_type last, func\_type func = [](int& elem) {

*std*::*cout* << "haha! default!"<<*std*::*endl*;

}) {

for (auto iter = first;iter!=last;++iter) {

func(\*iter);

}

}

}

比如上面的代码制定了for\_each的默认函数类型 后面也得进行实现这个默认的类型！

* 1. 类模板使用模板参数

**代码演示：**

类模板使用模板参数的注意点：

#### （\*）模板的重载，全特化和偏特化

1. 模板的重载
   1. 函数模板是可以重载的（类模板不能被重载），通过重载可以应对更加复杂的情况。比如在处理char\*和string对象时，虽然都可以代表字符串，但char\*在复制时直接拷贝内存效率明显更高，string就不得不依次调用构造函数了。所以在一些比较最求效率的程序中对不同的类型进行不同的处理还是非常有意义的。

**代码演示：**

其实函数模板的重载和普通函数的重载没有什么区别。

template<typename T>

void test(const T& parm) {

*std*::*cout* << "T&" << *std*::*endl*;

}

template<typename T>

void test(T\* parm) {

*std*::*cout* << "T\*" << *std*::*endl*;

}

template<typename T>

void test(double parm) {

*std*::*cout* << "double" << *std*::*endl*;

}

在讲完类模板的特化后就能知道重载和特化的区别了，这一点暂时不用在意。

1. 模板的特化
   1. 模板特化的意义：函数模板可以重载以应对更加精细的情况。类模板不能重载，但可以特化来实现类似的功能。
   2. 模板的特化也分为两种，全特化和偏特化。模板的全特化：就是指模板的实参列表与与相应的模板参数列表一一对应。

这么说可能有些繁琐，直接看代码其实并不复杂，

**代码演示：**

* 1. 模板的偏特化：偏特化就是介于普通模板和全特化之间，只存在部分类型明确化，而非将模板唯一化。

**代码演示：**

**对于模板特化 最经典的利用就是利用特化来获取指针对象的类型!**

* 1. **其实对于函数模板来说，特化与重载可以理解为一个东西。**

**总结：函数模板的重载，类模板的特化。还是比较重要的知识点，应当掌握，在一些比较复杂的程序中，模板重载与特化是经常使用的。**

## Part6：stl标准库

#### （\*）stl介绍与6大模块介绍

1. stl的介绍：
   1. stl就是（standard template library）的简称，定义在std命名空间中，定义了C++常用的容器与算法等。

可以说stl极大的提高了我们的程序开发效率。

**在C++开发中，可以说：不会用stl的人，会用stl但不懂stl实现原理的人，既会使用stl，又懂得stl实现原理的人是完完全全的三个档次。**

* 1. 泛型编程的概念：用模板进行编程，可以实现一些其它方式难以实现的功能，但对于新手来说，泛型编程可能会难以理解，摸不着头脑。

也就是说，模板是学习泛型编程的基础。

**注意：泛型编程不属于面向对象编程的范畴，泛型编程和面向对象编程是并列的。**

* 1. stl作为泛型编程的最典型代表，它实现了其它编程方式难以实现的效果，比如将整个模板库分为六个部分，每个部分可以单独设计。举个最简单的例子，vector和map在数据结构方面完全不一样，但stl可以设计出“迭代器”这个模块，让该模块可以在不同的数据结构中按照同样的方式运行。这种技术没有泛型编程是难以实现的。

1. 学习stl的注意事项
   1. **学习stl一定要有全局观念，不要局限于单个容器，重点在于明白六大组件之间的联系。**
   2. **当然，如果只是单纯为了应付当前的业务，单独学一下某个容器的用法也没有问题。**
2. SLT的六大容器介绍：
   1. 容器（container）：是一种数据结构，也就是真正用来存储数据的地方。分为三类
      1. 顺序式容器：
      2. 关联式容器：
      3. 无序式容器：其实无序式容器也是一种关联式容器，但是既然C++标准委员会将无序容器与关联式容器平行的列了出来，那么我们这里也就让无序式容器和关联式容器平行吧。
   2. 迭代器（iterator）：提供了可以访问任何容器的方法。
   3. 算法（alogorithm）：用来操作容器中的数据的模板函数。
   4. 仿函数（functor）
   5. 适配器（adaptor）
   6. 分配器（allocator）

这一课只要知道有这六大模块就可以了。至于这六大模块是干什么的，后面慢慢介绍。

#### （\*）容器

**容器的各项操作我已经单独列出来的了，就在附页3。这里只介绍最核心的操作。**

**这门课就不讲基础的数据结构了，这些东西建议熟练之后用来提升自己。数组，链表，树，哈希表如果不明白，可以去百度一下，新手了解概念就可以了。**

1. 顺序容器（sequence container）：每个元素都有固定的位置，**位置取决于插入时间和地点，与元素的值无关**
   1. vector：**将元素置于一个动态数组中，可以随机存储元素（也就是用索引直接存取）。**

数组尾部添加或删除元素非常迅速。但在中部或头部就比较费时。

* vector.assign 表示容器被重新赋值
* vector.swap表示两个容器呼唤

**代码演示：**

* 1. deque：“double end queue”的缩写，也就是双端队列。**deque的实现相比于vector有些复杂，但本质仍然是优化过的动态数组，只不过相比于单纯的动态数组，在前面添加或删除元素非常快了。**

**可以随机存储元素。**头部和尾部添加或删除元素都非常快（略慢与vector）。但在 中间插入元素比较费时（和vector差不多）。

**代码演示：**

* 1. list：本质就是链表，所以自然具有了链表的属性。

**不能随机存取元素（也就是list无法用索引存取元素）**。在任何位置插入和删除元 素都比较迅速。（在任何位置插入删除元素的时间相同，在元素头部操作慢于deque，在元素尾部操作慢于deque和vector）

**代码演示：**

* 1. string：没什么好说的，就是把普通字符串封装了一下

代码演示：

* 1. forward\_list：单项链表，简单来说就是受限的list，凡是list不支持的功能，它都不支持。做各种支持的操作效率都会高于list，最典型的就排序算法了，forword\_list要优于list。
     1. ForwordList 只提供前向迭代器，而不是双向迭代器。因此它也不支持反向迭代器。
     2. ForwordList不提供成员函数 size()。
     3. ForwordList 没有指向最末元素的锚点。基于这个原因，不提供用以处理最末元素的成员 back(),push\_back(),pop\_back()。

1. 关联容器（associated container）：**元素位置取决于元素的值，和插入顺序无关。**
   1. set/multiset：使用“红黑树”实现，是一种高度平衡的二叉树，如果大家不了解红黑树，可以去百度一下。了解个大概就可以了。**二叉树的本质决定了set/multiset的元素存取值取决于元素本身的值，和插入顺序无关。**

内部元素的值依据元素的值自动排列，与插入顺序无关。set内部相同数值的元素只能出现一次，multiset内部相同数值的元素可出现多次。容器用二叉树实现，便于查找。

**代码演示：**

* 1. map/multimap：使用“红黑树”实现，是一种高度平衡的二叉树。

内部元素是成对的“key/value”，也就是“键值/实值”，内部元素依据其键值自动排序，map内部相同的键值只能出现一次，multimap则可以出现多次。

**代码演示：**

1. 无序式容器（unordered container）：
   1. unordered\_map/unordered\_multimap：使用“哈希表”实现的，由于哈希表的特性，实现了真正的无序。如果不理解为什么使用“哈希表”就是真正无序的，可以去百度一下“哈希表”，或者干脆直接记住就可以了。

使用方法也是“key/value”，和map/multimap类似。

* 1. unordered\_set/unorder\_multiset：同样使用“哈希表”实现的。自然具有了哈希表实现的容器的特点。

使用方法和setl/multiset类似。

1. 关联式容器和无序式容器的对比：
   1. **关联式容器都是有序的，对于那些对顺序有要求的操作，关联式容器效率会高很多。（比如增加元素，删除元素）**
   2. **无序容器都是真正的无序，在查找数据方面有着优势。（比如修改特定元素，查找元素）**
   3. **从内存消耗的角度讲，无序容器要高于关联容器不过这并不重要。**

**一句话来说，如果从这两类容器中选一个使用的话。如果是增加，删除元素比较频繁，就使用关联式容器。如果修改元素，查找元素比较平凡，就使用无序容器。**

1. **我们在处理数据时应该选择什么容器呢？**
   1. **在我们需要使用存储“key/value”的容器时，只能使用map/multimap/unoredered\_map/unordered\_multimap。如果增加删除频繁，就使用map/multimap，修改，查找频繁，就使用unordered\_map/unoredered\_multimap。**

**在真正的大型项目中，常常会对这两种容器进行测试，普通练习靠感觉就可以了**

* 1. **在处理普通元素：**
     1. **当元素需要频繁插入删除时，选择顺序容器。**
        1. **如果在尾部插入删除，选择vector**
        2. **在头部，尾部插入删除，选择deque**
        3. **在中间插入，删除，选择list**
     2. **当元素需要频繁查找时，选择.set/multiset/unorder\_set/unorder\_multiset。**
        1. **频繁增加，删除时，选set，**
        2. **频繁查找，修改时，选ordered\_set**

**我们发现，对于普通元素，容器的选择不怎么容易判断。**

**其实在真正的大型项目中，要对各种容器进行测试的，普通练习一般选择vector或set就可以了。这两个使用是比较频繁的，**

#### （\*）迭代器

1. 迭代器介绍：**迭代器提供了一种可以顺序访问容器各个元素的方法，可以让我们无视不同容器存储方式的不同，用同一的方式访问数据。经过前面对容器的学习，相信大家已经体会到这一点了。**
2. 迭代器的作用：能够让迭代器与容器，算法在设计，使用时互不干扰，又能无缝耦合起来。使用迭代器可以灵活操作各种容器算法，而不需要考虑不同容器间的差异。

#### （\*）算法

1. stl的算法可以分为九个种类，具体有什么已经在“**附录一**”中完全列举了。
   1. 查找算法：
   2. 排序算法：
   3. 删除和替换算法：
   4. 排列组合算法：
   5. 算数算法：
   6. 生成和异变算法：
   7. 关系算法：
   8. 集合算法：
   9. 堆算法：

**在这里只列举一些比较常用的，剩下的那些大家如果使用可以在“附录一”中查找。**

**代码演示：**

#### 仿函数

1. 仿函数定义：就是一个可以调用“()”运算符的类对象，在Part2的第10节，Part3的第五节就已经详细介绍过仿函数了。**将operator()重载的类的对象就是仿函数。**

简单来说，就是我们在用算法时最后一个参数需要一个可调用对象，stl本身已经帮我们定义了很多可调用对象，不用我们自己再去定义了。

**具体去看stl库里面的仿函数,在stl的算法,我们经常需要指定调用对象,借助这个我们可以快速地指定,比如sort升序还是降序等等**

#### 适配器与分配器

1. 什么是容器适配器：“适配器是使一种事物的行为类似于另外一种事物行为的一种机制”。适配器对容器进行包装，使其表现出另外一种行为。例如：stack<int> 实现了栈的功能，内部默认使用deque<int>容器来存储数据。
2. STL的适配器有哪些：标准库提供了三种顺序容器适配器，没有关联型容器的适配器。分别是queue（队列），priority\_queue（优先级队列），stack（栈）。
3. 适配器的使用：
   1. 要使用适配器，首先需要引入对应的头文件
      1. 要使用stack， 需要#include<stack>
      2. 要使用queue或priority\_queue， 需要#include<queue>

代码演示：

* 1. 容器适配器必须有匹配的容器：如图所示

|  |  |  |  |
| --- | --- | --- | --- |
| 种类 | 默认顺序容器 | 可用顺序容器 | 说明 |
| stack | deque | vector, deque,list |  |
| queue | deque | list,deque | 基础容器必须提供push\_front()y运算 |
| priority\_queue | vector | vector, deque | 基础容器必须提供随机访问的功能 |

代码演示：

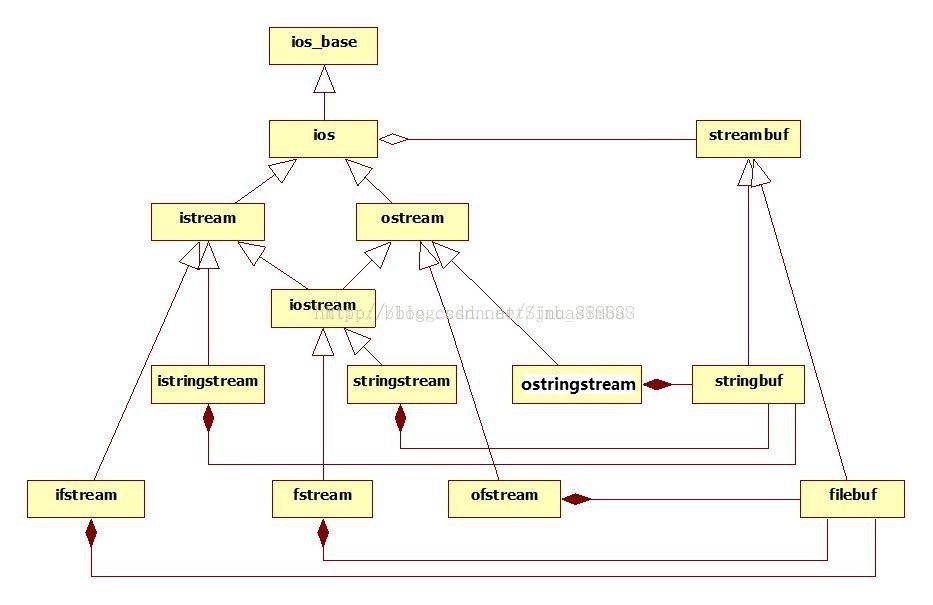
* 1. 适配器的初始化：
     1. 普通的初始化方式： stack<int> stk。
     2. 覆盖默认容器类型的初始化方式： stack<int, vector<int>> stk

1. 分配器提一下就可以了。在分配动态内存时，直接使用new，delete容易产生内存碎片化的问题，不同的分配器有不同的分配内存的方法，可以大幅提高程序对堆内存的使用效率，我们直接使用默认的分配器就可以了

## Part7：io库

#### io库介绍

1. io就是input，output的简写，也就是输入输出功能。在Part2的第4节课，就已经介绍过io功能的本质，数据在内存，磁盘，输入输出设备之间移动就是io功能。



1. io库组成部分：
   1. C++定义了ios这个基类来定义输入输出的最基本操作，这个类的具体功能我们无需了解，只需了解C++io库所有的类都继承自这个类即可。
   2. istream，ostream这两个类直接继承自ios类。
      1. ostream类定义了从**内存到输出设备（比如显示器）的功能**，我们最常使用的cout就是ostream类的对象。
      2. istream**类定义了从输入设备比如键盘）到内存的功能**，**我们最常用的cin就是istream类的对象。**
      3. **iostream文件定义了ostream和istream类的对象，就是cout和cin。所以我们只要简单的引入iostream这个头文件，就可以方便的使用这两个对象**

**注意：这个输入，输入时相对于内存来说的，输入到内存，是istream。**

* 1. ifstream，ofstream类分别继承自istream类和ostream类。
     1. ifstream定义了从磁盘到内存的功能。因为istream重载了“<<”运算符，所以ifstream对象也可以用“<<”运算符来将文件数据写入内存。除了“=”的所有重载运算符都是可以被继承的。
     2. ofstream定义了从内存到磁盘的功能。与ifstream同理，也可以用“>>”操作数据流。
     3. **fstream文件引入了ifstream和ofstream，所以我们只要引入ftream这个头文件，就可以使用文件流功能了。**

注意：这个输入输出同样是相对内存来说的。

内存与输入输出设备的数据流动，磁盘与内存的数据流动已经介绍完了。磁盘和输入输出设备直接无法直接交互，必须通过内存。

io库还为我们额外定义了字符串的输入输出类，因为对字符串的操作极为频繁，所以这个库还是很有意义的。

* 1. istringstream，ostringstream分别继承自istream类和ostream类
     1. istringstream定义了从指定字符串到特定内存的功能。与ifstream同理，也可以用“<<”运算符操作数据。
     2. ostringstream定义了从特定内存到指定字符串的功能。可以用“>>”操作数据。
     3. **sstream头文件就引入了istringstream和ostringstream，所以我们只要引入sstream这个头文集，就可以使用字符串与内存直接交互数据的功能。**

所以我们使用io库主要就三个头文件，iostream，fstream，sstream。接下来三节课会对这三个文件的使用依次讲解。

所以就是键盘<->内存 用iostream 而磁盘<->内存 用fstream 字符串<->内存 用sstream

#### （\*）io库的注意事项

**提示：这节课的介绍，这节课都是一些理论性的东西，有疑惑很正常，可以带着疑惑去学下一节课，有了代码就好理解了。**

1. **io对象无法使用拷贝构造函数和赋值运算符**

代码演示：

*std*::*istream* s(*std*::*cin*);

所以我们使用流对象无法使用值传递，一般使用引用进行传递。

1. **Io对象的状态**
   1. **io操作是非常容易出现错误的操作，一些错误是可以修复的，另一部分则发生在系统更深处，已经超出了应用程序可以修正的范围。**

比如我们使用cin向一个int类型的数中输入一个字符串，会使cin这个对象出现错误。

**代码演示：**

**所以我们在使用io对象时都应该判断io对象的状态。**

int i = 10;

while (*std*::*cin*>>i,!*std*::*cin*.*eof*())

{

if (*std*::*cin*.*bad*()) {

throw *std*::*runtime\_error*("error can not resume!");

}if (*std*::*cin*.*fail*()) {

*std*::*cin*.*clear*();

//这一句是清空缓冲区

*std*::*cin*.*ignore*(*std*::*numeric\_limits*<*std*::*streamsize*>::*max*(), '\n');

*std*::*cout* << "data format err!\r\n" << *std*::*endl*;

continue;

}

*std*::*cout* << i << *std*::*endl*;

}

*std*::*cout* << "process terminate" << *std*::*endl*;

return 0;

上面的代码非常重要,eof是指end of file 也就是结束了,否则必须判断cin的状态

**比如：while(cin >> val) 或if(cin >> val)（不要只用这两个进行控制，最好搭配iostate来使用）**

**代码演示：**

* 1. 我们需要知道流对象错误的原因，因为不同的错误需要不同的处理方法。

io库定义了iostate类型，可以完整的表示io对象当前的状态。在不同的平台中， iostate实现方法略有区别，在vs中直接用int来代表iostate类型，将不同的位置1 以表示不同的状态。可以与位操作符一起使用来一次检测或设置多个标志位。

可以用**rdstat函数**来获得io对象当前用iostat类型来表示的状态：

**代码演示：**

* 1. iostata类型有以下状态
     1. badbit状态，系统级错误，一旦表示badbit的位被置为1，流对象就再也无法使用了。
     2. failbit状态，代表可恢复错误，比如想读取一个数字却读取了一个字符，这种错误就是可以恢复的。当badbit位被置1时，failbit位也会被置1。
     3. eofbit状态，当到达文件结束位置时，eofbit和failbit位都会被置1。
     4. goodbit状态，表示流对象没有任何错误。

只要badbit，failbit，eofbit有一位被置为1，则检测流状态的条件就会失败。

* 1. 标准库还定义了一组成员函数来查询这些标志位的状态
     1. good()函数在所有错误位均未置1的情况下返回true。
     2. bad()，fail()，eof()函数在对应位被置1的情况下返回true。因为badbit位被置1或eofbit位被置1时，failbit位也会被置为1。所以用fail()函数可以准确判断出流对象是否出现错误。
     3. 实际上，我们将流对象当做条件使用的代码就等价于“!fail()”
  2. 流对象的管理
     1. rdstate函数，返回一个iostate值，对应当前流状态
     2. setstate(flag) 函数，将流对象设置为想要的状态
     3. clear函数：是一个重载的函数。
        1. clear()，将所有位置0，也就是goodbit状态。
        2. clear(flag)，将对应的条件状态标志位复位。
     4. ignore函数：

作用：提取输入字符并丢弃他们。

函数原型：istream& ignore (streamsize n = 1, int delim = EOF)

*std*::*cin*.*ignore*(*std*::*numeric\_limits*<*std*::*streamsize*>::*max*(), '\n');

*这句话的意思就是找最大容量 一直找到\n为止 全部丢弃*

              读取到前n个字符或在读这n个字符进程中遇到delim字符就停止，把读 取的这些东西丢掉

代码演示：

#### 内存与输入输出设备的交互（iostream）

1. **getline：**

其实iostream已经没什么好讲的了，比较常用的就是这个getline了，getline其实并不复杂，不过是按行接收数据罢了，**因为存储在string对象中，所以不容易出现格式错误，但仍然可能出现系统错误，所以在企业级程序中，还是应当对bad的情况进行处理。**

代码演示：

*std*::*string* str;

while (*std*::*getline*(*std*::*cin*, str), !*std*::*cin*.*eof*()) {

if (*std*::*cin*.*bad*()) throw *std*::*runtime\_error*("1!!");

*std*::*cout* << str << *std*::*endl*;

}

1. **get:**

还有个不怎么常用的get函数。get函数的用法和getline类似，只不过get是以字符的格式进行接收。**在企业级代码中仍然需要对bad的情况进行处理。**

**剩下的也没什么了，iostream常见的用法在上一课已经讲过了，iostream就这些了。**

char c;

while (*std*::*cin*.*get*(c), !*std*::*cin*.*eof*()) {

if (*std*::*cin*.*bad*()) throw *std*::*runtime\_error*("1!!");

*std*::*cout* << c << *std*::*endl*;

}

#### （\*）内存与磁盘的交互（fstream）

1. fstream相对于iostream。多了很多自己独有的操作
   1. i**o库默认没有给ifstream和ofstream类提供对象，需要我们自己去定义。**
   2. fstream对象创建方式有三种
      1. 可以使用默认构造函数进行定义。例如： ifstream fstrm，

代码演示：

* + 1. 也可以在创建流对象时打开想要打开的文件。例如ifstream fstrm(s)。s可以是字符串，也可以是c风格的字符串指针。文件的mode依赖于流对象的类型。

代码演示：

* + 1. 也可以在打开文件时就指定文件的mode。例如ifstream fstrm(s, mode)

代码演示：

* 1. fstrm.open(s)函数，打开名为s的文件，并将文件与fsrm绑定，s可以是一个string，也可以是一个c风格的字符串指针。

代码演示：

* 1. fstrm.close()函数，关闭文件。**注意，一定不要忘了。**

代码演示：

* 1. fstrm.is\_open()函数，返回一个bool值，指出与fstrm关联的文件是否成功打开且尚未关闭。

代码演示：

**如果新手只看这些描述，可能会很迷糊，所以我接下来要写一段代码，大家只要把这段代码记住，文件部分就没有问题了。**

**这段代码的目的是：让客户输入文件名称，如果文件不存在，就让客户重新输入文件名称，如果文件存在，就将文件全部内容输出。**

*std*::*string* fileName;

*std*::*string* fileContent;

while (*std*::*cin* >> fileName, !*std*::*cin*.*eof*()) {

if (*std*::*cin*.*bad*()) throw *std*::*runtime\_error*("!");

*std*::*ifstream* ifs(fileName);

if (ifs.*is\_open*()) {

while (*std*::*getline*(ifs, fileContent)) {

*std*::*cout* << fileContent;

}

if (ifs.*bad*()) {

throw *std*::*runtime\_error*("!");

}

ifs.*close*();

}

else {

//没打开

ifs.*clear*();//清空状态

ifs.*ignore*(*std*::*numeric\_limits*<*std*::*streamsize*>::*max*(), '\n');

*std*::*cout* << "FILE NOT EXIES!" << *std*::*endl*;

continue;

}

}

1. 文件模式：
   1. in以读的方式打开
   2. out以写的方式打开
   3. app在进行写操作时定位到文件末尾
   4. ate打开文件后立即定位到文件末尾
   5. trunc截断文件（也就是文件已有的全部删除，重新开始写）
   6. binary以二进制方式打开文件
2. 文件模式需要强调以下几点
   1. 与ifstream关联的文件默认in模式打开。
   2. 与ofstream关联的文件默认out模式打开
   3. 与fstream关联的文件默认in和out模式打开 也就是读写
   4. 默认情况下，即使我们没有指定trunc，以out模式打开的文件也会被截断。为了保持以out模式打开的文件的内容，我们必须同时指定app模式或in模式。
   5. 只可以对ifstream或fstream的对象设定in的模式
   6. 只可以对ofstream或fstream的对象设定out的模式
   7. 只有当out模式被设置时才可以设置trunc模式
   8. ate和binary模式可以应用与任何类型的文件流对象，且可以与任何其它文件模式组合使用。

代码演示：

1. **总结：文件流这部分还是有一些东西的，新手理解起来可能有些困难，没办法，用的多了就好了。其实常用的就那么几点。**

#### 内存之中对于字符串的操作（sstream）

1. string流介绍：string流可以向string对象写入数据，也可以从string对象读取数据。与文件操作类似，只不过数据交互变成了从内存到内存。

代码演示：string流默认包含一个string对象，当然，我们也可以指定。

1. string流有哪些
   1. istringstream从string对象读取数据
   2. ostringstream向string对象写数据
   3. stringstream既可以从string对象读取数据，也可以向string对象写数据
2. string流对象继承自iostream对象，除了继承得来的操作，string流对象还有自己的成员来管理流相关的string。
   1. 对于string流，io库是没有像cout，cin这样的自定流对象的。流对象需要我们自己去定义 *std*::*cout* << istrStream.str() << *std*::*endl*; 必须这样写 不能直接cout<<!
      1. sstream strm：sstream代表一个string流对象的类型，以下同理。strm是一个未绑定的stringstream对象。
      2. Sstream strm(s)：strm是一个绑定了s的拷贝的string流对象。s是一个string对象
   2. strm.str()：返回strm所保存的string的拷贝。
   3. strm.str(s)：将s拷贝到strm中，返回void
3. string流对象的作用
   1. **对数据类型进行转化，也就是string和其它类型的转化，这是string流对象最重要的功能。**
      1. string转化为int等类型。

**代码演示：**

*std*::*string* hh("d11s2");

*std*::*stringstream* strStream(hh);

int \_i = 0;

strStream >> \_i;

if (strStream.*bad*()) throw *std*::*runtime\_error*("failed to !");

if (strStream.*fail*()) {

strStream.*clear*();

strStream.*ignore*(*std*::*numeric\_limits*<*std*::*streamsize*>::*max*(),'\n');

*std*::*cout* << "format err!" << *std*::*endl*;

}

else {

*std*::*cout* << \_i << *std*::*endl*;

}

* + 1. int等类型转化为string。

1. int ss;
2. *std*::*stringstream* strS;
3. strS << ss << *std*::*endl*;
4. if (strS.*bad*()) throw *std*::*runtime\_error*("ss");
5. *std*::*cout* << strS.str() << *std*::*endl*;

**代码演示：**

* 1. 用于对空格分隔的字符串的切分，

**代码演示：**

*std*::*string* st("hello world hah hh");

*std*::*string* destString;

*std*::*stringstream* divStream(st);

while (divStream >> destString) {

if (divStream.*bad*()) throw *std*::*runtime\_error*("hha");

*std*::*cout* << destString << *std*::*endl*;

}

## Part8：多线程

**注意：多线程的东西其实有很多，这门课只讲常用的部分，把这些学会基本就够用了。**

#### 多线程基本概念介绍

1. **多线程的重要性：**
   1. 对于一个专业的C++开发来说，多线程是必须掌握的模块。
   2. 现代程序都是多线程程序了。因为单核处理器的性能早已经达到了瓶颈，只能往多核方向发展。现代的个人计算机都是4核起步，工作站，服务器就更不用说了。
   3. 工作站可以理解为处理能力更强的大型个人计算机，常见的12核，16核。服务器有48核的，甚至更多。
   4. 对于一个计算机来说，是不是说核越多好呢？不是，多核会导致单核的工作性能下降。当核数多到一定程度后，反而总体运行效率下降了。不过，这并不影响现代计算机核数越来越多的趋势。
   5. 传统的单线程程序同时只能在一个核上运行，这是不是太浪费资源了。计算机有8个核，你就用了一个，暴殄天物啊。多线程程序可以使用多个核，极大提高程序运行效率。现在网络通信，音频，视频，游戏服务都是多线程程序。
2. **并发与并行的概念介绍：**
   1. 一句话来说：并行是同时在不同的处理器上处理不同的任务，并发是“同时”在一个处理器上处理多个任务。

解释一下：

* + 1. 并行是指有多个处理器。每个处理器各执行一个线程，互不抢占cpu资源，如果线程数量多于CPU，也没有办法，只能将处理器的时间划分为多个时间段，再将时间段分配给各个线程。
    2. 并发是指只有一个处理器，但多个线程被轮换快速执行，使得宏观上有了同时执行的效果。作用原理是将单处理器的时间划分为多个时间段，再分配给不同的线程。同一时间段只能有一个线程在运行，其它线程均处于挂起状态。

1. **进程的概念：**
   1. 进程的概念在面向进程设计的操作系统（就是unix，也包括后面衍生出的linux，mac）和面向线程设计的操作系统（说的就是windows）上有很大区别，两种设计方式的共同点与不同点还是需要理解的。
      1. 进程是计算机中的程序对某些数据集合的一次运行活动，是系统进行资源分配和调度的最基本单位，是操作系统的结构基础。再用大白话说一遍，一个可执行程序执行起来，就是一个进程。当然，一个程序要执行起来需要各种资源，这些资源就是数据集合。
      2. 在面向进程设计的计算机结构中，进程是程序的基本执行单位，进程包括程序执行的所有资源，同时自己也可以执行。
      3. 在面向线程设计的计算机结构中，线程才是程序的基本执行单位，进程不过是线程的容器罢了。进程就像一个仓库，里面存放了程序的所有资源，进程中的线程才是真正执行程序的单元。
2. **线程的概念：**
   1. linux的线程和windows的线程还是有很大区别的。
   2. linux的线程就是一种轻量级的进程，只有依靠进程才可以存在。也模拟出了windows线程的方式，让线程成为真正的执行单元。
   3. windows的线程就简单多了，真正执行程序的最小单元。
3. **总结：**  
    说了这么多：其实对进程，线程只是个介绍，这里面水很深。而且windows多线程和linux多线程的区别并不影响我们学习C++11的多线程，C++标准任何平台通用。

**现代C++程序，C++11的多线程功能才是主流，C++11的多线程就是windows模式的，进程为一个仓库，线程才是程序执行的最小单元。linux同样完美支持了这些功能。**

#### （\*）线程的创建

1. 主线程介绍：一个程序执行起来就是一个进程。而main函数就是主线程，一旦主线程执行完毕，主线程结束，整个进程就会结束。
2. 子线程介绍：在一个线程执行时，我们可以创建出另外一个线程。两个线程各自执行，互补干涉。注意，当主线程执行完毕，就会强制结束所有子线程，然后进程结束，从这个角度来说，可以认为子线程是主线程的辅助线程。**但是要明白主线程和子线程是平级的，只不过主线程执行完毕后会给所有子线程发送一个信号，使所有子线程强制结束。**
3. 子线程的创建方式：很简单，直接使用thread类就可以了。

**代码演示：**

**括号中只要是一个可调用对象就没有问题了。**

*std*::*thread* myThread([]() {

*std*::*cout* << "hello world!\r\n" << *std*::*endl*;

});

1. **子线程创建后如果就不管了，那么会出现非常严重的问题。**
   1. 有些子线程负责对部分数据的处理，主线程必须要等到子线程处理完毕才能继续执行，所以join函数就诞生了。

**代码演示**：

使用了join函数后，主线程就会处于挂起状态，直到子线程执行完毕才可以继续执行。

* 1. 有些子线程和主线程完全分离，各自执行各自的。但主线程执行完毕，**子线程就会立马被强制结束，容易导致各种bug，查都不知道从哪里开始查。**于是deatch函数就诞生了。

**代码演示：**

detach()函数可以让**子线程被C++运行库接管**，就算主线程执行完毕，子线程也会由C++运行时库清理相关资源。保证不会出现各种意想不到的bug。

myThread.*detach*();

也就是需要等,就调用join,不等就调用detach!

#### （\*）传递线程参数

1. 传递子线程函数的参数：直接传递即可，注意：传递参数分为三种方式，值传递，引用传递，指针传递。

代码演示：

Std::thread(func,param1,param2..);

1. 传递参数注意事项：
   1. 在使用detach时不要传递指针，或者说在设置子线程函数时，不要设置指针参数。因为值传递和引用传递并未直接传递地址，而指针传递却直接传递地址。所以当使用deatch时，传指针就会导致错误，指针已经被系统回收，所以不要千万不要传指针。

**代码演示：**

* 1. **在使用detach时不要使用隐式类型转化，因为很有可能子线程参数还没来的及将参数转化为自己的类型，主线程就已经执行完毕了。**

1. **总结：**
   1. **普通类型在传递子线程函数参数时，直接值传递即可。**
   2. **类类型传递引用就可以了，类类型传递引用会首先调用一次复制构造函数生成一个临时变量，故而导致地址不相同。如果采用值传递，需要两次复制构造函数，开销更大。**
2. **std::ref的用法：**

根据刚才的演示，使用普通的引用传递会调用一次复制构造函数，导致函数无法对引用对象进行修改，于是std::ref诞生了，它可以使子线程在传递参数时不再调用复制构造函数。

代码演示：

int s = 0;

*std*::ref(s);

*std*::*thread* myThread([](int& refS) {

*std*::*cout* << "hello world!\r\n" << *std*::*endl*;

refS = 100;

for (auto i = 0; i < 1000000000; i++) {}}, *std*::ref(s)

);

//myThread.detach();

myThread.*join*();//如果子线程没完毕 主线程会等待

*std*::*cout* << s << *std*::*endl*;

但是一定要`join!`

#### （\*）线程id的概念：

1. 线程id定义：每个线程都有自己的id，不管是主线程还是子线程都有自己的id。直接使用std::this\_thread::get\_id()就可以获得当前线程的id。

代码演示：

1. 注意：线程是依附于进程存在的，所以不同的进程可以有相同的线程id。

这一课很简单，但这个知识点不知道往哪里放，就单独拿出来了。

#### （\*）数据共享与数据保护

1. 多个线程的执行顺序是乱的，具体执行方法和处理器的调度机制有关系。从开发者的角度讲，就是没有规律的。

代码演示：

1. 在讲数据保护问题之前，为了帮助大家理解数据保护问题，这里额外扩展一些关于汇编的知识。科班的同学应该很熟悉，给非科班的人介绍一下。

一个进程运行时，数据存储在内存中。如果一个数据要进行运算，必须先将数据拷贝到寄存器中。比如要对栈上的一个int i进行“++”操作，需要将i的值拷贝到寄存器中，将该值自加后再拷贝到原来的内存。

如果此时有两个线程均进行的是这样的操作，可能出现两个进程都拷贝了i原来的值到寄存器，然后各种加一，再拷贝到i对应内存的情况，最终导致i这个变量只自加了一次。

这是同时写数据的情况，那么一读一写呢？这也是有问题的，谁知道读数据时写数据步骤已经到了哪里，谁知道读出来的是个什么东西。

1. 数据保护问题：
   1. 数据保护问题总共有三种情况：
      1. 至少两个线程对共享数据均进行读操作，完全不会出现数据安全问题。
      2. 至少两个线程对共享数据均进行写操作，会出现数据安全问题，需要数据保护。
      3. 至少两个线程对共享数据有的进行读，有的进行写，也会出现数据安全问题，需要进行数据保护。

代码演示：

*std*::*thread* t1([]() {

for(unsigned i=0;i<1000000;i++)

g\_num++;

});

*std*::*thread* t2([]() {

for (unsigned i = 0; i < 1000000; i++)

g\_num++;

});

t1.*join*();

t2.*join*();

*std*::*cout* << g\_num << *std*::*endl*;

数据保护的方法一共就两种：互斥锁，原子操作。

1. **互斥锁：**
   1. 互斥锁的作用原理很简单，对共享数据加锁，当一个线程对这块数据进行操作时，别的线程就无法对该区域数据进行操作。

代码演示：

*std*::*thread* t1([]() {

for (unsigned i = 0; i < 1000000; i++) {

m.*lock*();

g\_num++;

m.*unlock*();

}

});

*std*::*thread* t2([]() {

for (unsigned i = 0; i < 1000000; i++) {

m.*lock*();

g\_num++;

m.*unlock*();

}

});

t1.*join*();

t2.*join*();

*std*::*cout* << g\_num << *std*::*endl*;

* 1. 这种方式的互斥锁有个弊端，就是lock()之后容易忘记unlock()，就和指针类似。于是和智能指针类似，也有了lock\_guard，用来防止开发人员忘了解锁。

代码演示：

1. 原子操作：（使用频率远远不及互斥锁）
   1. 原子操作的原理：将一个数据设置为原子状态，使得该数据处于无法被分割的状态，意思就是处理器在处理被设置为原子状态的数据时，其它处理器无法处理该段数据，该处理器也会保证在处理完该数据之前不会处理其他数据。

代码演示：

std::atomic<unsigned> g\_num = 0;

总结：在编写多线程代码时，数据保护是一个必须考虑，非常常用的功能。互斥锁的使用频率是远远高于原子操作，原子操作看似简单，但当需要保护的数据很多时，就会极其复杂。

**所以：对于单个数据，可以使用原子操作，其它的使用互斥锁就可以了。**

#### （\*）死锁

死锁就像两个人在互相等对方。A说，等B来了就去B现在所在的地方；B说，等A来了我就去A所在的地方，结果就是A和B都在等对面过来才能去对面。这就导致了一个死循环，放在多线程中，就是死锁。

举个例子：

解决方法也很简单。

1. 只要让两个锁顺序一致就可以了。

**代码演示：**

1. 但是让两个锁顺序一致常常是说起来容易，做起来难。于是C++11提供了std::lock。这个模板可以保证多个互斥锁绝对不会出现死锁的问题。同时提供了std::adopt\_lock的功能来避免忘记释放锁的问题。

**总结：死锁是一个比较常见的bug，面试时也经常询问死锁相关的知识。**

这一节课上完，多线程的主体部分就讲完了，后面都是使用频率较低的东西，也就是个补充。所以前六节课必须学会，每一课都是重点。后面的我就不讲了。

## Part9：异常处理

**异常处理的前情提要：很多人不喜欢使用异常处理，认为它麻烦，应对可能出现的错误要写那么多代码，会非常麻烦。**

**但实际上不是这样的，我们只需要在一些开发人员难以控制，比较容易出错的地方对异常进行处理就可以了，需要进行异常处理的地方并不多。**

举几个例子。

1. 接收传递过来的被除数，我们难以判断被除数是否为0，此时异常处理就很有意义了。
2. 接收文件名，如果文件不存在，我们可以按照之前的写法要求重发一遍，也可以直接报异常，异常就是文件不存在。
3. 我们在动态分配内存时，经常出现内存不足的情况（在大型程序中，这是非常常见）。比如我们需要动态分配一个未知大小的数组，数组大小等待传入。使用new操作符会直接抛出bad\_alloc的异常。

对new的处理非常重要，大家如果做专业的C++开发，会经常用到。

此外使用智能指针时如果内存分配不够也会抛出bad\_alloc的异常

1. 有个vector，我们需要接受一个参数，然后取出参数对应的数组元素。此时就经常出现数组的越界问题。

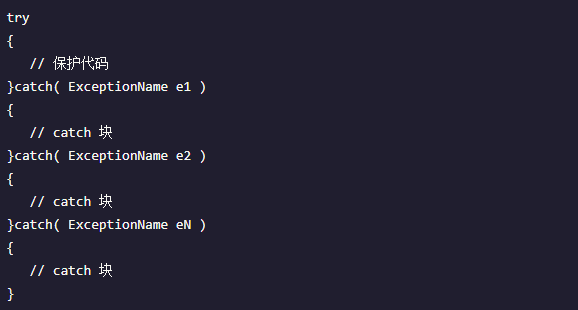
**最常用的基本就这些例子了，剩下的也都和这些类似。**

**异常处理这一章东西不多，一会儿把这些例子演示一下就可以了。**

#### 异常处理的介绍：

1. **异常是程序在执行期间产生的问题（编译期出现的错误在写代码时开发环境就有提示）**。C++的异常是指程序运行时发生的特殊情况。
2. 异常提供了一种转移程序控制权的方式。C++的异常处理涉及到三个关键字：try，catch，throw。
   1. throw：当问题出现时，程序会抛出一个异常。这是通过throw关键字来完成的。
   2. catch：在你想要处理问题的地方，通过异常处理程序捕获异常。catch关键字用于捕获异常。
   3. try：try块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个catch块。

如果有一个块抛出一个异常，捕获异常的方法会使用try和catch关键字。try块中放可能抛出异常的代码，try块中的代码被称为保护代码。常见的异常处理格式如图所示。



1. 抛出异常：throw语句可以在代码块的任何地方抛出异常，throw抛出的表达式的结果决定了抛出的异常的类型。

代码演示：

int \_div(int divisor, int dividned) {

if (dividned == 0) {

throw *std*::*string*("divined is zero");

}

else return divisor / dividned;

}

template<typename \_T>

void clientInput(const *std*::*string*& show\_str, \_T& num) {

*std*::*cout* << show\_str << *std*::*endl*;

while (*std*::*cin* >> num, !*std*::*cin*.*eof*()) {

if (*std*::*cin*.*bad*()) throw *std*::*runtime\_error*("fuck");

if (*std*::*cin*.*fail*()) {

*std*::*cin*.*clear*();

*std*::*cin*.*ignore*(*std*::*numeric\_limits*<*std*::*streamsize*>::*max*(), '\n');

*std*::*cout* << "please input format!" << *std*::*endl*;

continue;

}

break;

}

}

int main() {

int div1 = 100, div2 = 0;

clientInput("input div1", div1);

clientInput("input div2", div2);

try {

\_div(div1, div2);

}

catch (const *std*::*string*& exp) {

*std*::*cout* << exp << *std*::*endl*;

}

return 0;

}

上面是一个很好的输入输出的异常演示,但是抛出的是字符串,不是std::exception,所以只能catch接住字符串! 如何接受任何种类的异常catch(…){}即可!

try {

while (1) {

int\* p1 = new int[0x1000000]();

}

}

catch (const *std*::*bad\_alloc*& exp) {

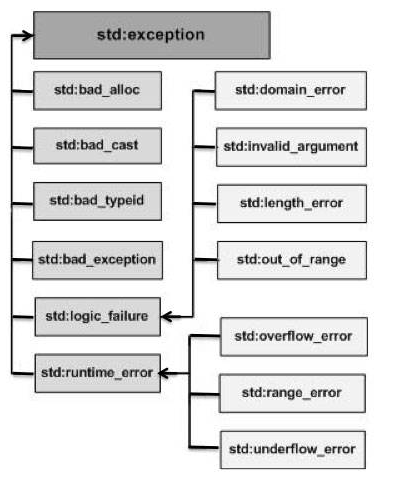
*std*::*cout* << exp.*what*() << *std*::*endl*;

}

#### C++的标准异常

1. C++提供了一系列标准的异常，定义在头文件“<exception>”中，它们是以父子层次结构组织起来的，如下图所示。

**图只要有个大致印象就可以了， 不需要全背住**





1. 别看图很复杂，异常种类有很多，但经常使用的其实就几个。
   1. bad\_alloc错误，使用new分配内存失败就会抛出bad\_alloc错误。
   2. out\_of\_range错误，在使用at时，容器越界就会抛出这个错误，这也是“at”比“[]”更加优秀的原因。
   3. runtime\_error错误，运行时错误，只有在程序运行时才能检测到的错误。这是一个相对的概念，和logic\_error形成对比。logic\_error可以读代码读出来，runtime\_error就不行。

我们也经常将一些读代码无法判断的异常标识为runtime\_error。

* 1. ... 错误，可以接受任何错误，我们一般都会在catch最后加上“...”，这样就可以接受所有类型的异常了。

**代码演示:**

**剩下的异常，我也用代码演示一下，这些异常使用频率比较低，但在某些情况下也是需要使用的。**

1. 自定义异常类型，其实需要自定义异常类型的情况真的非常少，这里就不介绍了，其实和标准异常也是一样的。

## Part10：各种难以归类但有使用价值的知识点

#### 2万能引用与引用折叠

1. 万能引用的概念：
   1. C++11除了带来了右值引用，还带来了万能引用，也就是既能当做左值，又能当做右值的引用。

注意：万能引用是既可以被编译期处理为左值引用，又可以被编译期处理为右值引用。**不是既是左值引用又是右值引用，不违背C++一个引用不是左值引用，就是右值引用的基本说法，万能引用会在编译期被当做左值引用或右值引用处理。**

1. 万能引用的格式：万能引用的格式有两种
   1. 模板型：

template<typename T>

void func(T&& parm)

代码演示：

**这个T&& 就是万能引用类型。**

**注意：只有T&& 是万能引用类型**

**以下的写法均不是万能引用：**

**const T&& parm 这就是普通的右值引用。**

**vector<T> && parm 这也是右值引用。**

**template<typename T>**

**class MyVector  
 ｛**

**public:**

**void push\_back(T&& elem)**

**｝**

**注意：此时T&& 不是万能引用，因为T影响的是MyVector的类型。**

**只有这样写，才是万能引用。**

**template<typename T>**

**class MyVector  
 ｛**

**public:**

**template<typename T2>**

**void push\_back(T2&& elem)**

**｝**

**此时T2&& 的类型完全独立于MyVector类了，每调用一次push\_back函数，都要推断T2的类型。**

* 1. auto型：

auto && var = var2；

**代码演示：**

这个auto&& 就是万能引用。

**const auto&& var 就不是万能引用了。**

1. 万能引用的作用，就是当参数为左值时。T&&为左值。当参数为右值时，T&& 为右值。就这么简单。

**代码演示：**

1. 引用折叠：引用折叠其实概念很简单

一个引用不是左值引用就是右值引用，当一个万能引用被认为左值引用时，类型应该是T& &&，此时类型就会折叠为T&。

简单来说，就是引用符号太多了，折叠为“&”或“&&”

**看代码：**

#### 完美转发

1. C++完美转发的定义：完美转发是什么呢？说到底，它描述的其实就是一个参数传递的过程，能够将一个传递到一个函数的参数，再通过该函数原封不动的传递给另一个函数（**这里的原封不动不单是指参数的值，更包括参数的类型，参数的限定符**）

光用语言描述确实描述不清楚，所以。

**代码演示：**

1. 我们发现以前的传递参数的方法都无法在万能引用中解决完美转发的问题。

**代码演示：**

1. 于是C++提供了forward模板来解决完美转发的问题，forward模板可以使参数推断出它原来的类型，实现了完美转发。
2. **总结：完美转发就是一个专门配合万能引用的知识点，专门用来在使用万能引用的地方原封不动的传递参数。其实记住它是和万能引用配合使用的就掌握的差不多了，**

**最后再说一点，其实Part10原本还是打算讲一些东西的，后来想了想，这些都是比较复杂的东西了，新手根本用不到，用的到的人也都是一些老家伙了，完全有了自己查找资料的能力，所以像萃取这种知识就没有讲。**

**这些东西，绝对够新手看了，非常的全，也非常实用。**

## 附页1：STL全部的算法

**注意：我这个文档的主要功能还是给STL的算法分个类，要查看算法严格的描述，还是去微软官网查看吧，官方文档又准又全。**

#### <一>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** 这个重复是可以重载的，找到则返回指向这对元素的第一个元素forwardIterator。否则返回最后一个元素的forwardIterator。

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

auto iter = *std*::*adjacent\_find*(iVec.*cbegin*(), iVec.*cend*(), [](int elem,int elem2) {

if (elem >= 12 && elem2 < 12) return true;

else return false;

});

if (iter != iVec.*cend*()) {

*std*::*cout* << "distance" << *std*::*distance*(iVec.*cbegin*(), iter) << *std*::*endl*;

}

else {

*std*::*cout* << "no same !" << *std*::*endl*;

}

//std::cout << iVec[4] << std::endl;

*std*::*cout* << iVec.*at*(4) << *std*::*endl*;

return 0;

##### 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。

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

//先排序

*std*::*sort*(iVec.*begin*(), iVec.*end*(), [](int elem,int elem2) {

return elem > elem2; //降序排序

//return elem < elem2; //升序

});

*std*::*cout* << *std*::*binary\_search*(iVec.*begin*(), iVec.*end*(), 4, [](int elem1, int elem2) {

return elem1 > elem2;

});

这个4就是要找的值 如果有 就返回1 否则返回0

##### 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。

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

//先排序

*std*::*cout* << *std*::*count\_if*(iVec.*begin*(), iVec.*end*(), [](int elem) {

return elem >= 12;

}) << *std*::*endl*;

##### equal\_range:

**注意，必须对有序容器进查找，下面的lower\_bound和upper\_bound也是同理。**

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迭代器，否则返回输入的"另外一对"的第一个迭代器。重载版本使用用户输入的操作符代替等于操作。

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

//先排序

*std*::*vector*<int> iVec2{3, 4};

auto \_iter=*std*::*find\_end*(iVec.*begin*(), iVec.*end*(), iVec2.*cbegin*(), iVec2.*cend*());

*std*::*cout* << "distance " << *std*::*distance*(iVec.*begin*(), \_iter) << *std*::*endl*;

##### 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。

这个和上一个区别就是这个是查找第一次出现的完全匹配的位置 上一个是最后一个

##### find\_if:

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

//先排序

*std*::*vector*<int> iVec2{3, 4};

//find\_if 有谓词

auto \_\_iter=*std*::*find\_if*(iVec.*begin*(), iVec.*end*(), [](int elem) {

if (elem % 2)

return true;

});

使用输入的函数代替等于操作符执行find。

##### lower\_bound:

返回一个iterator，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iterator，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:

**这两个是真的不好描述，去微软官网查看一下吧，简单，比我在这里总结的强多了。**

search\_n:

#### <二>排序和通用算法(14个)：提供元素排序策略

##### inplace\_merge:

##### merge:

*std*::*vector*<int> iVec{1, 12,12, 3, 4};

//先排序

*std*::*vector*<int> iVec2{3, 4};

*std*::*vector*<int> nVec{1};

*std*::*sort*(iVec.*begin*(), iVec.*end*());

*std*::*sort*(iVec2.*begin*(), iVec2.*end*());

*std*::*merge*(iVec.*cbegin*(), iVec.*cend*(), iVec2.*cbegin*(), iVec2.*cend*(), *std*::*back\_inserter*(nVec));

for (const auto elem : nVec) {

*std*::*cout* << elem << *std*::*endl*;

}

这里需要注意的是,如果使用initialize\_list初始化,那么这个大小就固定了,vector就不是固定数组了;

##### nth\_element:

##### partial\_sort:

##### partial\_sort\_copy:

##### partition:

##### random\_shuffle:

就是随机洗一遍 std::random\_shuffle(begin,end);

##### reverse:

##### reverse\_copy:

##### rotate:

##### rotate\_copy:

##### sort:

##### stable\_sort:

##### stable\_partition:

#### <三>删除算法(15个)

##### copy:

##### copy\_backward:

##### iter\_swap:

##### remove:

##### remove\_copy:

##### remove\_if:

##### remove\_copy\_if:

##### replace:

##### replace\_copy:

##### replace\_if:

##### replace\_copy\_if:

##### swap:

##### swap\_range:

##### unique:

删除相同的元素

##### unique\_copy:

#### <四>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:

##### prev\_permutation:

#### <五>算术算法(4个)

##### accumulate:

##### partial\_sum:

##### inner\_product:

##### adjacent\_difference:

#### <六>生成和异变算法(6个)

##### fill:

##### fill\_n:

##### for\_each:

##### generate:

##### generate\_n:

##### transform:

#### <七>关系算法(8个)

##### equal:

##### includes:

##### lexicographical\_compare:

##### max:

##### max\_element:

##### min:

##### min\_element:

##### mismatch:

#### <八>集合算法(4个)

##### set\_union:

##### set\_intersection:

##### set\_difference:

##### set\_symmetric\_difference:

#### <九>堆算法(4个)

##### make\_heap:

##### pop\_heap:

##### push\_heap:

##### sort\_heap:

## 附页2：STL标准库提供的仿函数

#### 算术类仿函数

##### 加：

plus<T>

##### 减：

minus<T>

##### 乘：

multiplies<T>

##### 除：

divides<T>

##### 模取：

modulus<T>

##### 取负：

negate<T>

#### 关系运算类仿函数

##### 等于：

equal\_to<T>

##### 不等于：

not\_equal\_to<T>

##### 大于：

greater<T>

##### 大于等于：

greater\_equal<T>

##### 小于：

less<T>

##### 小于等于：

less\_equal<T>

#### 逻辑运算仿函数

##### 逻辑与：

logical\_and<T>

##### 逻辑或：

logical\_or<T>

##### 逻辑否：

logical\_no<T>

## 附页3：STL各种容器的操作：

**注意：**

1. **不需要死记硬背，用的多了自然就会了。**
2. **容器的各种函数最好的方式就是打开vs，函数所有的参数都有显示，看不懂就去微软官网查一查。**

#### vector的各种函数

##### 构造函数

1. vector()：创建一个空的vector
2. vector(const std::allocator<T>& al)：使用指定的分配器来分配内存。**allocator就是一个内存分配器，vector已经指定了默认的分配器了，不需要我们去主动调用，以后设计allocator直接忽略就可以了，其实这个构造函数只不过是用指定的分配器去创建一个空的vector罢了。**
3. vector(std::vector<T>&& right, const std::allocator<T>& al)：就是移动构造函数，第二个参数表示我们指定分配器。
4. vector(const std::vector<T>& vec, const std::alloctor<T>& al)：就是复制构造函数，分配器可以自己指定，当然，一般来说，vector默认的分配器就够用了。
5. vector(std::initializer\_list<T>& initList, const std::allocator<T>& al)：就是使用initializer\_list来初 始化容器，第二个参数表示我们可以指定分配器。
6. vector(iter first, iter last, const std::allocator<T>& al)：就是容器初始有迭代器[first, last)的内容（**这里使用deque，list的迭代器也可以**），第三个参数还是表示我们可以指定分配器。
7. vector(const size\_t count, const std::alloctor<T>& al)：创建一个vector，元素个数为count。元素均为默认值，如果是普通类型，则赋值为0。如果是类类型，则均使用默认构造函数进行初始化。
8. vector(const size\_t count，const T& t)：创建一个vector，元素个数为count，且值均为t。

##### 增加函数

1. void push\_back(const T& value)：向容器尾部增加一个元素value。
2. void push\_back(T&& value)：向容器尾部增加一个元素value，这不过这次以右值引用的形式添加。
3. std::vector<T>::iterator insert(std::vector<T>::const\_iterator& where, std::initializer\_list<T> initList)：在where迭代器指定的地方添加initList，**返回值为指向新添加的第一个元素的迭代器，insert函数虽然有很多重载，但返回值是完全类似的，所以接下来insert函数的返回值就不介绍了。**
4. std::vector<T>::iterator insert(std::vector<T>::const\_iterator& where, iter first, iter last)：将迭代器[first, last)添加到迭代器where指定的位置。
5. std::vector<T>::iterator insert(std::vector<T>::const\_iterator& where, size\_t count, const int& value)：在where处插入count个value。
6. std::vector<T>::iterator insert(std::vector<T>::const\_iterator& where, const T& value)：在where处插入value。
7. std::vector<T>::iterator insert(std::vector<T>::const\_iterator& where, T&& value)：在where处插入value，只不过这次以右值引用的形式插入了。

##### 删除函数

1. std::vector<T>::iterator erase(std::vector<T>::const\_iterator where)：删除容器迭代器指向的元素。**返回指向被删除元素后面的那个元素的迭代器。**
2. iterator erase(iterator first, iterator last)：删除容器中[first, last)中的元素。**返回指向被删除元素后面的那个元素的迭代器。**
3. void pvoid op\_back()：删除容器中最后一个元素。
4. clear()：删除容器中所有元素。

##### 遍历函数

1. T& at(const size\_t pos)：返回pos位置元素的引用。
2. const T& at(const size\_t pos) const：at函数的常量版本。
3. T& front()：返回首元素的引用。
4. const T& front() const：front函数的常量版本。
5. T& back()：返回尾元素的引用。
6. const T& back() const：back函数的常量版本。
7. std::vector<T>::iterator begin()：返回指向容器第一个元素的迭代器。
8. std::vector<T>::const\_iterator begin() const：begin函数的常量版本。
9. std::vector<T>::const\_iterator cbegin() const：可以主动调用的begin函数的常量版本。
10. std::vector<T>::iterator end()：返回指向容器最后一个元素的下一个元数的迭代器。

end()函数也有两个常量版本，和begin类似，就不写了。

1. std::vector<T>::reverse\_iterator rbegin()：反向迭代器，指向最后一个元素。

同样有两个常量版本。

1. reverse\_iterator rend()：反向迭代器，指向第一个元素之前的元素。

同样有两个常量版本。

##### 判断函数

1. bool empty() const：判断容器是否为空，若未空，则返回true，否则返回false。

##### 大小函数

1. size\_t size() const：返回当前容器中元素的个数。
2. size\_t capacity() const：返回当前容器不扩张所能容纳的最大元素数量。
3. size\_t max\_size() const：返回当前机器可以存储的元素数量最大值。

##### 其它函数

1. void swap(std::vector<T> & vec)：交换两个同类型容器的的数据。
2. void assign(int n, const T& x)：将容器设置为n个x。
3. void assign(const\_iterator first, const iterator last)：将当前容器的元素设置为[first, last)。

first,last都是迭代器，可以不是vector类型的迭代器，deque，list类型也可以。

1. void assign(const std::initialize\_list<T> initList)：将容器元素设置为initialize\_list的元素。
2. void res(size\_t newSize)：将容器的容量设置为newSIze。

#### deque的各种函数：

deque的各种函数与vector类似，我就不再重复一遍了。

这里只介绍vector不同的地方：

1. deque支持在容器前面插入删除，操作。也就是支持以下的三个函数
   1. void push\_front(const T& value）；
   2. void push\_front(T&& value)；
   3. void pop\_front()；

#### list的各种函数

list和deque类似，只讲一下和deque不同的部分。list都用支持在前面，后面增加，删除。

list和deque在函数上的唯一区别就是不支持随机缩影，也**就是不支持at函数**。

#### string的各种函数

string虽然也是顺序容器，但因为本质是对字符串的封装，所以和其它容器在用法上有较大区别。

1. 获取封装字符串的函数。
   1. const char\* c\_str() const：返回string对象内部的函数的指针**。注意，c\_str()函数返回的直接就是string对象内部的指针，也就是说string对象指向的对象发生了改变，返回的对象也会发生改变的。**

**代码演示一下：**

* 1. const char\* data()const：返回string对象内部的函数指针。和c\_str()函数的区别就是返回的字符串后面没有’\0’。
  2. size\_t copy(char\* const ptr, size\_t count, const size\_t off) const：

讲string对象的一部分复制到ptr数组中。

ptr表示复制到哪个数组。

count表示复制string对象的几个字符

off表示从string的哪个字符开始复制。

1. 字符串比较函数。

compare函数：这个函数重载比较多，用的时候在vs中查看一下就可以了。可以用string对象的任意部分与另一个字符串进行比较，

其它函数就和vector类似了，同样支持随机选取，支持容器末尾插入。

#### forward\_list:

和list差不多，只不过是没有size()函数，没有push\_back和pop\_back函数。

#### 关联容器的函数和顺序容器有差别的也就那么几个

在讲map，set时基本已经讲过了，这里就不单独讲了。