# C与C++11的区别

## 1.面向对象的基本概念

抽象、封装、继承、多态是面向对面的特点，本质是消息传递。c是面向过程。

### 1.1.对象

软件领域的对象是真实世界对象的一个映射。对象是由一组属性和对这组属性进行操作的一组服务构成。对象包含三个基本要素，分别是对象标识、对象状态和对象行为。

每一个对象必须有一个名字以区别于其他对象，这就是对象标识；对象状态用来描述对象的某些特征；对象行为用来封装对象所拥有的业务操作。

### 1.2.类

类是对象的抽象，反映了该类所有对象的属性、操作的抽象定义。所有对象在这个类定义的集合中，每一个对象只是类的实例。类将该实体的属性和操作封装在一起。类的属性也叫数据、状态或特征，它表现类静态的一面。类的操作也叫功能、函数或服务，它表现类动态的一面。

### 1.3.抽象

抽象是通过特定的实例抽取共同特征以后形成概念的过程。它强调主要特征，忽略次要特征。对象是对现实世界中实体的抽象，类一个是对对象的抽象，模板是对数据类型的抽象，基类是对派生类的抽象，接口是对操作的抽象。

### 1.4.封装

封装是把对象的属性、操作结合在一起，构成一个独立的对象，不允许直接访问或修改对象的属性，必须通过对象的方法。外部只能看到对象对操作的反应，而不知道对象是如何做出这一反应的。所以，封装包含两个方面的含义，一个是信息隐藏，另一个局部开放，使外界可以使用该对象，而不必关心其内部机制。

### 1.5.继承

继承表示类之间的层次关系，这种关系使得某类对象可以继承另外一类对象的属性（attributes）和操作（operations），继承又可分为单继承和多继承，单继承是子类只从一个父类继承，而多继承中的子类可以从多于一个的父类继承，Java是单继承的语言，而C++允许多继承。

### 1.6.多态

多态是指对于同一个消息、同一种调用，在不同的场合，不同的情况下，执行不同的行为。

C++多态的实现主要靠以下几个机制：

（1）重载，包括函数重载和运算符重载以及模板重载。；

（2）虚。包括虚函数和虚继承。

### 1.7.接口

所谓接口就是对操作规范的说明。接口是说明操作应该做什么( What)，没有定义如何做( How)。

C++接口包括：

（1）函数接口：自由函数的声明或实现。

（2）类接口：类向外部提供的操作的声明。

（3）接口类（接口）：C++中接口类就是具有纯虚函数的抽象类。注意：抽象类不一定是接口类。

### 1.8.消息

在系统中的对象只有对外提供服务，才能发挥自己的作用，当系统中的其他对象请求这个对象提供服务时，该对象就给予响应，并完成指定的操作。在这个过程中，其他对象要求某对象提供服务这个信息就叫作消息。

前面讲过的封装使对象具有了独立性，现在讨论的消息则使对象之间发生联系，这种联系是动态的，是根据需要发出的。为了完成整个系统的功能，对象之间只有不断发出消息进行通信，才能相互配合完成系统的职责。

一般来讲，消息应包含以下信息：提供服务的对象标识、服务标识、输入参数、响应信息。消息的接收者提供服务，它必须能够理解其他对象发出的消息，也就是说消息必须采用规定的消息协议格式。在具体的实现语言上，消息就是调用函数、子程序。对象和对象之间这种动态的联系，叫消息链接，消息链接是有方向性的。

### 1.9.结构（关系）

结构包括：继承、实现、消息连接、依赖、关联、聚合、组合。其中继承属于一般-特殊结构，依赖、关联、聚合、组合属于整体-部分结构。

**纵向包括：**

继承，指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，其中单继承叫层次结构，多继承叫网状结构。

实现，在C++中也是一种继承，指的是一个接口类与其非抽象的派生类之间的关系。

**横向包括：**

消息连接，如果对象之间的关系是动态的、可变的，并且是暂时性的，这种关系叫做消息连接关系。

依赖，是A use B，这种关系是偶然的、临时的、非常弱的，但是B类的变化会影响到A。表现在代码层面为：B是A的成员函数的参数或局部变量，或A调用了B的静态方法。如人与空气。

关联，也叫实例连接，是两个类之前的强依赖关系，是长期性的、固定的。彼此不负责对方的声明周期。分为单向关联、双向关联、自身关联；应聘者和面试官就是双向关联，双方都知道对方的存在，并且可以调用对方的公共属性及方法。双向关联在代码的表现为双方都拥有对方的一个指针，引用或者是值；单向关联类似明星和粉丝的关系，粉丝肯定知道明星，但是明星就不认识粉丝；表现出了一种引用的关系。此外还有一种自身关联。

聚合，也是一种关联，他体现的是整体与部分、拥有的关系，即A has B的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；表现在代码层面，和关联关系是一致的，只能从语义级别来区分； 如大雁与雁群。

组合，也是一种关联，他体现的是一种A contain B的关系，这种关系比聚合更强，也称为强聚合；他同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体与部分有相同的生命周期；表现在代码层面，和关联关系是一致的，只能从语义级别来区分；

### 1.10.组件

组件是软件系统可替换的、物理的组成部分，它封装了实现体（实现某个职能）．并提供了一组接口的实现方法。可以认为组件是一个封装的代码模块或大粒度的运行对的模块，也可将组件理解为具有一定功能、能够独立工作或同其他组件组合起来协同工作的对象。

对于组件，应当按可复用的要求进行设计、实现、打包、编写文档。组件应当是内聚的，并具有相当稳定的公开的接口。

为了使组件更切合实际、更有效地被复用，组件应当具有“可变性”（variability），以提高其通用性。组件应向复用者提供一些公共“特性”，另一方面还要提供可变的“特性”。针对不同的应用系统，只需对其可变部分进行适当的调节，复用者要根据复用的具体需要，改造组件的可变“特性”，即“客户化”。

### 1.11.模式

模式是一条由三部分组成的规则，它表示了一个特定环境、一个问题和一个解决方案之间的关系。每一个模式描述了一个不断重复发生的问题，以及该问题的解决方案。这样就能一次又一次地使用该方案而不必做重复劳动。

将设计模式引入软件设计和开发过程的目的在于充分利用已有的软件开发经验，这是因为设计模式通常是对于某一类软件设计问题的可重用的解决方案。

设计模式使得人们可以更加简单和方便地去复用成功的软件设计和体系结构，从而能够帮助设计者更快更好地完成系统设计。

### 1.12.复用

软件复用是指将已有的软件及其有效成分用于构造新的软件或系统。组件技术是软件复用实现的关键。

### 1.13.永久对象

永久对象指生存周期可以超越程序的执行时间而长期存在的对象，又叫持久对象。一般情况下，对象的生命周期不能超过程序的运行时间，一旦程序运行停止，所有对象都会立即消失，下次运行程序，再生成对象；或者在上次结束时，把所有的对象都用文件或数据库保存起来，下次运行再恢复出来，虽然从表面看好像延长了对象的生命周期，但存在以下缺陷：首先要编制对象与文件或数据库之间要进行格式转换操作，其次是对象有一段时间是以普通数据的身份而存在的（不是面向对象的）。

利用永久对象的概念，无须额外编制任何程序，只要在程序中对某个对象声明为永久对象，则系统会自动完成它的存储、转换、恢复等一系列问题，对象和数据库之间直接交互，对象从库中提取出来放到内存，和原来的一摸一样，有相同的属性和操作，能和其他对象直接交互，如果有多个对象，它们之间的继承和关联关系也和原来一样，能够使对象在不同的程序之间实现共享，这将大大减轻编程的工作量。

永久对象的实现要求有较高的技术支持，其中重要的技术如存储和管理永久对象的对象管理系统，同时还需解决对象的共享、并发、一致性等问题

### 1.14.主动对象与被动对象

主动对象是指对象的某个操作不需要消息驱动，能够主动执行。主动对象的行为被设计成一个任务，用来描述业务领域识别的任务和分析时识别的任务。它封装了一组属性和操作，但其中至少有一个操作，不需要接收参数，就可以主动执行，并能完成相应的任务。实现时要求能并发，并能主动执行，通常是用进程或线程来实现的。当然，主动对象的一些操作，也可以是在接收消息之后才执行的一般操作。就是设计用来刻画对象的这种行为，主动对象的行为被设计成一个任务，用来描述业务领域识别的任务和分析时识别的任务。它

被动对象是指对象的每个操作都是在消息驱动下被动地执行。向对象发一个消息，它就响应这个消息，执行被请求的服务，否则，它的操作就不能被执行，通常是用函数、例程或过程来实现。

## 2.头文件

c++的有些头文件是由模板编写而成的，与c的头文件有差别，c是c++的子集，c++是c的超集，如果c头文件与c++头文件都要使用的时候，c头文件放在c++头文件前面。否则可能出现一些问题。

## 3.标准输入输出

标准输出cout<< 值，标准输入cin>>变量

endl、ends、\n、fllush

endl=\n+fllush，注意c中\n会刷新缓冲区，但c++中不会

## 4.名称空间

### 4.1.命名空间

名称空间是用于解决相同命名的对象在使用时发成冲突的问题。原理就是将变量，函数等的作用域限定在某个命名空间中。

C++中定义名称空间的基本格式为：

namespace 名称空间名

{

变量类型 变量；

函数返回类型 函数原型；

}

c++中使用命名空间的实体的基本格式有：

①作用域限定符：空间名::实体名//每次在名称空间外使用实体都需要使用作用域限定符

②using声明机制：using 空间名::实体名//使得某个实体可见

③using编译指令：using namespace 空间名//使得所有实体可见

定义在名称空间中的变量或者函数都称为实体。名称空间中的实体作用域是全局的, 并不意味着其可见域是全局的。如果不使用作用域限定符和using机制，抛开名称空间嵌套和内部屏蔽（局部变量屏蔽全局变量）的情况，实体的可见域是从实体创建到该名称空间结束，在名称空间外，该实体是不可见的。因而在命名空间外使用实体时候，必须用作用域限定符使其可见：

空间名::实体名

如果不希望在每次使用名称空间中实体时都使用作用域限定符，可使用using声明机制扩展其可见域，using声明的基本格式为：

using 名称空间::实体名; //如果是函数只需声明函数名

如“using A::dispA;”，至于using声明语句将该实体的可见域扩展到什么程度，这取决于using语句的书写位置，换言之，这取决于using语句的可见域。

using声明机制使得某个空间中的特定实体可见，using编译指令比using声明更进一步，使得名称空间中的所有实体都可见，不再需要作用域限定符：

using namespace 名称空间名;

注意：

如果using声明使用不当，很容易引起多重声明错误,比如:

已经定义了全局函数disp，却还使用全局using声明语句“using A::disp”，假设没有屏蔽发生，那么调用disp()时，编译器不确定是全局函数版本还是A::disp()，引发多重声明错误。变量名同样存在这种问题，假设有两个名称空间A、B中都定义了int型变量num，在程序的某处需要使用num，如果写出如下代码：

using A::num;

using B::num;

num=5;

因此，应合理使用using声明机制。c++推荐使用using声明机制。

using声明机制应该在名称空间空间内使用，如果在全局空间中使用，则会污染全局空间中的其他空间。

### 4.2.匿名空间

C++中定义匿名空间的基本格式为：

namespace

{

变量类型 变量；

函数返回类型 函数原型；

}

编译器在内部会为这个命名空间生成一个唯一的名字，而且还会为这个匿名的命名空间生成一条using指令。所以上面的代码在效果上等同于：

namespace \_\_UNIQUE\_NAME\_

{

变量类型 变量；

函数返回类型 函数原型；

}

using namespace \_\_UNIQUE\_NAME\_; //这样就相当于把匿名空间的实体导入了全局空间之中。

在匿名命名空间中声明的名称也将被编译器转换，与编译器为这个匿名命名空间生成的唯一内部名称(即这里的\_\_UNIQUE\_NAME\_)绑定在一起。还有一点很重要，就是这些名称具有internal链接属性，这和声明为static的全局名称的链接属性是相同的，即名称的作用域被限制在当前文件中，无法通过在另外的文件中使用extern声明来进行链接。如果不提倡使用全局static声明一个名称拥有internal链接属性，则匿名命名空间可以作为一种更好的达到相同效果的方法。

注意:命名空间都是具有external 连接属性的,只是匿名的命名空间产生的\_\_UNIQUE\_NAME\_\_在别的文件中无法得到,这个唯一的名字是不可见的.

C++ 新的标准中提倡使用匿名命名空间,而不推荐使用static,因为static用在不同的地方,涵义不同,容易造成混淆.另外,static不能修饰class。

匿名空间的作用仅仅相当于在全局变量之前加了static，这样这些成员不能在其他文件中用extern扩展作用域，可以这么认为定义在匿名空间就相当于全局作用域（文件作用域），但是都具有static属性。也就是命名空间具有external外链属性，而匿名空间具有internal内链属性。

访问全局空间的成员用

::实体名

匿名空间的成员无法通过作用域限定符来访问。

注意：（1）同一文件内的多个匿名空间是同一个空间。

（2）全局空间是匿名空间的上一层空间。匿名空间和命名空间是对等的，只不过由于编译器会自动导入匿名空间的所有成员，所以匿名空间的成员是内链的，且时全局可用的。

（3）宏定义 不属于任何空间。

### 4.3.include与namespace的关系

#include的作用是帮助编译（这个编译是狭义上的编译，不包含链接）过程的，包含进来的各种声明是告诉编译器这个名字是合法的，可以在文件中出现的，这样编译器就将各个cpp文件编译成了多个目标模块。到此为止，两个fun()还相安无事的呆在自己的模块里。下面就是链接了，链接的过程是为每个名字找到对应的定义，也就是对应的内存地址，这个时候就出问题了，在我们的项目中出现了两个fun()的定义，那使用时该用哪个呢？这显然是个问题，所以就出现了一个连接错误，说fun()重复定义了，不知道用哪个了。而namespace就是这个时候出现的，它实现了一种为名字找定义的机制（using声明并不会复制什么代码到你的文件里！），即链接时只在同一个锅里找定义，其他锅里有没有不管。总而言之，#include管编译，告诉编译器这些声明是合法的；namespace管链接，告诉编译器去哪里找定义，二者谁也替代不了谁！

## 5.const关键字

### 5.1.const与define

常量位于文字常量区，分为数值型常量和字符型常量。数值型常量没有空间，也就没有地址，而字符型常量由于不能被指令直接操作,因此必须先分佩空间。在定义常量指针的时候，实质是用字符型常量所在地址。

只要是常量就不能直接取地址。

宏定义与const的区别：宏定义发生在预编译阶段，没有语法检测，仅仅是简单的字符串替换；const 发生在编译阶段，有语法检测，实质仍然是定义一个变量。

### **5.2.const的作用**

const是一个关键字，起保护，防止意外的变动的作用！可以修饰变量，指针变量，参数，返回值,甚至函数体。const可以提高程序的健壮性，你只管用到你想用的任何地方。

（1）const修饰传入参数（引用，指针）

①如果输入参数是指针型的，用const修饰可以防止指针或者指针内容被意外修改，对于常量指针，c++11要求必须使用const修饰。

②如果参数采用值传递的方式，无需const，因为函数自动产生临时变量复制该参数。

③非内部基本数据类型的参数，需要临时对象复制参数，而临时对象的构造，析构，复制较为费时，因此建议采用const+引用方式传递非内部基本数据类型。而内部基本数据类型无需引用传递。

④对传入参数要做修改时有两种方法，一是传指针，二是传引用，如果是对指针修改则需传入二级指针或者指针的引用。

（2）const修饰函数返回值（引用，指针）。

①函数返回const指针，表示该指针不能被改动，只能把该指针赋给const修饰的同类型指针变量。（很少使用）

②函数返回值为值传递，函数会把返回值赋给外部临时变量，用const无效！不管是内部还是非内部数据类型。

③函数采用引用方式返回的场合不多，只出现在类的赋值函数、运算符重载函数中，以及成员函数中，目的是为了实现链式表达。

④函数返回const 引用，很少使用。

（3）const修饰成员函数。任何不修改数据成员的函数都应该声明为const类型，如果const成员函数修改了数据成员或者调用了其他函数修改数据成员，编译器都将报错！如果此时确实需要返回一个指针或者引用则需要使用const关键字，以防止通过指针或引用来修改数据成员。

（4）const定义变量或对象，表示该变量不能被修改，当定义常对象时候只能调用const成员函数。

定义常量：

const 数据类型 常量名=初值 //常量定义的时候必须初始化,位于文字常量区

定义常量指针：

const 数据类型 \*指针名=&常量名//指针所指空间可变，但所指空间的内容不可变。

定义指针常量：

数据类型 \* const指针名= 地址//指针所指空间不可变，但所指空间的内容可变

定义指向常量的指针常量：

const 数据类型 \* const指针名= &常量名 //指针所指空间不可变，但所指空间的内容也不可变

c++中所有的常量定义都需加上const关键字

## 6.c++申请堆空间

**为基本数据类型数据：**

开辟单个地址单元，可以同时赋予指定初值：

指针=new 基本数据类型 //空间不会清零

指针=new 基本数据类型（）//空间会清零

指针=new 基本数据类型（初值）//空间初始化为指定值

delete 指针

开辟多个地址单元，不能执行赋予指定初值

指针=new 基本数据类型[元数个数]//开辟空间不清零

指针=new 基本数据类型[元数个数]() //开辟空间全部清零

delete []指针名

**为类对象：**

开辟单个地址单元，可以同时赋予指定初值：

指针=new类名 //开辟空间不清零, 但由调用的构造函数决定了初始化赋值操作

指针=new类名（）//开辟空间全部清零，再由调用的构造函数决定了初始化赋值操作

指针=new类名（构造参数）//new开辟空间未初始化，但由构造函数进行初始化赋值，不能用括号

delete 指针

开辟多个地址单元，不能执行赋予指定初值

指针=new 类名[元数个数]//开辟连续空间不清零，但由调用的构造函数决定了初始化赋值操作

指针=new 类名[元数个数]() //开辟空间全部清零，再由调用的构造函数决定了初始化赋值操作

delete []指针名

区别：

（1）malloc和free是c的标准库函数，new和delete是运算符

（2）malloc只负责开辟空间，返回值为void\*，不负责转类型，需要强制转换类型，并需要显式指出空间大小,不会自动清理空间数据，new开辟空间的时候，会自动转换为所需类型，自动计算空间大小,同样也不会清理空间。

（3）new的效率高于malloc

（4）但注意，指针本身是放在栈区的，指向堆区。

## 7.使用栈空间

**基本数据类型**

开辟单个地址单元

基本数据类型 变量名; //开辟空间不初始化

基本数据类型 变量名=初值//开辟空间初始化为指定值

基本数据类型 变量名（初值）//开辟空间初始化为指定值

开辟多个地址单元，

基本数据类型 数组名[元数个数]//开辟空间不初始化

基本数据类型 数组名[元数个数] ={初始化列表}//开辟空间全部清零

**类对象：**

开辟单个地址单元

类名 对象名//开辟空间不清零, 但由调用的构造函数决定了初始化赋值操作

类名 对象名（构造参数）//开辟空间全部清零，再由调用的构造函数决定了初始化赋值操作

类名 对象名=类名（构造参数）//开辟空间未初始化，由拷贝构造函数进行初始化赋值

开辟多个地址单元

类名 对象名[元数个数]//开辟空间不清零，但由调用的构造函数决定了初始化赋值操作

类名 对象名[元数个数]={初始化列表}//开辟空间不清零，由构造函数决定了初始化赋值操作

## 8.c++支持引用

c++中引用而c中没有

数据类型 &别名=变量名 //变量必须存在

const数据类型 &别名=常量 //对常量的引用必须使用常引用

引用与指针的区别：

（1）引用必须初始化，指针不用

（2）引用一旦定义就不可改变，而指针指向可改变。

（3）对引用的操作就是对原对象的操作，

（4）引用作为函数形参，对形参的操作就是对实参的操作，调用函数时不会开辟形参空间，可以节约传值的时间和空间开销。

指针作为形参，对指针的操作不会改变实参。对指针所指内容的操作会改变实参。但参数传递时会开辟一个指向指针的空间。实质仍是传值。

（5）引用作为函数返回值，从而可以对函数赋值。

int a[3]={1,2,3};

int &func(int idx)

{

return a[idx];

}

func(0)=10

（6）同函数不能返回指向局部变量的指针一样，函数也不能返回局部变量的引用。也不能返回一个指向堆空间变量的引用。

（7）传引用的作用相当于传指针，传指针的引用相当于传二级指针。到底参数传递时要不要使用指针或者引用？取决于传递的参数大小和是否需要在函数中对实参做改变，如果实参很大，或者需要在函数中中实参要改变，则需要使用指针或引用。

（8）引用和指针都有地址的概念

（9）指针可以赋值为NULL，引用不可以。

## 9.c++支持函数重载

函数名相同而参数不同时，c不能区分，但c++会自动匹配。其原理是：名字改编。当函数名相同时根据参数个数，类型和顺序来进行函数改编。不能只依靠返回值来重载。

如果想在c++中以c的方式调用同名函数则需在函数实现前加上extern “C” ，实现c和c++的混合编程，但不能对多个同名函数都以c方式调用，否则会发生冲突。

#ifdef #endif 和#ifndef #endif 避免重复的宏定义冲突，形式如下：

#ifdef \_cplusplus

extern “C”{

#endif

函数定义

#ifdef \_\_cplusplus

}

#endif

函数实现

## 10.变量类型的强制转换

有时我们希望显式地将对象强制类型转换成另外一种类型。例如，如果想在下面的代码中执行浮点数除法：

　　int i, j;

　　double slope = i / j;

就要使用某种方法将i和/或j显式地转换成double，这种方法称作强制类型转换。

c中的强制转换方式为：（类型名）变量名或者类型名（变量名）,这种强制转换功能最强大，但，是匿名的，不安全的。

c++中的强制转换：命名的强制类型转换

一个命名的强制类型转换具有如下形式：

cast\_name<type>(expr);

其中，type是转换成的目标类型而expr是要转换的值。如果type是引用类型，则结果是左值，cast-name是static\_cast, dynamic\_cast, const\_cast和reinterpret\_cast中的一种。dynamic\_cast支持运行时类型识别。cast-name指定了执行的是哪种转换。

注意：

（1）C++中任意类型的指针可以隐式转为void\*，但void\*不可以隐式转换为其他类型，

（2）C++中派生类指针可以隐式的转为基类指针。

### 10.1.static\_cast

static\_cast运算符把非const的expr转为type类型，但没有运行时类型检查来保证转换的安全性。并且要求源对象不能有const修饰。主要用于：

①用于基类和派生类的指针或引用之间的转换。

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；

进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。

②用于基本数据类型之间的转换，这种转换的安全性也要开发人员来保证。

③把空指针转换成目标类型的空指针。

④把任何类型强转void类型（丢弃表达式的值），或者把任意类型指针转为void\*及其相反过程。

注意：static\_cast不能转换掉expression的const、volatile属性。

例如，通过将一个运算对象强制转换成double类型就能使表达式执行浮点数除法：

double slope = static\_cast<double>(j) / i;

当需要把一个较大的算术类型赋值给较小的类型时，static\_cast非常有用。此时，强制类型转换告诉程序的读者和编译器：我们知道并且不在乎潜在的精度损失。一般来说，如果编译器发现一个的算术类型试图赋值给较小的类型，就会给出警告信息；但是当我们执行了显式的类型转换后，警告信息就会被关闭了。

static\_cast对于编译器无法自动执行的类型转换也非常有用。例如，我们可以使用static\_cast找回存在于void\*指针中的值：

void \*p = &d;　　//正确：任何非常量对象的地址都能存入void\*

　　double \*dp = static\_cast<double\*>(p); //正确：将void\*转换回初始的指针类型

当我们把指针存放在void\*中，并且使用static\_cast将其强制转换回原来的类型时，应该确保指针的值保持不变。也就是说，强制转换的结果将与原始的地址值相等，因此我们必须确保转换后所得的类型就是指针所指的类型。类型一旦不符，将产生未定义的后果。

### 10.2.const\_cast

const\_cast运算符唯一且只能用于增加或删除**指针或引用**的底层const属性，即将常量改为变量或将变量变为常量，不能改变数据型。

const char \*pc;

char \*p = const\_cast<char\*> (pc);　　//正确：但是通过p写值是未定义的行为

对于将常量对象转化为非常量对象的行为，我们一般称之为”去掉const性质（cast away the const）“。一旦我们去掉了某个对象的const性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，再使用const\_cast执行写操作就会产生未定义的后果。

　只有 const\_cast能改变表达式的常量属性，使用其他形式的命名强制类型转换改变表达式的常量属性都将引发编译器错误。同样地，也不能用const\_cast改变表达式的类型：

const char \*cp;

char\* q = static\_cast<char\*> (cp); //错误：static\_cast不能转换掉const性质

static\_cast<string> (cp); //正确，字符串字面值转换成string类型

const\_cast<string>(cp); //错误，const\_cast只改变常量属性

总结来说：const\_cast只能改变底层const，即它只能改变对象或引用的const属性，不能改变对象的其他属性。const\_cast即可以添加const性质，也可以删除const性质。

### 10.3.reinterpret\_cast

reinterpret\_cast在源对象的二进制位重新解释，但不保证安全，它会产生一个新的值，这个值会有与原始参数（expressoin）有完全相同的比特位。用于**无关类型**（非继承）的指针与指针，引用与引用，指针与整数之间的相互转换。

举个例子，假设有如下的转换：

int \*ip;

char \*pc = reinterpret\_cast<char\*>(ip);

我们必须牢记pc所指的真实对象是int而非字符，如果把pc当成一个普通的字符指针使用就可能在运行时发生错误。例如：

string str(po);

可能导致异常的运行时行为。

使用reinterpret\_cast是非常危险的，用pc初始化的例子很好地证明了这一点。其中的关键问题是类型改变了，但编译器没有给出任何警告或者错误的提示信息。当我们用一个int的地址初始化pc时，由于显式地声称这种转换合法，所以编译器不会发出任何警告或错误信息。接下来再使用pc时就会认定它的值是char\*类型，编译器没法知道它实际存放的是指向int的指针。

### 10.4.dynamic\_cast

dynamic\_cast只用于基类与派生类的指针或引用之间的转换，尤其是有虚函数时。

dynamic\_cast 主要用于下行转换，在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的。

dynamic\_cast 会检查基类指针是否真正指向派生类对象，如果不是，则会失败。

对指针进行dynamic\_cast，失败返回null，成功返回正常cast后的对象指针；

对引用进行dynamic\_cast，失败抛出一个异常，成功返回正常cast后的对象引用。

## 11.c++支持bool类型（c99又新增了bool）

c99之前的c语言是没有bool类型的

c++中有bool类型，取值为true和 false，代表0和非零（正数和负数都为真）。其大小为1个字节。

## 12.c++的函数可带默认参数

c++中如果函数的形参有初值，调用函数如果不给定特值则使用初值参与函数调用。如果某个参数是默认参数，那么它后面的参数必须都是默认参数，也就是默认参数的设置必须要从右到左的顺序进行。而函数调用时形参必须从左向右给定，如果有一个参数省略，则后面的参数都必须省略。如果有函数重载时，要注意避免二义性的问题。比如定义了 add（int x，int y）与int add（int x,int y=10,int z=10），此时发生函数调用add（x,y）就会发生function ambiguous的错误。

注意：c++和c中，函数声明和函数实现只能在声明中带默认参数。在定义时不能带。

## 13.inline函数

### 13.1.普通内联函数

为了解决程序中函数调用效率问题。

c中可以使用带参数的宏来处理，c++ 中可以在函数之前加上inline关键字。

函数是一种更高级的抽象。它的引入使得编程者只关心函数的功能和使用方法，而不必关心函数功能的具体实现；函数的引入可以减少程序的目标代码，实现程序代码和数据的共享。但是，函数调用也会带来降低效率的问题，因为调用函数实际上将程序执行顺序转移到函数所存放在内存中某个地址，将函数的程序内容执行完后，再返回到转去执行该函数前的地方。这种转移操作要求在转去前要保护现场并记忆执行的地址，转回后先要恢复现场，并按原来保存地址继续执行。因此，函数调用要有一定的时间和空间方面的开销，于是将影响其效率。特别是对于一些函数体代码不是很大，但又频繁地被调用的函数来讲，解决其效率问题更为重要。引入内联函数实际上就是为了解决这一问题。

在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来进行**替换**。显然，这种做法不会产生转去转回的问题，但是由于在编译时将函数休中的代码被替代到程序中，因此会增加目标程序代码量，进而增加空间开销，而在时间代销上不象函数调用时那么大，可见它是以目标代码的增加为代价来换取时间的节省。

定义内联函数的方法很简单，只要在函数定义的头前加上关键字inline即可。内联函数的定义方法与一般函数一样。如：

inline int add\_int (int x, int y, int z)

{

return x+y+z;

}

在程序中，调用其函数时，该函数在编译时被替代，而不是像一般函数那样是在运行时被调用。

使用内联函数应注意的事项

内联函数具有一般函数的特性，它与一般函数所不同之处公在于函数调用的处理。一般函数进行调用时，要将程序执行权转到被调用函数中，然后再返回到调用它的函数中，而内联函数在调用时，是将调用表达式用内联函数的函数体来替换。在使用内联函数时，(尽量保证只有顺序语句)应注意如下几点：

（1）在内联函数内不允许用循环语句和开关语句以及条件语句。

（2）普通内联函数的必须先定义后调用，不能先声明内联函数原型再调用，再后定义。因为 inline是一种“用于实现的关键字”，而不是一种“用于声明的关键字”

（3）内联函数应该简洁，只有几个语句，如果语句较多，不适合于定义为内联函数。

（4）在一个文件中定义的内联函数不能在另一个文件中使用。它们通常放在头文件中共享。

（5）inline提示的函数不满足内联函数要求时，编译器会忽略。

（6）函数声明和定义时都应应加inline。

此外inline可以解决多次重复定义的问题。

### 13.2.内联成员函数

在类中定义的成员函数，当函数中没有选择和开关，循环语句时，编译器将其默认为inline函数。

成员函数也可用inline关键字显式指明为内联函数，也可以在类内声明，类外实现，此时声明不应加上关键字inline，但实现必须加，且必须在同一个源文件中(否则会找不到定义)。而普通成员函数则只需在定义以及声明时使用一个inline关键字。

## 14.避免重复定义变量或函数

第一种方法是c中的，在头文件中只进行函数声明，不做定义。

第二种方法是在函数前加关键字inline

## 15.预处理指令

#define 宏定义

#undef 取消宏

#include 文本包含

#ifdef 如果宏被定义就进行编译

#ifndef 如果宏未被定义就进行编译

#endif 结束编译块的控制

#if 表达式非零就对代码进行编译

#else 作为其他预处理的剩余选项进行编译

#elif 这是一种#else和#if的组合选项

#line 改变当前的行数和文件名称

#error 输出一个错误信息

#pragma 为编译程序提供非常规的控制流信息，可跟once，message等许多参数。

### 15.1.使用条件编译指令

常用方式为：#ifndef,#define, #include，#endif.

优点：

（1） 不光可以保证同一个文件不被包含多次，也能保证内容完全相同的两个文件（或者代码片段）不被同时包含。

（2）受C/C++语言标准的支持，不受编译器的任何限制

缺点：

（1）如果不同头文件中的宏名不小心“撞车”，可能就会导致你看到头文件明明存在，编译器却硬说找不到声明的状况——这种情况有时非常让人抓狂。

（2）由于编译器每次都需要打开头文件才能判定是否有重复定义，因此在编译大型项目时，ifndef会使得编译时间相对较长，因此一些编译器逐渐开始支持#pragma once的方式。

### 15.2.pragma once指令

同一个文件不会被包含多次。注意这里所说的“同一个文件”是指物理上的一个文件，而不是指内容相同的两个文件。你无法对一个头文件中的一段代码作pragma once声明，而只能针对文件。

优点：

（1）你不必再费劲想个宏名了，当然也就不会出现宏名碰撞引发的奇怪问题。

（2）大型项目的编译速度也因此提高了一些。

缺点：

（1）就是如果某个头文件有多份拷贝，本方法不能保证他们不被重复包含。当然，相比宏名碰撞引发的“找不到声明”的问题，这种重复包含很容易被发现并修正。

（2）而#pragma once方式却不受一些较老版本的编译器支持，一些支持了的编译器又打算去掉它，所以它的兼容性可能不够好。

## 16.NULL、nullptr、0、\0、‘\0’、”\0”、“”

在C语言中，我们使用NULL表示空指针，NULL通常被定义为如下：

#define NULL ((void \*)0)

也就是说NULL实际上是一个void \*的指针，然后吧void \*指针赋值给某指针的时候，隐式转换成相应的类型。而如果换做一个C++编译器来编译的话是要出错的，因为C++是强类型的，void \*不能隐式的转为其他类型，所以通常情况下，编译器提供的头文件会这样定义NULL：

#ifdef \_\_cplusplus

#define NULL 0

#else

#define NULL ((void \*)0)

#endif

这样在C++98中我们用0表示空指针。但这在函数重载时会带来问题，如：

func(int \*a,int \*b);

func(int \*a,int b);

func(b,NULL)我们本想调用func(int \*a,int \*b);但由于c++中NULL就是0，所以会产生ambiguous的错误。

c++11中我们用nullptr表示空指针。就不会出现c++98中的问题了。nullptr是std::nullptr\_t类型的（constexpr）变量（字面量）。std::nullptr\_t可以显式或隐式地转换为任何指针（包括类的成员函数指针），但不能显式或隐式地转换为任何其他类型。

注意：

（1）std::nullptr\_t的对象不能用于算术表达式，即，不能+、-、\*等；

（2）std::nullptr\_t可以用于二元关系表达式，两个std::nullptr\_t类型的对象，进行==、<=、>=运算都会返回true，其他的关系运算返回false；不可以用于一元表达式

（3）nullptr不能进行&（取地址）运算。

### 16..两种方式混合

#pragma once

#ifndef xxxxx

#define xxxxx

#include<xxxxx>

#else

#include”xxxxx”

#endif

## 17.c++的for循环

C++中新增了几种便利方式方式，以vector<int>vec为例子

for(i=0;i<max;i++){……}//传统下标

for(vector<int>::iteratoe it=vec.begin();it!=vec.end();++it) {……}//传统迭代器

for(int elem:vec ){……}//简化遍历

for\_each(vec.begin(),vec.end(),function);//算法库

## 18.抽象类与接口的区别

含有纯虚函数的类叫做接口类，提供一个规范接口的作用，可以创建指针，但是不可创建对象，接口类是一种抽象类。

只有protected构造函数的类是另一种抽象类。不能直接创建对象，但可以通过静态成员函数创建对象。

## 19.constexpr

constexpr于C++11 中引入。它表示常量表达式。与 const 相同，它可应用于变量，因此如果任何代码试图修改该值，均将引发编译器错误。与 const 不同，constexpr 也可应用于函数和类构造函数。 constexpr 指示值或返回值是常数，并且如果可能，将在编译时而不是运行时计算值或返回值。

### 19.1.修饰变量

const 和 constexpr 变量之间的主要区别在于：const 变量的初始化可以延迟到运行时，而 constexpr 变量必须在编译时进行初始化。声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化。

一般来说，如果你认定变量是一个常量表达式，那就把它声明成为constexpr类型。

### 19.2.修饰普通函数

constexpr函数是在编译时就可以计算其返回值的函数。constexpr函数要求所定义的函数足够简单以使得编译时就可以计算其结果。当其参数全为 constexpr 值并且在编译时使用代码且需要其返回值时，它会生成编译时常量。使用非constexpr 参数调用时，或编译时不需要其值时，它将与一般函数一样，在运行时生成一个值。

### 19.3.修饰构造函数

constexpr还能修饰类的构造函数，即保证传递给该构造函数的所有参数都是constexpr，那么产生的对象的所有成员都是constexpr，

该对象也是constexpr对象了，可用于只使用constexpr的场合。

注意：constexpr构造函数的函数体必须为空，所有成员变量的初始化都放到初始化列表中。

### 19.3.修饰和指针

还记得const与指针的规则吗？如果关键字const出现在星号左边，表示被指物是常量；如果出现在星号右边，表示指针本身是常量；如果出现在星号两边，表示被指物和指针两者都是常量。

与const不同，在constexpr声明中如果定义了一个指针，限定符constexpr仅对指针有效，与指针所指对象无关。

与其它常量指针类似，const指针既可以指向常量也可以指向一个非常量：

### 19.4 . constexpr优点

（1）constexpr是一种很强的约束，更好的保证程序的正确定语义不被破坏；

（2）编译器可以对constexper代码进行非常大的优化，如：将用到的constexpr表达式直接替换成结果；

（3）相比宏来说没有额外的开销。

## 20.c++异常处理

相比于断言适用于排除逻辑上不可能存在的状态，异常通常是用于逻辑上可能发生的错误。在C++中，我们看到了一套完整的不同于C的异常处理系统。通过这套异常处理系统，C++拥有了远比C强大的异常处理功能。特别的在类的构造函数和析构函数由于没有返回值，用异常表示构造是否成功就尤为重要。

### 20.1.抛出异常（C++98/11）

C++ 异常处理的流程，具体为：

抛出（Throw）--> 检测（Try） --> 捕获（Catch）

异常必须显式地抛出，才能被检测和捕获到；如果没有显式的抛出，即使有异常也检测不到。

抛出异常（也称为抛弃异常）即检测是否产生异常，在C++中，其采用throw语句来实现，如果检测到产生异常，则抛出异常。该语句的格式为：

throw exceptionData表达式;

exceptionData表达式是“异常数据”的意思，它可以包含任意的信息，完全有程序员决定。exceptionData 可以是 int、float、bool 等基本类型，也可以是指针、数组、字符串、结构体、类等聚合类型。由于C++使用数据的类型来区分不同的异常，因此在判断异常时，throw语句中的表达式的值就没有实际意义，而表达式的类型就特别重要。

如果在try语句块的程序段中（包括在其中调用的函数中）发现了异常，且抛出了该异常，则这个异常就可以被try语句块后的某个catch语句所捕获并处理，捕获和处理的条件是被抛弃的异常的类型与catch语句的异常类型相匹配。

#include<iostream.h> //包含头文件

#include<stdlib.h>

double fuc(double x, double y) //定义函数

{

if(y==0)

{

throw y; //除数为0，抛出异常

}

return x/y; //否则返回两个数的商

}

void main()

{

double res;

try //定义异常

{

res=fuc(2,3);

cout<<"The result of x/y is : "<<res<<endl;

res=fuc(4,0); //出现异常

}

catch(double ) //捕获并处理异常

{

cerr<<"error of dividing zero.\n";

exit(1); //异常退出程序

}

}

（1）程序接受到throw语句后就会自动调用析构器，把该域（try后的括号内）对象clean up，然后再进入catch语句（如果在循环体中就退出循环）。这种机制会引起一些致命的错误，比如，当“类”有指针成员变量时，在 “类的构造函数”中的throw语句引起的退出，会导致这个指针所指向的对象没有被析构。把指针改为类就行了，比如模板类来代替指针，在模板类的内部设置一个析构函数。c++11可使用智能指针来解决。

（2）语句“throw”抛出一个未被捕获的异常，这时进入终止函数。terminate().然后调用abort()函数，终止执行。

### 20.2.异常捕获（C++98/11）

#### 20.2.1.语法形式

try

{

//程序中抛出异常

throw value;

}

catch(valuetype v)

{

//异常处理程序段

}

语法小结：throw抛出值，catch捕获，当然，throw必须在“try语句块”中才有效。

#### 20.2.2.深入catch

一般的catch出现的形式是：

try

{

//可能有异常的代码

}

catch(except1&)

{

//异常处理程序

}

catch(except2&)

{

//异常处理程序

}

catch(...)//接受所有异常

{

//异常处理程序

}

一般都写成引用（except1&），原因很简单，效率。

#### 20.2.3.其他用法

（1）try一个函数体，形式如下

void fun(type1,type2) try //try放在函数体后

{

函数体

}

catch(type X)

{

//异常处理程序

}

这个用法的效果就相当于:

void fun()

{

try

{

函数定义

}

}

catch(type X)

{

//异常处理程序

}

问题：抛出异常，但是没有被捕获怎么办？（注意没有Java类似的finally语句）

答：调用terminate()，terminate调用terminate\_handler。terminate\_handle默认为abort（），可以用set\_terminate()设置terminate\_handle。

#### 20.3.4.catch(…)

C++异常处理模型的try catch中catch关键字是用来定义catch block的，它后面带一个参数，用来与异常对象的数据类型进行匹配。注意catch关键字只能定义一个参数，因此每个catch block只能是一种数据类型的异常对象的错误处理模块。如果要想使一个catch block能抓获多种数据类型的异常对象的话，怎么办？C++标准中定义了一种特殊的catch用法，那就是” catch(…)”。

catch(..)能匹配成功所有的数据类型的异常对象，包括C++语言提供所有的原生数据类型的异常对象，如int、double，还有char\*、int\*这样的指针类型，另外还有数组类型的异常对象。同时也包括所有自定义 的抽象数据类型。

#### 20.3.5.nothrow

在C中，使用malloc等分配内存的函数时，一定要检查其返回值是否为”空指针”，并以此作为检查内存操作是否成功的依据，这种Test-for-NULL代码形式是一种良好的编程习惯，也是编写可靠程序所必需的。

在C++中new在申请内存失败时默认会抛出一个std::bad\_alloc 异常。如果出现这个异常，那就意味着内存耗尽，或者有其它原因导致内存分配失败。所以，按照C++标准，如果想检查new是否成功，则应该通过try-catch捕捉异常。但有些编译器不支持try-catch。C++标准化委员会不想遗弃这些Test-for-NULL的代码，所以他们提供了operator new的另一种可选形式nothrow，用以提供传统的Failure-yields-NULL行为。

当使用new申请一块内存失败时，抛出异常std::bad\_alloc是C++标准中规定的标准行为，所以使用

try

{

p = new int[size];

}

catch(std::bad\_alloc)

{

…

}

来处理new分配内存失败的异常。

struct nothrow\_t

{

// placement new tag type to suppress exceptions

};

extern const nothrow\_t nothrow; // constant for placement new tag

其中，nothrow\_t通常是一个空结构，其唯一目的是提供编译器一个可根据重载规则识别具体调用的类型。用户一般简单地使用"new(std::nothrow) 类型"(nothrow是一个nothrow\_t类型的常量)来调用这个placement new操作符。它与标准new的区别是，new在分配内存失败时会抛出异常，而"new(std::nothrow)"在分配内存失败时会返回一个空指针。

new operator分配内存失败后，缺省的行为不是返回空指针，而是抛出异常std::bad\_alloc。所以判断返回值是否为NULL没有任何意义。

int \*p1= new int //分配内存失败，抛出出std::bad\_alloc异常，可用try-catch来捕获和处理异常

int \*p2 = new（std::nothrow）int //分配内存失败，不抛出异常，返回nullptr，可用Test-for-NULL来处理。

### 20.3.异常规范

#### 20.3.1 用throw作异常规范（C++98）

throw 关键字除了可以用在函数体中抛出异常，还可以用在函数头和函数体之间，指明当前函数能够抛出的异常类型，这称为异常规范（Exception specification），有些教程也称为异常指示符或异常列表。请看下面的例子：

double func (char param) throw (int);

这条语句声明了一个名为 func 的函数，它的返回值类型为 double，有一个 char 类型的参数，并且只能抛出 int 类型的异常。如果抛出其他类型的异常，try 将无法捕获，只能终止程序。

如果函数会抛出多种类型的异常，那么可以用逗号隔开：

double func (char param) throw (int, char, exception);

如果函数不会抛出任何异常，那么( )中什么也不写：

double func (char param) throw ();

如此，func() 函数就不能抛出任何类型的异常了，即使抛出了，try 也检测不到。

（1）虚函数中的异常规范

C++ 规定，派生类虚函数的异常规范必须与基类虚函数的异常规范一样严格，或者更严格。只有这样，当通过基类指针（或者引用）调用派生类虚函数时，才能保证不违背基类成员函数的异常规范。请看下面的例子：

class Base

{

public:

virtual int fun1(int) throw();

virtual int fun2(int) throw(int);

virtual string fun3() throw(int, string);

};

class Derived:public Base

{

public:

int fun1(int) throw(int); //错！异常规范不如 throw() 严格

int fun2(int) throw(int); //对！有相同的异常规范

string fun3() throw(string); //对！异常规范比 throw(int,string) 更严格

}

（2）异常规范与函数定义和函数声明

C++ 规定，异常规范在函数声明和函数定义中必须同时指明，并且要严格保持一致，不能更加严格或者更加宽松。

请看下面的几组函数：

//错！定义中有异常规范，声明中没有

void func1();

void func1() throw(int) { }

//错！定义和声明中的异常规范不一致

void func2() throw(int);

void func2() throw(int, bool) { }

//对！定义和声明中的异常规范严格一致

void func3() throw(float, char\*);

void func3() throw(float, char\*) { }

请抛弃异常规范，不要再使用它

异常规范的初衷是好的，它希望让程序员看到函数的定义或声明后，立马就知道该函数会抛出什么类型的异常，这样程序员就可以使用 try-catch 来捕获了。如果没有异常规范，程序员必须阅读函数源码才能知道函数会抛出什么异常。

不过这有时候也不容易做到。例如，func\_outer() 函数可能不会引发异常，但它调用了另外一个函数 func\_inner()，这个函数可能会引发异常。再如，您编写的函数调用了老式的库函数，此时不会引发异常，但是库更新以后这个函数却引发了异常。总之，异常规范的初衷实现起来有点困难，所以大家达成的一致意见是，最好不要使用异常规范。

异常规范是 C++98 新增的一项功能，但是后来的 C++11 已经将它抛弃了，不再建议使用。

C++98中如果函数违背了异常规范，则会调用std::unexpected()，unexpected()调用unexpected\_handler，默认设置为std：：terminate（）。

### 20.4.noexpect（C++11）

C++98中使用throw异常规范来对函数异常说明，事实上，该特性很少被使用，因为编译器在编译时能过做的检测非常有限，因此在C++11中被弃用了，而表示函数不会抛出异常的动态异常声明throw()也被新的noexcept异常声明所取代。

在C++11中异常规范被简化为以下两种情况：

（1）函数可以抛出任何异常(和不带异常规范一样)；

（2）函数不可抛出任何异常。

在C++11中，声明一个函数不可以抛出任何异常使用关键字noexcept.

void mightThrow(); // could throw any exceptions.

void doesNotThrow() noexcept; // does not throw any exceptions.

从语法上讲，noexcept修饰符有两种形式，一种就是简单地在函数声明后加上noexcept关键字。比如：

void excpt\_func() noexcept;

另外一种则可以接受一个常量表达式作为参数，如下所示：

void excpt\_func() noexcept (常量表达式);

常量表达式的结果会被转换成一个bool类型的值。该值为true，表示函数不会抛出异常，反之，则有可能抛出异常。这里，不带常量表达式的noexcept相当于声明了noexcept(true)，即不会抛出异常。

下面两个函数声明的异常规格在语义上是相同的，都表示函数不抛出任何异常。

void old\_stytle() throw();

void new\_style() noexcept;

它们的区别在于程序运行时的行为和编译器优化的结果：

noexcept形如其名地，表示其修饰的函数不会抛出异常。不过与throw()动态异常声明不同的是，使用throw（）修饰的函数抛出了异常，编译器直接调用std::unexpected（）函数。而在C++11中如果noexcept修饰的函数抛出了异常，编译器直接调用std::terminate()函数来终止程序的运行，这比基于异常机制的throw()在效率上会高一些。这是因为异常机制会带来一些额外开销，比如函数抛出异常，会导致函数栈被依次地展开（unwind），并依帧调用在本帧中已构造的自动变量的析构函数等。

简而言之，如果你知道你的函数绝对不会抛出任何异常，应该使用noexcept, 而不是throw().

在通常情况下，在C++11中使用noexcept可以有效地阻止异常的传播与扩散。我们可以看看下面这个例子，如代码清单2-12所示。

#include <iostream>

using namespace std;

void Throw() { throw 1; }

void NoBlockThrow() { Throw(); }

void BlockThrow() noexcept { Throw(); }

int main()

{

try

{

Throw();

}

catch(...)

{

cout << "Found throw." << endl; // Found throw.

}

try

{

NoBlockThrow();

}

catch(...)

{

cout << "Throw is not blocked." << endl; // Throw is not blocked.

}

try

{

BlockThrow(); // terminate called after throwing an instance of 'int'

}

catch(...)

{

cout << "Found throw 1." << endl;

}

}

// 编译选项:g++ -std=c++11 2-6-1.cpp

我们定义了Throw函数，该函数的唯一作用是抛出一个异常。而NoBlockThrow是一个调用Throw的普通函数，BlockThrow则是一个noexcept修饰的函数。从main的运行中我们可以看到，NoBlockThrow会让Throw函数抛出的异常继续抛出，直到main中的catch语句将其捕捉。而BlockThrow则会直接调用std::terminate中断程序的执行，从而阻止了异常的继续传播。从使用效果上看，这与C++98中的throw()是一样的。

而noexcept作为一个操作符时，通常可以用于模板。比如：

template <class T>

void fun() noexcept(noexcept(T())) {}

这里，fun函数是否是一个noexcept的函数，将由T()表达式是否会抛出异常所决定。这里的第二个noexcept就是一个noexcept操作符。当其参数是一个有可能抛出异常的表达式的时候，其返回值为false，反之为true（实际noexcept参数返回false还包括一些情况，这里就不展开讲了）。这样一来，我们就可以使模板函数根据条件实现noexcept修饰的版本或无noexcept修饰的版本。从泛型编程的角度看来，这样的设计保证了关于“函数是否抛出异常”这样的问题可以通过表达式进行推导。因此这也可以视作C++11为了更好地支持泛型编程而引入的特性。

虽然noexcept修饰的函数通过std::terminate的调用来结束程序的执行的方式可能会带来很多问题，比如无法保证对象的析构函数的正常调用，无法保证栈的自动释放等，但很多时候，“暴力”地终止整个程序确实是很简单有效的做法。事实上，noexcept被广泛地、系统地应用在C++11的标准库中，用于提高标准库的性能，以及满足一些阻止异常扩散的需求。

比如在C++98中，存在着使用throw()来声明不抛出异常的函数。

template<class T> class A

{

public:

static constexpr T min() throw() { return T(); }

static constexpr T max() throw() { return T(); }

static constexpr T lowest() throw() { return T(); }

}

而在C++11中，则使用noexcept来替换throw()。

template<class T> class A

{

public:

static constexpr T min() noexcept { return T(); }

static constexpr T max() noexcept { return T(); }

static constexpr T lowest() noexcept { return T(); }

}

又比如，在C++98中，new可能会包含一些抛出的std::bad\_alloc异常。

void\* operator new(std::size\_t) throw(std::bad\_alloc);

void\* operator new[](std::size\_t) throw(std::bad\_alloc);

而在C++11中，则使用noexcept(false)来进行替代。

void\* operator new(std::size\_t) noexcept(false);

void\* operator new[](std::size\_t) noexcept(false);

当然，noexcept更大的作用是保证应用程序的安全。比如一个类析构函数不应该抛出异常，那么对于常被析构函数调用的delete函数来说，C++11默认将delete函数设置成noexcept，就可以提高应用程序的安全性。

void operator delete(void\*) noexcept;

void operator delete[](void\*) noexcept;

而同样出于安全考虑，C++11标准中让类的析构函数默认也是noexcept(true)的。当然，如果程序员显式地为析构函数指定了noexcept，或者类的基类或成员有noexcept(false)的析构函数，析构函数就不会再保持默认值。我们可以看看下面的例子。

#include <iostream>

using namespace std;

struct A

{

~A() { throw 1; }

};

struct B

{

~B() noexcept(false) { throw 2; }

};

struct C

{

B b;

};

int funA() { A a; }

int funB() { B b; }

int funC() { C c; }

int main()

{

try

{

funB();

}

catch(...)

{

cout << "caught funB." << endl; // caught funB.

}

try

{

funC();

}

catch(...)

{

cout << "caught funC." << endl; // caught funC.

}

try

{

funA(); // terminate called after throwing an instance of 'int'

}

catch(...)

{

cout << "caught funA." << endl;

}

}

// 编译选项:g++ -std=c++11 2-6-2.cpp

在代码中，无论是析构函数声明为noexcept(false)的类B，还是包含了B类型成员的类C，其析构函数都是可以抛出异常的。只有什么都没有声明的类A，其析构函数被默认为noexcept(true)，从而阻止了异常的扩散。这在实际的使用中，应该引起程序员的注意。

## 21. assert与static\_assert

断言：断言就是确认摸一个谓词或者bool表达式为真，如果不为真则退出程序。

c中的void assert(scalar expression); 是运行时断言。是宏定义，不是函数。

c++11 新增static\_assert（常量表达式，const char\*）,是静态断言，发生在编译期。

# C++11新增功能(部分)和VS12支持状况

此处只列举了部分功能以及VisualStudio 2012对其的支持，其他更多的功能请参考C++ 11的标准文档。

|  |  |  |  |
| --- | --- | --- | --- |
| **C++ 11新增功能** | **VS12是否支持** | **C++ 11新增功能** | **VS12是否支持** |
| 右值引用 | 是 | 继承构造函数 | 否 |
| 引用限定符 | 否 | 枚举前置声明 | 是 |
| 非静态成员初始值 | 否 | Unicode支持 | 否 |
| 可变参数模板 | 否 | override和final | 是 |
| 初始化列表 | 否 | noexpect | 否 |
| static\_assert | 是 | 基于范围的for | 是 |
| 自动推导 | 是 | 原生字符串 | 否 |
| 追踪返回类型 | 是 | default和delete | 否 |
| Lambda | 是 | 内联命名空间 | 否 |
| nullptr | 是 | 对齐 | 部分支持 |
| 强类型enum | 是 | 多线程 | 否 |
| constexpr | 否 | \_\_func\_\_ | 部分支持 |
| 委托构造 | 否 | long long | 支持 |

## 1.Lambda表达式

Lambda表达式是C++ 11标准中非常实用的一个功能，也是非常重要的一个功能，应该是每个人都应该熟练掌握的。

### 1.1 基本声明

Lambda表达式是一个匿名函数，即只有函数体，没有名字的函数。C++ 11中Lambda表达式的基本语法为：

[capture list] (parameterlist) mutable exception attribute ->return type { function body }

Lambda表达式的使用要注意几点。

第一，非黑色部分标注的为可省略的部分，即一个最简单的表达式可以为：

void S1\_SimpleLambda()

{

auto simpleLambda = []{std::cout << "SimpleLambda" << std::endl;};

return simpleLambda();

}

第二，一个Lambda表达式默认是const类型的，代码块内不能改变父作用域内任意变量的值。如果想要改变，则需要增加mutable，添加mutable声明后，参数列表()即使没有参数也不可省略。例如：

void ValueCaptureTest2()

{

int intValue = 0;

auto test2 = [=]() mutable {std::cout << "Lambda: " << ++intValue << std::endl;};

test2();

std::cout << "Out Lambda: " << intValue << std::endl;

}

第三，Lambda表达式默认是一个内联的函数，编译器会对其调用进行优化，所以一个Lambda表达式应该短小精悍。

第四，省略了返回值类型的 lambda 表达式的返回类型按照下列规则推演：

如果 lambda 代码块中包含了 return 语句，则该 lambda 表达式的返回类型由 return 语句的返回类型确定。如果没有 return 语句，则类似 void f(...) 函数。

第五，如省略了参数列表，类似于无参函数 f()。

第六，exception 说明lambda表达式是否抛出异常(noexcept)，以及抛出何种异常，类似于void f() throw(X, Y)。

第七，attribute 用来声明属性。

### 1.2 捕获列表

Lambda表达式的捕获列表（Capture List）指定了在可见域范围内，代码内可见的外部变量的列表：

[a,&b] a变量以值的方式呗捕获，b以引用的方式被捕获。

[=] 以值的方式捕获所有的外部自动变量。（包括this）。

[&变量名] 表示引用传递方式捕捉变量var。

[&] 以引用的方式捕获所有的外部自动变量。（包括this）。

[this] 表示值传递方式捕捉当前的this指针。

[] 不捕获外部的任何变量。

捕获列表是一个很容易出错的地方，所以应该加强注意。下面是几个简单的示例。

第一，使用值传递方式进行捕获：

class FMNValueCapture

{

public:

FMNValueCapture() : m\_intValue(0) {}

void ValueCaptureTest1()

{

auto test1 = [=]{std::cout << "Lambda: " << ++m\_intValue << std::endl;};

test1();

std::cout << "Out Lambda: " << m\_intValue << std::endl;

}

void ValueCaptureTest2()

{

int intValue = 0;

auto test2 = [=]() mutable {std::cout << "Lambda: " << ++intValue << std::endl;};

test2();

std::cout << "Out Lambda: " << intValue << std::endl;

}

private:

int m\_intValue;

};

第二，使用引用传递方式进行捕获：

void S3\_ReferenceCapture()

{

int intValue = 0;

auto refCap = [&intValue]{std::cout << "Lambda: " << ++intValue << std::endl;};

refCap();

std::cout << "Out Lambda: " << intValue << std::endl;

}

上面仅列举了几个例子，具体还要大家亲自进行尝试才能融会贯通。

### 1.3 Lambda与STL

Lambda表达式与STL中各种容器和算法组合使用，才是威力最为强大的所在。下面列举一个例子，大家可以参考一下，例子为遍历map，并对map的值进行处理（示例分别对其加1并输出）：

typedef std::map<int, int> FMNIntMap;

void S4\_StlExample()

{

FMNIntMap intMap;

for (int i = 0; i < 5; ++i)

{

intMap.insert(std::make\_pair(i, i));

}

std::cout << "first time." << std::endl;

std::transform(intMap.begin(), intMap.end(),

std::ostream\_iterator<std::string>(std::cout, "\n"),

[](FMNIntMap::value\_type& val)->std::string

{

std::stringstream ss;

ss << "first: " << val.first << ", second: " << val.second;

return ss.str();

});

std::cout << "second time." << std::endl;

std::for\_each(intMap.begin(), intMap.end(), [](std::pair<const int, int>& val)

{

++val.second;

});

std::transform(intMap.begin(), intMap.end(),

std::ostream\_iterator<std::string>(std::cout, "\n"),

Map::value\_type& val)->std::string

{

std::stringstream ss;

ss << "first: " << val.first << ", second: " << val.second;

return ss.str();

});

}

### 1.4 Lambda与仿函数

通过上面的例子，大家可以发现，Lambda表达式和仿函数的用法非常相似。但是与仿函数还是有所区别。

第一，Lambda表达式默认为内联函数，而仿函数不是。

第二，仿函数可以具有状态（即成员），通过状态来分别执行不同的分支，但是Lambda不可以。

第三，仿函数可以跨作用域，而Lambda不可以，例如下面这段代码是错误的：

static int g\_intVal = 1;

void S5\_OutScope()

{

auto outScope = [g\_intVal]{std::cout << "Lambda: " << ++g\_intVal << std::endl;};

outScope();

}

## 2 类型推导与返回值类型追踪

C++ 11中增加了强大的类型推导功能，使得开发效率得以提升，并且大幅度简化简洁了代码。

### 2.1 类型推导

类型推导主要有两个，auto和decltype，通过一个简单的例子，就可以明白它们的具体功能。

语法格式：

auto 变量； //定义一个变量，类型由其值的类型决定

decltype（表达式） 变量； //定义一个与表达式同类型的变量

void S6\_AutoDecltype()

{

std::vector<int> intVec;

std::vector<int>::iterator intVecIter1 = intVec.begin();

auto intVecIter2 = intVec.begin();

decltype(intVec.begin()) intVecIter3 = intVec.begin();

}

这样，就可以不必定义复杂的迭代器，也不必担心各种计算类型。但是需要注意的是，要区分推导类型是引用传递还是值传递。

### 2.2 返回值追踪

既然有了类型推导，想必大家会想如果返回值也是自动推导，岂不不用担心各种复杂的逻辑了？例如下面的代码：

template <class T1, class T2>

decltype(t1 + t2) Sum(T1& t1, T2& t2)

{

return (t1 + t2);

}

这样，就可以任意类型，只要支持加法运算，就可以进行加和了，例如int和double值进行加和等。但是编译器推导(t1 + t2)时，由于是从左向右解析，故此时尚未知t1和t2的类型，所以推导失败。

为了解决这个问题，C++11提出了返回值追踪的功能。

template <class T1, class T2>

auto Sum(T1& t1, T2& t2) -> decltype(t1 + t2)

{

return (t1 + t2);

}

## 3 基于范围的for循环

这个比较简单，直接参考示例代码：

语法格式：

for（元素类型 ： 范围或容器）

{

每个元素的操作

}

void S11\_ForRange()

{

std::vector<int> intVec;

intVec.push\_back(1);

intVec.push\_back(3);

intVec.push\_back(5);

intVec.push\_back(7);

for (auto i : intVec)

{

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

}

int intArray[] = {2, 4, 6, 8};

for (auto i : intArray)

{

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

}

}

## 4 空指针nullptr

对于常用的NULL来说，其本质上是具有二义性的，例如：

void NullFunc(int val)

{

std::cout << "null is int." << std::endl;

}

void NullFunc(void\* pVal)

{

std::cout << "null is a pointer" << std::endl;

}

void S9\_Nullptr()

{

static\_assert(NULL == nullptr, "Error: nullptr is not NULL");

NullFunc(NULL);

}

NULL本质上为指针，C++中定义却为0，即该函数会将NULL做为一个int值处理，产生了二义性。所以对于这个场景，C++ 11提出了nullptr，我们的代码中应该尽量用nullptr来标识一个空指针。

## 5 强类型enum和前置声明

首先说enum的前置声明，这个与struct和class的前置声明类似，例子如下：

enum class EnumClass : char; //前置声明

class RCEnumPreDef

{

private:

EnumClass m\_type;

};

enum class EnumClass : char //增加作用域， enum class或enum struct

{

TYPE\_A,

TYPE\_B,

TYPE\_C,

TYPE\_D = 30000,

};

其次，可以将enum定义为class类型，这样使用时候必须加作用域限制，提高了安全性。并且可以指定enum的范围，当超过范围时，作为溢出处理。

但是声明为强类型后，不能使用int或者char等进行enum的遍历比较操作，这个应在使用时进行权衡。

## 6 override与final

override和final主要用来对基类virtual函数的覆盖做声明。

override即声明该函数是对基类函数的覆盖，当参数不同或者返回值不同或者基类函数不为虚函数时，编译器会提示error。

final则是对某一函数的覆盖做终止。即子类不能再对此函数覆盖，如果覆盖，编译器会提示error。

通过override和final，可以提高代码的安全性，明确接口，终止接口覆盖等好处。

示例代码如下：

class FMNBase

{

public:

FMNBase() {}

virtual ~FMNBase() {}

virtual void Func(int) {std::cout << "Base" << std::endl;}

};

class FMNChildA : public FMNBase

{

public:

void Func(int) override {std::cout << "A Override" << std::endl;}

//声明该函数是对基类函数的覆盖，当参数不同或者返回值不同或者基类函数不为虚函数

时，编译器会提示error。

};

class FMNChildB : public FMNChildA

{

public:

void Func(int) final {std::cout << "B Override" << std::endl;}

//对某一函数的覆盖做终止。即子类不能再对此函数覆盖，如果覆盖，编译器会提示error。

};

class FMNChildC : public FMNChildB

{

public:

// Error

void Func(int) override {std::cout << "C Override" << std::endl;}

};

class FMNChildD : public FMNChildA

{

public:

// Error

void Func() override {std::cout << "D Override" << std::endl;}

};

## 7 右值引用与移动语义

为了定义“右值引用”的概念，首先说明左值，右值的含义，加深对此的理解。

左值，是指可以放在赋值符号“=”的左边，但其实也表示能作为&和++等操作符的操作数。可以取其地址和名字。

右值，指的是引用了一个存储在某个内存地址里的数据。不能取其地址，且没有名字。又分为“将亡值”（临时变量或者栈变量）和“纯右值”（文字常量区的内容及立即数）两种。纯右值即C++98标准中的右值，例如一些字面量等。将亡值即C++11标准中的右值引用。

通过右值引用，可以大幅度提高代码的效率，因为本质上通过内存的转移，可以减少拷贝构造等过程。

下面仅为一个简单的示例，C++ 11实际上提供了对右值引用非常多的支持和定义，所以具体还请参考C++ 11的标准文档。

void ValueFunc(int& val)

{

std::cout << "Left Value: " << val << std::endl;

}

void ValueFunc(int&& val)

{

std::cout << "Right Value: " << val << std::endl;

}

void S8\_RightValue()

{

int intVal = 0;

ValueFunc(intVal);

ValueFunc(1);

}

## 8 初始化列表

## 9 其他

除了上述重要的特性外，还有一些较简单的特性，不在详细讨论，例如：

（1）static\_assert 编译期的静态断言。

（2）STL库新增内容，如array，forward\_list，unordered\_map，unordered\_set等容器，以及新增的算法。

（3）UTF8等编码转换，可以参考头文件<codecvt>。

（4）类型萃取，可以参考头文件<type\_trains>

当然，还有其他很多Visual Studio 2012不支持，或者部分支持的特性，如果使用，还请务必参考微软的官方文档。

# 正则表达式

## 1.简介

除非您以前使用过正则表达式，否则您可能不熟悉一此术语。但是，毫无疑问，您已经使用过不涉及脚本的某些正则表达式概念。

例如，您很可能使用 ? 和 \* 通配符来查找硬盘上的文件。? 通配符匹配文件名中的单个字符，而 \* 通配符匹配零个或多个字符。像 data?.dat 这样的模式将查找下列文件：

data1.dat

data2.dat

datax.dat

dataN.dat

使用 \* 字符代替 ? 字符扩大了找到的文件的数量。data\*.dat 匹配下列所有文件：

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

尽管这种搜索方法很有用，但它还是有限的。通过理解 \* 通配符的工作原理，引入了正则表达式所依赖的概念，但正则表达式功能更强大，而且更加灵活。

正则表达式的使用，可以通过简单的办法来实现强大的功能。下面先给出一个简单的示例：

^[0-9]+abc$

^ 为匹配输入字符串的开始位置。

[0-9]+匹配多个数字， [0-9] 匹配单个数字，+ 匹配一个或者多个。

abc$匹配字母 abc 并以 abc 结尾，$ 为匹配输入字符串的结束位置。

实例

匹配以数字开头，并以 abc 结尾的字符串。：

var str = "123abc";

var patt1 = /^[0-9]+abc$/;

document.write(str.match(patt1));

以下标记的文本是获得的匹配的表达式：

123abc

### 1.1.为什么使用正则表达式？

典型的搜索和替换操作要求您提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

通过使用正则表达式，可以：

测试字符串内的模式。

例如，可以测试输入字符串，以查看字符串内是否出现电话号码模式或信用卡号码模式。这称为数据验证。

替换文本。

可以使用正则表达式来识别文档中的特定文本，完全删除该文本或者用其他文本替换它。

基于模式匹配从字符串中提取子字符串。

可以查找文档内或输入域内特定的文本。

例如，您可能需要搜索整个网站，删除过时的材料，以及替换某些 HTML 格式标记。在这种情况下，可以使用正则表达式来确定在每个文件中是否出现该材料或该 HTML 格式标记。此过程将受影响的文件列表缩小到包含需要删除或更改的材料的那些文件。然后可以使用正则表达式来删除过时的材料。最后，可以使用正则表达式来搜索和替换标记。

### 1.2.发展历史

正则表达式的"祖先"可以一直上溯至对人类神经系统如何工作的早期研究。Warren McCulloch 和 Walter Pitts 这两位神经生理学家研究出一种数学方式来描述这些神经网络。

1956 年, 一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为"神经网事件的表示法"的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为"正则集的代数"的表达式，因此采用"正则表达式"这个术语。

随后，发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

如他们所说，剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

应用领域

目前，正则表达式已经在很多软件中得到广泛的应用，包括 \*nix（Linux, Unix等）、HP 等操作系统，PHP、C#、Java 等开发环境，以及很多的应用软件中，都可以看到正则表达式的影子。

可在每种语言的教程中找到对应的正则表达式教程

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 命令或环境 | . | [ ] | ^ | $ | \( \) | \{ \} | ? | + | | | ( ) |
| vi | √ | √ | √ | √ | √ |  |  |  |  |  |
| Visual C++ | √ | √ | √ | √ | √ |  |  |  |  |  |
| awk | √ | √ | √ | √ |  | 要在命令行加入 --posix or --re-interval | √ | √ | √ | √ |
| sed | √ | √ | √ | √ | √ | √ |  |  |  |  |
| delphi | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| python | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| java | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| javascript | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| php | √ | √ | √ | √ | √ |  |  |  |  |  |
| perl | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| C# | √ | √ | √ | √ |  |  | √ | √ | √ | √ |

## 2.语法

正则表达式(regular expression)描述了一种字符串匹配的模式（pattern），可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。

例如：

runoo+b，可以匹配 runoob、runooob、runoooooob 等，+ 号代表前面的字符必须至少出现一次（1次或多次）。

runoo\*b，可以匹配 runob、runoob、runoooooob 等，\* 号代表字符可以不出现，也可以出现一次或者多次（0次、或1次、或多次）。

colou?r 可以匹配 color 或者 colour，? 问号代表前面的字符最多只可以出现一次（0次、或1次）。

构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符如aa+中的第二个a、字符集合[xyz],[^xyz]、字符范围[a-z],[^a-z]、字符间的选择(a|b)或者所有这些组件的任意组合。

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

### 2.1.普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

### 2.2.非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

|  |  |
| --- | --- |
| **字符** | **描述** |
| \cx | 匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。 |
| \f | 匹配一个换页符。等价于 \x0c 和 \cL。 |
| \n | 匹配一个换行符。等价于 \x0a 和 \cJ。 |
| \r | 匹配一个回车符。等价于 \x0d 和 \cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于 \x09 和 \cI。 |
| \v | 匹配一个垂直制表符。等价于 \x0b 和 \cK。 |

### 2.3.特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的 runoo\*b 中的 \*，简单的说就是表示任何字符串的意思。如果要查找字符串中的 \* 符号，则需要对 \* 进行转义，即在其前加一个 \: runo\\*ob 匹配 runo\*ob。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"\"转义，即，将反斜杠字符\ 放在它们前面。下表列出了正则表达式中的特殊字符：

|  |  |
| --- | --- |
| **字符** | **描述** |
| ( ) | 标记一个子表达式的开始和结束位置。子表达式的匹配结果可以获取供以后使用。要匹配()，请使用 \( 和 \)。 |
| \* | 匹配前面的表达式零次或多次。要匹配 \* 字符，请使用 \\*。 |
| + | 匹配前面的表达式一次或多次。要匹配 + 字符，请使用 \+。 |
| . | 匹配除换行符\r\n 之外的任何单字符。要匹配 . ，请使用 \. 。 |
| [] | 标记一个中括号表达式的开始和结束。表示要匹配的字符是一个范围。要匹配 []，请使用 \[和\]。 |
| ? | 匹配前面的表达式0次或一次将贪婪限定符转为非贪婪限定符，只进行最少匹配。要匹配 ? 字符，请使用 \?。 |
| \ | 将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。 |
| {} | 标记限定符表达式的开始和结束。要匹配 {}，请使用 \{和\}。 |
| | | 指明两项之间的一个选择。要匹配 |，请使用 \|。 |
| ^ | 匹配输入字符串开始的位置。如果设置了 RegExp对象的Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。 |
| $ | 匹配输入字符串结尾的位置。如果设置了RegExp对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。 |

### 2.4.限定符

限定符用来指名前面的**紧邻组件**必须要出现多少次（只能管理紧紧在其之前的组件，也叫子表达式）才能满足匹配。有 \* 或 + 或 ? 或 {n} 或 {n,} 或 {n,m} 共6种。

正则表达式的限定符有：

|  |  |
| --- | --- |
| 字符 | 描述 |
| \* | 匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。 |
| {n} | n 是一个非负整数。匹配确定的 n次。例如，'o{2}' 不能匹配"Bob"中的 'o'，但是能匹配 "food" 中的两个 o。 |
| {n,} | n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。 |
| {n,m} | m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。 |

由于章节编号在大的输入文档中会很可能超过九，所以您需要一种方式来处理两位或三位章节编号。限定符给您这种能力。下面的正则表达式匹配编号为任何位数的章节标题：

/Chapter [1-9][0-9]\*/

请注意，限定符出现在范围表达式之后。因此，它应用于整个范围表达式，在本例中，只指定从 0 到 9 的数字（包括 0 和 9）。

这里不使用 + 限定符，因为在第二个位置或后面的位置不一定需要有一个数字。也不使用？字符，因为它将章节编号限制到只有两位数。您需要至少匹配 Chapter 和空格字符后面的一个数字。

如果您知道章节编号被限制为只有 99 章，可以使用下面的表达式来至少指定一位但至多两位数字。

/Chapter [0-9]{1,2}/

上面的表达式的缺点是，大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下：

/Chapter [1-9][0-9]?/

或

/Chapter [1-9][0-9]{0,1}/

\*、+和?限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上一个?就可以实现非贪婪或最小匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

<H1>Chapter 1 - 介绍正则表达式</H1>

贪婪：下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

/<.\*>/

非贪婪：如果您只需要匹配开始和介绍 H1 标记，下面的非贪婪表达式只匹配 <H1>。

/<.\*?>/

如果只想匹配开始的 H1 标签，表达式则是：

/<\w+?>/

通过在 \*、+ 或 ? 限定符之后放置 ?，该表达式从"贪心"表达式转换为"非贪心"表达式或者最小匹配。

### 2.5.定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。

定位符用来描述字符串或单词的边界，**^** 和 **$** 分别指字符串的开始与结束，span class="marked">\b 描述单词的前或后边界，span class="marked">\B 表示非单词边界。

正则表达式的定位符有：

|  |  |
| --- | --- |
| **字符** | **描述** |
| ^ | 匹配输入字符串开始的位置。如果设置了 RegExp对象的Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。 |
| $ | 匹配输入字符串结尾的位置。如果设置了RegExp对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。 |
| \b | 匹配一个字边界，即字与空格间的位置。 |
| \B | 非字边界匹配。 |

注意：不能将限定符与定位点一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^\* 之类的表达式。

若要匹配一行文本开始处的文本，请在正则表达式的开始使用 ^ 字符。不要将 ^ 的这种用法与中括号表达式内的用法混淆。

若要匹配一行文本的结束处的文本，请在正则表达式的结束处使用 $ 字符。

若要在搜索章节标题时使用定位点，下面的正则表达式匹配一个章节标题，该标题只包含两个尾随数字，并且出现在行首：

/^Chapter [1-9][0-9]{0,1}/

真正的章节标题不仅出现行的开始处，而且它还是该行中仅有的文本。它即出现在行首又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式，就可做到这一点。

/^Chapter [1-9][0-9]{0,1}$/

匹配字边界稍有不同，但向正则表达式添加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 Chapter 的开头三个字符，因为这三个字符出现字边界后面：

/\bCha/

\b 字符的位置是非常重要的。如果它位于要匹配的字符串的开始，它在单词的开始处查找匹配项。如果它位于字符串的结尾，它在单词的结尾处查找匹配项。例如，下面的表达式匹配单词 Chapter 中的字符串 ter，因为它出现在字边界的前面：

/ter\b/

下面的表达式匹配 Chapter 中的字符串 apt，但不匹配 aptitude 中的字符串 apt：

/\Bapt/

字符串 apt 出现在单词 Chapter 中的非字边界处，但出现在单词 aptitude 中的字边界处。对于 \B 非字边界运算符，位置并不重要，因为匹配不关心究竟是单词的开头还是结尾。

### 2.6.选择

用圆括号将所有选择项括起来，相邻的选择项之间用|分隔。但用圆括号会有一个副作用，是相关的匹配结果会被缓存，此时可用?:放在第一个选项前来禁止缓存匹配结果。

?: 是非捕获元之一，还有两个非捕获元是 ?= 和 ?!，这两个还有更多的含义，前者为正向肯定预查，在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串，后者为正向否定预查，在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

### 2.7.反向引用

对一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中，所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 \number 引用，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符 ?:、?= 或 ?! 来重写捕获，忽略对相关匹配的保存。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力。以下面的句子为例：

Is is the cost of of gasoline going up up?

上面的句子很显然有多个重复的单词。如果能设计一种方法定位该句子，而不必查找每个单词的重复出现，那该有多好。下面的正则表达式使用单个子表达式来实现这一点：

实例

查找重复的单词：

var str = "Is is the cost of of gasoline going up up";

var patt1 = /\b([a-z]+) \1\b/;

document.write(str.match(patt1));

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。

字边界元字符确保只检测整个单词。否则，诸如 "is issued" 或 "this is" 之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 g 指定将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。

表达式的结尾处的不区分大小写 i 标记指定不区分大小写。

多行标记指定换行符的两边可能出现潜在的匹配。

反向引用还可以将通用资源指示符 (URI) 分解为其组件。假定您想将下面的 URI 分解为协议（ftp、http 等等）、域地址和页/路径：

http://www.runoob.com:80/html/html-tutorial.html

下面的正则表达式提供该功能：

实例

输出所有匹配的数据：

var str = "https://www.runoob.com:80/html/html-tutorial.html";

var patt1 = /(\w+):\/\/([^/:]+)(:\d\*)?([^# ]\*)/;

arr = str.match(patt1);

for (var i = 0; i < arr.length ; i++) {

document.write(arr[i]);

document.write("<br>");

}

第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。

第二个括号子表达式捕获地址的域地址部分。子表达式匹配 / 和 : 之外的一个或多个字符。

第三个括号子表达式捕获端口号（如果指定了的话）。该子表达式匹配冒号后面的零个或多个数字。只能重复一次该子表达式。

最后，第四个括号子表达式捕获 Web 地址指定的路径和 / 或页信息。该子表达式能匹配不包括 # 或空格字符的任何字符序列。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

第一个括号子表达式包含"http"

第二个括号子表达式包含"www.runoob.com"

第三个括号子表达式包含":80"

第四个括号子表达式包含"/html/html-tutorial.html"

### 2.8.零宽断言

#### 2.8.1肯定断言

用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(即断言)，因此它们也被称为零宽断言。最好还是拿例子来说明吧：

**(?=exp)也叫零宽度正预测先行断言** ，它断言自身出现的位置的后面能匹配表达式exp。比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)，如查找I'm singing while you're dancing.时，它会匹配sing和danc。

**(?<=exp)也叫零宽度正回顾后发断言** ，它断言自身出现的位置的前面能匹配表达式exp。比如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找reading a book时，它匹配ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\D)\D{3})+\b，用它对xxxxxxxxxx进行查找时结果是xxxxxxxxx

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)

断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配。

#### 2.8.2.否定断言

如果我们只是想要确保某个字符没有出现，但并不想去匹配它时怎么办？

(?!exp)，零宽度负预测先行断言，断言此位置的后面不能匹配表达式exp。例如：\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字；\b((?!abc)\w)+\b匹配不包含连续字符串abc的单词。

(?<!exp),零宽度负回顾后发断言，断言此位置的前面不能匹配表达式exp：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字。

### 2.9.元字符

下表包含了元字符的完整列表以及它们在正则表达式上下文中的行为：

|  |  |
| --- | --- |
| **字符** | **描述** |
| \ | 将下一个字符标记为特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。 |
| ^ | 匹配输入字符串的开始位置。并不是一个单词的开头，如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n'或'\r'之后的位置。 |
| $ | 匹配输入字符串的结束位置。并不是一个单词的结尾，如果设置了RegExp 对象的 Multiline 属性，$也匹配 '\n'或 '\r' 之前的位置。 |
| \* | 匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。 |
| {n} | n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。 |
| {n,} | n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。 |
| {n,m} | m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符 (\*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。 |
| **.** | 匹配除 "\r\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用"(.|\n)"或者“[\s\S]”的模式。 |
| (expr) | 匹配 pattern 并获取匹配结果，匹配结果可以从产生的 Matches 集合得到，在VBScript 中使用 SubMatches 集合，在JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。 |
| (?:expr) | 非获取匹配，匹配expr但不获取匹配结果，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分时很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=expr) | 非获取匹配，正向肯定预查，在任何匹配expr的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?<=expr) | 非获取匹配，负向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?!expr) | 非获取匹配，正向否定预查，在任何不匹配expr的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。 |
| (?<!expr) | 非获取匹配，负向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|20)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“20Windows”中的“Windows”。注意，此处用或任意一项都不能超过2位，如“(?<!95|98|NT|20)Windows正确“(?<!95|980|NT|20)Windows 报错，若是单独使用则无限制，如(?<!2000)Windows 正确匹配 |
| x|y | 匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'、'l'、'i'、'n'。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。 |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 |
| \B | 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。 |
| \cx | 匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。 |
| \d | 匹配一个数字字符。等价于 [0-9]。 |
| \D | 匹配一个非数字字符。等价于 [^0-9]。 |
| \f | 匹配一个换页符。等价于 \x0c 和 \cL。 |
| \n | 匹配一个换行符。等价于 \x0a 和 \cJ。 |
| \r | 匹配一个回车符。等价于 \x0d 和 \cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于 \x09 和 \cI。 |
| \v | 匹配一个垂直制表符。等价于 \x0b 和 \cK。 |
| \w | 匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9\_]'。 |
| \W | 匹配任何非单词字符。等价于 '[^A-Za-z0-9\_]'。 |
| \xn | 匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。 |
| \num | 匹配 num，其中num是一个正整数。对所获取的匹配的反向引用。例如，'(.)\1'匹配两个连续的相同字符。 |
| \n | 标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。 |
| \nm | 标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。 |
| \nml | 如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。 |
| \un | 匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如， \u00A9 匹配版权符号 (?)。 |
| \p{P} | 小写 p 是 property 的意思，表示 Unicode 属性，用于 Unicode 正表达式的前缀。大括号内的“P”表示Unicode 字符集七个字符属性之一：标点字符。其他六个属性：L：字母；M：标记符号（一般不会单独出现）；Z：分隔符（比如空格、换行等）；S：符号（比如数学符号、货币符号等）；N：数字（比如阿拉伯数字、罗马数字等）；C：其他字符。  \*注：此语法部分语言不支持，例：javascript。 |
| \<  \> | 匹配词（word）的开始（\<）和结束（\>）。例如正则表达式\<the\>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。 |
| | | 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。 |

## 3.运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

相同优先级的从左到右进行运算，不同优先级的运算先高后低。下表从最高到最低说明了各种正则表达式运算符的优先级顺序：

|  |  |
| --- | --- |
| **运算符** | **描述** |
| \ | 转义符 |
| (), (?:), (?=), [] | 圆括号和方括号 |
| \*, +, ?, {n}, {n,}, {n,m} | 限定符 |
| ^, $, \任何元字符、任何字符 | 定位点和序列（即：位置和顺序） |
| | | 替换，"或"操作。字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"。 |

## 4.匹配规则

### 4.1.基本模式匹配

一切从最基本的开始。模式，是正规表达式最基本的元素，它们是一组描述字符串特征的字符。模式可以很简单，由普通的字符串组成，也可以非常复杂，往往用特殊的字符表示一个范围内的字符、重复出现，或表示上下文。一般格式为：

定位符表达式限制符

例如：

^once

这个模式包含一个特殊的字符^，表示该模式只匹配那些以once开头的字符串。例如该模式与字符串"once upon a time"匹配，与"There once was a man from NewYork"不匹配。正如如^符号表示开头一样，$符号用来匹配那些以给定模式结尾的字符串。

bucket$

这个模式与"Who kept all of this cash in a bucket"匹配，与"buckets"不匹配。字符^和$同时使用时，表示精确匹配（字符串与模式一样）。例如：

^bucket$

只匹配字符串"bucket"。如果一个模式不包括^和$，那么它与任何包含该模式的字符串匹配。例如：模式

once

与字符串

There once was a man from NewYork Who kept all of his cash in a bucket.

是匹配的。

在该模式中的字母(o-n-c-e)是字面的字符，也就是说，他们表示该字母本身，数字也是一样的。其他一些稍微复杂的字符，如标点符号和白字符（空格、制表符等），要用到转义序列。所有的转义序列都用反斜杠(\)打头。制表符的转义序列是：\t。所以如果我们要检测一个字符串是否以制表符开头，可以用这个模式：

^\t

类似的，用\n表示"新行"，\r表示回车。其他的特殊符号，可以用在前面加上反斜杠，如反斜杠本身用\\表示，句号.用\.表示，以此类推。

### 4.2.字符簇

在INTERNET的程序中，正规表达式通常用来验证用户的输入。当用户提交一个FORM以后，要判断输入的电话号码、地址、EMAIL地址、信用卡号码等是否有效，用普通的基于字面的字符是不够的。

所以要用一种更自由的描述我们要的模式的办法，它就是字符簇。要建立一个表示所有元音字符的字符簇，就把所有的元音字符放在一个方括号里：

[AaEeIiOoUu]

这个模式与任何元音字符匹配，但只能表示一个字符。用连字号可以表示一个字符的范围，如：

[a-z] //匹配所有的小写字母

[A-Z] //匹配所有的大写字母

[a-zA-Z] //匹配所有的字母

[0-9] //匹配所有的数字

[0-9\.\-] //匹配所有的数字，句号和减号

[ \f\r\t\n] //匹配所有的白字符

同样的，这些也只表示一个字符，这是一个非常重要的。如果要匹配一个由一个小写字母和一位数字组成的字符串，比如"z2"、"t6"或"g7"，但不是"ab2"、"r2d3" 或"b52"的话，用这个模式：

^[a-z][0-9]$

尽管[a-z]代表26个字母的范围，但在这里它只能与第一个字符是小写字母的字符串匹配。

前面曾经提到^表示字符串的开头，但它还有另外一个含义。当在一组方括号里使用^是，它表示"非"或"排除"的意思，常常用来剔除某个字符。还用前面的例子，我们要求第一个字符不能是数字：

^[^0-9][0-9]$

这个模式与"&5"、"g7"及"-2"是匹配的，但与"12"、"66"是不匹配的。下面是几个排除特定字符的例子：

[^a-z] //除了小写字母以外的所有字符

[^\\\/\^] //除了(\)(/)(^)之外的所有字符

[^\"\'] //除了双引号(")和单引号(')之外的所有字符

特殊字符"." (点，句号)在正则表达式中用来表示除了"\n"之外的所有字符。所以模式"^.5$"与任何两个字符的、以数字5结尾和以其他非"新行"字符开头的字符串匹配。模式"."可以匹配任何字符串，除了空串和只包括一个"新行"的字符串。

PHP的正规表达式有一些内置的通用字符簇，列表如下：

|  |  |
| --- | --- |
| 字符簇 | 描述 |
| [[:alpha:]] | 任何字母 |
| [[:digit:]] | 任何数字 |
| [[:alnum:]] | 任何字母和数字 |
| [[:space:]] | 任何空白字符 |
| [[:upper:]] | 任何大写字母 |
| [[:lower:]] | 任何小写字母 |
| [[:punct:]] | 任何标点符号 |
| [[:xdigit:]] | 任何16进制的数字，相当于[0-9a-fA-F] |

### 4.3.确定重复出现的次数

到现在为止，你已经知道如何去匹配一个字母或数字，但更多的情况下，可能要匹配一个单词或一组数字。一个单词有若干个字母组成，一组数字有若干个单数组成。跟在字符或字符簇后面的花括号{}用来确定前面的内容的重复出现的次数。

|  |  |
| --- | --- |
| 字符簇 | 描述 |
| ^[a-zA-Z\_]$ | 所有的字母和下划线 |
| ^[[:alpha:]]{3}$ | 所有的3个字母的单词 |
| ^a$ | 字母a |
| ^a{4}$ | aaaa |
| ^a{2,4}$ | aa,aaa或aaaa |
| ^a{1,3}$ | a,aa或aaa |
| ^a{2,}$ | 包含多于两个a的字符串 |
| ^a{2,} | 如：aardvark和aaab，但apple不行 |
| a{2,} | 如：baad和aaa，但Nantucket不行 |
| \t{2} | 两个制表符 |
| .{2} | 所有的两个字符 |

这些例子描述了花括号的三种不同的用法。一个数字，{x}的意思是"前面的字符或字符簇只出现x次"；一个数字加逗号，{x,}的意思是"前面的内容出现x或更多的次数"；两个用逗号分隔的数字，{x,y}表示"前面的内容至少出现x次，但不超过y次"。我们可以把模式扩展到更多的单词或数字：

^[a-zA-Z0-9\_]{1,}$ //所有包含一个以上的字母、数字或下划线的字符串

^[1-9][0-9]\*$ //所有的正整数

^\-{0,1}[0-9]{1,}$ //所有的整数

^[-]?[0-9]+\.?[0-9]+$ //所有的浮点数

最后一个例子不太好理解，是吗？这么看吧：与所有以一个可选的负号([-]?)开头(^)、跟着1个或更多的数字([0-9]+)、和一个小数点(\.)再跟上1个或多个数字([0-9]+)，并且后面没有其他任何东西($)。下面你将知道能够使用的更为简单的方法。

特殊字符"?"与{0,1}是相等的，它们都代表着："0个或1个前面的内容"或"前面的内容是可选的"。所以刚才的例子可以简化为：

^\-?[0-9]{1,}\.?[0-9]{1,}$

特殊字符"\*"与{0,}是相等的，它们都代表着"0个或多个前面的内容"。最后，字符"+"与 {1,}是相等的，表示"1个或多个前面的内容"，所以上面的4个例子可以写成：

^[a-zA-Z0-9\_]+$ //所有包含一个以上的字母、数字或下划线的字符串

^[0-9]+$ // 0 及所有的正整数

^\-?[0-9]+$ //所有的整数

^\-?[0-9]+\.?[0-9]\*$ //所有的浮点数

当然这并不能从技术上降低正规表达式的复杂性，但可以使它们更容易阅读。

## 5.常用正则表达式

1.验证用户名和密码：（"^[a-zA-Z]\w{5,15}$"）正确格式："[A-Z][a-z]\_[0-9]"组成,并且第一个字必须为字母6~16位；

2.验证电话号码：（"^(\d{3,4}-)\d{7,8}$"）正确格式：xxx/xxxx-xxxxxxx/xxxxxxxx；

3.验证手机号码："^1[3|4|5|7|8][0-9]\\d{9}$"；

4.验证身份证号（15位或18位数字）："\d{14}[[0-9],0-9xX]"；

5.验证Email地址：("^\w+([-+.]\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$")；

6.只能输入由数字和26个英文字母组成的字符串：("^[A-Za-z0-9]+$")；

7.整数或者小数：^[0-9]+([.][0-9]+){0,1}$

8.只能输入数字："^[0-9]\*$"。

9.只能输入n位的数字："^\d{n}$"。

10.只能输入至少n位的数字："^\d{n,}$"。

11.只能输入m~n位的数字："^\d{m,n}$"。

12.只能输入零和非零开头的数字："^(0|[1-9][0-9]\*)$"。

13.只能输入有两位小数的正实数："^[0-9]+(\.[0-9]{2})?$"。

14.只能输入有1~3位小数的正实数："^[0-9]+(\.[0-9]{1,3})?$"。

15.只能输入非零的正整数："^\+?[1-9][0-9]\*$"。

16.只能输入非零的负整数："^\-[1-9][0-9]\*$"。

17.只能输入长度为3的字符："^.{3}$"。

18.只能输入由26个英文字母组成的字符串："^[A-Za-z]+$"。

19.只能输入由26个大写英文字母组成的字符串："^[A-Z]+$"。

20.只能输入由26个小写英文字母组成的字符串："^[a-z]+$"。

21.验证是否含有^%&',;=?$\"等字符："[%&',;=?$\\^]+"。

22.只能输入汉字："^[\u4e00-\u9fa5]{0,}$"。

23.验证URL："^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]\*)?$"。

24.验证一年的12个月："^(0?[1-9]|1[0-2])$"正确格式为："01"～"09"和"10"～"12"。

25.验证一个月的31天："^((0?[1-9])|((1|2)[0-9])|30|31)$"正确格式为；"01"～"09"、"10"～"29"和“30”~“31”。

26.获取日期正则表达式：\\d{4}[年|\-|\.]\d{\1-\12}[月|\-|\.]\d{\1-\31}日?

评注：可用来匹配大多数年月日信息。

27.匹配双字节字符(包括汉字在内)：[^\x00-\xff]

评注：可以用来计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

28.匹配空白行的正则表达式：\n\s\*\r

评注：可以用来删除空白行

29.匹配HTML标记的正则表达式：<(\S\*?)[^>]\*>.\*?</>|<.\*? />

评注：网上流传的版本太糟糕，上面这个也仅仅能匹配部分，对于复杂的嵌套标记依旧无能为力

30.匹配首尾空白字符的正则表达式：^\s\*|\s\*$

评注：可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式

31.匹配网址URL的正则表达式：[a-zA-z]+://[^\s]\*

评注：网上流传的版本功能很有限，上面这个基本可以满足需求

32.匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9\_]{4,15}$

评注：表单验证时很实用

33.匹配腾讯QQ号：[1-9][0-9]{4,}

评注：腾讯QQ号从10 000 开始

34.匹配中国邮政编码：[1-9]\\d{5}(?!\d)

评注：中国邮政编码为6位数字

35.匹配ip地址：([1-9]{1,3}\.){3}[1-9]。

评注：提取ip地址时有用

36.匹配MAC地址：([A-Fa-f0-9]{2}\:){5}[A-Fa-f0-9]

## 6.c中的正则表达式（最快）

使用正则表达式可简单的分成几步：

编译正则表达式、执行匹配、释放内存。

首先，编译正则表达式

int regcomp(regex\_t \*preg, const char \*regex, int cflags);

reqcomp()函数用于把正则表达式编译成某种格式，可以使后面的匹配更有效。

preg： regex\_t结构体用于存放编译后的正则表达式；

regex： 指向正则表达式指针；

cflags：编译模式

REG\_EXTENDED：使用功能更强大的扩展正则表达式

REG\_ICASE：忽略大小写

REG\_NOSUB：不用存储匹配后的结果

REG\_NEWLINE：识别换行符，这样‘$’就可以从行尾开始匹配，‘^’就可以从行的开头开始匹配。否则忽略换行符，把整个文本串当做一个字符串处理。

其次，执行匹配

int regexec(const regex\_t \*preg, const char \*string, size\_t nmatch, regmatch\_t pmatch[], int eflags);

preg： 已编译的正则表达式指针；

string：目标字符串；

nmatch:pmatch数组的长度；

pmatch：结构体数组，存放匹配文本串的位置信息；

eflags：匹配模式

共两种匹配模式：

REG\_NOTBOL：The match-beginning-of-line operator always fails to match (but see the compilation flag REG\_NEWLINE above). This flag may be used when different portions of a string are passed to regexec and the beginning of the string should not be interpreted as the beginning of the line.

REG\_NOTEOL:The match-end-of-line operator always fails to match (but see the compilation flag REG\_NEWLINE above)

最后，释放内存

void regfree(regex\_t \*preg);

当使用完编译好的正则表达式后，或者需要重新编译其他正则表达式时，一定要使用这个函数清空该变量。

其他，处理错误

size\_t regerror(int errcode, const regex\_t \*preg, char \*errbuf, size\_t errbuf\_size);

当执行regcomp 或者regexec 产生错误的时候，就可以调用这个函数而返回一个包含错误信息的字符串。

errcode： 由regcomp 和 regexec 函数返回的错误代号。

preg： 已经用regcomp函数编译好的正则表达式，这个值可以为NULL。

errbuf： 指向用来存放错误信息的字符串的内存空间。

errbuf\_size： 指明buffer的长度，如果这个错误信息的长度大于这个值，则regerror 函数会自动截断超出的字符串，但他仍然会返回完整的字符串的长度。所以我们可以用如下的方法先得到错误字符串的长度。

# C/C++11关键字

## 1.关键字列表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| alignas | 陌生 | continue |  | friend |  | register |  | true |  |
| alignof | 陌生 | decltype | 陌生 | goto |  | reinterpret\_cast |  | try | 陌生 |
| asm | 陌生 | default |  | if |  | return |  | typedef |  |
| auto |  | delete |  | inline |  | short |  | typeid | 陌生 |
| bool |  | do |  | int |  | signed |  | typename |  |
| break |  | double |  | long |  | sizeof |  | union |  |
| case |  | dynamic\_cast |  | mutable | 陌生 | static |  | unsigned |  |
| catch |  | else |  | namespace |  | static\_cast |  | using |  |
| char |  | enum |  | noexcept |  | struct |  | virtual |  |
| char16\_t |  | explicit |  | nullptr |  | static\_assert |  | void |  |
| char32\_t |  | export | 陌生 | new |  | switch |  | volatile | 陌生 |
| class |  | extern |  | operator |  | template |  | wchar\_t |  |
| const |  |  |  | private |  | this |  | while |  |
| const\_cast |  | float |  | protected |  | throw |  |  |  |
| constexpr | 陌生 | for |  | public |  | thread\_local | 陌生 |  |  |

注： （1）黄色背景为C的32个关键字。

（2）蓝色是C++11中发生意义变化的关键字。

（3）褐色是C++11中新加的10个的关键字。

（4）红色时陌生的关键词。

（5）C++98共63个关键字。

（6）C++11共73个关键字。

## 2.关键字详解

### 2.1.C++11详解编辑

#### 2.1.1.alignas

alignof用于获取指定表达式指定的（类似sizeof，可以直接是类型名）的对齐(alignment)。alignas用于声明时指定对齐，类似于现有的类型。和sizeof类似，两者的操作数都不被求值。

#### 2.1.2.constexpr

类似const但更强大，修饰函数或对象，表示函数结果或对象是编译时决定的常量，以便优化。（const不能修饰一般的函数，也不一定指定声明的对象能编译期的常量表达式，更可能只是只读对象。而在C语言中，const完全只能指定只读对象。）

#### 2.1.3.char16\_t 和 char32\_t

二者分别表示16位字符型和32位字符型，类似char和wchar\_t，也是一般只专用于表示字符的整数类型，且设计上用于表示Unicode字符。char16\_t和char32\_t是C++11新增的，以克服wchar\_t在不同平台上无法保证确定宽度的缺点。

#### 2.1.4.decltype

用于编译时推断类型。此外参与函数声明的返回值类型追踪：指定返回auto，同时decltype引导trailing-return-type指定实际应该返回类型。decltype的操作数也不被求值。

类型推断（从左往右推断）

int a =5;

decltype(a) b = 10;

auto c= 20;

返回值类型追踪

template <typename T1, tyname T2>

auto add(T1 a,T2 b )->decltype(a+b)

{

return a+b;

}

#### 2.1.5.nullptr

字面量nullptr是具有std::nullptr\_t类型的右值，是空指针常量。C++98/03中表示空指针常量的NULL或0都会在重载中引起混淆，而纯库的解决方案在这里也遇到困难，所以有必要加入新的关键字来专门表示空指针。

#### 2.1.6.noexcept

实践表明动态异常规范会影响运行时性能。新增的noexcept表示静态异常规范，只指定函数（模版）是否有异常抛出，这里noexcept即noexcept(true)，表示没有异常抛出。除了异常规范，noexcept可以作用于一个表达式来判断是否有异常，这对于模版代码非常有用。

#### 2.1.7.static\_assert

用于编译时的静态断言：若指定的表达式为false则编译失败。

#### 2.1.8.auto

C++11标准和C++98/03标准的auto是不同的。C++98/03标准中，auto表示自动储存类型[6] ；C++11标准中，auto表示由编译器静态判断其应有的类型[4] 。

### 2.2.C++98关键字详解

#### 2.2.1.asm

asm是一个语句的分隔符。不能单独出现，必须接汇编指令。一组被大括号包含的指令或一对空括号。也可以在每个汇编指令前加asm

#### 2.2.2.auto

在C++98/03中这个这个关键字用于声明块中的变量的生存期为自动生存期，若是对象同时具有自动存储类，即生存期在块结束时结束。这样的变量被称为局部变量。这个关键字不常用，因为即便省略，声明的默认就是auto的。[6]

在C++11中，auto的含义改变为自动通过初值符推断声明的类型占位符。如声明auto i = 1;，auto就相当于int，因为1是int类型，可以推断出i的类型。也可以使用auto& i等声明，具体推导规则同模版参数类型推导。[4]

#### 2.2.3.\*\_cast

即const\_cast、dynamic\_cast、reinterpret\_cast、static\_cast。

C++风格的强制类型转换。相比C风格的类型转换更加细化，增强了类型安全性。

const\_cast删除或增加指针或引用的const属性，方便赋值；

dynamic\_cast用于将一个基类对象的指针或引用转换为派生类对象的指针或引用，并做安全检查；

reinterpret\_cast将一种类型的指针或引用转换为另一种不相关的的类型；

static\_cast用于静态转换基本数据类型，空指针，任何类型转为void，void\*转为任何类型的引用，基类与派生类的指针或引用的下行转换，任何转换都可以用它，但他不能用于两个不相关的类型转换。

#### 2.2.4.bool,true,false

bool即布尔类型，属于基本类型中的整数类型，取值为真和假。true和false是具有bool类型的字面量，是右值，分别表示真和假

注：字面量：用于表达源代码中一个固定值的表示法。类似的还由nullptr，nothrow都是字面量。

#### 2.2.5.break,continue,goto

break用于跳出for或while循环或switch。continue用于跳转到循环起始。goto用于无条件跳转到函数内的标号。结构化程序较少使用goto，更多使用循环代替。

#### 2.2.6.case,default,switch

switch分支语句的起始，根据switch条件跳转到case标号或defalut标记的分支上。

#### 2.2.7.catch,throw,try

用于异常处理。try指定try块的起始，try块后的catch可以捕获异常。异常由throw抛出。throw在函数中还表示动态异常规范，但在C++11中被标记为过时（由noexcept部分取代）。

#### 2.2.8.char,wchar\_t

表示字符型和宽字符型这些整数类型，但一般只专用于表示字符。char和signed char、unsigned char一起）事实上定义了字节的大小。

一个字节表示一个字符的叫做窄字符，如acsii码；多个字节表示一个字符的叫宽字符，如汉字。

#### 2.2.9.const,volatile

const和volatile是类型修饰符，语法类似，在C++中合称为cv-限定符(cv-qualifier)。可以共同使用。用于变量或函数参数声明，也可以限制非静态成员函数。const表示只读类型（指定类型安全性，保护对象不被意外修改），volatile指定被修饰的对象类型的读操作是副作用（因此读取不能被随便优化合并，适合映射I/O寄存器等）。

volatile:

a、当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中，以后再取变量值时，就直接从寄存器中取值。

b、优化器在用到volatile变量时必须每次都小心地重新读取这个变量的值，而不是使用保存到寄存器里的备份。

c、volatile适用于多线程应用中被几个任务共享的变量。

#### 2.2.10.struct,class,union

用于类型声明。class是一般的类类型。struct在C++中是特殊的类类型，声明中仅默认隐式的成员和基类访问限定与class不同（struct是public，class是private）。union是联合体类型。满足特定条件类类型——POD struct或POD union可以和C语言中的struct和union对应兼容。

class还有个用途是在模版类型声明中作为表示模版类型参数或模版模版参数的语法的必要组成部分。前者也可被typename代替。

注：POD类型（Plain Old Data）,plain---代表普通类型，old---代表可以与C语言兼容。

#### 2.2.11.delete,new

delete单独使用，表示释放具有动态存储期对象，默认版本调用全局的去配器(deallocator)::operator delete和析构函数。new单独使用，表示请求分配动态存储期对象，默认版本调用全局的分配器(allocator)::operator new和指定的析构函数。和operator连用表示分别表示去配器(operator delete)和分配器(operator new)，用于释放分配器(allocator)的内存和分配内存。operator delete也在分配内存被异常中断时被调用。如果某成员函数“=delete”;表示默认删除这个函数，也不用再实现，效果类似置于private区域。但是本质绝对不相同，delete是表示此成员函数已经删除，而至于private区则此函数仍存在，只是在类外不可调用。

new、delete属于操作符，可以被重载。new表示向内存申请一段新的空间，申请失败会抛出异常。new会先调用operator new函数，再在operator new函数里调用malloc函数分配空间，然后再调构造函数。delete不仅会清理资源，还会释放空间。delete县调用析构函数，其次调用operator delete函数，最后在operator delete函数里面调用free函数。malloc申请内存失败会返回空。free只是清理了资源，并没有释放空间。

#### 2.2.12.do,for,while

循环语句的组成部分。C++支持do-while循环、for循环和while循环。C++11新增了ranged-based for循环，用:分隔声明的对象和指定循环的范围。

#### 2.2.13.数值类型

即double、float、long、int、short、signed、unsigned。

signed和unsigned作为前缀修饰整数类型，分别表示有符号和无符号。signed和unsigned修饰char类型，构成unsigned char和signed char，和char都不是相同的类型；不可修饰wchar\_t、char16\_t和char32\_t。其它整数类型的signed省略或不省略，含义不变。signed或unsigned可单独作为类型，相当于signed int和unsigned int。

double和float专用于浮点数，double表示双精度，精度不小于float表示的浮点数。long double则是C++11指定的精度不小于double的浮点数。

其它关键字表示整数类型。从占用空间大小(sizeof)来看，保证char<=short<=int<=long<=long long。注意这些都不是相同的类型，即便大小和范围都一致；各自具有unsigned版本。其中long long是C++11指定的不小于long的整数。

#### 2.2.14.if,else

条件语句的组成部分。if表示条件，之后else表示否定分支。

#### 2.2.15.enum

构成枚举类型名的关键字。C++11新增带作用域的枚举，用enum class或enum struct（两者等价）声明。

#### 2.2.16.explicit

这个关键字修饰构造函数声明，表示显式构造函数（模版），显式构造函数不参与特定的重载。

C++11从两个角度扩展了用法。其一是适用于转换函数（模版），类似构造函数，避免不需要的重载。其二是列表初始化，除非直接使用std::initializer\_list，显式构造函数被列表初始化忽略（在C++98/03中，explicit仅对单一参数调用构造函数有意义，这里打破了这个限制）。

#### 2.2.17.export

使用该关键字可实现模板函数的外部调用。对模板类型，可以在头文件中声明模板类和模板函数；在代码文件中，使用关键字export来定义具体的模板类对象和模板函数；然后在其他用户代码文件中，包含声明头文件后，就可以使用该这些对象和函数。便于导出模版，用于分离编译。当初标准委员会未经充分实践支持匆忙通过了这一决定，被EDG证明具备不现实的可用性（即便EDG唯一地实现了这一个特性，他们也表示反对）[7] 。MSVC、GCC、Clang等其它主流编译器前端都没有实现这一特性，甚至这是唯一明显约定不符合C++98/03的特性。C++11废除了这个export关键字的含义，但保留这个关键字，供以后使用[3] 。

#### 2.2.18.extern

extern意为“外来的”，是存储类声明修饰符。这个关键字在C语言中即很不明确，容易被混淆（尤其是extern inline——好在C++没有这种用法）。extern表示被修饰的声明中，名称的链接保持和之前的一致（若已有相同声明存在），或保持默认值（若之前没有相同名称的声明存在）——取决于声明的位置。在命名空间作用域（注意包括全局命名空间），extern表示外部链接；若这里省略extern，则对于const对象，默认具有内部链接；否则表示具有和命名空间一致的链接（若顶层没有嵌套未命名的命名空间，就是外部链接，否则是内部链接）。

C++11新增extern用于模版显式实例化声明，表示不在当前翻译单元实例化模版，而在别处有定义。这可以提升编译效率。

当出现extern “C”时，表示 extern “C”之后的代码按照C语言的规则去编译；当extern修饰变量或函数时，表示其具有外部链接属性，即其既可以在本模块中使用也可以在其他模块中使用。

#### 2.2.19.friend

声明友元，使其不受访问权限控制的限制。

#### 2.2.20.inline

声明定义内联函数（模版），提示编译时内联——将所调用的代码嵌入到主调函数中。注意是否内联取决于实现——编译器有权不实际内联，如果它认为这是必要的或更符合预期的目标代码质量。但inline还改变了ODR(One Definition Rule)的适用性。类似函数模版，在头文件直接定义inline函数不会视为重复定义违反ODR而编译出错。C++中，一个函数若声明inline，则每处声明都必须保证是inline，和C语言允许extern inline或static inline不同——尽管C++实现可以提供类似非标准的扩展。注意类成员函数若在类定义内给出定义则隐含inline。

C++11中，新增inline namespace，指示命名空间中的名称同时是外层命名空间直接包含的名称。这便于命名空间的版本管理，减少冲突。

inline 函数函数不能有选择、开关、循环语句。

#### 2.2.21.mutable

用于类的非静态非const数据成员，表示不受到成员函数的const的限制，可以在const成员函数中使用。mutable也是为了突破const的限制而设置的。被mutable修饰的变量，将永远处于可变的状态，即使在一个const函数中。

#### 2.2.22.namespace

表示命名空间——其中可以声明若干标识符，组成的名称与其它命名空间不冲突。

可以声明一个命名空间或命名空间别名。命名空间具有全局唯一性。

#### 2.2.23.operator

和操作符连用，指定一个重载了的操作符函数，也包括operator new和operator delete。

#### 2.2.24.private,protected,public

指定类成员或基类中的名称的访问权限控制，分别表示仅供本类使用、供本类和派生类使用、不设限制。

#### 2.2.24.register

提示声明的对象被放入寄存器中以便得到更好的性能。同inline类似，并非强制；不同的是这个提示经常被现代的编译器无视，因此C++11中被标记为过时的。

#### 2.2.25.return

子程序返回语句，终止当前函数执行，使控制流返回到主调函数的调用后。若返回类型不是void可以同时带返回值。return表示从被调函数返回到主调函数继续执行，返回时可附带一个返回值，由return后面的参数指定。return通常是必要的，因为函数调用的时候计算结果通常是通过返回值带出的。如果函数执行不需要返回计算结果，也经常需要返回一个状态码来表示函数执行的顺利与否（-1和0就是最常用的状态码），主调函数可以通过返回值判断被调函数的执行情况.

#### 2.2.26.static

和C语言类似，声明静态存储期对象，或指定一个函数的名称具有内部链接。在C++还有另一个用途——用于类作用域声明，表示声明的成员是类共有的，不需要通过类的对象访问。类的静态数据成员也具有静态存储期，除非被thread\_local修饰。可修饰变量（静态全局变量，静态局部变量），也可以修饰函数和类中的成员函数。static修饰的变量的周期为整个函数的生命周期。具有静态生存期的变量，只在函数第一次调用时进行初始化，在没有显示初始化的情况下，系统把他们初始化微0.

#### 2.2.27.sizeof

返回类型名或表达式具有的类型对应的大小,返回值为size\_t 类型，也就是usigned long。不能对函数使用（虽然可能有非标准扩展）。C++11新增支持直接对类的数据成员作为操作数（之前无法使用）。sizeof的操作数是不被求值的，利用这一点可以配合模版元编程测试类型正确性。sizeof的结果一般编译时确定，认为是常量，不必强制转换

#### 2.2.28.template

声明一个模板、模版的特化或显式实例化（另见extern）。模版用于打破类型系统的某些限制，推迟类型检查到实例化得到具体的模版实例进行以复用代码，实现泛型和参数化编程。

#### 2.2.29.this

this是一种实体，仅在类的非静态成员中使用，是指向类的对象的指针右值。静态成员函数没有this指针。

#### 2.2.30.typedef

用以给数据类型取别名。字面名义上是定义，实际只是声明——这点和C语言的说法不同。

C++11中可被using新增的用法代替。

#### 2.2.31.virtual

声明虚基类或虚函数。具有虚基类或虚函数的类是多态类(polymorphic class)，需要运行时提供支持来判断成员函数调用分派到的具体类型。

#### 2.2.32.typeid

获取表达式的类型，以std::type\_info表示结果，可能抛出std::bad\_typeid。当操作数非多态类（引用）类型在编译时即可确定结果，否则需要在运行时取得结果，即RTTI。

#### 2.2.33.typename

告诉编译器一个嵌套的限定名（包含::）中的未知的标识符是一个类型。这只在模板中需要区分依赖名称时使用。

另一种用法是在模版声明参数列表中表示模版类型参数，可被class代替。

#### 2.2.34.using

有两种基本用法：using声明和using指示（using namespace ...）。

前者是声明，引入命名空间或基类作用域内已经被声明的名称。后者引入命名空间内所有的名称。

C++11新增了一类用法，可以代替typename，同时可以配合template使用（typedef基于语法考虑未被允许加入这种用法）。

#### 2.2.35.void

特殊的“空”类型，指定函数无返回值或无参数（在参数列表中只能够唯一地使用），用于表达式转换为void类型以丢弃计算值的必要（C++11加入了discarded-value-expression显式地指出了这一点），或用于void\*指针类型。

# C++运算符优先级

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 优先级 | 运算符 | | 描述 | | 重载 | 结合 |
| 1 | **::** | | 作用域限定 | | NO | 从左至右 |
| 2 | **()** | | 函数调用、类成员初始化 | | YES |
| **[]** | | 数组元素访问 | | YES |
| **->** | | 通过指针访问成员 | | YES |
| **.** | | 通过对象访问成员 | | NO |
| **++** | **--** | 后置自增 | 后置自减 | YES |
| **const\_cast** | | const转换 | | NO |
| **dynamic\_cast** | | 动态转换 | | NO |
| **static\_cast** | | 静态转换 | | NO |
| **reinterpret\_cast** | | 底层转换 | | NO |
| **typeid** | | 类型识别 | | NO |
| 3 | **! 或 not** | | 逻辑非 | | YES | 从右至左 |
| **~ 或 compl** | | 按位取反 | | YES |
| **++** | **--** | 前置自增 | 前置自减 | YES |
| **-** | **+** | 负数 | 正数 | YES |
| **\*** | **&** | 取内容 | 取地址 | YES |
| **new new []** | **delete delete []** | 动态内存分配 | 动态内存销毁 | YES |
| **(type)** | | C风格类型强制转换 | | YES |
| **sizeof** | | 大小运算符 | | NO |
| 4 | **->\*** | | 成员指针访问运算符 | | YES | 从左至右 |
| **.\*** | | 成员对象访问运算符 | | NO |
| 5 | **\* / %** | | 乘 除 取余 | | YES |
| 6 | **+** | **-** | 加 | 减 | YES |
| 7 | **<<** | **>>** | 左移位运算 | 右移位运算 | YES |
| 8 | **< <= > >=** | | 小于 小于 等于 大于 大于 等于 | | YES |
| 9 | **== 或eq** | **!= 或 not\_eq** | Comparison equal-to | | YES |
| 10 | **& 或bitand** | | 按位与 | | YES |
| 11 | **^ 或 xor** | | 按位异或 | | YES |
| 12 | **| 或bitor** | | 按位与 | | YES |
| 13 | **&& 或 and** | | 逻辑与 | | YES |
| 14 | **|| 或 or** | | 逻辑或 | | YES |
| 15 | **? :** | | 条件运算符 | | NO | 从右至左 |
| 16 | **= += -= \*= -= \*= /= %= &= 或 and\_eq ^= 或 xor\_eq**  **|= 或 or\_eq <<= >>=** | | 赋值 加赋值 减赋值 乘赋值 除赋值  取余赋值 位与赋值 位异或赋值  位或赋值 左移赋值 右移赋值 | | YES |
| 17 | **throw** | | 抛出异常 | | NO | 从左至右 |
| 18 | **,** | | 逗号运算符 | | YES |

# 类和对象

## 1.面向对象和面向过程

面向过程：

复杂的大问题->层层分解/模块化->若干子问题->自顶向下，逐步求精

程序 ＝ 变量 980-= + 函数 （程序 ＝ 数据结构 + 算法）

问题：数据结构和算法之间没有直观的联系,理解难 修改难 重用难

面向对象：

认为现实世界是由对象组成的，用程序来模拟现实世界中对象与对象之间的交互，对象由相关的数据和操作组成，对象具有相对独立性，对外提供一定的服务。

程序 ＝ 对象 + 对象 + … + 对象

面向对象的四大特性：抽象、封装、继承、多态

## 2.类和对象的含义

从哲学角度说，先有对象，然后才有类，类和对象是“一般和特殊”这一哲学原理在程序世界中的具体体现。类是对象的抽象，对象是类的实例，相当于数据类型和变量，结构体类型和结构体变量的关系。

对对象的抽象（又叫类的提取）往往是从两个方面来考虑的：

一是特征（C++常称为“属性”，即类的数据成员），

二是功能（C++中常称为“行为或方法”，即操作数据成员的的函数）

## 3.类的定义

C++中使用关键字class定义一个类，其基本形式如下：

class 类名

{

public:

公共数据成员和函数

protected:

保护数据成员和函数

private:

私有数据成员和函数

}; //不要漏写了这个分号;

C++98规定，类成员的访问权限默认是private，不加申明的成员默认是private.数据成员的类型前面不可使用auto、extern和register等，也不能在定义时对非静态常量整型数据成员初始化，如果将float price写成float price = 0，编译器会报错。只有静态常量整型数据成员，才可以在类定义时候初始化。

当时认为，类定义中的数据定义，是一种声明，不是数据定义，当用类定义对象（变量，常量）时候，才开始定义数据。静态常量整型数据成员const static关键字：

1)不是对象的一部分

2)可以产生常量表达式，所以可以在类中初始化。

静态常量整型数据成员，能够用来当作常量表达式使用，不在内部定义的话，则该常量表达式未定义，就不能使用了。

C++11中：

（1）一般数据成员，const修饰的常量，const static修饰的整型数据成员，constexpr修饰的非整型数据成员都可以在定义时直接初始化。

（2）static修饰的静态非常量数据成员必须在类外初始化。

## 4.类的实现

类的实现就是定义其成员函数的过程，类的实现有两种方式：

1）在类定义时同时完成成员函数的定义，但要注意类内定义的成员函数(没有循环分支，开关语句)都是inline的成员函数。

2）在在类定义内部进行声明，在类定义外部实现。在类定义的外部定义成员函数时，应使用作用域操作符（::）来标识函数所属的类域，即有如下形式：

返回值类型 类名::成员函数名(参数列表)

{

函数体

}

其中，返回类型、成员函数名和参数列表必须与类定义时的函数原型一致。

## 5.类的使用

定义类之后，便可以像使用int、double等数据类型定义变量一样，创建该类的对象。由此看来，类的定义实际上是定义了一种类型，类不接收或存储具体的值，只作为生成具体对象的“蓝图”，只有将类实例化，创建对象（定义类的变量）后，系统才为对象分配存储空间。格式如下：

使用不带参数构造函数：

类名 对象名;

使用带参数的构造函数：

类名 对象名（构造参数）

类名 对象名=类名（构造参数）//调用构造函数，拷贝构造函数、析构函数来创建和销毁临时对象

使用指针：

类名 指向对象的指针名；

成员访问：

对象名.成员名 或 指针名->成员名

对象的作用域、可见域和生存期与普通变量，如int型变量的作用域、可见域和生存期并无不同，对象同样有局部、全局和类内之分，对于在代码块中声明的局部对象，在代码块执行结束退出时，对象会被自动撤销，对应的内存会自动释放（当然，如果对象的成员函数中使用了new或malloc申请了动态内存，却没有使用delete或free命令释放，对象撤销时，这部分动态内存不会自动释放，造成内存泄露）。

注意：

数据成员占据不同的内存区域(堆、栈)；成员函数共用同一内存区域(代码段)

无参数的非静态成员函数调用时必须加上空括号。

## 6.class 与 struct的区别

class的定义看上去很像struct定义的扩展，事实上，类定义时的关键字class完全可以替换成struct，也就是说，结构体变量也可以有成员函数。class和struct的唯：一区别在于：struct的默认访问方式是public，而class默认为private。struct 也可以用private关键字

提示：通常使用class来定义类，而把struct用于只表示数据对象、没有成员函数的类。

## 7.public,private,protected

1）类的一个特征就是封装，public和private作用就是实现这一目的。所以：

用户代码（类外）可以访问public成员而不能访问private成员；private成员只能由类成员（类内）和友元访问。

2）类的另一个特征就是继承，protected的作用就是实现这一目的。所以：

protected成员可以被派生类对象访问，不能被用户代码（类外）访问。

继承中的特点：

先记住：不管是否继承，上面的规则永远适用！

有public, protected, private三种继承方式，它们相应地改变了基类成员的访问属性。

1）public继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：public, protected, private（原样继承）

2）protected继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：protected, protected, private（公有变保护）

3）private继承：基类public成员，protected成员，private成员的访问属性在派生类中分别变成：private, private, private（全变私有）

## 8.对象的创建和撤销

C++为类提供了两种特殊的成员函数来完成相反的工作：

一是构造函数，在对象创建时自动调用，用以完成对象数据成员等的初始化及其他操作（如为指针成员动态申请内存空间等）；如果程序员没有显式的定义它，系统会提供一个默认的构造函数。一旦显示的定义了构造函数，系统就不会提供默认的构造函数了。

另一个是析构函数，在对象撤销时自动调用，用以执行一些清理任务，如释放成员函数中动态申请的内存等。如果程序员没有显式的定义它，系统也会提供一个默认的析构函数。

### 8.1.构造函数与初始化表达式

#### 8.1.1.构造函数

当对象被创建时，构造函数自动被调用。构造函数有一些独特的地方：构造函数的函数名与类名相同，没有返回类型和返回值，可以有函数参数或带默认值的参数，并且构造函数支持函数重载。即使是void也不能有。其主要工作有：

给对象一个标识符。

为对象数据成员开辟内存空间。

完成对象数据成员的初始化（函数体内的工作，由程序员完成）。

上述3点也说明了构造函数的执行顺序，在执行函数体之前，构造函数已经为对象的数据成员开辟了内存空间，这时，在函数体内对数据成员的初始化便是顺理成章了。构造函数有如下特点：

（1）如果用户没有显示定义构造函数，编译器自动生成的缺省构造函数是无参的；

（2）实际上，构造函数可以接收参数，也可置默认参数，但需遵守相应规则，同样需要注意二义性的问题；

（3）一旦用户定义了构造函数，系统便不再提供默认构造函数。如果还想使用无参的构造函数，必须在类定义中显式定义一个无参构造函数；

（4）如果显式的提供了拷贝构造函数，要想使用的其他构造函数，都必须显示提供。原因在于拷贝构造函数是特殊的构造函数。

（5）构造函数支持重载，在创建对象时，根据传递的具体参数决定采用哪个构造函数；

（6）构造函数是特殊的成员函数，也可以在类内定义或类内声明类外实现；

（7）大多为公有；

（8）构造函数由对象创建时，系统根据参数自动调用,虽然可以由用户显式定义但不能由用户通过对象主动调用。

（9）构造函数不能是static成员函数，const员函数，

构造函数的形式：

默认构造函数：

类名（）

{

}

自定义构造函数：

类名（参数表）

{

函数体；

}

调用构造函数定义对象：

类名 对象名（参数表）

#### 8.1.2.初始化表达式

除了在构造函数体内用赋值语句初始化数据成员外，还可以通过成员初始化表达式来完成。

成员初始化表可用于初始化类的任意数据成员（后面要介绍的static数据成员除外），该表达式由逗号分隔的数据成员表组成，初值放在一对圆括号中。将成员初始化表达式放在构造函数的头和体之间，并用冒号将其与构造函数的头分隔开，便可实现数据成员表中元素的初始化。

类名（参数表）：数据成员1（初始值）, 数据成员2（初始值），……

{

函数体；

}

每个成员在初始化表中只能出现一次。初始化的顺序不是由数据成员在初始化表中的顺序决定的，而是由数据成员在类中被申明时的顺序决定的。初始化成员列表先执行，构造函数体中的赋值语句后执行。

注意：虽然大多数情况下使用赋值语句也可以达到初始化表达式的效果，但在初始化某些特殊的数据成员时却只能使用初始化成员列表。如：引用类型的数据成员或const数据成员。

初始化表达式只能用于构造函数。如果函数在类内声明，则只需在类外定义时写初始化表达式。

### 8.2.析构函数

构造函数在创建对象时被系统自动调用，而析构函数在对象被撤销时被自动调用，相比构造函数，析构函数要简单的多。析构函数有如下特点：

（1）与类同名，之前冠以波浪号，以区别于构造函数。

（2）析构函数没有返回类型，也不能指定参数。

（3）析构函数只能有一个，不能被重载。

（4）析构函数也是特殊成员函数，可以在类内定义或者类内声明类外实现；

（5）如果用户没有显式地定义析构函数，编译器将为类生成“**缺省**析构函数”，缺省析构函数是个空的函数体，只清除类的数据成员所占据的空间，但对类的数据成员通过new和malloc动态申请的内存无能为力，因此，对于动态申请的内存，应在类的析构函数中通过delete或free进行释放，这样能有效避免对象撤销造成的内存泄漏。

对象超出其作用域被销毁时，析构函数会被自动调用。析构函数的形式如下：

~类名（）

{

函数体

}

**显示调用析构函数**

程序员不能显式调用构造函数，但却可以调用析构函数控制对象的撤销，释放对象所占据的内存空间，以更高效地利用内存。虽然可以显式调用析构函数，但不推荐这样做，因为可能带来重复释放指针类型数据成员所指向的内存等问题。

### 8.3构造函数和析构函数的调用时机

对于全局定义的对象，每当程序开始运行，在主函数main(或WinMain)接受程序控制权之前，就调用全局对象的构造函数。整个程序结束时调用析构函数。

对于局部定义的对象，每当程序流程到达该对象的定义处就调用构造函数，在程序离开局部对象的作用域时调用对象的析构函数。（栈对象）

对于关键字static定义的静态局部对象，当程序流程第一次到达该对象定义处调用构造函数，在整个程序结束时调用析构函数。

对于用new运算符创建的对象，每当创建该对象时调用构造函数，当用delete删除该对象时，调用析构函数。（堆对象）

## 9.拷贝构造函数

C++中经常使用一个常量或变量初始化另一个变量，象简单变量的初始化一样，也可以直接用一个对象来初始化另一个对象呢？答案是肯定的，以point类为例：

point pt1(2,3);point pt2=pt1;或point pt2(pt1);

上述语句用pt1初始化pt2，相当于将pt1中每个数据成员的值复制到pt2中，这是表面现象。实际上，系统调用了一个拷贝构造函数。如果类定义中没有显式定义该拷贝构造函数时，编译器会隐式定义一个缺省的拷贝构造函数，它是一个inline、public的成员函数，**缺省**的拷贝构造函数原型形式为：

类名：：类名（const 类名 &）

{

}

如：point:: point (const point &);

注意：如果不加&则调用构造函数时候会无穷的调用自身，直到函数栈满。推荐加上const,以避免右值为常量时不能满足c++的规则。

**拷贝构造函数在以下三种情况会自动调用**：

（1）当用一个已经存在的对象初始化另一个新的对象时。

（2）当实参和形参都是对象，进行形参和实参的结合时。

（3）当函数的返回值是对象，在函数调用完成时。（在将函数返回值给另一个新对象之前）

默认的拷贝构造函数并非万金油，因为其为浅拷贝（拷贝时只是简单的赋值操作，而没有开辟新空间），在一些情况下，必须由程序员显式定义缺省拷贝构造函数，（-fno-elide-constructors可以看到没有优化的具体过程）先来看一段错误代码示例：

其中语句computer comp2(comp1)等价于：

comp2.brand = comp1.brand;

comp2.price = comp1.price;

后一句没有问题，但comp2.brand = comp1.brand却有问题：经过这样赋值后，两个对象的brand指针都指向了同一块内存，当两个对象释放时，其析构函数都要delete[]同一内存块，便造成了2次delete[]，从而引发了错误。

如果类中含有指针型的数据成员、需要使用动态内存，程序员最好显式定义自己的拷贝构造函数（也就是要深拷贝），避免各种可能出现的内存错误，类外实现如下：

类名：：类名（const 类名 &引用名）

{

函数体

}

computer(const computer &cp) //自定义拷贝构造函数

{

//重新为brand开辟和cp.brand同等大小的动态内存

brand = new char[strlen(cp.brand) + 1];

strcpy(brand, cp.brand); //字符串复制

price = cp.price;

}

拷贝构造函数可以看成是一种特殊的构造函数，这里姑且区分为“拷贝构造函数”和“普通构造函数”，因此，它也支持初始化表达式。

创建对象时，只有一个构造函数会被系统自动调用，具体调用哪个取决于创建对象时的参数和调用方式。C++对编译器何时提供缺省构造函数和缺省拷贝构造函数有着独特的规定，如表8.1所示。

拷贝构造函数也支持类内定义或者类内声明类外实现。

拷贝构造函数只有一个,不能被重载。

如果显式的提供了拷贝构造函数，要想使用的其他构造函数，都必须显示提供。原因在于拷贝构造函数是特殊的构造函数。



## 10.运算符函数

### 10.1.运算符函数

运算符函数与普通成员函数类似：

（1）在类内定义运算符函数使其成为inline型，格式为：

class 类名

{

……

返回类型 operator 运算符（参数列表）

{

……

}

……

}

（2）在类内声明，类外实现，但要使用作用域限定符“::”，类外定义的基本格式为：

class 类名

{

……

返回类型 operator 运算符（参数列表）；

……

}

返回类型 类名::operator 运算符（参数列表）

{

…

}

### 10.2.赋值运算符函数

赋值运算是一种很常见的运算，如果不重载赋值运算符，编译器会自动为每个类生成一个**缺省**的赋值运算符重载函数，如下面的语句：

对象1＝对象2;

赋值运算符函数也是特殊的成员函数，可在类内定义或者类内声明类外实现，格式为：

class 类名

{

……

返回类型 & operator 运算符（const 类名 &引用名）

{

……

}

……

}

赋值运算符函数中：

首先要判断是否是自复制（ 对象1=对象1），判断的条件时this=&ref？如果相等则返回this指针的对象，如果不等且有指针成员时，要进行深拷贝（先释放目标对象的指针成员，再分配空间，最后赋值）。

实际上是完成了由对象2各个成员到对象1相应成员的复制，包括指针成员，这和拷贝构造函数有些类似，如果对象1中含指针成员，并且牵扯到类内指针成员动态申请内存时，问题就会出现。

将已存在的对象赋值给另一个已存在对象，调用赋值运算符函数

## 11.构造函数调用示例

类名 对象1＝对象2; //用已存在对象初始化新对象时，用拷贝构造函数

类名 对象1; 创建对象时，调用无参或者参数都有默认值的构造函数

类名 对象1（构造参数）；//用特定参数创建对象，调用自定义构造函数

类名 名称（）； //返回值为对象的函数声明

对象1＝对象2; //将已存在的对象赋值给另一个已存在对象，调用赋值运算符函数

类名 对象1=类名（构造参数）//先掉构造函数生成临时对象，再调用拷贝构造函数，最后调用析构函数释放临时对象

类名 对象1(对象2) //调用拷贝构造函数

**示例：**

CTest t0(); //这是函数的声明，不是实例化类

CTest t1; //缺省构造函数

CTest t2(1); //一个参数的构造函数

CTest t3(1, 2); //两个参数的构造函数

CTest t4 = 1; //等价于CTest t4(1);先用1生成一个对象，再用该对象去初始化t4，在每个函数前用explicit关键字禁止隐形转换

CTest t5 = t1; //CTest(t1);

CTest t6 = CTest(); //CTest(1); CTest(1,2);

t6 = CTest(1);

t6 = 1; //首先调用单个参数的构造函数，生成临时对象CTest(1), 然后调用赋值运算符函数

t6 = t1; //调用赋值运算符函数

## 12.this指针

一个类的所有对象共用成员函数代码段，不管有多少个对象，每个成员函数在内存中只有一个版本，每个对象都有一个自己的“this指针”，用以区分不同的对象。对于类的成员函数而言，并不是一个对象对应一个单独的成员函数体，而是此类的所有对象共用这个成员函数体。 当程序被编译之后，此成员函数地址即已确定。而成员函数之所以能把属于此类的各个对象的数据区别开, 就是靠这个this指针。函数体内所有对类数据成员的访问， 都会被转化为this->数据成员的方式。t

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。

this指针是类的一个自动生成、自动隐藏的私有成员，它存在于类的非静态成员函数中，指向被调用函数所在的对象。全局仅有一个this指针，当一个对象被创建时，this指针就存放指向对象数据的首地址。

在 C++ 中，每一个对象都能通过 this 指针来访问自己的地址。this 指针是每一个成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象。

友元函数没有 this 指针，因为友元不是类的成员。只有成员函数才有 this 指针。

使用this指针的情况：

一种情况就是，在类的非静态成员函数中返回对象本身的时候，直接使用 return \*this；另外一种情况是当参数与成员变量名相同时使用this指针，如this->n = n （不能写成n = n）。

## 13.特殊数据成员的初始化

### 13.1.const数据成员的初始化

数据成员可以由const修饰，这样，一经初始化，该数据成员便具有“只读属性”，在程序中无法对其值修改。

事实上，在构造函数体内或复制构造函数体内初始化const数据成员是非法的， const数据成员只能通过成员初始化表进行初始化。

注意：const 数据成员指针，可以在初始化列表中，也可以在构造函数中初始化。

### 13.2.引用类型数据成员的初始化

对于引用类型的数据成员，同样只能通过成员初始化表达式进行初始化。引用数据成员只有一个实例，对一个实例的改变将使所有相关对象中的所有的引用都发生改变。

### 13.3. 对象成员的初始化

类数据成员也可以是另一个类的对象,推荐在初始化列表中用构造参数直接初始化对象成员，而不是调用构造函数创建一个临时对象后再调用拷贝拷贝构造函数：。

### 13.4. static数据成员的初始化

C++允许使用static（静态存储）修饰数据成员，这样的成员在编译时就被创建并初始化的（与之相比，对象是在运行时被创建的），且其实例只有一个，被所有该类的对象共享，就像住在同一宿舍里的同学共享一个房间号一样。静态数据成员和第6章中介绍的静态变量一样，程序执行时，该成员已经存在，一直到程序结束，任何对象都可对其进行访问。

静态非常量数据成员的初始化必须在类定义之外进行，且不再包含static关键字，格式如下：

数据类型 类名::变量名 = 初始化表达式; //普通变量

数据类型 类名::对象名(构造参数); //对象变量

如float computer::total\_price = 0;

调用静态数据成员，可以直接使用类名调用,而不用通过对象。

## 14.特殊成员函数

除了构造函数、拷贝构造函数和析构函数外，其他成员函数被用来提供特定的功能，一般来说，提供给外部访问的函数称为接口，访问权限为public，而一些不供外部访问，仅仅作为内部功能实现的函数，访问权限设为private。本节主要讨论函数成员的一些特殊用法。

### 14.1.static成员函数

成员函数也可以定义成静态的，与静态成员变量一样，系统对每个类只建立一个函数实体，该实体为该类的所有对象共享。

静态成员函数体内不能使用非静态的成员变量和非静态的成员函数；只能调用静态数据成员和静态成员函数。若想访问非静态成员，应该设置函数参数为对象的引用。

静态成员函数的参数列表中不含有this指针

和静态数据成员类似，可以使用类名直接调用。

### 14.2.const成员函数

前面已经介绍了const在函数中的应用，实际上，const在类成员函数中还有种特殊的用法，把const关键字放在函数的参数表和函数体之间（与放在函数前修饰返回值不同），称为const成员函数，其特点有二：

只能读取类数据成员，而不能修改之

只能调用const成员函数，不能调用非const成员函数

如Point类中的打印函数：

void print(Point \* const this)

void print() const

void print(const Point \* const this) const

其基本定义格式为：

（1）类内定义时：

类型 函数名(参数列表) const

{

函数体

}

（2）类外定义时，共分两步：

类内声明：

类型 函数名(参数列表) const；

类外定义

类型 类名::函数名(参数列表) const

{

函数体

}

## 15. 对象的组织

有了自己定义的类，或者使用别人定义好的类创建对象，与使用int等创建普通变量几乎完全一致，同样可以const对象、创建指向对象的指针，创建对象数组，还可使用new和delete等创建动态对象。

### 15.1.const对象

类对象也可以声明为const对象，一般来说，能作用于const对象的成员函数除了构造函数和析构函数，便只有const成员函数了，因为const对象只能被创建、撤销以及只读访问，改写是不允许的。

const 类名 对象名

非const对象可以使用const成员函数, const对象不能调用非const成员函数,只能调用const成员函数,这是因为const对象是不能修改其内部数据成员的。

### 15.2.指向对象的指针

对象占据一定的内存空间，和普通变量一致，C++程序中采用如下形式声明指向对象的指针：

类名\* 指针名 [=初始值];

初始值是可选的，既可以通过取地址（&对象名）给指针初始化，也可以通过申请动态内存给指针初始化，或者干脆不初始化（比如置为NULL），在程序中再对该指针赋值。

指针中存储的是对象所占内存空间的首地址。

定义point类如备注中的代码

针对上述定义，则下列形式都是合法的：

point pt; //默认构造函数

point \*ptr = NULL; //空指针

point \*ptr = &pt; //取某个对象的地址

point \*ptr = new point(1, 2.2);//动态分配内存并初始化,调用了拷贝构造函数，构造函数，析构函数

point \*ptr = new point[5]; //动态分配5个对象的数组空间

使用指针对象：

ptr->print();

(\*ptr).print();

都是合法的。

空指针也可以访问不需初值的成员或者成员函数。

### 15.3.对象的大小(sizeof)

对象占据一定大小的内存空间。总的来说，对象在内存中是以结构形式（只包括非static数据成员）存储在数据段或堆中，对象的大小（sizeof）不一定是类中所有非static成员的大小之和，这和结构体一样涉及内存对齐的问题。在程序编译期间，就已经为static变量在静态存储区域分配了内存空间，并且这块内存在程序的整个运行期间都存在。而类中的成员函数存在于代码段中，不管多少个对象都只有一个副本。

计算对象大小有一些特殊之处需要强调：

（1）C++将类中的引用成员当成“指针”来维护，占据4或8个内存字节。

（2）如果类中有虚函数或采用虚继承时，虚析构函数除外，还会额外分配一个指针用来指向虚函数表（vtable）或虚基表，因此，这个时候对象的大小还要增加。

（3）指针成员和引用成员要根据系统的字长来确定。

（4）就算类没有一个成员（空类），其大小也不为0。因为如果为0，则无法区分不同的空类。

### 15.4.静态对象数组

对象数组和标准类型数组的使用方法并没有什么不同，也有声明、初始化和使用3个步骤。

(1)静态对象数组的声明：

类名 数组名[对象个数];

这种格式会自动调用无参或所有参数都有缺省值的构造函数，类定义要符合该要求，否则编译报错。

(2)静态对象数组可以在声明时初始化。

对于只有point(int ix,int iy){}这种没有缺省参数值的构造函数：

point pt[2]={point(1,2), point(3,4)}; //#1 正确

point pt[ ]={point(1,2), point(3,4)}; //#2 正确

point pt[5]={point(1,2), point(3,4)}; //#3 错误

语句#1和#2是正确的，但语句#3错误，因为pt的后3个元素会自动调用无参的或者所有参数都有缺省值的构造函数，但这样的构造函数不存在。

### 15.5. 为对象动态分配内存

和把一个简单变量创建在动态存储区一样，可以用new和delete为对象分配动态存储区，在拷贝构造函数一节中已经介绍了为类内的指针成员分配动态内存的相关范例，本节主要讨论如何为对象和对象数组动态分配内存。

#### 15.5.1为单个对象申请和释放动态内存

开辟单个地址单元，可以同时赋予指定初值：

指针=new类名 //开辟连续空间不清零, 但由调用的构造函数决定了初始化赋值操作

指针=new类名（）//开辟空间全部清零，再由调用的构造函数决定了初始化赋值操作

指针=new类名（构造参数）//new开辟空间未初始化，但由构造函数进行初始化赋值，不能用括号

delete 指针

此时，如果写了析构函数，并且是用new动态分配数组的内存空间的话，那么还会多开辟4个字节的内存空间，用来存放分配数组的大小。

#### 15.5.2.为对象数组申请和释放动态内存

开辟多个地址单元，不能执行赋予指定初值

指针=new 类名[元数个数]//开辟连续空间不清零，但由调用的构造函数决定了初始化赋值操作

指针=new 类名[元数个数]() //开辟空间全部清零，再由调用的构造函数决定了初始化赋值操作

delete []指针名

使用new为对象数组分配动态空间时，不能显式调用对象的构造函数，因此，对象要么没有定义任何形式的构造函数（由编译器缺省提供），要么显式定义了一个（且只能有一个）所有参数都有缺省值的构造函数。

malloc/free无法满足动态对象的要求，因为malloc和free无法像new/delete及new/delete[]那样自动调用对象的构造函数和析构函数

## 16.类中类（嵌套类）

类中不仅可以有成语函数和数据成员，还可以有成员类。

将一个类定义在另一个类中，则该类就称为这个类的成员，作于域为类内，位于public是都可以见的，其余的只有类内可见。

类中类可以采取类中实现，也可以采取类中声明，类外实现。实现格式为：

class 外层类名::类名::…………::类名

{

}

## 17.单例设计模式（singleton)

### 17.1.单例模式的实现

单例设计模式：类在内存中只能有1个对象

要求：不能是全局或者栈对象，只能是堆对象

分析：

通常情况下我们可以用一个类创建无穷多对象，这是因为创建对象时调用的构造函数是public成员函数，因而可在类外访问。

要创建栈对象则构造和析构函数都必须在public区域。

要创建堆对象，则operator new与operator delete 必须都在public区。

要求不能创建栈对象和全局对象，则构造函数和析构函数至少有一个在private区或者protected区，①若只是析构函数在private区，虽然不能创建栈对象，但仍可以在类外任意new空间，再调用构造函数完成对象初始化，又因为要创建堆对象，构造函数不能删除，因此只能将构造函数放在private区，此时仍可以开辟空间，但在类外无法调用构造函数初始化。②若只是构造函数在private区，此时类外可以开空间，但无论堆对象还是栈对象都无法初始化。

因此构造函数一定在private区域，而析构可以在也可以不在。

（1）将构造函数私有化。（此时不能创建全局对象和栈对象），此时在类外部定义栈对象就会报错。因此只能创建堆对象，但是类外直接new堆空间来创建堆对象也要调用构造函数，同样会失败。因此只能在类内部用一个公有成员函数来创建堆对象，又因为在类外创建对象时必需访问该函数，而此时对象还未创建，因此只能利用静态成员函数的特性。于是有了第二步：

（3）定义一个返回值为类指针的静态成员函数，返回开辟的空间。但是每次在类外调用该静态成员函数都会开辟新的空间，不满足只有一个对象的要求，因此需要将上一次开辟的空间保留下来（这就需要一个静态数据成员，于是有了第三步），并进行如下判断，如果指针为空，即上一次没有开辟空间，则开辟空间并附，否则直接返回上次开辟的空间，保证类在内存中只有一个实例。

（2）在类中定义一个静态的指针对象（可为私有，可为公有），并在类外初始化。

这三部经过整理后的实现步骤：

（1）在类中定义一个静态的类指针（可为私有，保护，可为公有），并在类外初始化为空

（2）将构造函数私有化，或者保护化

（3）定义一个返回值为类指针的静态成员函数，如果1中的指针对象为空，则初始化对象，以后再有对象调用该静态成员函数的时候，不再初始化对象，而是直接返回对象，保证类在内存中只有一个实例。建立对象时调用这个函数。

（4）推荐将析构函数私有化。

（5）建立一个回收内存的函数,可以是静态函数也可不是。销毁对象时调用这个函数。

### 17.2.单例模式自动回收内存的方法

前面实现了单例模式的要求，由于静态指针初始化为NULL，因此是每次创建对象，需要手动调用内存分配函数，（这是一种懒汉式的行为），而对象撤销时需要手动调用空间回收函数。效率很低。因此最好能够实现自动创建对象时自动分配内存和撤销对象时自动回收内存。

方法1：嵌套类+该类的静态对象（懒汉式或者饿汉式）

利用栈对象/全局对象/静态对象具有自动调用析构函数的属性，在类中定义个私有类，将内存回收放入该类的析构函数中，再创建一个该类的静态对象，并在类外初始化。同时当静态对象指针在类外初始化时，除了赋NULL外，还可以直接调用空间分配函数。（这是一种饿汉式的行为）

方法2：atexit + 饿汉式

将atexit放在分配内存的全局静态函数中，参数为静态内存释放函数。

#include <stdlib.h>

int atexit( void (\*func)(void) );

功能： 当程序终止执行时，函数调用函数指针func所指向的函数。可以执行多重调用(至少32个)，这些函数以其注册的倒序执行。执行成功返回零值，失败则返回非零值。

方法3：atexit + pthread\_once

在分配内存时不在进行判断，而是利用同一线程无论调用多少次pthread\_once函数都仅能执行一次分配内存操作来达到只有一个对象的目的。在分配内存后用atexit函数注册释放内存的操作。需要将指定线程设为静态成员，并在类外由 PTHREAD\_ONCE\_INIT初始化。静态指针也要初始化。

#include <pthread.h>

int pthread\_once(pthread\_once\_t \*once\_control, void (\*init\_routine)(void));

功能：无论once\_control指向的线程对pthread\_once调用多少次，都只能执行一次init\_routine指向的函数。once\_control由 PTHREAD\_ONCE\_INIT初始化。

### 17.3.单例模式的常见应用场景

好多没怎么使用过的人可能会想，单例模式感觉不怎么用到，实际的应用场景有哪些呢？以下，我将列出一些就在咱们周边和很有意义的单例应用场景。

1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。

4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

5. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

6. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

7. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

8. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

9. HttpApplication 也是单位例的典型应用。熟悉ASP.NET(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

总结以上，不难看出，单例模式应用的场景一般发现在以下条件下：

（1）资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。

（2）控制资源的情况下，方便资源之间的互相通信。如线程池等。

# C++的作用域与生存期

## 1.基本概念：

作用域：变量或对象的有效范围叫做作用域。是理论上的。C++的作用域主要有：局部作用域{}，函数作用域，类作用域、名称空间作用域，全局作用域（也叫文件作用域）。**总的来说一个大括号就是一个局部作用域，一个文件就是文件作用域或者（全局作用域）。**

可见性：标识符在某个地方能否被访问。

生存期：变量或者对象从创建到撤销的时间。

### 1.1.普通变量、对象的作用域和生存期

变量的定义格式：

存储类型 数据类型 变量名

函数或语句块内部：

auto 局部作用域，动态生存期，存放在堆栈区；(默认)

Static 局部作用域，静态生存期，存放在全局静态区；

register 局部作用域，动态生存期，放在cpu的寄存器中，register变量不能取址。

函数外部：

static 文件作用域(全局作用域)，静态生存期，存放在全局静态区；限制该变量不可被extern扩展到其他文件使用。

extern 扩大作用域；扩大其他文件定义的非static全局变量到本文件中使用（默认，实质是变量的声明而不是定义，因为不需申请空间,也不能赋予初值）

### 1.2.普通函数的作用域和生存期

函数定义格式：

存储类型 返回值类型 函数名（形参表）

static 文件作用域（全局作用域），静态生存期，存放在程序代码区；限制该函数不能被extern扩展到其他文件中使用，在C++中建议用匿名空间代替，因为static不可以修饰类。

extern 扩大作用域（默认，可以省略），将其他文件的非static全局函数扩展到当前文件使用。

## 2.类、结构体、联合体的成员的作用域与可见性

（1）类、结构体、联合体的成员的作用域只在定义范围内。公有成员在定义内部与定义外部都可见，其余成员都只有定义内部可见。

（2）就像同名的局部变量屏蔽全局变量的一样，在具有包含关系的作用域中，同名成员也存在“屏蔽”和“覆盖”，当然，也可使用作用域限定符“::”指定要使用哪一个作用域的。

## 3.类、结构体、联合体作用域与可见性

和函数一样，类、结构体、联合体是静态生存期的，也有作用域和可见性。

（1）文件作用域

定义在函数和其他类、结构体、联合体的外部的类、结构体、联合体的作用域为本文件，绝大多数的结构体、联合体、联合体的作用域是文件作用域。本文件都可见。

（2）局部作用域

定义在一个函数、类、结构体、联合体中的类、结构体、联合体，等价于一个成员，其作用域仅在外部的这个自定义数据类型内部。

公有成员在自定义数据类型内部与外部都可见，其余成员都只有内部可见。

（3）块作用域

定义在语句块中的类、结构体、联合体，其作用域和可见性仅仅限于所在语句块。

（4）就像同名的局部变量屏蔽全局变量的一样，在具有包含关系的作用域中，同名类、同名结构体、同名联合体也存在“屏蔽”和“覆盖”，当然，也可使用作用域限定符“::”指定要使用哪一个作用域的。

## 4.对象，结构体、联合体变量的生存期、作用域和可见域

对象有生存期，对象的生存期也是对象中所有非静态成员变量的生存期，这些成员变量都随着对象的创建而创建，随着对象的撤销而撤销。

对象的生存期、作用域和可见域取决于对象的创建位置，和前面关于普通变量的并无区别，这里便不在赘述。-

## 5.名称空间的作用域

具名名称空间的作用域为整个程序的，一个程序中同名的名称空间是同一个空间，即使在不同文件中。

匿名空间的作用域为本文件，同一文件的匿名空间是同一空间，但不同文件的匿名空间不是同一空间。

C++全局作用域就是一个匿名空间，但是只有显式定义的匿名空间才具有internal属性。

宏定义是编译指令不属于任何空间。

## 6.类声明和动态内存申请

类的定义一定要在定义对象之前，因为编译器要知道需要为类分配多大的内存空间，仅仅对类进行声明是不够的，如：

class B;//前向声明，减少循环依赖

B objectB; /创建B类的对象。错误

class B

{

……

}; //B类定义

但是，如果不创建类的对象，而仅仅是定义一个指向对象的指针（或引用），是可行的。如：

class B; //声明

B\* pB=NULL; //创建B类的对象。正确

B\* pC=new B; //创建B类的对象。错误

class B

{

……

}; //B类定义

一种普遍的误解是“如果对象被撤销，其占据的内存空间被释放，那么对象创建时和函数执行中通过new和malloc申请的动态内存也会被自动释放”，实际上，除非显式地用delete或free释放，申请的动态内存不会随着对象的撤销而撤销，相反，撤销了对象，却没有释放动态内存反而会引起内存泄露。

当然，在程序结束时，操作系统会回收程序所开辟的所有内存。尽管如此，还是要养成new/delete、malloc/free配对的编程习惯，及时释放已经无用的内存

# 友元

一般来说，类的私有成员只能在类的内部访问，类外的函数是不能访问它们的。但是，可以将一个函数定义为类的友元函数，这时该函数就可以访问该类的私有成员了。同时，也可以把另一个类B定义为本类A的友元类，这样B类就可以访问A类的任何成员了。

下面来具体讨论友元函数和友元类的概念。

注意：友元关系是单向的。

## 1.友元函数

在类的定义中用friend声明了一个自由函数或其他类的成员函数(public和private均可）后，这个函数称为类的友元函数。

### 1.1.自由函数作本类的友元函数

声明的基本格式为：

friend 返回值类型 函数名（参数表）;//声明外部函数

友元函数中可访问类的private成员。

用下面的比喻形容友元函数可能比较恰当，将类比作一个家庭，类的private成员相当于家庭的秘密，一般的外人是不允许探听这些秘密的，只有friend（朋友）才有能力探听这些秘密。

### 1.2.其他类的成员函数作本类的友元函数

声明的基本格式为：

friend 返回值类型 函数所属类：：函数名（参数表）;//声明外部函数

A类的成员函数作为B类的友元函数时 ，可以访B的私有成员，要注意：

①若类A的定义在类B之前，类B要前置声明，须在A内部声明，在B类定义后的外部实现。

②若类A的定义在类B之后，类A要前置声明，

③若A中有B类对象，则应定义为指针。

注意：将其他类的成员函数申明为本类的友元函数后，该友元函数并不能变成本类的成员函数。也就是说，朋友并不能变成家人。

### 1.3.友元函数的重载

要想使一组重载函数全部成为类的友元函数，必须一一声明，否则只有匹配的那个函数会成为类的友元，编译器仍将其他的当成普通函数来处理。

## 2.友元类、友元结构体、友元联合体

在类A中把类B声明为类A的友元，则类B称为友元类。B中的所有成员函数都是A的友元函数，都可以访问A中的所有成员。

B可以在A的public部分或private部分进行声明，方法如下：

friend class/struct/union<类名>; //友元类类名

在类A中把类B声明为类A的友元，注意：

①若类A定义在类B之前：需在类A前前置声明类B

②若类A定义在类B之后：需使用前置声明类A，且类B的所有访问类A数据成员的成员函数，都必须在先在类中声明，在A定义完成后才实现。

不可否认，友元在一定程度上将类的私有成员暴露出来，破坏了信息隐藏机制，似乎是种“副作用很大的药”，但俗话说“良药苦口”，好工具总是要付出点代价的，拿把锋利的刀砍瓜切菜，总是要注意不要割到手指的。

友元的存在，使得类的接口扩展更为灵活，使用友元进行运算符重载从概念上也更容易理解一些，而且，C++规则已经极力地将友元的使用限制在了一定范围内，它是**单向的、不具备传递性、不能被继承**，所以，应尽力合理使用友元。

# 运算符重载

C++对基本数据类型的数据的操作用运算符表示，其使用形式是表达式；对用户自定义的数据类型的数据的操作则用函数表示，其使用形式是函数调用。为了使对用户自定义数据类型的数据与基本数据类型的数据的操作形式一致，C++提供了运算符的重载，通过把C++中预定义的运算符重载为类的成员函数或者友元函数，使得对用户的自定义数据类型的数据或对象的操作形式与C++内部定义的类型的数据一致。

运算符是一种通俗、直观的函数，比如：int x＝2＋3;语句中的“＋”操作符，系统本身就提供了很多个重载版本：

int operator + (int，int);

double operator + (double,double);

## 1.运算符重载规则

**（1）可以重载的运算符**

算术运算符：+ - \* / % 通过自由函数重载

关系运算符： == != < > <= >=

逻辑运算符：|| && ！

单目运算符：+ - \* &

自增自减运算符：++ --

位运算符：| & ~ ^ << >>

复合赋值运算符：= += -= \*= /= %= &= |= ^= <<= >>=

空间申请和释放：new delete new[] delete[]

其他运算符：() -> ->\* , []

**（2）不能重载的运算符**

成员访问符 .

成员指针访问运算符 .\*

域运算符 ::

长度运算符 sizeof

条件运算符号 ?:

（3）操作数至少有一个是自定义类型。

int operator+(int, int);//内置的基本运算功能不能重载

（4）操作符的优先级、结合性或操作数个数不能改变

（5）不再具备短路求值特性，比如或运算时，第一个操作数为真，则无需判断第二个操作数

（6）重载操作符并不保证操作数的求值顺序 && || & ,

（7）不能臆造并重载一个不存在的运算符，如@, #，$等。

（8）[],(),=,->,只能以成员函数的方式重载,因为定义在类外时没有this指针，不满足（3）的要求。

（9）>>和<<运算符不能以成员函数的方式重载，因为成员函数形式重载隐含的this指针做左操作数。

（10）重载原则：

算术运算符推荐使用自由函数的方式重载(如果涉及对像的私有成员，则需要能够访问私有成员)，复合赋值运算符，自增，自减运算符，new、delete，推荐以返回引用类型的成员函数方式重载。逻辑运算符不推荐重载，关系运算符三者都可以。

注：重载后置自增自减运算符时，参数应写上int，表示后置。

## 2.运算符重载的方式

### 2.1.采用自由函数的重载形式

自由函数形式重载运算符，要求操作数应当有方法能够访问到自己的私有成员，因为外部函数只能访问类的public成员，不能访问私有成员。

### 2.2.采用成员函数的重载形式

成员函数形式的运算符声明和实现与成员函数类似，首先应当在类定义中声明该运算符（赋值运算符函数就是其中一种），声明的具体形式为：

返回类型 operator 运算符（参数列表）;

既可以在类定义的同时定义运算符函数使其成为inline型，也可以在类定义之外定义运算符函数，但要使用作用域限定符“::”，类外定义的基本格式为：

返回类型 类名::operator 运算符（参数列表）

{

……

}

用成员函数重载双目运算符时，左操作数不能用参数输入，而是通过隐含的this指针传入，这种做法的效率比较高

通过返回this指针可以重载赋值运算符，前缀++，--返回引用，后缀++，--返回对象，前缀效率更高，因此推荐尽可能使用前缀形式。

### 2.3.采用友元函数的重载形式

操作符还可重载为自由函数做本类友元函数形式，这将没有隐含的参数this指针。对双目运算符，友元函数有2个参数，对单目运算符，友元函数有一个参数。

重载为友元函数的自由运算符重载函数的声明格式为：

friend 返回类型operator 运算符 (参数表);

### 2.4.友员函数形式和成员函数形式的比较

对于绝大多数可重载操作符来说，两种重载形式都是允许的。但对下标运算符[] 、赋值运算符=、函数调用运算符()、指针运算符->，只能使用成员函数形式。

对于如下代码：

complex c1(1.0, 2.0), cRes;

cRes = c1 + 5; //#1

cRes = 5 + c1; //#2

友元函数形式重载的都是合法的，可转换成：

cRes = operator+(c1, 5); //#1 合法

cRes = operator+(5, c1); //#2 合法

但成员函数形式的重载，只有语句#1合法，语句#2非法

cRes = c1.operator+(complex(5)); //#1 可能合法

cRes = 5.operator+(c1); //#2 非法，5不会隐式转换成complex

### 2.5.对运算符重载的补充说明

运算符重载可以改变运算符内置的语义，如以友元函数形式定义的加操作符：

complex operator +(const complex& C1,const complex& C2)

{

return complex(C1.real-C2.real,C1.imag-C2.imag);

}

明明是加操作符，但函数内却进行的是减法运算，这是合乎语法规则的，不过却有悖于人们的直觉思维，会引起不必要的混乱，因此，除非有特别的理由，尽量使重载的运算符与其内置的、广为接受的语义保持一致。

## 3.几种特殊的运算符重载

### 3.1. 赋值运算符=

赋值运算是一种很常见的运算，如果不重载赋值运算符，赋值运算符只能重载为成员函数，编译器会自动为每个类生成一个缺省的赋值运算符重载函数，先看下面的语句：

对象1＝对象2;

实际上是完成了由对象2各个成员到对象1相应成员的复制，其中包括指针成员，这和第8章中复制构造函数和缺省复制构造函数有些类似，如果对象1中含指针成员，并且牵扯到类内指针成员动态申请内存时，问题就会出现。

注意下述两个代码的不同：

类名 对象1＝对象2; //拷贝构造函数

类名 对象1(对象2);

类名 对象1(构造参数);//构造函数

类名（构造参数）//构造函数农户

和

类名 对象1; //默认构造函数

对象1＝对象2; //赋值运算符函数

当数据成员中有指针

### 3.2. 函数调用运算符()

函数调用运算符()同样只能重载为成员函数形式。其形式为：

返回类型 operator()(arg1,arg2,……)

参数个数可以有多个，没有限制。

针对如下定义：

void computer::operator()(){};

int computer::operator()(int x){};

char computer::operator()(char x, char y){};

可以这样调用：

computer com1;

int z = com1(3200);

char c = com1(‘a’, ‘b’);

一个类如果重载了函数调用运算符operator()，就可以将该类对象作为一个函数使用，这样的类的对象也称为仿函数。函数也是一种对象，这是泛型思考问题的方式。

在我们写代码时有时会发现有些功能实现的代码，会不断的在不同的成员函数中用到，但是又不好将这些代码独立出来成为一个类的一个成员函数。但是又很想复用这些代码。写一个公共的函数，可以，这是一个解决方法，不过函数用到的一些变量，就可能成为公共的全局变量，再说为了复用这么一片代码，就要单立出一个函数，也不是很好维护。这时就可以用仿函数了，写一个简单类，除了那些维护一个类的成员函数外，就只是实现一个operator()，在类实例化时，就将要用的，非参数的元素传入类中。这样就免去了对一些公共变量的全局化的维护了。又可以使那些代码独立出来，以便下次复用。而且这些仿函数，还可以用关联，聚合，依赖的类之间的关系，与用到他们的类组合在一起，这样有利于资源的管理（这点可能是它相对于函数最显著的优点了）。如果在配合上模板技术和policy编程思想，那就更是威力无穷了，大家可以慢慢的体会。

C语言使用函数指针和回调函数来实现仿函数，例如一个用来排序的函数可以这样使用仿函数。

在C++里，我们通过在一个类中重载括号运算符的方法使用一个仿函数而不是一个普通函数。

### 3.3.下标运算符[]

下标运算符是个二元运算符，只能重载为成员函数形式，C++编译器将表达式sz[x];解释为sz.operator[](z);

一般情况下，下标运算符的重载函数原型如下：

返回类型& operator[ ](参数类型);

下标运算符的重载函数只能有一个参数，不过该参数并没有类型限制，任何类型都可以，如果类中未重载下标运算符，编译器将会给出下标运算符的缺省定义，此时，参数必须是int型，并且要声明数组名才能使用下标变量，如

computer com[3];

则com[1]等价于com.operator[](1)，如果[]中的参数类型非int型，或者非对象数组要使用下标运算符时，需要重载下标运算符[]。

### 3.4.指针运算符->的重载

->操作符重载只能重载为成员函数。

#include <iostream>

using namespace std;

class CData

{

public:

int GetLen() {return 5;}

};

class CDataPtr

{

private:

CData \*m\_pData;

public:

CDataPtr(){m\_pData = new CData;}

~CDataPtr(){delete m\_pData;}

CData \* operator->()

{

cout << "->操作符重载函数被调用." << endl;

return m\_pData;

}

};

int main()

{

CDataPtr p;

cout << p->GetLen() << endl; //等价于下面1句:

cout << (p.operator->())->GetLen() << endl;

return 0;

}

### 3.5.new和delete重载

通过重载new和delete,我们可以自己实现内存的管理策略。new和delete只能重载为类的静态运算符。而且重载时，无论是否显示指定static关键字，编译器都认为是静态的运算符重载函数。

重载new时，必须返回一个void \*类型的指针，它可以带多个参数，但第1个参数必须是size\_t类型，该参数的值由系统确定。

static void \* operator new(size\_t nSize)

{

cout << "new操作符被调用, size = " << nSize << endl;

void \* pRet = new char[nSize];

return pRet;

}

重载delete时必须返回void类型，它可以带有多个参数，第1个参数必须是要释放的内存的地址void \*，如果有第2个参数，它必须为size\_t类型。

static void operator delete(void \* pVoid)

{

cout << "delete操作符被调用." << endl;

delete [] pVoid;

}

### 3.4.输入>>输出<<的重载

>>和<<运算符只能重载为友元函数形式或者自由函数形式。因为流对象不能进行复制，所以返回值类型只能是流对象的引用,如果以成员函数形式进行重载，则其第一次参数为this指针，则左操作数就是Complex对象，而不是流对象，这与输出流运算符的要求不符，所以流运算符不能以成员函数形式进行重载

对操作符<<的重载

friend ostream& operator<<(ostream& os,const Complex& C1)

{

os<<C1.real<<"+i\*"<<C1.imag<<endl;

return os;

}

对操作符>>的重载

friend istream& operator>>(istream& is,Complex& C1)

{

is>>C1.real;

while (is.get()!='\*‘);

is>>C1.imag;

return is;

}

# 类型转换

前面已经对普通变量的类型转换进行了介绍，本节来讨论类对象和其他类型对象的转换

转换场合有：

赋值转换

表达式中的转换

显式转换

函数调用，传递参数时的转换

转换方向有：

由定义类向其他类型的转换

由其他类型向定义类的转换

## 1. 由其他类型向定义类转换

由其他类型（如int、double、自定义类型）等向自定义类的转换是由构造函数来实现的，只有当类的定义和实现中提供了合适的构造函数时，转换才能通过。什么样的构造函数才是合适的构造函数呢？主要有以下几种情况，为便于说明，假设由int类型向自定义point类转换：

（1）point类的定义和实现中给出了仅包括只有一个int类型参数的构造函数 Point pt1 = 5;等效于point pt1=point（5）；

（2）point类的定义和实现中给出了包含一个int类型参数，且其他参数都有缺省值的构造函数,或者有参数都有默认值。

（3）point类的定义和实现中虽然不包含int类型参数，但包含一个非int类型参数如float类型，此外没有其他参数或者其他参数都有缺省值，且int类型参数可隐式转换为float类型参数。

在构造函数前加上关键字explicit可以关闭隐式类型转换

## 2. 由自定义类向其他类型转换

可以通过operator int()这种类似操作符重载函数的类型转换函数来实现由自定义类型向其他类型的转换。如将point类转换成int类型等。

在类中定义类型转换函数的形式一般为：

operator 目标类型名();

有以下几个使用要点：

（1）转换函数必须是成员函数，不能是友元形式。

（2）转换函数不能指定返回类型，且要在函数体内必须用return语句以传值方式返回一个目标类型的变量。

（3）转换函数不能有参数

# 移动语义与智能指针

右值引用 (Rvalue Referene) 是 C++ 新标准 (C++11, 11 代表 2011 年 ) 中引入的新特性 , 它实现了移动语义 (Move Sementics) 和精确传递 (Perfect Forwarding)。它的主要目的有两个方面：

（1）消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。

（2）能够更简洁明确地定义泛型函数。

左值的声明符号为”&”， 为了和左值区分，右值的声明符号为”&&”。

## 1.左值与右值的定义

左值：非临时对象，可以在多条语句中使用的对象。

右值：临时对象。只在当前的语句中有效。

注意：

右值只在当前语句有效；// int a=0;,立即数都是右值

右值可以被修改；// string（）.push\_back(‘a’)

右值可以出现在=号的左边，但不能对其赋值。// (a>0?a:b)=10;

临时对象是作为右值处理的，但是如果临时对象通过一个接受右值的函数传递给另一个函数时，就会变成左值，因为这个临时对象在传递过程中，变成了命名对象。

非常量左值引用只能绑定到非常量左值，不能绑定到常量左值、非常量右值和常量右值。因为：

（1）如果允许绑定到常量左值和常量右值，则非常量左值引用可以用于修改常量左值和常量右值， 这明显违反了其常量的含义。

（2） 如果允许绑定到非常量右值，则会导致非常危险的情况出现，因为非常量右值是一个临时对象，

非常量左值引用可能会使用一个已经被销毁了的临时对象。

## 2.移动语义

右值引用是用来支持移动语义的。移动语义可以将资源 ( 堆，系统对象等 ) 从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝以及销毁，能够大幅度提高 C++ 应用程序的性能。临时对象的维护 ( 创建和销毁 ) 对性能有严重影响。

转义语义是和拷贝语义相对的，可以类比文件的剪切与拷贝，当我们将文件从一个目录拷贝到另一个目录时，速度比剪切慢很多。

通过移动语义，临时对象中的资源能够转移到其它的对象里。或者说临时对象放弃了资源的所有权。

在C++机制中，要实现拷贝语义，需要定义拷贝构造函数和拷贝赋值运算符函数；要实现移动语义，需要定义转移构造函数和转移赋值运算符函数。对于右值的拷贝和赋值会调用**转移构造函数和转移赋值运算符函数**。如果转移构造函数和转移赋值运算符函数没有定义，那么就遵循现有的机制，拷贝构造函数和赋值运算符函数会被调用。

示例程序 :

vector<string>v;

string a;

a=string(“hello”);

v.push\_back(string(“world”));

String(“Hello”) 和 String(“World”) 都是临时对象，也就是右值。虽然它们是临时的，但程序仍然调用了拷贝构造和拷贝赋值，造成了没有意义的资源申请和释放的操作。如果能够直接使用临时对象已经申请的资源，既能节省资源，又能节省资源申请和释放的时间。这正是定义移动语义的目的。

我们先定义转移构造函数。

String(String&& str)

{

std::cout << "Move Constructor is called! source: " << str.\_data << std::endl;

\_len = str.\_len;

\_data = str.\_data;

str.\_len = 0;

str.\_data = NULL;

}

和拷贝构造函数类似，有几点需要注意：

（1）参数（右值）的符号必须是右值引用符号，即“&&”。

（2）参数（右值）不可以是常量，不能加cosnt，因为我们需要修改右值。

（3）参数（右值）的资源所有权和标记必须修改。否则，右值的析构函数就会释放资源。转移到新对象的资源也就无效了。

现在我们定义转移赋值操作符。

String& operator=(String&& str)

{

std::cout << "Move Assignment is called! source: " << str.\_data << std::endl;

if (this != &str)

{

\_len = str.\_len;

\_data = str.\_data;

str.\_len = 0;

str.\_data = NULL;

}

return \*this;

}

这里需要注意的问题和转移构造函数是一样的。

增加了转移构造函数和转移复制操作符后，我们的程序运行结果为 :

Move Assignment is called! source: Hello

Move Constructor is called! source: World

由此看出，编译器区分了左值和右值，对右值调用了转移构造函数和转移赋值操作符。节省了资源，提高了程序运行的效率。

有了右值引用和移动语义，我们在设计和实现类时，对于需要动态申请大量资源的类，应该设计转移构造函数和转移赋值函数，以提高应用程序的效率。

## 3.强制移动语义move

只对右值引用才能调用转移构造函数和转移赋值函数，所有命名对象都只能是左值引用，如果已知一个命名对象不再被使用而想对它调用转移构造函数和转移赋值函数，也就是把一个左值引用当做右值引用来使用，怎么做呢？标准库提供了函数 std::move，这个函数以非常简单的方式将左值引用强制转右值引用，适用于已知一个对象不再需要时。

std::move在提高 swap 函数的的性能上非常有帮助，一般来说，swap函数的通用定义如下：

template <class T> swap(T& a, T& b)

{

T tmp(a); // copy a to tmp

a = b; // copy b to a

b = tmp; // copy tmp to b

}

有了 std::move，swap 函数的定义变为 :

template <class T> swap(T& a, T& b)

{

T tmp(std::move(a)); // move a to tmp

a = std::move(b); // move b to a

b = std::move(tmp); // move tmp to b

}

通过 std::move，一个简单的 swap 函数就避免了 3 次不必要的拷贝操作。

**综上所说：**

（1）移动语义的引入是为了解决在进行大数据复制的时候，将动态申请的内存空间的所有权直接转让出去，不用进行大量的数据移动，既节省空间又提高效率。要实现移动语义，就必须让编译器知道什么时候复制，什么时候移动语义，而这就是右值引用发挥作用的地方。移动语义可能修改右值的值，所以，右值引用参数不能是const。

（2）通过拷贝构造函数和移动构造函数来实现拷贝和移动语义。复制构造使用const &引用，而移动构造函数使用非const && 引用。

被移动语义的对象交出了所有权，为了不出现析构两次同一数据区，要将交出所有权的对象的指向动态内存区指针赋值为nullptr，即空指针，对空指针执行delete[]是合法的。

（3）编译器判断构造函数中是左值还是右值，然后调用相应的拷贝构造函数或者移动构造函数来构造数据。

（4）强制移动，就是让左值使用移动构造函数，强制让其交出所有权。使用std::move()函数。

（5）利用匿名的变量，让其交出资源所有权，避免复制数据，可以提高程序的效率，因此，如果一个临时变量再也用不着了，可以让其强制移动语义，这样，程序不用再进行大量的数据复制了，尤其是在vector作为返回值的时候。

## **4.精确传递 (Perfect Forwarding)**

精确传递是指将一组参数原封不动的传递给另一个函数。

c++中参数具有四个属性：值，类型，左值／右值，const/non-const。

精确传递就是在参数传递过程中，所有这些都不能改变。在泛型函数中，这样的需求非常普遍。

C++11 中定义的推导规则为：

右值实参为右值引用，左值实参仍然为左值引用。

一句话，就是参数的属性不变。这样也就完美的实现了参数的完整传递

## 5.临时对象的来源

C++真正的临时对象是不可见的匿名对象，不会出现在你的源码中，但是程序在运行时确实生成了这样的对象.

（1）为了使函数调用成功而进行隐式类型转换的时候。如：实参int，形参float。

（2）当函数返回对象的时候

## 6.编译器优化技术

为避免对临时对象进行不必要的拷贝，C++编译器常使用一种名为Copy Ellision（拷贝去除）的优化技术，该技术至少包括以下两项内容：

（1）返回值优化（RVO），即通过将返回值所占内存空间的分配地点从被调用函数转移至调用函数，来避免拷贝操作。（返回临时对象时，编译器**偷偷**传入接收者的引用来代替临时对象，就不再需要拷贝了）包括具名返回值优化（NRVO）与无名返回值优化（URVO），两者的区别在于返回值是具名的局部变量还是无名的临时对象。

（2）右值拷贝优化（类似移动语义），当某一个类型的临时对象被拷贝赋予同一类型的另一个对象时，通过直接利用该临时对象的方法来避免拷贝操作。这项优化只能用于右值（临时对象），不能用于左值。

## 7.资源管理

void UseFile(char const\* fn)

{

FILE\* f = fopen(fn, “r”);  // 获取资源

…… // 使用资源

if (!g()) { fclose(f); return; }

// ...

if (!h()) { fclose(f); return; }

// ...

fclose(f);           // 释放资源

}

用于释放资源的代码需要在不同的位置重复书写多次。

如果再加入异常处理，fclose(f)情况会变得更加复杂。

## 8.资源管理--RAII

RAII(Resource Acquisition Is Initialization)是一种由 C++创造者 Bjarne Stroustrup 提出的， 利用对象生命周期管理程序资源（包括内存、文件句柄、锁等）的技术。

使用 RAII 时，一般在资源获得的同时构造对象， 在对象生存期间，资源一直保持有效；对象析构时，资源被释放。

关键：要保证资源的释放顺序与获取顺序严格相反。

RAII类的常见特征

（1）在构造时初始化资源， 或托管已构造的资源

（2）析构时释放资源

（3）要保证资源的释放顺序与获取顺序严格相反。

（4）一般不允许复制或赋值(对象语义) - 值语义

（5）提供若干访问资源的方法

（6）禁止隐式转换。

RAII的本质是用栈对象来管理资源，因为栈对象在离开作用域时，会自动调用析构函数

## 9.智能指针(Smart Pointer)

智能指针是存储指向堆对象指针的类

在面对异常的时候格外有用，因为他们能够确保正确的销毁动态分配的对象

C++11提供了以下几种智能指针,位于头文件<memory>，它们都是类模板

std::auto\_ptr(复制/赋值)

std::unique\_ptr c++11

std::shared\_ptr c++11

std::weak\_ptr c++11

**使用智能指针的原则：**一个裸指针只交给一个智能指针进行托管，托管以后的所有操作都通过智能指针来进行。

### 9.1.auto\_ptr

auto\_ptr在构造时获取对某个对象的所有权(ownership),在析构时释放该对象,要求其对“裸”指针的完全占有性,在拷贝构造或赋值操作时,会发生所有权的转移。

**本身存在缺陷,通常不使用。**

（1）不要使用auto\_ptr对象保存指向静态分配对象的指针。否则，当auto\_ptr对象本身被撤销时，它将试图删除指向非动态分配对象的指针，导致未定义的行为。

int a=1;

auto\_ptr<int> ap(&a); //编译没有问题，会导致未定义行为

（2）不要使两个auto\_ptr对象指向同一对象。

auto\_ptr<int> ap1(new int (1024));

auto\_ptr<int> ap2(ap1.get());

（3）不要使用auto\_ptr对象保存指向动态分配数组的指针。从源代码中可以看出，它用的是delete操作符，而不是delete [ ] 操作符

（4）不要将auto\_ptr对象存储在容器中。因为auto\_ptr的复制和赋值具有破坏性。不满足容器要求：复制或赋值后，两个对象必须具有相同值。

（5）构造函数的explicit关键词有效阻止从一个“裸”指针隐式转换成auto\_ptr类型。

（6）两个auto\_ptr对象赋值或者拷贝时，原对象会失效。

### 9.2.unique\_ptr

unique\_ptr是一个独享某个对象的所有权的智能指针，它提供了一种严格语义上的所有权，包括：

（1）拥有它所指向的对象

（2）无法进行复制、赋值操作

（3）保存指向某个对象的指针，当它本身被删除释放的时候，会使用给定的删除器释放它指向的对象

（4）具有移动(std::move)语义，可做为容器元素

### 9.3.shared\_ptr

shared\_ptr是一个引用计数智能指针，用于共享某个对象的所有权。

（1）引进了一个计数器shared\_count,用来表示当前有多少个智能指针对象共享指针指向的内存块

（2）析构函数中不是直接释放指针对应的内存块,如果shared\_count大于0则不释放内存只是将引用计数减1,只有计数等于0时释放内存

（3）复制构造与赋值操作符只是提供一般意义上的复制功能,并且将引用计数加1.

问题：循环引用，会引起内存泄漏。

class Parent;

class Child

{

public:

Child(){ cout << "Child()" << endl; }

~Child(){ cout << "~Child()" << endl;}

shared\_ptr<Parent> \_parentPtr;

};

class Parent

{

public:

Parent(){ cout << "Parent()" << endl; }

~Parent() { cout << "~Parent()" << endl; }

shared\_ptr<Child> \_childPtr;

};

int main(void)

{

shared\_ptr<Child> child(new Child);

shared\_ptr<Parent> parent(new Parent);

cout << "child's use\_count() = " << child.use\_count() << endl;

cout << "parent's use\_count() = " << parent.use\_count() << endl;

cout << endl;

child->\_parentPtr = parent;//引用计数加１

parent->\_childPtr = child;

cout << "child's use\_count() = " << child.use\_count() << endl;

cout << "parent's use\_count() = " << parent.use\_count() << endl;

return 0;

}

解决方案：利用share\_ptr可直接向weak\_ptr赋值的特点，将一个类中的share\_ptr成员换成weak\_ptr。

### 9.4.weak\_ptr

强引用，只要有一个引用存在，对象就不能被释放

弱引用，并不增加对象的引用计数，但它知道对象是否存在。如果存在，提升为shared\_ptr成功；否则，提升失败。

weak\_ptr 是弱引用智能指针，而shared\_ptr是强引用智能指针

通过weak\_ptr访问对象的成员的时候，要提升为shared\_ptr。

## 10.辅助类

### 10.1.default\_delete类

模板

template< class T > struct default\_delete

template< class T > struct default\_delete<T[]>

默认删除器，指正删除数组或者单个对象。对于一些特别的指针如FILE，应自定义删除器（函数对象）。

### 10.2.bad\_weak\_ptr类

模板

class bad\_weak\_ptr;

bad\_weak\_ptr是std :: weak\_ptr作为std :: shared\_ptr的构造函数参数时，如果std :: weak\_ptr引用已经销毁对象时，发生异常抛出的对象的类型

### 10.3.enable\_shared\_from\_this

模板

template< class T > class enable\_shared\_from\_this;

从类内部获取指向本对象的智能指针

shared\_from\_this（）方法：返回指向该对象的shared\_ptr指针。

### 10.4.owner\_less

模板

template< class T >struct owner\_less; /\* undefined \*/

# 写时复制COW与小字符串优化SSO

## 1.COW思想

copy-on-write（以下简称COW）是一种很重要的优化手段。它的核心思想是懒惰处理多个实体的资源请求，在多个实体之间共享某些资源，直到有实体需要对资源进行修改时，才真正为该实体分配私有的资源。适合于长字符串。

## 2.COW经典应用

一个经典应用是Linux内核在进程fork时对进程地址空间的处理。由于fork产生的子进程需要一份和父进程内容相同但完全独立的地址空间，一种做法是将父进程的地址空间完全复制一份，另一种做法是将父进程地址空间中的页面标记为”共享的“（引用计数+1），使子进程与父进程共享地址空间，但当有一方需要对内存中某个页面进行修改时，重新分配一个新的页面（拷贝原内容），并使修改进程的虚拟地址重定向到新的页面上。

另一个经典应用是std::string类

GNU-GCC 4.x.x COW

GNU-GCC 5.x.x SSO

VC++ SSO

C++曾在性能问题上被广泛地质疑和指责过，为了提高性能，STL中的许多类都采用了Copy-On-Write技术。这种偷懒的行为的确使用STL的程序有着比较高的性能。

## 3.string的COW

string类中有一个私有成员，其实是一个char\*，用以记录从堆上分配的内存的地址，其在构造时分配内存，在析构时释放内存。

因为是从堆上分配内存，所以string类在维护这块内存上是格外小心的，string类在返回这块内存地址时，只返回const char\*，也就是只读的：

const char\* c\_str() const;

如果要写，则只能通过string提供的方法进行数据的改写。

### 3.1.string的COW感性认识

int main()

{

string str1 = “hello world”;

string str2 = str1;

printf (“Sharing the memory:/n”);

printf (“/tstr1‘s address: %p/n”, str1.c\_str() );

printf (“/tstr2’s address: %p/n”, str2.c\_str() );

str1[1]=‘q’;

str2[1]=‘w’;

printf ("After Copy-On-Write:/n");

printf ("/tstr1's address: %p/n", str1.c\_str() );

printf ("/tstr2's address: %p/n", str2.c\_str() );

return 0;

}

可以看到在修改前，str1.c\_str()与str2.c\_str()的地址是一样的，但是在str1.c\_str()修改后，地址就不一样了。这就是COW的思想：在多个实体之间共享某些资源，直到有实体需要对资源进行修改时，才真正为该实体分配私有的资源。

### 3.2.string的COW原理

#### 3.2.1.COW的原理

Copy-On-Write使用了“引用计数”,必然有一个记录引用计数的变量，当第一个string对象str1构造时，string的构造函数会根据传入的参数从堆上分配内存，当有其它string对象复制str1时，这个变量自动加1，当有对象析构时，这个变量会减1，直到最后一个对象析构时，变量为0，此时，程序才会真正的释放这块从堆上分配的内存

问题的关键在于：RefCnt该存在在哪里？如果存放在string类中，那么每个string的实例都各自拥有自己的RefCnt，根本不能共有一个 RefCnt，如果是声明成全局变量，或是静态成员，那就是所有的string类共享一个了，这也不行。

#### 3.2.2.string类共享内存的时机

1）用一个已存在的对象构建一个新对象时（调用拷贝构造函数）

2）以一个已存在对象对另一个已存在对象赋值（调用赋值运算符函数）

思路：

只需要在string类的拷贝构造函数或者赋值运算符函数中做点处理，让其引用计数累加

#### 3.2.3.string类COW触发时机

在共享同一块内存的类发生内容改变时，才会发生Copy-On-Write。

比如string类的 []、=、+=、+、操作符赋值，还有一些string类中诸如insert、replace、append等成员函数。

#### 3.2.4.COW触发过程

if  ( --RefCnt>0 )

{

char\* tmp =(char\*) malloc(strlen(\_Ptr)+1);

strcpy(tmp, \_Ptr);

\_Ptr = tmp;

}

引用计数RefCnt 大于1，表示这个内存是被共享的

#### 3.2.5.COWrite的具体实现

String类创建的对象的内存是在堆上动态分配的，既然共享内存的各个对象指向的是同一个内存区，那我们就在这块共享内存上多分配一点空间来存放这个引用计数RefCnt。

这样一来，所有共享一块内存区的对象都有同样的一个引用计数

当调用拷贝构造函数或赋值运算符函数时，这个内存的值就会加1。而在内容修改时，string类为查看这个引用计数是否大于１，如果refcnt大于1，表示有人在共享这块内存，那么自己需要先做一份拷贝，然后把引用计数减去1，再把数据拷贝过来。

## 4.COW技术优点

一方面减少了分配（和复制）大量资源带来的瞬间延迟（注意仅仅是latency，但实际上该延迟被分摊到后续的操作中，其累积耗时很可能比一次统一处理的延迟要高，造成throughput下降是有可能的）。

另一方面减少不必要的资源分配。（例如在fork的例子中，并不是所有的页面都需要复制，比如父进程的代码段(.code)和只读数据(.rodata)段，由于不允许修改，根本就无需复制。而如果fork后面紧跟exec的话，之前的地址空间都会废弃，花大力气的分配和复制只是徒劳无功。）

## 5.COW存在的问题

### 5.1.线程安全问题

想要实现COW，必须要有引用计数（reference count）。string初始化时rc=1，每当该string赋值或者拷贝给了其他sring，引用计数加1。当需要对string做修改时，如果引用计数大于1，则重新申请空间并复制一份原字符串的拷贝，然后引用计数减1。当引用计数减为0时，释放原内存。

基于“共享”和“引用计数”计数的COW在多线程环境下必然面临线程安全的问题。std::string是线程安全的吗？ 答案是：不是。

从在多线程环境下对共享的string对象进行并发操作的角度来看，std::string不是线程安全的，也不可能是线程安全的，像其他STL容器一样。

C++11之前的标准对STL容器和string的线程安全属性不做任何要求，甚至根本没有线程相关的内容。即使是引入了多线程编程模型的C++11，也不可能要求STL容器的线程安全：因为线程安全意味着同步，同步意味着性能损失，贸然地保证线程安全必然违背了C++的哲学：Don't pay for things you don't use.

但从不同线程中操作“独立”的string对象来看，std::string必须保证线程安全。因为COW的实现使两个逻辑上独立的string对象在物理上共享同一片内存，因此必须实现逻辑层面的隔离。C++0x草案(N2960)中就有这么一段：

The C++0x draft (N2960) contains the section "data race avoidance" which basically says that library components may access shared data that is hidden from the user if and only if it activly avoids possible data races.

简单说来就是：你瞒着用户使用共享内存是可以的（比如用COW实现string），但你必须负责处理可能的竞态条件。

而COW实现中避免竞态条件的策略就是：

（1）只对引用计数增减操作为原子操作；

（2）需要修改时，先分配和复制，后将引用计数-1（当引用计数为0时负责销毁）。

#### 5.1.2.原子操作带来的问题

不同的体系结构一般会有不同的底层原语以支持原子操作。如Intel CPU本身就引入了#LOCK指令前缀，该前缀允许置于指定的操作（如算法指令、逻辑指令、bit指令、exchange指令等）之前使用，如lock inc会在执行inc指令时锁总线（锁定包含目标地址的一片内存区域，防止其他CPU在此期间的并发访问），从而序列化对同一地址的访问。

比起mutex之类的同步手段，原子操作自然要轻上不少，但比起普通的算术指令，依然算得上完全的重量级：

（1）系统通常会lock住比目标地址更大的一片区域，影响逻辑上不相关的地址访问。

（2）lock指令具有”同步“语义，会阻止CPU本身的乱序执行优化。

Intel Developer's Manual vol 3的chapter 8 : Multiple-Processor Management中就有提到：

"Locked instructions can be used to synchronize data written by one processor and read by another processor."

也就是会等待之前发出的load和store指令的完成（由于CPU store buffer的存在，如果数据之前没有依赖，不需要等待load和store的结果）

（3）两个CPU对同一个地址进行原子操作，必然会导致缓存失效。SMP系统中由于Cache一致性协议的存在，一个CPU对共享内存的修改必然会invalidate另一个CPU对该地址的cache，最终导致两个CPU对同一片内存不断”争夺“（cache不断被对方invalidate，需要重新从内存中读取），这是多线程编程中经典的False Sharing问题。

归根结底，COW为了保证“线程安全”而使用了原子操作，而原子操作本身的效率并不十分高。而且在多线程环境下，多个CPU对同一个地址的原子操作开销更大。COW中“共享”的实现，反而影响了多线程环境下string“拷贝”的性能，并不划算。

#### 5.1.2.操作顺序带来的问题

为了避免竞态条件，在需要修改string时，如果引用计数>1，必须先分配和拷贝，然后才能将引用计数减1。（而不能先减1再拷贝）

某些条件下，这样的操作序列会导致不必要的额外操作：

string A在线程1中访问，string B在线程2中访问，string A 和 string B 共享同一片内容（引用计数 = 2）。假设当线程1操作string A时线程2恰好也在操作string B，双方发现该string的内容是共享的，都遵守先分配复制，后减引用计数的执行序列。（最终会有一方发现rc=0，销毁原string内容）。

到此为止，COW一共进行了3次内存分配和复制（初始化时1次，修改时2次）和1次内存释放

实际上，如果没有使用COW技术，从string的初始化到目前为止也只进行了2次内存分配和复制（都是在初始化时进行）

### 5.2.“失效”问题：草木皆兵！

假设当前的string实现是COW，考虑下面的代码：

std::string a ="some string";

std::string b =a;

assert(b.data() == a.data());//没有触发cow

//宏assert(),如果表达式的结果为零，宏写错误信息到STDERR并退出程序执行

std::cout << b[0] << std::endl;

assert(b.data() != a.data()); // 触发cow

程序仅仅以”只读“的方式访问了b中一个字符，但b已经进行了一份私有的拷贝，a与b不再共享同一片内容。

由于string的operator[]和at()会返回某个字符的引用，而判断程序是否更改了该字符实在是超出string本身能力范围的东西，为了保证COW实现的正确性，string只得统统认定operator[]和at()具有修改的“语义”。

解决方法：

由于s[index]=’a’和cout<<s[index]，cow不能判别是否是写操作，因此只能认定operator[]和at()具有修改的“语义”。为了使其能够区分修改和访问，就必须对”=”和”<<”运算符进行重载，而s[index]的返回值char 类型，根据运算符重载的规定，操作数中至少要有一个为自定义数据类型，因此必须使s[index]的返回值为自定义类型。要获取s[index]，需要两个参数s本身和下标index。因此这个类的构造函数应该有这两个参数, 有可能要要发生修改，因此参数s应该作为非常量引用。又由于要对外界隐藏，应将类放在private区域。并且重载operator=。

## 6.结论

（1） string的COW实现确有诸多的弊端，并不如想象中那般美好，也因此受到了Visual C++和clang++的抛弃，转而使用实现简单，且对小字符串更友好的SSO实现。

（2）Alexandrescue在他的“Scalable Use of STL”中建议对性能敏感的程序实现自己的string，比如针对string的长度进行选择优化（短字符串SSO，中等长度eager copy，长字符串COW），以及提供更加友好的接口等，并预期至少会有10%的整体性能提升。但我觉得，这实在不是普通程序员该干的事：暂且不论10%的底限能否达到，维护非标准的接口本身就是一笔重大的开销。

（3）虽然我们的选择不多，但了解COW的缺陷依然可以使我们优化对string的使用：尽量避免在多个线程间false sharing同一个“物理string“，尽量避免在对string进行只读访问（如打印）时造成了不必要的内部拷贝。

# 继承与派生

面向对象程序设计的一个重要特点就是可以在既有类的基础上定义新的类，而不用将既有类的内容重新书写一遍，这称为“继承”（inheritance），既有类称为“基类”或“父类”，在它的基础上建立的类称为“派生类”或“子类”，在本章节的描述中，统一使用“基类”和“派生类”。

## 1.继承的概念

### 1.1.派生类定义

继承的定义形式一般如下：

class 派生类 : [public|private|protected] 基类 //缺省为private

{

[private|private|protected]:

数据成员；

成员函数；

};

派生类生成过程包含3个步骤：

吸收基类的成员

改造基类的成员

添加自己新的成员

### 1.2.继承的层次性

任何一个类都可以派生出新类，派生类还可以再派生出新的类，因此，基类和派生类是相对而言的。一个基类可以是另一个基类的派生类，这样便构建了层次性的类结构。

## 2.继承方式与规则

### 2.1.继承的方式

public继承：基类的公有和保护成员原样传给派生类，私有成员被屏蔽

protected继承：基类公有和保护成员都变成派生类的保护成员，私有成员被屏蔽。

private继承：基类公有和保护成员都变成派生类的私有成员，私有成员被屏蔽。

### 2.2.无法继承的成员和关系

不论何种继承方式，下面这些成员都不能被继承：

（1）构造函数、析构函数

（2）用户重载的new 、delete、= 运算符

（3）友元关系(单向性，不具备传递性)

### 2.3.派生类权限

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **继承方式** | **基类** | **基类内部访问性** | **基类外部访问性** | **派生类** | **派生类内部访问性** | **派生类外部访问性** |
| 公有继承 | public | 可以访问 | 可以访问 | public | 可以访问 | **可以访问** |
| protected | 可以访问 | 不可访问 | protected | 可以访问 | 不可访问 |
| private | 可以访问 | 不可访问 | 被屏蔽 | **不可访问** |  |
| 保护继承 | public | 可以访问 | 可以访问 | protected | 可以访问 | 不可访问 |
| protected | 可以访问 | 不可访问 | protected | 可以访问 | 不可访问 |
| private | 可以访问 | 不可访问 | 被屏蔽 | **不可访问** |  |
| 私有继承 | public | 可以访问 | 可以访问 | private | 可以访问 | 不可访问 |
| protected | 可以访问 | 不可访问 | private | 可以访问 | 不可访问 |
| private | 可以访问 | 不可访问 | 被屏蔽 | **不可访问** |  |

（1）无论哪种继承方式，在派生类内部，只有基类的私有成员不可访问。

（2）无论哪种继承方式，在派生类外部，只有基类的public成员被公有继承时可以访问。

## 3.多基派生

派生类只有一个基类时，称为单基派生，在实际运用中，我们经常需要派生类同时具有多个基类，这种方法称为多基派生或多重继承。

### 3.1.多基派生与二义性问题

在C++中，声明和定义多基派生类与声明单基派生类的形式类似，只需将要继承的多个基类用逗号分开即可

class 派生类名:继承方式1 基类名1,继承方式2 基类名2,…,继承方式n 基类名n

{

private:

新增私有成员列表;

public:

新增公开成员列表;

protected：

新增保护成员列表;

};

一般来说，在派生类中对基类成员的访问应当具有唯一性，但在多基继承时，如果多个基类中存在同名成员，当在派生类内部或者外部访问同名成员时，就会造成编译器无从判断到底要访问的哪个基类中的成员，称为访问基类成员的二义性问题。

解决放法：

使用作用域限定符指明要访问的的同名成员具体属于哪一个基类。

### 3.2共同基派生与二义性问题

多基派生中，如果在多条继承路径上有一个共同的基类，则称为共同基派生， 当在派生类内部或者外部访问该基类的成员时，就会造成编译器无从判断从哪条继承路径访问该基类中的成员，称为访问基类成员的路径二义性问题。（实质仍然是直接基类中有同名成员）

解决方法

（1）使用关键字virtual将共同基类A声明为虚基类，可有效解决上述问题。在定义由共同基类直接派生的类（类B和类C）时，使用下列格式定义：

class 派生类名 : virtual 派生方式 基类名

{

//类定义

};

（2）使用作用域限定符指明要从那一条路径访问

### 3.3.共同基派生二义性与多基派生二义性不同

尽管看起来很相似，但共同基派生和多基派生带来的二义性有些细微的差别：

（1）多基派生的二义性主要是同名成员的二义性，通过加作用域限定符来解决。

（2）共同基派生的二义性则是共同基类成员的多重拷贝带来的存储二义性，使用virtual派生来解决。

另外，二义性的检查是在访问权限检查之前进行的，成员的访问权限是不能消除二义性的

## 4.派生类的构造函数和析构函数

派生时，构造函数和析构函数是不能继承的，为了对基类成员进行初始化，必须对派生类重新定义构造函数和析构函数，并在构造函数的初始化列表中调用基类的构造函数。创建派生类对象时，系统**首先通过派生类的构造函数来调用基类的构造函数，**完成基类成员的初始化，而后对派生类中新增的成员进行初始化。

|  |  |
| --- | --- |
| 派生类构造函数 | 对基类的要求 |
| 缺省 | 或缺省、或有无参构造函数、或有全部参数有默认值的有参构造函数 |
| 显式定义构造函数（不显式调用基类构造函数） | 或缺省、或有无参构造函数、或有全部参数有默认值的有参构造函数 |
| 显式定义构造函数（显式调用基类构造函数） | 拥有被调用的构造函数，基类必须显式的给出。 |

### 4.1.单基派生类的构造函数和析构函数

#### 4.1.1单基派生类的构造函数

派生类构造函数的一般格式为：

派生类名(总参数表): 基类构造函数(参数表)

{

//函数体

};

必须将基类的构造函数放在派生类的初始化表达式中，以调用基类构造函数完成基类数据成员的初始化，派生类构造函数实现的功能，或者说调用顺序为：

（1）完成对象所占整块内存的开辟，由系统在调用构造函数时自动完成。

（2）调用基类的构造函数完成基类成员的初始化。

（3）若派生类中含对象成员、const成员或引用成员，则必须在初始化表中完成其初始化。

（4）派生类构造函数体执行

#### 4.1.2.单基派生类的析构函数

当对象被删除时，派生类的析构函数被执行。析构函数同样不能继承，因此，在执行派生类析构函数时，基类析构函数会被自动调用(与虚函数)。

执行顺序是先执行派生类的析构函数，再执行基类的析构函数，这和执行构造函数时的顺序正好相反。

### 4.2.多基派生类的构造函数和析构函数

多基派生时，派生类的构造函数格式如（假设有N个基类）：

派生类名(总参数表)：基类名1(参数表1)，基类名2(参数表2)，……，基类名N(参数表N)

{

//函数体

}

和前面所讲的单基派生类似，总参数表中包含了后面各个基类构造函数需要的参数。

多基派生和单基派生构造函数完成的任务和执行顺序并没有本质不同，唯一一点区别在于首先要执行所有基类的构造函数，再执行派生类构造函数中初始化表达式的其他内容和构造函数体，各基类构造函数的执行顺序与其在初始化表中的顺序无关，而是由定义派生类时指定的基类顺序决定的。

析构函数的执行顺序同样是与构造函数的执行顺序相反。

### 4.3.派生类的构造与析构总结

首先通过派生类的构造函数来调用基类的构造函数，完成基类成员的初始化，而后对派生类中新增的成员进行初始化。

（1）如果派生类还包括成员对象，则对成员对象的构造函数的调用，仍然在初始化列表中进行，此时，当创建派生类的一个对象时，首先基类构造函数被调用，成员对象所在类构造函数次之（在有多个成员对象的情况下，成员对象的初始化顺序取决于他们在派生类中的声明顺序，与初始化列表中的顺序无关），最后执行派生类构造函数的函数体。

（2）如果是多继承，在调用基类构造函数的时候，处于同一层次的各基类构造函数的调用顺序取决于定义派生类时所指定的继承基类顺序，与派生类构造函数中所定义的初始化列表顺序无关。

（3）如果派生类有一个虚基类作为祖先类，则需要在派生类的构造函数中显式调用虚基类的构造函数，如果未列出则表明调用的是虚基类的无参构造函数，不管初始化列表中次序如何，对虚基类构造函数的调用总是先于普通基类的构造函数。

（4）继承机制下析构函数的调用顺序：

先调用派生类的析构函数

然后调用派生类中成员对象的析构函数

再调用普通基类的析构函数

最后调用虚基类的析构函数

## 5. 派生类对基类成员的覆盖

oversee 隐藏：基类与派生类同名成员（相同纯虚函数除外）。只能以对象.基类名::成员的方式访问。

override 覆盖：基类与派生类有相同的虚函数 。

overload 重载：同一个类，同名函数，但参数不同（类型，顺序，个数）

## 6.组合与继承的区别

### 6.1.定义

继承：A是B的一种（a kind of ），并且B的所有行为对A都适用，这是A继承自B。

组合：A是B的“一部分”(a part of)，则不允许B从A派生，而是要用A和其它东西组合出B。

### 6.2.继承的优点和缺点

优点：

（1）易于新的实现，因为其大多数可继承而来。

（2）易于修改或扩展那些被复用的实现。

缺点：

（1）破坏了封装性，因为这会将父类的实现细节暴露给子类。

（2）“白盒”复用，因为父类的内部细节对于子类而言通常是可见的。

（3）高耦合，基类的实现更改时，派生类也不得不随之更改。

（4）从基类继承来的实现将不能在运行期间进行改变。

### 6.3.组合的优点和缺点

优点：

（1）封装性好。

（2）“黑盒”复用，因为被包含对象的内部细节对外是不可见。

（3）低耦合，实现上的相互依赖性比较小。（被包含对象与容器对象之间的依赖关系比较少）

（4）通过获取指向其它的具有相同类型的对象引用，可以在运行期间动态地定义（对象的）组合。

（5）每一个类只专注于一项任务。

（6）容器类仅能通过被包含对象的接口来对其进行访问。

缺点：

（1）导致系统中的对象过多。

（2）为了能将多个不同的对象作为组合块（composition block）来使用，必须仔细地定义接口。

### 6.4两者的选择

（1）is-a关系用继承表达，has-a关系用组合表达

（2）继承体现的是一种特殊化的概念，而组合则是一种组装的概念

（3）当我们只想重用原类型作为新类型的内部实现，则选择组合，如果我们不仅想重用原类型作为新类型的内部实现还想重用原类型的接口，那就用继承。

（4）优先使用（对象）组合，而非（类）继承

## 7.基类对象与派生类对象间的相互转换

“类型适应”是指两种类型之间的关系，说A类适应B类是指A类的对象能直接用于B类对象所能应用的场合，从这种意义上讲，派生类适应于基类（也叫里氏替换原则），派生类的对象适应于基类对象，派生类对象的指针和引用也适应于基类对象的指针和引用。

可以把派生类的对象赋值给基类的对象

可以把派生类的对象赋值给基类的引用

可以把派生类的对象的指针赋值给指向基类对象的指针 (向上转型)

也就是说如果函数的形参是基类对象或者基类对象的引用或者基类对象的指针类型，在进行函数调用时，相应的实参可以是派生类对象

## 8.派生类对象之间的拷贝与赋值

如果用户定义了派生类的拷贝构造函数或者重载了派生类的赋值运算符=，则在用已有派生类对象初始化新的派生类对象时，或者在派生类对象间赋值时，将会执行用户定义的派生类的拷贝构造函数或者重载赋值函数，而不会再自动调用基类的拷贝构造函数和基类重载的赋值运算符，这时，通常需要用户在派生类的拷贝构造函数或者派生类的赋值函数中手动显式调用基类的拷贝构造或赋值运算符函数。

否则，派生部分执行缺省操作，基类部分执行基类定义的操作。

# 多态

多态性是面向对象设计语言的基本特征之一。仅仅是将数据和函数捆绑在一起，进行类的封装，使用一些简单的继承，还不能算是真正应用了面向对象的设计思想。多态性是面向对象的精髓。多态性可以简单地概括为“一个接口，多种方法”，前面讲过的函数重载就是一种简单的多态，一个函数名（调用接口）对应着几个不同的函数原型（方法）。

多态是指对于同一个消息、同一种调用，在不同的场合，不同的情况下，执行不同的行为。

多态性分为静态多态性和动态多态性，其中函数重载和运算符重载属于静态多态, 虚函数属于动态多态。

## 1.联编

确定程序中的操作调用（函数调用）与执行该操作（函数）的代码段之间的映射关系称为联编（binding）。联编分为静态联编和动态联编。

静态联编：在编译过程中进行的联编叫静态联编（static binding）或早期联编（early binding）,C++通过函数或运算符重载来实现静态联编。

动态联编：必须在程序运行时才能完成的联编叫动态联编（dynamic binding或晚期联编（late binding），C++通过虚函数，来实现动态联编

### 2.虚函数的声明和定义

虚函数的定义很简单，只要在**非静态**成员函数原型前加一个关键字virtual即可。

如果一个成员函数在基类中定义为虚函数，那么，它在所有派生类中也保持为虚函数；即使在派生类中省略了virtual关键字，也仍然是虚函数。

派生类中要根据需要对虚函数重定义，要求与虚函数声明时完全相同：

（1）与基类的虚函数有相同的参数个数；

（2）与基类的虚函数有相同的参数类型；

（3）与基类的虚函数有相同的返回类型。

如果参数不完全相同，则会发生隐藏，如果只是返回值不同，则会发生冲突。

**形成多态**：有继承关系并且派生类中对基类的virtual函数现实了覆盖，用基类的引用或指针指向派生类对象，再用基类的指针或引用去调用virtual函数的时候会实现多态

### 2.虚函数的访问

### 2.1.通过对象名访问

和普通函数一样，虚函数一样可以通过对象名来调用，此时编译器采用的是静态联编。

通过对象名访问虚函数时, 调用哪个类的函数取决于定义对象使用的类。对象类型是基类时，就调用基类的函数；对象类型是派生类时，就调用派生类的函数。如：

obj\_base.disp(); //调用基类虚函数

obj\_child.disp(); //调用子类虚函数

在派生类中还可以使用作用域限定符来指定调用哪个基类的函数。如：

obj\_child.base::disp(); //调用基类虚函数

obj\_child.child::disp();//调用子类虚函数

### 2.2.通过指针访问

使用指针访问非虚函数时，要调用哪个函数，取决于指针本身类型，而不是根据指针指向的对象类型；

使用指针访问虚函数时，要调用哪个函数，取决于指针所指对象类型，而与指针本身的类型无关。

使用指针访问是虚函数调用的最主要形式。

### 2.3.引用访问

使用引用访问虚函数，与使用指针访问虚函数类似

不同的是，引用一经声明后，引用变量本身无论如何改变，其调用的函数就不会再改变，始终指向其开始定义时的函数。因此在使用上有一定限制，但这在一定程度上提高了代码的安全性，特别体现在函数参数传递等场合中，可以将引用理解成一种“受限制的指针”。

### 2.4.在成员函数中访问

在类内的成员函数中访问该类中的虚函数，如果想要采用动态联编，就要使用this指针，（当然也可以省略）。如果想采用静态联编则使用作用域限定符。

### 2.5.在构造函数或析构函数中访问

构造函数和析构函数是特殊的成员函数，在其中访问虚函数时，C++采用静态联编，即在构造函数或析构函数内，即使是使用“this->虚函数名”的形式来调用，编译器仍将其解释为静态联编的“本类名::虚函数名”。即它们所调用的虚函数是自己类中定义的函数，如果在自己的类中没有实现该函数，则调用的是基类中的虚函数。但绝不会调用任何在派生类中重定义的虚函数，因为对象构造过程中看不到派生类重定义的虚函数。

### 2.6.虚函数表vftable

含有虚函数的类的对象的第一个成员将是一个指向虚函数表(vftable)的指针(vfptr)。该虚函数表记录了该类的所有虚函数，在表中，派生类的虚函数会覆盖基类的相同虚函数。这样就实现了多态。

### 2.7.虚基表vbtable

虚基派生类的对象的第一个成员将是一个指向虚基表(vbtable)的指针（vbptr）。该虚基表记录了该类的所有虚基类，在表中，从不同路径继承而来的同一虚基类会相互覆盖

### 2.7.求包含虚函数的类的sizeof

包含有虚函数的类，在用sizeof求类对象所占用的内存空间的时候，因为此时对象有个指向虚函数表的指针，所以结果应该多4（64位为8字节）字节，但是当同一个类中有多个虚函数的时候，也只有4（64位为8字节）个字节。当另一个类虚基继承时，又会多4个（64位为8字节）字节。

## 3.纯虚函数与抽象类

当在基类中无法为虚函数提供任何有实际意义的定义时，可以将该虚函数声明为纯虚函数，它的实现留给该基类的派生类去做。

### 3.1.纯虚函数的声明和定义

纯虚函数是一种特殊的虚函数，其格式一般如下：

class 类名

{

virtual 类型 函数名 (参数表)=0;

…

};

纯虚函数不能被直接调用，仅提供一个与派生类一致的接口。

### 3.2.抽象类

一个类可以包含多个纯虚函数。只要类中含有一个纯虚函数，该类便为抽象类。一个抽象类只能作为基类来派生新类，不能创建抽象类的对象，可声明一个指向抽象类的指针。

和普通的虚函数不同，在派生类中一般要对基类中纯虚函数进行重定义，或者在派生类中再次将该虚函数声明为纯虚函数（也可什么也不做）。这说明，抽象类的派生类也可以是抽象类，只有在派生类中给出了基类中所有纯虚函数的实现时，该派生类便不再是抽象类。和纯虚函数一样，抽象类只起到提供统一接口的作用。

### 3.3.定义了protected型构造函数的类也是抽象类

对一个类来说，如果只定义了protected型的构造函数而没有提供public构造函数，无论是在外部还是在派生类中都不能直接创建该类的对象，但可以由其派生出新的类，这种能派生新类，却不能创建自己对象的类是另一种形式的抽象类。

### 3.4.用static成员函数为包含private构造函数的类创建对象

前面讲了构造函数可以为public型、也可以为protected型。其实，构造函数也可以是private型的。

此时，不能直接在外部函数和派生类中使用“类名+对象名”的形式来创建该类对象，但可以通过类的static成员函数来创建类的对象。

与此类似，可以使用static成员函数为包含protected型构造函数的类创建对象。

### 3.5.虚析构函数

构造函数不能时虚函数，因为虚函数的使用必须在对象创建后,通过vfptr指针来使用，而构造函数是用来创建对象的。

虽然构造函数不能被定义成虚函数，但析构函数可以定义为虚函数，一般来说，如果基类类中定义了虚函数，析构函数也应被定义为虚析构函数，尤其是类内有申请的动态内存，需要清理和释放的时候。原因在于如果析构函数不是虚函数，则通过指针访问的析构函数只能是基类的析构函数。

### 3.6.析构函数调用规则

如果有一个基类的指针指向派生类的对象，并且想通过该指针delete派生类对象,系统将只会执行基类的析构函数，而不会执行派生类的析构函数。为避免这种情况的发生，往往把基类的析构函数声明为虚的，此时，系统将先执行派生类对象的析构函数，然后再执行基类的析构函数。

如果基类的析构函数声明为虚的，派生类的析构函数也将自动成为虚析构函数，无论派生类析构函数声明中是否加virtual关键字。

## 4.含虚函数的多基派生的二义性

多个基类中有相同虚函数时，如果派生类中没有覆盖，也会引起二义性。

## 5.重载、覆盖、隐藏

override 覆盖：基类与派生类有相同的虚函数 。

oversee 隐藏：基类与派生类有同名数据成员或者同名成员函数（相同的虚函数除外）。只能以对象.基类名::成员的方式访问。

overload 重载：同一个类，同名函数，但参数不同（类型，顺序，个数）

## 6.虚继承与虚函数

C++中的virtual关键字所修饰的事物或现象在本质上是存在的，但是没有直观的形式表现，无法直接描述或定义，需要通过其他的间接方式或手段才能够体现出其实际上的效果。

虚函数的关键就在于存在、间接和共享这三种特征

（1）虚函数是存在的

（2）虚函数必须要通过一种间接的运行时（而不是编译时）机制（虚函数表和虚函数指针

）才能够激活（调用）的函数

（3）共享性表现在基类会共享被派生类重定义后的虚函数

虚拟继承如何表现：

（1）存在即表示虚继承体系和虚基类确实存在

（2）间接性表现在当访问虚基类的成员时同样也必须通过某种间接机制来完成（通过虚基表和虚基指针来完成）

（3）共享性表现在虚基类会在虚继承体系中被共享，而不会出现多份拷贝

### 6.1.虚继承

虚继承：在继承定义时包含了virtual关键字的继承关系

虚基类：在虚继承体系中的通过virtual继承而来的基类

语法：

class Subclass : public virtual Baseclass

{

public: //...

private: //...

protected: //...

};

其中Baseclass称之为Subclass的虚基类；而不是说Baseclass就是虚基类

虚继承而来的派生类对象的第一个成员是指向虚基表的虚基指针。(增加8个字节)

### 6.2.虚基派生的构造函数和析构函数

对普通的多层继承而言，构造函数的调用是嵌套的，如由C1类派生C2类，C2类又派生C3类时，C1—>C2—>C3，有：

C2(总参数表):C1(参数表)

C3(总参数表):C2(参数表)

而对虚基派生来说，如有A派生出B、C而B、C又派生出D，如果按照上述规则，应该有：

B(总参数表):A(参数表)

C(总参数表):A(参数表)

D(总参数表):B(参数表),C(参数表)

这样“A(参数表)”将被执行2次，这显然不行。

实际情况是：

B(总参数表):A(参数表)

C(总参数表):A(参数表)

D(总参数表):B(参数表),C(参数表),A(参数表)

根据虚基派生的性质，每个派生类中只有一份虚基类的拷贝，因此虚基类的构造函数在派生类中只能被调用一次。

对虚基类构造函数的调用总是先于普通基类的构造函数。

如果是在若干类层次中，从虚基类直接或间接派生出来的类的构造函数初始化列表均有对该虚基类构造函数的调用，那么创建一个派生类对象的时候只有该派生类列出的虚基类的构造函数被调用，其他未列出的将被忽略，这样就保证虚基类的唯一副本只被初始化一次。即虚基类的构造函数只被执行一次。

继承机制下构造函数的调用顺序：

先调用虚基类的构造函数

然后调用普通基类的构造函数

再调用成员对象的构造函数

最后调用派生类的构造。

继承机制下析构函数的调用顺序：

先调用派生类的析构函数

然后调用派生类中成员对象的析构函数

再调用普通基类的析构函数

最后调用虚基类的析构函数

### 6.3.效率

多重继承、虚拟继承相比单一继承更加复杂，造成了成员访问的效率低下，主要表现在两个方米：一是对象构建时多次设定vptr,二是对this指针的调整。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 继承方式 | vptr设定 | 成员访问 | 虚函数访问 | 效率 |
| 单一继承  no vptr | 无 | 指针/引用/对象访问效率相同 | 直接访问 | 效率较高 |
| 单一继承 | 一次 | 指针/引用/对象访问效率相同 | 通过vptr和vtable访问 | 多态的引入，带来了设定vptr和间接访问虚函数等效率降低的 |
| 多重继承 | 多次 | 指针/引用/对象访问效率相同 | 通过vptr和vtable访问，通过第二或者后继基类指针访问需要调整this指针 | 除了单一继承效率降低的原因，调整this指针也是一个原因 |
| 虚拟继承 | 多次 | 指针/引用/对象访问效率相同 | 通过vptr和vtable访问  访问虚基类需要调整this指针 | 除了单一继承效率降低的原因，调整this指针也是一个原因 |

## 7.RTTI、typeid、dynamic\_cast

### 7.1.RTTI

RTTI：是RunTime Type Identification的缩写，称“运行时类型识别”，一些老式的编译器可能不支持，不同编译器的实现方法也不尽相同。

RTTI只能应用于包含虚函数的类层次中，只有在虚函数处理上，使用派生类对象给基类指针赋值才有意义。如果类层次中没有虚函数，将派生类赋值给基类指针没有实质意义。RTTI的引入，可检查基类指针向派生类指针的转换是否安全，为类层次中非虚函数和数据成员的调用提供了方法。

### 7.2. typeinfo类和typeid操作符

在头文件typeinfo中还定义了type\_info类和typeid操作符，从typeid的字面即可看出，该操作符用以返回表达式的类型信息，其基本调用格式为：

std::type\_info&  typeid(expr);

参数expr可以是单个对象，变量，也可以是返回结果为对象的表达式，还可以是类型名；

返回值是一个type\_info对象的引用，如果expr是类对象（或类名）、且至少包含有一个虚函数，typeid操作符返回的type\_info对象需要在运行时计算；否则，返回一个静态对象，在编译时就可以计算得到。

typeinfo类中包含了一个name()成员，返回一个字符串，通常是类名，如下述语句返回的均为静态typeinfo对象:

cout<<typeid(5).name()<<endl; //输出字符串“int”

cout<<typeid(double).name()<<endl; //输出字符串“double”

来看一个返回动态typeinfo对象的例子：

class A{……}； //包含虚函数

class B：pulic A{……}

A\* pa=new B;

cout<<typeid(\*pa).name()<<endl; //输出结果为B

typeinfo类中对==和!=进行了重载，因此可以使用typeid来判断变量是哪种类型。如语句if (typeid(x) == typeid(double))用来判断变量x的类型是否double类型。

### 7.3.dynamic\_cast

dynamic\_cast只用于将指向派生类对象的基类（基类必须有虚函数）指针（或引用）安全的转换成派生类的指针（或引用） 以及相反的过程。dynamic\_cast 主要用于下行转换，在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的。

dynamic\_cast 会检查基类指针是否真正指向派生类对象，如果不是，则会失败。

对指针进行dynamic\_cast，失败返回null，成功返回正常cast后的对象指针；

对引用进行dynamic\_cast，失败抛出一个异常，成功返回正常cast后的对象引用。

# 模板

现在的C++编译器实现了一项新的特性：模板（Template），简单地说，模板是一种通用的描述机制，也就是说，使用模板允许使用通用类型来定义函数或类等，在使用时，通用类型可被具体的类型，如int、double甚至是用户自定义的类型来代替。模板引入一种全新的编程思维方式，称为“泛型编程”或“通用编程”。可以认为，类是类模板的特例，函数时函数模板的特例。

## 1.为什么定义模板

形象地说，把函数比喻为一个游戏过程，函数的流程就相当于游戏规则，在以往的函数定义中，总是指明参数是int型还是double型等等，这就像是为张三（好比int型）和李四（好比double型）比赛制定规则。可如果王五（char\*型）和赵六（bool型）要比赛，还得提供一套函数的定义，这相当于又制定了一次规则，显然这是很麻烦的。模板的的引入解决了这一问题，不管是谁和谁比赛，都把他们定义成A与B比赛，制定好了A与B比赛的规则（定义了关于A和B的函数）后，比赛时只要把A替换成张三，把B替换成李四就可以了，大大简化了程序代码量，维持了结构的清晰，大大提高了程序设计的效率。核心思想是**“类型参数化**”。

强类型程序设计中，参与运算的所有对象的类型在编译时即确定下来，并且编译程序将进行严格的类型检查。为了解决强类型的严格性和灵活性的冲突。有以下3中方式解决：

（1）带参数宏定义（原样替换）

（2）重载函数（函数名相同，函数参数不同）

（3）模板（将数据类型作为参数）

### 1.1.类型参数化

在讲解类型参数化之前，先来看一个示例：

int add(int x, int y) //定义两个int类型相加的函数

{

return x + y;

}

double add(double x, double y) //重载double类型

{

return x + y;

}

可以明显看出上面的的函数的计算规则是一样的，唯一不同的只是参数类型和返回值类型不一样而已。可以用一个通用的写法来代替：

T add(T x, T y)//重载字符数组

{

return x+y;

}

在使用函数时，在具体指明是哪一种类型。这就是类型参数化的思想。

### 1.2.模板

模板的引入使得函数定义摆脱了类型的束缚，代码更为高效灵活。C＋＋中，通过下述形式定义一个模板：

template <class T>

或

template<typename T>

早期模板定义使用的是class，关键字typename是最近才加入到标准中的，相比class，typename更容易体现“类型”的观点，虽然两个关键字在模板定义时是等价的，但从代码兼容的角度讲，使用class较好一些。

模板有函数模板和类模板之分。通过参数实例化构造出具体的函数或类，称为模板函数或模板类。

## 2.函数模板

### 2.1.函数模板的定义

函数模板的定义形式如下：

template <模板参数表>

返回值类型 函数名（参数列表）

{

//函数体

}

关键字template放在模板的定义与声明的最前面，其后是用逗号分隔的模板参数表，用尖括号（<>）括起来。模板参数表不能为空，模板参数有两种类型：

（1）class或typename修饰的类型参数，代表一种类型；

（2）非类型参数表达式,必须是int类型，使用已知类型符，代表一个常量

编译器根据函数模板的定义，检查传入的参数类型，生成相应的函数，并调用之。

并非所有的类型都必须参数化，可以具体指明某参数或者返回值的类型。

### 2.2.函数模板的使用

函数模板的使用规则和普通函数是相同的，如果模板定义在使用函数模板之后，必须对函数模板进行声明，此声明必须在外部进行，必须直接在名称空间中，也就是说不能在任何一个函数（包括main函数）中声明，声明的格式为：

template <typename T1[，typename T2，……]>

函数原型

和普通函数一样，如果在使用函数模板前对函数模板进行了定义，函数模板的声明可以省略。

注意：模板的声明和模板实现必须放在同一文件,或头文件，或源文件，不能像函数那样从在头文件声明，在其他文件实现。

### 2.3.实例化

函数模板实际上不是个完整的函数定义，因为其中的类型参数还不确定，只是定义了某些类型的角色（或变量）在函数中的操作形式，因此，必须将模板参数实例化才能使用函数，用模板实例化后的函数也称为模板函数.

分为隐式实例化和显式实例化

隐式实例化：使用时不指明参数的具体类型，由编译器自动推断。如add(2+3)。

显式实例化：使用时指明参数的具体类型。如add<int>(2+3)。

### 2.4.模板重载

函数模板支持重载，既可以模板之间重载（同名模板），也可以实现模板和普通同名函数间的重载，但模板的重载相比普通函数的重载要复杂一点。

（1）不能重载**本质相同**的同名模板。

template <class T1,class T2>

T1 Max(T1 a,T2 b){……}

与

template <class T3,class T4>

T3 Max(T3 c,T4 d){……}

看似不同的两个模板，仔细分析后发现，其本质是一样的，如果调用“Max(2,3.5);”，都实例化为“Max(int,double);”，会出现重复定义的错误。

（2）仅仅依靠返回值不同的模板重载也是不合法的，如：

template <class T1,class T2>

T1 Greater(T1 a,T2 b){……}

与

template <class T3,class T4>

T3\* Greater(T3 c,T4 d){……}

（3）允许根据**本质不同**的类型参数列表来重载同名模板

template <class T,int num>

T Greater(T a,T b){……}

与

template <class T1,class T2>

T1 Greater(T1 a,T2 b){……}

### 2.5.优先级与执行顺序

总体来说，一般函数优先于同名模板函数的执行

## 3.类模板

### 3.1.定义类模板

理解了函数模板的应用，类模板的提出似乎是水到渠成的事

模板的引入使得类的定义更为灵活，可以在类创建时指明其中的元素类型T，以及非类型常量num的大小，需要注意的是，不管是类定义还是成员函数定义，都要遵守模板的定义形式。

（1）不能重载本质相同的同名模板。

（2）允许根据不同的类型参数列表来重载同名模板

注意，类模板和成员函数模板不是传统意义上的类定义和成员函数定义，由类模板实例化而生成的具体类称为模板类。定义模板类对象的格式为：

类模板名<T>对象名; eg: Stack<int, 10> stack;

其中类模板的类型参数可以给默认值，但函数模板不能给默认值。如果出现非类型参数，则必须是int型，表示常量。即使模板的类型参数都有默认值，在使用模板时，也必须加上尖括号。

类模板的的成员函数在类模板之外实现时，类要用模板实例化。此时模板参数列表不能带有默认值。

也就是默认值只能在声明时出现，而不能在定义时出现。

## 4.模板嵌套

模板的套嵌可以理解成在另外一个模板里面定义一个模板。以模板（类，或者函数）作为另一个模板（类，或者函数）的成员，也称成员模板。

成员模版是不能声明为virtual的。

### 4.1.函数成员模板

可以将函数模板作为另一个类（必须是类模板）的成员，称为函数成员模板，其用法和普通成员函数类似。

函数成员模板可以采取模板类中声明，在模板类外实现。

template<类类型参数表>

template<函数类型参数表>

返回值类型 类名<类型参数>::函数名（参数）

{

……

}

### 4.2 对象成员模板

类模板的定义可以放在另一个类中，实例化后的模板类对象可以作为另一个类的成员。

对象成员模板可以在类模板中定义，也可以在类模板中声明，然后在类模板外实现。

template<外层类类型参数表>

template<内层类类型参数表>

class 外部类名<类型参数>::类名

{

……

}

## 5.模板做参数

模板包含类型参数（如class Type）和非类型参数（如int NUM，NUM是常量），实际上，模板的参数可以是另一个模板，也就是说，下述形式是合法的：

template<template <class T1> class T2, class T3,int Num>;

上述简单示例将原来简单的“class T2”或“Typename T2”扩充为“template <class T1> class T2”，来看一段示例代码12-11。

上述代码中定义了函数模板disp()，该模板的类型参数表中又包含了一个类模板TypeClass，在函数模板disp内可以对类TypeClass进行实例化处理。

## 6.模板的继承

模板类的继承包括四种：

（1）普通类继承类模板（要将类模板实例化为模板类）

template<class T>

class TBase

{

T data;

……

};

class Derived:public TBase<int>

{

……

};

（2）类模板继承普通类

class TBase

{

……

};

template<class T>

class TDerived:public TBase

{

T data;

……

};

（3）类模板继承类模板（实例化为模板类）

template<class T>

class TBase

{

T data1;

……

};

template<class T1,class T2>

class TDerived:public TBase<T1>

{

T2 data2;

……

};

（4）模板类继承类模板

class BaseA

{

public:

BaseA(){cout<<"BaseA founed"<<endl;}

};

class BaseB

{

public:

BaseB(){cout<<"BaseB founed"<<endl;}

};

template<typename T, int rows>

class BaseC

{

private:

T data;

public:

BaseC():data(rows){

cout<<"BaseC founed "<< data << endl;}

};

template<class T>

class Derived:public T

{

public:

Derived():T(){cout<<"Derived founed"<<endl;}

};

void main()

{

Derived<BaseA> x;// BaseA作为基类

Derived<BaseB> y;// BaseB作为基类

Derived<BaseC<int, 3> > z; // BaseC<int,3>作为基类

}

# 面向对象

## 1.面向对象的概念（了解）

面向对象：按人们认识客观世界的系统思维方式，采用基于对象（实体）的概念建立模型，模拟客观世界分析问题、设计解决方案、实现方案的过程。

通过面向对象的理念使计算机软件系统能与现实世界中的系统一一对应。包括：

面向对象分析(OOA) ：Object-Oriented Analysis

面向对象设计(OOD)：Object-Oriented Design

面向对象编程(OOP)：Object-Oriented Program

面向对象的特点：

抽象、封装、继承、多态

### 1.1.OOA的概念

OOA是在一个系统的开发过程中进行了系统业务调查以后，按照面向对象的思想来分析问题。

OOA所强调的是在系统调查资料的基础上，针对OO方法所需要的素材进行的归类分析和整理，而不是对管理业务现状和方法的分析。

OOA(面向对象的分析)模型由5个层次(主题层、对象类层、结构层、属性层和服务层)和5个活动(标识对象类、标识结构、定义主题、定义属性和定义服务)组成。

在这种方法中定义了两种对象类之间的结构，一种称为分类结构，一种称为组装结构。分类结构就是所谓的一般与特殊的关系。组装结构则反映了对象之间的整体与部分的关系。

OOA在定义属性的同时要识别实例连接。实例连接是一个实例与另一个实例的映射关系。

OOA在定义服务的同时要识别消息连接。当一个对象需要向另一对象发送消息时，它们之间就存在消息连接。

OOA 中的5个层次和5个活动继续贯穿在OOD(画向对象的设计)过程中.

#### 1.1.1.OOA的主要原则。

（1）抽象：从许多事物中舍弃个别的、非本质的特征，抽取共同的、本质性的特征，就叫作抽象。抽象是形成概念的必须手段。

　　抽象原则有两方面的意义：第一，尽管问题域中的事物是很复杂的，但是分析员并不需要了解和描述它们的一切，只需要分析研究其中与系统目标有关的事物及其本质性特征。第二，通过舍弃个体事物在细节上的差异，抽取其共同特征而得到一批事物的抽象概念。

　　抽象是面向对象方法中使用最为广泛的原则。抽象原则包括过程抽象和数据抽象两个方面。

　　过程抽象是指，任何一个完成确定功能的操作序列，其使用者都可以把它看作一个单一的实体，尽管实际上它可能是由一系列更低级的操作完成的。

　　数据抽象是根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察。数据抽象是OOA的核心原则。它强调把数据（属性）和操作（服务）结合为一个不可分的系统单位（即对象），对象的外部只需要知道它做什么，而不必知道它如何做。

（2）封装就是把对象的属性和服务结合为一个不可分的系统单位，并尽可能隐蔽对象的内部细节。

（3）继承：特殊类的对象拥有的其一般类的全部属性与服务，称作特殊类对一般类的继承。

在OOA中运用继承原则，就是在每个由一般类和特殊类形成的一般—特殊结构中，把一般类的对象实例和所有特殊类的对象实例都共同具有的属性和服务，一次性地在一般类中进行显式的定义。在特殊类中不再重复地定义一般类中已定义的东西，但是在语义上，特殊类却自动地、隐含地拥有它的一般类（以及所有更上层的一般类）中定义的全部属性和服务。继承原则的好处是：使系统模型比较简练也比较清晰。

（4）分类：就是把具有相同属性和服务的对象划分为一类，用类作为这些对象的抽象描述。分类原则实际上是抽象原则运用于对象描述时的一种表现形式。

（5）聚合：又称组装，其原则是：把一个复杂的事物看成若干比较简单的事物的组装体，从而简化对复杂事物的描述。

（6）关联：是人类思考问题时经常运用的思想方法：通过一个事物联想到另外的事物。能使人发生联想的原因是事物之间确实存在着某些联系。

（7）消息通信：这一原则要求对象之间只能通过消息进行通信，而不允许在对象之外直接地存取对象内部的属性。通过消息进行通信是由于封装原则而引起的。在OOA中要求用消息连接表示出对象之间的动态联系。

（8）粒度控制：一般来讲，人在面对一个复杂的问题域时，不可能在同一时刻既能纵观全局，又能洞察秋毫。因此需要控制自己的视野：考虑全局时，注意其大的组成部分，暂时不详察每一部分的具体的细节；考虑某部分的细节时则暂时撇开其余的部分。这就是粒度控制原则。

（9）行为分析：现实世界中事物的行为是复杂的。由大量的事物所构成的问题域中各种行为往往相互依赖、相互交织。

#### 1.1.2.面向对象分析产生三种分析模型

（1）对象模型:对用例模型进行分析,把系统分解成互相协作的分析类,通过类图/对象图描述对象/对象的属性/对象间的关系,是系统的静态模型

（2）动态模型:描述系统的动态行为,通过时序图/协作图描述对象的交互,以揭示对象间如何协作来完成每个具体的用例,单个对象的状态变化/动态行为可以通过状态图来表达

（3）功能模型(即用例模型à作为输入)。

#### 1.1.3.OOA的主要优点

（1）加强了对问题域和系统责任的理解；

（2）改进与分析有关的各类人员之间的交流；

（3）对需求的变化具有较强的适应性；

（4）支持软件复用。

（5）贯穿软件生命周期全过程的一致性。

（6）实用性；

（7）有利于用户参与。

#### 1.1.4.OOA方法的基本步骤

在用OOA具体地分析一个事物时，大致上遵循如下五个基本步骤：

第一步，确定对象和类。这里所说的对象是对数据及其处理方式的抽象，它反映了系统保存和处理现实世界中某些事物的信息的能力。类是多个对象的共同属性和方法集合的描述，它包括如何在一个类中建立一个新对象的描述。

第二步，确定结构（structure）。结构是指问题域的复杂性和连接关系。类成员结构反映了泛化-特化关系，整体-部分结构反映整体和局部之间的关系。

第三步，确定主题（subject）。主题是指事物的总体概貌和总体分析模型。

第四步，确定属性（attribute）。属性就是数据元素，可用来描述对象或分类结构的实例，可在图中给出，并在对象的存储中指定。

第五步，确定方法（method）。方法是在收到消息后必须进行的一些处理方法：方法要在图中定义，并在对象的存储中指定。对于每个对象和结构来说，那些用来增加、修改、删除和选择一个方法本身都是隐含的（虽然它们是要在对象的存储中定义的，但并不在图上给出），而有些则是显示的。

### 1.2.OOD的概念

面向对象设计(Object-Oriented Design，OOD)方法是OO方法中一个中间过渡环节。其主要作用是对OOA分析的结果作进一步的规范化整理，以便能够被OOP直接接受。

面向对象设计（OOD）是一种软件设计方法，是一种工程化规范。面向对象的编程范式：

决定你要的类；

给每个类提供完整的一组操作；

明确地使用继承来表现共同点。

由这个定义，我们可以看出：OOD就是“根据需求决定所需的类、类的操作以及类之间关联的过程”。

　OOD的目标是管理程序内部各部分的相互依赖。为了达到这个目标，OOD要求将程序分成块，每个块的规模应该小到可以管理的程度，然后分别将各个块隐藏在接口（interface）的后面，让它们只通过接口相互交流。比如说，如果用OOD的方法来设计一个服务器-客户端（client-server）应用，那么服务器和客户端之间不应该有直接的依赖，而是应该让服务器的接口和客户端的接口相互依赖。

这种依赖关系的转换使得系统的各部分具有了可复用性。还是拿上面那个例子来说，客户端就不必依赖于特定的服务器，所以就可以复用到其他的环境下。如果要复用某一个程序块，只要实现必须的接口就行了。

OOD是一种解决软件问题的设计范式（paradigm），一种抽象的范式。使用OOD这种设计范式，我们可以用对象（object）来表现问题领域（problem domain）的实体，每个对象都有相应的状态和行为。我们刚才说到：OOD是一种抽象的范式。抽象可以分成很多层次，从非常概括的到非常特殊的都有，而对象可能处于任何一个抽象层次上。另外，彼此不同但又互有关联的对象可以共同构成抽象：只要这些对象之间有相似性，就可以把它们当成同一类的对象来处理。

#### 1.2.1.OOD背景知识

计算机硬件技术却在飞速发展。从几十年前神秘的庞然大物，到现在随身携带的移动芯片；从每秒数千次运算到每秒上百亿次运算。当软件开发者们还在寻找能让软件开发生产力提高一个数量级的“银弹”[Brooks, 95]时，硬件开发的生产力早已提升了百倍千倍。

硬件工程师们能够如此高效，是因为他们都很懒惰。他们永远恪守“不要去重新发明轮子”的古训。Grady Booch把这些黑箱称为类属（class category），现在我们则通常把它们称为“组件（component）”。

类属是由被称为类（class）的实体组成的，类与类之间通过关联（relationship）结合在一起。一个类可以把大量的细节隐藏起来，只露出一个简单的接口，这正好符合人们喜欢抽象的心理。所以，这是一个非常伟大的概念，因为它给我们提供了封装和复用的基础，让我们可以从问题的角度来看问题，而不是从机器的角度来看问题。

软件的复用最初是从函数库和类库开始的，这两种复用形式实际上都是白箱复用。到90年代，开始有人开发并出售真正的黑箱软件模块：框架（framework）和控件（control）。框架和控件往往还受平台和语言的限制，现在软件技术的新潮流是用SOAP作为传输介质的Web Service，它可以使软件模块脱离平台和语言的束缚，实现更高程度的复用。但是想一想，其实Web Service也是面向对象，只不过是把类与类之间的关联用XML来描述而已。

在过去的十多年里，面向对象技术对软件行业起到了极大的推动作用。在可以预测的将来，它仍将是软件设计的主要技术——至少我看不到有什么技术可以取代它的。

#### 1.2.2.OOD起源

有很多人都认为：OOD是对结构化设计（Structured Design，SD）的扩展，其实这是不对的。OOD的软件设计观念和SD完全不同。SD注重的是数据结构和处理数据结构的过程。而在OOD中，过程和数据结构都被对象隐藏起来，两者几乎是互不相关的。不过，追根溯源，OOD和SD有着非常深的渊源。

1967年前后，OOD和SD 的概念几乎同时诞生，它们分别以不同的方式来表现数据结构和算法。当时，围绕着这两个概念，很多科学家写了大量的论文。其中，由Dijkstra和 Hoare两人所写的一些论文讲到了“恰当的程序控制结构”这个话题，声称goto语句是有害的，应该用顺序、循环、分支这三种控制结构来构成整个程序流程。这些概念发展构成了结构化程序设计方法；而由Ole-Johan Dahl所写的另一些论文则主要讨论编程语言中的单位划分，其中的一种程序单位就是类，它已经拥有了面向对象程序设计的主要特征。

这两种概念立刻就分道扬镳了。在结构化这边的历史大家都很熟悉：NATO会议采纳了Dijkstra的思想，整个软件产业都同意goto语句的确是有害的，结构化方法、瀑布模型从70年代开始大行其道。同时，无数的科学家和软件工程师也帮助结构化方法不断发展完善，其中有很多今天足以使我们振聋发聩的名字，例如Constantine、Yourdon、DeMarco和Dijkstra。有很长一段时间，整个世界都相信：结构化方法就是拯救软件工业的 “银弹”。当然，时间最后证明了一切。

而此时，面向对象则在研究和教育领域缓慢发展。结构化程序设计几乎可以应用于任何编程语言之上，而面向对象程序设计则需要语言的支持[1]，这也妨碍了面向对象技术的发展。实际上，在60年代后期，支持面向对象特性的语言只有Simula-67这一种。到70年代，施乐帕洛阿尔托研究中心（PARC）的 Alan Key等人又发明了另一种基于面向对象方法的语言，那就是大名鼎鼎的Smalltalk。但是，直到80年代中期，Smalltalk和另外几种面向对象语言仍然只停留在实验室里。

到90年代，OOD突然就风靡了整个软件行业，这绝对是软件开发史上的一次革命。不过，登高才能望远，新事物总是站在旧事物的基础之上的。70年代和80年代的设计方法揭示出许多有价值的概念，谁都不能也不敢忽视它们，OOD也一样。

#### 1.2.3.OOD和SD的区别？

还记得结构化设计方法吗？程序被划分成许多个模块，这些模块被组织成一个树型结构。这棵树的根就是主模块，叶子就是工具模块和最低级的功能模块。同时，这棵树也表示调用结构：每个模块都调用自己的直接下级模块，并被自己的直接上级模块调用。

那么，哪个模块负责收集应用程序最重要的那些策略？当然是最顶端的那些。在底下的那些模块只管实现最小的细节，最顶端的模块关心规模最大的问题。所以，在这个体系结构中越靠上，概念的抽象层次就越高，也越接近问题领域；体系结构中位置越低，概念就越接近细节，与问题领域的关系就越少，而与解决方案领域的关系就越多。

但是，由于上方的模块需要调用下方的模块，所以这些上方的模块就依赖于下方的细节。换句话说，与问题领域相关的抽象要依赖于与问题领域无关的细节！这也就是说，当实现细节发生变化时，抽象也会受到影响。而且，如果我们想复用某一个抽象的话，就必须把它依赖的细节都一起拖过去。

而在OOD中，我们希望倒转这种依赖关系：我们创建的抽象不依赖于任何细节，而细节则高度依赖于上面的抽象。这种依赖关系的倒转正是OOD和传统技术之间根本的差异，也正是OOD思想的精华所在。

#### 1.2.4.OOD步骤

　（1）细化重组类

（2）细化和实现类间关系,明确其可见性

（3）增加属性,指定属性的类型与可见性

（4）分配职责,定义执行每个职责的方法

（5）对消息驱动的系统,明确消息传递方式

（6）利用设计模式进行局部设计

（7）画出详细的类图与时序图

#### 1.2.5.OOD中要开展的主要工作

（1）对象定义规格的求精过程

对于OOA所抽象出来的对象-＆-类以及汇集的分析文档，OOD需要有一个根据设计要求整理和求精的过程，使之更能符合OOP的需要。这个整理和求精过程主要有两个方面：一是要根据面向对象的概念模型整理分析所确定的对象结构、属性、方法等内容，改正错误的内容，删去不必要和重复的内容等。二是进行分类整理，以便于下一步数据库设计和程序处理模块设计的需要。整理的方法主要是进行归类，对类一＆一对象、属性、方法和结构、主题进行归类。

（2）数据模型和数据库设计

数据模型的设计需要确定类-＆-对象属性的内容、消息连接的方式、系统访问、数据模型的方法等。最后每个对象实例的数据都必须落实到面向对象的库结构模型中。

（3）优化

OOD的优化设计过程是从另一个角度对分析结果和处理业务过程的整理归纳，优化包括对象和结构的优化、抽象、集成。

对象和结构的模块化表示OOD提供了一种范式，这种范式支持对类和结构的模块化。这种模块符合一般模块化所要求的所有特点，如信息隐蔽性好，内部聚合度强和模块之间耦合度弱等。

集成化使得单个构件有机地结合在一起，相互支持。

#### 1.2.6.OO方法的特点和面临的问题

OO方法以对象为基础，利用特定的软件工具直接完成从对象客体的描述到软件结构之间的转换。这是OO方法最主要的特点和成就。OO方法的应用解决了传统结构化开发方法中客观世界描述工具与软件结构的不一致性问题，缩短了开发周期，解决了从分析和设计到软件模块结构之间多次转换映射的繁杂过程，是一种很有发展前途的系统开发方法。

但是同原型方法一样,OO方法需要一定的软件基础支持才可以应用，另外在大型的MIS开发中如果不经自顶向下的整体划分，而是一开始就自底向上的采用OO 方法开发系统，同样也会造成系统结构不合理、各部分关系失调等问题。所以OO方法和结构化方法目前仍是两种在系统开发领域相互依存的、不可替代的方法。

#### 1.2.7.OOD能给我带来什么？

问这个问题的人，脑子里通常是在想“OOD能解决所有的设计问题吗？”没有银弹。OOD也不是解决一切设计问题、避免软件危机、捍卫世界和平……的银弹。OOD只是一种技术。但是，它是一种优秀的技术，它可以很好地解决目前的大多数软件设计问题——当然，这要求设计者有足够的能力。

OOD可能会让你头疼，因为要学会它、掌握它是很困难的；OOD甚至会让你失望，因为它也并不成熟、并不完美。OOD也会给你带来欣喜，它让你可以专注于设计，而不必操心那些细枝末节；OOD也会使你成为一个更好的设计师，它能提供给你很好的工具，让你能开发出更坚固、更可维护、更可复用的软件。

### 1.3.OOP的概念

面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）是一种计算机编程架构。OOP 的一条基本原则是计算机程序是由单个能够起到子程序作用的单元或对象组合而成。OOP 达到了软件工程的三个主要目标：重用性、灵活性和扩展性。为了实现整体运算，每个对象都能够接收信息、处理数据和向其它对象发送信息。OOP 主要有以下的概念和组件：

组件 － 数据和功能一起在运行着的计算机程序中形成的单元，组件在 OOP 计算机程序中是模块和结构化的基础。

抽象性 － 程序有能力忽略正在处理中信息的某些方面，即对信息主要方面关注的能力。

封装 － 也叫做信息封装：确保组件不会以不可预期的方式改变其它组件的内部状态；只有在那些提供了内部状态改变方法的组件中，才可以访问其内部状态。每类组件都提供了一个与其它组件联系的接口，并规定了其它组件进行调用的方法。

多态性 － 组件的引用和类集会涉及到其它许多不同类型的组件，而且引用组件所产生的结果得依据实际调用的类型。

继承性 － 允许在现存的组件基础上创建子类组件，这统一并增强了多态性和封装性。典型地来说就是用类来对组件进行分组，而且还可以定义新类为现存的类的扩展，这样就可以将类组织成树形或网状结构，这体现了动作的通用性。

#### 1.3.1.挑战

由于抽象性、封装性、重用性以及便于使用等方面的原因，以组件为基础的编程在脚本语言中已经变得特别流行。Python 和 Ruby 是最近才出现的语言，在开发时完全采用了 OOP 的思想，而流行的 Perl 脚本语言从版本5开始也慢慢地加入了新的面向对象的功能组件。用组件代替“现实”上的实体成为 JavaScript（ECMAScript） 得以流行的原因，有论证表明对组件进行适当的组合就可以在英特网上代替 HTML 和 XML 的文档对象模型（DOM）。

设计模式、技术和直觉构成严峻的挑战。这是选择编程语言之前必须认识到的，尽管不同语言的设计特性可能促进或者阻碍这一转化。

在网络应用的增长中，一个很重要的部分是小型移动设备和特殊Internet设备的爆炸性增长。这些设备各有各的操作系统，或者只在某种特定的设备领域内有共同的操作系统。我们现在还可以一一列举出这些设备——家庭接入设备、蜂窝电话、电子报纸、PDA、自动网络设备等等。但是这些设备领域的数量和深入程度将会很快变得难以估量。我们都知道这个市场大得惊人，PC的兴起与之相比不过小菜一碟。因此在这些设备的应用程序市场上，竞争将会相当残酷。获胜的重要手段之一，就是尽快进入市场。开发人员需要优秀的工具，迅速高效地撰写和调试他们的软件。平台无关性也是制胜秘诀之一，它使得程序员能够开发出支持多种设备平台的软件。

　我预期的另一个变化是，我们对于代码(Java)和数据(XML)协同型应用程序的开发能力将会不断提高。这种协同是开发强大应用程序的核心目标之一。我们从XML的迅速流行和ebXML规范的进展中，已经看到了这个趋势。ebXML是一个针对电子商务和国际贸易的，基于XML的开放式基础构架，由联合国贸易促进和电子商务中心(UN/CEFACT)与结构性信息标准推进组织(OASIS)共同开发。

#### 1.3.2.面向组件

我们能否期望出现一个真正的面向组件(component-oriented)的语言？它的创造者会是谁呢？

Stroustrup: 我怀疑，这个领域中之所以缺乏成果，正是因为人们——主要是那些非程序员们——对“组件”这个意义含糊的字眼寄予了太多的期望。这些人士梦想，有朝一日，组件会以某种方式把程序员赶出历史舞台。以后那些称职的“设计员”只需利用预先调整好的组件，把鼠标拖一拖放一放，就把系统组合出来。对于软件工具厂商来说，这种想法还有另一层意义，他们认为，到时候只有他们才保留有必要的技术，有能力

这种想法有一个最基本的谬误：这种组件很难获得广泛欢迎。一个单独的组件或框架(framework)，如果能够满足一个应用程序或者一个产业领域对所提出的大部分要求的话，对于其制造者来说就是划算的产品，而且技术上也不是很困难。可是该产业内的几个竞争者很快就会发现，如果所有人都采用这些组件，那么彼此之间的产品就会变得天下大同，没什么区别，他们将沦为简单的办事员，主要利润都将钻进那些组件/框架供应商的腰包里！

小“组件”很有用，不过产生不了预期的杠杆效应。中型的、更通用的组件非常有用，但是构造时需要非同寻常的弹性。

在C++中，我们综合运用不同共享形式的类体系(class hierarchies)，以及使用templates精心打造的接口，在这方面取得了一定的进展。我期待在这个领域取得一些有趣和有用的成果，不过我认为这种成果很可能是一种新的C++程序设计风格，而不是一种新的语言。

Lindholm: 编写面向组件的应用程序，好像更多的是个投资、设计和程序员管理方面的问题，而不是一个编程语言问题。当然某些语言在这方面具有先天优势，不过如果说有什么魔术般的新语言能够大大简化组件的编写难度，那纯粹是一种误导。

#### 1.3.3.C++未来

微软已经将全部赌注押在C#上，其他语言何去何从？

Stroustrup: C++在下一个十年里仍然将是一种主流语言。面对新的挑战，它会奋起应对。一个创造了那么多出色系统的语言，绝不会“坐视落花流水春去也”。

我希望微软认识到，它在C++(我指的是ISO标准C++)上有着巨大的利益，C++是它与IT世界内其他人之间的一座桥梁，是构造大型系统和嵌入式系统的有效工具，也是满足高性能需求的利器。其他语言，似乎更注重那些四平八稳的商用程序。

#### 1.3.4.竞争

C#会不会获得广泛的接受，并且挤掉其他的语言？

Lindholm: 通常，一种语言既不会从别的语言那里获利，也不会被挤掉。那些坚定的Fortran程序员不还用着Fortran吗？对于个人来说，语言的选择当然因时而异，但就整体而言，语言的种类只会递增，也就是说，它们之间的关系是“有你有我”而不是“有你没我”。

对于一个新语言的接受程度，往往取决于其能力所及。Java技术被迅速接受，原因是多方面的，Internet和World Wide Web接口，在其他技术面前的挫折感，对于Java技术发展方向的全面影响能力，都是原因。另一个重要的原因是Java独立于厂商，这意味着在兼容产品面前可以从容选择。

C#是否会获得广泛接受？视情况而定。总的来说，那些对于平台无关性和厂商无关性漠不关心的程序员，可能会喜欢C#。那些跟微软平台捆在一起人当然可能想要寻找VB 和VC的一个出色的替代品。但是对于程序跨平台执行能力特别关注的程序员，将会坚守Java之类的语言。这种能力对于多重访问设备（multiple access devices）和分布式计算模型至关重要，而Java语言提供了一个标准的、独立于厂商运行时环境。

Stroustrup:C#的流行程度几乎完全取决于微软投入的资金多少。看上去C#的兴起肯定会牺牲掉其他一些语言的利益，但是事实上未必如此。Java的蓬勃发展并没有给C++带来衰败。C++的应用仍然在稳定增长（当然，已经不是爆炸性的增长了）。也许其他的语言也还能获得自己的一席之地。

不过，我实在看不出有什么必要再发明一种新的专有语言。特别是微软，既生VB，何需C#？

#### 1.3.5.发展vs革新

C++是一种发展型的语言，Java和C#似乎更像是革新型语言（它们是从头设计的）？什么时候，革新型的语言才是必需的呢？

Lindholm: Java技术并非凭空出世，反而更像是发展型的。Java所有的特性，在Java平台推出之前，都至少已经存在于另一种环境之中。Java的贡献在于，在众多的特性和权衡中，做出了合理的选择，使得产品既实用，又优雅。Java技术对于程序员的态度是：抚养，但不溺爱。

Stroustrup：从技术上讲，我并不认为Java和C#是什么“从头设计的”革新型语言。倘若Java是从技术原则出发，从头设计，大概就不会模仿C/C++那种丑陋和病态的语法了（不必惊讶，Stroustrup在很多场合表示过，C++采用C的语法形式，实在是迫于兼容性。他本人更偏爱Simula的语法——译者）。

我认为，只有当程序员们面对的问题发生了根本的变化的时候，或者当我们发现了全新的、极其优越的程序设计技术，又完全不能为现存语言所支持的时候，我们才需要全新的语言。问题是，我们恐怕永远也碰不到那些“根本”、“全新”的情况。

我以为，自从OOP问世以来，可称为“根本”的新型程序设计技术，唯有泛型程序设计（generic programming）和生成式程序设计（generative programming）技术，这两项技术主要是源于C++ templates技术的运用，也有一部分曾经被视为面向对象和函数式语言(functional languages)的次要成分，现在都变成正式、可用和可承受的技术了。我对于目前C++模板(template)程序设计的成果非常兴奋。例如，像POOMA, Blitz++和MTL等程序库，在很多地方改变了数值计算的方式。

C#的一个“卖点”，就是它们的简单性。现在Java是不是快失去这个卖点了？

Stroustrup：新语言总是宣称自己如何如何简单，对老语言的复杂性颇多非议。其实这种所谓的“简单性”，简单地说，就是不成熟性。语言的复杂性，是在解决现实世界中极为烦琐和特殊的复杂问题的过程中逐渐增加的。一个语言只要活的时间够长，总会有某些地方逐渐复杂起来，或者是语言本身，或者是程序库和工具。C++和Java显然都不例外，我看C#也一样。如果一种语言能够度过自己的幼年时代，它会发现，自己无论是体积还是复杂性都大大增加了。

Lindholm：Java技术的的功能在增加，需要学习的东西也在增加。不过功能的增加并不一定带来复杂性的增加。Java技术的发展，并没有使学习曲线更加陡峭，只是让它继续向右方延展了。

#### 1.3.6.标准

标准化语言和开放型语言各自的优点和缺点何在？

Lindholm：对于一个开放、不允许专有扩展、具有权威的强制性标准语言或者运行环境来说，不存在什么缺点。允许专有扩展就意味着允许厂商下套子绑架客户。特别重要的是，必须让整个平台，而不只是其中一部分完全标准化，才能杜绝厂商们利用高层次的专有API下套子。客户要求有选择厂商的自由，他们既要有创造性，又需要兼容性。

Stroustrup：对于一个语言，如C/C++来说，建立正式标准（如ISO标准）最大的好处，在于可以防止某一个厂商操纵这种语言，把它当成自己的摇钱树。多个厂商的竞争给用户带来的是较低的价位和较好的稳定性。

专有语言的好处，一是流行，二是便宜（不过等你被套牢了之后，情况就会起变化），三是对于商业性需求可以做出快速的反应。

标准化语言的特点之一是，它不能忽略特殊用户的需求。比如我在AT&T中所考虑的东西，其规模、可靠性和效率要求，跟那些普通厂商关注的大众软件相比，根本不可同日而语。那些公司很自然只关注主要的需求。

　然而，多数大机构和身处前沿的公司，都有着特殊的需求。C++的设计是开放、灵活和高效的，能够满足我所能想象的任何需求。跟其他的现代语言相比，C++的家长式作风可谓少之又少，原因就在这。当然，不能赞赏这一点的人会诟病C++的“危险”。

拥有正式和开放标准的语言主要是为编程工具的使用者和客户服务的，而拥有专属“标准”的语言，主要是为厂商服务的。

## 2.面向程序设计

### 2.1.UML

软件工程－－标准与规范－－统一语言

Unifed Modeling Language(UML)， 又称统一建模语言或标准建模语言，是始于1997年一个OMG(Object Management Group)标准，它是一个模型化和软件系统开发的图形化语言。

UML为软件开发提供了一些标准的图例(10个)，统一开发思想，从而促进团队协作。

（1）类图(class diagram)

（2）对象图(object diagram)

（3）用例图(use case diagram)

（4）组件图(component diagram)

（5）部署图(deployment diagram)

（6）组合结构图(composite structure diagram)

（7）序列图(sequence diagram)

（8）协作图(collaboration diagram)

（9）状态图(state diagram)

（10）活动图(activity diagram)

### 2.2.软件开发过程

软件开发过程：分析，设计，编码，测试，维护

分析：是什么，包括需求与初始分析、分析阶段静态建模、分析阶段动态建模。

设计：怎么做，包括OO设计原则与模式、设计阶段静态建模、设计阶段动态建模、系统设计。

软件开发过程包括统一软件开发过程RUP（Rational Unified Process）与极限编程XP（eXtreme Programming）等。

RUP——重设计：以架构设计为中心，适用于大型软件团队开发大型项目

XP——重实践：收集用户素材，短周期交付；结对编程；测试驱动开发，编写测试用例；重构，以不改变软件功能作为前提，重构代码，从而改善既有代码的设计

### 2.3.类与类之间的关系

类与类之间的关系：继承(泛化)、关联、聚合、组合、依赖。

#### 2.3.1.继承

A是B的一种（a kind of ），

B的所有行为对A都适用，这是A继承自B。

A is B



#### 2.3.2.实现

A 时抽象类，B继承自A，并对A中的纯虚函数进行了覆盖。

#### 2.3.3.关联

双向或单向的关联关系

关系是长期的

彼此并不负责对方的生命周期

一般使用指针或者引用来实现

A has B



#### 2.3.4.聚合

比较强的一种关联关系

对象之间的关系表现为分为整体和局部

整体部分并不负责局部对象的销毁

一般使用指针或者引用来实现

A has B



#### 2.3.5.组合

更强的一种关联关系

对象之间的关系表现为分为整体和局部

整体部分负责局部对象的销毁

一般使用对象成员来实现

A has B



#### 2.3.6.依赖

从语义上来说是 A use　B，是偶然的，临时的，并非固定的

B作为A的成员函数参数

B作为A的成员函数的局部变量

A的成员函数调用B的静态方法



#### 2.3.7.各关系的比较

继承、实现体现的是类与类之间的纵向关系,其他四种体现的是类与类之间的横向关系

耦合强弱：依赖 < 关联 < 聚合 < 组合

从语义上来看：继承(A is B)，关联、聚合、组合(A has B)，依赖(A use B)

当组合与依赖结合时，可以替代继承：组合+依赖（基于对象） vs 继承（面向对象）

### 2.4.面向对象设计原则

一个优良的系统设计，强调模块间保持低耦合、高内聚的关系

SOLID的5原则：

（1）单一职责原则(Single Responsibility Principle)

（2）开闭原则(Open Closed Principle)

（3）里氏替换原则(Liscov Substitution Principle)

（4）接口分离原则(Interface Segregation Principle)

（5）依赖倒置原则(Dependency Inversion Principle)

#### 2.4.1.单一职责原则

核心思想：一个类，最好只做一件事，只有一个引起它变化的原因。

如果一个类承担的职责过多，就等于把这些职责耦合在一起了。一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当发生变化时，设计会遭受到意想不到的破坏。而如果想要避免这种现象的发生，就要尽可能的遵守单一职责原则。此原则的核心就是解耦和增强内聚性。

#### 2.4.2.开放闭合原则

从OOD角度看，软件实体(类,模块,函数等等)应当对扩展开放，对修改闭合。通俗来讲，它意味着应当能在不修改类的前提下扩展一个类的行为。

核心思想就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，所以修改就是封闭的；而通过面向对象的继承和多态机制，又可以实现对抽象类的继承，通过覆写其方法来改变固有行为，实现新的拓展方法，所以就是开放的。

#### 2.4.3.里氏替换原则

核心思想是：派生类必须能够替换其基类。

Liskov替换原则，主要着眼于抽象和多态建立在继承的基础上，因此只有遵循了Liskov替换原则，才能保证继承复用是可靠的。实现的方法是面向接口编程：将公共部分抽象为基类接口或抽象类，通过提取抽象类，在子类中通过覆写父类的方法实现新的方式支持同样的职责。

#### 2.4.4.接口分离原则

核心思想：使用多个小的专门的接口，而不要使用一个大的总接口

接口隔离原则体现在：接口应该是内聚的，应该避免“胖”接口。一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。

#### 2.4.5.依赖倒置原则

核心思想：面向接口编程，具体依赖抽象。

具体而言：

a.高层模块不依赖于底层模块，二者都同依赖于抽象；

b.抽象不依赖于具体，具体依赖于抽象。

当两个模块之间存在紧密的耦合关系时，最好的方法就是分离接口和实现：在依赖之间定义一个抽象的接口使得高层模块调用接口，而底层模块实现接口的定义，以此来有效控制耦合关系，达到依赖于抽象的设计目标。

## 3.值语义与对象语义

|  |  |  |
| --- | --- | --- |
|  | **值语义** | **对象语义** |
| **定义** | 一个对象被系统标准的复制方式复制后，与原对象毫无关系，可以彼此独立改变互不影响。 | 通常是指一个对象被系统标准的复制方式复制后，与原对象依然共享底层资源，对任何一个的改变都将改变另一个 |
| **标准复制方式** | 赋值、拷贝构造 | 赋值、拷贝构造 |
| **基本数据类型** | 包括指针都是值语义 |  |
| **自定义类型** | 不包含资源的自定义类型，系统提供的缺省拷贝构造函数与赋值操作符保证了值语义；  包含资源的自定义类型，需要提供深拷贝操作的拷贝构造函数和赋值操作符，并在构造函数中获取资源，在析构函数中释放资源 | 包含资源的自定义类型，没有提供拷贝构造函数和赋值操作符，或者在拷贝构造函数和赋值操作符中有意共享资源，则此时的对象具有指针语义 |
| **优点** | 避免别名问题导致的意外修改  避免对共享资源的引用引起的释放时机和额外的引用计数问题  生命周期容易控制 | 占用较少内存 |
| **缺点** | 占用较多内存 | 别名问题导致的意外修改，尤其用于模块间接口的参数和返回值  资源释放的额外负担，通常是引用计数  生命周期不容易控制，智能指针实际上是将对象语义转化为值语义 |

# 基于对象OB与面向对象OO

## 1.std::function

std::function<R(T1, T2, ..., TN)>

这是一个模板实现的函数对象类，它可以包装其它任意的函数对象，而被包装的函数对象具有类型为T1,T2,…,TN的参数，其返回值为R类型。是bind和mem\_fn的返回值类型。

function 对象的最大用处在于实现函数回调，等效于C中的函数指针。

## 2.bind

bind(可调用对象，指定值… C++11)

bind接收的第一个参数必须是一个可调用的对象f，有无参数信息都可以，包括函数、函数指针、函数对象、和成员函数指针和数据成员指针，之后bind最多接受9个参数，参数数量必须与f的参数数量相等，这些参数被传递给f作为入参。 绑定完成后，bind会返回一个函数对象，它内部保存了f的拷贝，具有operator()，返回值类型被自动推导为可调用对象的返回类型。在发生调用时这个函数对象将把之前存储的参数转发给f完成调用。

必须在绑定表达式中提供函数要求的所有参数，无论是真实参数还是占位符均可以。对于不事先绑定的参数，需要传std::placeholders：：\_nummber进去，从\_1开始，依次递增\_2,\_3……。表示占位符所在位置的形参，将接收调用时传入的第number个实参。调用函数时，可以传入更多的参数但是没有对应占位符的参数都是无效参数。占位符可以出现也可以不出现，出现的顺序和数量没有限定，但不能使用超过函数参数数量的占位符， bind完全可以代替标准库中的bind1st和bind2nd，使用bind(f,N,\_1)和bind(f,\_1,N)。要注意的是它们均使用了一个占位符

注意:

（1）成员函数的第一参数是隐藏的this指针，如绑定的是成员函数，要取地址，变为成员函数指针。因为成员函数指针不能直接调用operator(),它必须被绑定到一个对象或指针，然后才能得到this指针进而调用成员函数。因此bind需要 “牺牲”一个占位符，要求提供一个类的实例、引用或者指针，通过对象作为第一个参数来调用成员函数。

（2）如果绑定对象的形参是引用或常引用，而绑定实参是左值则要用ref或cref转换，否则会发生值传递。失去原本传引用的目的的。

（3）bind的返回值可以用auto关键字自动推定。

（4）不能将占位符绑定到引用

（5）bind采用拷贝的方式保存绑定对象和参数，这意味着绑定表达式中的每一个变量都会有一份拷贝，如果函数对象或值参数很大、拷贝代价很高，或者无法拷贝，那么bind的使用就会受到限制。因此bind库可以搭配ref库使用，ref库包装了对象的引用，可以让bind存储对象引用的拷贝，从而降低了拷贝的代价。但这也带来了一个隐患，因为有时候bind的调用可能会延后很久，程序员必须保证bind被调用时引用是有效的。如果调用是引用的变量或者函数对象你被销毁了，那么将会发生未定义行为。

ref和cref

std::reference\_wrapper<T> ref(T& t);

template< class T >std::reference\_wrapper<T> ref( std::reference\_wrapper<T> t );

template <class T>void ref(const T&&) = delete;

template< class T >std::reference\_wrapper<const T> cref( const T& t );

template< class T >std::reference\_wrapper<const T> cref( std::reference\_wrapper<T> t );

template <class T>void cref(const T&&) = delete;

## 3.OB vs OO

面向对象的三大特点（封装，继承，多态）缺一不可。

通常“基于对象”是使用对象，但是并不利用现有的对象模板产生新的对象类型，继而产生新的对象，即“基于对象”没有继承的特点。

“面向对象”和“基于对象”都实现了“封装”的概念，但是面向对象实现了“继承和多态”，而“基于对象”没有实现这些。

继承(面向对象) vs 组合(基于对象)

std::function +std::bind 可以替换虚函数来表现多态功能，使用起来更来灵活。

# new/delete扩展

## 1.new/delete工作机制

### 1.1.new与delete的工作步骤

使用new表达式时发生的三个步骤：

第一步：调用名为operator new的标准库函数，分配足够大的原始的未类型化的内存，以保存指定类型的一个对象

第二步：运行该类型的一个构造函数去初始化对象。

第三步：返回指向新分配并构造的构造函数对象的指针。

使用delete表达式时，发生的步骤：

第一步：调用对象的析构函数。

第二部：调用名为operator delete的标准库函数释放该对象所用的内存。

### 1.2.operator new/delete库函数

operator new 和operator delete函数有两个重载版本，不指定则使用默认提供的库函数。

void \* operator new (size\_t);

void \* operator new[](size\_t);

void operator delete(void \*);

void operator delete[](void \*);

## 2.只能生成栈对象

只能生成栈对象，就是说不能生成堆对象，亦即不能通过new表达式[在类之外]生成对象

不能生成堆对象能想到的方法：

要生成栈对象就将构造函数/析构函数放入public区域，不能在类外生成堆对象，类外无法访问operator new/delete函数，自然要将operator new与 operator delete函数中的一个或者两个放在private区域。

## 3.只能生成堆对象

只能生成堆对象，就是说不能生成栈对象，亦即在创建栈对象时，不能[在类之外]调用构造函数或者析构函数

不能生成栈对象能想到的方法：

在类外调用new和delete，则operator new/delete函数应在public区域，又因为要完成对象的构造，构造函数应该public区，不能生成栈对象，则析构函数应类外无法访问。

## 4.不能用一个对象初始化另一个新建对象

将拷贝构造函数放在private区。

## 5.不能让一个已存在对象向另一个已存在对象赋值

将赋值构造函数妨碍private区。

# 标准模板库STL概论

STL，即Standard Template Library，不是面向对象的编程，而是一种新的编程模式：泛型编程（Generic Programming）。STL是C++标准库的组成部分，STL是很庞大复杂的系统，单单就STL就可写出厚达千页的技术书籍，所以，本章不可能做到面面俱到，重点在于介绍泛型编程的思想和本质，介绍一些常用的方法，为初学者学习STL提供一些感性认识，起到抛砖引玉的作用。

## 1.理解STL

STL库是用模板（template）写出来的，模板是STL库的基础。STL大致由以下几部分组成：

容器（container）、迭代器（iterator）、适配器（Adapter）、算法（algorithm）、函数对象（functor）、配置器（allocator）

容器、迭代器、容器适配器都是用类模板实现的，迭代器用于遍历容器中的每一个元素，算法用于操作数据。

### 1.1容器

如果没有STL的支持，在处理一些复杂问题时，要自行设计存储模式，如数组管理，插入删除操作等，这不但很繁琐，而且bug频出，是程序出问题最多的地方。STL运用模板类库机制，为数据存储，查找和其他操作提供了一整套方案，大大提高了程序的正确性。不仅如此，类库对常用的很多操作进行了优化处理，大大提高了程序的效率。

容器是可容纳一些数据的类模板，STL中有vector、list、deque、set/multiset、map/multimap等容器。

### 1.2.适配器

适配器就是Interface(接口)，对容器、迭代器和算法进行包装，但其实质还是容器、迭代器和算法，只是不依赖于具体的标准容器、迭代器和算法类型。

容器适配器可以理解为容器的模板，迭代器适配器可理解为迭代器的模板，算法适配器可理解为算法的模板。

容器适配器：

|  |  |  |
| --- | --- | --- |
| [stack](http://classfoo.com/ccby/article/s4SvF) | [<stack>](http://classfoo.com/ccby/article/fmoWdS) | LIFO 栈 |
| [queue](http://classfoo.com/ccby/article/DMS8D) | [<queue>](http://classfoo.com/ccby/article/WdLxdR) | FIFO 队列 |
| [priority\_queue](http://classfoo.com/ccby/article/Q3jxC) | [<queue>](http://classfoo.com/ccby/article/WdLxdR) | 优先级队列 |

### 1.3.迭代器

在有的专业书籍中，迭代器也称游标，可以将迭代器初步理解为广义指针，迭代器和指针功能很像，迭代器是通过重载一元的”\*”和”->”来从容器中间接地返回一个值。**指向数组内部的指针可以直接当做迭代器**。

迭代器有5种，依次为：

随机访问迭代器（Random Access Iterator）

双向迭代器（Bidirectional Iterator）

前向迭代器（Forward Iterator）

输入迭代器（Input Iterator）

输出迭代器（Output Iterator）

### 1.4.算法

STL包含了很多对容器进行处理的函数，它们的处理思路大体相同：使用**迭代器**来标识要处理的数据或数据段、以及结果的存放位置，有的函数还作为对象参数传递给另一个函数，实现数据的处理。

## 2.容器分类（C++11）

容器是STL的基础，容器分3种：

**（1）序列式容器**（sequential container）

|  |  |  |
| --- | --- | --- |
| [array](http://classfoo.com/ccby/article/bnrZv) **C++11** | [<array>](http://classfoo.com/ccby/article/gNnddI) | 数组类 |
| [vector](http://classfoo.com/ccby/article/jnevK) | [<vector>](http://classfoo.com/ccby/article/KCu6dM) | 向量 |
| [deque](http://classfoo.com/ccby/article/ryIpw) | [<deque>](http://classfoo.com/ccby/article/sJhydJ) | 双端队列 |
| [list](http://classfoo.com/ccby/article/Hdj4y) | [<list>](http://classfoo.com/ccby/article/El1KdL) | 双向链表 |
| [forward\_list](http://classfoo.com/ccby/article/5wWLx) **C++11** | [<forward\_list>](http://classfoo.com/ccby/article/zLZQdK) | 正向链表 |

序列式容器会强调元素的次序，依次维护第一个元素、第二个元素……，直到最后一个元素，面向序列式容器的操作主要是迭代操作。

**（2）关联式容器**（associative container）

|  |  |  |
| --- | --- | --- |
| [set](http://classfoo.com/ccby/article/w2pUE) | [<set>](http://classfoo.com/ccby/article/XQA8dO) | 集合 |
| [multiset](http://classfoo.com/ccby/article/kSPmB) | [<set>](http://classfoo.com/ccby/article/XQA8dO) | 多键集合 |
| [map](http://classfoo.com/ccby/article/WQ9qz) | [<map>](http://classfoo.com/ccby/article/aOZxdN) | 映射表 |
| [multimap](http://classfoo.com/ccby/article/a8VKA) | [<map>](http://classfoo.com/ccby/article/aOZxdN) | 多键映射表 |

**（3）无序关联容器**（unordered associative container）

|  |  |  |
| --- | --- | --- |
| [unordered\_set](http://classfoo.com/ccby/article/qNNOJ) **C++11** | [<unordered\_set>](http://classfoo.com/ccby/article/G9Z6dQ) | 无序集合 |
| [unordered\_multiset](http://classfoo.com/ccby/article/C18YI) **C++11** | [<unordered\_set>](http://classfoo.com/ccby/article/G9Z6dQ) | 无序多键集全 |
| [unordered\_map](http://classfoo.com/ccby/article/S3XoG) **C++11** | [<unordered\_map>](http://classfoo.com/ccby/article/pKVIdP) | 无序映射表 |
| [unordered\_multimap](http://classfoo.com/ccby/article/hedAH) **C++11** | [<unordered\_map>](http://classfoo.com/ccby/article/pKVIdP) | 无序多键映射表 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1-array 2-vector 3-deque 4-list 5-forward-list 6-set、multiset、multimap 7-map**  **8-** unordered **set、**unordered **multiset、**unordered **multimap 9-unordered map** | | | | | | | | | | | |
|  | **方法** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **功能** |
| 迭  代  器 | begin |  |  |  |  |  |  |  |  |  | 返回指向容器第一个元素的迭代器 |
| end |  |  |  |  |  |  |  |  |  | 返回指向容器最后一个元素的下一个的迭代器 |
| rbegin |  |  |  |  |  |  |  |  |  | 返回指向容器逆序起始位置的逆序迭代 |
| rend |  |  |  |  |  |  |  |  |  | 返回指向容器逆序最后一个元素的下一个的逆序迭代器 |
| cbegin |  |  |  |  |  |  |  |  |  | 返回指向容器第一个元素的常迭代器 |
| cend |  |  |  |  |  |  |  |  |  | 返回指向容器最后一个元素的常下一个的迭代器 |
| crbegin |  |  |  |  |  |  |  |  |  | 返回指向容器逆序起始位置的常逆序迭代器 |
| crend |  |  |  |  |  |  |  |  |  | 返回指向容器逆序最后一个元素的下一个的常逆序迭代器 |
| cbefore\_begin |  |  |  |  |  |  |  |  |  | 返回指向容器起始位置前的常迭代器 |
| before\_begin |  |  |  |  |  |  |  |  |  | 返回指向容器起始位置前的迭代器 |
| 容  量 | size |  |  |  |  |  |  |  |  |  | 返回有效元素个数 |
| max\_size |  |  |  |  |  |  |  |  |  | 返回系统规定容器支持的最大元素个数 |
| resize |  |  |  |  |  |  |  |  |  | 改变有效元素的个数 |
| capacity |  |  |  |  |  |  |  |  |  | 返回当前已分配最大元素内存块数（即存储容量） |
| empty |  |  |  |  |  |  |  |  |  | 判断是否为空 |
| reserve |  |  |  |  |  |  |  |  |  | 请求改变存储容量 |
| shrink\_to\_fit |  |  |  |  |  |  |  |  |  | 请求移除未使用的存储空间 |
| 元  素  访  问 | operator[] |  |  |  |  |  |  |  |  |  | 访问元素 |
| at |  |  |  |  |  |  |  |  |  | 访问元素 |
| front |  |  |  |  |  |  |  |  |  | 访问第一个元素 |
| back |  |  |  |  |  |  |  |  |  | 访问最后一个元素 |
| data |  |  |  |  |  |  |  |  |  | 返回当前容器内部数组的指针 |
| 修  改 | operator= |  |  |  |  |  |  |  |  |  | 值赋操作 |
| assign |  |  |  |  |  |  |  |  |  | 将新的内容赋值给容器 |
| push\_back |  |  |  |  |  |  |  |  |  | 在末尾增加一个元素 |
| pop\_back |  |  |  |  |  |  |  |  |  | 删除最后一个元素 |
| push\_front |  |  |  |  |  |  |  |  |  | 在开头增加一个元素 |
| pop\_front |  |  |  |  |  |  |  |  |  | 删除第一个元素 |
| insert |  |  |  |  |  |  |  |  |  | 插入元素 |
| erase |  |  |  |  |  |  |  |  |  | 删除元素 |
| swap |  |  |  |  |  |  |  |  |  | 交换内容 |
| clear |  |  |  |  |  |  |  |  |  | 清空内容 |
| emplace |  |  |  |  |  |  |  |  |  | 构造及插入一个元素 |
| emplace\_back |  |  |  |  |  |  |  |  |  | 在容器末尾构造及插入一个元素 |
| emplace\_front |  |  |  |  |  |  |  |  |  | 在容器开头构造及插入一个元素 |
| fill |  |  |  |  |  |  |  |  |  | 用新的值填充容器 |
| emplace\_hint |  |  |  |  |  |  |  |  |  | 按提示的位置构造及插入一个元素 |
| insert\_after |  |  |  |  |  |  |  |  |  | 插入元素 |
| erase\_after |  |  |  |  |  |  |  |  |  | 删除元素 |
| emplace\_after |  |  |  |  |  |  |  |  |  | 构造及插入一个元素 |
| 操  作 | splice |  |  |  |  |  |  |  |  |  | 使元素从一个列表移动到另一个列表 |
| remove |  |  |  |  |  |  |  |  |  | 删除值为指定值的元素 |
| remove\_if |  |  |  |  |  |  |  |  |  | 删除满足指定条件的元素 |
| unique |  |  |  |  |  |  |  |  |  | 删除重复值 |
| merge |  |  |  |  |  |  |  |  |  | 合并已排序的列表 |
| sort |  |  |  |  |  |  |  |  |  | 为容器中的所有元素排序 |
| reverse |  |  |  |  |  |  |  |  |  | 反转元素的顺序 |
| splice\_after |  |  |  |  |  |  |  |  |  | 使元素从一个正向列表移动到另一个正向列表 |
| find |  |  |  |  |  |  |  |  |  | 通过给定值查找元素 |
| count |  |  |  |  |  |  |  |  |  | 返回匹配给定值的元素的个数 |
| lower\_bound |  |  |  |  |  |  |  |  |  | 返回指向第一个值大于等于给定值的元素的迭代器 |
| upper\_bound |  |  |  |  |  |  |  |  |  | 返回指向第一个值大于给定值的元素的迭代器 |
| equal\_range |  |  |  |  |  |  |  |  |  | 返回元素值匹配给定值的元素组成的范围 |
| 观  察 | key\_comp |  |  |  |  |  |  |  |  |  | 返回主键比较对象 |
| value\_comp |  |  |  |  |  |  |  |  |  | 返回值比较对象 |
| hash\_function |  |  |  |  |  |  |  |  |  | 返回 hash 函数 |
| key\_eq |  |  |  |  |  |  |  |  |  | 返回主键等价性判断谓词 |
| 槽 | bucket\_count |  |  |  |  |  |  |  |  |  | 返回槽（Bucket）数 |
| max\_bucket\_count |  |  |  |  |  |  |  |  |  | 返回最大槽数 |
| bucket\_size |  |  |  |  |  |  |  |  |  | 返回槽大小 |
| bucket |  |  |  |  |  |  |  |  |  | 返回元素所在槽的序号 |
| 哈  希  策  略 | load\_factor |  |  |  |  |  |  |  |  |  | 返回载入因子，即一个元素槽（Bucket）的最大元素数 |
| max\_load\_factor |  |  |  |  |  |  |  |  |  | 返回或设置最大载入因子 |
| rehash |  |  |  |  |  |  |  |  |  | 设置槽数 |
| reserve |  |  |  |  |  |  |  |  |  | 请求改变容器容量 |
|  | std::swap |  |  |  |  |  |  |  |  |  | 交换两个容器的内容 |
|  | 关系运算符 |  |  |  |  |  |  |  |  |  |  |
|  | get\_allocator |  |  |  |  |  |  |  |  |  | 获得内存分配器 |

## 3.序列式容器

### 3.1.序列式容器的创建和元素的访问

使用序列式容器，须包含相关的头文件，vector、list及deque分别对应：

<vector>、<list>、<deque>、<array>、<forward\_list>

创建序列式容器的对象，大体有6种方式：

（1）创建空的容器，此时容器中的元素个数为0。

容器类型<数据类型>容器名；

（2）产生特定大小的容器

容器类型<数据类型>容器名（初始大小）；

产生特定大小的容器，此时容器中的元素被创建，编译器使用默认值为元素隐式初始化，像int、float和double等内建的数据类型会被初始化为0，对于类对象元素，将调用其无参构造函数（用户定义的或编译器缺省提供的）或每个参数都有默认值的构造函数。象obV中含10个double型元素, 初始化为0

（3）创建特定大小的容器，并且为其中的每个元素指定初始值，此时在元素多少的参数后增加一个参数。

容器类型<数据类型>容器名（初始大小，填充值）；

（4）根据已有同类型的容器创建新容器，并将其中的元素完全复制或者移动过来。

容器类型<数据类型>容器名（已创建容器）；

（5）通过一对迭代器（可暂时理解为指针），以使编译器决定元素的个数和初值，这对迭代器用以标识一组元素区间。

容器类型<数据类型>容器名（起始迭代器，结束迭代器）；

vector和deque类的容器创建后就可以通过容器名[下标]或容器名.at(序号)的形式对元素进行随机访问(这是因为这2种类模板对下标运算符[]进行了重载)；也支持迭代器访问。

但list类的容器不支持下标运算符[]，无法使用[]对元素进行随机访问。但支持双向迭代器访问，如：

list<int>::iterator iter = obL.begin();

（6）通过初始化列表。

容器类型<数据类型>容器名{初始化列表}；

容器类型<数据类型>容器名={初始化列表}；

### 3.2.所有容器都支持的方法

|  |  |  |  |
| --- | --- | --- | --- |
| **方法** | **返回值** | **功能** | **复杂度** |
| 容器名.begin() | 迭代器 | 返回第一个元素 | O(1) |
| 容器名.end() | 迭代器 | 返回最后一个元素的下一个元素 | O(1) |
| swap(容器1，容器2) | void | 交换两个容器的内容 | O(n) |
| 容器1.swap(容器2) | void | 与另一个同类型容器交换内容 | O(1) |
| 关系运算符 | bool | 判断两个同类型容器是否满足关系 | O(n) |
| 赋值运算符 | bool | 用一个容器向另一个容器赋值 |  |

### 3.3.序列式容器中元素的插入和删除

在创建普通数组时，需要指定元素的个数，元素的插入和删除很繁琐。但在序列式容器中，只要调用操作函数，所有的事情都由STL类库自动完成，而且容器对象都能随着元素的插入和删除自动地增大或缩小。

下面介绍5类相关函数：

（1）在容器尾部进行插入和删除(list、deque和vector都适用)：

void push\_back(t)和void pop\_back(void)

（2）在容器头部进行插入和删除(list和deque适用，vector不适用):

void push\_front(t)和void pop\_front(void)

（3）获取容器头部和尾部元素(list、deque和vector都适用)：

front(void)和back(void)，见代码13-4

（4）在容器中间插入元素。有如下3种重载形式：

将元素t插到p之前，返回的迭代器指向被插入的元素。

iterator insert(iterator p, elemType t);

在p之前插入n个t，无返回值。

void insert(iterator p, int n, elemType t);

在p之前插入[first, last)之间的所有元素。

void insert(iterator p, iterator first, iterator last);

（5）erase删除操作，有2种重载形式：

删除迭代器p所指向的元素，返回p指向的下一个迭代器：

iterator erase(iterator p);

删除[frist,last)之间的所有元素，返回last指向的下一个迭代器：

iterator erase(iterator first, iterator last);

（6）clear操作：用于将容器对象清空。

void clear(void);

对于std::list还有一种删除方法: 成员函数 remove/remove\_if

### 3.4.vector容器

介绍完vector、list以及deque的通用用法，下面分别讨论下其特别之处。首先是vector，字面翻译为向量，其用法类似于数组，但其功能比数组更强大。简单地说，vector是数组的类表示，它提供了自动管理内存的功能，可以动态改变vector对象的长度，并随着元素的增删而增大或缩小(?)，提供了对元素的随机访问，和数组一样，在vector尾部添加和删除元素（push\_back和pop\_back）的时间是固定的，但在vector中间或头部增删元素(insert,erase)的时间和复杂度线性正比于vector容器对象中元素的多少。

### 3.5.deque容器

deque表示双端队列（double-ended queue），deque容器对象支持下标随机访问，在deque头部和尾部添加删除元素时的时间都是固定的，因此，如果有很多操作是针对序列的头部位置的，建议使用deque容器。但是，如果是在deque的中间进行元素的增删处理，操作的复杂度和时间正比于deque对象中元素的多少。

### 3.6.list容器

list类模板表示双向链表，除了首尾元素外，list容器对象中的每个元素都和前面的元素相链接。list不支持下标随机访问，只能通过迭代器双向遍历。

和vector和deque不同的是，在list的任何位置增删元素的时间都是固定的。

说明：除了上述介绍的基本操作外，序列式容器还有其它用于特定场合的成员函数操作，限于篇幅原因，本章只对STL作入门式介绍，更详细的内容可查阅相关资料。

## 4 关联式容器

关联式容器（associative）又称“联合容器”，将值（value）和关键字（keyword）成对关联，举例来说，在学生管理系统设计时，可以将学号作为关键字，起索引的目的，而将学生姓名、性别、籍贯等信息作为值与学号配对。

标准的STL提供了4种联合容器类模板：set、map、multiset和multimap，总体来说，set中仅仅包含关键字，而没有值的概念，map中存储的是“关键字――值”对，map和set中不会出现多个相同的关键字，multiset和multimap可以分别看作是对set和map的扩展，multimap和multiset允许相同关键字的存在。

4种关联式容器都会根据指定的或默认的排列函数，以关键字为索引，对其中的元素进行排序。

### 4.1.set容器

使用set容器必须#include <set>。

其使用形式如下：

set<存储类型[,排序函数或函数对象]> 容器对象名;

第1个参数用以指定存储类型，第2个参数是可选的，用来指定对关键字进行排序的函数或函数对象(函数对象后面介绍)，在默认情况下，将使用less<>类模板，字面意义上可理解为按从小到大进行排列。可以自己设定排序方式：set<int, greater<int> >

根据set的特点，STL提供了4种创建set的方式：

创建空set容器对象，如：

set<存储类型[,排序函数或函数对象]> 容器对象名;

将迭代器的区间作为参数的构造函数，如：

set<存储类型[,排序函数或函数对象]> 容器对象名(起始迭代器，结束迭代器);

根据已有同类型的容器创建新容器，如

set<存储类型[,排序函数或函数对象]> 容器对象名（已经创立的容器）

通过初始化列表。

set<存储类型[,排序函数或函数对象]> 容器对象名{初始化列表，排序函数};

set<存储类型[,排序函数或函数对象]> 容器对象名={初始化列表，排序函数};

set不支持[]下标式的随机访问，必须通过迭代器访问元素。

元素添加到set中后不能修改

### 4.2.multiset容器

使用multiset需要包含头文件<set>。

multiset的创建方式与set相同，也是4种方式。

multiset与set不同之处在于其允许出现相同的关键字。

### 4.3.map容器

使用map必须包括头文件<map>。

map的元素是一对对的“关键字-值”组合，“关键字”用于搜寻，而“值”用来表示我们要存取的数据。

在map容器中，每个关键字只能出现一次，不能重复。

可使用类模板pair<class T1,class T2>来表示map容器中形如“关键字-值”的每个元素。如下述语句就生成了map容器对象的一个元素t：

pair<const int, string> t(600036,”招商银行”);

cout << t.first << t.second << endl; //t.first表示600036，t.second表示,”招商银行”

也可创建pair<class T1,class T2>的匿名元素：

pair<const int, string>(600036,”招商银行”);

const表示关键字是只读的。

创建map容器对象的格式如下，各参数含义和set的类似：

map<关键字类型,值类型[,排序函数或对象]> 容器对象名;

创建方式也是4种：

创建空map容器对象，如：

map<存储类型[,排序函数或对象]> 容器对象名;

将迭代器的区间作为参数的构造函数，如：

map<存储类型[,排序函数或对象]> 容器对象名(起始迭代器，结束迭代器);

根据已有同类型的容器创建新容器，如

map<存储类型[,排序函数或对象]> 容器对象名（已经创立的容器）

通过初始化列表。

map<存储类型[,排序函数或对象]> 容器对象名{初始化列表，排序函数};

map<存储类型[,排序函数或对象]> 容器对象名={初始化列表，排序函数};

### 4.4.multimap容器

multimap与map的关系，类似于multiset与set的关系，使用multimap同样需要包含头文件<map>。

multimap的创建方式与map相同，有4种方式。

multimap与map不同之处在于其允许出现相同的元素。此时不支持[]随机操作。

不同的是将map换成了multimap，以帮助读者理解两者的不同。

### 4.5.关联式容器支持方法

#### 4.5.1.元素的插入

在下面的4种insert函数中，ob代表关联式容器对象名，t对set和multiset来说，是个关键字，而对map和multimap来说，是个pair结构。

（1）pair<iterator, bool> ob.insert(t)

适用于map和set容器。返回一个pair<iterator,bool>值，当ob中不包含t时，将t插入ob中，bool为true；否则不进行插入，bool为false。无论是否进行了插入操作，iterator都指向关键字和t相同的元素。

（2） iterator ob.insert(t)

适用于multiset和multimap，将t插入到ob中，并返回指向t所在位置的迭代器。

（3）iterator ob.insert(p,t)

从迭代器p开始搜索应将t插入到ob中的位置。 适用于4个关联式容器。

对于set/map，只有当ob中不包含t时才进行插入。无论是否插入，都返回一个指向t关键字的iterator。

对于multiset/multimap，将t插入到ob中，返回指向t的iterator；p的作用是加快查找速度,并不是指定位置。

（4）void insert(i, j)

i,j是一对输入迭代器，此函数用于将[i,j)区间的元素插入到ob中。

（5）void insert（{initlist}）;

#### 4.5.2.元素的删除

关联式容器支持以下4种删除元素的方式：

（1）UINT ob.erase(keyword)

删除容器ob中所有关键字为keyword的元素，返回删除元素的个数。示例见代码13-13

（2）void ob.erase(p)

从ob中删除迭代器p所指向的元素，p必须存在，且不能为ob.end()，因为ob.end()是最后一个元素的下一个位置，并不真正存在。

（3）void ob.erase(q1,q2)

从ob中删除半开区间[q1,q2)之间的所有元素。

（4）void clear(void)

删除ob中的所有元素，等价于void ob.erase(ob.begin(), ob.end());

#### 4.5.3.为容器定义的类型

STL为所有容器定义了下列类型：

容器::value\_type，如 vector<int>::valut\_type表示int类型

(注:对于普通的就是其类型，对于map，则是pair<int, string>类型)

容器::reference，如list<int>::reference表示int &

容器::const\_reference，如set<int>::const\_reference表示const int &

容器::iterator，如list<int>::iterator表示指向int的迭代器

容器::const\_iterator，const迭代器

容器::different\_type，表示2个迭代器间的距离，类似指针差值。

容器::size\_type,无符号整数，表示容器对象的元素个数。

STL专门为关联式容器定义的类型有：

容器::key\_type,表示关键字的类型,如map<int,string>::key\_type=int

容器::mapped\_type, map/multimap中值的类型,上例中的string

容器::key\_compare, 关键字比较函数对象类, 默认为 less<Tkey>

容器::value\_compare, 对set/multiset, 同key\_compare; 对map/multimap, 为value\_type提供了排序功能

#### 4.5.4.元素的查找与访问

|  |  |
| --- | --- |
| 方法 | 功能 |
| find(k) | 返回第一个关键字为k的元素的迭代器，如果没有返回end() |
| count(k) | 返回关键字与k相同的元素的数目 |
| lower\_bound(k) | 返回第一个关键字不小于k的元素的迭代器 |
| upper\_bound(k) | 返回第一个关键字大于k的元素的迭代器 |
| equal\_range(k) | 返回一个pair结构，第一成员为lower\_bound(k) 第二个成员为upper\_bound(k) |
| operator[] | 仅适用于map结构，返回一个引用，该引用指向第一个关键字k关联的值 |

## 5.迭代器

### 5.1.迭代器本质

模板的引入使得函数和类定义脱离了存储类型的限制，在需要时指定即可，是一种泛化的思维观念．迭代器是种更高层次的抽象，它使得算法独立于容器,这使得算法独立于类型。

迭代器是种类型，在程序中使用的是其对象。从迭代器的层面上看，对所有类型容器元素的访问应该是等价的，因此，迭代器对象应具备以下功能：

（1）间接访问（\*p）,即deference，在迭代器类中必须对一元\*操作符定义。

（2）迭代器对象之间的赋值，如p＝q，在迭代器类中必须对赋值操作符定义。

（3）迭代器对象间的比较，比较两个迭代器是否相等，因此，在迭代器类内必须对关系运算符==和!=进行定义，原则上讲，不需要对迭代器进行大小比较（<、>）等，就像比较指针实际上是比较其中存储的地址大小一样，没有意义。

（4）能使用迭代器遍历容器中所有的元素，在本章已给出的示例代码中已经大量应用了诸如“p++”之类的操作，因此，在迭代器类必须对前缀增1和后缀增1进行定义。

### 5.2迭代器类型

不同的算法对迭代器的要求不同，为此，STL定义了5种迭代器，分别是

随机访问迭代器（RandomAccessIterator）

双向迭代器（BidirectionalIterator）

前向迭代器（ForwardIterator）

输出迭代器（OutputIterator）

输入迭代器（InputIterator）



不同的算法要求的迭代器类型不同，之所以定义了5种迭代器，是为了使用“最合适”的工具，编写算法时在满足要求的基础上尽可能地使用功能少的迭代器，减少迭代器引入的副作用，假设要编写一个查找函数find()，只要能读取容器中的元素即可，最理想的方案是使用输入迭代器，这样，有效防止了在find()函数内对元素的修改，真正“物尽其用”，正如指针一章中曾经介绍的，一把刀既能削铁如泥，又能砍瓜切菜，还能理发，真正用其理发是很危险的，不如剃头刀来的安全方便。

对5种迭代器初步了解后，重新看一下图13.1，实际上，依照箭头的方向，迭代器实现的功能越来越少，除了输出迭代器和输入迭代器是功能相反的并列关系外，箭头左侧的迭代器不仅都实现了右侧迭代器所有的功能，而且在其基础上增加了一些功能。所以说，箭头左侧的迭代器“适应”于箭头右侧的迭代器，因此，如果某个算法的形参为前向迭代器，则实参可以是双向迭代器和随机访问迭代器，但不能是输出迭代器或输入迭代器。

### 5.3.容器迭代器的具体类型

7种容器的迭代器中，array，vector和deque是随机访问迭代器，foward是前向迭代器，list和4种关联容器种均为双向迭代器。

7种容器迭代器p均支持如下操作：

\*p读，\*p写，++p，p++，--p，p--

只有vector和deque迭代器p支持如下操作：

p += n和p -= n

### 5.4.流迭代器

流迭代器是特殊的迭代器，包括两种：

输出流迭代器(ostream\_iterator)

输入流迭代器(istream\_iterator)

理解的要点是将输入/输出流作为容器看待。因此，任何接受迭代器参数的算法都可以和流一起工作。使用流迭代器必须要包含头文件<iterator>。

输出流迭代器的定义为：

ostream\_iterator<class T1, class T2=char> 迭代器名(ostream\_type &ost, const T2 \*p=0);

如：

ostream\_iterator<int> osi(cout, "\n");

现在osi就是一个输出流迭代器，通过osi可使用cout输出int类型的数据，结束符为换行符。

输入流迭代器的定义为：

istream\_iterator<class T1,class T2＝char> 迭代器名(istream\_type& ist);

如：

istream\_iterator<int,char> isi(cin);

现在isi就是一个输入流迭代器，通过isi可使用cin接收int数据类型的键盘输入，结束符为非数字字符。

## 6.泛型算法

本节讨论的算法不是指容器的成员函数，而是STL类库中提供的一些通用的非成员函数操作。算法是STL的苦力，是处理容器里面数据的方法，操作。这些算法可以操作在多种容器类型上,所以称为“泛型”，泛型算法不是针对容器编写，而只是单独依赖迭代器和迭代器操作实现。

使用泛型算法必须先包含algorithm头文件，使用泛化的算术算法则必须包含numeric头文件：

#include <algorithm>

#include <numeric>

### 6.1什么是函数对象

C语言中介绍了函数指针这一概念，使得函数可以作为参数传递给另一个函数；在第9章中介绍了函数调用操作符的重载，这使得在程序中可以使用诸如“对象名(参数表)”的形式，实际上，上述两个概念都属于函数对象。

函数对象是可以以函数方式与()结合使用的任意对象，包括：

函数名；

指向函数的指针；

重载了()操作符的类的对象。（仿函数）

### 6.2.算法分类

STL将算法库分为4组，前3个在algorithm头文件中描述，而第4个在numeric头文件中描述：

非修改式序列操作：不改变容器的内容，如find()、for\_each()等。

修改式序列操作：可以修改容器中的内容，如transform()、random\_shuffle()、copy等。

排序和相关操作：包括各种排序函数等，如sort()等。

通用数字运算：计算两个容器的内部乘积等。

STL中算法函数太多，无法一一介绍，下面仅以简表形式分别列出这4组常用的算法。

#### 6.2.1.非修改式序列操作

for\_each():将一个非修改式函数用于指定区间中的每个成员

find():在区间中查找某个值首次出现的位置

find\_if():在区间中查找第一个满足断言测试条件的值

find\_end():在序列中查找最后一个与另一个序列匹配的值。匹配时可使用等式或二元断言

find\_first\_of():在第二个序列中查找第一个与第一个序列的值匹配的元素。匹配时可使用等式或二元断言。

adjacent\_find():查找第一个与其后面的元素匹配的元素。匹配时可使用等式或二元断言。

count():返回特定值在区间中出现的次数。

count\_if():

mismatch():

equal():

search():

search\_n():

#### 6.2.2.修改式序列操作

copy():

copy\_backword():

swap():

swap\_ranges():

iter\_swap():

transform():

replace():

replace\_if()

remove/remove\_if

……

注意：remove并不会立即删除元素，只是移动了末尾，要用erase才会删除。这和list::remove不一样，这是防止迭代器失效。

#### 6.2.3.排序

sort():

stable\_sort():

partial\_sort():

nth\_element():

lower\_bound():

upper\_bound():

equal\_range():

#### 6.2.4.数字操作

accumulate():计算区间中的值的总和

inner\_product():计算两个区间的内部乘积

partial\_sum():将使用一个区间计算得到的小计复制到另一个区间中

adjacent\_different():将使用一个区间的元素计算得到的相邻差集复制到另一个区间中

## 7.适配器

前面简要提到了适配器的概念，适配器相当于提供了一个接口，使得某些不适用于特定对象的方法可以被该对象所用，适配器形象的功能图解如所示，图中，容器或函数对象无法直接应用于算法，因此，必须有一种中间过渡机制来实现两者的匹配，这就是适配器，本质上，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。



### 7.1.容器适配器

容器适配器可以看作是**对容器的封装**，使其具有某些特殊的功能。容器适配器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实现，只是发生了接口转换而已。标准库提供了三种序列容器适配器：

stack(vector,deque,list)

queue(deque,list)

priority\_queue(vector, deque)。

所有适配器都定义了两个构造函数：默认构造函数用于创建空对象，而带一个容器参数的构造函数将参数容器的副本作为其基础值。

默认的stack和queue都基于deque容器实现，而priority\_queue则在vector容器上实现。在创建适配器时，通过将一个顺序容器指定为适配器的第二个类型参数，可覆盖其关联的基础容器类型。例如，下述代码创建的ob栈是基于vector实现的。

stack<int, vector<int> > ob;

### 7.2.迭代器适配器

从某种意义上讲，前面介绍的流迭代器也是迭代器适配器，此外，还包括反向迭代器和插入迭代器。和使用流迭代器一样，使用反向和插入迭代器都必须包含头文件<iterator>。

（1）反向迭代器

详见MSDN和代码13-18

（2）插入迭代器适配器

详见MSDN和代码 13-18-2

### 7.3.函数适配器

函数适配器共有如下3种：

#### 7.3.1.绑定器

bind1st(函数对象,指定参数值);（c++11弃用）

将指定值绑定到二元函数对象(带参数信息)的第1个参数上，返回一元函数对象。

bind2nd(函数对象,指定值); （c++11弃用）

将指定值绑定到二元函数对象(带参数信息)的第2个参数上。返回一元函数对象。

ptr\_fun( \*ptr) （c++11弃用）

将普通函数或函数指针转为带参数信息的函数对象

注意：bind1st和bind2nd绑定的是带有参数信息**函数对象**,普通函数或函数指针必须通过**ptr\_fun**函数转变为函数对象才能绑定，而自定义的仿函数则需要以public**继承binary\_function**基类或者在**类中显式定义同样的public参数信息**。其模板为：

template <class Arg, class Arg2, class Res>

struct **binary\_function**

{

typedef Arg first\_argument\_type;

typedef Arg2 second\_argument\_type;

typedef Res result\_type;

};

template <class Arg, class Result>

struct **unary\_function**

{

typedef Arg argument\_type;

typedef Result result\_type;

};

bind(可调用对象，指定值… C++11)

bind接收的第一个参数必须是一个可调用的对象f，有无参数信息都可以，包括函数、函数指针、函数对象、仿函数、和成员函数指针和数据成员指针，之后bind最多接受9个参数，参数数量必须与f的参数数量相等，这些参数被传递给f作为入参。 绑定完成后，bind会返回一个函数对象，它内部保存了f的拷贝，具有operator()，返回值类型被自动推导为可调用对象的返回类型。在发生调用时这个函数对象将把之前存储的参数转发给f完成调用。

对于不事先绑定的参数，需要传std::placeholders：：\_nummber进去，从\_1开始，依次递增\_2,\_3……。表示占位符所在位置的形参，将接收调用时传入的第number个实参。调用函数时，可以传入更多的参数但是没有对应占位符的参数都是无效参数。

注意:

（1）成员函数的第一参数是隐藏的this指针，必须传入一个对象。如绑定的是成员函数，一般要取地址，变为成员函数指针。

（2）如果待绑定的对象的形参是引用或常引用而绑定实参不是引用，或者实参不能进行值传递，此时要用ref或cref转换引用或，否则会发生值传递。失去原本传引用的目的的。

（3）bind的返回值可以用auto关键字自动推定。

（4）不能将占位符绑定到引用

ref和cref

std::reference\_wrapper<T> ref(T& t);

template< class T >std::reference\_wrapper<T> ref( std::reference\_wrapper<T> t );

template <class T>void ref(const T&&) = delete;

template< class T >std::reference\_wrapper<const T> cref( const T& t );

template< class T >std::reference\_wrapper<const T> cref( std::reference\_wrapper<T> t );

template <class T>void cref(const T&&) = delete;

### 7.3.2.否定器

#### 7.3.2.1.谓词

接受一个参数的函数，叫做一元函数，如果一元函数返回布尔值，则该函数成为一元谓词；接受两个参数的函数为二元函数，如果返回一个布尔值，则该函数称为二元谓词。

#### **7.3.2.2.**断言

断言就是用于在程序中捕捉某些假设条件的代码句。断言表示为对一些表达式（0代表false，非0代表true）或者谓词进行判断，程序员相信在程序中的某个特定点该表达式或谓词的值为真。

C中有宏定义assert，如果表达式的结果为零，就退出执行。

C++11有static\_assert这个关键字，它不是宏。但是他的用途是在编译期检查一些常量或者传入函数的参数是不是符合预期。

static\_assert ( bool\_constexpr , message )

not1(一元谓词) //用于逆转一元谓词，如not1(bind2nd(lt,3))

not2(二元谓词) //用于逆转二元谓词，如not2(a>b)

说明：not1是构造一个与谓词结果相反的一元函数对象，not2是构造一个与谓词结果相反的二元函数对象。这里的谓词应当是函数对象，而不是普通函数。

### 7.3.3.成员函数适配器

mem\_fun()

mem\_fun\_ref()

将成员函数转为函数对象，当容器中存放的是对象实体时用mem\_fun\_ref，当是指针时用,mem\_fun.

mem\_fn()

两者都能用(C++11)，当多个对象都要执行同一个成员函数时。

注意：绑定成员函数要取地址，

## 8.C++内存管理

C++提供下面两种方法分配和释放未构造的原始内存：

（1）allocator类，它提供可感知类型的内存分配，这个类提供allocate和deallocate方法分配和释放内存，并随后使用该内存保存对象。

（2）标准库中的 普通operator new和operator delete ，用以分配和释放所需大小的原始未类型化内存。

C++提供不同的方法在原始内存中构造和撤销对象。

（1）allocator类提供了construct和destory方法，construct用来在分配好的未构造的原始内存中初始化对象，destory用以在对象上运行其析构函数。

（2）定位new表达式。作用是接收指向分配好的未构造原始内存的指针，并在该空间中初始化一个对象或者数组。

（3）未初始化内存算法uninitialized\_fill，uninitialized\_fill\_n和uninitiallized\_copy, uninitiallized\_copy\_n就像std::fill,std::fill\_n和std::copy,std::copy\_n一样执行，但他们的目的不是给对象赋值而是构造对象。

### 8.1.allocator

allocator定义在<memory>中，memory该包括三个文件：

（1）<stl\_construct.h>：定义了全局函数construct（）和destory（）负责对象的构造和析构。

（2）<stl\_alloc.h>：定义了一、二级配置器，彼此合作。名为alloc。

（3）<stl\_uninitialized.h>:定义了一些全局函数用来填充或者复制大块内存。如：uninitialized\_copy()等。

allocator提供的方法：

allocator<T>对象名

定义一个allocator对象，可以分配内存或者构造T类型对象

allocate(n)

分配原始未构造内存用以保存n个T类型对象。但不构造对象。

deallocate(p,n)

释放内存，在T\*指针p中包含的地址处保存n个T类型对象。运行deallocate之前在该内存中构造的任意对象的销毁是用户的责任。但不负责销毁对象。

construct(p,t)

在T\*型指针p所指原始内存中构造一个新元素，运行T类型的拷贝构造函数用t初始化该对象。但不负责分配原始内存。

destroy（p）

运行T\*指针p所指对象的析构函数。但不负责回收原始内存。

### 8.2.placement new表达式

头文件<stl\_construct.h>中使用定位new构造对象

定位new表达式在**已分配**的未构造原始内存中初始化一个对象

它与new的其它版本的不同之处在于:不分配内存，它只在特定的,预分配的内存地址构造一个对象，它的形式如下:

new (place\_address) type

new (place\_address) type (initializers)

new (place\_address) type [size]

new (place\_address) type [size] {braced initializer list}

注意：place\_address必须是一个指针，initializer list 是初始化列表（可能为空）用来在构造新分配的对象时使用。

### 8.3.未初始化内存操作

uninialized\_copy(b,e,b2)

从迭代器b和e的范围将元素复制到从迭代器b2开始的未构造后的原始内存中。该函数在目标内存构造元素而不是给它们复制。假设b2指定的目标内存足以保存输入范围中元素的的副本。

uninialized\_fill(b,e,t)

将迭代器b和e的范围的对象初始为t的副本。假定该范围是未构造的原始内存。使用拷贝构造函数构造对象。

uninialized\_fill\_n(b,e,t,n)

将迭代器b和e的范围中至多n个对象初始为t的副本。假定该范围是未构造的原始内存。使用拷贝构造函数构造对象。

### 8.4.allocator底层实现

真正具备空间分配功能的std::alloc。

对象构造前的空间配置和构造后的空间释放都是由<stl\_alloc.h>中的alloc分配器负责，SGI的设计思路是：

（1）向system heap 要求空间

（2）考虑多绪(multi-threads)状态

（3）考虑内存不足的应变措施

（4）考虑过多小型区块可能造成的内存破碎(fragment)问题

alloc分配策略：

考虑到小型区块可能造成的内存破碎问题，SGI设计双层配置器，究竟只开放一级配置器还是都开放取决于\_USE\_MALLOC是否被定义。

第一级配置器直接使用malloc()和free（），

template<int inst>class \_\_malloc\_alloc\_tempalte

{

（1）allocate直接使用malloc(),deallocate直接使用free()

（2）模拟c++的set\_new\_handler以处理内存不足

}

第二级配置器则视情况采用不同的策略：

当请求区块超过128B时，视为足够大，调用第一级的配置器，反之，视为过小，为了降低额外负担，则采用复杂的memory pool整理方式，而不再求助于第一级配置器。

template<bool threads,int inst>class \_\_default\_alloc\_template

{

（1）维护16个自由链表，共128B，每个链表8个字节，负责16种小型区块的配置能力。

内存池也用由一级配置器的malloc配置而得。如果内存不足，转调用第一级配置器，（那里有

应变策略）

（2）如果需求区块大于128字节，转调用第一级配置器。

}

详细架构如下：

template<int inst>

class \_\_malloc\_alloc\_tempalte//一级配置器

{

public:

static void\* allocate(size\_t \_n);

static void deallocate(void\* \_p,size\_t /\*\_n\*/);

static void\* reallocate(void \*\_p,size\_t /\*\_old\_sz\*/,,size\_t \_new\_sz);

private:

static void\* \_S\_oom\_malloc(size\_t \_n);

static void\* \_S\_oom\_realloc(void \*,size\_t \_n);

};

template<bool threads,int inst>

class \_\_default\_alloc\_tempalte//二级配置器

{

public:

static void\* allocate(size\_t \_n);

static void deallocate(void\* \_p,size\_t \_n);

static void\* reallocate(void \*\_p,size\_t \_old\_sz,,size\_t \_new\_sz);

private:

static size\_t \_S\_round\_up(size\_t \_bytes);

static size\_t \_S\_freelist\_index(size\_t \_bytes);

static void\* \_S\_refill(size\_t \_n);

static char\* \_S\_chunk\_alloc(size\_t \_size,int &\_bobjs);

static char\* \_S\_start\_free;

static char\* \_S\_end\_free;

static size\_t \_S\_heap\_size;

union \_Obj

{

union \_Obj\* \_M\_free\_list\_link;

char \_M\_client\_data[1];

};

static \_Obj\* \_STL\_VOLATILE\_S\_free\_list[\_NFREELISTS];

};







源码解释参考http://qqsunkist.iteye.com/blog/1504895

# C++11标准头文件

**C++ 标准库**提供了大量的头文件用以支持程序特性并简化相关开发。最新标准库（C++11）提供了约 80 个头文件，本专题将其分为以下几类。

**注意！**一些头文件（比如：[<cstdlib>](http://classfoo.com/ccby/article/XTBfcd)）可能会涉及多个子库。如想了解一头文件中具体哪些内容与某个库相关，可以查询该子库的详细说明，也可以从这里开始。

## 1.语言支持库

|  |  |
| --- | --- |
| [<cstddef>](http://classfoo.com/ccby/article/eCLbb8) | 定义（typedef）了一些类型，比如：[std::size\_t](http://classfoo.com/ccby/article/ewOzb5)、[NULL](http://classfoo.com/ccby/article/A2sZb2)及其它 |
| [<limits>](http://classfoo.com/ccby/article/3YOeb9) | 提供了查询基础类型属性的标准方法 |
| [<climits>](http://classfoo.com/ccby/article/FvqFca) | 整数类型的限制 |
| [<cfloat>](http://classfoo.com/ccby/article/b7Klcb) | 浮点数类型的限制 |
| [<cstdint>](http://classfoo.com/ccby/article/Fn54cc) | 一些字节大小固定（Fixed-size）的类型及其它类型的限制 |
| [<cstdlib>](http://classfoo.com/ccby/article/XTBfcd) | 已在“通用工具库”中定义 |
| [<new>](http://classfoo.com/ccby/article/pS0Pce) | 底层内存管理 |
| [<typeinfo>](http://classfoo.com/ccby/article/pM3tcf) | 运行时类型信息 |
| [<exception>](http://classfoo.com/ccby/article/9vr4cg) | 异常处理 |
| [<initializer\_list>](http://classfoo.com/ccby/article/if0Jch) C++11 | [std::initializer\_list](http://classfoo.com/ccby/article/XycMbP) 类模板 |
| [<csignal>](http://classfoo.com/ccby/article/hudBci) | 用与信号量（Signal）管理的函数及宏常量 |
| [<csetjmp>](http://classfoo.com/ccby/article/ivpLcj) | 包含用来跳转到执行上下文（Execution context）的宏或函数 |
| [<cstdalign>](http://classfoo.com/ccby/article/Mfgyck) C++11 | 定义了 \_\_alignas\_is\_defined 宏常量 |
| [<cstdarg>](http://classfoo.com/ccby/article/dnSbcl) | 可变长参数列表的处理 |
| [<cstdbool>](http://classfoo.com/ccby/article/Xvw1cm) C++11 | 定义了 \_\_bool\_true\_false\_are\_defined 宏常量 |
| [<ctime>](http://classfoo.com/ccby/article/u3IrcL) | 已在“通用工具库”中定义 |

## 2.[诊断库](http://classfoo.com/ccby/article/WlfKr#sec_NUEYiF)

|  |  |
| --- | --- |
| [<stdexcept>](http://classfoo.com/ccby/article/nFS4cy) | 提供了一些标准异常对象 |
| [<cassert>](http://classfoo.com/ccby/article/ivYdcz) | 提供了用于断言的宏 |
| [<cerrno>](http://classfoo.com/ccby/article/S3NhcA) | 提供了一个保存了最近错误号的宏 |
| [<system\_error>](http://classfoo.com/ccby/article/hIsccB) C++11 | 定义了 std::error\_code ，用来依赖于平台的（Platform-dependent）错误码 |

## 3.[通用工具库](http://classfoo.com/ccby/article/WlfKr#sec_wYB8iG)

|  |  |
| --- | --- |
| [<utility>](http://classfoo.com/ccby/article/7coecC) | 大量实用程序组件 |
| [<tuple>](http://classfoo.com/ccby/article/Iv1RcD) C++11 | [std::tuple](http://classfoo.com/ccby/article/irTzbR) 类模板，使C++程序支持多元组 |
| [<bitset>](http://classfoo.com/ccby/article/p82BcE) | [std::bitset](http://classfoo.com/ccby/article/3pCnbS) 类模板，提供位集操作 |
| [<memory>](http://classfoo.com/ccby/article/ePpjcF) | 用于上层内存管理，包含智能指针、内存分配器、哈希（Hash）、类型转换等。 |
| [<cstdlib>](http://classfoo.com/ccby/article/XTBfcd) | 一般用途的实用程序，比如程序控制、动态内存分配、随机数、排序及搜索 |
| [<cstring>](http://classfoo.com/ccby/article/RpRRcG) | 已在“字符串库”中定义 |
| [<functional>](http://classfoo.com/ccby/article/Qz6HcH) | 提供被设计用来支持标准算法的函数对象 |
| [<type\_traits>](http://classfoo.com/ccby/article/VCIVcI) C++11 | 编译时类型信息 |
| [<ratio>](http://classfoo.com/ccby/article/bLW2cJ) C++11 | 编译时有理数运算 |
| [<chrono>](http://classfoo.com/ccby/article/5We1cK) C++11 | C++时间实用程序 |
| [<ctime>](http://classfoo.com/ccby/article/u3IrcL) | C型时间/日期实用程序 |
| [<scoped\_allocator>](http://classfoo.com/ccby/article/JT8WcM)C++11 | 支持可嵌套的内存分配器（Allocator） |
| [<typeindex>](http://classfoo.com/ccby/article/nf5jcN) C++11 | [std::type\_index](http://classfoo.com/ccby/article/DibgcY) 类 |

## 4.[字符串库](http://classfoo.com/ccby/article/WlfKr#sec_qPGliH)

|  |  |
| --- | --- |
| [<string>](http://classfoo.com/ccby/article/TWd5cZ) | std::basic\_string 类模板 |
| [<cctype>](http://classfoo.com/ccby/article/uaR5c0) | 用来判断字符数据类型的函数，比如：std::isdigit、std::isspace、std::islower |
| [<cwctype>](http://classfoo.com/ccby/article/Uw8gc1) | 用来判断宽字符数据类型的函数，比如：std::iswdigit、std::iswspace、std::iswlower |
| [<cstring>](http://classfoo.com/ccby/article/RpRRcG) | 大量窄字符字符串处理函数 |
| [<cwchar>](http://classfoo.com/ccby/article/47uDc2) | 大量宽字符及多字符字符串处理函数 |
| [<cstdlib>](http://classfoo.com/ccby/article/XTBfcd) | 已在“通用工具库”中定义 |
| [<cuchar>](http://classfoo.com/ccby/article/TcnBc3) C++11 | C型Unicode字符转换函数 |

## 5.[本地化库](http://classfoo.com/ccby/article/WlfKr#sec_4DCPiI)

|  |  |
| --- | --- |
| [<locale>](http://classfoo.com/ccby/article/Hgkxc4) | 本地化相关，包括对国际字符分类的的支持，对字符串排序规则的支持等 |
| [<codecvt>](http://classfoo.com/ccby/article/hstYc5) **C++11** | Unicode转换机制 |
| [<clocale>](http://classfoo.com/ccby/article/YUVkc6) | C型本地化相关 |

## 6.[容器库](http://classfoo.com/ccby/article/WlfKr#sec_1nP3iJ)

|  |  |
| --- | --- |
| [<array>](http://classfoo.com/ccby/article/gNnddI) C++11 | [std::array](http://classfoo.com/ccby/article/bnrZv) 容器 |
| [<deque>](http://classfoo.com/ccby/article/sJhydJ) | [std::deque](http://classfoo.com/ccby/article/ryIpw) 容器 |
| [<forward\_list>](http://classfoo.com/ccby/article/zLZQdK) C++11 | [std::forward\_list](http://classfoo.com/ccby/article/5wWLx) 容器 |
| [<list>](http://classfoo.com/ccby/article/El1KdL) | [std::list](http://classfoo.com/ccby/article/Hdj4y) 容器 |
| [<vector>](http://classfoo.com/ccby/article/KCu6dM) | [std::vector](http://classfoo.com/ccby/article/jnevK) 容器 |
| [<map>](http://classfoo.com/ccby/article/aOZxdN) | [std::map](http://classfoo.com/ccby/article/WQ9qz) 及 [std::multimap](http://classfoo.com/ccby/article/a8VKA) 关联容器 |
| [<set>](http://classfoo.com/ccby/article/XQA8dO) | [std::set](http://classfoo.com/ccby/article/w2pUE) 及 [std::multiset](http://classfoo.com/ccby/article/kSPmB) 关联容器 |
| [<unordered\_map>](http://classfoo.com/ccby/article/pKVIdP) C++11 | [std::unordered\_map](http://classfoo.com/ccby/article/S3XoG) 及 [std::unordered\_multimap](http://classfoo.com/ccby/article/hedAH) 无序关联容器 |
| [<unordered\_set>](http://classfoo.com/ccby/article/G9Z6dQ) C++11 | [std::unordered\_set](http://classfoo.com/ccby/article/qNNOJ) 及 [std::unordered\_multiset](http://classfoo.com/ccby/article/C18YI) 无序关联容器 |
| [<queue>](http://classfoo.com/ccby/article/WdLxdR) | [std::queue](http://classfoo.com/ccby/article/DMS8D) 及 [std::priority\_queue](http://classfoo.com/ccby/article/Q3jxC) 容器适配器 |
| [<stack>](http://classfoo.com/ccby/article/fmoWdS) | [std::stack](http://classfoo.com/ccby/article/s4SvF) 容器适配器 |

## 7.[迭代器库](http://classfoo.com/ccby/article/WlfKr#sec_9bn5iK)

|  |  |
| --- | --- |
| [<iterator>](http://classfoo.com/ccby/article/7ZPydZ) | 容器迭代器 |

## 8.[算法库](http://classfoo.com/ccby/article/WlfKr#sec_lH34iL)

|  |  |
| --- | --- |
| [<algorithm>](http://classfoo.com/ccby/article/7ZqseK) | 用于操作容器的算法 |
| [<cstdlib>](http://classfoo.com/ccby/article/XTBfcd) | 已在“通用工具库”中定义 |

### 9.数值库

|  |  |
| --- | --- |
| [<cfenv>](http://classfoo.com/ccby/article/THraf4) C++11 | 浮点数环境访问函数 |
| [<complex>](http://classfoo.com/ccby/article/W82Sf5) | 复数类型 |
| [<random>](http://classfoo.com/ccby/article/ISjmf6) | 随机数生成器与随机数分布 |
| [<valarray>](http://classfoo.com/ccby/article/vRVof7) | 包含用来表示及操作值的数组的类，比如获得一个数组中所有值的和、最大值、最小值等 |
| [<numeric>](http://classfoo.com/ccby/article/Md8qf8) | 对容器中的值进行数值操作 |
| [<cmath>](http://classfoo.com/ccby/article/JMI7f9) | 普通的数学函数 |
| [<ctgmath>](http://classfoo.com/ccby/article/bxXRga) C++11 | 简单的包含头文件[<ccomplex>](http://classfoo.com/ccby/article/Legjgc)及[<cmath>](http://classfoo.com/ccby/article/JMI7f9) |
| [<cstdlib>](http://classfoo.com/ccby/article/XTBfcd) | 已在“通用工具库”中定义 |

## 10.[输入/输出库](http://classfoo.com/ccby/article/WlfKr#sec_XghTiN)

|  |  |
| --- | --- |
| [<iosfwd>](http://classfoo.com/ccby/article/ZLOKgd) | 前置声明所有输入/输出库中的类 |
| [<iostream>](http://classfoo.com/ccby/article/Yxj7ge) | [std::basic\_iostream](http://classfoo.com/ccby/article/ukxDY) 类模板 |
| [<ios>](http://classfoo.com/ccby/article/TZEigf) | [std::ios\_base](http://classfoo.com/ccby/article/Rr2tbd) 及 [std::basic\_ios](http://classfoo.com/ccby/article/hxetX) 类模板 |
| [<streambuf>](http://classfoo.com/ccby/article/7HhWgg) | [std::basic\_streambuf](http://classfoo.com/ccby/article/wN6t4) 类模板 |
| [<istream>](http://classfoo.com/ccby/article/hkttgh) | [std::basic\_istream](http://classfoo.com/ccby/article/g5F8Z) 类模板 |
| [<ostream>](http://classfoo.com/ccby/article/War6gi) | [std::basic\_ostream](http://classfoo.com/ccby/article/VzWg2) 类模板 |
| [<iomanip>](http://classfoo.com/ccby/article/I21ggj) | 控制输入及输出格式的输助函数 |
| [<sstream>](http://classfoo.com/ccby/article/a7Xlgk) | [std::basic\_stringstream](http://classfoo.com/ccby/article/fCIS6)、[std::basic\_istringstream](http://classfoo.com/ccby/article/rWCL0)、[std::basic\_ostringstream](http://classfoo.com/ccby/article/tasF3) 类模板 |
| [<fstream>](http://classfoo.com/ccby/article/1Pqmgl) | [std::basic\_fstream](http://classfoo.com/ccby/article/qsM1V)、[std::basic\_ifstream](http://classfoo.com/ccby/article/BPrCW)、[std::basic\_ofstream](http://classfoo.com/ccby/article/4bfJ1) 类模板 |
| [<cstdio>](http://classfoo.com/ccby/article/XFEwgm) | C型输入/输出函数 |
| [<cinttypes>](http://classfoo.com/ccby/article/JDMsgn) | 格式化相关的宏 |
| <strstream> 已弃用 | std::strstream、std::istrstream、std::ostrstream 类模板 |

## 11.[正则表达式库](http://classfoo.com/ccby/article/WlfKr#sec_gGMMiO)

|  |  |
| --- | --- |
| [<regex>](http://classfoo.com/ccby/article/bHzdgx) C++11 | 提供了支持处理正则表达式的类、算法及迭代器 |

## 12.[原子操作库](http://classfoo.com/ccby/article/WlfKr#sec_XjTFiP)

|  |  |
| --- | --- |
| [<atomic>](http://classfoo.com/ccby/article/8koogL) C++11 | 原子操作相关 |

## 13.[线程支持库](http://classfoo.com/ccby/article/WlfKr#sec_SDdRiQ)

|  |  |
| --- | --- |
| [<thread>](http://classfoo.com/ccby/article/Bd12gO) C++11 | [std::thread](http://classfoo.com/ccby/article/zvLngS) 类及一些用来创建或管理线程的线程支持函数 |
| [<mutex>](http://classfoo.com/ccby/article/hBSsgP) C++11 | 互斥原语 |
| [<condition\_variable>](http://classfoo.com/ccby/article/avNigQ) C++11 | 线程等待条件 |
| [<future>](http://classfoo.com/ccby/article/NaCngR) C++11 | 提供支持异步执行的原语 |

## 14.[其它头文件](http://classfoo.com/ccby/article/WlfKr#sec_Si0CiR)

|  |  |
| --- | --- |
| [<ciso646>](http://classfoo.com/ccby/article/QARUha) | 空的头文件，c头文件<iso646.h>中定义的宏在C++中已成为关键字 |
| [<ccomplex>](http://classfoo.com/ccby/article/Legjgc) | 仅仅包含头文件[<complex>](http://classfoo.com/ccby/article/W82Sf5) |

## 15.[过时的头文件](http://classfoo.com/ccby/article/WlfKr#sec_sPhXiS)

|  |  |
| --- | --- |
| <assert.h> 已弃用 | 跟[<cassert>](http://classfoo.com/ccby/article/ivYdcz)的区别是内部所有名字都是全局的 |
| <complex.h> 已弃用 | 跟[<ccomplex>](http://classfoo.com/ccby/article/Legjgc)的区别是内部所有名字都是全局的 |
| <ctype.h> 已弃用 | 跟[<cctype>](http://classfoo.com/ccby/article/uaR5c0)的区别是内部所有名字都是全局的 |
| <errno.h> 已弃用 | 跟[<cerrno>](http://classfoo.com/ccby/article/S3NhcA)的区别是内部所有名字都是全局的 |
| <fenv.h> 已弃用 | 跟[<cfenv>](http://classfoo.com/ccby/article/THraf4)的区别是内部所有名字都是全局的 |
| <float.h> 已弃用 | 跟[<cfloat>](http://classfoo.com/ccby/article/b7Klcb)的区别是内部所有名字都是全局的 |
| <inttypes.h> 已弃用 | 跟[<cinttypes>](http://classfoo.com/ccby/article/JDMsgn)的区别是内部所有名字都是全局的 |
| <iso646.h> 已弃用 | 跟[<ciso646>](http://classfoo.com/ccby/article/QARUha)的区别是内部所有名字都是全局的 |
| <limits.h> 已弃用 | 跟[<climits>](http://classfoo.com/ccby/article/FvqFca)的区别是内部所有名字都是全局的 |
| <locale.h> 已弃用 | 跟[<clocale>](http://classfoo.com/ccby/article/YUVkc6)的区别是内部所有名字都是全局的 |
| <math.h> 已弃用 | 跟[<cmath>](http://classfoo.com/ccby/article/JMI7f9)的区别是内部所有名字都是全局的 |
| <setjmp.h> 已弃用 | 跟[<csetjmp>](http://classfoo.com/ccby/article/ivpLcj)的区别是内部所有名字都是全局的 |
| <signal.h> 已弃用 | 跟[<csignal>](http://classfoo.com/ccby/article/hudBci)的区别是内部所有名字都是全局的 |
| <stdalign.h> 已弃用 | 跟[<cstdalign>](http://classfoo.com/ccby/article/Mfgyck)的区别是内部所有名字都是全局的 |
| <stdarg.h> 已弃用 | 跟[<cstdarg>](http://classfoo.com/ccby/article/dnSbcl)的区别是内部所有名字都是全局的 |
| <stdbool.h> 已弃用 | 跟[<cstdbool>](http://classfoo.com/ccby/article/Xvw1cm)的区别是内部所有名字都是全局的 |
| <stddef.h> 已弃用 | 跟[<cstddef>](http://classfoo.com/ccby/article/eCLbb8)的区别是内部所有名字都是全局的 |
| <stdint.h> 已弃用 | 跟[<cstdint>](http://classfoo.com/ccby/article/Fn54cc)的区别是内部所有名字都是全局的 |
| <stdio.h> 已弃用 | 跟[<cstdio>](http://classfoo.com/ccby/article/XFEwgm)的区别是内部所有名字都是全局的 |
| <stdlib.h> 已弃用 | 跟[<cstdlib>](http://classfoo.com/ccby/article/XTBfcd)的区别是内部所有名字都是全局的 |
| <string.h> 已弃用 | 跟[<cstring>](http://classfoo.com/ccby/article/RpRRcG)的区别是内部所有名字都是全局的 |
| <tgmath.h> 已弃用 | 跟[<ctgmath>](http://classfoo.com/ccby/article/bxXRga)的区别是内部所有名字都是全局的 |
| <time.h> 已弃用 | 跟[<ctime>](http://classfoo.com/ccby/article/u3IrcL)的区别是内部所有名字都是全局的 |
| <uchar.h> 已弃用 | 跟[<cuchar>](http://classfoo.com/ccby/article/TcnBc3)的区别是内部所有名字都是全局的 |
| <wchar.h> 已弃用 | 跟[<cwchar>](http://classfoo.com/ccby/article/47uDc2)的区别是内部所有名字都是全局的 |
| <wctype.h> 已弃用 | 跟[<cwctype>](http://classfoo.com/ccby/article/Uw8gc1)的区别是内部所有名字都是全局的 |

# C++11之输入输出库

## 1.C++输入输出机制

C++语言的输入输出机制包含3层，前两层是从传统的C语言继承而来，分别是底层I/O和高层I/O，第3层是C++中增添的流类库，这是本章讨论的重点。

（1）底层I/O:底层I/O依赖于操作系统来实现，调用操作系统的功能对文件进行输入输出处理，具有较高的速度。底层I/O将外部设备和磁盘文件都等同于逻辑文件，采用相同的方法进行处理，一般过程为“打开文件”、“读写文件”，“关闭文件”，这些是通过一组底层I/O函数来完成的，这些函数定义在头文件io.h中。

（2）高层I/O:高层I/O是在底层I/O的基础上扩展起来的，仍旧将外部设备和磁盘文件统一处理，但处理的方式更为灵活，提供的一组处理函数定义在头文件stdio.h中，新的C++标准头文件为<cstdio>，提供的这些函数大体可分为两类：一般文件函数（外部设备和磁盘文件）和标准I/O函数。

（3）流类库：除了从C语言中继承了上述两种I/O机制外，C++还特有一种输出机制：流类库(即iostream类库)，这是C++所特有的，iostream类库为内置类型对象提供了输入输出支持，也支持文件的输入输出，另外，类的设计者可以通过运算符重载机制对iostream库的扩展，来支持自定义类型的输入输出操作。

## 2.流类库

“流”的定义为“流是（表达）读写数据的一种可移植的方法，它为一般的I/O操作提供了灵活有效的手段。一个流是一个由指针操作的文件或者是一个物理设备，而这个指针正是指向了这个流。” 可以把流看做“水管”，这样输入流指的是从程序外部输入数据，类似于水通过水管流进程序，输出流恰好相反，是从程序内部输出到外部。这里要注意的是，输入输出的参照物是程序。就C++程序而言， I/O操作可以简单地看作是从程序移进或移出字节，这种搬运的过程便称为流(stream)。程序员只需要关心是否正确地输出了字节数据，以及是否正确地输入了要读取字节数据，特定I/O设备的细节对程序员是隐藏的。

**注意：流类不可拷贝，**标准流**不可赋值，**不可移动，只有指定缓冲区才可新建。

同string类一样，流也分为宽流和窄流。如：istream,wistream。

### 2.1.流类库的层次

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **流** | **类** | **模板** | **对象** | **头文件** | **前置声明** | **作用** |
| 基本标准流 | [ios\_base](http://classfoo.com/ccby/article/Rr2tbd) | 类是模板的实例 | 对象是类的实例 | [<ios>](http://classfoo.com/ccby/article/TZEigf) |  | 用于流的基类 |
| [ios](http://classfoo.com/ccby/article/syZAbb) | [basic\_ios](http://classfoo.com/ccby/article/hxetX) |  | [<ios>](http://classfoo.com/ccby/article/TZEigf) | iosfwd | 用于流的基类 |
| 标准流 | [istream](http://classfoo.com/ccby/article/HE5obe) | [basic\_istream](http://classfoo.com/ccby/article/g5F8Z) | cin | [<istream>](http://classfoo.com/ccby/article/hkttgh) | 输入流 |
| [ostream](http://classfoo.com/ccby/article/wMZDbh) | [basic\_ostream](http://classfoo.com/ccby/article/VzWg2) | cou、cer、clog | [<ostream>](http://classfoo.com/ccby/article/War6gi) | 输出流 |
| [iostream](http://classfoo.com/ccby/article/f1nMbc) | [basic\_iostream](http://classfoo.com/ccby/article/ukxDY) |  | [<iostream>](http://classfoo.com/ccby/article/Yxj7ge) | 输入、输出流 |
| 文件流 | [ifstream](http://classfoo.com/ccby/article/FooLba) | [basic\_ifstream](http://classfoo.com/ccby/article/BPrCW) |  | [<fstream>](http://classfoo.com/ccby/article/1Pqmgl) | 输入文件流 |
| [ofstream](http://classfoo.com/ccby/article/wlXDbg) | [basic\_ofstream](http://classfoo.com/ccby/article/4bfJ1) |  | 输出文件流 |
| [fstream](http://classfoo.com/ccby/article/IPQH9) | [basic\_fstream](http://classfoo.com/ccby/article/qsM1V) |  | 输入、输出文件流 |
| 字符串流 | [istringstream](http://classfoo.com/ccby/article/UfSabf) | [basic\_istringstream](http://classfoo.com/ccby/article/rWCL0) |  | [<sstream>](http://classfoo.com/ccby/article/a7Xlgk) | 输入字符串流 |
| [ostringstream](http://classfoo.com/ccby/article/Od1pbi) | [basic\_ostringstream](http://classfoo.com/ccby/article/tasF3) |  | 输出字符串流 |
| [stringstream](http://classfoo.com/ccby/article/XzXcbl) | [basic\_stringstream](http://classfoo.com/ccby/article/fCIS6) |  | 输入/输出字符串流 |
| 流缓冲区 | [streambuf](http://classfoo.com/ccby/article/0RsDbj) | [basic\_streambuf](http://classfoo.com/ccby/article/wN6t4) |  | [<streambuf>](http://classfoo.com/ccby/article/7HhWgg)[​​](http://classfoo.com/ccby/article/ZLOKgd) | 用于流的缓存基类 |
| [filebuf](http://classfoo.com/ccby/article/af588) | [basic\_filebuf](http://classfoo.com/ccby/article/DpwWU) |  | [<fstream>](http://classfoo.com/ccby/article/1Pqmgl)[​​](http://classfoo.com/ccby/article/ZLOKgd) | 文件流缓存 |
| [stringbuf](http://classfoo.com/ccby/article/V6Bnbk) | [basic\_stringbuf](http://classfoo.com/ccby/article/XVms5) |  | [<sstream>](http://classfoo.com/ccby/article/a7Xlgk) |  | 字符串流缓存 |

### 2.2.ios继承关系



### 2.3流类的构造函数

标准输入输出流类：

explicit \*stream (streambuf<char\_type,traits\_type>\* sb);//**指定缓冲区才可新建**

\*stream& (const \*stream&) = delete;// **不可复制**

protected: *\**stream& (\*stream&& x);//**不可转移**

文件流类：

\*fstream();//建立空文件流

explicit \*fstream (const \*char\* filename, ios::openmode mode = std::ios::in| std::ios::out);

explicit \*fstream (const \*string& filename, ios::openmode mode =std:: ios::in| std::ios::out);

\*fstream (const\*fstream&) = delete;**//不可复制**

\*fstream (\*fstream&& x);

字符串流：

explicit \*stringstream (ios::openmode mode = std::ios::in| std::ios::out);//建立空字符流

explicit \*stringstream ( const \*string &str, ios ::openmode mode = std::ios::in|std::ios::out);//建立流

\*stringstream (const \*stringstream&) = delete;**//不可复制**

\*stringstream (\*stringstream&& x);

filebuf

filebuf();

filebuf (const filebuf&) = delete;/**/不可复制**

filebuf (filebuf&& x);

streambuf

protected:streambuf();//**不可新建**

protected:streambuf (const streambuf& x);/**/不可复制**

stringbuf

explicit stringbuf (ios::openmode which = std::ios::in | std::ios::out);

explicit stringbuf (const \*string&str, ios::openmode mode= std::ios::in | std::ios::out);

stringbuf (const stringbuf&) = delete;

stringbuf (stringbuf&& x)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 标准流：1-ios，2- istream，3-ostream，4-iostream，  文件流：5- ifstream，6- ofstream，7- fstream，  字符流：8- istringstream，9- ostringstream，10- stringstream | | | | | | | | | | | |
| 方法 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 功能 |
| good |  |  |  |  |  |  |  |  |  |  | 测试流的状态是否是goodbit |
| eof |  |  |  |  |  |  |  |  |  |  | 测试流的状态是否是eofbit |
| fail |  |  |  |  |  |  |  |  |  |  | 测试流的状态是否是failbit |
| bad |  |  |  |  |  |  |  |  |  |  | 测试流的状态是否是badbit |
| rdstate |  |  |  |  |  |  |  |  |  |  | 读取流的状态 |
| setstate |  |  |  |  |  |  |  |  |  |  | 设置流的状态 |
| clear |  |  |  |  |  |  |  |  |  |  | 重置流的状态位为goodbit |
| tie |  |  |  |  |  |  |  |  |  |  | 获取或设置流对象的输出流 |
| rdbuf |  |  |  |  |  |  |  |  |  |  | 获取或设置流对象的流缓冲区 |
| get |  |  |  |  |  |  |  |  |  |  | 从流中提取一个或多个字符 |
| unget |  |  |  |  |  |  |  |  |  |  | 放回上一次从流中提取的字符 |
| peek |  |  |  |  |  |  |  |  |  |  | 返回从流中读取的一个字符 |
| putback |  |  |  |  |  |  |  |  |  |  | 在流的末尾添加一个字符 |
| getline |  |  |  |  |  |  |  |  |  |  | 获取多个字符，遇到换行符停止 |
| ignore |  |  |  |  |  |  |  |  |  |  | 获取多个字符，忽略换行符 |
| read |  |  |  |  |  |  |  |  |  |  | 以二进制方式读取指定个数的字符 |
| readsome |  |  |  |  |  |  |  |  |  |  | 从缓冲区以二进制读取指定数目的字符 |
| gcount |  |  |  |  |  |  |  |  |  |  | 返回从流中读取的字节数 |
| tellg |  |  |  |  |  |  |  |  |  |  | 获取流中输入标志位的当前位置 |
| seekg |  |  |  |  |  |  |  |  |  |  | 设置流中输入标志位的当前位置 |
| sync |  |  |  |  |  |  |  |  |  |  | 将当前缓冲区与流的源同步 |
| swap |  |  |  |  |  |  |  |  |  |  | 与指定流交换除缓冲区外的所有信息 |
| std::wap |  |  |  |  |  |  |  |  |  |  | 交换两个流的除缓冲区外的所有信息 |
| put |  |  |  |  |  |  |  |  |  |  | 输出一个字符 |
| write |  |  |  |  |  |  |  |  |  |  | 以二进制方式输出指定个数的字节 |
| tellp |  |  |  |  |  |  |  |  |  |  | 获取流中输出标志位的当前位置 |
| seekp |  |  |  |  |  |  |  |  |  |  | 设置流中输出标志位的当前位置 |
| fllush |  |  |  |  |  |  |  |  |  |  | 清空缓冲区，并与流的源同步 |
| open |  |  |  |  |  |  |  |  |  |  | 打开文件 |
| is\_open |  |  |  |  |  |  |  |  |  |  | 当前文件流是否打开了文件 |
| close |  |  |  |  |  |  |  |  |  |  | 关闭文件 |
| str |  |  |  |  |  |  |  |  |  |  | 返回字符流的内容 |
| operator= |  |  |  |  |  |  |  |  |  |  | 一个流对象向另一个流对象赋值 |

### 2.4.基本输入输出流

头文件： <ios>

ios\_base类：定义了各种数据类型和格式类型以及格式操作函数

basic\_ios类：定义了基本的流操作。

std::ios::state取值为：

badbit 系统级故障，不可恢复

failbit 可以恢复的错误

eofbit 碰到了文件结尾

goodbit 有效状态

#### 2.4.1.查询流的状态

bool bad()

如果当前的流发生致命的错误，bad()函数返回**true**，否则返回**false**

bool fail()

如果当前流发生错误fail()函数返回true ，否则返回false

bool eof()

如果到达相关联的输入文件的末尾，eof()函数返回true，否则返回false。

bool good()

如果当前流没有发生错误，函数good()返回true ，否则返回false

#### 2.4.2.读取流的状态

iostate rdstate()const

#### 2.4.3.设置流的状态

void setstate( iostate state )

#### 2.4.4.重置流的状态

void clear( iostate flags = goodbit );

clear清除与当前流相关联的标志。默认标志是goodbit它清除所有标志.否则只有指定的标志被清除

#### **2.4.5.获取**或设置当前对象的的输出流

（1）ostream\* tie() const;

（2）ostream\* tie(ostream \* str );

（1）获取当前流绑定的输出流，没有返回NULL。

（2）将对象的输出流绑定为为str，返回之前绑定的输出流。

#### 2.4.6.管理关联的流缓冲区

（1）streambuf \* rdbuf（） const ;

（2）streambuf \* rdbuf（ std :: basic\_streambuf \* sb）;

（1）返回对象关联的流缓冲区。如果没有关联的流缓冲区，则返回NULL。

（2）将对象关联的流缓冲区设置为sb。并通过调用clear （）来清除错误状态。并返回之前关联的流缓冲区。如果没有关联的流缓冲区，则返回NULL。

#### 2.4.7.protected成员函数

void init( std::basic\_streambuf \*sb );

init将关联的流缓冲区设置为sb并初始化内部状态。

void swap( basic\_ios& other )

swap（）用以与other交换状态，但不交换流缓冲区，rdbuf()和 other.rdbuf() 返回调用前的值。只能派生类访问。

void move( basic\_ios& other );

move 用other的状态替换当前对象的状态，但不替换流缓冲区，调用该函数后, rdbuf()返回NULL, other.rdbuf()返回调用前的值, other.tie() 返回0.

void set\_rdbuf( std::basic\_streambuf<CharT,Traits>\* sb );

set\_rdbuf将对象关联的流缓冲区设置为sb但不调用clear （）来清除错误状态。

### 2.5.标准输入流

头文件： <istream>

#### 2.5.1格式化输入>>

istream& operator>>( 基本数据类型& value );

istream& operator>>( ios& (\*func)(std::ios &) );

istream& operator>>( ios& (\*func)( ios&) );

istream& operator>>( istream& (\*func)( istream&) );

istream& operator>>(streambuf\* sb );

#### 2.5.2.提取字符：

int get();

istream &get( char &ch );

istream &get( char \*buffer, streamsize num );

istream &get( char \*buffer, streamsize num, char delim );

istream &get( streambuf &buffer );

istream &get( streambuf &buffer, char delim );

get()方法的重载对应：以下这些:

读入一个字符并返回它的值，

读入一个字符并把它存储在ch，

读取字符到buffer直到num - 1个字符被读入, 或者碰到EOF或换行标志，

读取字符到buffer直到已读入num - 1 个字符，或者碰到EOF或delim(delim直到下一次不会被读取)，

读取字符到buffer中，直到碰到换行或EOF，

读取字符到buffer中，直到碰到换行，EOF或delim。(相反, delim直到下一个get()不会被读取 ).

#### 2.5.3.读取一个字符

int\_type peek();

函数peek()用于输入流中，并返回在流中的下一个字符，如果是处于被入的文件的结尾处返回EOF。与get()的peek()不会把字符从流中移除。

#### 2.5.4.放回上一次提取的字符

istream& unget()

将之前读取的对象放回流。与putback()的区别在于只能放回上次提取的字符。

#### 2.5.5.放入一个字符

istream &putback( char ch );

将字符ch放入流中，以便下次读取。与unget()的区别在于可以放回任意字符ch。

#### 2.5.6.读取字符串

istream &getline( char \*buffer, streamsize num );

istream &getline( char \*buffer, streamsize num, char delim );

getline()函数用于输入流，读取字符到buffer中，直到下列情况发生：

num - 1个字符已经读入,

碰到一个换行标志，

碰到一个EOF，

或者，任意地读入，直到读到字符delim。delim字符不会被放入buffer中

#### 2.5.7.读取字符串（忽略回车，换行）

istream &ignore( streamsize num=1, int delim=EOF );

ignore()函数用于输入流。它读入字符，直到已经读了num 个字符(默认为1)或是直到字符delim 被读入(默认为EOF).与getline()的区别在于忽略了换行符。

#### 2.5.8.读取字节块

istream &read( char \*buffer, streamsize num );

函数read()用于输入流，从流中读取num 个字节，再将字符放入buffer。如果碰到EOF，read()中止，丢弃不论多少个字节已经放入。可以是设备或者缓冲区。

#### 2.5.9.读取字节块（从缓冲区）

streamsize readsome( char\* buffer, std::streamsize num );

函数read()用于输入流，从流中读取num 个字节，再将字符放入buffer。如果碰到EOF，read()中止，丢弃不论多少个字节已经放入。与read()的区别在于只能读取缓冲区的流.

#### 2.5.10.计算读入的字符数

streamsize gcount();

函数gcount()被用于输入流，并返回上一次非格式化输入操作被读入的字符的数目。可以被其他读取函数修改。

#### 2.5.11.获取输入标志位

pos\_type tellg();

tellg()函数用于输入流，并返回流中输入标志位的当前位置。

#### 2.5.12.设置输入标志位

istream &seekg( off\_type offset, ios::seekdir origin );

istream &seekg( pos\_type position );

函数seekg()用于重新设置"输入标志位"到当前流的从origin偏移offset个字节的位置上，或是置输入标志位在position位置。

origin取值： beg 流开始

end 流结束

cur 当前位置

#### 2.5.13.同步缓冲区

int sync （）

sync（）函数将当前缓冲区与流的源同步，成功返回0，失败或者源未采用缓冲返回-1。

#### 2.5.14.交换输入流状态

void swap(basic\_istream& rhs);

swap（）函数将调用basic\_ios::swap(rhs) 来与rhs交换除流缓冲区外的所有状态信息，包括gcount（）的值。

#### 2.5.15.全局对象

窄字符输入流对象cin和宽字符输入流对象wcin。

### 2.6.标准输出流

头文件<ostream>

#### 2.6.1.格式化输出<<

ostream& operator<<( 基本数据类型&value );

ostream& operator<<( std::nullptr\_t );

ostream& operator<<(streambuf\* sb);

ostream& operator<<(ios & (\*func)( ios &) );

ostream& operator<<(ios& (\*func)( ios&) );

ostream& operator<<( ostream& (\*func)( ostream&) );

#### 2.6.2.输出一个字符

ostream &put( char ch );

函数put()用于输出流，并把字符ch写入流中

#### 2.6.3.输出字符块

ostream& write( const char \* s, streamsize count );

write（）函数将s中的前count个字符输出到流中。

#### 2.6.4.获取输出标志位

pos\_type tellp()

#### 2.6.5设置输出标志位

ostream &seekp( off\_type offset, ios::seekdir origin );

ostream &seekp( pos\_type position );

函数seekp()用于重新设置"输出标志位"到当前流的从origin偏移offset个字节的位置上，或是置输出标志位在position位置。

origin取值： beg 流开始

end 流结束

cur 当前位置

#### 2.6.6.刷新缓冲区

ostream ＆ flush （）;

flush()函数可以引起当把前流的缓冲区写出到附属设备中去。这个函数对于打印调试信息很用处，因为程序有可能在把缓冲区内容写出到屏幕之前被中断。

#### 2.6.7.交换输出流状态

void swap(basic\_istream& rhs);

swap（）函数将调用basic\_ios::swap(rhs) 来与rhs交换除流缓冲区外的所有状态信息。

#### 2.6.8.全局对象

窄字符输出流对象cout和宽字符输出流对象wcout。

### 2.7标准输入输出流

头文件： <iostream>

标准输入输出流继承了标准输入流，标准输出流的方法，但是中注意：swap函数是保护成员，只能被派生类访问。

保护成员函数：

void swap(iostream& other );

swap（）函数将调用basic\_istream::swap(rhs) 来与othre交换除流缓冲区外的所有状态信息，且只能被fstream::swap(rhs)和stringsteram::swap(rhs)调用。

另外cin，cout就是定义在iostream类中的两个全局对象。

### 2.7.文件输入输出流

头文件： <fstream>

ifstream：文件输入流，只能读。

ofstream：文件输出流，只能写。

fstream：文件输入输出流，可读可写。

文件打开模式：

std::ios::in 打开文件做读操作

std::ios::out 打开文件做写操作，会删除原有数据

ios::app 在每次写之前找到文件尾

ios::trunc 打开文件时清空已存在的文件流

ios::ate 打开文件后立即定位到文件末尾

ios::binary 以二进制模式进行IO操作

ifstream/ ofstream/ fstream的构造函数和各自的open方法类似

文件操作方法：

open方法：

void open( const char \*filename, openmode)；

open用来打开一个文件，并返回文件流

参数filename：文件名或文件全名；

参数openmode：ifstream缺省为ios:in, ofstream缺省为std::ios::out，fstream缺省为ios:in|std::ios::out。

close方法：

void close()

close用来关闭流

is\_open方法：

bool is\_open()

监测文件流是否与文件关联，是返回true，否返回false。

rdbuf方法：

filebuf\* rdbuf() const

rdbuf返回指向底层原始文件设备对象的指针，此外还继承了basic\_ios::rdbuf()方法。

swap方法：

ifstream类中：

void swap（ifstream&rhs）

交换输入流状态，调用istream::swap

void swap(ifstream&lhs, ifstream &rhs )

交换输入流状态，调用iftream::swap

ofstream类中：

void swap（ofstream&rhs）

交换输出流状态，调用ostream::swap

void swap(ofstream&lhs, ofstream &rhs )

交换输出流状态，调用oftream::swap

fstream类中：

void swap（fstream&rhs）

交换输入流状态，调用iostream::swap

void swap(fstream&lhs, fstream &rhs )

交换输入流状态，调用ftream::swap

operator=

basic\_\*fstream& operator=( basic\_\*fstream&& other );

### 2.8.字符串输入输出流

头文件：<sstream>

istringstream：字符串输入流，只读

ostringstream：字符串输出流，只写

stringstream：字符串输入输出流，可读可写

构造函数：

basic\_stringstream(std::basic\_string & str, openmode mode);

参数str：string类，也可以是字符串，或者buf指针。

参数openmode：istringstream缺省为ios:in, ostringstream缺省为std::ios::out，stringstream缺省为ios:in|std::ios::out。

操作方法：

str方法：

string str()const

void str（basic\_string & str）；

返回当前str中的内容

rdbuf方法：

stringbuf \* rdbuf（） const

rdbuf返回指向底层原始字符串设备对象的指针，此外还继承了basic\_ios::rdbuf()方法。

swap方法：

istringstream类中：

void swap（istringstream&rhs）

交换输入流状态，调用istream::swap

void swap(istringstream&lhs, istringstream &rhs )

交换输入流状态，调用istringtream::swap

ostringstream类中：

void swap（ostringstream&rhs）

交换输出流状态，调用ostream::swap

void swap(ostringstream&lhs, ostringstream &rhs )

交换输出流状态，调用ostringtream::swap

stringstream类中：

void swap（stringstream&rhs）

交换输入流状态，调用iostream::swap

void swap(stringstream&lhs, stringstream &rhs )

交换输入流状态，调用stringstream::swap

operator=

basic\_\*stringstream& operator=( basic\_\*stringstream&& other );

# C++11字符串库

## 1.string概念

### 1.1.定义

c++中的具有字符串类型string。其实质仍然是用字符串数组的模板来实现的。string本质也是一种Vector，适用于vector的方法很大一部分也适用string。

注意：C++string不以‘\0’终止，但是以但是其提供的C风格方法是以‘\0’结尾。

### 1.2分类

string有四个类分别为string，wstring，u16string，u32string。分别对应basic\_string <char>、basic\_string <wchar\_t>、basic\_string <char16\_t>、basic\_string <char32\_t>，这只是对字符的编码方式不同，其余的成员函数，常量等完全一致。实际上是由于不同编译器对wchar\_t的大小规定不同，才导致C++11中新增char16\_t和char32\_t。

GNU：使用前缀u表示char16\_t字符常量和字符串常量如：u‘L’；u“lilili”;使用前缀U表示char32\_t字符常量和字符串常量如：U'L';U"lilili";

vs：只能加前缀L。

### 1.3.窄字符与宽字符：

宽字符与窄字符的区别在于对同一个字符的编码方式。一个字节表示一个字符叫做窄字符，如ASCii编码，多个字节来代表的一个字符称之为宽字符，而Unicode只是宽字符编码的一种实现，宽字符并不一定是Unicode。

比如一个汉字字符编码为2个字节，就是宽字符，一个汉字字符[ADCB]，在计算机中如果采用窄字符，则存储为[AD][CB],若采用宽字符则存储为[ADCB],采用宽字节的好处在于避免不同字符集的二义性，尽可能是全世界的语言都用一个字符集就可以表示。

### 1.4.定义对象

#### 1.4.1模板：

#include<vector>

template<class CharT,class Traits=std::char\_traits<CharT>,class Alloc=std::allocator<CharT>>class basic\_\*string;

typedef basic\_\*string<\*char\*> \*string;

string 对象名（构造参数）；

参数CharT：字符类型。必须是一个非数组的 C 型简单类型（POD type）。

参数Traits：字符特征类。该类定义了被 basic\_string 对象使用的字符的本质属性。Traits::char\_type 必须等同于 CharT。在类模板内部，使用其别名为 traits\_type 的成员类型。

参数Alloc：容器内部用来管理内存分配及释放的内存分配器的类型。这个参数是可选的，它的默认值是 std::allocator<T>，这个是一个最简单的非值依赖的（Value-independent）内存分配器。在类模板内部，使用其别名为 allocator\_type 的成员类型。

#### 1.4.2. 构造函数

string是字符串类，首先来看一下如何通过构造函数来声明一个字符串，string类的构造函数如所示:

explicit string ();// 默认（1）

string (const string& str); // 拷贝（2）

string (const string& str, size\_type pos, size\_type len = npos); // 子串（3）

string (const char\* s); // 来自 C 型字符串（4）

string (const char\* s, size\_type n); // from buffer (5)

string (size\_type n, char c); // 填充（6）

template <class InputIterator>string (InputIterator first, InputIterator last); // 范围（7）

string (initializer\_list<char> il); // 初始化列表（8）

string (string&& str) noexcept; // 移动（9）

## 2. 常用public成员函数

### 2.1.元素访问

下标操作符[]

reference operator[] (size\_type pos);

​const\_reference operator[] (size\_type pos) const;

at方法

reference at (size\_type pos);

​const\_reference at (size\_type pos) const;

可以使用下标操作符[]和函数at()对字符串中包含的字符进行访问，需要注意的是操作符[]并不检查索引是否有效（有效索引0~str.length()），如果索引失效，会引起未定义的行为，而at()会检查，如果使用 at()的时候索引无效，会抛出out\_of\_range异常。比如：对const修饰的string常量str来说，操作符[]对索引值str.length()仍然有效，返回值是’\0’，其他情况下（即str不是const修饰的字符串常量），str.length()索引都是无效的。

front方法

char& front();

​const char& front() const;

front()函数返回字符串第一个元素，空字符串中未定义。

back方法

char& back();

​const char& back() const;

back()函数返回字符串最后一个元素，空字符串中未定义。

**2.2.迭代器**

string支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 string::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 string 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前字符串反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的字符。

使用迭代器访问某个字符类似使用指针访问某个字符。可以简单地认为迭代器就是指针。

**2.2.1.普通迭代器**

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 string 对象中第一个字符的迭代器。如果 string 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 string::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 string::front 返回的是对第一个字符的引用（Reference）。string::begin 和 string::end 两个函数一起使用所代表的范围正好包含一个容器中所有的字符。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 string 对象中末尾之后（ Past-the-end）的字符的迭代器。如果 string 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的字符是容器中最后一个字符之后的字符，是一个“理论上的字符”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。string::begin 和 string::end 两个函数一起使用所代表的范围正好包含一个容器所有的字符。如果当前容器是空的，该函数的返回值跟 string::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 string 对象中最后一个字符的逆序迭代器。如果 string 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的字符。string::rbegin 返回值指向的字符刚好是 string::end 返回值指向的字符的前一个字符。区别于 string::back，当前函数返回的是指向最后一个字符的逆序随机访问迭代器（Reverse random access iterator），而 string::back 返回指向最后一个字符的引用（Reference）。string::rbegin 和 string::rend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 string 对象中第一个字符之前（ Preceding the first）的字符的逆序迭代器，如果 string 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个字符之前的字符是一个“理论上的字符”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。string::rbegin 和 string::rend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

**2.2.2.常迭代器**

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 string 对象中第一个字符的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 string::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 string 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。string::cbegin 和 string::cend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 string 对象中末尾之后（ Past-the-end）的字符的常迭代器（Const iterator）。末尾之后的字符是容器中最后一个字符之后的字符，是一个“理论上的字符”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 string::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 string 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 string::cbegin 相同。string::cbegin 和 string::cend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 string 对象中最后一个字符的常逆序迭代器。string::crbegin 和 string::crend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 string 对象中第一个字符之前（ Preceding the first）的字符的常逆序迭代器。第一个字符之前的字符是一个“理论上的字符”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。string::crbegin 和 string::crend 两个函数一起使用所代表的范围正好包含一个容器所有的字符。

### 2.3.字符串容量

size 方法

size\_type size() const noexcept;

返回当前字符串的字符数。等同于 std::distance(string::begin(), string::end())，这是当前字符串的字符个数，它不需要与当字符串的容量相等。

length方法

size\_type length() const noexcept;

返回当前字符串的字符数。等同于 std::distance(string::begin(), string::end())，这是当前字符串的字符个数，它不需要与当字符串的容量相等。

max\_size

size\_type max\_size() const noexcept;

返回 string 能容纳的最大字符个数。是一个当前字符串能够达到的受限于操作系统或编译器的最大可能大小。但实事上字符串并不保证能够达到这个大小：在达到这个上限之前，已经出现内存分配失败。

resize方法

void resize (size\_type n);

void resize (size\_type n, charT val);

调整字符串的大小使其能够存放 n 个字符。返回新string的大小。会改变size和capacity。

如果 n 比当前字符串大小（Size）小，字符串中的内容将被减少到只剩前 n 个字符，超出的部份将被移除（或销毁）。

如果 n 比当前字符串大小（Size）大，将通过在字符串尾部插入足够的字符使内容增加到 n 个字符。如果提供了 val ，新的字符将被初始化为 val 的拷贝，否则将被值初始化（Value-initialized）。

如果 n 比当前字符串容量（Capacity）大，将会对已分配内存空间自动进行一次内存重分配。

capacity 方法

size\_type capacity() const noexcept;

返回当前分配给字符串的存储空间的大小，以字符大小为单位。存储容量不一定要跟字符串大小（Size）一样，可以相等或更大。

empty 方法

bool empty() const;

如果字符串大小（Size）为 0，则返回 true，否则返回 false。

reserve

void reserve (size\_type n=0);

返回字符串的最小容量。最终实际的容量可能大于或等于 n。

如果n小于capacity（）且大于size（），就调整容量为n，若n小于size（），则调整容量为size（）；

如果 n 比当前容量大，当前函数就会引起字符串重分配内存，使字符串的容量增加到 n （或更大）。但不影响size（）。

shrink\_to\_fit方法（C++11）

void shrink\_to\_fit();

使字符串内部可以随意优化，最后的实际容量大于或等于字符串的有效字符的个数（Size）。用该请求可能会引起一次内存重分配，但不会对 size 产生影响。

### 2.4.修改

赋值运算符=

string& operator= (const string& str)

string& operator= (const charT\* s); // C 型字符串

string& operator= (charT c); // 字符

string& operator= (initializer\_list<charT> il) //初始化列表

string& operator= (string&& str) noexcept; ;// 移动

使用“=”对string对象赋值，等号右侧的操作数可以是string、C风格字符串、还可以是单个字符。原有内容会被清空，当新string的容量大于原容量时会重新分配内存空间。

assign方法

string& assign (const string& str); // 字符串

string& assign (const string& str, size\_type subpos, size\_type sublen); // 子串

string& assign (const charT\* s); // C 型字符串

string& assign (const charT\* s, size\_type n); // 缓存

string& assign (size\_type n, charT c); // 填充

template <class InputIterator> string& assign (InputIterator first, InputIterator last); // 范围

string& assign (initializer\_list<charT> il); // 初始化列表

string& assign (string&& str) noexcept; // 移动

使用assign对string对象赋值，等号右侧的操作数可以是string、C风格字符串、还可以是单个字符。原有内容会被清空，当新string的容量大于原容量时会重新分配内存空间。

追加运算符+=

string& operator+= (const string& str); // 字符串

string& operator+= (const charT\* s); // C 型字符串

string& operator+= (charT c); // 字符

string& operator+= (initializer\_list<charT> il); // 初始化列表

使用“+=”对string对象追加字符或字符串，等号右侧的操作数可以是string、C风格字符串、还可以是单个字符。原有内容会被清空，当新string的容量大于原容量时会重新分配内存空间。

append方法

string& append (const string& str); // 字符串

string& append (const string& str, size\_type subpos, size\_type sublen); // 子串

string& append (const charT\* s); // C 型字符串

string& append (const charT\* s, size\_type n); // 缓存

string& append (size\_type n, charT c); // 填充

string& append (InputIterator first, InputIterator last); // 范围

string& append (initializer\_list<charT> il); // 初始化列表

使用append对string对象追加字符或字符串，等号右侧的操作数可以是string、C风格字符串、还可以是单个字符。原有内容会被清空，当新string的容量大于原容量时会重新分配内存空间。

push\_back方法

void push\_back (charT c);

在字符串的末尾追加一个字符c。使字符串大小（Size）直接加一。当且仅当新的字符串的大小超过当前字符串的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

pop\_back方法

void pop\_back();

删除string对象中的最后一个字符。使字符床大小（Size）直接减一。被删除的字符将被销毁。

insert方法

string& insert (size\_type pos, const string& str); 字符串

string& insert (size\_type pos, const string& str, size\_type subpos, size\_type sublen); // 子串

string& insert (size\_type pos, const charT\* s); // C 型字符串

string& insert (size\_type pos, const charT\* s, size\_type n); // 缓存

string& insert (size\_type pos, size\_type n, charT c); // 填充

void insert (iterator p, size\_type n, charT c); // 填充

iterator insert (iterator p, charT c); // 单字节

void insert (iterator p, InputIterator first, InputIterator last); // 范围

string& insert (const\_iterator p, initializer\_list<charT> il); // 初始化列表

在指定位置处的字符前插入新的字符。直接使容器的大小（Size）增加，增加的值就是所插入元素的个数。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

因为string数组作为它的内部存储结构，将元素插入除了末尾的其它位置会导致容器迁移该位置后的所有字符到新的位置。相比其它类型的顺序容器（如 std::list 或 std::forward\_list C++11），该操作是极其耗性能的。

返回 插入后的string对象的引用。

replace方法

string& replace (size\_type pos, size\_type len, const string& str); //字符串

string& replace (iterator i1, iterator i2, const string& str); //字符串

string& replace (size\_type pos, size\_type len, const string& str, size\_type subpos, size\_type sublen);

// 子串

string& replace (size\_type pos, size\_type len, const charT\* s); // C 型字符串

string& replace (iterator i1, iterator i2, const charT\* s); // C 型字符串

string& replace (size\_type pos, size\_type len, const charT\* s, size\_type n); // 缓存

string& replace (iterator i1, iterator i2, const charT\* s, size\_type n); // 缓存

string& replace (size\_type pos, size\_type len, size\_type n, charT c); // 填充

string& replace (iterator i1, iterator i2, size\_type n, charT c); // 填充

string& replace (iterator i1, iterator i2, InputIterator first, InputIterator last); // 范围

string& replace (const\_iterator i1, const\_iterator i2, initializer\_list<charT> il); // 初始化列表

replace用来替换string对象中的一个或者连续多个字符。返回替换string对象的引用。

erase方法

basic\_string& erase (size\_type pos = 0, size\_type len = npos); // 序列

iterator erase (const\_iterator p); // 字符

iterator erase (const\_iterator first, const\_iterator last); // 范围

erase从string对象中删除一个或多个字符，返回删除的字符的迭代器或者删除后的string对象的引用。

clear方法

void clear();

清空字符串，使string对象的大小（Size）变为 0。不保证是否会发生内存重分配（Reallocation），同样的，string的容量（Capacity）也不保证会发生改变。

swap方法

void swap(basic\_string &str);

使另一个string对象交换内容。容器大小（Size）可以不同。容器中的内存分配器是否也被交换没有明确规定。

### 2.5.字符串的输入

C风格字符串的输入方式大致有“>>”、“\*.getline()”和“\*.get()”3种，对string字符串来说，除了重载了“>>”实现输入外，string头文件还定义了getline函数用以输入string字符串，抛弃了“cin.getline()”和“cin.get()”两种输入方式，原因在于getline外部函数能自动调整目标string的大小，使其能恰好存储输入的字符。>>输入同样会自动调整string对象的大小。

外部getline函数第一个参数必须为输入流对象，第二个参数是待输入的string对象，第3个参数是分界符，getline函数将输入流中的字符存储到string变量中，直到满足下列条件之一：

到达文件尾，此时输入流的eofbit置位有效。

遇到分界字符，默认为换行符’\n’，此时，分解符会从流中删除，但并不会存储到string对象中。

读取的字符达到最大允许值（string类中的常量npos决定了string所能存储的最大字符数，一般都很大，不会对输入产生影响，另一个因素是可用内存数，因此，最大允许值便是从npos和可用内存字节数中选取较小的一个），输入流的failbit置位有效。

### 2.6.字符串操作

c\_str方法

const charT\* c\_str() const noexcept;

c\_str返回C型字符串，末尾带 ‘\0’。string字符串实际上是类对象，其并不以空字符’\0’结尾。C++11以后，data和c\_str成员函数的功能相同。

data方法

const charT\* data() const;

data返回string内部数组的首地址，实质返回C型字符串，但末尾并不添加’\0’。C++11以后，data和c\_str成员函数的功能相同

copy方法

size\_type copy (charT\* s, size\_type len, size\_type pos = 0) const;

从字符串中pos 开始拷贝len个字符序列。返回拷贝的字符数。

find方法

size\_type find (const basic\_string& str, size\_type pos = 0) const noexcept; 字符串

size\_type find (const charT\* s, size\_type pos = 0) const; // C 型字符串

size\_type find (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type find (charT c, size\_type pos = 0) const noexcept; // 字符

find方法返回待查找字符或字符串在原字符串中第一次出现的位置，从0开始计数。

rfind方法

size\_type rfind (const basic\_string& str, size\_type pos = npos) const noexcept; // 字符串

size\_type rfind (const charT\* s, size\_type pos = npos) const; // C 型字符串

size\_type rfind (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type rfind (charT c, size\_type pos = npos) const noexcept; // 字符

rfind方法返回待查找字符或字符串在原字符串中最后一次出现的位置，从0开始计数。

find\_first\_of方法

size\_type find\_first\_of (const basic\_string& str, size\_type pos = 0) const noexcept; // 字符串

size\_type find\_first\_of (const charT\* s, size\_type pos = 0) const; // C 型字符串

size\_type find\_first\_of (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type find\_first\_of (charT c, size\_type pos = 0) const noexcept; // 字符

find\_first\_of返回待查找字符或字符串中的任意元素在原字符串中第一次出现的位置。

find\_last\_of方法

size\_type find\_last\_of (const basic\_string& str, size\_type pos = npos) const noexcept;

size\_type find\_last\_of (const charT\* s, size\_type pos = npos) const; // C 型字符串

size\_type find\_last\_of (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type find\_last\_of (charT c, size\_type pos = npos) const noexcept; // 字符

find\_last\_of返回待查找字符或字符串中的任意元素在原字符串中的最后一次出现的位置。

find\_first\_not\_of方法

size\_type find\_first\_not\_of (const basic\_string& str, size\_type pos = 0) const noexcept; // 字符串

size\_type find\_first\_not\_of (const charT\* s, size\_type pos = 0) const; // C 型字符串

size\_type find\_first\_not\_of (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type find\_first\_not\_of (charT c, size\_type pos = 0) const noexcept; // 字符

find\_first\_not\_of返回不属于待查找字符串的字符在原字符串中第一次出现的位置。

find\_last\_not\_of方法

size\_type find\_last\_not\_of (const basic\_string& str, size\_type pos = npos) const noexcept; // 字符串

size\_type find\_last\_not\_of (const charT\* s, size\_type pos = npos) const; // C型字符串

size\_type find\_last\_not\_of (const charT\* s, size\_type pos, size\_type n) const; // 缓存

size\_type find\_last\_not\_of (charT c, size\_type pos = npos) const noexcept; // 字符

find\_last\_not\_of返回不属于待查找字符串的字符在原字符串中最后一次出现的位置。

substr方法

basic\_string substr (size\_type pos = 0, size\_type len = npos) const;

substr返回原字符串的子串。以string str(“98765432112345678”)为例：

（1）str.substr(); //返回str的全部内容

（2）str.substr(9); //截断前九个字符，即“12345678”

（3）str.substr(5,6);//从下标为5的字符开始往后截取，一共截取6个字符，即“432112”

compare方法

int compare (const basic\_string& str) const noexcept; // 字符串

int compare (size\_type pos, size\_type len, const basic\_string& str) const; // 子串

int compare (size\_type pos, size\_type len, const basic\_string& str, size\_type subpos, size\_type sublen) const; // 子串

int compare (const charT\* s) const; // C 型字符串

int compare (size\_type pos, size\_type len, const charT\* s) const; // C 型字符串

int compare (size\_type pos, size\_type len, const charT\* s, size\_type n) const; // 缓存

string类还提供了成员函数compare()用于字符串比较，支持多参数处理，支持用索引值和长度定位子串来进行比较，返回一个整数（0：相等、>0：大于、<0：小于）来表示比较结果。

### 2.7.获取分配器

get\_allocator方法

allocator\_type get\_allocator() const noexcept;

get\_allocator() 函数返回与string关联的分配器成员类型。

allocator\_type 是给stroing内部使用的内存分配器的类型，如果使用默认的内存分配器allocator\_type 的值是不确定的。

## **3.常量**

npos

static const size\_type npos = -1;

成员类型 size\_type 的最大值，,用来表示不存在的位置。许多容器都提供这个东西

## 4.非成员函数模板的特例化

连接运算符+

string operator+ (const string& lhs, const string& rhs);

string operator+ (string&& lhs, string&& rhs);

string operator+ (string&& lhs, const string& rhs);

string operator+ (const string& lhs, string&& rhs);

string operator+ (const string& lhs, const charT\* rhs); // C 型字符串

string operator+ (string&& lhs, const charT\* rhs); // C 型字符串

string operator+ (const charT\* lhs, const string& rhs); // C 型字符串

string operator+ (const charT\* lhs, string&& rhs); // C 型字符串

string operator+ (const string<charT,traits,Alloc>& lhs, charT rhs); // 字符

string operator+ (string<charT,traits,Alloc>&& lhs, charT rhs); // 字符

string operator+ (charT lhs, const string<charT,traits,Alloc>& rhs); // 字符

string operator+ (charT lhs, string<charT,traits,Alloc>&& rhs); // 字符

连接运算符+用来连接两个字符串或一个字符串和一个字符。

关系运算符>,>=,<,<=,==,!=

string对象支持常见的比较操作符（>,>=,<,<=,==,!=），也支持string与C风格字符串的比较。根据字符的字典顺序（字典排序靠前的字符小）从前往后逐一比较，遇到不等的字符就按该位置上的这两个字符的比较结果确定两个字符串的大小。返回一个整数（0：相等、>0：大于、<0：小于）来表示比较结果。

swap函数

void swap (string& x, string& y);

交换x与y的内容

输出重定向<<

输出string字符串

输入重定向>>

输入string字符串

getline函数

istream & getline (stream& is, string & str, delim);

istream & getline (istream&& is, string& str, delim);

istream & getline (istream& is, string& str);

istream & getline (istream&& is, string& str);

从流中读取一行字符。并自动调整string对象的容量。如果读到末尾，则置failbit位

## 5.相关全局函数（c++11）

stoi，stol，stoll，stoul，stoull函数

数据类型sto\* (const string& str, size\_t\* idx = 0, int base = 10);

数据类型sto\*(const wstring& str, size\_t\* idx = 0, int base = 10);

将字符串数字转为对应的数值。字符串要符合对应范围否则会报错，而标准C的ato\*则不会报错。

参数str为要转换的字符串

参数idx为数开始的位置

参数base为str数的进制

stod，stof，stold函数

数据类型sto\* (const string& str, size\_t\* idx = 0);

​数据类型sto\* (const wstring& str, size\_t\* idx = 0);

将字符串数字转为对应的数值。字符串要符合对应范围否则会报错，而标准C的ato\*则不会报错。

to\_string,to\_wstring函数(要首先声明)

string to\_\*string (int val);

string to\_\*string (long val);

string to\_\*string (long long val);

string to\_\*string (unsigned int val);

string to\_\*string (unsigned long val);

string to\_\*string (unsigned long long val);

string to\_\*string (float val);

string to\_\*string (double val);

string to\_\*string (long double val);

将数值转为对应的字符串。

hash支持

template<> struct hash<std::\*string>

## **6.对字符串进行遍历**

（1）用for循环

（2）用迭代器 string：：iterator 迭代器；

for（it =字符串变量.begin（）;迭代器！=字符串标量.end（））

{visit（迭代器）}

获取字符串的子串

# C++11序列容器库

## 1.序列容器之array

### 1.1.array慨念

#### 1.1.1定义

数组（Array）是一个固定大小（Fixed-size）的顺序容器（Sequence container）。容器容纳了具体（Specific）数目的元素，这些元素逐个排列在一个严格的线性序列中。

#### 1.1.2.容性特性

顺序序列：顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。

连续存储空间：所有元素存储在一个连续的内存位置处，使得可以以常量时间随机访问元素。指向元素的指针可以通过偏移访问其它元素。

固定大小：容器使用隐式（Implicit）构造函数（Constructor）及析构函数（Destructor）来静态地分配所需的存储空间。它的大小是一个编译时（Compile-time）确定的常量。除了自身所需外，不会消耗额外的内存或时间。

#### 1.1.3.详细说明（ISO/IEC 14882:2011）

在 array 容器内部，不会保存除了元素的任意其它数据（甚至没保存容器的大小，这个是一个模板参数，在编译时就已固定）。该容器同C中的括号语义（[]）定义的普通数组一样高效。array 仅仅是为普通数组添加了一些成员或全局函数，这使得数组能够被当成标准容器来使用。

区别于其它标准容器，array 拥有一个固定的大小，且不需要通过内存分配器（Allocator）管理元素的储存空间。它是一个封装了固定数量元素的数组的聚合类型（Aggregate）。因此，array 不能被动态地扩展或压缩。

大小为 0 的 array 容器是允许的，但是这样的数组里的元素是不能被解引用的（Dereferenced）（因为没有元素可以被解引用，相关成员函数有 front、back、及 data）。

区别于其它标准容器，交换（Swap）两个 array 容器是一个线性操作（即时间复杂度为 O(n)），涉及分别交换范围内的所有元素，这通常被认为是一个低效的操作。

array 容器唯一具有的另一个特征是：可以把 array 容器当成一个多元组（Tuple）对象。在头文件 <array> 中，通过重载全局函数 get 使得通过 get 访问 array 中的元素时，array 容器就像一个多元组。

#### 1.1.4.定义对象

#include<array>

template < class T, size\_t N > class array;

array<数据类型，元素个数>数组名；

array<数据类型，元素个数>数组名{{初始化列表}}；

array<数据类型，元素个数>数组名={{初始化列表}}；

array<数据类型，元素个数>\*指针名=new array<数据类型，元素个数>{{初始化列表}}；

参数T：容器所包含的元素的类型。在类模板内部，使用其别名为 value\_type 的成员类型。

参数N：数组的大小，以元素的个数为单位。

由于array没有重载“=”，数组和vector不同，一个数组不能用另一个数组初始化，也不能将一个数组赋值给另一个数组；更不能向数组的某成员用“=”赋值。

### 1.2.常用public成员函数

#### 1.2.1.元素访问

operator[]

reference operator[] (size\_type n);

const\_reference operator[] (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。array::at 成员函数与该操作符函数有类似的行为，只是 array::at 会做边界检测，当请求的位置超出有效元素的范围（Size）时将抛出 std::out\_of\_range 异常。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

参数n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

参数size\_type ：一个无符号整数类型。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

at方法

reference at (size\_type n);

const\_reference at (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。该函数会自动地做边界检测，当请求的位置超出容器中有效元素的范围时将抛出 std::out\_of\_range 异常。该点正好与 array::operator[] 的形成对比，后者不做边界检测。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

size\_type ：一个无符号整数类型。

front方法

reference front();

const\_reference front() const;

返回对当前容器中第一个元素的引用（Reference）。区别于 array::begin 函数，array::begin 返回的是指向同一个元素的迭代器（Iterator），而当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

back方法

reference back();

const\_reference back() const;

返回对当前容器中最后一个元素的引用（Reference）。区别于 array::end 函数，array::end 返回的是最后一个元素的下一个元素的迭代器（Iterator），当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

data方法

T \* data（）

返回指向容器内部用于存储自身元素的数组的直接指针。因为容器中的元素保证是存储在连续的内存空间且存储顺序与容器中的顺序是一致的，所以可以对返回的指针直接做偏移操作以获得数组中的任意元素。如果容器对象是 const 限定的，函数返回指向 const value\_type 类型的指针，否则返回指向 value\_type 类型的指针。

#### 1.2.2.迭代器

array支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 array::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 array 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

##### 1.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 array 对象中第一个元素的迭代器。如果 array 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 array::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 array::front 返回的是对第一个元素的引用（Reference）。array::begin 和 array::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 array 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 array 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。array::begin 和 array::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 array::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 array 对象中最后一个元素的逆序迭代器。如果 array 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。array::rbegin 返回值指向的元素刚好是 array::end 返回值指向的元素的前一个元素。区别于 array::back，当前函数返回的是指向最后一个元素的逆序随机访问迭代器（Reverse random access iterator），而 array::back 返回指向最后一个元素的引用（Reference）。array::rbegin 和 array::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 array 对象中第一个元素之前（ Preceding the first）的元素的逆序迭代器，如果 array 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。array::rbegin 和 array::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

##### 1.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 array 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 array::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 array 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。array::cbegin 和 array::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 array 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 array::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 array 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 array::cbegin 相同。array::cbegin 和 array::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 array 对象中最后一个元素的常逆序迭代器。array::crbegin 和 array::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 array 对象中第一个元素之前（ Preceding the first）的元素的常逆序迭代器。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。array::crbegin 和 array::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

#### 1.2.3.容量

size方法

constexpr size\_type size() noexcept;

返回当前容器中元素的个数。数组对象的大小总是等于用来实例化 array 类模板的第二个参数（N）。区别于语言操作符函数 std::sizeof ，std::sizeof 返回的是以字节为单位的大小，而当前函数返回的是以元素个数为单位的大小。

max\_size方法

constexpr size\_type max\_size() noexcept;

返回当前容器能够纳的最大元素个数。跟 array 对象的当前元素个数一样，array 对象的最大元数个数总是等于用来实例化 array 类模板的第二个参数（N）。

empty方法

constexpr bool empty() noexcept;

返回当前容器是否为空。

#### 1.2.4.修改

fill方法

void fill (const value\_type& val);

array 类模板的公共成员函数。将容器中所有元素的值设为 val 。

swap方法

void swap (array& x) noexcept

使另一个同类型（包括相同类型的大小）的数组对象 x 与当前数组交换内容。在调用该函数之后，当前容器中的内容是那些原来在 x 中的，而 x 中的内容则是之前在当前容器中的。跟其它容器的 swap 成员函数不同，array::swap 需要消耗线性时间，即时间复杂度为 O(n)，当发生异常时，该操作可能会退出，且该操作不会导致迭代器关联到另一个容器。

### 1.3.非成员函数特例化

关系运算符>,>=,<,<=,==,!=

array对象支持常见的比较操作符（>,>=,<,<=,==,!=）。根据元素类型从前往后逐一比较，遇到不等的元素就按该位置上的这两个字符的比较结果确定两个元素的大小。返回一个整数（0：相等、>0：大于、<0：小于）来表示比较结果。

swap函数

void swap (T& x, T& y);

交换x与y的内容

## 2.序列容器之vector

### 2.1.vector的概念

#### 2.1.1.定义

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。

#### 2.1.2.容器特性

顺序序列：顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。

动态数组：支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算述进行该操作。操供了在序列末尾相对快速地添加/删除元素的操作。

能够感知内存的配器（Allocator-aware）：容器使用一个内存分配器对象来动态地处理它的存储需求。

#### 2.1.3.详细说明-ISO/IEC 14882:2011

在向量中，所有元素都是连续存储的。也就是说，不仅可以通过迭代器（Iterators）访问各个元素，也可以通过指向元素的指针加上偏移来访问。还意味着，当向任意函数传递向量的一个元素的指针时，这个指针可以直接被认为指向了一个数组中的某个元素。

向量内部的存储调整是自动处理的，按需扩展或压缩。通常，相比静态数组（Static arrays），向量将会占用更多的存储空间，因为额外的内存将被未来增长的部分所使用。就因为这点，当插入元素时，向量不需要太频繁地重分配（Reallocate）内存。当前最大容量可以通过函数 capacity() 查询。额外的内存可以通过调用 shrink\_to\_fit() 函数返还给操作系统。

当增加向量对象中的序列的长度时，若超出当前存储容量上限，就会发生内存重分配（Reallocation），即内部将会重新分配一个数组，然后按顺序逐个拷贝元素。其它的插入及删除操作将会修改序列中部分元素的内存地址。在上述所有情况下，指向序列中被修改部分的迭代器或引用将会失效。当未发生内存重分配，仅指向插入或删除点之前元素的迭代器或引用才会保持有效性。标准库可以执行不同的增长策略来平衡内存的使用量与重分配所耗的性能。但不管哪种情况下，重分配内存的大小必须以指数方式增长，只有这样，才能将在向量末尾逐个插入元素所需的时间复杂度整体分摊（Amortized）为一个恒定值。内存重分配就性能而言是一个高代价操作。如果在使用向量前知道元素的数量，可以通过 reserve()或者定义时使用[MAXSIZE]消除内存重分配。向量支持在序列末尾恒定耗时的插入及删除元素。而在向量的中间插入或删除元素则需要线性的时间。在只涉及向序列起始或未尾插入及删除元素操作时，std::deque​ 容器的性能将会高出很多。当涉及向序列中的任意位置进行插入及删除操作时，std::list 容器的性能将会高出很多。

#### 2.1.4.定义对象

##### 2.1.4.1.模板

#include<vector>

template < class T, class Alloc = allocator<T> > class vector;

vector<元素类型，内存分配器>向量名（构造参数）;

参数T：存储在容器中的元素的数据类型。必须保证在执行移动操作时不会抛出异常，因为在重分配时，实现者（标准库内部）可以进行优化使仅仅移动元素而不是拷贝所有。在类模板内部，使用其别名为 value\_type 的成员类型。

参数Alloc：容器内部用来管理内存分配及释放的内存分配器的类型。这个参数是可选的，它的默认值是 std::allocator<T>，这个是一个最简单的非值依赖的（Value-independent）内存分配器。在类模板内部，使用其别名为 allocator\_type 的成员类型。

##### 2.1.4.2.构造函数：

explicit vector (const allocator\_type& alloc = allocator\_type());//default (1)

explicit vector (size\_type n); //fill (2)

vector (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type());

template <class InputIterator>vector (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type());//range (3)

vector (const vector& x); //copy (4)

vector (const vector& x, const allocator\_type& alloc); //copy (4)

vector (vector&& x); //move (5)

vector (vector&& x, const allocator\_type& alloc); //move (5)

vector (initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type());//initializer list (6)

(1) 空容器构造函数（Default）：构造一个空的容器，元素数为0。

(2) 填充（Fill）构造函数：构造一个带 n 个元素的容器，每个元素都是一份 val（如果有提供，否则调用元素类型的默认构造函数创建一个临时的）的拷贝。

(3) 范围（Range）构造函数：用范围 [first,last) 的内容构造容器，容器中的元素按顺序对应于范围中的各个元素。

(4) 拷贝（Copy）构造函数（及带内存分配器拷贝）：

按顺序拷贝另一个同类型的容器对象的内容来构造新的容器。

(5) 移动（Move）构造函数（及带内存分配器移动）：以Move语义获得另外一个同类型容器的元素来构造容器。如果未提供 alloc 参数，则从另一个容器的内存分配器移动构造获得。如果提供了 alloc 参数且与被移动容器自带的内存分配器不一样，元素将被移动，否则无任一元素将被构造（而是直接传递所有权）。被移动后的 x 的状态是未定义的。

(6) 初始化列表（Initializer list）构造函数：用初始化列表的内容按序构造容器。

容器内部保存了 alloc 的一份拷贝，可以用它分配或释放元素的存储空间，且可以用它来构造或销毁已分配存储空间的元素对象（由 alloc 类型的 std::allocator\_traits 指定）。对于拷贝构造函数(4)，如果 alloc 参数未提供，则内存分配器由通过调用std::allocator\_traits<allocator\_type>::select\_on\_copy\_construction(x) 获得。对于移动构造函数(5)，如果 alloc 参数未提供，则直接获取 x 的内存分配器。

所有被拷贝、移动或其它类似操作的元素都是通过用合适的参数调用 std::allocator\_traits::construct 来构造的。

alloc内存分配器对象。容器内部将会保存且使用该内存分配器的拷贝。可以使用 vector::get\_allocator 函数获得容器内部的内存分配器的一份拷贝。

成员类型 allocator\_type 是给容器内部使用的内存分配器的类型，作为 vector 实例化时第一个模板参数的别名而存在。如果使用默认的内存分配器，allocator\_type 的值是不确定的。

n初始容器大小。

成员类型 size\_type 是一个无符号整数类型。

val用来填充容器的值。容器中 n 个元素中的每一个都将被初始化为该值的拷贝。

成员类型 value\_type 是该容器的元素的类型，作为 vector 实例化时第一个模板参数的别名而存在。

first，last分别指向一个序列中初始及末尾位置的输入迭代器。这个范围即 [first,last) ，包括 first 到 last 间的所有元素，包括 first 指向的元素，但不包括 last 指向的元素。

x另一个同类型的 vector 对象（有相同的模板参数 T 及 Alloc）。

il一个初始化列表对象。编译器将根据初始化列表声明体自动创建该对象。

##### 2.1.4.3.声明一维向量

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)。

##### 2.1.4.4.声明多维向量

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.

### 2.2.常用public成员函数

#### 2.2.1.元素访问

operator[]

reference operator[] (size\_type n);

const\_reference operator[] (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。vector::at 成员函数与该操作符函数有类似的行为，只是 vector::at 会做边界检测，当请求的位置超出有效元素的范围（Size）时将抛出 std::out\_of\_range 异常。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

参数n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

参数size\_type ：一个无符号整数类型。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

at方法

reference at (size\_type n);

const\_reference at (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。该函数会自动地做边界检测，当请求的位置超出容器中有效元素的范围时将抛出 std::out\_of\_range 异常。该点正好与 vector::operator[] 的形成对比，后者不做边界检测。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

size\_type ：一个无符号整数类型。

front方法

reference front();

const\_reference front() const;

返回对当前容器中第一个元素的引用（Reference）。区别于 vector::begin 函数，vector::begin 返回的是指向同一个元素的迭代器（Iterator），而当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

back方法

reference back();

const\_reference back() const;

返回对当前容器中最后一个元素的引用（Reference）。区别于 vector::end 函数，vector::end 返回的是最后一个元素的下一个元素的迭代器（Iterator），当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

data方法

T \* data（）

返回指向容器内部用于存储自身元素的数组的直接指针。因为容器中的元素保证是存储在连续的内存空间且存储顺序与容器中的顺序是一致的，所以可以对返回的指针直接做偏移操作以获得数组中的任意元素。如果容器对象是 const 限定的，函数返回指向 const value\_type 类型的指针，否则返回指向 value\_type 类型的指针。

#### 2.2.2.迭代器

vector支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 vector::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 vector 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

##### 2.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 vector 对象中第一个元素的迭代器。如果 vector 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 vector::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 vector::front 返回的是对第一个元素的引用（Reference）。vector::begin 和 vector::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 vector 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 vector 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。vector::begin 和 vector::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 vector::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 vector 对象中最后一个元素的逆序迭代器。如果 vector 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。vector::rbegin 返回值指向的元素刚好是 vector::end 返回值指向的元素的前一个元素。区别于 vector::back，当前函数返回的是指向最后一个元素的逆序随机访问迭代器（Reverse random access iterator），而 vector::back 返回指向最后一个元素的引用（Reference）。vector::rbegin 和 vector::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 vector 对象中第一个元素之前（ Preceding the first）的元素的逆序迭代器，如果 vector 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。vector::rbegin 和 vector::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

##### 2.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 vector 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 vector::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 vector 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。vector::cbegin 和 vector::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 vector 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 vector::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 vector 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 vector::cbegin 相同。vector::cbegin 和 vector::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 vector 对象中最后一个元素的常逆序迭代器。vector::crbegin 和 vector::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 vector 对象中第一个元素之前（ Preceding the first）的元素的常逆序迭代器。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。vector::crbegin 和 vector::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

#### 2.2.3.容量

size 方法

size\_type size() const noexcept;

返回当前容器的元素个数。等同于 std::distance(vector::begin(), vector::end())，这是当前容器的有效元素个数，它不需要与当前容器的容量（Capacity）相等。

max\_size

size\_type max\_size() const noexcept;

返回 vector 能容纳的最大元素个数。是一个当前容器能够达到的受限于操作系统或编译器的最大可能大小。但实事上容器并不保证能够达到这个大小：在达到这个上限之前，已经出现内存分配失败。

resize方法

void resize (size\_type n);

void resize (size\_type n, value\_type val);

调整容器的大小使其能够存放 n 个元素。返回新vector的大小。

如果 n 比当前容器大小（Size）小，容器中的内容将被减少到只剩前 n 个元素，超出的部份将被移除（或销毁）。但容量不变。

如果 n 比当前容器大小（Size）大，将通过在容器尾部插入足够的元素使内容增加到 n 个元素。如果提供了 val ，新的元素将被初始化为 val 的拷贝，否则将被值初始化（Value-initialized）。

如果 n 比当前容器容量（Capacity）大，将会对已分配内存空间自动进行一次内存重分配。

capacity 方法

size\_type capacity() const noexcept;

返回当前分配给容器的存储空间的大小，以元素大小为单位。存储容量不一定要跟容器大小（Size）一样，可以相等或更大。

empty 方法

bool empty() const;

如果容器大小（Size）为 0，则返回 true，否则返回 false。

reserve

void reserve (size\_type n);

返回容器的最小容量。最终实际的容量可能大于或等于 n。

如果 n 比当前容量大，当前函数就会引起容器重分配内存，使容器的容量增加到 n （或更大）。在其它所有情况下，该函数不会引起内存重分配，容器的存储容量不会受到影响。

shrink\_to\_fit方法（C++11）

void shrink\_to\_fit();

请求容器减小容量使其足够用来存放所有有效元素。使容器内部可以随意优化，最后的实际容量大于或等于容器的有效元素的个数（Size）。用该请求可能会引起一次内存重分配，但不会对 size 产生影响。

#### 2.2.4.修改

赋值运算符=

vector& operator= (const vector& x);//copy

vector& operator= (vector&& x);//move

vector& operator= (initializer\_list<value\_type> il); //initializer

赋值新的内容到容器， 以替换当前内容，并根据情况修改容器的大小（Size）。进行该调用前的所有元素或者被重新赋值，或者被销毁。各个声明的具体含义如下：

(1) 拷贝（Copy）赋值：将 x 中的所有元素拷贝到当前容器，x 保留之前的内容。

(2) 移动（Move）赋值：将 x 中的所有元素拷贝到当前容器，x 进入未定义状态。

(3) 用初始化列表（Initializer list）赋值：拷贝列表中的元素到当前容器。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

容器将保留当前的内存分配器（Allocator），除非内存分配器特征表明 x 的的内存分配器需要传递进当前容器。

assign方法

template <class InputIterator>void assign( input\_iterator start, input\_iterator end ); //range

void assign( size\_type num, const type &val );//fill

void assign (initializer\_list<value\_type> il); //initializer list

assign() 函数要么将区间[start, end)的元素赋到当前vector,或者赋num个值为val的元素到vector中或者用初始化列表的内容按序赋值给容器。这个函数将会清除掉为vector赋值以前的内容。该函数被调用前的所有元素将被销毁，然后被新构建的元素所替换（而不会发生元素的赋值）。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

当发生内存重分配时（Reallocation），容器内部的内存分配器（std::allocator\_traits）被用来分配或释放元素的存储空间，同时被用来销毁所有已经存在的元素对象，然后构建新的元素对象。

push\_back方法

void push\_back (value\_type& val)

在向量末尾添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

pop\_back方法

void pop\_back();

删除向量对象中的最后一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

insert方法

iterator insert (const\_iterator position, const value\_type& val);

iterator insert (const\_iterator position, value\_type&& val);

iterator insert (const\_iterator position, size\_type n, const value\_type& val);

iterator insert (const\_iterator position, InputIterator first, InputIterator last);

iterator insert (const\_iterator position, initializer\_list<value\_type> il);

在指定位置处的元素前插入一个新的元素。直接使容器的大小（Size）增加，增加的值就是所插入元素的个数。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

因为向量用数组作为它的内部存储结构，将元素插入除了末尾的其它位置会导致容器迁移该位置后的所有元素到新的位置。相比其它类型的顺序容器（如 std::list 或 std::forward\_list C++11），该操作是极其耗性能的。

返回指向第一个新插入元素的迭代器，如果无元素插入（如：n == 0、first == last、il 为容），则返回 position。

erase方法

iterator erase (const\_iterator position);

iterator erase (const\_iterator first, const\_iterator last);

从当前向量中移除单个（position 所指向的）或一个范围（[first,last)）的元素。直接减小容器大小（Size），减小的值等于所移除元素的个数。被移除的元素将被销毁（Destroy）。

因为向量用数组作为它的内部存储结构，移除除了末尾的其它位置的元素会导致容器迁移该被移除段后的所有元素到新的位置。相比其它类型的顺序容器（如 std::list 或 std::forward\_list），该操作是极其耗性能的。

返回指向最后一个被移除的元素之后的元素的迭代器。当移除容器中的最后一个元素时，该返回值为当前容器的末尾（End，最后一个元素之后的元素，是一个“理论上”的元素）。

swap方法

void swap (vector& x);

使另一个同类型的向量对象 x 与当前向量交换内容。容器大小（Size）可以不同。容器中的内存分配器是否也被交换没有明确规定。

clear方法

void clear() noexcept

移除当前容器中的所有元素，使容器的大小（Size）变为 0。不保证是否会发生内存重分配（Reallocation），同样的，向量的容量（Capacity）也不保证会发生改变。

emplace方法（C++11）

template <class... Args> iterator emplace (const\_iterator position, Args&&... args)

在指定位置处的元素前插入一个新的元素。新的元素是以 args 作为其构造函数的参数而构造的。直接使容器的大小（Size）增加 1。

当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

因为向量用数组作为它的内部存储结构，将元素插入除了末尾的其它位置会导致容器迁移该位置后的所有元素到新的位置。相比其它类型的顺序容器（如 std::list 或 std::forward\_list C++11），该操作是极其耗性能的。vector::emplace\_back 成员函数直接在末尾插入元素。

返回一个指向新放置（Emplaced）的元素的迭代器

emplace\_back方法（C++11）

template <class... Args> void emplace\_back (Args&&... args);

在向量末尾添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

#### 2.2.5.获取分配器

get\_allocator方法

allocator\_type get\_allocator() const noexcept;

get\_allocator() 函数返回与vector关联的分配器成员类型。

allocator\_type 是给容器内部使用的内存分配器的类型，如果使用默认的内存分配器allocator\_type 的值是不确定的。

## 3.序列容器之list

### 3.1.list概念

#### 3.1.1.定义

列表（List）是一个允许在序列中任何一处位置以常量耗时插入或删除元素且可以双向迭代的顺序容器（Sequence container）。

#### 3.1.2.容器特性

顺序序列：顺序容器中的元素按照严格的线性顺序排序。可通过元素在序列中的位置访问对应的元素。

双向链表：容器中的每个元素保存了定位前一个元素及后一个元素的信息，允许在任何一处位置以常量耗时进行插入或删除操作，但不能进行直接随机访问（Direct random access）。

能够感知内存分配器的（Allocator-aware）：容器使用一个内存分配器对象来动态地处理它的存储需求。

#### 3.1.3详细说明- ISO/IEC 14882:2011

标准模板库（Standard Template Library，简称STL）中的列表容器采用的是双向链表（Doubly-linked lists）数据结构。双向链表可以保存以不同且无关的储存位置容纳的元素。元素间的顺序是通过各个元素中指向前一个元素的链接及指向后一个元素的链接等联系维护的。

C++11 该特点跟 std:: forward\_list 容器的有点相似：主要的区别就是 std:: forward\_list 是一个单向链表，因此它只能被正向迭代（ Iterated forward），以换取更小更高效的特点。

增加或移动列表中的元素不会使指向它们的指针、引用、迭代器失效，只有当对应的元素被销毁时才会如此。

你可能会好奇，为什么STL同时提供列表（std::list）跟向量（std::vector）及其它等多种容器，原因是它们的底层实现方式是不一样的，每一种实现方式都有各自的消耗（空间或时间）及优点。

相比较于其它的基本标准顺序容器（数组 std::array、向量 std::vector、双端队列 std::deque 等），列表通常在容器中已获得迭代器的任意位置处插入、获得（Extracting，提取）、移动元素等操作中表现出更出色的性能，对那些密集使用上述操作的算法，使用列表同样能提升性能，比如排序算法。

双向链表（std::list）及正向链表（std:: forward\_list C++11）相比于其它顺序容器的主要缺点是它们不能够通过元素在容器中的位置直接访问（Direct access）元素。举个例子：如果想访问一个列表中的第六个元素，那么就必须从一个已知位置（比如开头或末尾）处开始迭代，这会消耗与两个位置之间距间相关的线性时间。而且它们还为保存各个元素间的链接信息消耗更多额外的内存（这点对由小尺寸元素组成的大列表尤为明显）。

快速的在容器中间部份插入元素是列表的最大优势。可以使用成员函数 insert 完成这一操作：需要指定插入的元素及指向插入位置的迭代器（新的元素将会插入到当前所指向元素的前面）。从 C++11 开始，你也可以使用 emplace 来完成这一操作，具体区别请查看两个函数的详细介绍。

可以使用 sort 成员函数对列表进行排序，该操作时间复杂度保证为 O(nlogn)。

注意，标准库中提供的 std::sort 排序函数需要容器支持随机访问迭代器，而 list 模板类并未提供该支持（直接原因是 list 提供的迭代器不支持 operator- 操作符函数），所以 std::sort 不支持对 list 的排序。

可以使用 reverse 成员函数来反转列表。标准C++库的算法库中存在一个相似的函数：std::reverse，该函数同样用于反转容器中的元素。当前成员函数（list::reverse）与 std::reverse 的区别就是调用 list::reverse 后不会影响其它正在使用的迭代器所指向的值。

#### 3.1.4.定义对象

##### 3.1.4.1.模板

#include<list>

template < class T, class Alloc = allocator<T> > class list;

T存储在容器中的元素的数据类型。在类模板内部，使用其别名为 value\_type 的成员类型。

Alloc容器内部用来管理内存分配及释放的内存分配器的类型。这个参数是可选的，它的默认值是 std::allocator<T>，这个是一个最简单的非值依赖的（Value-independent）内存分配器。 在类模板内部，使用其别名为 allocator\_type 的成员类型。

##### 3.1.4.2.构造函数

explicit list (const allocator\_type& alloc = allocator\_type());// default (1)

explicit list (size\_type n); // fill (2)

list (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type());// fill (2)

template <class InputIterator>

list (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type());// range (3)

list (const list& x); // copy (4)

list (const list& x, const allocator\_type& alloc); // copy (4)

list (list&& x); // move (5)

list (list&& x, const allocator\_type& alloc); // move (5)

list (initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type());// initializer list (6)

(1) 空容器构造函数（Default）

构造一个空的容器，元素数为 0。

(2) 填充（Fill）构造函数

构造一个带 n 个元素的容器，每个元素都是一份 val（如果有提供，否则调用元素类型的默认构造函数创建一个临时的）的拷贝。

(3) 范围（Range）构造函数

用范围 [first,last) 的内容构造容器，容器中的元素按顺序对应于范围中的各个元素。

(4) 拷贝（Copy）构造函数（及带内存分配器拷贝）

按顺序拷贝另一个同类型的容器对象的内容来构造新的容器。

(5) 移动（Move）构造函数（及带内存分配器移动）

以Move语义获得另外一个同类型容器的元素来构造容器。如果未提供 alloc 参数，则从另一个容器的内存分配器移动构造获得。如果提供了 alloc 参数且与被移动容器自带的内存分配器不一样，元素将被移动，否则无任一元素将被构造（而是直接传递所有权）。被移动后的 x 的状态是未定义的。

(6) 初始化列表（Initializer list）构造函数

用初始化列表的内容按序构造容器。

容器内部保存了 alloc 的一份拷贝，可以用它分配或释放元素的存储空间，且可以用它来构造或销毁已分配存储空间的元素对象（由 alloc 类型的 std::allocator\_traits 指定）。对于拷贝构造函数(4)，如果 alloc 参数未提供，则内存分配器由通过调用std::allocator\_traits<allocator\_type>::select\_on\_copy\_construction(x) 获得。对于移动构造函数(5)，如果 alloc 参数未提供，则直接获取 x 的内存分配器。

所有被拷贝、移动或其它类似操作的元素都是通过用合适的参数调用 std::allocator\_traits::construct 来构造的。

参数及返回值

ClassFoo 编辑于 07-28版本0评论0我来编辑

alloc内存分配器对象。容器内部将会保存且使用该内存分配器的拷贝。可以使用 list::get\_allocator 函数获得容器内部的内存分配器的一份拷贝。

成员类型 allocator\_type 是给容器内部使用的内存分配器的类型，作为 list 实例化时第一个模板参数的别名而存在。如果使用默认的内存分配器，allocator\_type 的值是不确定的。

n初始容器大小。

成员类型 size\_type 是一个无符号整数类型。

val用来填充容器的值。容器中 n 个元素中的每一个都将被初始化为该值的拷贝。

成员类型 value\_type 是该容器的元素的类型，作为 list 实例化时第一个模板参数的别名而存在。

first，last分别指向一个序列中初始及末尾位置的输入迭代器。这个范围即 [first,last) ，包括 first 到 last 间的所有元素，包括 first 指向的元素，但不包括 last 指向的元素。

x另一个同类型的 list 对象（有相同的模板参数 T 及 Alloc）。

il一个初始化列表对象。编译器将根据初始化列表声明体自动创建该对象。

### 3.2.常用public成员函数

#### 3.2.1.元素访问

front方法

reference front();

const\_reference front() const;

返回对当前容器中第一个元素的引用（Reference）。区别于 list::begin 函数，list::begin 返回的是指向同一个元素的迭代器（Iterator），而当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

back方法

reference back();

const\_reference back() const;

返回对当前容器中最后一个元素的引用（Reference）。区别于 list::end 函数，list::end 返回的是最后一个元素的下一个元素的迭代器（Iterator），当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

#### 3.2.2.迭代器

list支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 list::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 list 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

##### 3.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 list 对象中第一个元素的迭代器。如果 list 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 list::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 list::front 返回的是对第一个元素的引用（Reference）。list::begin 和 list::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 list 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 list 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。list::begin 和 list::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 list::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 list 对象中最后一个元素的逆序迭代器。如果 list 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。list::rbegin 返回值指向的元素刚好是 list::end 返回值指向的元素的前一个元素。区别于 list::back，当前函数返回的是指向最后一个元素的逆序随机访问迭代器（Reverse random access iterator），而 list::back 返回指向最后一个元素的引用（Reference）。list::rbegin 和 list::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 list 对象中第一个元素之前（ Preceding the first）的元素的逆序迭代器，如果 list 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。list::rbegin 和 list::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

##### 3.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 list 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 list::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 list 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。list::cbegin 和 list::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 list 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 list::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 list 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 list::cbegin 相同。list::cbegin 和 list::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 list 对象中最后一个元素的常逆序迭代器。list::crbegin 和 list::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 list 对象中第一个元素之前（ Preceding the first）的元素的常逆序迭代器。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。list::crbegin 和 list::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

#### 3.2.3.容量

size 方法

size\_type size() const noexcept;

返回当前容器的元素个数。等同于 std::distance(list::begin(), list:end())，这是当前容器的有效元素个数，它不需要与当前容器的容量（Capacity）相等。

max\_size

size\_type max\_size() const noexcept;

返回list 能容纳的最大元素个数。是一个当前容器能够达到的受限于操作系统或编译器的最大可能大小。但实事上容器并不保证能够达到这个大小：在达到这个上限之前，已经出现内存分配失败。

empty方法

bool empty() const;

如果容器大小（Size）为 0，则返回 true，否则返回 false。

resize方法

void resize (size\_type n);

void resize (size\_type n, value\_type val);

调整容器的大小使其能够存放 n 个元素。

如果 n 比当前容器大小（Size）小，内容减少到容器前的 n 个元素，超出的部份将被移除（或销毁）。

如果 n 比当前容器大小（Size）大，将通到在容器尾部插入足够的元素使内容增加到 n 个元素。如果提供了 val ，新的元素将被初始化为 val 的拷贝，否则将被值初始化（Value-initialized）。

#### 3.2.4.修改

赋值运算符=

list& operator= (const list& x);//copy

list& operator= (list&& x);//move

list& operator= (initializer\_list<value\_type> il); //initializer

赋值新的内容到容器， 以替换当前内容，并根据情况修改容器的大小（Size）。进行该调用前的所有元素或者被重新赋值，或者被销毁。各个声明的具体含义如下：

(1) 拷贝（Copy）赋值：将 x 中的所有元素拷贝到当前容器，x 保留之前的内容。

(2) 移动（Move）赋值：将 x 中的所有元素拷贝到当前容器，x 进入未定义状态。

(3) 用初始化列表（Initializer list）赋值：拷贝列表中的元素到当前容器。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

容器将保留当前的内存分配器（Allocator），除非内存分配器特征表明 x 的的内存分配器需要传递进当前容器。

assign方法

template <class InputIterator>void assign (InputIterator first, InputIterator last); // range (1)

void assign (size\_type n, const value\_type& val); // fill (2)

void assign (initializer\_list<value\_type> il); // initializer list (3)

assign() 函数要么将区间[start, end)的元素赋到当前list,或者赋num个值为val的元素到list中或者用初始化列表的内容按序赋值给容器。这个函数将会清除掉为list赋值以前的内容。该函数被调用前的所有元素将被销毁，然后被新构建的元素所替换（而不会发生元素的赋值）。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

当发生内存重分配时（Reallocation），容器内部的内存分配器（std::allocator\_traits）被用来分配或释放元素的存储空间，同时被用来销毁所有已经存在的元素对象，然后构建新的元素对象。

push\_back方法

void push\_back (value\_type& val)

void push\_back (value\_type&& val);

在列表末尾添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。这里存在一个相似的成员函数：list::emplace\_back，该函数向容器插入用提供的参数构造的元素。

pop\_back方法

void pop\_back();

删除列表对象中的最后一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

push\_front方法

void push\_front (const value\_type& val);

void push\_front (value\_type&& val);

在容器开头即当前第一个元素前添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小直接加一。这里存在一个相似的成员函数：list::emplace\_front，该函数向容器中插入用提供的参数构造的元素

pop\_front方法

void pop\_front();

删除列表对象中的第一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

insert方法

iterator insert (const\_iterator position, const value\_type& val); // single element

iterator insert (const\_iterator position, size\_type n, const value\_type& val); // fill

template <class InputIterator>

insert (const\_iterator position, InputIterator first, InputIterator last);// range (3)

iterator insert (const\_iterator position, value\_type&& val); //move

iterator insert (const\_iterator position, initializer\_list<value\_type> il); // initializer list

在指定位置处的元素前插入一个新的元素。直接使容器的大小（Size）增加，增加的值就是所插入元素的个数。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。相比其它类型的顺序容器， std::list 或 std::forward\_list 是专为在任意位置高效地插入或删除元素而设计的。这里存在一个相似的成员函数：list::emplace，该函数向容器中插入用提供的参数构造的元素。

erase方法

iterator erase (const\_iterator position);

iterator erase (const\_iterator first, const\_iterator last);

从当前列表中移除单个（position 所指向的）或一个范围（[first,last)）的元素。直接减小容器大小（Size），减小的值等于所移除元素的个数。被移除的元素将被销毁（Destroy）。相比其它类型的顺序容器， std::list 或 std::forward\_list 是专为在任意位置高效地插入或删除元素而设计的。

swap方法

void swap (list& x);

使另一个同类型的列表对象 x 与当前列表交换内容。容器大小（Size）可以不同。容器中的内存分配器是否也被交换没有明确规定。与两个被交换对象有关的所有迭代器、指针及引用仍保持有效。

clear方法

void clear() noexcept

移除当前容器中的所有元素，使容器的大小（Size）变为 0。不保证是否会发生内存重分配（Reallocation），同样的，列表的容量（Capacity）也不保证会发生改变。

emplace方法（C++11）

template <class... Args> iterator emplace (const\_iterator position, Args&&... args)

在指定位置处的元素前插入一个新的元素。新的元素是以 args 作为其构造函数的参数而构造的。直接使容器的大小（Size）增加 1。相比其它类型的顺序容器， std::list 或 std::forward\_list 是专为在任意位置高效地插入或删除元素而设计的。成员函数 list::emplace\_front 及 list::emplace\_back 能够直接在容器起始或末尾插入元素。这里存在一个相似的成员函数：list::insert，该函数直接拷贝（或移动 C++11）已存在的元素到容器。返回一个指向新放置（Emplaced）的元素的迭代器

emplace\_front方法（C++11）

template <class... Args>void emplace\_front (Args&&... args);

在容器开头（即第一个元素前）添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。这里存在一个相似的成员函数：list::push\_front，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

emplace\_back方法（C++11）

template <class... Args>void emplace\_back (Args&&... args);

在列表末尾添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。这里存在一个相似的成员函数：list::push\_back，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

#### 3.2.5.获取分配器

get\_allocator方法

allocator\_type get\_allocator() const noexcept;

get\_allocator() 函数返回与vector关联的分配器成员类型。

allocator\_type 是给容器内部使用的内存分配器的类型，如果使用默认的内存分配器allocator\_type 的值是不确定的。

#### 3.2.6.操作

splice方法

void splice (const\_iterator position, list& x); // entire list (1)

void splice (const\_iterator position, list&& x); // entire list (1)

void splice (const\_iterator position, list& x, const\_iterator i); // single element (2)

void splice (const\_iterator position, list&& x, const\_iterator i); // single element (2)

void splice (const\_iterator position, list& x, const\_iterator first, const\_iterator last); // element range (3)

void splice (const\_iterator position, list&& x, const\_iterator first, const\_iterator last); // element range (3) 将 x 中的元素转移到当前容器中，插入到指定位置处。该函数高效地从一个容器中删除元素，并插入到另一个容器中，同时修改两个容器的大小。这一操作不会涉及任何元素的构造或析构，而是仅仅发生转移。不管 x 是左值（Lvalue）还是右值（Rvalue）， 或者 value\_type 是否支持移动构造（Move-construction），都能够正常进行转移操作。如果当前容器与 x 的内存分配（Allocator）的比较结果不相等，将会导致无法预测的行为。

转移方式有以下几种：

(1) 整个列表：转移 x 中的所有元素到当前容器。

(2) 单个元素：转移 x 中由 i 指向的元素到当前容器。

(3) 元素范围：转移 x 中范围 [first,last) 内的所有元素到当前容器。

remove方法

void remove (const value\_type& val);

删除容器中所有等值于 val 的元素。该请求调用每个元素对象的析构函数（Destructor），并直接减小容器的大小（Size），减小的值等于所移除元素的个数。区别于成员函数 list::erase，list::erase 通过位置（使用一个迭代器）来清除元素，而当前函数（list::remove）通过值来删除元素。这里存在一个相似的成员函数：list::remove\_if，该函数通过一个判断条件而不是值比较来决定一个元素是否需要删除。

remove\_if方法

template <class Predicate>void remove\_if (Predicate pred);

删除容器中所有使谓词函数 pred 返回 true 的元素。该请求调用每个元素对象的析构函数（Destructor），并直接减小容器的大小（Size），减小的值等于所移除元素的个数。当前函数依次对容器中的每个元素调用 pred(\*i) （i 即指向对应元素的迭代器）。使上述调用返回 true 的任何元素都将从容器中移除。区别于成员函数 list::erase，list::erase 通过位置（使用一个迭代器）来清除元素，而当前函数（list::remove\_if）通过谓词函数删除元素。这里存在一个相似的成员函数：list::remove，该函数通过判断容器中的元素与一个给定值的等值比较结果来决定对应元素是否需要删除。

unique方法

void unique()// (1)

template <class BinaryPredicate>void unique (BinaryPredicate binary\_pred);//（2）

删除容器中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。版本 (1) 使用 operator== 操作符来比较两个元素。版本 (2) 使用给定的二元谓词（Binary predicate）函数 binary\_pred 来比较两个元素。被移除的元素将被销毁。

merge方法

void merge (list& x);//(1)

void merge (list&& x);//（1）

template <class Compare>/void merge (list& x, Compare comp);//(2)

template <class Compare>void merge (list&& x, Compare comp);//（2）

将 x 中的所有元素转移到当前容器中各自对应的排序位置处。两个容器中的元素都必须排过序。该函数高效地从一个容器中删除元素，并插入到当前容器中（直接使当前容器的大小（Size）增加，增加的值就是所转移元素的个数）。这一操作不会涉及任何元素的构造或析构，而是仅仅发生转移。不管 x 是左值（Lvalue）还是右值（Rvalue）， 或者 value\_type 是否支持移动构造（Move-construction），都能够正常进行转移操作。

该合并操作是通过运用使用了全局操作符 std::operator<（版本(1)）或谓词函数 comp （版本(2)）来比较元素的法算进行的。该比较将会产生元素的一个严格弱序（Strict weak ordering）（比如一个连续的可传递的比较，且不用考虑该比较的自反性）。

使用该函数前，两个容器中的元素必须已经通过使用全局操作符 std::operator<（或 comp 谓词函数）排过序。如果想合并未排序列表，可以选择使用成员函数 list::splice。

如果 x 中的元素按由操作符 std::operator< （或 comp 谓词函数）定义的严格弱序（Strict weak ordering）插入到当前容器中对应值的位置处，值相等元素的排序结果是稳定的，比如：同一容器中值相等的元素在调用该函数后保持调用前的相对顺序，当前容器的元素排在转移自 x 的等值元素前。

如果 &x == this，该函数不会做任何事。

sort方法

void sort();// (1)

template <class Compare>void sort (Compare comp); // (2)

为列表中的元素排序，改变它们在容器中的位置。该排序操作是通过运用使用了全局操作符 std::operator<（版本(1)）或谓词函数 comp （版本(2)）来比较元素的法算进行的。该比较将会产生元素的一个严格弱序（Strict weak ordering）（比如一个连续的可传递的比较，且不用考虑该比较的自反性）。

相等元素的排序结果是稳定的，比如：值相等的元素在调用该函数后保持调用前的相对顺序。在该操作的整个过程中，不会涉及任何元素的构造、析构或拷贝，仅仅是在容器内部移动元素。

reverse方法

void reverse() noexcept;

反转列表容器中元素的顺序。标准C++库的算法库中存在一个相似的函数：std::reverse，该函数同样用于反转容器中的元素。当前成员函数（list::reverse）与 std::reverse 的区别就是调用 list::reverse 后不会影响其它正在使用的迭代器所指向的值。

## 4.序列容器之deque

### 4.1.deque的概念

#### 4.1.1.定义

双端队列（Double-ended queue，缩写为Deque）是一个大小可以动态变化（Dynamic size）且可以在两端扩展或收缩的顺序容器。

#### 4.1.2.容器特性

顺序序列：顺序容器中的元素按照严格的线性顺序排序。可通过元素在序列中的位置访问对应的元素。

动态数组：通常采用动态数组这一数据结构。支持对序列中的任意元素进行快速直接访问。操供了在序列末尾相对快速地添加/删除元素的操作。

能够感知内存的分配器（Allocator-aware）：容器使用一个内存分配器对象来动态地处理它的存储需求。

#### 4.1.3. 详细说明-ISO/IEC 14882:2011

不同的库可能会按不同的方式来实现双端队列，通常实现为某种形式的动态数组。但不管通过哪种方式，双端队列都允许通过随机迭代器直接访问各个元素，且内部的存储空间会按需求自动地扩展或收缩。

容器实际分配的内存数超过容纳当前所有有效元素所需的内存，因为额外的内存将被未来增长的部分所使用。就因为这点，当插入元素时，容器不需要太频繁地分配内存。

因此，双端队列提供了类似向量（std::vector）的功能，且不仅可以在容器末尾，还可以在容器开头高效地插入或删除元素。

但是，相比向量，双端队列不保证内部的元素是按连续的存储空间存储的，因此，不允许对指针直接做偏移操作来直接访问元素。

在内部，双端队列与向量的工作方式完全不同：向量使用单数组数据结构，在元素增加的过程中，需要偶尔的内存重分配，而双端队列中的元素被零散地保存在不同的存储块中，容器内部会保存一些必要的数据使得可以以恒定时间及一个统一的顺序接口直接访问任意元素。因此，双端队列的内部实现比向量的稍稍复杂一点，但这也使得它在一些特定环境下可以更高效地增长，特别是对于非常长的序列，内存重分配的代价是及其高昂的。

对于大量涉及在除了起始或末尾以外的其它任意位置插入或删除元素的操作，相比列表（std::list）及正向列表（std::forward\_list），deque 所表现出的性能是极差的，且操作前后的迭代器、引用的一致性较低。

#### 4.1.4.定义对象

##### 4.1.4.1.模板

#include<deque>

template < class T, class Alloc = allocator<T> > class deque;

##### 4.1.4.2构造函数

explicit deque (const allocator\_type& alloc = allocator\_type());// default (1)

explicit deque (size\_type n); // fill (2)

deque (size\_type n, const value\_type& val,const allocator\_type& alloc = allocator\_type());

template <class InputIterator>

deque (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type());// range (3)

deque (const deque& x); // copy (4)

deque (const deque& x, const allocator\_type& alloc);

deque (deque&& x); // move (5)

deque (deque&& x, const allocator\_type& alloc);

deque (initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type());// initializer list (6)

(1)空容器构造函数（Default）

构造一个空的容器，元素数为0。

(2)填充（Fill）构造函数

构造一个带 n 个元素的容器，每个元素都是一份 val（如果有提供，否则调用元素类型的默认构造函数创建一个临时的）的拷贝。

(3)范围（Range）构造函数

用范围 [first,last) 的内容构造容器，容器中的元素按顺序对应于范围中的各个元素。

(4) 拷贝（Copy）构造函数（及带内存分配器拷贝）

按顺序拷贝另一个同类型的容器对象的内容来构造新的容器。

(5) 移动（Move）构造函数（及带内存分配器移动）

以Move语义获得另外一个同类型容器的元素来构造容器。如果未提供 alloc 参数，则从另一个容器的内存分配器移动构造获得。如果提供了 alloc 参数且与被移动容器自带的内存分配器不一样，元素将被移动，否则无任一元素将被构造（而是直接传递所有权）。被移动后的 x 的状态是未定义的。

(6) 初始化列表（Initializer list）构造函数

用初始化列表的内容按序构造容器。

容器内部保存了 alloc 的一份拷贝，可以用它分配或释放元素的存储空间，且可以用它来构造或销毁已分配存储空间的元素对象（由 alloc 类型的 std::allocator\_traits 指定）。对于拷贝构造函数(4)，如果 alloc 参数未提供，则内存分配器由通过调用std::allocator\_traits<allocator\_type>::select\_on\_copy\_construction(x) 获得。对于移动构造函数(5)，如果 alloc 参数未提供，则直接获取 x 的内存分配器。

所有被拷贝、移动或其它类似操作的元素都是通过用合适的参数调用 std::allocator\_traits::construct 来构造的。

**参数及返回值**

alloc内存分配器对象。容器内部将会保存且使用该内存分配器的拷贝。可以使用 deque::get\_allocator 函数获得容器内部的内存分配器的一份拷贝。

成员类型 allocator\_type 是给容器内部使用的内存分配器的类型，作为 deque 实例化时第一个模板参数的别名而存在。如果使用默认的内存分配器，allocator\_type 的值是不确定的。

n初始容器大小。

成员类型 size\_type 是一个无符号整数类型。表示元素个数或者空间大小。

val用来填充容器的值。容器中 n 个元素中的每一个都将被初始化为该值的拷贝。

成员类型 value\_type 是该容器的元素的类型，作为 deque 实例化时第一个模板参数的别名而存在。

first，last分别指向一个序列中初始及末尾位置的输入迭代器。这个范围即 [first,last) ，包括 first 到 last 间的所有元素，包括 first 指向的元素，但不包括 last 指向的元素。

x另一个同类型的 deque 对象（有相同的模板参数 T 及 Alloc）。

il一个初始化列表对象。编译器将根据初始化列表声明体自动创建该对象。

### 4.2.常用public成员函数

#### 4.2.1.元素访问

operator[]

reference operator[] (size\_type n);

const\_reference operator[] (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。deque::at 成员函数与该操作符函数有类似的行为，只是 deque::at 会做边界检测，当请求的位置超出有效元素的范围（Size）时将抛出 std::out\_of\_range 异常。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

参数n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

参数size\_type ：一个无符号整数类型。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

at方法

reference at (size\_type n);

const\_reference at (size\_type n) const;

返回当前容器第 n 个位置处的元素的引用。该函数会自动地做边界检测，当请求的位置超出容器中有效元素的范围时将抛出 std::out\_of\_range 异常。该点正好与 deque::operator[] 的形成对比，后者不做边界检测。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

n：容器中一个元素的位置。注意第一个元素的位置是 0（而不是1）。

size\_type ：一个无符号整数类型。

front方法

reference front();

const\_reference front() const;

返回对当前容器中第一个元素的引用（Reference）。区别于 deque::begin 函数，deque::begin 返回的是指向同一个元素的迭代器（Iterator），而当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

back方法

reference back();

const\_reference back() const;

返回对当前容器中最后一个元素的引用（Reference）。区别于 deque::end 函数，deque::end 返回的是最后一个元素的下一个元素的迭代器（Iterator），当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

#### 4.2.2.迭代器

deque支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 deque::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 deque 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

##### 4.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 deque 对象中第一个元素的迭代器。如果 deque 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 deque::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 deque::front 返回的是对第一个元素的引用（Reference）。deque::begin 和 deque::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 deque 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 deque 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。deque::begin 和 deque::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 deque::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 deque 对象中最后一个元素的逆序迭代器。如果 deque 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。deque::rbegin 返回值指向的元素刚好是 deque::end 返回值指向的元素的前一个元素。区别于 deque::back，当前函数返回的是指向最后一个元素的逆序随机访问迭代器（Reverse random access iterator），而 deque::back 返回指向最后一个元素的引用（Reference）。deque::rbegin 和 deque::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 deque 对象中第一个元素之前（ Preceding the first）的元素的逆序迭代器，如果 deque 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。deque::rbegin 和 deque::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

##### 4.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 deque 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 deque::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 deque 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。deque::cbegin 和 deque::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 deque 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 deque::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 deque 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 deque::cbegin 相同。deque::cbegin 和 deque::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 deque 对象中最后一个元素的常逆序迭代器。deque::crbegin 和 deque::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 deque 对象中第一个元素之前（ Preceding the first）的元素的常逆序迭代器。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。deque::crbegin 和 deque::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

#### 4.2.3.容量

size 方法

size\_type size() const noexcept;

返回当前容器的元素个数。等同于 std::distance(deque::begin(), deque::end())，这是当前容器的有效元素个数，它不需要与当前容器的容量（Capacity）相等。

max\_size

size\_type max\_size() const noexcept;

返回 deque 能容纳的最大元素个数。是一个当前容器能够达到的受限于操作系统或编译器的最大可能大小。但实事上容器并不保证能够达到这个大小：在达到这个上限之前，已经出现内存分配失败。

resize方法

void resize (size\_type n);

void resize (size\_type n, value\_type val);

调整容器的大小使其能够存放 n 个元素。返回新deque的大小。

如果 n 比当前容器大小（Size）小，容器中的内容将被减少到只剩前 n 个元素，超出的部份将被移除（或销毁）。但容量不变。

如果 n 比当前容器大小（Size）大，将通过在容器尾部插入足够的元素使内容增加到 n 个元素。如果提供了 val ，新的元素将被初始化为 val 的拷贝，否则将被值初始化（Value-initialized）。

如果 n 比当前容器容量（Capacity）大，将会对已分配内存空间自动进行一次内存重分配。

capacity 方法

size\_type capacity() const noexcept;

返回当前分配给容器的存储空间的大小，以元素大小为单位。存储容量不一定要跟容器大小（Size）一样，可以相等或更大。

empty 方法

bool empty() const;

如果容器大小（Size）为 0，则返回 true，否则返回 false。

shrink\_to\_fit方法（C++11）

void shrink\_to\_fit();

请求容器减小容量使其足够用来存放所有有效元素。因为双端队列容器实际分配的内存数超过容纳当前所有有效元素所需的，因为额外的内存将被未来增长的部分所使用。就因为这点，当插入元素时，容器不需要太频繁地分配内存。使容器内部可以随意优化，最后的实际容量大于或等于容器的有效元素的个数（Size）。用该请求可能会引起一次内存重分配，但不会对 size 产生影响。

#### 4.2.4.修改

赋值运算符=

deque& operator= (const deque& x);//copy

deque& operator= (deque&& x);//move

deque& operator= (initializer\_list<value\_type> il); //initializer

赋值新的内容到容器， 以替换当前内容，并根据情况修改容器的大小（Size）。进行该调用前的所有元素或者被重新赋值，或者被销毁。各个声明的具体含义如下：

(1) 拷贝（Copy）赋值：将 x 中的所有元素拷贝到当前容器，x 保留之前的内容。

(2) 移动（Move）赋值：将 x 中的所有元素拷贝到当前容器，x 进入未定义状态。

(3) 用初始化列表（Initializer list）赋值：拷贝列表中的元素到当前容器。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

容器将保留当前的内存分配器（Allocator），除非内存分配器特征表明 x 的的内存分配器需要传递进当前容器。

assign方法

template <class InputIterator> void assign( input\_iterator start, input\_iterator end ); //range

void assign( size\_type num, const type &val );//fill

void assign (initializer\_list<value\_type> il); //initializer list

assign() 函数要么将区间[start, end)的元素赋到当前deque,或者赋num个值为val的元素到deque中或者用初始化列表的内容按序赋值给容器。这个函数将会清除掉为deque赋值以前的内容。该函数被调用前的所有元素将被销毁，然后被新构建的元素所替换（而不会发生元素的赋值）。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

当发生内存重分配时（Reallocation），容器内部的内存分配器（std::allocator\_traits）被用来分配或释放元素的存储空间，同时被用来销毁所有已经存在的元素对象，然后构建新的元素对象。

push\_back方法

void push\_back (value\_type& val)

void push\_back (value\_type&& val);

在双端队列末尾添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。这里存在一个相似的成员函数：deque::emplace\_back，该函数向容器插入用提供的参数构造的元素。

pop\_back方法

void pop\_back();

删除双端队列对象中的最后一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

push\_front方法

void push\_front (const value\_type& val);

void push\_front (value\_type&& val);

在容器开头即当前第一个元素前添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小直接加一。这里存在一个相似的成员函数：deque::emplace\_front，该函数向容器中插入用提供的参数构造的元素

pop\_front方法

void pop\_front();

删除列表对象中的第一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

insert方法

iterator insert (const\_iterator position, const value\_type& val); // single element

iterator insert (const\_iterator position, size\_type n, const value\_type& val); // fill

template <class InputIterator>

insert (const\_iterator position, InputIterator first, InputIterator last);// range (3)

iterator insert (const\_iterator position, value\_type&& val); //move

iterator insert (const\_iterator position, initializer\_list<value\_type> il); // initializer list

在指定位置处的元素前插入一个新的元素。直接使容器的大小（Size）增加，增加的值就是所插入元素的个数。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。双端队列被设计为能够在容器两端进行高效地插入或删除元素操作，相比 std::list 或 std::forward\_list C++11，将元素插入到其它位置通常不是那么高效。

这里存在一个相似的成员函数：deque::emplace，该函数向容器中插入用提供的参数构造的元素。

erase方法

iterator erase (const\_iterator position);

iterator erase (const\_iterator first, const\_iterator last);

从当前双端队列中移除单个（position 所指向的）或一个范围（[first,last)）的元素。直接减小容器大小（Size），减小的值等于所移除元素的个数。被移除的元素将被销毁（Destroy）。双端队列被设计为能够在容器两端进行高效地插入或删除元素操作，相比 std::list 或 std::forward\_list C++11，删除其它位置的元素通常不是那么高效。

swap方法

void swap (deque& x);

使另一个同类型的双端队列对象 x 与当前双端队列交换内容。容器大小（Size）可以不同。容器中的内存分配器是否也被交换没有明确规定。与两个被交换对象有关的所有迭代器、指针及引用仍保持有效。

clear方法

void clear() noexcept

移除当前容器中的所有元素，使容器的大小（Size）变为 0。不保证是否会发生内存重分配（Reallocation），同样的，双端队列的容量（Capacity）也不保证会发生改变。

emplace方法（C++11）

template <class... Args> iterator emplace (const\_iterator position, Args&&... args)

在指定位置处的元素前插入一个新的元素。新的元素是以 args 作为其构造函数的参数而构造的。直接使容器的大小（Size）增加 1。

双端队列被设计为能够在容器两端进行高效地插入或删除元素操作，相比 std::list 或 std::forward\_list C++11，将元素插入到其它位置通常不是那么高效。成员函数 deque::emplace\_front 及 deque::emplace\_back 能够直接在容器起始或末尾插入元素。

这里存在一个相似的成员函数：deque::insert，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

返回一个指向新放置（Emplaced）的元素的迭代器

emplace\_front方法（C++11）

template <class... Args>void emplace\_front (Args&&... args);

在容器开头（即第一个元素前）添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。这里存在一个相似的成员函数：deque::push\_front，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

emplace\_back方法（C++11）

template <class... Args>void emplace\_back (Args&&... args);

在双端队列末尾添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。这里存在一个相似的成员函数：deque::push\_back，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

#### 4.2.5.获取分配器

get\_allocator方法

allocator\_type get\_allocator() const noexcept;

get\_allocator() 函数返回与vector关联的分配器成员类型。

allocator\_type 是给容器内部使用的内存分配器的类型，如果使用默认的内存分配器allocator\_type 的值是不确定的。

## 5.序列容器之forward\_list

### 5.1.forward\_list概念

#### 5.1.1.定义

正向列表（Forward list）是一个允许在序列中任何一处位置以常量耗时插入或删除元素的顺序容器（Sequence container）。

#### 5.1.2.容器特性

顺序序列：顺序容器中的元素按照严格的线性顺序排序。可通过元素在序列中的位置访问对应的元素。

单链表：容器中的每个元素保存了定位前一个元素及后一个元素的信息，允许在任何一处位置以常量耗时进行插入或删除操作，但不能进行直接随机访问（Direct random access）。

能够感知内存分配器的（Allocator-aware）：容器使用一个内存分配器对象来动态地处理它的存储需求。

#### 5.1.3.详细说明- ISO/IEC 14882:2011

标准模板库（Standard Template Library，简称STL）中的正向列表容器采用的是单链表（Singly-linked lists）数据库构。单链表可以保存以不同且无关的储存位置容纳的元素。元素间的顺序是通过各个元素中指向下一个元素的链接这一联系维护的。

该特点跟 std:: list 容器的有点相似：主要的区别就是 std:: list 的元素中不仅保存了指向下一个元素的链接，还保存了指向上一个元素的链接，这使得 std::list 允许双向迭代，但消耗更多的存储空间且在插入删除元素时会有稍高（Slight higher）的耗时。因此 std::forward\_list 对象比 std::list 对象更高效，尽管它们只能正向迭代。

增加或移动正向列表中的元素不会使指向它们的指针、引用、迭代器失效，只有当对应的元素被销毁时才会如此。

相比较于其它的基本标准顺序容器（数组 std::array、向量 std::vector、双端队列 std::deque 等），正向列表通常在容器中已获得迭代器的任意位置处插入、获得（Extracting，提取）、移动元素等操作中表现出更出色的性能，对那些密集使用上述操作的算法，使用正向列表同样能提升性能，比如排序算法。

双向链表（std::list）及正向链表（std:: forward\_list）相比于其它顺序容器的主要缺点是它们不能够通过元素在容器中的位置直接访问（Direct access）元素。举个例子：如果想访问一个正向列表中的第六个元素，那么就必须从一个已知位置（比如开头或末尾）处开始迭代，这会消耗与两个位置之间距间相关的线性时间。而且它们还为保存各个元素间的链接信息消耗更多额外的内存（这点对由小尺寸元素组成而元素数较大的正向列表尤为明显）。

forward\_list 类模板是专为极度考虑性能的程序而设计的，它就跟自实现的C型单链表（C-style singly-linked list）一样高效。甚至为了性能，**它是唯一一个缺少 size 成员函数的标准容器**：这是出于其单链表的性质，如果拥有 size 成员函数，就必须消耗常量时间来维护一个用于保存当前容器大小的内部计数器，这会消耗更多的存储空间，且使得插入及删除操作略微降低性能。如果想获得 forward\_list 的大小，可以使用 std::distance 算法且传递 forward\_list::begin 及 forward\_list::end 参数，该操作的时间复杂度为 O(n)。

对任意列表（std::list）进行插入或删除元素操作需要访问插入位置前的元素，但对 forward\_list 来说访问该元素没有常数时间（Constant-time）的方法。因为这个原因，对传递给清除（Erase）、拼接（Splice）等成员函数的范围参数作了修改，这些范围必须为开区间（即不包括末尾元素的同时也不包括起始元素）。

#### 5.1.4.定义对象

##### 5.1.4.1.模板

#include <forward\_list>

template < class T, class Alloc = allocator<T> > class forward\_list;

T存储在容器中的元素的数据类型。在类模板内部，使用其别名为 value\_type 的成员类型。

Alloc容器内部用来管理内存分配及释放的内存分配器的类型。这个参数是可选的，它的默认值是 std::allocator<T>，这个是一个最简单的非值依赖的（Value-independent）内存分配器。 在类模板内部，使用其别名为 allocator\_type 的成员类型。

##### 5.1.4.2.构造函数

explicit forward\_list (const allocator\_type& alloc = allocator\_type());// default (1)

explicit forward\_list (size\_type n); // fill (2)

explicit forward\_list (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type());/

template <class InputIterator>

forward\_list (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type());// (3)

forward\_list (const forward\_list& x); // copy (4)

forward\_list (const forward\_list& x, const allocator\_type& alloc); // copy (4)

forward\_list (forward\_list&& x); // move (5)

forward\_list (forward\_list&& x, const allocator\_type& alloc); // move (5)

forward\_list (initializer\_list<value\_type> il,const allocator\_type& alloc = allocator\_type());//(6)

(1) 空容器构造函数（Default）

构造一个空的容器，元素数为 0。

(2) 填充（Fill）构造函数

构造一个带 n 个元素的容器，每个元素都是一份 val（如果有提供，否则调用元素类型的默认构造函数创建一个临时的）的拷贝。

(3) 范围（Range）构造函数

用范围 [first,last) 的内容构造容器，容器中的元素按顺序对应于范围中的各个元素。

(4) 拷贝（Copy）构造函数（及带内存分配器拷贝）

按顺序拷贝另一个同类型的容器对象的内容来构造新的容器。

(5) 移动（Move）构造函数（及带内存分配器移动）

以Move语义获得另外一个同类型容器的元素来构造容器。如果未提供 alloc 参数，则从另一个容器的内存分配器移动构造获得。如果提供了 alloc 参数且与被移动容器自带的内存分配器不一样，元素将被移动，否则无任一元素将被构造（而是直接传递所有权）。被移动后的 x 的状态是未定义的。

(6) 初始化列表（Initializer list）构造函数

用初始化列表的内容按序构造容器。

容器内部保存了 alloc 的一份拷贝，可以用它分配或释放元素的存储空间，且可以用它来构造或销毁已分配存储空间的元素对象（由 alloc 类型的 std::allocator\_traits 指定）。对于拷贝构造函数(4)，如果 alloc 参数未提供，则内存分配器由通过调用std::allocator\_traits<allocator\_type>::select\_on\_copy\_construction(x) 获得。对于移动构造函数(5)，如果 alloc 参数未提供，则直接获取 x 的内存分配器。

所有被拷贝、移动或其它类似操作的元素都是通过用合适的参数调用 std::allocator\_traits::construct 来构造的。

参数及返回值

alloc存分配器对象。容器内部将会保存且使用该内存分配器的拷贝。可以使用 list::get\_allocator 函数获得容器内部的内存分配器的一份拷贝。

成员类型 allocator\_type 是给容器内部使用的内存分配器的类型，作为 list 实例化时第一个模板参数的别名而存在。如果使用默认的内存分配器，allocator\_type 的值是不确定的。

n初始容器大小。

成员类型 size\_type 是一个无符号整数类型。

val用来填充容器的值。容器中 n 个元素中的每一个都将被初始化为该值的拷贝。

成员类型 value\_type 是该容器的元素的类型，作为 list 实例化时第一个模板参数的别名而存在。

first，last

分别指向一个序列中初始及末尾位置的输入迭代器。这个范围即 [first,last) ，包括 first 到 last 间的所有元素，包括 first 指向的元素，但不包括 last 指向的元素。

x另一个同类型的 list 对象（有相同的模板参数 T 及 Alloc）。

il一个初始化列表对象。

编译器将根据初始化列表声明体自动创建该对象。

### 5.2.常用public成员函数

#### 5.2.1.元素访问

front方法

reference front();

const\_reference front() const;

返回对当前容器中第一个元素的引用（Reference）。区别于 forward\_list::begin 函数，forward\_list::begin 返回的是指向同一个元素的迭代器（Iterator），而当前函数返回的是一个直接引用。当对一个空的容器调用该函数时将会导致不确定（Undefined）的行为。如果容器对象是 const 限定的，函数返回类型为 const\_reference，否则返回 reference。

#### 5.2.2.迭代器

forward\_list支持迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 forward\_list::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 forward\_list 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

##### 5.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 forward\_list 对象中第一个元素的迭代器。如果 forward\_list 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 forward\_list::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 forward\_list::front 返回的是对第一个元素的引用（Reference）。forward\_list::begin 和 forward\_list::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 forward\_list 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 forward\_list 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。forward\_list::begin 和 forward\_list::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 forward\_list::begin 相同。

before\_begin方法

iterator before\_begin() noexcept;

const\_iterator before\_begin() const noexcept;

返回指向容器中第一个元素之前位置的迭代器。如果容器对象是 const 限定的，函数返回类型为 const\_iterator，否则返回 iterator。第一个元素之前的位置，是一个“理论上的位置”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。该函数返回的迭代器主要被用来作为 emplace\_after、insert\_after、erase\_after 或 splice\_after 等成员函数的参数，以提供该类操作可能会用到的序列的起始位置。

##### 5.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 forward\_list 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 forward\_list::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 forward\_list 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。forward\_list::cbegin 和 forward\_list::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 forward\_list 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 forward\_list::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 forward\_list 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 forward\_list::cbegin 相同。forward\_list::cbegin 和 forward\_list::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cbefore\_begin方法

const\_iterator cbefore\_begin() const noexcept;

返回指向容器中第一个元素之前位置的常迭代器。第一个元素之前的位置，是一个“理论上的位置”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 forward\_list::before\_begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 forward\_list 对象未被 const 限定。该函数返回的迭代器主要被用来作为 emplace\_after、insert\_after、erase\_after 或 splice\_after 等成员函数的参数，以提供该类操作可能会用到的序列的起始位置。

#### 5.2.3.容量

max\_size

size\_type max\_size() const noexcept;

返回list 能容纳的最大元素个数。是一个当前容器能够达到的受限于操作系统或编译器的最大可能大小。但实事上容器并不保证能够达到这个大小：在达到这个上限之前，已经出现内存分配失败。

empty方法

bool empty() const;

如果容器大小（Size）为 0，则返回 true，否则返回 false。

void resize (size\_type n);

void resize (size\_type n, const value\_type& val);

resize方法

void resize (size\_type n);

void resize (size\_type n, value\_type val);

调整容器的大小使其能够存放 n 个元素。

如果 n 比当前容器大小（Size）小，内容减少到容器前的 n 个元素，超出的部份将被移除（或销毁）。

如果 n 比当前容器大小（Size）大，将通到在容器尾部插入足够的元素使内容增加到 n 个元素。如果提供了 val ，新的元素将被初始化为 val 的拷贝，否则将被值初始化（Value-initialized）。

#### 5.2.4修改

赋值运算符=

forward\_list& operator= (const forward\_list& x);//copy

forward\_list& operator= (forward\_list&& x);//move

forward\_list& operator= (initializer\_forward\_list<value\_type> il); //initializer

赋值新的内容到容器， 以替换当前内容，并根据情况修改容器的大小（Size）。进行该调用前的所有元素或者被重新赋值，或者被销毁。各个声明的具体含义如下：

(1) 拷贝（Copy）赋值：将 x 中的所有元素拷贝到当前容器，x 保留之前的内容。

(2) 移动（Move）赋值：将 x 中的所有元素拷贝到当前容器，x 进入未定义状态。

(3) 用初始化正向列表（Initializer forward\_list）赋值：拷贝正向列表中的元素到当前容器。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

容器将保留当前的内存分配器（Allocator），除非内存分配器特征表明 x 的的内存分配器需要传递进当前容器。

assign方法

template <class InputIterator>void assign (InputIterator first, InputIterator last); // range (1)

void assign (size\_type n, const value\_type& val); // fill (2)

void assign (initializer\_forward\_list<value\_type> il); // initializer forward\_list (3)

assign() 函数要么将区间[start, end)的元素赋到当前forward\_list,或者赋num个值为val的元素到forward\_list中或者用初始化正向列表的内容按序赋值给容器。这个函数将会清除掉为forward\_list赋值以前的内容。该函数被调用前的所有元素将被销毁，然后被新构建的元素所替换（而不会发生元素的赋值）。

当且仅当新的容器的大小（Size）超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。

当发生内存重分配时（Reallocation），容器内部的内存分配器（std::allocator\_traits）被用来分配或释放元素的存储空间，同时被用来销毁所有已经存在的元素对象，然后构建新的元素对象。

push\_front方法

void push\_front (const value\_type& val);

void push\_front (value\_type&& val);

在容器开头即当前第一个元素前添加新的元素。val 的内容被拷贝（或移动 C++11）到新增的元素中。使容器大小直接加一。这里存在一个相似的成员函数：forward\_list::emplace\_front，该函数向容器中插入用提供的参数构造的元素

pop\_front方法

void pop\_front();

删除正向列表对象中的第一个元素。使容器大小（Size）直接减一。被删除的元素将被销毁。

insert\_after方法

iterator insert \_after (const\_iterator position, const value\_type& val); // single element

iterator insert \_after (const\_iterator position, value\_type&& val); //move

iterator insert \_after (const\_iterator position, size\_type n, const value\_type& val); // fill

template <class InputIterator>

insert \_after (const\_iterator position, InputIterator first, InputIterator last);// range (3)

iterator insert\_after (const\_iterator position,initializer\_forward\_list<value\_type> il);//initializer list

在指定位置处的元素前插入一个新的元素。直接使容器的大小（Size）增加，增加的值就是所插入元素的个数。当且仅当新的容器的大小超过当前容器的容量（Capacity），才会导致一次已分配的存储空间的自动重分配。相比其它类型的顺序容器， std:: list 或 std::forward\_forward\_list 是专为在任意位置高效地插入或删除元素而设计的。这里存在一个相似的成员函数：forward\_list::emplace\_after，该函数向容器中插入用提供的参数构造的元素。不会进行任何拷贝或移动操作。

erase\_after方法

iterator erase\_after (const\_iterator position);

iterator erase \_after (const\_iterator first, const\_iterator last);

从当前正向列表中移除单个（position 所指向的）或一个范围（[first,last)）的元素。直接减小容器大小（Size），减小的值等于所移除元素的个数。被移除的元素将被销毁（Destroy）。相比其它类型的顺序容器， std:: list 或 std::forward\_forward\_list 是专为在任意位置高效地插入或删除元素而设计的。

swap方法

void swap (forward\_list& x);

使另一个同类型的正向列表对象 x 与当前正向列表交换内容。容器大小（Size）可以不同。容器中的内存分配器是否也被交换没有明确规定。与两个被交换对象有关的所有迭代器、指针及引用仍保持有效。

clear方法

void clear() noexcept

移除当前容器中的所有元素，使容器的大小（Size）变为 0。不保证是否会发生内存重分配（Reallocation），同样的，正向列表的容量（Capacity）也不保证会发生改变。

emplace\_afer方法（C++11）

template <class... Args> iterator emplace\_after (const\_iterator position, Args&&... args)

在指定位置处的元素前插入一个新的元素。新的元素是以 args 作为其构造函数的参数而构造的。直接使容器的大小（Size）增加 1。相比其它类型的顺序容器， std::list 或 std::forward\_forward\_list 是专为在任意位置高效地插入或删除元素而设计的。成员函数 forward\_list::emplace\_front 能够直接在容器起始插入元素。这里存在一个相似的成员函数：forward\_list::insert\_after，该函数直接拷贝（或移动 C++11）已存在的元素到容器。返回一个指向新放置（Emplaced）的元素的迭代器

emplace\_front方法（C++11）

template <class... Args>void emplace\_front (Args&&... args);

在容器开头（即第一个元素前）添加新的元素。该元素是以 args 作为其构造函数的参数而构造的。使容器大小（Size）直接加一。这里存在一个相似的成员函数：forward\_list::push\_front，该函数直接拷贝（或移动 C++11）已存在的元素到容器。

#### 5.2.5.获取分配器

get\_allocator方法

allocator\_type get\_allocator() const noexcept;

get\_allocator() 函数返回与vector关联的分配器成员类型。

allocator\_type 是给容器内部使用的内存分配器的类型，如果使用默认的内存分配器allocator\_type 的值是不确定的。

#### 5.2.6.操作

splice\_after方法

void splice\_after (const\_iterator position, forward\_list& x); // entire forward\_list (1)

void splice \_after (const\_iterator position, forward\_list&& x); // entire forward\_list (1)

void splice \_after (const\_iterator position, forward\_list& x, const\_iterator i); // single element (2)

void splice\_after (const\_iterator position, forward\_list&& x, const\_iterator i); // single element (2)

// element range (3)

void splice\_after (const\_iterator position, forward\_list& x, const\_iterator first, const\_iterator last);

void splice\_after (const\_iterator position, forward\_list&& x, const\_iterator first, const\_iterator last);

将 x 中的元素转移到当前容器中，插入到指定位置处。该函数高效地从一个容器中删除元素，并插入到另一个容器中，同时修改两个容器的大小。这一操作不会涉及任何元素的构造或析构，而是仅仅发生转移。不管 x 是左值（Lvalue）还是右值（Rvalue）， 或者 value\_type 是否支持移动构造（Move-construction），都能够正常进行转移操作。如果当前容器与 x 的内存分配（Allocator）的比较结果不相等，将会导致无法预测的行为。

转移方式有以下几种：

(1) 整个正向列表：转移 x 中的所有元素到当前容器。

(2) 单个元素：转移 x 中由 i 指向的元素到当前容器。

(3) 元素范围：转移 x 中范围 [first,last) 内的所有元素到当前容器。

remove方法

void remove (const value\_type& val);

删除容器中所有等值于 val 的元素。该请求调用每个元素对象的析构函数（Destructor），并直接减小容器的大小（Size），减小的值等于所移除元素的个数。区别于成员函数 forward\_list::erase\_after，forward\_list::erase\_after 通过位置（使用一个迭代器）来清除元素，而当前函数（forward\_list::remove）通过值来删除元素。这里存在一个相似的成员函数：forward\_list::remove\_if，该函数通过一个判断条件而不是值比较来决定一个元素是否需要删除。

remove\_if方法

template <class Predicate>void remove\_if (Predicate pred);

删除容器中所有使谓词函数 pred 返回 true 的元素。该请求调用每个元素对象的析构函数（Destructor），并直接减小容器的大小（Size），减小的值等于所移除元素的个数。当前函数依次对容器中的每个元素调用 pred(\*i) （i 即指向对应元素的迭代器）。使上述调用返回 true 的任何元素都将从容器中移除。区别于成员函数 forward\_list::erase\_after，forward\_list::erase\_after 通过位置（使用一个迭代器）来清除元素，而当前函数（forward\_list::remove\_if）通过谓词函数删除元素。这里存在一个相似的成员函数：forward\_list::remove，该函数通过判断容器中的元素与一个给定值的等值比较结果来决定对应元素是否需要删除。

unique方法

void unique()// (1)

template <class BinaryPredicate>void unique (BinaryPredicate binary\_pred);//（2）

删除容器中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。版本 (1) 使用 operator== 操作符来比较两个元素。版本 (2) 使用给定的二元谓词（Binary predicate）函数 binary\_pred 来比较两个元素。被移除的元素将被销毁。

merge方法

void merge (forward\_list& x);//(1)

void merge (forward\_list&& x);//（1）

template <class Compare>/void merge (forward\_list& x, Compare comp);//(2)

template <class Compare>void merge (forward\_list&& x, Compare comp);//（2）

将 x 中的所有元素转移到当前容器中各自对应的排序位置处。两个容器中的元素都必须排过序。该函数高效地从一个容器中删除元素，并插入到当前容器中（直接使当前容器的大小（Size）增加，增加的值就是所转移元素的个数）。这一操作不会涉及任何元素的构造或析构，而是仅仅发生转移。不管 x 是左值（Lvalue）还是右值（Rvalue）， 或者 value\_type 是否支持移动构造（Move-construction），都能够正常进行转移操作。

该合并操作是通过运用使用了全局操作符 std::operator<（版本(1)）或谓词函数 comp （版本(2)）来比较元素的法算进行的。该比较将会产生元素的一个严格弱序（Strict weak ordering）（比如一个连续的可传递的比较，且不用考虑该比较的自反性）。

使用该函数前，两个容器中的元素必须已经通过使用全局操作符 std::operator<（或 comp 谓词函数）排过序。如果想合并未排序正向列表，可以选择使用成员函数 forward\_list::splice。

如果 x 中的元素按由操作符 std::operator< （或 comp 谓词函数）定义的严格弱序（Strict weak ordering）插入到当前容器中对应值的位置处，值相等元素的排序结果是稳定的，比如：同一容器中值相等的元素在调用该函数后保持调用前的相对顺序，当前容器的元素排在转移自 x 的等值元素前。

如果 &x == this，该函数不会做任何事。

sort方法

void sort();// (1)

template <class Compare>void sort (Compare comp); // (2)

为正向列表中的元素排序，改变它们在容器中的位置。该排序操作是通过运用使用了全局操作符 std::operator<（版本(1)）或谓词函数 comp （版本(2)）来比较元素的法算进行的。该比较将会产生元素的一个严格弱序（Strict weak ordering）（比如一个连续的可传递的比较，且不用考虑该比较的自反性）。

相等元素的排序结果是稳定的，比如：值相等的元素在调用该函数后保持调用前的相对顺序。在该操作的整个过程中，不会涉及任何元素的构造、析构或拷贝，仅仅是在容器内部移动元素。

reverse方法

void reverse() noexcept;

反转正向列表容器中元素的顺序。标准C++库的算法库中存在一个相似的函数：std::reverse，该函数同样用于反转容器中的元素。当前成员函数（forward\_list::reverse）与 std::reverse 的区别就是调用 forward\_list::reverse 后不会影响其它正在使用的迭代器所指向的值。

## 6.容器适配器

### 6.1.概念

容器适配器可以看作是对容器的封装，使其具有某些特殊的功能。

stack(deque,list,vector)

queue(deque,list)

priority\_queue(vector, deque)。

所有适配器都定义了两个构造函数：默认构造函数用于创建空对象，而带一个容器参数的构造函数将参数容器的副本作为其基础值。

默认的stack和queue都基于deque容器实现，而priority\_queue则在vector容器上实现。在创建适配器时，通过将一个顺序容器指定为适配器的第二个类型参数，可覆盖其关联的基础容器类型。例如，下述代码创建的ob栈是基于vector实现的。

stack<int, vector<int> > ob;

### 6.2.分类

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **stack** | **queue** | **priority\_queue** | **功能** |
| 元素访问 | front |  | ✓ |  | 访问队首元素 |
| back |  | ✓ |  | 访问队尾元素 |
| top | ✓ |  | ✓ | 访问容器顶部元素 |
| 容量 | size | ✓ | ✓ | ✓ | 返回容器元素个数 |
| empty | ✓ | ✓ | ✓ | 判断容器是否为空 |
| 修改 | push | ✓ | ✓ | ✓ | 在顶部插入一个元素 |
| pop | ✓ | ✓ | ✓ | 删除一个元素 |
| emplace | ✓ | ✓ | ✓ | 插入一个由参数构造的匿名元素 |
| swap | ✓ | ✓ | ✓ | 同另一格容器交换 |
| = | ✓ | ✓ | ✓ | 一个容器向另一个容器赋值 |
| 自由函数 | 关系运算符 | ✓ | ✓ |  | 比较两个容器 |
| swap | ✓ | ✓ | ✓ | 交换两个容器 |

### 6.3.stack

#### 6.3.1.stack概念

栈（Stack）是一个容器适配器（Container adaptor）类型，被特别设计用来运行于LIFO（Last-in first-out）场景，在该场景中，只能从容器末尾添加（Insert）或提取（Extract）元素。

#### 6.3.2.详细说明-ISO/IEC 14882:2011

stack 通常被实现为容器适配器，即使用一个特定容器类的封装对象作为它的底层容器。stack 提供了一系列成员函数用于操作它的元素，只能从容器“后面”压进（Push）元素或从容器“后面”提取（Pop）元素。容器中的“后面”位置也被称为“栈顶”。

用来实现栈的底层容器必须满足顺序容器的所有必要条件。同时，它还必须提供以下语义的成员函数： back()、push\_back()​、pop\_back()

满足上述条件的标准容器有 std::vector、std::deque 及 std::list，如果未特别指定 stack 的底层容器，标准容器 std::deque 将被使用。

#### 6.3.3.定义对象

##### 6.3.3.1.模板

#include<stack>

template < class T, class Container = deque<T> > class stack;

T容器所包含的元素的类型。在类模板内部，使用其别名为 value\_type 的成员类型。

Container底层的用于存储元素的容器的类型。

##### 6.3.3.2.构造函数

explicit stack ( const Container& ctnr = Container() );

#### 6.3.4.public成员函数

##### 6.3.4.1.元素访问

top方法

value\_type& top ( );

​const value\_type& top ( ) const;

访问顶部元素

##### 6.3.4.2.容量

empty方法

bool empty ( ) const;

判断是否为空。

size方法

size\_type size ( ) const;

返回有效元素个数

##### 6.3.4.3.修改

operator=

stack<T, Container>&operator=( const stack<T,Container>& other );

stack<T, Container>&operator=( stack<T,Container>&& other );

push方法

void push ( const T& x );

在容器顶部插入元素

pop方法

void pop ( );

移除容器顶部的元素。

emplace方法

template< class... Args >void emplace( Args&&... args );

在容器顶部放置插入元素。

swap方法

void swap( stack& other );

交换容器的内容。

### 6.4.queue

#### 6.4.1.queue概念

队列（Queue）是一个容器适配器（Container adaptor）类型，被特别设计用来运行于FIFO（First-in first-out）场景，在该场景中，只能从容器一端添加（Insert）元素，而在另一端提取（Extract）元素。

#### 6.4.2.详细说明-ISO/IEC 14882:2011

queue 通常被实现为容器适配器，即使用一个特定容器类的封装对象作为它的底层容器。queue 提供了一系列成员函数用于操作它的元素，只能从容器“后面”压进（Push）元素,从容器“前面”提取（Pop）元素。

用来实现队列的底层容器必须满足顺序容器的所有必要条件。同时，它还必须提供以下语义的成员函数： front()、back()、push\_back()​、pop\_front()

满足上述条件的标准容器有 std::deque 及 std::list，如果未特别指定 queue 的底层容器，标准容器 std::deque 将被使用。

#### 6.4.3.定义对象

##### 6.4.3.1.模板

#include<queue>

template < class T, class Container = deque<T> > class stack;

T容器所包含的元素的类型。在类模板内部，使用其别名为 value\_type 的成员类型。

Container底层的用于存储元素的容器的类型。

##### 6.4.3.2.构造函数

explicit stack ( const Container& ctnr = Container() );

#### 6.4.4.public成员函数

##### 6.4.4.1.元素访问

front方法

value\_type& front ( );

​const value\_type& front ( ) const

访问队首元素

back方法

value\_type& back ( );

​const value\_type& back ( ) const

访问队尾元素

##### 6.4.4.2.容量

empty方法

bool empty ( ) const;

判断是否为空。

size方法

size\_type size ( ) const;

返回有效元素个数

##### 6.4.4.3.修改

operator=

queue<T, Container>&operator=( const queue<T,Container>& other );

queue<T, Container>&operator=( queue<T,Container>&& other );

push方法

void push ( const T& x );

在队尾插入元素

pop方法

void pop ( );

移除队首的元素。

emplace方法

template< class... Args >void emplace( Args&&... args );

在队首插入元素。

swap方法

void swap( stack& other );

交换容器的内容。

### 6.5.priority\_queue

#### 6.5.1. priority\_queue概念

先级队列（Priority queue）是一个容器适配器（Container adaptor）类型，被特别设计使其第一个元素总是容器所包含的元素中按特定的严格弱序排序规则排序后最大的一个。

#### 6.5.2.详细说明-ISO/IEC 14882:2011

priority\_queue 的定义使得它类似一个堆（Heap），该堆只能获得它的最大堆元素（在 priority\_queue 中即为队列头）。

priority\_queue 通常被实现为容器适配器，即使用一个特定容器类的封装对象作为它的底层容器。priority\_queue 提供了一系列成员函数用于操作它的元素，只能从容器“后面”提取（Pop）元素，该元素也可认为是 priority\_queue 的“顶部（Top）”。

用来实现优先级队列的底层容器必须满足顺序容器的所有必要条件。同时，它还必须提供以下语义的成员函数： front()、push\_back()​、pop\_back()

为了使内部始终保持堆结构，底层容器必须支持随机访问迭代器。这样，容器适配器内部就可以在适当的时候自动调用算法函数 std::make\_heap、std::push\_heap、std::pop\_heap。

满足上述条件的标准容器有 std::vector 及 std::deque，如果未特别指定 priority\_queue 的底层容器，标准容器 std::vector 将被使用。

#### 6.5.3.定义对象

##### 6.5.3.1.模板

#include<queue>

template < class T, class Container = vector<T>,class Compare = less<typename Container::value\_type> > class priority\_queue;

T容器所包含的元素的类型。在类模板内部，使用其别名为 value\_type 的成员类型。

Container底层的用于存储元素的容器的类型。

Compare一个二元谓词，以两个元素为参数返回一个 bool 值。可以是函数指针（Function pointer）类型或函数对象（Function object）类型。

##### 6.5.3.2.构造函数

explicit priority\_queue ( const Compare& x = Compare(),const Container& y = Container() );

template <class InputIterator>

priority\_queue ( InputIterator first, InputIterator last, const Compare& x = Compare(), const Container& y = Container() );

#### 6.5.4.public成员函数

##### 6.5.4.1.元素访问

top方法

​const value\_type& top ( ) const;

访问顶部元素

##### 6.5.4.2.容量

empty方法

bool empty ( ) const;

判断是否为空。

size方法

size\_type size ( ) const;

返回有效元素个数

##### 6.5.4.3.修改

operator=

priority\_queue<T, Container>&operator=( const priority\_queue <T,Container>& other );

priority\_queue<T, Container>&operator=( priority\_queue <T,Container>&& other );

push方法

void push ( const T& x );

在队尾插入元素

pop方法

void pop ( );

移除队首的元素。

emplace方法

template< class... Args >void emplace( Args&&... args );

在队首插入元素。

swap方法

void swap( stack& other );

交换容器的内容。

# C++11之迭代器库

## 1.迭代器

### 1.1.迭代器的概念

#### 1.1.1.定义

定义代器提供对集合（容器）元素的操作能力。迭代器提供的基本操作就是访问和遍历，这构成了迭代器的需求。在C++中，迭代器的需求称为Iterator需求：\*r成立和++r成立

#### 1.1.2分类

一共支持**五种**迭代器： 输入迭代器（InputIterator）、输出迭代器（OutputIterator）、正向迭代器（ForwardIterator）、双向迭代器（BidirectionalIterator）、随机访问迭代器（RandomAccessIterator）。

常用public成员函数根据具体的性质基本一致。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **别类** | | | | **属性** | **有效表达式** |
| 所有类别 | | | | 支持拷贝构造、拷贝赋值及析构 | X b(a); b = a; |
| 可以递增 | ++a; a++; |
| 随机访问 | 双向 | 正向 | 输入 | 支持等式比较或不等式比较 | a == b; a != b; |
| 可以作为右值而被解引用 | \*a; a->m; |
| 输出 | 可以作为左值而被解引用 | \*a = t; \*a++ = t; |
|  | 支持默认构造函数 | X a; X(); |
| 多道：解引用及递增都不会影响解引用性 | { b=a; \*a++; \*b; } |
|  | | 可以递减 | --a; a--; \*a--; |
|  | | | 支持算述操作符函数operator+及operator- | a + n; n + a; a - n; a – b; |
| 支持两个迭代器间的不等式比较 | a < b; a > b; a <= b; a >= b |
| 支持赋值操作符operator+=及operator-= | a += n; a -= n |
| 支持偏移解引用操作符operator[] | a[n] |

#### 1.1.3定义对象

模板：

迭代器类型 迭代器名

|  |  |
| --- | --- |
| **迭代器原语** | **意义** |
| i[terator\_traits](http://classfoo.com/ccby/article/2wr53q) | 为查询迭代器信息提供了统一的接口（Interface） |
| [input\_iterator\_tag](http://classfoo.com/ccby/article/gYSi3r) | 这是一个空类，表明迭代器类型为输入迭代器，是一个标签类型（Tag type） |
| [output\_iterator\_tag](http://classfoo.com/ccby/article/4N5U3s) | 这是一个空类，表明迭代器类型为输出迭代器，是一个标签类型 |
| [forward\_iterator\_tag](http://classfoo.com/ccby/article/xNT63t) | 这是一个空类，表明迭代器类型为正向迭代器，是一个标签类型 |
| [bidirectional\_iterator\_tag](http://classfoo.com/ccby/article/7wYF3u) | 这是一个空类，表明迭代器类型为双向迭代器，是一个标签类型 |
| [random\_access\_iterator\_tag](http://classfoo.com/ccby/article/Vfuc3v) | 这是一个空类，表明迭代器类型为随机访问迭代器，是一个标签类型 |
| [iterator](http://classfoo.com/ccby/article/zIeqik) | 迭代器类型的基类 |
| **迭代器适配器** | **意义** |
| [reverse\_iterator](http://classfoo.com/ccby/article/RDvFd0) | 用于逆序遍历的迭代器适配器（Adaptor） |
| [move\_iterator](http://classfoo.com/ccby/article/IeQler)（C++11） | 用于解引用到一个右值引用的迭代器适配器 |
| [make\_move\_iterator](http://classfoo.com/ccby/article/E2eR3w)（C++11） | 创建一个move\_iterator |
| [back\_insert\_iterator](http://classfoo.com/ccby/article/27iTec) | 用于在容器末尾插入元素的迭代器适配器 |
| [back\_inserter](http://classfoo.com/ccby/article/Fja1eg) | 创建一个back\_insert |
| [front\_insert\_iterator](http://classfoo.com/ccby/article/bGAIeh) | 用于在容器开头插入元素的迭代器适配器 |
| [front\_inserter](http://classfoo.com/ccby/article/EK9kel) | 创建一个front\_insert\_iterator |
| [insert\_iterator](http://classfoo.com/ccby/article/saVoem) | 用于在容器任意位置插入元素的迭代器适配器 |
| [inserter](http://classfoo.com/ccby/article/XqEqeq) | 创建一个insert\_iterator |
| **流迭代器** | **意义** |
| [istream\_iterator](http://classfoo.com/ccby/article/EN7AeC) | 用于从basic\_istream中读取数据的输入（Input）迭代器 |
| [ostream\_iterator](http://classfoo.com/ccby/article/jCJOeD) | 用于写入数据到basic\_ostream的输出（Output）迭代器 |
| [istreambuf\_iterator](http://classfoo.com/ccby/article/khZmeE) | 用于从basic\_streambuf中读取数据的输入迭代器 |
| [ostreambuf\_iterator](http://classfoo.com/ccby/article/IzgheJ) | 用于写入数据到basic\_streambuf的输出迭代器 |

### 1.2.常用public成员函数（基本迭代器）

#### 1.2.1.迭代器操作(非成员函数)

advance方法

void advance (InputIterator& it, Distance n);

以指定距离向前移动（Advance）一个迭代器。

distance方法

difference\_type​ distance (InputIterator first, InputIterator last);

返回两个迭代器之间的距离

next方法

ForwardIterator next (ForwardIterator it, difference\_type n = 1);

递增一个迭代器。。

prev方法

BidirectionalIterator prev (BidirectionalIterator it, difference\_type n = 1);

递减一个迭代器。

#### 1.2.2.获取迭代器

容器支持基本迭代器迭代器分为iterator和const\_iterator。普通迭代器可以用于修改原内容，而常迭代器不能。常迭代器是一个指向常内容的迭代器。就像 容器::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 容器 对象未被 const 限定。

迭代器还可分为顺序迭代器和逆序迭代器。逆序迭代器相当于把当前向量反向后视为顺序迭代器。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。

使用迭代器访问某个元素类似使用指针访问某个元素。可以简单地认为迭代器就是指针。

将普通迭代器赋值给常迭代器没有问题，但不能将常迭代器赋值给普通迭代器。因此，当函数参数有const修饰时，迭代器要使用常迭代器。

##### 1.2.2.1.普通迭代器

begin方法

iterator begin() noexcept;

const\_iterator begin() const noexcept;

返回指向当前 容器 对象中第一个元素的迭代器。如果 容器 对象是 const 限定的，函数返回类型为 const \_iterator，否则返回iterator。区别于 容器::front，当前函数返回的是一个随机访问迭代器（Random access iterator），而 容器::front 返回的是对第一个元素的引用（Reference）。容器::begin 和 容器::end 两个函数一起使用所代表的范围正好包含一个容器中所有的元素。

end方法

iterator end() noexcept;

const\_iterator end() const noexcept;

返回指向当前 容器 对象中末尾之后（ Past-the-end）的元素的迭代器。如果 容器 对象是 const 限定的，函数返回类型为 cons \_iterator，否则返回iterator。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。容器::begin 和 容器::end 两个函数一起使用所代表的范围正好包含一个容器所有的元素。如果当前容器是空的，该函数的返回值跟 容器::begin 相同。

rbegin方法

reverse\_iterator rbegin() noexcept;

const\_reverse\_iterator rbegin() const noexcept;

返回指向当前 容器 对象中最后一个元素的逆序迭代器。如果 容器 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。逆序迭代器是反方向迭代的：增加它们的值使它们指向更靠近容器开头的元素。容器::rbegin 返回值指向的元素刚好是 容器::end 返回值指向的元素的前一个元素。区别于 容器::back，当前函数返回的是指向最后一个元素的逆序随机访问迭代器（Reverse random access iterator），而 容器::back 返回指向最后一个元素的引用（Reference）。容器::rbegin 和 容器::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

rend方法

reverse\_iterator rend() noexcept;

const\_reverse\_iterator rend() const noexcept;

返回指向当前 容器 对象中第一个元素之前（ Preceding the first）的元素的逆序迭代器，如果 容器 对象是 const 限定的，函数返回类型为 const\_reverse\_iterator，否则返回 reverse\_iterator。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。容器::rbegin 和 容器::rend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

##### 1.2.2.2.常迭代器

cbegin方法（C++11）

const\_iterator cbegin() const noexcept;

返回指向当前 容器 对象中第一个元素的常迭代器（Const iterator）。const\_iterator 是一个指向常内容的迭代器。就像 容器::begin 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能被用来修改它所指向的内容，即使指向的 容器 对象未被 const 限定。如果当前容器是空的，就不能解引用（Dereferenced）返回的迭代器值。容器::cbegin 和 容器::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

cend方法（C++11）

const\_iterator cend() const noexcept;

返回指向当前 容器 对象中末尾之后（ Past-the-end）的元素的常迭代器（Const iterator）。末尾之后的元素是容器中最后一个元素之后的元素，是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。const\_iterator 是一个指向常内容的迭代器。就像 容器::end 返回的普通迭代器（Iterator）一样，该迭代器也能够被增加或减小。但是它不能用来修改它指向的内容，即使指向的 容器 对象未被 const 限定。如果当前容器是空的，该函数的返回值跟 容器::cbegin 相同。容器::cbegin 和 容器::cend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crbgin方法（C++11）

const\_reverse\_iterator crbegin() const noexcept;

返回指向当前 容器 对象中最后一个元素的常逆序迭代器。容器::crbegin 和 容器::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

crend方法（C++11）

const\_reverse\_iterator crend() const noexcept;

返回指向当前 容器 对象中第一个元素之前（ Preceding the first）的元素的常逆序迭代器。第一个元素之前的元素是一个“理论上的元素”，实际并不存在，因此指向它的迭代器不能够被解引用（Dereferenced）。容器::crbegin 和 容器::crend 两个函数一起使用所代表的范围正好包含一个容器所有的元素。

## 2.迭代器适配器

### 2.1.reverse\_iterator

模板

template <class Iterator> class reverse\_iterator;

用于逆序遍历的迭代器适配器（Adaptor），它从已知适配器定义的顺序序列的尾部迭代到头部。

构造函数

reverse\_iterator();//(1)

constexpr reverse\_iterator();//(1)

explicit reverse\_iterator( Iterator x );//(2)

constexpr explicit reverse\_iterator( Iterator x );//(2)

template< class U > reverse\_iterator( const reverse\_iterator<U>& other );//(3)

template< class U > constexpr reverse\_iterator( const reverse\_iterator<U>& other );//(3)

### 2.2.move\_iterator

模板

template <class Iterator> class move\_iterator;

用于解引用（Dereference）到一个右值引用（Rvalue reference）的迭代器适配器，强制移动语义。

构造函数

move\_iterator();

constexpr move\_iterator();

explicit move\_iterator( Iterator x );

constexpr explicit move\_iterator( Iterator x );

template< class U > move\_iterator( const move\_iterator<U>& other );

template< class U > constexpr move\_iterator( const move\_iterator<U>& other );

### 2.3.back\_insert\_iterator

模板

template <class Container> class back\_insert\_iterator;

用于在容器末尾插入元素的迭代器适配器。只用于有push\_back方法的容器。

构造函数

explicit back\_insert\_iterator( Container& c );

### 2.4.front\_insert\_iterator

模板

template <class Container> class front\_insert\_iterator;

用于在容器开头插入元素的迭代器适配器。只用于有push\_front方法的容器。

构造函数

explicit front\_insert\_iterator( Container& c );

### 2.5.insert\_iterator

模板

template <class Container>class insert\_iterator;

用于在容器任意位置插入元素的迭代器适配器。只用于有insert方法的容器。

构造函数

explicit insert\_iterator( Container& c, typename Container::iterator i );

### 2.6.迭代器适配器的创建函数

注意：4种迭代器适配器都可以直接定义，也可以调用创建函数。

make\_move\_iterator

template< class Iterator >std::move\_iterator<Iterator> make\_move\_iterator( const Iterator& i );

template< class Iterator >std::move\_iterator<Iterator> make\_move\_iterator( Iterator i );

template< class Iterator >constexpr std::move\_iterator<Iterator> make\_move\_iterator( Iterator i );

创建一个 move\_iterator。

front\_inserter

template< class Container >std::front\_insert\_iterator<Container> front\_inserter( Container& c );

创建一个 front\_insert\_iterator

back\_inserter

template< class Container >std::back\_insert\_iterator<Container> back\_inserter( Container& c );

创建一个back\_insert\_iterator。

inserter

template< class Container >

std::insert\_iterator<Container> inserter( Container& c, typename Container::iterator i );

创建一个 insert\_iterator

### 2.7.流迭代器

#### 2.7.1.istream\_iterator

模板

template <class T, class charT=char, class traits=char\_traits<charT>, class Distance = ptrdiff\_t>

class istream\_iterator;

用于从 basic\_istream 中读取数据的输入（Input）迭代器。

构造函数：

constexpr istream\_iterator();

istream\_iterator( istream\_type& stream );

istream\_iterator( const istream\_iterator& other ) = default;

#### 2.7.2.ostream\_iterator

模板

template <class T, class charT=char, class traits=char\_traits<charT>>class ostream\_iterator

用于向到 basic\_ostream写入数据的输出（Output）迭代器。

构造函数：

stream\_iterator(ostream\_type& stream, const CharT\* delim)(1)

ostream\_iterator(ostream\_type& stream)(2)

#### 2.7.3.istreambuf\_iterator

模板

template <class charT, class traits=char\_traits<charT>​>class istreambuf\_iterator;

用于从 basic\_streambuf 中读取数据的输入迭代器。

构造函数：

streambuf\_iterator();

constexpr istreambuf\_iterator();

istreambuf\_iterator( std::basic\_istream<CharT,Traits>& is );

istreambuf\_iterator( std::basic\_streambuf<CharT,Traits>\* s );

istreambuf\_iterator( const istreambuf\_iterator& ) = default;

#### 2.7.4.ostreambuf\_iterator

模板

template <class charT, class traits=char\_traits<charT>>class ostreambuf\_iterator;

用于向basic\_streambuf 写入数据的输出迭代器。

构造函数：

ostreambuf\_iterator( streambuf\_type\* buffer );(1)

ostreambuf\_iterator( ostream\_type& stream );(2)

# C++11工具库

## 1.动态内存管理

### 1.1.智能指针(Smart Pointer)

智能指针是存储指向堆对象指针的类,**且只能托管堆指针。**

在面对异常的时候格外有用，因为他们能够确保正确的销毁动态分配的对象。（防止内存泄漏）

C++11提供了以下几种智能指针,位于头文件<memory>，它们都是类模板

std::auto\_ptr(复制/赋值)、std::unique\_ptr c++11、std::shared\_ptr c++11、std::weak\_ptr c++11

**使用智能指针的原则：**一个裸指针只托管给一个智能指针，托管以后的所有操作都通过智能指针进行。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1-auto\_ptr，2-unique\_ptr，3-shared\_ptr，4-weak\_ptr | | | | | | |
|  | 方法 | 1 | 2 | 3 | 4 | 功能 |
| 成  员  函  数 | get | ✓ | ✓ | ✓ |  | 获取托管的指针 |
| get\_deleter |  | ✓ |  |  | 返回指向托管对象的销毁或析构函数的指针。 |
| reset | ✓ | ✓ | ✓ | ✓ | 修改托管的指针为p,并销毁原内存空间 |
| release | ✓ | ✓ |  |  | 释放托管的指针，但不销毁内存空间 |
| unique |  |  | ✓ |  | 当前智能指针是否有拷贝 |
| owner\_before |  |  | ✓ | ✓ | 按照owner-based order比较两个智能指针 |
| swap |  | ✓ | ✓ | ✓ | 交换托管的指针和销毁函数 |
| use\_count |  |  | ✓ | ✓ | 返回当前智能指针有多少份拷贝 |
| expired |  |  |  | ✓ | 托管指针是否已被销毁 |
| lock |  |  |  | ✓ | 返回weak\_ptr对应的shared\_ptr |
| operator \* | ✓ | ✓ | ✓ |  | 访问内容 |
| operator -> | ✓ | ✓ | ✓ |  | 访问成员 |
| operator [] |  | ✓ | ✓ |  | 访问数组元素 |
| operator = | ✓ | ✓ | ✓ | ✓ | 一个智能指针向另一个智能指针赋值 |
| operator bool |  | ✓ | ✓ |  | 判断智能指针是否为空，也就是智能指针也可作为布尔表达式 |
| operator auto\_ptr | ✓ |  |  |  | 自动将派生对象的auto\_ptr转为基类对象的auto\_ptr |
| operator auto\_ptr\_ref | ✓ |  |  |  | 自动将临时对象转为auto\_ptr形式引用 |
| 非  成  员  函  数 | std::swap |  | ✓ | ✓ | ✓ | 交换托管的指针 |
| make\_unique |  | ✓ |  |  | 构造一个对象并托管到unique\_ptr |
| hash |  | ✓ | ✓ |  | 获取unique\_ptr<T, Deleter>的哈希值。 |
| 关系运算符 |  | ✓ | ✓ |  | 返回托管的指针所指对象的比较结果 |
| *\**\_pointer\_cast |  |  | ✓ |  | 强制转换share\_ptr的类型 |
| std::get\_deleter |  |  | ✓ |  | 返回指向托管指针所指对象的销毁或析构函数的指针。 |
| make\_shared |  |  | ✓ |  | 造一个对象并托管到shared\_ptr，指明构造参数 |
| allocate\_shared |  |  | ✓ |  | 构造一个对象并托管到shared\_ptr，指明分配器和构造参数 |
| 原子操作 |  |  | ✓ |  |  |
| operator<< |  |  | ✓ |  | 输出托管的指针,等效于<<shp.get() |
| 备注 | 黄色为通用，蓝色为独有 | | | | | |

#### 1.1.1.auto\_ptr（弃用）

auto\_ptr在构造时获取对某个对象的所有权(ownership),在析构时释放该对象,要求其对“裸”指针的完全占有性,在拷贝构造或赋值操作时,会发生所有权的转移。

**本身存在缺陷,通常不使用。**

（1）**不要使用auto\_ptr对象保存指向静态分配对象的指针**。否则，当auto\_ptr对象本身被撤销时，它将试图删除指向非动态分配对象的指针，导致未定义的行为。

int a=1;

auto\_ptr<int> ap(&a); //编译没有问题，会导致未定义行为

（2）**不要把一个指针托管给多个智能指针**。

auto\_ptr<int> ap1(new int (1024));

auto\_ptr<int> ap2(ap1.get());

（3）**不要使用auto\_ptr对象保存指向动态数组的指针**。从源代码中可以看出，它用的是delete操作符，而不是delete [ ] 操作符

（4）**不要将auto\_ptr对象存储在容器中**。因为auto\_ptr的复制和赋值具有破坏性。不满足容器要求：复制或赋值后，两个对象必须具有相同值。

（5）构造函数的explicit关键词有效**阻止从一个“裸”指针隐式转换成auto\_ptr类型**。

（6）**两个auto\_ptr对象赋值或者拷贝时，原对象会失效**。

##### 1.1.1.1.模板

template< class T > class auto\_ptr;

##### 1.1.1.2.构造函数

explicit auto\_ptr( X\* p = 0 );//创建

auto\_ptr( auto\_ptr& r );//拷贝

template< class Y >auto\_ptr( auto\_ptr<Y>& r );//拷贝派生对象的auto\_ptr

template< class Y >auto\_ptr( auto\_ptr\_ref<Y> m );// 用临时对象的指针构造

##### 1.1.1.3.成员函数

operator=

auto\_ptr& operator=( auto\_ptr& r );

template< class Y >auto\_ptr& operator=( auto\_ptr<Y>& r ); //用派生对象的auto\_ptr赋值

auto\_ptr& operator=( auto\_ptr\_ref m );// 用临时对象的指针赋值

类型转换

template< class Y >operator auto\_ptr\_ref<Y>();//自动将临时对象转为auto\_ptr形式引用

template< class Y >operator auto\_ptr<Y>();//自动将派生对象的auto\_ptr转为基类对象的auto\_ptr

get

T\* get() const;

返回持有的指针

operator \*, operator ->

T& operator\*() const;

T\* operator->() const;

访问成员或内容

reset

void reset( T\* p = 0 );

释放之前持有的指针，重置该内存空间，并修改持有的指针为p,

release

T\* release();

释放持有的指针,但不销毁对象。

#### 1.1.2.unique\_ptr

unique\_ptr是一个独享某个对象的所有权的智能指针，它提供了一种严格语义上的所有权，

（1）完全拥有它所指向的对象

（2）**无法进行拷贝复制、拷贝赋值操作，但可转移复制，转移赋值**。

（3）保存指向某个对象的指针，当它本身被删除释放的时候，会使用给定的删除器释放它指向的对象

（4）使用其移动(std::move)语义时候可做为容器元素

（5）可以托管动态数组

##### 1.1.2.1.模板

template<class T, class Deleter = std::default\_delete<T>> class unique\_ptr;//单个对象

template <class T, class Deleter> class unique\_ptr<T[], Deleter>;//数组

##### 1.1.2.2.构造函数

constexpr unique\_ptr() noexcept;// default (1)

constexpr unique\_ptr (nullptr\_t) noexcept : unique\_ptr() {}/ /from null pointer (2)

explicit unique\_ptr (pointer p) noexcept;// from pointer (3)

unique\_ptr (pointer p,typename conditional<is\_reference< Deleter >::value, Deleter const Deleter &> del)

noexcept;// frompointer + lvalue deleter (4)

unique\_ptr (pointer p,typename remove\_reference< Deleter >::type&& del) noexcept;

// from pointer + rvalue deleter (5)

unique\_ptr (unique\_ptr&& x) noexcept;// move (6)

template <class U, class E>

unique\_ptr (unique\_ptr<U,E>&& x) noexcept;// move-cast (7)

template <class U>unique\_ptr (auto\_ptr<U>&& x) noexcept;// move from auto\_ptr (8)

unique\_ptr (const unique\_ptr&) = delete;// copy (deleted!) (9)

##### 1.1.2.3.成员函数

operator=

unique\_ptr& operator= (unique\_ptr&& x) noexcept;// move assignment (1)

unique\_ptr& operator= (nullptr\_t) noexcept;// assign null pointer (2)

template <class U, class E>

unique\_ptr& operator= (unique\_ptr<U,E>&& x) noexcept;// type-cast assignment (3)

unique\_ptr& operator= (const unique\_ptr&) = delete;// copy assignment (deleted!) (4)

类型转换

explicit operator bool() const;

是否拥有一个对象，是返回true否则返回false

get

pointer get() const;

返回指向托管对象的指针。

get\_deleter

Deleter& get\_deleter();(since C++11)

const Deleter& get\_deleter() const;

返回指向托管对象销毁器+或析构函数。

operator \*,

typename std::add\_lvalue\_reference<T>::type operator\*() const;

访问托管对象内容

operator ->

pointer operator->() const;

访问托管对象成员。

operator[]

T& operator[](size\_t i) const;

访问托管的数组的元素

swap

void swap(unique\_ptr& other);(since C++11)

交换托管的对象和销毁函数

reset

void reset (pointer p = pointer()) noexcept;

销毁目前由unique\_ptr（如果有的话）管理的对象，托管p的所有权。如果p是空指针（例如默认初始化指针），则unique\_ptr变为空，在调用后不管对象。要释放存储的指针的所有权而不破坏它，请改用成员函数release释放。

release

poiner release();

释放持有的指针，返回nullptr.不会销毁空间。

##### 1.1.2.4.非成员函数

swap

template< class T, class Deleter >void swap( unique\_ptr<T,Deleter>& lhs, unique\_ptr<T,Deleter>& rhs );

交换托管的指针

>,>=,<,<=,==,!=

返回托管对象的比较结果

hash

template<class T, class Deleter> struct hash<unique\_ptr<T, Deleter>>;

获取unique\_ptr<T, Deleter>的哈希值。

make\_unique(c++14)

template< class T, class... Args >unique\_ptr<T> make\_unique( Args&&... args );(用于单个)

template< class T >unique\_ptr<T> make\_unique( std::size\_t size );(用于不指明大小的数组)

template< class T, class... Args >/\* unspecified \*/ make\_unique( Args&&... args ) = delete;

构造一个对象并托管到unique\_ptr

#### 1.1.3.shared\_ptr

shared\_ptr是一个引用计数智能指针，用于共享某个对象的所有权。

（1）引进了一个计数器shared\_count,用来表示当前有多少个智能指针对象共享指针指向的内存块

（2）析构函数中不是直接释放指针对应的内存块,如果shared\_count大于0则不释放内存只是将引用计数减1,只有计数等于0时释放内存

（3）复制构造与赋值操作符只是提供一般意义上的复制功能（浅拷贝）,并且将引用计数加1.

（4）share\_ptr可降级为weak\_ptr。

（5）auto\_ptr,unique\_ptr可直接向shared\_ptr赋值

问题：循环引用,造成内存泄漏

解决方案：利用share\_ptr可直接向weak\_ptr赋值的特点，将一个类中的share\_ptr成员换成weak\_ptr。

##### 1.1.3.1.模板

template< class T > class shared\_ptr;

##### 1.1.3.2.构造函数

constexpr shared\_ptr() noexcept;// default (1)

constexpr shared\_ptr(nullptr\_t) : shared\_ptr() {}// from null pointer (2)

template <class U> explicit shared\_ptr (U\* p);// from pointer (3)

template <class U, class D> shared\_ptr (U\* p, D del);// with deleter (4)//

template <class D> shared\_ptr (nullptr\_t p, D del); // with deleter (4)

template <class U, class D, class Alloc> shared\_ptr (U\* p, D del, Alloc alloc); //with allocator (5)

template <class D, class Alloc> shared\_ptr (nullptr\_t p, D del, Alloc alloc); //with allocator (5)

shared\_ptr (const shared\_ptr& x) noexcept; //copy (6)

template <class U> shared\_ptr (const shared\_ptr<U>& x) noexcept;// copy (6)

template <class U> explicit shared\_ptr (const weak\_ptr<U>& x); //copy from weak (7)

shared\_ptr (shared\_ptr&& x) noexcept; //move (8)

template <class U> shared\_ptr (shared\_ptr<U>&& x) noexcept; //move (8)

template <class U> shared\_ptr (auto\_ptr<U>&& x);// move from auto\_ptr (9)

template <class U, class D> shared\_ptr (unique\_ptr<U,D>&& x);// move from unique\_ptr (9)

template <class U> shared\_ptr (const shared\_ptr<U>& x, element\_type\* p) noexcep// aliasing (10)

(1), (2)创建一个空对象

(3)将指针p托管给对象，置引用计数为1.

(4)和（3）一样，但求提供指出销毁函数，实际模板中D没有实现，但构造参数可以给出

(5)和（4）一样，但要求提供分配器，实际模板中alloc没有实现，但构造参数可以给出

(6)如果x不为空，则共享指针，引用计数加1；如果为空则创建一个对象

(7)和（6）一样，但如果x已经expired，抛出bad\_weak\_ptr异常

(8)转义x所管理的指针，x变成空

(9)从其他auto\_ptr和unique\_ptr接管指针

(10)和6一样，但只是x的一个别名，与x相同的引用计数，当他释放指针时，x也释放。

注意：托管数组需要制定删除器

##### 1.1.3.3.成员函数

operator=

shared\_ptr& operator=( const shared\_ptr& r );//copy

template< class Y > shared\_ptr& operator=( const shared\_ptr<Y>& r );//copy

shared\_ptr& operator=( shared\_ptr&& r );//move

template< class Y > shared\_ptr& operator=( shared\_ptr<Y>&& r );//move

template< class Y > shared\_ptr& operator=( std::auto\_ptr<Y>&& r );// move from

template< class Y, class Deleter > shared\_ptr& operator=( std::unique\_ptr<Y,Deleter>&& r ); //move from

类型转换

explicit operator bool() const;

是否拥有一个对象，是返回true否则返回false

get

T\* get() const;

element\_type\* get() const;

返回指向托管对象的指针。

use\_count

long use\_count() const;

返回共享当前对象的shared\_ptr个数，只有拷贝才会增加。新建或者移动都不会。

unique

bool unique() const noexcept;

如过当前托管对象只有当前一个shared\_ptr，返回true，否则返回false。只有拷贝才为false。新建或者移动都不会。

owner\_before

template< class Y > bool owner\_before( const shared\_ptr<Y>& other) const;

template< class Y >bool owner\_before( const std::weak\_ptr<Y>& other) const;

用当前指针与other比较：如果同为空或者同时指向同一个对象（包含继承关系,新建立不会，拷贝才会），就返回false;如果是其它情况，则用指针所指向的对象的地址来比较大小，若当前指针的地址<other的地址，则返回true，否则返回false。

operator \*, operator ->

T& operator\*() const;

T\* operator->() const;

访问托管对象的成员或内容。

operator[]（C++17）

element\_type& operator[]( std::ptrdiff\_t idx )

访问托管的数组的元素

swap

void swap( shared\_ptr& r );

交换托管的对象和销毁函数

reset

void reset();(1)

template< class Y > void reset( Y\* ptr );

template< class Y, class Deleter > void reset( Y\* ptr, Deleter d );

template< class Y, class Deleter, class Alloc > void reset( Y\* ptr, Deleter d, Alloc alloc );

释放之前持有的指针，并修改持有的指针为p,可以指定删除器和分配器

##### 1.1.3.4.非成员函数

get\_deleter

template< class Deleter, class T >Deleter\* get\_deleter( const std::shared\_ptr<T>& p );

返回指向托管对象销毁或析构函数的指针。

swap

template< class T >void swap( shared\_ptr<T>& lhs, shared\_ptr<T>& rhs );

交换托管的指针

>,>=,<,<=,==,!=

返回托管对象的比较结果

hash

template<class T> struct hash<shared\_ptr<T>>;

获取unique\_ptr<T, Deleter>的哈希值。

make\_shared

template< class T, class... Args >shared\_ptr<T> make\_shared( Args&&... args );

构造一个对象并托管到shared\_ptr,args为构建T所需的参数

allocate\_shared

template< class T, class Alloc, class... Args >shared\_ptr<T> allocate\_shared( const Alloc& alloc, Args&&...

args );

构造一个对象并托管到shared\_ptr.指明分配器和构建T所需的参数args。

static\_pointer\_cast, dynamic\_pointer\_cast, const\_pointer\_cast, reinterpret\_pointer\_cast

template< class T, class U > std::shared\_ptr<T> static\_pointer\_cast( const std::shared\_ptr<U>& r );(1)

template< class T, class U > std::shared\_ptr<T> dynamic\_pointer\_cast( const std::shared\_ptr<U>& r );(2)

template< class T, class U > std::shared\_ptr<T> const\_pointer\_cast( const std::shared\_ptr<U>& r );(3)

template< class T, class U > std::shared\_ptr<T> reinterpret\_pointer\_cast( const std::shared\_ptr<U>& r );(4)

operator<<

template <class T, class U, class V>basic\_ostream<U, V>& operator<<(basic\_ostream<U, V>& os, const

shared\_ptr<T>& ptr);

输出托管的指针，等效于cout<<ptr.get()

原子操作std::atomic\_is\_lock\_free(std::shared\_ptr)

std::atomic\_load(std::shared\_ptr)

std::atomic\_load\_explicit(std::shared\_ptr)

std::atomic\_store(std::shared\_ptr)

std::atomic\_store\_explicit(std::shared\_ptr)

std::atomic\_exchange(std::shared\_ptr)

std::atomic\_exchange\_explicit(std::shared\_ptr)

std::atomic\_compare\_exchange\_weak(std::shared\_ptr)

std::atomic\_compare\_exchange\_strong(std::shared\_ptr)

std::atomic\_compare\_exchange\_weak\_explicit(std::shared\_ptr)

std::atomic\_compare\_exchange\_strong\_explicit(std::shared\_ptr)

#### 1.1.4.weak\_ptr

强引用，要有一个引用存在，对象就不能被释放

弱引用，并不增加对象的引用计数，但它知道对象是否存在。如果存在，提升为shared\_ptr成功；否则，提升失败。

weak\_ptr 是弱引用智能指针，而shared\_ptr是强引用智能指针

通过weak\_ptr，访问对象的成员的时候，要用lock（）方法提升为shared\_ptr。

##### 1.1.4.1.模板

template< class T > class weak\_ptr;

##### 1.1.4.2.构造函数

constexpr weak\_ptr();//(1) default

weak\_ptr( const weak\_ptr& r );//(2)copy

template< class Y > weak\_ptr( const weak\_ptr<Y>& r );//(2)copy

template< class Y > weak\_ptr( const std::shared\_ptr<Y>& r );(3) from shared\_ptr

##### 1.1.4.3.成员函数

operator=

weak\_ptr& operator= (const weak\_ptr& x) noexcept; //copy (1)

template <class U> weak\_ptr& operator= (const weak\_ptr<U>& x) noexcept; // //copy (1)

template <class U> weak\_ptr& operator= (const shared\_ptr<U>& x) noexcept;// from shared\_ptr (2)

swap

void swap( weak\_ptr& r );

交换托管的对象和销毁函数

reset

void reset （）;

释放托管对象

use\_count

long use\_count() const;

返回共享当前对象的weak\_ptr个数

expired

bool expired() const;

如果托管对象已经被销毁返回true，否则返回false

lock(提升为shared\_ptr)

std::shared\_ptr<T> lock() const;

返回weak\_ptr对应的shared\_ptr，共享对象的所有权。如果没有托管对象，即\*this是空的，那么返回的shared\_ptr也是空的。

owner\_before

template< class Y > bool owner\_before( const weak\_ptr<Y>& other) const;

template< class Y >bool owner\_before( const std::shared\_ptr<Y>& other) const;

用当前指针与other比较：如果同为空或者同时指向同一个对象（包含继承关系,新建立不会，拷贝才会），就返回false;如果是其它情况，则用指针所指向的对象的地址来比较大小，若当前指针的地址<other的地址，则返回true，否则返回false。

##### 1.1.4.4.非成员函数

swap

template< class T >void swap( weak\_ptr<T>& lhs, weak\_ptr<T>& rhs );

交换托管的指针

### 1.2.辅助类

#### 1.2.1.default\_delete

模板

template< class T > struct default\_delete

template< class T > struct default\_delete<T[]>

构造函数

constexpr default\_delete() noexcept = default;

template <class U, class = typename enable\_if<is\_convertible<U\*,T\*>::value>::type>

default\_delete (const default\_delete<U>& del) noexcept {}

#### 1.2.2.bad\_weak\_ptr

模板

class bad\_weak\_ptr;

bad\_weak\_ptr是std :: weak\_ptr作为std :: shared\_ptr的构造函数参数时，如果std :: weak\_ptr引用已经销毁对象时，发生异常抛出的对象的类型

#### 1.2.3.enable\_shared\_from\_this

模板

template< class T > class enable\_shared\_from\_this;

构造函数

constexpr enable\_shared\_from\_this();

enable\_shared\_from\_this( const enable\_shared\_from\_this<T>&obj );

operator=

enable\_shared\_from\_this<T>& operator=( const enable\_shared\_from\_this<T> &obj )

shared\_from\_this

shared\_ptr<T> shared\_from\_this();

shared\_ptr<T const> shared\_from\_this() const;

继承此基类的派生类中有shared\_from\_this成员函数。提供了允许派生类对象在类中获取指向自己的shared\_ptr并与现有shared\_ptr对象共享所有权的功能。请注意，简单地返回shared\_ptr <T>（this）将会导致重复托管问题，因为这将新建一个shared\_ptr，而引用计数不变，而用shared\_from\_this()则引用计数会加一，相当于拷贝。

#### 1.2.4.owner\_less

模板

template< class T >struct owner\_less; /\* undefined \*/

### 1.3.内存分配器allocator

allocator是定义在<mempry>标准库的内存分配模板类，是STL容器没有指定内存分配器时使用的默认内存分配器。它提供可感知类型的内存分配，这个类支持一个抽象接口以分配内存，并随后使用该内存保存对象。

allocator只是对的底层的alloc类封装，真正具备空间分配功能的std::alloc。

#### 1.3.1.模板

template <class T> class allocator;

#### 1.3.2.构造函数

allocator() noexcept;

allocator (const allocator& alloc) noexcept;

template <class U>allocator (const allocator<U>& alloc) noexcept

#### 1.3.3.成员函数

construct

template <class T, class... Args>void construct (T\* p, Args&&... args);

在T\*型指针p所指原始内存中构造一个新元素，运行T类型的拷贝构造函数用args初始化该对象。并不会分配内存，p所指内存应已经存在。construct只负责构造对象，不负责分配原始内存。

destroy

template <class T>void destroy (T\* p);

运行T\*指针p所指对象的析构函数。销毁p所指向的对象。注意，不会释放元素的存储空间。destroy只负责销毁对象，不负责回收原始内存。

allocate

pointer allocate (size\_type n, allocator<void>::const\_pointer hint=0);

分配原始未构造内存用以保存n个T类型对象，但并不构建对象。返回一个指向第一个元素的指针。如果它不能分配请求的存储总量，则抛出bad\_alloc的异常。allocate只负责分配原始内存，不负责构造对象。

deallocate

void deallocate (pointer p, size\_type n);

释放先前分配的但尚未释放的内存块，在T\*指针p中包含的地址处保存n个T类型对象。运行deallocate之前在该内存中构造的对象需要手动销毁。已构造的对象不会因为该函数的调用而销毁。deallocate只负责回收allocate分配的原始内存，不负责销毁对象。

address

pointer address ( reference x ) const noexcept;

const\_pointer address ( const\_reference x ) const noexcept;

返回x的地址。在标准默认分配器中，返回&X

max\_size

size\_type max\_size() const noexcept;

返回可以分配的最大大小，返回元素的最大数量，每个元素的类型为value\_type（分配器模板参数的别名）。

### 1.4.内存资源（c++17）

### 1.5.未初始化内存操作

#### 1.5.1.原始存储迭代器：

class raw\_storage\_iterator;

template <class OutputIterator, class T>class raw\_storage\_iterator;

该迭代器类对未初始化的内存块进行操作。常规迭代器对已经构建的某种类型的对象进行操作。一个raw\_storage\_iterator将这些常规迭代器中的一个包装到一个特殊的输出迭代器中，该迭代器在写入前，在指向的位置构造对象。

#### 1.5.2.临时缓冲区

get\_temporary\_buffer

template <class T>pair <T\*,ptrdiff\_t>get\_temporary\_buffer ( ptrdiff\_t n ) noexcept;

请求一个临时的存储块，最多可以包含多达n个T型元素。基本对齐方式，则该块存储空间适当地对齐以包含类型T的元素，尽管它未被初始化（没有对象被构造）。 该功能专门设计用于获得临时性的存储器（例如用于算法的操作）。一旦内存块不再需要了，它将通过调用return\_temporary\_buffer来释放。 请注意，返回的缓冲区的实际大小可能小于请求的实际大小（实际大小表示为返回值的一部分）

return\_temporary\_buffer

template <class T> void return\_temporary\_buffer (T\* p);

释放p指向的内存块。

p将是由前一次调用get\_temporary\_buffer返回的指针值。

#### 1.5.3.专用算法

uninitialized\_copy

template< class InputIt, class ForwardIt >

ForwardIt uninitialized\_copy( InputIt first, InputIt last, ForwardIt d\_first );

返回指向最后一个已复制的元素的迭代器，将[first, last)的元素复制到以d\_dirst开始的未初始化的存储区域。

uninitialized\_copy\_n

template< class InputIt, class Size, class ForwardIt >

ForwardIt uninitialized\_copy\_n( InputIt first, Size count, ForwardIt d\_first);

返回指向最后一个已复制的元素的迭代器，将从first开始的count个元素复制到以d\_dirst开始的未初始化的存储区域。

uninitialized\_fill

template< class ForwardIt, class T >

void uninitialized\_fill( ForwardIt first, ForwardIt last, const T& value );

构造范围[first，last]中的所有元素将其初始化为x的值。

uninitialized\_fill\_n

template< class ForwardIt, class Size, class T >

ForwardIt uninitialized\_fill\_n( ForwardIt first, Size count, const T& value );

无返回值或者返回指向最后填充元素的迭代器，用给定值从first开始填充n个未初始化存储区域

### 1.6.垃圾回收支持

### 1.7.C型内存管理

malloc

void\* malloc( std::size\_t size );

calloc

void\* calloc( std::size\_t num, std::size\_t size )

realloc

void\* realloc( void\* ptr, std::size\_t new\_size );

free

void free( void\* ptr );

### 1.8.低级别内存管理

#### 1.8.1.可重载new和delete

普通new，为兼容以前c++的普通new会分配空间，创建对象。

普通new、delete

void\* operator new ( std::size\_t count );

void\* operator new[]( std::size\_t count );

void operator delete ( void\* ptr );

void operator delete[]( void\* ptr );

普通兼容new、delete

void\* operator new ( std::size\_t count, const std::nothrow\_t& tag);

void\* operator new[]( std::size\_t count, const std::nothrow\_t& tag);

void operator delete ( void\* ptr, const std::nothrow\_t& tag );

void operator delete[]( void\* ptr, const std::nothrow\_t& tag );

#### 1.8.2.不可重载new和delete

定位new，类普通同new，类定位new不分配空间，只构在原始内存中建对象

普通定位new

void\* operator new ( std::size\_t count, void\* ptr );

void\* operator new[]( std::size\_t count, void\* ptr );

void operator delete ( void\* ptr, void\* place );

void operator delete[]( void\* ptr, void\* place );

自定义定位new（不一定有）

void\* operator new ( std::size\_t count, user-defined-args... );

void\* operator new[]( std::size\_t count, user-defined-args... );

void operator delete ( void\* ptr, args... );

void operator delete[]( void\* ptr, args... );

类普通new（不一定有）

void\* T::operator new ( std::size\_t count );

void\* T::operator new[]( std::size\_t count );

void T::operator delete ( void\* ptr );

void T::operator delete[]( void\* ptr );

void T::operator delete ( void\* ptr, std::size\_t sz );

void T::operator delete[]( void\* ptr, std::size\_t sz );

类定位new（不一定有）

void\* T::operator new ( std::size\_t count, user-defined-args... );

void\* T::operator new[]( std::size\_t count, user-defined-args... );

void T::operator delete ( void\* ptr, args... );

void T::operator delete[]( void\* ptr, args... );

## 2.二元组和多元组

### 2.1.二元组pair

#### 2.1.1定义

pair是一个结构模板，提供了一种将两个异构对象存储为一个单元的方法。pair是具有两个元素的std :: tuple的特殊情况。第一个成员和第二个成员分别通过first和second访问。

#### 2.1.2定义对象

##### 2.1.2.1.模板

#include<utility>

template<class T1, class T2> struct pair;

##### 2.1.2.2.构造函数

pair()//新建空对

constexpr pair();//新建空对

pair( const T1& x, const T2& y ); //以左值新建

template< class U1, class U2 >pair( U1&& x, U2&& y ); //以右值新建

template< class U1, class U2 >pair( const pair<U1, U2>& p ); //以左值拷贝

template< class U1, class U2 >pair( pair<U1, U2>&& p ); //以右值拷贝

template< class... Args1, class... Args2 >

pair( std::piecewise\_construct\_t, std::tuple<Args1...> first\_args, std::tuple<Args2...> second\_args );

pair( const pair& p ) = default; //以左值拷贝

pair( pair&& p ) = default; //以右值拷贝

##### 2.1.2.3.公有成员

first：第一个异构元素

second：第二个异构元素

#### 2.1.3.常用成员函数

operator=

pair& operator=( const pair& other );

template< class U1, class U2 >pair& operator=( const pair<U1,U2>& other );

template< class U1, class U2 >pair& operator=( pair<U1,U2>&& other );

pair& operator= (pair&& pr) noexcept(is\_nothrow\_move\_assignable<first\_type>

::value &&is\_nothrow\_move\_assignable<second\_type>::value);

swap方法

void swap (pair& pr) noexcept ( noexcept(swap(first,pr.first)) &&noexcept(swap(second,pr.second)) );

#### 2.1.4.非成员函数

make\_pair

template< class T1, class T2 >std::pair<T1,T2> make\_pair( T1 t, T2 u );

template< class T1, class T2 >std::pair<V1,V2> make\_pair( T1&& t, T2&& u );

operator==,!=,<,<=,>,>=

template< class T1, class T2 >bool operator关系符 ( const pair<T1,T2>& lhs, const pair<T1,T2>& rhs );

swap

template <class T1, class T2>void swap (pair<T1,T2>& x, pair<T1,T2>& y) noexcept

(noexcept(x.swap(y)));

get

template <size\_t I, class T1, class T2>

typename tuple\_element< I, pair<T1,T2> >::type& get (pair<T1,T2>& pr) noexcept;

template <size\_t I, class T1, class T2>

typename tuple\_element< I, pair<T1,T2> >::type&& get (pair<T1,T2>&& pr) noexcept;

template <size\_t I, class T1, class T2>

const typename tuple\_element< I, pair<T1,T2> >::type&get (const pair<T1,T2>& pr) noexcept;

参数I:异构元素在pair中的位置，0表示first，1表示second。

返回pair中指定成员的引用。

#### 2.1.5.辅助类

tuple\_size

template< class T1, class T2 >struct tuple\_size<std::pair<T1, T2>>;

tuple\_element

template< class T1, class T2 >struct tuple\_element<0, std::pair<T1,T2> >;

template< class T1, class T2 >struct tuple\_element<1, std::pair<T1,T2> >;

## 3.函数对象

头文件<functional>

### 3.1.多态函数封装

#### 3.1.1.function

##### 3.1.1.1.定义

std::function<R(T1, T2, ..., TN)>

这是一个模板实现的函数对象类，它可以包装其它任意的函数对象，而被包装的函数对象具有类型为T1,T2,…,TN的参数，其返回值为R类型。

function 对象的最大用处在于类模版std::function是一种通用、多态的函数封装。std::function的实例可以对任何可以调用的目标实体进行存储、复制、和调用操作，这些目标实体包括普通函数、Lambda表达式、函数指针、以及其它仿函数等。std::function对象是对C++中现有的可调用实体的一种类型安全的包裹（我们知道像函数指针这类可调用实体，是类型不安全的）。

通常std::function是一个函数对象类，它包装其它任意的函数对象，被包装的函数对象具有类型为T1, …,TN的N个参数，并且返回一个可转换到R类型的值。std::function使用模板转换构造函数接收被包装的函数对象；特别是，被包装的类型可以隐式地转换为std::function。

最简单的理解就是：

通过std::function对C++中各种可调用实体（普通函数、Lambda表达式、函数指针、以及其它函数对象等）的封装，形成一个新的可调用的std::function对象；让我们不再纠结那么多的可调用实体。一切变的简单粗暴。实现函数回调，等效于C中的函数指针。

##### 3.1.1.2.定义对象

###### 3.1.1.2.1.模板

template <class T> function; //undefined

template <class Ret, class... Args> class function<Ret(Args...)>;

###### 3.1.1.2.2.构造函数

function() noexcept;//默认

function( std::nullptr\_t ) noexcept;//空

function( const function& other );//拷贝

function( function&& other );//移动

template< class F > function( F f );//初始化

//以下为带内存分配器的版本

template< class Alloc >function( std::allocator\_arg\_t, const Alloc& alloc ) noexcept;

template< class Alloc > function( std::allocator\_arg\_t, const Alloc& alloc, std::nullptr\_t ) noexcept;

template< class Alloc >function( std::allocator\_arg\_t, const Alloc& alloc, const function& other );

template< class Alloc > function( std::allocator\_arg\_t, const Alloc& alloc, function&& other );

template< class F, class Alloc > function( std::allocator\_arg\_t, const Alloc& alloc, F f );

##### 3.1.1.3常用public成员函数

operator=

function& operator=( const function& other );//拷贝

function& operator=( function&& other );//移动

function& operator=( std::nullptr\_t );//清空

template< class F > function& operator=( F&& f );//右值

template< class F > function& operator=( std::reference\_wrapper<F> f );//引用

assign

template <class Fn, class Alloc>void assign (Fn&& fn, const Alloc& alloc);

swap

void swap( function& other ) noexcept;

operator bool

explicit operator bool() const noexcept;，作为bool表达式

operator ()

R operator()( Args... args ) const;

target\_type

const std::type\_info& target\_type() const noexcept;

返回存放函数的返回值类型的结构体type\_info，类似于typeid

target

template< class T > T\* target() noexcept;

template< class T > const T\* target() const noexcept;

返回指向存放的函数的指针

##### 3.1.1.4.非成员函数

std::swap

template< class R, class... Args >void swap( function<R(Args...)> &lhs, function<R(Args...)> &rhs );

operator==,operator!=

template<class R, class... ArgTypes>

bool operator==(const std::function<R(ArgTypes...)>& f, std::nullptr\_t ) noexcept;

template<class R, class... ArgTypes>

bool operator==(std::nullptr\_t, const std::function<R(ArgTypes...)>& f ) noexcept;

template<class R, class... ArgTypes>

bool operator!=(const std::function<R(ArgTypes...)>& f, std::nullptr\_t ) noexcept;

template<class R, class... ArgTypes>

bool operator!=(std::nullptr\_t, const std::function<R(ArgTypes...)>& f ) noexcept;

##### 3.1.1.5.辅助类uses\_allocator

template< class R, class... ArgTypes, class Alloc >

struct uses\_allocator<std::function<R(ArgTypes...)>, Alloc> : std::true\_type { };

#### 3.1.2.mem\_fn

template< class R, class T >mem\_fn(R T::\* pm);

template< class R, class T, class... Args > mem\_fn(R (T::\* pm)(Args...));

mem\_fn用于将类的成员函数封装成一个函数对象。

mem\_fn最为人所熟知的作用是，将一个成员函数作用在一个容器上，就像这样std::for\_each(v.begin(), v.end(), boost::mem\_fn(&Shape::draw))就可以让容器vector中的每一个元素都执行一遍draw方法。

第二个用法是，它可以帮助把一个函数指针模拟得像一个函数实体(function object)。

#### 3.1.3. bad\_function\_call

class bad\_function\_call;

bad\_function\_call是调用空的 function 对象时抛出的一个异常。

### 3.2.绑定器

#### 3.2.1 bind

template< class F, class... Args >bind( F&& f, Args&&... args );

template< class R, class F, class... Args >bind( F&& f, Args&&... args );

bind，将特定参数绑定到目标，返回一个funciton对象。

bind接收的第一个参数必须是一个可调用的对象f，有无参数信息都可以，包括函数、函数指针、函数对象、仿函数、和成员函数指针和数据成员指针，之后bind最多接受9个参数，参数数量必须与f的参数数量相等，这些参数被传递给f作为入参。 绑定完成后，bind会返回一个函数对象，它内部保存了f的拷贝，具有operator()，返回值类型被自动推导为可调用对象的返回类型。在发生调用时这个函数对象将把之前存储的参数转发给f完成调用。

对于不事先绑定的参数，需要传std::placeholders：：\_nummber进去，从\_1开始，依次递增\_2,\_3……。表示占位符所在位置的形参，将接收调用时传入的第number个实参。调用函数时，可以传入更多的参数但是没有对应占位符的参数都是无效参数。

注意:

（1）成员函数的第一参数是隐藏的this指针，必须传入一个对象，或指针，或引用。如绑定的是成员函数，一般要取地址，变为成员函数指针。

（2）如果待绑定的函数对象的形参是引用或常引用而绑定进去实参不是引用，或者实参不能进行值传递，此时要用ref或cref转换为引用，否则会发生值传递。失去原本传引用的目的的。

（3）bind的返回值可以用auto关键字自动推定。

（4）不能将占位符绑定到引用

#### 3.2.2.is\_bind\_expression

template< class T >struct is\_bind\_expression;

is\_bind\_expression用来判断一个函数对象或者bind的参数是不是由bind生成的。

#### 3.2.3. is\_placeholder

template <class T> struct is\_placeholder;

is\_placeholder用来识别bind的参数是不是一个占位符。

#### 3.2.3. placeholders

namespace placeholders

{

extern /\* unspecified \*/ \_1;

extern /\* unspecified \*/ \_2;

extern /\* unspecified \*/ \_3;

// ...

}

placeholders是一个命名空间，声明未指定数量的对象： \_1， \_2， \_3，...，用于在bind中代替未绑定的参数。当使用bind返回的函数对象来调用时，\_1被第一个实参替代，\_i被第i个实参替代。

### 3.3.引用封装

#### 3.3.1 reference\_wrapper

##### 3.3.1.1.定义

将T类型的对象转为T类型的对象的引用。

##### 3.3.1.2定义对象

###### 3.3.1.2.1模板

template< class T >class reference\_wrapper;

###### 3.3.1.2.2构造函数

reference\_wrapper( T& x ) noexcept;//可以拷贝

reference\_wrapper( T&& x ) = delete;//不具有移动语义

reference\_wrapper( const reference\_wrapper<T>& other ) noexcept;//可以赋值

##### 3.3.1.2.public成员函数

operator=

reference\_wrapper& operator=( const reference\_wrapper<T>& other ) noexcept;

operator T&，get

operator T& () const noexcept;

T& get() const noexcept;

operator()

template< class... ArgTypes >

typename std::result\_of<T&(ArgTypes&&...)>::typename operator() ( ArgTypes&&... args ) const;

#### 3.3.2.ref和cref

ref和cref

std::reference\_wrapper<T> ref(T& t);

template< class T >std::reference\_wrapper<T> ref( std::reference\_wrapper<T> t );

template <class T>void ref(const T&&) = delete;

template< class T >std::reference\_wrapper<const T> cref( const T& t );

template< class T >std::reference\_wrapper<const T> cref( std::reference\_wrapper<T> t );

template <class T>void cref(const T&&) = delete;

将类型T的对象转为T类型的引用或常引用。返回一个reference\_wrapper对象，常用在bind的目标函数的形参是引用、不能拷贝和赋值时。

### 3.4.否定器

#### 3.4.1.谓词

接受一个参数的函数，叫做一元函数，如果一元函数返回布尔值，则该函数成为一元谓词；接受两个参数的函数为二元函数，如果返回一个布尔值，则该函数称为二元谓词。

#### **3.4.2.**断言

断言就是用于在程序中捕捉某些假设条件的代码句。断言表示为对一些表达式（0代表false，非0代表true）或者谓词进行判断，程序员相信在程序中的某个特定点该表达式或谓词的值为真。

C中有宏定义assert，如果表达式的结果为零，就退出执行。

C++11有static\_assert这个关键字，它不是宏。但是他的用途是在编译期检查一些常量或者传入函数的参数是不是符合预期。

static\_assert ( bool\_constexpr , message )

#### 3.4.3.not1和not2

template< class Predicate >std::unary\_negate<Predicate> not1(const Predicate& pred);

template< class Predicate >std::binary\_negate<Predicate> not2(const Predicate& pred);

not1是返回一个与一元谓词结果相反的unary\_negate对象，not2是返回一个与二元谓词结果相反的binary\_negate对象。这里的谓词应当是函数对象，而不是普通函数。

#### 3.4.3. unary\_negate和binary \_negate

template< class Predicate >struct binary\_negate

:public std::binary\_function<Predicate::first\_argument\_type,Predicate::second\_argument\_type,bool>;

template< class Predicate >

struct unary\_negate : public std::unary\_function<Predicate::argument\_type, bool>;

template< class Predicate >struct unary\_negate;（c++11）

template< class Predicate >struct binary\_negate; （c++11）

unary\_negate与 unary\_negate将一元或二元谓词转化为其补集，由于c++11废弃了unary\_function与binary\_funciton,因此重新定义了这两个类，而c++98则中则继承自unary\_function与binary\_funciton类。

### 3.5.仿函数

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **算术操作** | | **比较操作** | | **逻辑操作** | | **按位操作** | |
| **仿函数** | **功能** | **仿函数** | **功能** | **仿函数** | **功能** | **仿函数** | **功能** |
| [plus](http://classfoo.com/ccby/article/PY6pVo)(x,y) | x + y | [equal\_to](http://classfoo.com/ccby/article/hDqTVu)(x,y) | x == y | [logical\_and](http://classfoo.com/ccby/article/pEcjVA)(x,y) | x && y | [bit\_and](http://classfoo.com/ccby/article/NcvxVD)(x,y) C++11 | x & y |
| [minus](http://classfoo.com/ccby/article/u4frVp)(x,y) | x - y | [not\_equal\_to](http://classfoo.com/ccby/article/GwcsVv)(x,y) | x != y | [logical\_or](http://classfoo.com/ccby/article/R5RZVB)(x,y) | x || y | [bit\_or](http://classfoo.com/ccby/article/8shbVE)(x,y) C++11 | x | y |
| [multiplies](http://classfoo.com/ccby/article/oqOyVq)(x,y) | x \* y | [greater](http://classfoo.com/ccby/article/98YwVw)(x,y) | x > y | [logical\_not](http://classfoo.com/ccby/article/ZBCJVC)(x) | !x | [bit\_xor](http://classfoo.com/ccby/article/YlwzVF)(x,y) C++11 | x ^ y |
| [divides](http://classfoo.com/ccby/article/qpodVr)(x,y) | x / y | [less](http://classfoo.com/ccby/article/EtdVVx)(x,y) | x < y |  |  | bit\_not(x) C++14 | ~x |
| [modulus](http://classfoo.com/ccby/article/HAyvVs)(x,y) | x % y | [greater\_equal](http://classfoo.com/ccby/article/LWs5Vy)(x,y) | x >= y | 这些仿函数实际就是重载了operator()的c++的struct类 | | | |
| [negate](http://classfoo.com/ccby/article/zu4jVt)(x) | -x | [less\_equal](http://classfoo.com/ccby/article/yWmfVz)(x,y) | x <= y |

### 3.6.已弃用的函数适配器

#### 3.6.1.unary\_function和binary\_function

template <class Arg, class Result>

struct **unary\_function**

{

typedef Arg argument\_type;

typedef Result result\_type;

};

template <class Arg, class Arg2, class Res>

struct **binary\_function**

{

typedef Arg first\_argument\_type;

typedef Arg2 second\_argument\_type;

typedef Res result\_type;

};

unary\_function和binary\_function是一元仿函数和二仿函数的基类。函数对象要求具有参数位置的信息，自定义仿函数因此需要以public方式**继承**unary\_function和binary\_function基类，当然也可以**显式定义同样的参数位置信息**。

#### 3.6.2.binder1st和binder2nd

template< class Fn >

class binder1st : public std::unary\_function<typename Fn::second\_argument\_type, typename Fn::result\_type>

{

protected:

Fn op;

typename Fn::first\_argument\_type value;

public:

binder1st(const Fn& fn, const typename Fn::first\_argument\_type& value);

typename Fn::result\_type operator()(const typename Fn::second\_argument\_type& x) const;

typename Fn::result\_type operator()(typename Fn::second\_argument\_type& x) const;

};

template< class Fn >

class binder2nd: public unary\_function<typename Fn::first\_argument\_type, typename Fn::result\_type>

{

protected:

Fn op;

typename Fn::second\_argument\_type value;

public:

binder2nd(const Fn& fn, const typename Fn::second\_argument\_type& value);

typename Fn::result\_type perator()(const typename Fn::first\_argument\_type& x) const;

typename Fn::result\_type perator()(typename Fn::first\_argument\_type& x) const;

};

binder1st和binder2nd将带有参数位置信息的二元仿函数转化为一元仿函数，分别是bind1st和bind2nd返回类型。C++11中已经被std::funciton取代。

#### 3.6.3.bind1st和bind2nd

template< class F, class T >std::binder1st<F> bind1st( const F& f, const T& x );

template< class F, class T >std::binder2nd<F> bind2nd( const F& f, const T& x );

将指定值绑定到二元函数对象(带参数位置信息)的第1个参数或第2个参数上，返回一元函数对象。

注意：bind1st和bind2nd绑定的是带有参数信息**函数对象**,普通函数或函数指针必须通过**ptr\_fun**函数转变为函数对象才能绑定，而自定义的仿函数则需要以public**继承binary\_function**基类或者在**类中显式定义同样的public参数信息**。

C++11中已经被std::bind取代。

#### 3.6.4.pointer\_to\_unary\_function和pointer\_to\_binary\_function

template<class Arg, class Result >

class **pointer\_to\_unary\_function** : public std::unary\_function<Arg, Result>;

template<class Arg1, class Arg2, class Result >

class **pointer\_to\_binary\_function** : public std::binary\_function<Arg1, Arg2, Result>;

pointer\_to\_unary\_function和pointer\_to\_binary\_function是由一元或二元函数指针转换而成的带参数位置信息的一元或二元函数对象。

#### 3.6.5.ptr\_fun

template< class Arg, class Result >

std::pointer\_to\_unary\_function<Arg,Result> **ptr\_fun**( Result (\*f)(Arg) );

template< class Arg1, class Arg2, class Result >

std::pointer\_to\_binary\_function<Arg1,Arg2,Result> **ptr\_fun**( Result (\*f)(Arg1, Arg2) );

ptr\_fun将一元或二元函数转为相应的带参数位置信息的函数对象，

#### 3.6.6.mem\_fun\_t,mem\_fun1\_t, const\_mem\_fun\_t, const\_mem\_fun1\_t

template< class S, class T > class mem\_fun\_t

template< class S, class T >

class const\_mem\_fun\_t : public unary\_function<const T\*,S>

{

public:

explicit const\_mem\_fun\_t(S (T::\*p)() const);

S operator()(const T\* p) const;

};

template< class S, class T, class Arg >

class mem\_fun1\_t : public binary\_function<T\*,A,S>

{

public:

explicit mem\_fun1\_t(S (T::\*p)(A));

S operator()(T\* p, A x) const;

};

template< class S, class T, class A >

class const\_mem\_fun1\_t : public binary\_function<const T\*,A,S>

{

public:

explicit const\_mem\_fun1\_t(S (T::\*p)(A) const);

S operator()(const T\* p, A x) const;

};

mem\_fun\_t 通过一个无参数成员函数生成一个函数对象（指针版）

mem\_fun1\_t 通过一个带一个参数的成员函数生成一个函数对象（指针版）

const\_mem\_fun\_t 通过一个 const 限定的无参数成员函数生成一个函数对象（指针版）

const\_mem\_fun1\_t 通过一个 const 限定的带一个参数的成员函数生成一个函数对象（指针版）

作为mem\_fun的返回值

#### 3.6.7.mem\_fun

template< class Res, class T > std::mem\_fun\_t<Res,T> mem\_fun( Res (T::\*f)() );

template< class Res, class T > std::const\_mem\_fun\_t<Res,T> mem\_fun( Res (T::\*f)() );

template< class Res, class T, class Arg > std::mem\_fun1\_t<Res,T,Arg> mem\_fun( Res (T::\*f)(Arg) );

template< class Res, class T, class Arg > std::const\_mem\_fun1\_t<Res,T,Arg> mem\_fun( S (T::\*f)(Arg) );

将一个不带参数或者带一个参数的成员函数转化成函数对象（指针版），只用于容器中元素是对象指针时才能用。

#### 3.6.8.mem\_fun\_ref\_t、 mem\_fun1\_ref\_t、const\_mem\_fun\_ref\_t、const\_mem\_fun1\_ref\_t

template< class S, class T >

class mem\_fun\_ref\_t : public unary\_function<T,S>

{

public:

explicit mem\_fun\_ref\_t(S (T::\*p)());

S operator()(T& p) const;

};

template< class S, class T >

class const\_mem\_fun\_ref\_t : public unary\_function<T,S>

{

public:

explicit const\_mem\_fun\_ref\_t(S (T::\*p)() const);

S operator()(const T& p) const;

};

template< class S, class T, class A >

class mem\_fun1\_ref\_t : public binary\_function<T,A,S>

{

public:

explicit mem\_fun1\_ref\_t(S (T::\*p)(A));

S operator()(T& p, A x) const;

};

template< class S, class T, class A >

class const\_mem\_fun1\_ref\_t : public binary\_function<T,A,S>

{

public:

explicit const\_mem\_fun1\_ref\_t(S (T::\*p)(A) const);

S operator()(const T& p, A x) const;

};

mem\_fun\_ref\_t通过一个无参数成员函数生成一个函数对象（引用版）

mem\_fun1\_ref\_t 通过一个带一个参数的成员函数生成一个函数对象（引用版）

const\_mem\_fun\_ref\_t通过一个 const 限定的无参数成员函数生成一个函数对象（引用版）

const\_mem\_fun1\_ref\_t通过一个 const 限定的带一个参数的成员函数生成一个函数对象（引用版）

#### 3.6.9.mem\_fun\_ref

template< class Res, class T > mem\_fun\_ref\_t<Res,T> mem\_fun\_ref( Res (T::\*f)() );

template< class Res, class T > const\_mem\_fun\_ref\_t<Res,T> mem\_fun\_ref( Res (T::\*f)() );

template< class Res, class T, class Arg > mem\_fun1\_ref\_t<Res,T,Arg> mem\_fun\_ref( Res (T::\*f)(Arg) );

template< class Res, class T, class Arg > const\_mem\_fun1\_ref\_t<Res,T,Arg> mem\_fun\_ref( S (T::\*f)(Arg) );

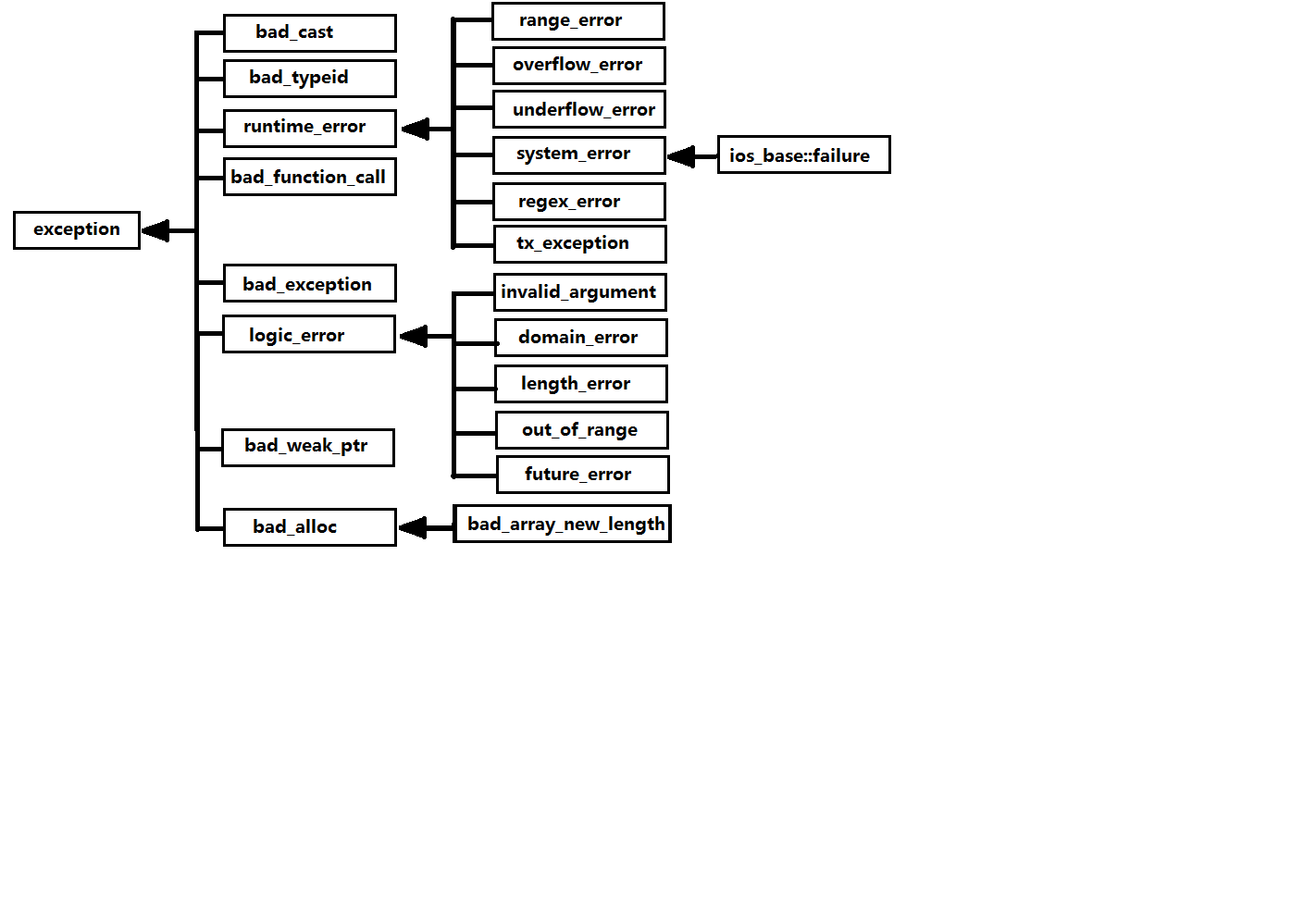
将一个不带参数或者带一个参数的成员函数转化成函数对象（引用版），只用于容器元素是对象时才能用。

# C++11 诊断库

## 1 异常处理

### 1.1 exception类

C++异常类是以exception为基类，派生出了许多类，主要分为逻辑异常类和运行异常类。主要的派生关系如图：



构造函数

exception();(1)

exception( const exception& other );（2）

成员函数

exception& operator=( const exception& other );//不可继承

virtual const char\* what() const; //获取描述信息

### 1.2 捕获和存储异常对象

#### 1.2.1 exception\_ptr类

用于操作异常对象的共享指针类型。

std::exception\_ptr是一个可为空的类似于指针的类型，用于管理已抛出并使用std :: current\_exception捕获的异常对象。std::exception\_ptr的对象可以传递给另一个函数或者线程，这样异常可以被再次抛出或者捕获。

默认构造的std :: exception\_ptr是一个空指针;它不指向异常对象。

std :: exception\_ptr的两个对象只有当它们都为null或两者指向同一个异常对象时才会相等。

std :: exception\_ptr不能隐式转换为任何算术，枚举或指针类型。可转换为bool，如果为null，则为false，否则为true。

只要至少有一个std :: exception\_ptr引用它，std :: exception\_ptr引用的异常对象仍然有效：std :: exception\_ptr是一个共享所有权智能指针（注意：除了通常的异常对象寿命规则）std :: exception\_ptr满足NullablePointer的要求。

#### 1.2.2 nested\_exception类

std :: nested\_exception是一个多态的mixin类，可以捕获和存储当前异常，可以嵌套任意类型的异常。

构造函数

nested\_exception() noexcept;(1) (since C++11) // 调用std::current\_exception()

nested\_exception( const nested\_exception& other ) noexcept = default;

成员函数

nested\_exception& operator=( const nested\_exception& other ) noexcept = default;

将存储的异常替换为其他异常。

void rethrow\_nested() const;

重新抛出存储的异常。如果没有存储的异常（即nested\_ptr（）返回空指针），则调用std :: terminate。

std::exception\_ptr nested\_ptr() const noexcept;

返回指向存储的异常的指针（如果有）。

非成员函数

template< class T >

void throw\_with\_nested( T&& t );

嵌套当前异常及参数中保存的异常且抛出

如果std :: decay <T> :: type是一个no-final n-uninon类型，既不是std :: nested\_exception也不是从std :: nested\_exception派生，就抛出一个从std:: nested\_exception和std :: decay <T> :: type公开派生的未指定类型的异常，并从std :: forward <T>（t）构造。 nested\_exception基类的默认构造函数调用std :: current\_exception，捕获std :: exception\_ptr中当前处理的异常对象（如果有的话）。否则，抛出std :: forward <T>（t）。需要std :: decay <T> :: type是可拷贝构造的。

template< class E >

void rethrow\_if\_nested( const E& e );

如果异常嵌套，则再次抛出异常。

如果E不是多态类类型，或者如果std :: nested\_exception是E的不可访问或不明确的基类，则没有任何效果。否则，执行

if（auto p = dynamic\_cast <const std :: nested\_exception \*>（std :: addressof（e）））

P-> rethrow\_nested（）;

#### 1.2.3 uncaught\_exception

bool uncaught\_exception() noexcept;

检测异常处理是否正在进行。

（1）检测当前线程是否具有实时异常对象，即异常已抛出或重新抛出，并且尚未输入匹配的catch子句，std :: terminate或std :: unexpected。换句话说，std :: uncaught\_exception检测堆栈展开是否正在进行中。

（2）检测当前线程中有多少异常被抛出或重新抛出，且尚未输入匹配的catch子句。

有时，即使std :: uncaught\_exception（）== true，也可以抛出异常。例如，如果堆栈展开导致堆栈分配的对象被破坏，则该对象的析构函数可以运行引发异常的代码，只要异常在转义析构函数之前被某些catch块捕获。

#### 1.2.4 make\_exception\_ptr

template< class E >

std::exception\_ptr make\_exception\_ptr ( E e ) noexcept;

创建一个指向异常对象e的拷贝的 exception\_ptr 对象

#### 1.2.5 current\_exception

std::exception\_ptr current\_exception() noexcept;

捕获当前异常。

如果在异常处理期间（通常在catch子句中）调用，则捕获当前异常对象并创建一个std :: exception\_ptr，该对象持有该异常对象的副本或引用（取决于实现）。引用对象保持有效，只要有一个引用它的exception\_ptr对象。

如果此函数的需要调用new并且调用失败，则返回的指针将持有对std :: bad\_alloc实例的引用。

该函数的执行需要复制捕获的异常对象，但其复制构造函数引发了异常，那么返回的指针将会保存对引发的异常的引用。如果抛出的异常对象的复制构造函数也抛出异常，返回的指针可能会持有对std :: bad\_exception的实例的引用，以破坏无限循环。

如果在没有异常要处理时调用该函数，则返回一个空的std :: exception\_ptr。

#### 1.2.6 rethrow\_exception

void rethrow\_exception( std::exception\_ptr p )

再次抛出异常。

抛出先前捕获的由异常指针p指向的异常对象。

#### 1.2.7 throw\_with\_nested

template< class T >

void throw\_with\_nested( T&& t );

嵌套当前异常及参数中保存的异常且抛出

如果std :: decay <T> :: type是一个no-final n-uninon类型，既不是std :: nested\_exception也不是从std :: nested\_exception派生，就抛出一个从std:: nested\_exception和std :: decay <T> :: type公开派生的未指定类型的异常，并从std :: forward <T>（t）构造。 nested\_exception基类的默认构造函数调用std :: current\_exception，捕获std :: exception\_ptr中当前处理的异常对象（如果有的话）。否则，抛出std :: forward <T>（t）。需要std :: decay <T> :: type是可拷贝构造的。

#### 1.2.8 rethrow\_if\_nested

template< class E >

void rethrow\_if\_nested( const E& e );

如果异常嵌套，则再次抛出异常。

如果E不是多态类类型，或者如果std :: nested\_exception是E的不可访问或不明确的基类，则没有任何效果。否则，执行

if（auto p = dynamic\_cast <const std :: nested\_exception \*>（std :: addressof（e）））

P-> rethrow\_nested（）;

### 1.3 异常处理失败的处理

#### 1.3.1 terminate

void terminate();

void terminate() noexcept;

当以下任何原因导致异常处理失败时，C++运行时调用std :: terminate（）：

（1）抛出异常并且没有被捕获（在这种情况下是否有堆栈展开与编译器实现有关）。

（2）在异常处理抛出异常（例如来自某个本地对象的析构函数或者在异常处理期间必须调用的函数）

（3）静态的构造函数或析构函数或线程本地对象抛出异常。

（4）使用std :: atexit或std :: at\_quick\_exit注册的函数抛出异常。

（5）违反noexception异常规范抛出异常。（在这种情况下是否有堆栈展开与编译器实现有关）（c++11）

（6）违反动态异常规范throw()，执行std :: unexpected的默认处理程序时。（c++98）

（7）std :: unexpected的非默认处理程序抛出违反先前的动态异常规范的异常，且异常规范不包括std :: bad\_exception。

（8）对于不持有捕获异常的对象，调用std :: nested\_exception :: rethrow\_nested。

（9）从std :: thread的初始函数抛出异常。

（10）一个等待回收的std::thread被销毁或分配。

（11）一个并行算法调用的函数退出通过未捕获的异常退出或者执行策略规定终止。

std::terminate()也可以直接在程序中调用。

在任何情况下，std :: terminate调用当前注册的std :: terminate\_handler。默认std :: terminate\_handler调用std :: abort。

如果析构函数在堆栈展开期间重置terminate\_handler，并且展开后来导致terminate被调用，则在throw表达式末尾注册的处理程序将被调用。 （注意：是否重新抛出应用的新的处理程序是模糊的）（until C++ 11）

如果一个析构函数在堆栈展开过程中重置了这个终止处理程序，那么如果展开后来导致终止被调用，那么哪一个处理程序将被调用是不明确的。

#### 1.3.2 terminate\_handler

typedef void (\*terminate\_handler)();

std :: terminate\_handler是函数指针类型（指向函数的指针，不带参数，返回void），它由函数std :: set\_terminate和std :: get\_terminate设置和查询，并由std :: terminate调用。

C++编译器默认提供了一个默认的std::terminate\_handler 函数，它调用std::abort（），如果terminate\_handler通过std::set\_terminate()设置为null，则c++编译器可以恢复默认值。

#### 1.3.3 get\_terminate

std::terminate\_handler get\_terminate() noexcept;

返回当前注册的std :: terminate\_handler，它可能是空指针。

#### 1.3.4 set\_terminate

std::terminate\_handler set\_terminate( std::terminate\_handler f );

将std :: terminate\_handler设置为f，返回之前注册的std :: terminate\_handler。

此功能是线程安全的。每次调用std :: set\_terminate同步（参见std :: memory\_order）后续的std :: set\_terminate和std :: get\_terminate。

#### 1.3.5 bad\_exception类

bad\_exception

（1）如果std :: exception\_ptr存储了捕获的异常的副本，并且被 [std::current\_exception](http://en.cppreference.com/w/cpp/error/current_exception)捕获的异常对象的拷贝构造函数抛出了异常。这时捕获的异常就是std::bad\_exception的实例。

（2）如果违反了动态异常规范，并且std :: unexpected抛出或重新抛出仍然违反异常规范的异常，但异常规范允许std :: bad\_exception，则抛出std :: bad\_exception。

### 1.4 违反异常规范时的处理

#### 1.4.1 unexpected

void unexpected();

当违反动态异常规范时，抛出了不在异常规范列表中的异常时，C++调用std :: unexpected（）。

std :: unexpected（）也可以直接从程序调用。

std :: unexpected调用当前注册的std :: unexpected\_handler。默认的std :: unexpected\_handler调用std :: terminate。

如果析构函数在堆栈展开期间重置terminate\_handler，并且展开后来导致terminate被调用，则在throw表达式末尾注册的处理程序将被调用。 （注意：是否重新抛出应用的新的处理程序是模糊的）（until C++ 11）

如果一个析构函数在堆栈展开过程中重置了这个终止处理程序，那么如果展开后来导致终止被调用，那么哪一个处理程序将被调用是不明确的。

#### 1.4.2 unexpected\_handler

typedef void (\*unexpected\_handler)();

std :: unexpected\_handler是函数指针类型（指向函数的指针，不带参数并返回void），它由函数std :: set\_unexpected和std :: get\_unexpected安装和查询，并由std :: unexpected调用。

C++编译器实现提供了一个默认的std :: unexpected\_handler函数，它调用std :: terminate（）。如果通过std :: set\_unexpected设置为空指针，则编译器可以恢复默认处理程序。

用户定义的std :: unexpected\_handler需要终止程序或抛出异常。如果抛出异常，可能会遇到以下三种情况之一：

（1）std :: unexpected\_handler抛出的异常满足先前违反的动态异常规范。新的异常允许从该函数退出函数和从堆栈展开处继续。

（2）std :: unexpected\_handler抛出的异常仍然违反了异常规范：

（2a）如果异常规范允许std :: bad\_exception，抛出的异常对象被销毁，然后抛出std：：bad\_exception。

（2b）异常规范不允许std :: bad\_exception，std :: terminate（）被调用。

#### 1.4.3 get\_unexpected

std::unexpected\_handler get\_unexpected() noexcept;

返回当前注册的std :: unexpected\_handler，它可能是空指针。此功能是线程安全的。

#### 1.4.4 set\_unexpected

std::unexpected\_handler set\_unexpected( std::unexpected\_handler f );

将unexpected\_handler设置为f，并返回之前注册的unexpected\_handler。此功能是线程安全的。每次调用std :: set\_unexpected都会同步（参见std :: memory\_order）对std :: set\_unexpected和std :: get\_unexpected的后续调用。

### 1.5异常种类

**所有的异常类都继承exception类。因此所有派生类都具有what（）方法**。

|  |  |
| --- | --- |
| logic\_error | 逻辑错误，是可以预防的，标准库不会直接抛出此异常。 |
| invalid\_argument | 无效参数，由std :: bitset :: bitset和std :: stoi和std :: stof系列函数抛出 |
| domain\_error | 域错误，输入数据在操作的定义域之外。标准库组件不会抛出此异常， |
| length\_error | 长度错误，试图生成一个超出该类型最大长度的对象时，例如vector的resize操作。 |
| out\_of\_range | 超出范围，由std :: bitset和std :: basic\_string的成员函数， std :: stoi和std :: stod函数系列以及受限制成员访问函数（std :: vector :: at和std :: map :: at）抛出 |
| future\_error | 由处理异步执行和共享状态（std :: future，std :: promise等）的线程库中的函数失败引发的异常对象。与system\_error类似，此异常携带与std :: error\_code兼容的错误代码。 |
| runtime\_error | 运行时异常，由std ::locale::locale 和std::locale::combine.抛出 |
| range\_error | 范围错误，计算结果不能由目标类型表示，抛出此异常的唯一标准库组件是std :: wstring\_convert :: from\_bytes和std :: wstring\_convert :: to\_bytes。 |
| overflow\_error | 算术上溢出，计算结果过大，由std :: bitset :: to\_ulong和std :: bitset :: to\_ullong抛出 |
| underflow\_error | 算术下溢出，计算结果是低于正常的浮点值，标准库组件不会抛出此异常 |
| regex\_error | 正则表达式库出现的错误。 |
| tx\_exception | 可用于取消和回滚由关键字atomic\_cancel启动的原子事务。 |
| system\_error | 各种库函数抛出的异常类型，通常是与OS设备接口的函数，例如std :: thread的构造函数，当异常有一个关联的std :: error\_code时，可能会被报告。 |
| ios\_base::failure | 输入输出库中函数平抛出的异常 |
| bad\_typeid | 使用typeid操作符，操作一个NULL指针，而该指针是带有虚函数的类，这时抛出bad\_typeid异常 |
| bad\_cast | 使用dynamic\_cast转换引用失败的时候 |
| bad\_weak\_ptr | 当std::shared\_ptr使用托管对象已经销毁的std::weak\_ptr作为构造参数时抛出。 |
| bad\_function\_call | 当调用没有包装任何内容的std::function时。 |
| bad\_alloc | new或new[]操作失败时抛出。 |
| bad\_array\_new\_length | 定义数组时如果长度为负数，或长队超过最大限制，或初值数量超过数组长度时抛出 |
| bad\_exception | 非预期的异常，这是个特殊的异常，如果函数的异常抛出列表里声明了bad\_exception异常，当函数内部抛出了异常抛出列表中没有的异常，这是调用的unexpected函数中若抛出异常，不论什么类型，都会被替换为bad\_exception类型。 |

#### 1.2.1 逻辑错误和运行时错误

logic\_error、invalid\_argument、domain\_error、length\_error、out\_of\_range与runtime\_error、range\_error、overflow\_error、underflow\_error。都具有类似的构造函数和成员函数。

构造函数

explicit 类名( const std::string& what\_arg );

explicit 类名( const char\* what\_arg );

使用what\_arg构造异常对象作为可以通过what（）访问的说明字符串。因为复制std :: exception时不允许抛出异常，所以此消息通常在内部存储为单独分配的引用计数字符串。这也是为什么没有构造函数使用std::string&&的原因。

成员函数

virtual const char\* what() const;

返回说明字符。

#### 1.2.3 future\_error

构造函数

future\_error( std::error\_code ec );

成员函数

const std::error\_code& code() const noexcept;

virtual const char\* what() const noexcept;

#### 1.2.4 regex\_error

构造函数

regex\_error(std::regex\_constants::error\_type ecode);

成员函数

std::regex\_constants::error\_type code() const;

virtual const char\* what() const noexcept;

#### 1.2.5 tx\_exception

构造函数

explicit tx\_exception( T value ) transaction\_safe;(1)

tx\_exception( T value, const std::string& what\_arg ) transaction\_safe;(2)

tx\_exception( T value, const char\* what\_arg ) transaction\_safe;

成员函数

T get() const transaction\_safe;

virtual const char\* what() const;

#### 1.2.6 system\_error

构造函数

system\_error( std::error\_code ec );(1) (since C++11)

system\_error( std::error\_code ec, const std::string& what\_arg );(2) (since C++11)

system\_error( std::error\_code ec, const char\* what\_arg );(2) (since C++11)

system\_error( int ev, const std::error\_category& ecat );(3) (since C++11)

system\_error( int ev, const std::error\_category& ecat,const std::string& what\_arg);(4) (since C++11)

system\_error( int ev, const std::error\_category& ecat,const char\* what\_arg);

成员函数

const std::error\_code& code() const noexcept;

virtual const char\* what() const noexcept;

virtual const char\* what() const;

#### 1.2.7 ios\_base::failure

构造函数

explicit failure( const std::string& message );(until C++11)

explicit failure(const std::string& message, const std::error\_code& ec = std::io\_errc::stream );(since C++11)

explicit failure( const char\* message,const std::error\_code& ec = std::io\_errc::stream );

成员函数

const std::error\_code& code() const noexcept;

virtual const char\* what() const noexcept;

virtual const char\* what() const;

#### 1.2.7 bad\_\*派生类

bad\_typeid,bad\_cast,bad\_wear\_ptr,bad\_function\_call,bad\_alloc,bad\_array\_new\_length,bad\_exception。都具有空的构造函数，和what成员函数。

类名（）；

virtual const char\* what() const;

此外。bad\_alloc ，bad\_exception，bad\_array\_new\_length 具有赋值运算符函数。

类名& operator = (const 类名& othre)

bad\_exception具有拷贝构造函数。

bad\_exception ( const bad\_exception& other)

## 2 错误码errno

errno是用于错误指示的预处理器宏。它扩展为类型为static（until C++ 11）thread-local（since C++ 11）可修改的左值int。几个标准库函数通过将正整数写入errno来指示错误。

通常，errno的值设置为<cerrno>列出的错误代码之一，以字母E开头，后跟大写字母或数字的宏常量。

在程序启动时，errno的值为0，无论是否发生错误，库函数都被允许将正整数写入errno，但库函数不会将0存储在errno中。

|  |  |
| --- | --- |
| E2BIG **C++11** | 参数列表太长（Argument list too long） |
| EACCES **C++11** | 拒绝访问（Permission denied） |
| EADDRINUSE **C++11** | 指定地址正在被使用（Address in use） |
| EADDRNOTAVAIL **C++11** | 指定地址不可用（Address not available） |
| EAFNOSUPPORT **C++11** | 地址族（或协议族）不支持（Address family not supported） |
| EAGAIN **C++11** | 资源获取失败，再次尝试（Resource unavailable try again） |
| EALREADY **C++11** | 联接已经在进行（Connection already in progress） |
| EBADF **C++11** | 无效的文件描述符（Bad file descriptor） |
| EBADMSG **C++11** | 无效的消息（Bad message） |
| EBUSY **C++11** | 设备或资源忙碌（Device or resource busy） |
| ECANCELED **C++11** | 操作被取消（Operation canceled） |
| ECHILD **C++11** | 不存在指定的子进程（No child process） |
| ECONNABORTED **C++11** | 连接停止（Connection aborted） |
| ECONNREFUSED **C++11** | 连接被拒绝（Connection refused） |
| ECONNRESET **C++11** | 连接重设（Connection reset） |
| EDEADLK **C++11** | 可能发生资源死锁（Resource deadlock would occur） |
| EDESTADDRREQ **C++11** | 不处于连接模式，没有指定对端地址（Destination address required） |
| EDOM | 参数超出定义域（Argument out of domain） |
| EEXIST **C++11** | 文件已经存在（File exists） |
| EFAULT **C++11** | 内存空间访问出错（Bad address） |
| EFBIG **C++11** | 文件太大（File too large） |
| EHOSTUNREACH **C++11** | 请求网络连接的远程主机不可到达（Host unreachable） |
| EIDRM **C++11** | 标识符已删除（Identifier removed） |
| EILSEQ **C++11** | 非法的字符序列（Illegal byte sequence） |
| EINPROGRESS **C++11** | 操作正在进行（Operation in progress） |
| EINTR **C++11** | 操作被信号中断（Interrupted） |
| EINVAL **C++11** | 参数无效（Invalid argument） |
| EIO **C++11** | IO 错误（Io error） |
| EISCONN **C++11** | 基于连接的套接字已被连接上（Already connected） |
| EISDIR **C++11** | 是一个目录（Is a directory） |
| ELOOP **C++11** | 太多层符号链接（Too many synbolic link levels） |
| EMFILE **C++11** | 当前进程打开太多文件（Too many files open） |
| EMLINK **C++11** | 太多链接（Too many links） |
| EMSGSIZE **C++11** | 套接字发送的消息大小超过了所支持的最大值（Message size） |
| ENAMETOOLONG **C++11** | 文件名太长（Filename too long） |
| ENETDOWN **C++11** | 网络宕机（Network down） |
| ENETRESET **C++11** | 网络重置（Network reset） |
| ENETUNREACH **C++11** | 网络不可到达（Network unreachable） |
| ENFILE **C++11** | 整个系统中打开了太多不同文件（Too many files open in system） |
| ENOBUFS **C++11** | 用于 IO 操作的所有内核缓存都在使用（No buffer space） |
| ENODATA **C++11** | 无数据（No message available） |
| ENODEV **C++11** | 无该设备（No such device） |
| ENOENT **C++11** | 不存在该文件或目录（No such file or directory） |
| ENOEXEC **C++11** | 执行格式错误（Executable format error） |
| ENOLCK **C++11** | 没有可用的记录锁（No lock available） |
| ENOLINK **C++11** | 链接正在服务中（Link has been severed） |
| ENOMEM **C++11** | 内存不足（Not enough memory） |
| ENOMSG **C++11** | 没有符合所需类型的消息（No message of the desired type） |
| ENOPROTOOPT **C++11** | 协议不可用（Protocol not available） |
| ENOSPC **C++11** | 设备空间不足（No space on device） |
| ENOSR **C++11** | 流资源不足（No stream resources） |
| ENOSTR **C++11** | 设备不是流（Not a stream） |
| ENOSYS **C++11** | 函数不支持（Function not supported） |
| ENOTCONN **C++11** | 套接字尚未连接，目标没有给出（Not connected） |
| ENOTDIR **C++11** | 目录（路径）不存在（Not a directory） |
| ENOTEMPTY **C++11** | 目录非空（Directory not empty） |
| ENOTRECOVERABLE **C++11** | 状态不可恢复（State not recoverable） |
| ENOTSOCK **C++11** | 在非套接字上进行套接字操作（Not a socket） |
| ENOTSUP **C++11** | 不支持（Not supported） |
| ENOTTY **C++11** | 不合适的 IO 控制操作（Inappropriate io control operation） |
| ENXIO **C++11** | 设备或地址不存在（No such device or address） |
| EOPNOTSUPP **C++11** | 操作不支持（Operation not supported） |
| EOVERFLOW **C++11** | 值太大（Value too large） |
| EOWNERDEAD **C++11** | 之前的所有者已经断开（Owner dead） |
| EPERM **C++11** | 操作不允许（Operation not permitted） |
| EPIPE **C++11** | 管道中断（Broken pipe） |
| EPROTO **C++11** | 协议错误（Protocol error） |
| EPROTONOSUPPORT **C++11** | 协议不支持（Protocol not supported） |
| EPROTOTYPE **C++11** | 错误的协议类型（Wrong protocol type） |
| ERANGE | 结果超出范围（Result out of range） |
| EROFS **C++11** | 只读文件系统（Read only file system） |
| ESPIPE **C++11** | 无效查找（Invalid seek） |
| ESRCH **C++11** | 进程不存在（No such process） |
| ETIME **C++11** | 流超时（Stream timeout） |
| ETIMEDOUT **C++11** | 连接超时（Timed out） |
| ETXTBSY**C++11** | 文本文件忙（Text file busy） |
| EWOULDBLOCK **C++11** | 操作会阻塞（Operation would block） |
| EXDEV **C++11** | 跨设备链路（Cross device link） |

## 3 断言

### 3.1 assert

#ifdef NDEBUG

#define assert(condition) ((void)0)

#else

#define assert(condition) /\*implementation defined\*/

#endif

宏定义assert的定义取决于另一个未由标准库定义的宏NDEBUG。

如果在源代码中NDEBUG被定义且包含头文件<cassert>，则assert不执行任何操作。

如果未定义NDEBUG，则assert将检查条件表达式是否为零。如果为0，断言输出标准错诊断信息，这与编译器有关，并调用std :: abort。诊断信息需要包括条件表达式以及标准宏\_\_FILE\_\_，\_\_LINE\_\_和标准变量\_\_func\_\_的值。

### 3.2 static\_assert

static\_assert ( bool\_constexpr , message ) (since C++11)

静态断言可能出现在命名空间和块范围（作为块声明）和类主体内（作为成员声明）。

如果bool\_constexpr返回true，则此声明不起作用。否则将发出编译时错误，并且message的文本（如果有）包含在诊断消息中。

由于message必须是字符串字面值，它不能包含动态信息，甚至是非字符串文字本身的常量表达式。特别是它不能包含模板类型参数的名称。

## 4 系统错误

### 4.1 error\_category类

std :: error\_category作为特定错误类型的基类，例如std :: system\_category，std :: iostream\_category等。每个特定error\_category类定义了error\_code - error\_condition映射，并保存了所有error\_conditions的说明字符串。error\_category的对象被视为单例对象，通过引用传递。

构造函数

error\_category( const error\_category& other ) = delete;(1) (since C++11)

constexpr error\_category() noexcept;(2) (since C++14)

成员函数

error\_category& operator = ( const error\_category&) =delete

不可进行赋值运算

virtual const char\* name() const noexcept = 0;

返回指向错误类别名称字符串的终止字符的指针。

virtual std::error\_condition default\_error\_condition( int code ) const noexcept;

返回给定错误代码的错误条件。等效于std::error\_condition(code, \*this).

virtual bool equivalent( int code, const std::error\_condition& condition ) const noexcept;（1）

virtual bool equivalent( const std::error\_code& code,int condition ) const noexcept;（2）

检查错误代码是否等同于由\* this表示的错误类别的错误条件。

（1）等效于 default\_error\_condition(code) == condition.

（2）等效于 \*this == code.category() && code.value() == condition.

virtual std::string message( int condition ) const = 0;

返回描述 \* this表示的错误类别的给定错误条件的字符串。

bool operator==( const error\_category& rhs ) const noexcept;（1）

bool operator!=( const error\_category& rhs ) const noexcept;（2）

bool operator<( const error\_category& rhs ) const noexcept;（3）

（1）检查\* this和rhs是否引用相同的对象。

（2）检查\* this和rhs是否不指向同一个对象。

（3）根据&rhs和this的顺序对\*this和rhs排序，等效于std::less<const error\_category\*>()(this, &rhs)

### 4.2 generic\_category

const std::error\_category& generic\_category() noexcept;

获取一般错误的error category对象的引用。该对象需要重写虚函数error\_category :: name（）才能返回一个指向字符串“generic”的指针。它用于识别与POSIX错误代码相对应的错误条件。

const std::error\_category& system\_category() noexcept;

获取操作系统报告的error\_category对象的引用。该对象需要覆盖虚拟函数std :: error\_category :: name（）以返回指向字符串“system”的指针。还需要覆盖虚拟函数std :: error\_category :: default\_error\_condition（）以将匹配POSIX errno值的错误代码映射到std :: generic\_category。

### 4.3 error\_condition类

std :: error\_condition是一个与平台无关的错误代码。像std :: error\_code一样，它由一个整数值和一个std :: error\_category唯一标识，但与std :: error\_code不同，该值与平台无关。典型的实现包含一个整数数据成员（值）和一个指向std :: error\_category的指针。

构造函数

error\_condition() noexcept;(1)

error\_condition( const error\_condition& other ) noexcept;(2)

error\_condition( int val, const error\_category& cat ) noexcept;(3)

template< class ErrorConditionEnum > error\_condition( ErrorConditionEnum e ) noexcept;(4)

1）默认构造函数。使用generic category和 error code 0初始化 error\_condition。

2）拷贝构造函数。用其他error\_condition构造error\_condition。

3）使用错误代码val和error\_category cat初始化错误条件。

4）使用枚举e初始化error\_condition。有高效地调用make\_error\_condition（）。除非is\_error\_condition\_enum <ErrorConditionEnum> :: value为true，否则不参与重载。

成员函数

error\_condition& operator=( const error\_condition& other ) noexcept;(1)

template< class ErrorConditionEnum > error\_condition& operator=( ErrorConditionEnum e ) noexcept;

（1）赋值运算符。用other赋值。

（2）用枚举e 像error\_condition赋值。有效地调用make\_error\_condition（）。除非is\_error\_condition\_enum <ErrorConditionEnum> :: value为true，否则不参与重载。

void assign( int val, const error\_category& cat ) noexcept;

将内容赋值到error\_condition。将错误代码设置为val，将错误类别设置为cat。

void clear() noexcept;

清除错误状态。将错误代码设置为0，将错误类别设置为std :: generic\_category。

int value() const noexcept;

返回存储的错误代码。

const error\_category& category() const noexcept;

返回存储的错误类别。

std::string message() const;

返回存储的错误代码和错误类别的解释性消息。有效地调用category（）.message（value（））。

explicit operator bool() const noexcept;

检查存储的错误代码是否不为零。

非成员函数

operator==、operator!=、operator<

比较error\_conditions和error\_codes。

#### 4.3.1 辅助类

template< class T >

struct is\_error\_condition\_enum;

如果T是错误条件枚举，则此模板将成员常量value设置为true。对于任何其他类型，value为false。此模板专用于用户定义的类型，以指示该类型符合std :: error\_condition自动转换的资格。标准库的类是错误条件枚举：std :: errc。

### 4.4 std::errc枚举

范围枚举std :: errc定义了与POSIX错误代码相对应的便携式错误条件的值。

|  |  |
| --- | --- |
| **Constant** | **Explanation（**POSIX code**）** |
| address\_family\_not\_supported | EAFNOSUPPORT |
| address\_in\_use | EADDRINUSE |
| address\_not\_available | EADDRNOTAVAIL |
| already\_connected | EISCONN |
| argument\_list\_too\_long | E2BIG |
| argument\_out\_of\_domain | EDOM |
| bad\_address | EFAULT |
| bad\_file\_descriptor | EBADF |
| bad\_message | EBADMSG |
| broken\_pipe | EPIPE |
| connection\_aborted | ECONNABORTED |
| connection\_already\_in\_progress | EALREADY |
| connection\_refused | ECONNREFUSED |
| connection\_reset | ECONNRESET |
| cross\_device\_link | EXDEV |
| destination\_address\_required | EDESTADDRREQ |
| device\_or\_resource\_busy | EBUSY |
| directory\_not\_empty | ENOTEMPTY |
| executable\_format\_error | ENOEXEC |
| file\_exists | EEXIST |
| file\_too\_large | EFBIG |
| filename\_too\_long | ENAMETOOLONG |
| function\_not\_supported | ENOSYS |
| host\_unreachable | EHOSTUNREACH |
| identifier\_removed | EIDRM |
| illegal\_byte\_sequence | EILSEQ |
| inappropriate\_io\_control\_operation | ENOTTY |
| interrupted | EINTR |
| invalid\_argument | EINVAL |
| invalid\_seek | ESPIPE |
| io\_error | EIO |
| is\_a\_directory | EISDIR |
| message\_size | EMSGSIZE |
| network\_down | ENETDOWN |
| network\_reset | ENETRESET |
| network\_unreachable | ENETUNREACH |
| no\_buffer\_space | ENOBUFS |
| no\_child\_process | ECHILD |
| no\_link | ENOLINK |
| no\_lock\_available | ENOLCK |
| no\_message\_available | ENODATA |
| no\_message | ENOMSG |
| no\_protocol\_option | ENOPROTOOPT |
| no\_space\_on\_device | ENOSPC |
| no\_stream\_resources | ENOSR |
| no\_such\_device\_or\_address | ENXIO |
| no\_such\_device | ENODEV |
| no\_such\_file\_or\_directory | ENOENT |
| no\_such\_process | ESRCH |
| not\_a\_directory | ENOTDIR |
| not\_a\_socket | ENOTSOCK |
| not\_a\_stream | ENOSTR |
| not\_connected | ENOTCONN |
| not\_enough\_memory | ENOMEM |
| not\_supported | ENOTSUP |
| operation\_canceled | ECANCELED |
| operation\_in\_progress | EINPROGRESS |
| operation\_not\_permitted | EPERM |
| operation\_not\_supported | EOPNOTSUPP |
| operation\_would\_block | EWOULDBLOCK |
| owner\_dead | EOWNERDEAD |
| permission\_denied | EACCES |
| protocol\_error | EPROTO |
| protocol\_not\_supported | EPROTONOSUPPORT |
| read\_only\_file\_system | EROFS |
| resource\_deadlock\_would\_occur | EDEADLK |
| resource\_unavailable\_try\_again | EAGAIN |
| result\_out\_of\_range | ERANGE |
| state\_not\_recoverable | ENOTRECOVERABLE |
| stream\_timeout | ETIME |
| text\_file\_busy | ETXTBSY |
| timed\_out | ETIMEDOUT |
| too\_many\_files\_open\_in\_system | ENFILE |
| too\_many\_files\_open | EMFILE |
| too\_many\_links | EMLINK |
| too\_many\_symbolic\_link\_levels | ELOOP |
| value\_too\_large | EOVERFLOW |
| wrong\_protocol\_type | EPROTOTYPE |

非成员函数

std::error\_code make\_error\_code( std::errc e ) noexcept;

为枚举errc e创建错误代码值。相当于std :: error\_code（static\_cast <int>（e），std :: generic\_category（））

std::error\_condition make\_error\_condition( std::errc e ) noexcept;

为枚举errc e创建错误条件。将错误代码设置为int（e），将错误类别设置为std :: generic\_category。

#### 4.4.1 辅助类

template< class T >

struct is\_error\_condition\_enum;

如果T是错误条件枚举，则此模板将成员常量value设置为true。对于任何其他类型，value为false。此模板专用于用户定义的类型，以指示该类型符合std :: error\_condition自动转换的资格。标准库的类是错误条件枚举：std :: errc。

### 4.5 error\_code类

std :: error\_code是与平台相关的错误代码。每个std :: error\_code对象包含源自操作系统或某些低级接口的错误代码，以及指向类型为std :: error\_category的对象的指针，该对象对应于所述界面。错误代码值根据不同的错误类别可能并非唯一。

构造函数

error\_code() noexcept;(1)

error\_code( int ec, const error\_category& ecat ) noexcept;(2)

template< class ErrorCodeEnum > error\_code( ErrorCodeEnum e ) noexcept;（3）

（1）构造具有默认值的错误代码。相当于error\_code（0，std :: system\_category（））。

（2）构造用ec作为平台相关的错误代码，用ecat作为相应的错误类别的错误代码。

（3）从错误代码枚举e中构造错误代码。相当于\* this = make\_error\_code（e）。只有在std :: is\_error\_code\_enum <ErrorCodeEnum> :: value == true的情况下，才会参重载。

成员函数

template< class ErrorCodeEnum >

error\_code& operator=( ErrorCodeEnum e ) noexcept;

将枚举e代表的代码错误代码和相应的类别赋值给error\_code。相当于\* this = std :: make\_error\_code（e）。只有在std :: is\_error\_code\_enum <ErrorCodeEnum> :: value == true的情况下，才会参与重载。

void assign( int ec, const error\_category& ecat ) noexcept;

用错误代码ec和相应的类别ecat向error\_code赋值。

void clear() noexcept;

用默认值替换错误代码和错误类别。相当于\* this = error\_code（0，std :: system\_category（））。

int value() const noexcept;

返回与平台相关的错误值。

const std::error\_category& category() const noexcept;

返回错误值的错误类别。

std::error\_condition default\_error\_condition() const noexcept;

返回当前错误值的默认错误条件。相当于category（）.default\_error\_condition（value（））

std::string message() const;

返回与当前错误值和类别相对应的消息。相当于category（）.message（value（））。

explicit operator bool() const noexcept;

检查错误值是否有效，即非零。

非成员函数

std::basic\_ostream<CharT,Traits>&

operator<<( basic\_ostream<CharT,Traits>& os, const error\_code& ec );

对错误代码ec执行流输出操作。相当于os << ec.category（）.name（）<<'：'<< ec.value。

bool operator==( const error\_code& lhs, const error\_code& rhs ) noexcept;(1)

bool operator!=( const error\_code& lhs, const error\_code& rhs ) noexcept;(2)

bool operator<( const error\_code& lhs, const error\_code& rhs ) noexcept;（3）

比较两对象

1）比较lhs和rhs的是否相等。

2）比较lhs和rhs的是否不等。

3）检查lhs是否小于rhs。

#### 4.5.1 辅助类

template< class T >

struct is\_error\_code\_enum;

如果T是错误代码枚举，则该模板提供成员常数value等于true。对于任何其他类型，value为false。此模板可能专用于用户定义的类型，以指示该类型符合std :: error\_code和std :: error\_condition自动转换的资格。以下类的标准库是一个错误代码枚举：

std::io\_errc

std::future\_errc.：

template<>

struct hash<error\_code>;

std :: error\_code的专业化std :: hash模板使得用户可以获得类型为std :: error\_code的对象的散列。

### 4.6 system\_error

各种库函数抛出的异常类型，通常是与OS设备接口的函数，例如std :: thread的构造函数，当异常有一个关联的std :: error\_code时，可能会被报告。

构造函数

system\_error( std::error\_code ec );(1) (since C++11)

system\_error( std::error\_code ec, const std::string& what\_arg );(2) (since C++11)

system\_error( std::error\_code ec, const char\* what\_arg );(2) (since C++11)

system\_error( int ev, const std::error\_category& ecat );(3) (since C++11)

system\_error( int ev, const std::error\_category& ecat,const std::string& what\_arg);(4) (since C++11)

system\_error( int ev, const std::error\_category& ecat,const char\* what\_arg);

成员函数

const std::error\_code& code() const noexcept;

virtual const char\* what() const noexcept;

virtual const char\* what() const;