# 写在前面：

本科期间我曾接触过C++语言，但从未系统性地、深入地学习和使用过。从2018年10月6日到2018年10月24日，我在老师的推荐下，仔细阅读了C++primer plus第六版，并独立完成了书中每一章节的思考题和编程练习题，同时将学习体会梳理到了笔记本上。今天是2018年11月5日，我正在阅读和学习《introduction of algorithms》，过程中间我发现这很利于巩固C++基础，于是趁此机会将我之前所学的，按照自己的理解，摆脱C++primer中书本身的结构，建立自己脑中的C++知识体系，同时复习重点，弥补缺漏，将精华汇集于此。

目录

[写在前面： 1](#_Toc530485007)

[输入与输出、文件I\O 4](#_Toc530485008)

[一、 缓冲区与流 4](#_Toc530485009)

[二、 IO有关的类体系 4](#_Toc530485010)

[三、 标准输出cout 4](#_Toc530485011)

[四、 标准输入cin 5](#_Toc530485012)

[五、 文件IO 6](#_Toc530485013)

[六、 stringstream 7](#_Toc530485014)

[基本数据类型 7](#_Toc530485015)

[一、 整型 8](#_Toc530485016)

[二、 浮点型 8](#_Toc530485017)

[三、 字符型char 9](#_Toc530485018)

[四、 算术运算 9](#_Toc530485019)

[复合数据类型 9](#_Toc530485020)

[一、 数组 9](#_Toc530485021)

[二、 指针 10](#_Toc530485022)

[三、 字符串 11](#_Toc530485023)

[四、 枚举(enumeration) 12](#_Toc530485024)

[五、 结构 12](#_Toc530485025)

[六、 复合数据类型的复合 12](#_Toc530485026)

[存储持续性、作用域和链接性 12](#_Toc530485027)

[一、 C++中的文件 13](#_Toc530485028)

[二、 存储持续性 13](#_Toc530485029)

[三、 作用域 13](#_Toc530485030)

[四、 链接性 13](#_Toc530485031)

[五、 名称空间 14](#_Toc530485032)

[函数 14](#_Toc530485033)

[一、 函数签名 14](#_Toc530485034)

[二、函数返回 16](#_Toc530485035)

[三、函数指针 16](#_Toc530485036)

[四、内联函数 17](#_Toc530485037)

[五、 函数重载 17](#_Toc530485038)

[六、 函数模板 18](#_Toc530485039)

[类和对象 19](#_Toc530485040)

[一、 理解面向对象编程 19](#_Toc530485041)

[二、类的声明和定义 19](#_Toc530485042)

[三、类的构造函数与析构函数 20](#_Toc530485043)

[四、this指针 21](#_Toc530485044)

[五、类的作用域 21](#_Toc530485045)

[六、抽象数据类型 22](#_Toc530485046)

[七、 类的运算符重载 22](#_Toc530485047)

[八、 友元函数 22](#_Toc530485048)

[九、 类的自动转换和强制转换 23](#_Toc530485049)

[十、 类和堆内存 23](#_Toc530485050)

[类的继承 24](#_Toc530485051)

[一、 继承的实现 25](#_Toc530485052)

[二、多态公有继承 26](#_Toc530485053)

[三、函数静态联编和动态联编 26](#_Toc530485054)

[四、派生与动态内存 27](#_Toc530485055)

[五、抽象基类 27](#_Toc530485056)

[六、 私有继承 27](#_Toc530485057)

[七、 多重继承 28](#_Toc530485058)

[C++代码重用 29](#_Toc530485059)

[一、 组合 29](#_Toc530485060)

[二、类模板 30](#_Toc530485061)

[友元、异常和RTTI 31](#_Toc530485062)

[一、 友元 31](#_Toc530485063)

[二、异常 32](#_Toc530485064)

[三、RTTI 33](#_Toc530485065)

[标准模板库 34](#_Toc530485066)

[一、 string类 34](#_Toc530485067)

[二、智能指针模板 35](#_Toc530485068)

[三、STL 36](#_Toc530485069)

[写在最后 37](#_Toc530485070)

# 输入与输出、文件I\O

我们使用程序的开端和结束，都与输入或输出有关。程序要运行，输入数据是原始材料；运行的结果如何显示，就涉及到输出的问题。IO与我们编程的需求直接相关，因此将这一部分的知识梳理清楚有助于我们为用户设计方便的输入输出接口。

## 缓冲区与流

程序使用的数据是存放在内存中的，通过同样存放在内存中的可执行命令，在CPU中进行处理。而在数据进入程序的内存之前，它存放在磁盘中。磁盘的读写速度远远小于内存的读写速度，就像内存的读写速度远小于CPU的频率一样。因此如果让磁盘读写与内存读写同步，将大大降低内存的读写效率。缓冲区的存在，就是为了解决这一问题。

缓冲区介于内存与磁盘中的存放数据的地方。对于输入数据，它可以一次从磁盘中读取一块（512字节或更多），然后根据程序需要放入到内存中；对于输出数据，它从内存中接收数据，当存满一块的量后，一次性将这些数据写入磁盘中。这样的模式，使得磁盘不必与内存读写同步，从而提高了IO效率。C++的输入与输出就采用缓冲区的方案，无论是标准IO还是文件IO，都用缓冲区这样的机制。

流我们可以把它想象成水管，一头衔接程序，一头衔接数据源：对于输入流，它的输入端是缓冲区，输出端是程序（变量）；对于输出流，它的输入端是程序（变量），输出端是缓冲区。这样，流与缓冲区两个概念的融合，帮助我理解了C++的输入输出模式。

## IO有关的类体系

C++库中有IO的基类：ios类，它包括两个成员——流指针和缓冲区buffer。ios是基于ios\_base类的，在后者中，定义了许多有关设置流的格式的字段和方法，如：ios\_base::fixed、ios\_base::scientific、ios\_base::setf()、ios\_base::precision()等。输出流ostream和输入流istream由ios类派生出来，主要包括一些输出和输入的方法。我们最常用iostream类，多重继承了ostream和istream，并且包含了标准输入输出对象cin、cout等，还有一系列输出控制符：endl、hex、oct等。fstream则针对文件的IO：ofstream负责文件输出、ifstream负责文件输入，一些细节后面总结。还有字符串流，用于字符串的规格化或者字符串与其他数据类型的转化：ostringstream和istringstream。熟悉这些类的概念和使用是掌握C++的基础。

## 标准输出cout

Cout作为一个输出流对象，可以将其理解为与外部连接的缓冲区（外部不一定是磁盘，有可能是屏幕、打印机，甚至重定向过后指向的文件）。Cout理论上不是得到程序给他的输入就直接输出的，而是当他的缓冲区积攒满之后flush到标准输出的，这是cout与cerr的区别，后者不会通过缓冲区。当然我们可以强制刷新cout，调用flush方法或flush控制符。

1. cout基本使用

一般情况下，使用插入运算符<<将程序的数据放入cout缓冲区中，前提是数据有<<的重载或友元函数。Cout<<返回cout对象本身的引用，这意味着我们可以连续插入<<，对于多个输出来说很方便。此外，控制符endl也可插入，表示换行；flush控制符表示强制刷新缓冲区。

1. cout的格式化

有时候为了输出的整齐，需要使用固定宽度、对齐方式和填充字符。首先我们需要知道什么是输出的自动宽度：对于字符来说，自动宽度为1；对于整型来说，自动宽度是其用十进制下的长度（可用控制符hex或oct调整至字面值为16进制或8进制）；对于字符串，自动宽度是刚好放下各个字符的长度。浮点型稍微复杂，将在后面具体说明。Cout的成员函数width()和fill()帮助我们固定任何的宽度和想要的填充字符；此外，iomanip头文件也提供了setprecision()、setw()和setfill()方法。对齐方式在ios\_base::adjustfield中，可以使用setf方法实现左、右、前缀与值分开对齐方式。

1. cout与浮点型

浮点型我们知道有两种表示方法：定点法和科学计数法。ios\_base中有几个标记常量和setf、precision方法，支持浮点型的各种表达：显示小数点标记showpoint、定点法表示fixed、科学计数法表示scientific。这些都是调用setf()方法实现的。Setf方法有两个重载，分别接受一个、两个参数，这些参数是ios\_base的标记常量，并且返回一个标记量表示更改之前的各种格式，类型为fmtflag（用户使用时可以直接用auto）。Precision则设置浮点型的显示有效位数，接收一个int参数，返回一个更改有效位数前的有效位数。如果记录了这两者，那么我们可以对于一些有要求的实现格式化输出，输出完毕之后恢复原来情况，以免影响其它输出。

## 标准输入cin

要知道，cin对象自带的缓冲区中，直接存储的是字符,如果是命令行，则通过用户的Enter将字符串放入缓冲区中，之所以能用抽取运算符>>得到数值类型的变量，是因为cin对>>进行了重载，将字符串转变成为了数值类型。

1. cin的抽取使用>>

>>以缓冲区中第一个遇到的的空白符（包括space、Tab、换行符）为界，将其前面的字符串转化成>>后面的变量的类型，并赋给该变量。>>运算符返回cin对象本身的引用，这意味着我们可以连续输入。注意到，使用>>给string或char\*字符串赋值的时候，不会包含空格，并且将会把空白符滞留在缓冲区中。

1. cin的状态

如果，cin缓冲区中的字符不符合>>后数据类型的要求，此时cin就会像关闭的水龙头一样，不能再被抽取。Cin有一个标记位failbit，当cin遇到eof或类型不匹配的抽取时，failbit会被设置为1.为恢复cin的可被抽取的状态，使用其成员函数.clear()将failbit位设置为0， 以继续我们的输入。

1. cin.get()和getline()

如果使用>>运算符，字符串中若有空白，则string和char\*会断开，为了避免这个问题，我们可以使用.get()或getline()方法。.get()有三种重载版本，因为接收参数的两个重载版本只是读入字符长度的区别，因此主要理解无参数版本和有参数版本两类。

无参数版本的.get()返回缓冲区中最近的一个字符，它可以是任何字符，包括空白；有参数版本的.get(char,…)返回cin对象本身的引用，并且默认将换行符作为分界点，将换行符前面的字符串放入第一个参数，换行符仍留在缓冲区中。

Getline(istream& ,string& ,char Delim)就比较简单，接收两个或三个参数,返回istream&对象本身。他将分界符之前的字符串赋值给string参数，并将分界符扔出缓冲区。分界符默认为\n。

1. 标准输入的处理异常方法

当我们的输入不符合程序的要求时，cin的failbit会被设置，通常来说，我们使用cin.clear()将水龙头打开，然后用cin.get()将滞留的不符合要求类型的字符一个个扔出，直到\n;对于字符串来说，需要对输入空行进行判断，直接检查str[0]是否为空即可。如果我们想要提前结束输入，可以输入Ctrl+Z制造EOF，这样cin的水龙头就会关闭，不再输入数据了。

## 文件IO

文件的输入和输出可以类比于标准输入输出：文件输入类似于从键盘向程序发送数据的cin；文件输出类似于程序向屏幕发送数据的cout。尽管文件IO不是那么直观，我们依然可以将它想象成缓冲区与流结合的模式，缓冲区就像水库，流就像水龙头，用以连接文件（磁盘中）和程序（内存中）。处理文件IO的类包含在<fstream>头文件中，使用时加入即可。

1. 文件模式

按照存储形式的不同，文件可以分为文本文件和二进制文件。前者存储字符，后者则以二进制的机器语言存储。两者各有利弊：文本文件直观易读，但有时占用空间大；二进制文件结构紧凑，有时节约空间，但是不能直观阅读。比如一个比较大的int数，在文本文件中每一个数字都会占一个字节，而在二进制文件中只占int应该有的长度（如四字节）。应该根据不同需求使用不同的文件形式。

文件模式规定了流以何种方式绑定磁盘中的文件。在ios类中定义有in、out、app、ate、truc、binary六种标记位，分别表示输入、输出、追加、打开并移动到结尾、打开已有文件并冲掉原来数据以及存取二进制文件。我们可以根据自己的需求，结合fstream中的类，使用位运算符(&、|)实现对文件读写的不同需求。

1. 文件输入（读）

用到fstream中的ifstream类。与iostream不同的是，头文件中并没有自带类的对象，我们需要自己创建一个ifstream对象，绑定数据源文件。可以使用ifstream的构造函数，给入文件路径；或使用对象的.open()方法，给入文件路径。如果对于二进制文件，则要在第二个参数给入ios::binary。Ifstream对象默认打开模式为ios::in，所以对于一般的文本文件可以不加第二个参数。以下理解基于：ifstream fin;展开。

fin.open()将流与数据源文件绑定后，首先应该检查打开是否正确。调用fin.is\_open()返回boolean值，如果打开失败返回false，我们可以在分支语句下对此种情况进行处理——抛出异常或者发出提醒。如果打开成功，那么fin的使用和标准输入对象cin如出一辙。同样有状态检查，只不过文件的状态检查可以确定流关闭的原因：由于到达结尾，eof()返回true；否则原因可能是文件损坏。

如果绑定输入流和二进制文件，二进制紧凑的优点便能够凸现出来。使用fin.read((char\*) var, sizeof(type))可以直接将对象读入到变量中。第一个参数是将var强制转化为字符指针，这样的原因是二进制文件中是以机器字节进行存储的；第二个参数是目标类型的长度，指示读入多少长度的字节。但是对于含有虚方法的对象不太适用，导致部分虚方法无法使用。

1. 文件输出（写）

使用ofstream类，同样需要实例化一个对象。常用的打开模式有：out、app、truc以及binary。Out模式会按照给入路径创建一个新的文件，如果先前有同路径文件，则会删除原文件，重新创建新的空白文件；app是append的缩写，顾名思义是在已有的文件内容尾部继续添加。下面以ofstream fout;进行举例。

创建输出流对象时，同样可以使用构造函数绑定和.open()方式绑定，根据不同需求打开文件。如果不指定ios::binary，则以文本文件格式输出。具体操作与标准输出流对象cout如出一辙。输出二进制文件时，可以直接使用fout.write()方法，将对象写入文件，读的时候配合.read()方法，实现二进制文件的紧凑读写。

1. 文件随机存取

Random access是指可以任意输入或输出文件中的任何位置，不需要按顺序访问。同时如果我们想要对一个文件进行边读边写，fstream对象将会是一个好的选择。创建fstream对象时必须指定文件的打开模式，我们可以使用ios::in|ios::out来实现边读边写。Fstream对象将包含两个缓冲区和两个文件指针确定读写位置。

为了随机访问，我们应该可以随意移动读写指针。fstream的seekp()、seekg()分别可以移动输出、输入的文件指针，它们接收两个参数，分别表示偏移量（以字节为单位）和偏移原点（文件头、文件尾、当前位置）。Tellp()、tellg()分别用于返回当前输出、输入指针的位置。

1. 多文件读写

如果同时要对文件进行读写，那么只好创建多个流对象；但如果读写的文件有顺序，我们可以使用一个流对象来依次绑定同种模式的文件，这是因为可以尽量节约内存（流对象要有缓冲区）。同时，如果判断一个文件是否先前存在，我们可以使用输入流先尝试打开它，如果返回false，则说明文件不存在，我们再继续使用out模式写入；如果存在，我们则可以使用app模式来做添加。总之，结合多种文件流对象，可以方便地处理不同的IO问题。

## stringstream

字符串流又被称为内核格式化，顾名思义是有关于各种格式化的问题。Stringstream类被包含在头文件<sstringstream>中，主要有三个类：istringstream、ostringstream和sstringstream。

对于istringstream来说，同样它是一个输入流，我们把字符串想象成为数据源，则istringstream的输入端连接字符串，输出端连接程序（变量）。使用istringstream的构造函数将数据源（字符串）与流对象绑定，这样，我们就可以将字符串转化成为其他数据类型（使用抽取运算符>>），或使用getline()将原有字符串按照分界符进行分割。

对于ostringstream来说，作为一个输出流，我们可以用插入运算符<<将各种数据类型的数据放入缓冲区，然后调用ostringstream对象的.str()方法，返回string，以此来实现其他数据类型，向字符串的转化。

Sstringstream将两者进行了结合，即可用作其他数据类型转为字符串，也可用作将字符串转化为其他类型。

# 基本数据类型

对基本数据类型的理解，在数据结构的范畴内，将其视作有意义的不能再分的最小数据单元。我们应该用最恰当的数据类型去描述客观世界中的事物，为此应该要知道计算机内部如何存储管理各种数据类型，方才能谈及恰当使用。计算机内存储的最小单位是字(bit)，只能表示0或1，这决定了计算机中以二进制来管理数据；再大的是字节(byte)，有8位，取值为0-255共256个数，剩下大的都是字节或多或少的扩大：K\M\G\T…为了了解数据类型，最常用的还是字节，因此再大的不过多介绍。为了获取某种类型或某变量的字节长度，我们使用sizeof运算符。基本数据类型有数值和文本，文本其实是一种特殊的数值，这样来把常用的基本数据类型都整理下来。

## 整型

整型数据是计算机存储的最自然的数据类型，一段内存有多少字，它就能够表达2的多少次方个连续的整数。此外整型还可以做延伸，不光用来表达整数：比如boolean来表达true或false、char来表达字符等。按照字节长度来排序，整型有：

Char（1字节）<short（2字节）<int（4字节）<=long（4字节或8字节）<long long（8字节以上）。如果我们想知道各种类型的取值范围，使用头文件<climit>，其中有多个定义的宏常量，指示各种类型的上下界。此外，从short开始，我们可以将内存中第一个位用以表达正负号的功能取消，使整型成为unsigned，来扩大取值范围。当然这样做的前提是我们使用的数据不会出现负值。当数值超过我们使用的类型所能表达的最大值或最小值时，就会出现overflow或underflow。这是一种很隐蔽的错误，因为编译器不会检查出它。所以选择一种恰当的数据类型需要很多编程经验与思考，既不会浪费内存，又能够顺利完成任务。

整型的实际值和字面值有所不同。实际存储的整型一定是二进制形式的，但是用户在使用时可以根据自己的需求，输入输出字面值为：十进制、八进制或十六进制形式的整数。输入时，默认整型为十进制，如果是八进制，则在数字前显式加0；如果是十六进制，则在数字前显式加0x。输出时，标准控制符oct、hex或函数oct()、hex()帮助我们输出八进制和十六进制的数字，转换回十进制时用dec控制符即可。

整型的算术运算很有意思，会将算式中不足int的数据类型提升至int之后进行运算，再把结果转换为应有的类型，可能是为了加快速度，我觉得不必深究。

整型数字的常量在声明时，可以加后缀L来表示是long类型，否则只要没有超过int的取值范围，const的整型常量的类型为int。

## 浮点型

常用的浮点型有float、double、long double三类。按照取值范围和有效位数从小到大是这样排序的。计算机内部是如何表达小数的？我们可以用科学计数法的形式来理解：在内存中有指数部分，用以表达数量级；有基数部分，包括正负号和小数点在第一位有效数字后面的基数。Float、double、long double的取值范围之所以不同，是因为指数部分的取值范围和基数部分的精度不同，由此也导致了浮点型数据有效位数的不同。浮点型的字面值可以使用定点表示法，也可以使用科学计数法。在输入输出中都可用两中形式操作。关于输出，使用setf()、precision()流对象的成员函数来进行控制。浮点型的常量使用const进行声明和初始化，默认浮点型常量为double，如果有需要声明为float的浮点型常量，加后缀F或+即可。

## 字符型char

Char是字符character的缩写，他在机器内部是一个字节的整型，之所以能够表示文本，是因为有编码转换。最常见的编码是ASCII编码。因为256个不同的值足够表示键盘或英文，所以就使用了一个字节。需要注意的是，对于特殊的无法显式表达的字符，通过\进行转义：\a、\t、\n、\b、[\\、\](file:///\\、\)”等，分别表示响铃、制表符、换行符、回格、反斜杠、双引号等，在使用时要注意进行转义才可。字符可以和int实现互相转换，最c++的方法是static\_cast运算符，但是简单的转化是像调用构造函数一样返回想要的类型。比如char(65)可以输出A字符。

C++自带的库函数cctype中有很多操作单个字符的函数，比如判断字符类型、大小写判断与转换等等，后续字符串的操作会用到这些函数（一般是C风格字符串）。

## 算术运算

常用的加减乘除取模比较简单，需要注意的是取模的优先级是和乘除相同的。自增自减的前置和后置在单独使用时没有什么不同，但是在与其他语句同用时：前置先做改变自身值的操作，而后进行其他的操作；后置则相反。另外，比较tricky的是算术运算中的数据类型转化。理解这一点，其实可以联想函数的返回值和签名表，运算符也可以用函数的显式形式表示。在算术运算时，如果有范围更大的数据类型，那么将对操作数进行强制转化，并返回较大的数据类型的值。

# 复合数据类型

复合数据类型给了我们操作基本类型、自定义类的极大空间。最常用的、重要的复合数据类型有数组、指针、枚举、结构、字符串。将这些复合数据类型使用和特点铭记于心，并且注意一些细节的、有重要意义的地方帮助我使用C++解决问题。

## 数组

数组是存储相同数据类型的变量的复合结构——同质是它的特点。标记一个数组的是它的名字，下标运算符[]操作一个数组中的元素，或是在声明的时候作为数组的特征。初始化时一定要用常量来规定数组元素的个数。这在实际中其实很不友好，很容易造成空间不足或资源浪费。对元素的初始化可使用list-initialized，这也是为了和类进行同化的进步；赋值可以使用for loop或迭代器进行。

数组名需要我们加深印象的理解，它实际是数组首个元素所在的内存地址，却代表了整个数组在内存中的占用。因此，sizeof运算符返回的是整个数组的字节数，除以数组元素的类型，我们便可以得到数组的长度。描述一个数组的量：数据类型、数组名、数组长度三个，少一个都不能完全形容一个数组。

## 指针

1. 指针的直接使用

和数组一样，不能把指针单独叫做一种类型，它必须与其他类型合称为：什么类型的指针。指针实际上是占4字节的量，存放的是内存地址。声明变量时，\*是指针的标志，使用时\*表示解除地址（引用），从而得到指针指向的内容。为指针初始化或赋值要使用地址，比如new运算符或其他变量的取地址符&。并且，sizeof(ptr)只能得到地址的长度，而非地址内容的长度,除非使用sizeof(\*ptr)。->在对象指针中常用，另外指针在类中的用法我会归到类这一部分。

1. 指针与自由存储

C++面向对象的思想、程序运行时动态分配内存的特点，由自由存储和指针来实现。New运算符将返回从堆内存中分配得到的空间的地址，因此可以赋给指针，由指针来操作这部分内存。New可以分配一个类对象的空间，也可以分配一个对象数组的空间。格式为new typename或new typename[num]。与静态数组不同的是，num参数可以是变量，这意味着在编译完成时编译器并不知道应该分配多大的内存，实际上这个长度是由用户来控制的，静态数组不能这样做。New分配内存失败时会抛出异常，提醒用户堆内存不足或其他错误，可以使用try-catch语句块捕获异常来处理。

指针对单独对象的操作，在类的派生中和虚方法结合功能十分强大：自动适应最匹配的数据类型的同名虚方法。其他的，可以使用\*绑定指针，来代表对象，用.来使用成员方法或者成员变量；也可以直接用指针->来调用成员方法或使用成员变量。

指针对数组的操作，和静态数组相似，将指针视作数组名（实际上数组名也表示地址，但是指针数组不能用指针的sizeof计算数组长度），用[]取下标来访问元素。此外，静态数组、动态数组取值时，可以用作输入迭代器，使用STL或其他模板的算法，有时候比for循环要方便。指针用作操作动态数组时，有特殊的算术运算：+、-、++、--。加减运算只能和整型进行运算，返回的是从当前指针偏移type大小指定数目后的内存地址；++和—是对指针自身的改变，有前置、后置之分，后置的优先级大于前置和\*，结合性从左向右，到底是改变指针指向还是改变内容的值，可以加括号方便理解。

New运算符分配堆内存，使用结束后一定要delete，并且要和[]配对使用。但如果是用new定位运算符分配的栈内存，则不必使用delete操作。Delete释放的是指针指向的内存，而非指针本身的内存——指针本身可能是自动变量，也有可能是堆变量。总之，要记得使用delete将堆内存释放掉。

1. 指针与const

Const是常量限定符，它可以与指针构成两种常量：常量指针和指针常量。

常量指针在声明时：const type \* ptr; 意为指向常量的指针，或者说，对于ptr来说它指向的内容是常量，而可能对其他指针或变量来说，这个内容并不是常量。

指针常量的声明：type \* const ptr; 意为ptr本身是一个常量，不能被赋值和更改，只能被初始化，但并不意味着ptr指向的是常量对象。

Const的用法很多，在函数传参、函数返回、成员函数中都有应用。

1. 指针的危险

指针虽然方便，但如果使用不慎，会出现很多问题。最突出的是分配动态内存，忘记delete或delete多次，前者导致内存泄露，后者导致程序数据丢失或者崩溃，后果都是严重的。此外，指针由于可以操纵变量，如果多个指针指向同一内容，则牵一发而动全身，这样的变化是十分隐蔽的，所以最好该用const限定的就限定，并且尽量初始化指针而非为其赋值，多个指针同时指向同一对象的操作也要慎重使用。

## 字符串

C++有两种字符串：C风格字符串和string类。后者拥有更为完整、简单的操作，但是结构也更为复杂。两者都以单个字符作为最小单位，又都可以用字符数组的方式来进行各个元素的操作。

1. C风格字符串

C风格本质上是char数组，但是特殊之处是，在串的结尾字符为’\0’，同时这也是程序判断字符串结束的标志，这意味着如果我们想使用一个20字节长度的char数组来做字符串，那么这个字符串最多存放19个字符。使用strlen()返回的长度是不包括\0的。

Cstr的声明可以用静态数组，也可以用动态数组（记得delete）。为C风格字符串初始化或赋值时，双引号是标志，双引号内部的字符会放入内存，并在最后自动加入\0。其实这是因为使用双引号的字符串，实质上是头个字符的内存地址，那么作为指针的字符串名，自然接受一个地址。

输入字符串如果使用抽取>>，则在第一个空白处断开，要想避免此种情况，使用.get()或.getline()方法，当先输入数值后，由于缓冲区中还有换行符存在，直接.get()会吧换行符也读入字符串，因此在之前要先吐出换行符方可；字符串输出时，插入运算符>>自动将整个字符串内容输出，而不是只输出头字符，直到\0停止，这是普通字符数组和C风格字符串在输出上的区别（字符串名本质上是内存地址）。

Cctype函数库中的很多函数都接受c\_str，并且很有用：strlen()返回字符串的长度、strcpy()将字符串之间互相拷贝、strcmp()比较两个字符串的大小（按照标准顺序和ascii码，返回-1、0、1分别表示第一个字符串在第二个字符串前、相同、后三种情况）。使用下标运算符[]取出的字符也可以调用各种cctype库函数。

1. string类

string类是包装好的字符串，包含在头文件<string>中，名称空间在std，可以将其当做普通类型来使用，与c风格字符串很大的不同是不必分配长度，直接初始化或赋值即可。String对象的结尾标志也是\0，就算是空string也是有一个\0的，我们可以借助这一点来判断空串’\0’==str[0]? String的构造函数很多：使用c风格字符串、单个字符和长度等，有很多都是为了向容器类靠拢，因此很多构造函数都与容器类相似。

String的输入通常使用getline(istream&, string&,char delim=’\n’); 也可使用抽取运算符，但问题是不会读入空白；string的输出直接使用插入运算符

String对象的成员函数有：.size()、append()、.substr()、c\_str()等，分别获取字符串长度、在原字符串后添加、取出当前字符串的子串、转化为c风格字符串，并且重载了运算符+、==、!=、，合并和互相比较变得简单。

1. C风格和string的互相转化

String有char\*的转化构造函数，所以意味着C字符串可以直接转化为string对象；实际上我们使用string的重载运算符时，部分参数可以使用char\*，正是由于传参时使用了类型自动转换。String对象转化为char\*时，要先给char\*分配长度，使用.size()、strlen()和strcpy()。需要注意的一点是要比size多一个字符空间，用来存放结尾标志\0。

## 枚举(enumeration)

枚举是很特殊的复合类型，它的元素是同质的，并且必须是整型。使用枚举可以说明多种状态，给各种状态起了别名，因此是声明一系列符号常量的一种方法。如果我们不需要枚举的变量，则并不需要为枚举类型起名字。

1. 枚举的声明：enum name{name1=num1, name2=num2,……};

默认第一个枚举量为0，后面的枚举量增加1.并且枚举没有重载运算符，只有赋值运算符。最好不要用窄缩类型转化来为枚举变量赋值。

1. 枚举使用。常用在类的设计中，规定了状态变量的取值范围。其中涉及重名的冲突，可以使用作用域解析符::来解决冲突。

## 结构

与数组、枚举最大的不同是，结构允许元素异质。使用关键字struct声明和定义结构，初始化时使用{}list-initialized按照定义的元素顺序进行成员初始化。结构后来被拓展为类，因此在这里不需过多解读。结构能够作为类中的嵌套辅助，实现抽象数据类型（ADT）。结构的神奇之处在于，我们可以像使用普通类型的实例一样使用结构对象，比如赋值，就算结构中有数组，也可以实现拷贝（虽然数组不能直接拷贝）。

## 复合数据类型的复合

数组、结构、字符串有着共同的特点：元素是连续存储的。多种复合数据类型可以再次复合，表示更丰富的内容：数组的数组、指针的数组、指针的指针、结构的数组……

初始化时的{}能够帮助我们一层层理解复杂数据类型的构成。每一对匹配的{}都意味着一个复合类型的初始化。我们常用数组的数组来表示多维数组：type\* var[size]表示指针数组，而每一个指针可以去分配一串数组；type\*\* var则是指针的指针，使用的时候要有两个层次，首先要分配多少个指针，接着为每个指针分配多大的内存，在释放的时候也要有两个层次：先对每个指针的内容进行释放，最后释放多个指针本身占用的内存。二维数组，可以用行和列形象地去理解，这样在操作起来才能有迹可循。

# 存储持续性、作用域和链接性

程序中的变量、常量和函数都有自己起作用的范围和寿命，这与它们在内存中如何存储有关。并且如果我们想在多个C++代码文件中使用同一个变量或函数，这就涉及到了该存储方式的链接性。简要地说，C++对变量和常量的存储方式有三种：自动存储、静态存储、动态存储，这是从它们的寿命角度进行的分类；也可以分为全局变量和块变量，这是从作用域的角度；从链接性上讲，有外部链接性、内部链接性和无链接性。这些分类方式各有交叉，但是理解存储持续性是梳理清楚这些分类的基础。

## C++中的文件

变量的链接性反映了他是否能够被此文件外部的文件使用，因此有必要解释c++中的文件概念。这里说的是参与编译的有源代码的文件，而非之间IO中的文件。编译这一过程是由编译器完成的，使得代码成为目标文件，目标文件之间相互链接，就形成了完整的程序。C++编程的思想包含有接口/用户，意味着代码文件单独编译的可行性。C++文件主要有头文件和源文件两种，而源文件又可以分为类和调用类，这里我们暂且搁置，主要弄清楚头文件。头文件中存放着函数、类的声明，通常使用#include来将头文件内容添加到源文件中，相当于把两个文件合并在一起，且头文件的内容放在了源文件的前面。这样会出现一个问题，那就是当头文件中也包含头文件时，编译时会出现多次声明。为了避免这一情况的发生，通常我们会在头文件的顶端加上#ifndef…#define…#endif语句块来跳过重复编译（在vs中使用#prama once）。单独编译的好处显而易见：我们可以按顺序编译各个文件，如果对其中一个文件进行了修改，则只需要重新编译它即可，无需重复对其他文件的编译。提高效率与增强可移植性体现的淋漓尽致。

## 存储持续性

计算机内存可以分为两种：栈内存、静态内存和堆内存。前面指针部分我们已经知道，使用new运算符分配得到的内存为堆；在代码块外部声明的为静态内存；在代码块内部声明的使用栈内存。区别在于，堆内存中变量的作用域从声明处到delete处，如果没有delete，则在程序结束后释放（也有可能不释放）；静态内存是机器为程序预留的一部分内存，其中变量从声明处开始起作用，到程序结束时自动释放；栈内存中的变量全是自动变量，意味着出了声明处的中括号，将被自动释放，并且顾名思义，释放是有顺序的：按照声明顺序的逆序释放，因为后声明的变量压在了先声明的变量上面。

## 作用域

作用域（scope）是指变量或函数起作用的区域。变量的作用域有三种：局部变量（代码块中起作用）、全局变量（名称空间或文件中起作用）、函数原型作用域（只在函数原型中起作用）。函数的作用域有两种：名称空间或文件，以及类内部。

## 链接性

链接性有三种：外部链接（能被外部文件使用）、内部链接（在定义的文件中任意处使用）、无链接性（不能被超出代码块的区域使用）。显然，局部变量是没有链接性的。

1. 外部链接性

静态内存中的变量，在声明时会被自动初始化为0，或者在编译前初始化，乃至运行时初始化（类的静态成员）。所有函数都被默认为静态存储。静态存储的变量或函数，满足单定义规则（SDR：single definition rule），意思是如果在外部文件中使用该变量或函数，要在各个外部文件中都声明一次，并用extern 修饰；但是给该变量或函数定义的地方只能有一处（所谓定义就是在编译中为其分配内存）。

1. 内部链接性

实现静态变量或函数的内部链接性，只要在声明处加上static修饰即可，这样外部文件便不能用extern关键字声明和使用它了。

1. 无链接性

当静态变量在代码块内部使用static修饰后，它便是没有链接性的静态变量，意味着不能在代码块外访问它，但是它在内存中是存在的。若我们还有机会进入这个代码块，那么便还能访问它。需要注意的是，静态变量只能被初始化一次。

## 名称空间

复合数据类型或类中我们经常会遇到成员变量、函数重名的情况，如果不加以修改，编译器会产生疑问，从而使得编译不通过。如果是在不同的类中，我们可以使用作用域解析符::来进行区分，对于如果是有外部链接性的函数或变量，我们同样可以声明名称空间将其包含在内，再使用::来加以区分。

名称空间的声明为namespace NAME{……}，使用有两种：using声明和using编译。Using声明是将名称空间中某个变量或函数进行声明，using编译则是将整个名称空间内的东西一并融入到当前区域中。一般地，我们使用using声明要优于using编译，但是在比较简单的任务中，using编译要简便于using声明。

名称空间的链接性是外部的，可以跨文件直接使用，因此函数、类都可以放在名称空间内，在使用时要using编译或using声明即可，或者使用作用域解析符::

# 函数

函数作为C++的编程模块，最直观地展现了接口这一概念。之后我们会把接口这一概念一般化，甚至成为理解C++的一种观念。同时函数也是构成C++源代码的重要部分，它有三个基本要素：参数列表（signature，签名）、函数名、返回类型。这三者，在函数原型（prototype）和函数定义（definition）都有明确体现。拿main函数来说，其签名有两个：int和char\*，分别表示命令行调用时的参数个数和参数值；其返回类型为int，正常情况下向操作系统返回0，用户可以使用自己的返回值来表示不同的情况；显而易见的，其函数名为main。函数原型一般放在头文件中，与定义分开编译（内联函数除外），联系之前所说文件的单独编译，我们最好修改时保持接口不变，既能方便使用，也能提高编译效率。三个基本要素的作用是：函数名用以标志；签名有时也用于标志（重载，但是虚函数除外），更主要的作用是进行函数内外信息的交流；返回类型说明函数将会产生一个什么类型的数据，并提示用户有东西返回给调用的地方。下面以此来介绍输入参数、返回类型和其他的一些问题。

## 函数签名

签名顾名思义其实已经说明了它的作用，用以被编译器来识别。用户在使用函数的时候，按照签名来给入参数，将信息传给函数，从而实现函数内部对数据的使用。参数有两种：实参和形参；按照参数传入方式，分为值传入和引用传入。此外指针作为变量类型也可以作为参数传入，并且和普通参数有很大不同，此外数组的传入也比较特殊。Const限定符也常加在签名中，起着他的作用。

1. 实参和形参。

实参（argument）是指外部环境给函数的参数；形参（parameter）是指在函数内部起作用的参数。两者在根本上常常是不一样的：这是因为在非引用传参的情况下，实参对形参就像初始化，在函数定义的函数头处完成，当然除了类型一致，值一致，他们在内存中是完全不同的两个变量。换句话说，形参只是在函数内部的自动变量，出函数后被释放（栈变量）。

1. 按值传入和引用传入

如果不对签名中的形参做任何修饰，那么就是传值：实参将形参初始化后，两者有了不同的内存，除了值一样，便再没有了关系，当函数结束后，形参不复存在，实参丝毫不受影响。这一点要很明确，当我们不希望函数形参影响函数外部的实参，便采用这种方法。所谓引用参数，就是将形参作为对实参的引用，等同于引用变量：两者标识同一内存。所以形参在函数中的一举一动，实际上就是函数外部实参正在进行的操作。因此，如果我们需要使用函数对实参进行改变，或者减少时间空间的开销，就在签名中使形参成为引用变量&。

1. 传入指针

指针传入的实际上是按值传入，但是因为指针的特殊性——表示变量的地址，所以传入指针又被称为传址。这时形参与实参虽然再内存中是不同的，但是他们的指向是相同的。因此如果我们在函数内部对形参指针的内容进行访问，本质上是在访问函数外部的变量。所以传址是除引用参数外，另一种在函数内部改变外部变量的方法。需要注意的是，如果prototype中要求传入指针，那么我们在外部调用时要对普通变量加上取址符&。

1. 传入数组

动态数组使用指针标识，静态数组的标识——数组名实际上也是地址，所以如果传入参数是数组，函数内部本质上也是访问外部变量的。数组名做形参时，有一点特殊，要加[]以做标识。并且，形参实际上是指针，不能使用sizeof计算出数组的长度。数组的三大要素：类型、数组名、长度都要在传入参数时完整给入。对于长度，要另外加一个形参，或者用以指示数组开头和超尾的两个指针或下标（容器中的迭代器），这样的话我们就能在函数中对于数组全部或指定的一部分做操作了，比如快速排序。

1. 传入对象

如果按照传值的方式，形参会用实参的值在内存中重新申请空间，如果对象过大，这将十分浪费资源，不仅是空间，还有复制、析构所需的时间，因此对于较大的对象或者结构，引用参数或者传地址比较划算（虽然指针做形参也占四字节内存）。相伴随的危险是，由于粗心操作可能会改变原对象，为了避免这种情况，const限定被用进来。

1. Const与函数签名

Const作为常量限定符，在声明中的符号，是不能改变他在内存中的值的，尽管这个值可能会被其他量所改变。用到函数签名中，作用依然不变：对于形参来说是不能改变其中的值的。这对于传值没什么特殊意义，因为形参不会影响实参，但是对于引用参数或者传址操作，意义重大。如果我们不想因为误操作在函数内将实参改变，又想节约内存和时间，使用const加引用，或者参数使用常量指针，就可以一箭双雕。

实际上，对于没有特殊要求改变实参的函数，我们最好都使用const签名，这不仅因为他有前面所说的好处：节约资源保证不变，而且在外部调用的时候，如果实参是常量，const形参是可以接受的，非const形参是不能接受的；如果实参是变量，const与非const形参都可以接受。并且，如果实参与形参类型不匹配，非const签名会报错，但是const应用签名会生成一个临时变量，使用转化构造函数使得实参的值也能传给形参。这样看来，const简直是万金油（实际上其中也有引用的作用）。

1. 函数的默认参数

默认参数的作用是使函数调用变得方便：如果一个函数接受的参数很多，但是在很多时候都是一样的实参，那么每次调用时都将如此多的实参都输入，会显得代码臃肿，所以默认参数应运而生。默认参数使得我们在函数内部使用同样的形参，而又不必在外部给入实参。要注意的是，默认参数必须从签名表的右侧向左侧拓展，且中间不能出现默认与非默认交替出现的情形。如果这样做，在调用的时候，编译器将不知道省略的是哪个参数；并且，默认参数只在函数原型中出现，在函数定义中不必说明（内联函数除外）。当我们需要用其他实参顶替默认值时，就只像使用普通函数一样即可，这时便将会把实参传入，实现目的。

函数默认参数在代码重用中起到十分关键的作用，它和函数重载一起实现OOP中的多态的特性。

## 二、函数返回

在函数原型和函数定义中，函数名前都有类型名，这就是函数的返回类型。有时，我们能够以看待一个普通变量的观点来看待一个函数，因为它会返回一个类型（除了void）。如果返回的是const，或者引用，又意味着什么？是否有特殊用处？

1. 返回const限定的类型

如果是非引用返回，实际上return的东西，会被创建为一个临时变量，返回的便是这个临时变量。加上了const，相当于说明对于这个临时变量来说，值是个常量，使用临时变量不能改变它，就这么简单，但是可以和返回引用结合起来。此外，如果return后面的量是常量，则函数的返回类型必须是const。

1. 返回引用

返回引用意味着返回的东西和return后面的量共用内存，那么就有一个特别需要注意的问题：函数内部的自动变量会释放，也就是说return之后，如果是自动变量，那么内存已经被释放了，函数返回的东西已经没有意义了（在类的赋值运算符重载时会用到这一点）。所以，一般返回引用的函数，其传入参数中也有引用。最典型的是ostream和istream的运算符>>和<<，之所以可以连续抽取或插入，就是因为返回的是流对象本身（引用）。如果在引用的基础上还加了const限定，那么函数不能作为左值出现。返回引用的另一个作用仍然是节省时间和空间：非引用返回也要创建临时变量，如果有外部赋值，那么又要将值复制一遍。

## 三、函数指针

符号常量和变量都有名字，标识着他们在内存中的占用；函数在编译过程中，同样会形成可执行指令，存储在内存中，运行时由CPU调用、计算。函数名就是函数在内存中的标识，更准确地说，函数名是函数在内存中的地址，这让我们联想到了指针——记录着变量或常量的地址。事实上我们可以像使用普通指针一样，使用函数指针。前面说过，函数一般为静态存储，作用域为名称空间或文件，有外部链接性，如果加static则只有内部链接性。函数指针的声明为：

返回类型 \* 指针名(函数签名);如void\* pf(int, int);

1. 函数指针的使用

函数指针初始化或赋值后，可以像普通函数一样调用——加圆括号和实参便可。比如我们可以使用pf(2,3)来调用一个接受两个int参数、返回类型为void的函数。函数指针在模板算法中很常用，比如transform、各种\_if都要求函数指针的参数。

1. 函数指针数组

数组是同质元素的集合，函数指针的数组中的元素为签名和返回类型都完全一样的函数指针。有了函数指针数组，我们可以简便地调用这些函数。声明方法为：

返回类型 (\*数组名[size])(签名);

注意这里方括号[]放在了数组名的后面而非整个声明语句的最后面，这样做的原因是把函数指针数组视作普通指针数组，只不过后面加了signature而已。如果我们在声明时使用已有的函数地址来初始化，使用auto自动推断类型比较简便。

函数指针的主要作用还是在模板算法那里，在那里将会有各种对函数的调用、接口适配器的使用等等，以及在异常处理中对set\_terminal等都会借助函数指针这一工具。

## 四、内联函数

通常情况下，函数原型与函数定义是在两个文件中的——头文件和源文件，这是为了编译方便和保持接口不变。这同时也意味着原型与定义在内存中也是存放在并不连续的位置的：在调用的时候，系统会在当前指令处，从内存中找到函数定义，进入其中，执行完毕后再跳回当前指令。如果大型程序中函数定义过于多，那么查找位置将会成为一项耗费时间的工作。对于比较短的、经常使用的函数，将其进行内联，实际上等同于在调用处将函数体整个复制到后面，如同普通代码一样进行编译，这样程序运行的时候就不用在内存中跳来跳去，影响效率，但是同时也意味着程序会更占空间一些。使用inline在函数原型处，表示这是一个内联函数，函数定义可以跟在原型后面，也可以放在源文件中。需要注意的是递归函数不能使用内联，因为这样编译器会在调用处无限复制内部的代码，没有尽头。

## 函数重载

C++实现代码重用和多态（即多种形态）的重要手段就是函数重载：是指同名但是签名不同的函数。重载函数内部可以实现完全不同的功能，但是函数名一样，这提醒我们：对于实现相似功能的函数，可以使用函数重载。编译器是如何处理重载的函数的呢？他会根据用户在调用函数时使用的实参类型或个数，匹配最为合适的函数进行调用。

1. 实现函数重载

函数重载与返回类型没有太大关系，只和函数名与签名有关：必须同名，签名要么形参个数不同，要么形参类型不同，要么两者都不相同。实际上编译器会将重载函数翻译成和签名有关的内部表示以区分他们。

1. 函数重载需要注意

使用函数重载是否合法要从便编译器的角度去看代码：签名中的引用和非引用形参，编译器在函数调用时是区分不出的；引用的const修饰、左值、右值在调用时，编译器会根据最匹配的版本执行。

1. 函数重载与默认参数

有时候默认参数能够减少一些不必要的函数重载，如由参数个数不同导致的函数重载。这样做即可以减少冗余度，也能为计算机节省内存。但是这也仅限于形参个数不同的情况，如果形参的类型不同，则应该使用函数重载。

## 函数模板

如果我们期望对不同类型的参数进行函数内部一模一样的操作过程，这时使用函数重载也成了一种冗余：仅仅由于参数类型不同就要将相似的代码重新编写编译。函数模板便是用来解决这一问题的。实际上函数模板蕴含着C++编程的另一种思想：泛型编程。意为以类型为参数的编程：函数、类都可以有这种特性。函数模板的原理是：编译器根据传入的类型参数，将定义好的模板像模具一样生产出一个符合传入类型的函数实例（占内存的），这个操作为实例化（instantiation）。也就是说在程序运行时，如果我们使用了函数模板生产的不同类型参数的函数，他们是互为重载的，并不会缩小编译量，都在内存中有位置。

1. 函数模板的实现

使用关键字template<typename（or class） T>，来表示下面使用T作为类型参数的模板函数体。对于返回类型，由于输入类型的不确定，可能我们并不知道函数最终会返回什么类型的值。对于这一点，我们可以使用后置返回类型：auto和->decltype(……)，放在模板原型前面和后面，使用自动推断来得到想要的返回类型。

使用函数模板时，可以显式实例化和隐式实例化，前者是在调用前相当于做一个实例的函数的声明；后者则不做声明，直接调用：函数名<具体类型>（参数表）来生成实例函数并使用它。显式实例化的格式为：

template 返回类型 函数名<具体类型>(形参表);

1. 显式具体化

具体化（specialization）包括实例化和显式具体化。后者是用于当某个特殊类型运用模板不能完成指定的工作，但又不能破坏整个模板的内容，从而将该特殊类型的模板函数显式地写出来以实现目的。格式为：

Template<> 返回类型 函数名<具体类型>(形参表);

注意到和显式实例化唯一的区别是，template后面的尖括号。作用是告诉编译器，对于具体类型，不要使用模板来生成定义，而是使用自定义的函数体。

1. 函数模板的重载

当使用的函数模板接受不同个数的参数时，可以使用模板的重载。和函数重载很像，都是同名不同形参表，甚至可以使用默认参数。并且，在模板中不是所有的形参都必须是泛型（不过至少要有一个）。

1. 函数重载的匹配

如果我们用了相同的函数名，重载了多个非模板函数，又重载了多个函数模板，并且对于特殊类型还有显式具体化，那么，当我们调用一个重载函数时，真正调用的是哪一个呢？编译器有它自己的一套规则：如果调用的函数可以对应于非模板、模板和显式具体化模板，那么非模板函数优先于后两者，如果非模板函数不匹配，显式具体化模板优先于模板函数。

首先，排除参数个数不相同，以及无法自动转换的参数类型（比如指针类型和非指针类型）的那些重载；其次，寻找最佳匹配。匹配有三种：完全匹配、提升转换、标准转换。

完全匹配是指：调用中的实参类型与重载函数形参类型完全一样或只存在无关紧要的转换。如果有const和非const版本，那么常量与const匹配，变量与非const匹配（只对引用参数或指针起作用，对于非引用或非指针参数，const形参和非const形参会有二义性）。

提升转换在整型的算术运算中讲过，是说把较窄的数据类型转化为较宽的数据类型。这种转化不存在数据丢失。

标准转换是可以用static\_cast进行的转换，他可能存在数据丢失的情况。因此它的匹配级别最低。

1. 模板的缺陷

模板不能用于递归，这是他的一个很大的缺陷。因为在实例化时，编译器要不断生成对应类型的函数体，就像内联函数一样，递归会导致无尽的生成。此外，在模板内部对形参进行的操作，对于实例化后的实参来说是有意义的。这一点可以通过我们自定义类重载一些运算符完成，也可以通过对特定类型的显式实例化完成。

# 类和对象

C++语言可以很好地实现面向对象编程风格（OOP），区别于面向过程编程，OOP更多地在程序运行时分配内存、进行决策，通过对类的外部接口的调用来实现编程目的，并且拥有较强的可移植性。OOP的特性有：封装、继承和多态，现在面向对象已经成为了编程风格的主流。

## 理解面向对象编程

对象是某个数据类型经过实例化后产生的，面向它的编程，意味着使用它的接口，对它进行各种允许的操作，以完成不同的任务。这样就把设计程序的人分为了两种：设计类库的人和使用类库的人，当然可以自己设计类自己使用。前者关注的是数据的组织方式和可进行的操作，实际上计算机科学中就将数据类型定义为数据结构和基本操作的有机结合。这里的基本操作就是接口：实现两个系统交互的方法，在这里理解为提供给外部使用者对类实例的操作，以直接实现有需求的结果。OOP还和用户\服务器（C\S）模式有互通之处，在这里，用户指的是调用接口的程序，服务器指的是设计好的类库。用户向服务器发送消息（这里是对类外部接口的调用），以获得服务器的帮助。

设计一个类需要抽象。为了描述和模拟现实世界的事物，我们需要对事物的特性和我们关注的方面进行抽象简化，使用计算机能够理解的语言进行编程。通常设计一个类，可以从两方面入手：从我们要描述的属性，对他们进行有效的组织，并完成接口设计；或者根据我们想进行的操作，反向确定我们需要的属性，然后再用合适的数据结构加以组织。最后将属性和操作结合起来，便形成了一个类。根据我不多的编程经验，由目的驱动的类的设计比较容易理解和实行。

## 二、类的声明和定义

OOP的最终实现，就是让用户自定义的类型和语言本身的类型归为一致，反过来说也一样。C++的文件结构，使得类的声明和定义自然而然地分到头文件和源文件中，结合之前的文件单独编译，我们可以将头文件给类库的用户，如需更新，只要对源文件和类定义进行改动，保持外部接口不变，既能实现封装也能对用户进行兼容。

1. 类的声明

关键字class + 类名。类名一般首字母大写，以区别于普通变量。类的声明可以放在名称空间之内，以避免混淆多个头文件中的重名类。类的内部，包括成员变量和成员函数原型（其他还有友元函数等），分别说明类在内存中占用的长度、如何使用内存以及可以进行的操作。

访问控制：

默认对成员变量为private限制，意为只允许在成员函数或友元中访问；private常常用于成员变量的修饰和辅助外部接口的成员函数的修饰。Private是为了封装，既能保证用户在使用时不必考虑太多以至于不会使用，也能保护类库设计人员的知识隐私。

Protected修饰一般在多重继承时加以使用，并且一般不会用于成员变量的修饰，而是对外部接口的辅助函数进行修饰，使得多重继承的派生类不仅继承外部接口，也继承辅助函数。

Public修饰是类设计中最要紧的，它公布了用户可以使用的部分，一般只有接口没有成员变量。设计一个类的核心便是对外部接口的设计。

关于成员函数，如果我们使用const跟在他的原型后面，意为该函数可以被常量对象使用，或者有另一层意思：这个函数不会改变对象的成员变量。在可以使用const成员函数的时候一定要使用。成员函数本质上也是函数，因此符合之前我们对函数进行讨论的一切特征。它的作用域为类的内部，如果在外部使用，则要使用它的限定名。此外，因为涉及类的继承，我们在后面会介绍虚方法和动态联编的有关知识，也是对成员函数的补充。

通过类的声明，我们能够知道这个类的成员、接口和访问限制等信息。他能帮助我们了解类的组织，更好地使用类。

1. 类的定义

对类的实现，最好放在源文件中：要#include类声明所在的头文件。这里我们对各个成员函数进行定义。如果我们将函数定义跟在头文件中的函数原型后面，那么默认这个函数是内联函数。如果我们想让内联函数也和其他函数一样统一放在源文件中，则要在函数名前面使用inline进行修饰。在源文件中的成员函数定义，都会有类的作用域解析符放在前面，以表明这是哪个类的成员函数。

要注意的是，类定义好后并不会在计算机中占用内存，因此重要的一点就是，不能在类的声明中对成员变量进行初始化或赋值，哪怕是静态变量和常量（除了const int）。

类定义好，并完成编译后，可以生成.dll（动态链接库）和.lib（静态链接库），将它与头文件放在一起，就完成了对整个类的封装，可以移植到任何一个我们要使用的工程中去了。

1. 类的实例化——对象

当我们要使用类的外部接口时，首先要实例化一个对象，对象是真实在内存中存在的，调用对象的外部接口，要用到对象的成员变量，方可执行。如果有同一个类的多个对象，他们的成员变量在内存中地址不同，但是他们的静态变量、静态常量和成员函数是在同一个地址的。所以调用接口必须要说明是哪个对象的接口。

## 三、类的构造函数与析构函数

类的构造函数和析构函数是每个类必须要有的外部接口，如果用户没有显式地写出，则编译器会自动补充上默认构造函数和默认析构函数。构造函数与析构函数没有返回值，顾名思义，前者是创建一个对象时调用的，用以实例化一个类；后者是释放一个对象所占用的内存时调用的。

1. 构造函数

函数名为类名，无返回类型。作用是为成员变量分配内存，进行初始化或赋值。需要特别注意的是，如果用户删除了默认构造函数，自定义了新的构造函数，则在使用类时无法调用无参数的构造函数。一般来说，无参数的构造函数一定要有，接受其他参数的构造函数也要有，他们之间形成重载，供用户自己选择使用。如果涉及堆成员，则考虑到深浅层拷贝，还要注意拷贝构造函数的重载，或许添加引用次数，或许进行堆成员的复制。

构造函数最好要用到另一种技术：成员列表初始化。因为这个技术只能在构造函数中使用，且有着一定优势：相当于初始化而非赋值，因此不会产生临时变量复制又删除。使用方法为在构造函数名后加冒号：而后对成员按照类声明中的顺序，根据参数表进行依次初始化。

构造函数的调用还分为隐式调用和显式调用。隐式调用是在声明对象时直接在对象名后加圆括号，只能用来初始化对象；显式调用是常规调用函数的模式：函数名加参数表。两者的区别是，隐式调用不会产生临时变量，而显式调用会产生临时变量，因此单论初始化效率，前者较高，但是后者可以用来赋值，前者不可。像使用普通类型一样，我们也可以使用类指针和new来自由存储对象。

无参数构造函数重要的一点就是，可以方便地使用对象数组。因为数组的初始化要先调用默认构造函数，之后按照初始化列表形成临时变量，再对每个元素进行赋值。如果有无参数的构造函数，则对象数组可以在初始化时调用它；如果没有无参数的构造函数，那就没有对象数组。所以谨记自定义类一定要有无参数构造函数（默认参数也可）。

1. 析构函数

函数名为~CLASS，无返回值。作用是释放对象占用的内存。如果没有重载，则编译器自动生成一个析构函数。如果对象是自动变量，则无需显式调用析构函数，这是因为自动变量过期后在栈中自动调用了析构函数；如果对象是堆变量，则要使用delete，作用就是调用析构函数。当类中含有堆成员时，一定要重载析构函数，在其中delete堆成员，否则很可能导致内存泄露。

## 四、this指针

This指针是对象形成时，在类内部隐藏的、自动存在的指针：它指向本对象所在的内存地址，并且只能在类内部使用。当我们需要在类的成员函数中返回当前的对象或对象指针时，this指针起到了关键的作用。而且，如果在类的成员函数中使用到了另一个对象作为参数，虽然可以不使用this就能够分别访问本对象和另一个对象的成员或函数，不过我们仍然可以使用this来做更明显的区分：因为类的成员函数在内存中只有一个副本，只有给入正确的对象，才能得到正确的结果，而this就给了我们这样的提示：调用的是当前对象的成员。

This指针在赋值运算符重载中也有用处，在运算符重载一章再详细说明。

## 五、类的作用域

成员变量由于大多数情况下使用private修饰，其作用域在类的内部，无链接性；成员函数使用静态存储，所有对象共用，作用域为名称空间或者类内部（protected或private成员函数）。需要特殊注意的是，类的符号常量成员，如果单纯用const限定，则会在每个对象中都出现一次，占用不同的内存，这是没有必要的。有两种办法解决这个问题：

其一是使用枚举常量。作为一种表示常量的复合类型，枚举如果在类中声明，则作用域就在类内部，并且枚举常量是属于整个类的，不会在各个对象中创建。

不同的枚举类型，他们的枚举量可能会冲突，为了避免冲突，可以在声明时在枚举名前加class或struct，将其视作作用域为类内的枚举，之后使用作用域解析符即可用于区别。

其二是使用静态常量。即static const。我们知道在类声明中不能对成员进行初始化或赋值，因为此时它并没有内存。但是加了static就让常量使用了静态存储，这一部分是在编译的时候就分配好的（和成员函数一样），因此可以直接初始化。静态符号常量是类的一部分，不会在每个对象中都创建，所以不能被某个对象调用。Static还可以用来修饰成员函数，这使得该成员函数不是被哪一个对象所调用的，而是被类所调用的，原理和static符号常量一样，同时静态成员函数也不能使用其他成员变量，因为并没有对象能调用他。

## 六、抽象数据类型

在数据结构与算法分析中，抽象数据结构常常用来说明某种数据结构的特点和操作，而不关注具体的数据对象到底是什么类型，因为无论数据类型都不影响这种操作模式。这和类定义有相通之处：我们只关注接口，并不关注实现接口的细节。抽象数据类型（abstract data type，ADT）更通用的作为一种容器出现，其实这里使用类模板能够更好地说明这一点，不过我们也依然可以使用typedef来起一个Item的类型名字，在ADT的实现中使用Item，之后要具体化的时候改变Item所代替的类型即可。

## 类的运算符重载

要想统一常规数据类型和自定义类的使用，理解和使用运算符重载是必不可少的。运算符实质上也是函数，运算符的重载相当于函数的重载，但是又有他自己特殊的特性。首先我们要明白，并不是所有的运算符都可以重载。常用的可以重载的运算符有：+、-、\*、/、%、[]、=>、<=、<、>、=、&、>>、<<、!等等，但比如：sizeof、static\_cast<>、dynamic\_cast<>等运算符就是不能重载的。而重载运算符的格式也比较特殊，它的函数名为：

operator运算符(参数表)。举个例子，x+y，实质上等同x.operator+(y);但明显前者更为美观易读。

1. 理解类对运算符的重载。最好的例子就是抽取运算符和插入运算符。我们知道对于整型和字符型的cout<<效果是不一样的，虽然都是以文本的形式，但是他们的内部形式却完全不一样，这说明<<对整型和字符型在函数定义的内部肯定是不同的。之所以这样的调用呈现多态，正是因为ostream类对整型和字符型进行了<<的重载。
2. 注意运算符重载的返回类型。其实这一点并不是运算符重载独有的，而是所有函数都应该注意的。那就是当我们想使用连续的算术运算或者赋值时，为了节省临时变量的开销，可以直接返回结果的引用
3. 重载运算符一定要使用自定义类，或是参数表中含有自定义类。这样做的原因是为了避免和已有类型的运算符发生混乱。并且，多数非自运算的运算符重载成员函数，最好在后面使用const限定，一是为了常量对象可以使用，二是为了确保不会更改对象成员。
4. 运算符重载的成员函数是有顺序的：对象必须在运算符的左边，参数在运算符的右边。如果对于单操作数的运算符，如取反，那么成员函数参数表为空。

## 友元函数

由于成员函数中运算符重载的顺序问题，导致自定义类只能在运算符左边，这样有时很不方便。友元函数在类内部声明，但作用域却远不止类内部。它能够访问类的成员变量，甚至进行修改。但是要注意的是，友元函数并不是成员函数，因此并不是被某个对象调用的，也不可以用const限定它。调用友元函数和调用一般普通函数一样的，只要给入符合类型的参数即可。友元函数的声明为：

friend 返回类型 函数名(参数表);

最为常见的友元函数就是抽取和插入运算符。因为这个运算符最好是让流对象在运算符左侧、自定义类对象在右侧，成员函数重载运算符无法实现，所以要用到友元函数：

friend ostream& operator<<(ostream& os,const class& obj);注意这里的返回类型带引用，参数表带引用和const，这样的效率是最高的。后面我们还会说到友元类、友元成员函数两种友元，同样是使得外部能够访问类内部的私有部分，他们有各自需要注意和独特适用的地方。

那么对于运算符重载，我们到底应该选择成员函数还是友元函数来实现呢？为了使用起来方便，当然是符合交换律的运算，各种顺序都要满足。所以对于不同类型之间的算术运算，可以友元函数与成员函数并用，都对运算符进行重载，以实现方便、友好的调用。

## 类的自动转换和强制转换

如果自定义类的构造函数中，有使用其他类的对象作为参数而对本类进行构造的重载，那么这种构造函数被我们称为转化构造函数。存在转化构造函数，很有可能会出现意想不到的类型隐式转换（出现在赋值、传参、临时变量等地方）。如果我们想要避免这种意想不到的隐式自动转换，则需要在它对应的构造函数前加上关键字explicit，这样就可以避免转换构造函数的隐式调用了。隐式转化很有意思，因为可能引起二次转化。如在为自定义类对象赋值的时候，使用了其他不在转换构造函数中的类型，但可以转化为转换构造函数的类型参数时，如果没有二义性，则可以转化成功。

1、其他类型转化为自定义类型：

将其他类型转化为自定义类时的显式转化为调用类的构造函数，这一点很像普通类型的强制转化，为类型名加圆括号，放参数。

2、自定义类型转化为其他类型：

这被称为转化函数。它是类的成员函数，形式为：

operator 目标类型()const;

它的形式表明，无返回值，没有参数表，不改变原对象。这说明类型转化不影响源头类型。调用转化函数，和普通类型的强制转化一样，用本类对象去构造目标对象：

目标类名(本类对象);

1. 友元函数和自动转换

在运算符重载时，我们有两种选择：成员函数重载或者友元函数重载。如果同时我们有和运算类型相同的转换构造函数，那么情况会变得多种多样。如果没有使用友元函数重载运算符，意味着做运算时，其他类型不能放在运算符左边，因为此时其他类型中并没有对自定义类型的运算符重载；如果有了友元函数，并且接受的两个操作数均为自定义类型，此时运算符左侧可以是其他类型的数据，这是因为会首先将其他类型通过转换构造函数得到一个临时的自定义类对象，再和右操作数的自定义类对象进行友元函数的调用。当其他类型的数据放在运算符右侧时，无论友元函数还是成员函数重载，都能够实现正确的运算。

## 类和堆内存

类的成员变量中可以拥有指针类型，而且因为类的成员变量的封装性，在外部使用自动变量为指针成员赋值是很不科学的。并且，面向对象编程的一大特点就是在程序运行时分配内存而非编译时分配内存，这样就可能需要在类的构造中使用new来利用自由存储；继而在析构函数中要使用delete来释放自由存储。但是事情没有我们想象的这么简单，在对象的使用过程中，临时变量层出不穷，拷贝赋值也是常常使用到的，不给予堆内存特殊关注的话是无法实现正确的结果的。

与堆内存有关的类的接口有：构造函数、析构函数、赋值运算符重载、拷贝构造函数，这四者都有默认形式。下面我们依次说明它们存在的问题和如何对他们进行改进。

1. 构造函数的问题和改进

默认构造函数会把类的所有成员变量初始化为任意可能的值。如果成员中有指针存在，那么默认构造函数就会使之成为野指针，如果对其进行释放，可能会将程序中重要的数据释放掉，从而造成严重的后果。对其进行改进，就是将指针成员初始化为空指针nullptr，在没有为该变量new堆中内存时，他一直指向空处，delete也不会有问题出现。如果重载构造函数，即为指针new堆内存，那么构造函数没有什么问题。

1. 析构函数的问题和改进

默认析构函数不会释放堆内存，所以，一定要配合前面构造函数new或new[]的分配，在重载析构函数中显式地delete或delete[]指针指向的内存。

1. 赋值运算符重载的问题和改进

默认的赋值运算符，是浅层拷贝：意味着只将右值的自动存储区域的类的成员变量复制到左值。指针本身作为一个自动变量，也会被复制到左值。问题出现了：两个对象中的指针指向了同一个堆内存。结果可想而知：当一个对象被释放后，堆中内存已然释放，如果再对它进行释放，则会重复释放堆内存，这是严重的错误。为解决这一问题，我们需要明确赋值运算符到底是要实现浅层拷贝还是深层拷贝，因为两种拷贝的重载不一样。首先应该解决自身赋值给自身的情况，这种比较简单，只需返回\*this即可。

浅层拷贝需要用到引用次数这一成员，当引用次数为零时，方可在析构函数中释放堆内存，否则不释放。在赋值运算符重载时，要对引用次数加一。

深层拷贝需要首先释放this的堆内存，然后重新new堆内存，再从目标的堆内存中拷贝过来。

1. 拷贝构造函数的问题和改进

拷贝构造函数就是使用同种类型的对象来构造一个新的对象。默认的拷贝构造函数是浅层拷贝，问题和赋值运算符一样，有可能导致多次释放同一指针。同样按照要求可以浅层或深层拷贝。浅层拷贝将目标指针复制，引用次数加一；深层拷贝用指针指向新分配的堆内存，再从目标中拷贝过来。

重载后的赋值运算符，返回类型一定要是引用类型，不仅是为了减少临时变量的使用提高效率，也是为了避免因为析构临时变量而误将原对象的堆内存释放，导致赋值失败，甚至目标对象的数据丢失。

1. 对象指针的使用

如果使用对象指针指向一个堆内存中的对象，而对象内部也有堆成员，那么我们知道在该对象使用结束后，需要delete它。这时系统会首先释放非堆成员（虽然现在他们都在堆上），再进入类的析构函数，将堆成员释放掉。

如果我们使用了一个堆上的缓冲区，并且在缓冲区上使用定位new分配了一个对象，那么当我们释放缓冲区时，不会调用对象的析构函数，这就导致了内存泄露。解决办法是，显式调用对象的析构函数~CLASS。这是迄今为止首次使用显式的析构函数。需要注意的是析构对象时，要按照与创建顺序相反的顺序来析构（有些像自动变量的释放），并且，当缓冲区上所有的对象都析构后，缓冲区才能被delete。

# 类的继承

继承是面向对象编程的重要特性，它使得代码重用增多，二次开发工作变得容易。这是因为我们可以在前面已有的类中新增自定义的功能。无论是对于有源文件的类，还是已经编译成库的类，我们都可以将它们继承。用于继承的称为基类（父类），生成的新类称为派生类（子类），派生类拥有基类允许继承的接口和成员，并且在此基础上新增了自己的特性成员，或者修改父类的接口（虚方法）。这一部分我的展开思路是：继承如何实现、多态公有继承、虚方法与动态联编、派生与动态内存、抽象基类的使用、私有继承的实现、多重继承。

## 继承的实现

当我们写了一个概念较为宽泛的类之后，又想在此基础上细化一个范围更具体的类时，如果与大类有相通的地方，那么我们就可以选择继承。类与类之间的关系有很多，简单而常见的关系有：is-a、has-a、is-like-a、is-implemented-as-a、uses-a等等。

顾名思义，最简单的公有继承就符合is-a的关系，比如香蕉是一种水果。能够套用这种模式的类，我们就可以使用公有继承。Has-a表示某种类包含另一种类，比如汽车包含轮胎，实现这种模式，我们可以使用多重继承或者combination的方法；is-like-a描述两种类之间一种比喻的关系，但是这只说明在某些方面他们相同，其他方面可能根本没有关系，为此可以使用抽象基类的方法将两者的共同特点抽象得出，然后分别继承抽象基类；uses-a的类关系接近友元，意为某种类可以操纵另一种类；is-implemented-as-a表示某一种类可以被当做某一种类来实现，如数组可以用作栈，或队列，但是这并不意味着，栈或队列是一种数组。理解这些类之间的模式，能够帮助我们更方便快捷地构建项目。

派生类在声明的时候就要体现继承关系：

Class 派生类：继承方式（有公有、私有、保护三种）基类……

{……类的内容}

这里的冒号表示继承关系，继承方式有public，表示公有继承，这样基类的public部分，将成为派生类的public部分，基类的protected部分派生类可以直接访问，但是外部不能访问，基类的private部分也会被派生类继承，但是派生类不能直接访问；private，表示私有继承，基类的protected和public部分将成为派生类的private部分，可以直接访问，基类的private部分将会被继承，但是不能直接访问。Protected作为继承方式用于传递内部接口，并且作为访问修饰时有特殊作用。

理解基类和派生类的关系：在派生类中，基类作为隐藏的一部分，甚至有些像成员变量一样的存在，只不过这个成员变量没有名称，只是作为组件以其类名表达。我们可以使用基类名加作用域解析符::来使用基类中的东西。使用组件化的角度理解基类在派生类中的地位，我们就好理解派生类在构造的时候会首先调用基类构造函数、析构的时候最后调用基类的析构函数的原因。我们还记得类的构造函数中，我们可以使用成员初始化列表方便地进行类的构造，在派生类中，使用成员列表初始化是必不可少的：只能通过这一手段来对基类组件进行初始化，因为基类组件的成员，派生类是无法访问的。如果没有显式地将基类组件进行成员列表初始化，那么程序会自动调用基类的无参数构造函数初始化基类组件。我们对成员初始化列表有了新的理解：调用组件的构造函数。

公有派生类和基类还有重要而又有趣的一个特点：基类的指针或者引用可以指向或引用一个派生类的对象，但派生类的指针或者引用，却不能指向或引用一个基类对象。这一点，是多态实现的重要基础。这样的做法，在编译器看来是可行的：公有派生类对象继承了基类的所有外部接口，所以使用基类指针操纵派生类对象不会僭越；反之，如果派生类指针指向一个基类对象，在编译器看来，他应该能够使用派生类的所有外部接口，但是一个基类对象是没有派生类全部的外部接口的，所以无法通过编译。有了这个关系，意味着在某些接受基类指针或基类引用作为形参的函数中，我们可给入派生类对象。

## 二、多态公有继承

多态的含义是对于不同的类型，同样的方法呈现不同的形态。在继承中，多态体现为派生类和基类相同的接口会有不同的结果出现。这不同于函数重载，因为重载要求特征标（签名不能相同），而多态的接口应该完全一样。如何实现？通过虚方法来完成。

1. 虚方法是什么

如果我们在基类的外部接口原型前，加上virtual修饰，那么就意味着，派生类如果重写了同名且同签名的外部接口，从基类继承得来的该接口便被隐藏了。外部调用派生类对象的该接口时，便会按照派生类重写来完成。通常使用virtual时，会在基类和派生类的要重写的接口原型前都用virtual修饰，这是因为在未来如果派生类作为基类时，我们仍旧可以使这个接口隐藏，否则外部调用时会使用继承关系最近的类的虚方法。

注意，只需在外部接口原型处使用virtual修饰，在接口定义处不需要。

1. 虚方法的作用

当我们在外部使用派生类时，调用虚方法接口，会使用派生类对象的新的接口。这是理所当然的，但是神奇的是，因为基类指针或引用可以指向或引用派生类对象，当我们用这样的指针或者引用调用虚方法接口时，它竟然使用的是派生类的虚方法！这一点的原因我们在动态联编中进行理解。但是在这里，我们常用的就是使用基类指针数组统一处理一大批不同派生类对象时，可以方便地调用他们的虚方法，得到正确的结果。

## 三、函数静态联编和动态联编

联编的含义是：将外部被调用的函数和其定义所在的内存地址进行绑定，以在程序运行过程中实现指令的调用。静态联编顾名思义就是在编译的过程中，将联编完成——编译完成后每一个函数调用都知道自己应该跳到内存中的位置；动态联编则不是：只有在程序运行时函数调用才知道自己应该跳到内存中的哪个可执行代码块处。虚方法就是实现动态联编的一种手段。前面说过基类指针或者引用指向或引用派生类对象时，编译器仍将该指针或引用视作基类，如果有virtual方法存在，编译器就不会贸然将接口调用与基类的方法绑定。因为此时指针指向或引用到的，是真实的派生类对象所在的内存。这便是动态联编的实现。

虚函数的工作原理是：对象中会存在一个隐藏的东西：虚函数表（vtbl），他来指向对象的接口应该绑定的接口定义在内存中的位置。当外部调用对象的接口时，会在vtbl中寻找最新的同名函数，从而实现调用。Virtual将基类的同名接口地址在vtbl使用新的函数地址顶替，故基类的接口不会被找到。因为基类指针或者引用指向派生类对象时，调用接口会进入该派生类对象内部，所以会有vtbl查找的过程。

C++默认静态联编是为了减少不必要的浪费：动态联编使用vtbl占用内存，且花费时间查找，对于不需要的类来说是累赘，所以静态联编是C++默认的函数联编方式。

Virtual实现了动态联编，如果在派生类中重写了接口，那么原接口会被顶替，如果没有重写，自然不会被顶替；如果在派生类中使用了同名的重载，那么基类的同名接口也会被顶替，使得无法使用和基类接口同样的签名表在外部调用派生类的接口：因为vtbl是按照函数名来顶替的，不会考虑签名的不同。所以如果基类接口有重载，且都有virtual修饰，那么派生类也一定要显式地重写所有的重载，哪怕不会做修改，如果不这样做，基类的重载就被覆盖了。（当然可以在基类中不用virtual修饰，但这样不利于继承）。

## 四、派生与动态内存

在类的设计中堆成员需要特别注意。在继承中出现的问题，是派生类相较于基类多出了堆成员。以下讨论的均为这一情况。

1、析构函数。当派生类中包含堆成员时（不是基类组件的堆成员），我们知道在其析构函数中要delete该成员。这时要注意的是，基类的析构函数应该是虚方法。这是因为当我们使用基类指针指向派生类对象时，编译器将其视作基类，释放它时，如果基类析构函数不是虚方法，则只调用基类的析构函数，而基类的析构函数中并不会释放派生类才有的堆成员，从而造成内存泄露。Virtual一加，问题迎刃而解，道理还是vtbl的道理。为了防止不经意犯这样的毛病，对每个基类的析构函数都要使用virtual修饰。

2、拷贝构造函数。派生类的拷贝构造中，形参是派生类的常量引用。我们如何使用派生类的常量引用，将基类组件进行初始化呢？唯有成员初始化列表，且调用的是基类的拷贝构造函数，传入的是派生类常量引用。这是因为基类的拷贝构造函数形参是基类的常量引用，通过前述知识可知这个引用是可以引用派生类对象的，故能够实现。基类组件初始化后，考虑到深层拷贝，要对派生类的堆成员进行内存分配和复制，这与普通类的堆成员拷贝没什么不同。

3、赋值运算符重载。基类的赋值运算符重载虽然是外部接口，但是不会被派生类继承，原因很简单：签名不同——基类的赋值运算符重载形参是基类常量引用，派生类的赋值运算符重载则是派生类常量引用。在派生类的赋值运算符重载中，如何对基类组件进行赋值？这需要调用基类的赋值运算符重载了，一定要加上基类的作用域解析符，否则默认为派生类的赋值运算符重载，形成没有尽头的递归，传入的参数还是派生类常量引用，道理是基类引用类型可以引用派生类对象。

## 五、抽象基类

抽象基类（abstract based class，ABC）指的是无法被实例化为对象的类。它只描述了属于此类应该有的外部接口，甚至可以不去定义某些因具体类而异的接口。对于抽象基类的每个派生类，都必须重写这些接口，所以在抽象基类中，这些接口应该是virtual的，并且有定义也没用，因为它既然被virtual，自己又不能被实例化为对象，所以肯定不会被调用。为此，我们在这种接口的原型后加=0，以表示不用函数定义，这种函数被成为纯虚函数。相应的，只要有一个纯虚函数，则这个类就是抽象基类。

何时使用抽象基类？一种情况就是多个类拥有共同的特性，符合is-like-a模式，这时可以将他们的特性抽象为基类，并从基类将之派生；另一种情况是，当两个类或多个类有is-a模式的影子，但是直接使用继承又会把问题变得复杂（比如圆是椭圆的特例，但是对椭圆的change和对圆的change有所不同），这时我们可以将他们的共同点再进行提炼，形成抽象基类，再分别继承抽象基类，使问题变得简单。

## 私有继承

公有继承符合is-a模式，是因为派生类同样有基类的外部接口。私有继承的派生类在声明时：

Class 派生类名：private 基类名{……}

他会将基类的外部接口和protected部分，在派生类中以private的限制出现，意味着外部不可以使用派生类对象调用基类外部接口，同样不可以使用基类的指针或引用来指向派生类的对象：因为对于编译器来说，该对象不能使用基类的接口。为此，私有继承符合的是has-a模式。这个角度是在派生类中，将基类视作组件的观点。

既然是has-a模式，那么在派生类中应该可以完全使用基类组件：

1. 使用基类组件的方法

私有继承中，基类的外部接口和保护部分都可以直接在派生类中使用，无需加作用域解析符（因为名正言顺继承而来）。但是对基类的private部分，派生类无论如何不能使用。

1. 使用基类组件对象

使用向上类型转化。从派生类对象转换为基类对象称为向上转换，它是可以隐式调用的，因为可以将派生类对象的基类组件拿出来；从基类对象转化为派生类对象是向下转换，他不能被隐式调用，并且大多数情况下，这种转换是有害的。综上，如果我们在派生类的实现中需要用到基类组件本身，就使用显式向上转换。这样，使用基类部分本身，则可以使用其private部分了。

1. 使用基类的友元函数

友元函数不是类的成员函数，因此并不会被继承。并且在基类的友元函数，只能够用于基类对象（即使是接受基类指针或引用的友元函数，倘若传入派生类对象，也只有基类的效果）。所以倘若我们要在派生类中使用基类的友元函数，首先要拿出基类组件本身（即上述的向上转化），之后将其作为实参调用基类友元函数。注意这里如果友元函数接受基类引用，最好也将this转换为基类引用，这样可以避免临时变量，提高效率。

1. 保护继承

保护继承是私有继承的变体，同样适合has-a模式。不同之处在于，protected继承将基类的protected部分放在了派生类的protected部分，而private继承将该部分放在了派生类的private部分。纵观public、private、protected三种继承方式，我们能够理解访问限制的互相约束原则：总是给予允许的最大访问权限。Public继承对基类的protected部分放在派生类的protected部分；protected继承将基类的public、protected部分放在派生类的protected部分；private继承将基类的public、protected部分放在派生类的private部分。

保护继承的主要作用就是为了传承：如果派生类成为了被继承的类，如果它是私有继承而来的，那么从它的基类传下来的protected方法将不能继续传下去；使用保护继承即可。

## 多重继承

多重继承（multiple inheritance）和私有继承配合，实现has-a模式更丰富的拓展。但是一个显著的问题是，如果派生类多重继承的类有着同一个祖先类，那么意味着该派生类相当于第三代派生类，它的基类为第二代派生类。在第三代派生类中会有多个第二代派生类的组件，而第二代派生类都有祖先基类组件，那么在第三代中就会有多个祖先基类组件，既会浪费空间，又会产生二义性：第三代的祖先基类应该是哪个？如果不作处理，那么在构造的时候会构造多个祖先基类组件，使用祖先基类组件时，需要显式地指出使用的是哪个第二代的基类组件，即首先拿出第二代的组件，再对其进行向上转化。不仅繁琐，而且第二代的基类组件应该都一样才对。为了解决这一问题，虚基类登场了。

1. 虚基类

多重继承中，若基类是同宗的，且只需要一个祖先组件时，在继承说明时，各个基类前加virtual关键字，即可实现这一目的：

Class 派生类名：virtual protected 基类1,virtual protected 基类2{……}

这里的virtual和虚方法的virtual不是同样的原理，但是内部意义却是相似的。

1. 新的成员初始化列表

普通基类的继承中，对派生类基类组件的构造使用成员初始化列表显式调用基类构造函数即可，但是虚基类不同：需要先对祖先基类组件调用构造函数，再按照继承顺序分别调用虚基类的构造函数。如果基类非虚，那么成员初始化列表中是不可以调用上上代基类的构造函数的，只限定调用直接基类的构造函数，这也区别了虚基类和非虚基类。

1. 基类方法的调用

如果直接基类都重写了祖先基类的虚方法时，在派生类中调用需要明确指出是哪个直接基类，使用作用域解析符即可。

除了对第三代派生类的构造函数成员初始化列表进行修改，当我们使用直接基类的方法时，有可能会对祖先组件产生多次影响。解决这个问题有两种手段：第一种比较直接和简单，将祖先基类和虚基类的数据成员改为protected访问，这样第三代派生类即可在成员函数中将数据项拿来使用；第二种方法封装性更好，即使用protected修饰基类的辅助函数，在第三代派生类中可以继承得到，直接使用即可。

# C++代码重用

有了类和类继承的基础，我们能够大大提高代码的重用率。无论是is-a、has-a还是is-like-a模式，都可以利用类和类继承的技术加以实现。本部分代码重用的内容不会与类和类继承重合，介绍一些其他重用的技术：has-a模式的组合实现、类模板（容器常见）。

## 组合

组合（combination）是在类的设计中，将某个成员变量设置为另一个类的对象。这种技术也叫包含（containment）或层次化（layering）。可以看出，它是实现has-a模式最为直接的方法，而不是继承。在这里，只要将另一个类的对象视作普通的成员变量即可（前提是这个类能够正常使用）。组合技术能够让我们在类内部使用被包含类的外部接口，这就是最重要的作用。因为当前类相对于被包含类已经是外部。

组合类的构造函数中，成员初始化列表对各个变量名（对象）调用隐式构造函数，而不是像继承时用基类名构造基类组件；并且使用初始化列表的顺序要以类中声明的顺序一致，否则会提前调用成员变量的无参数构造函数，造成浪费甚至出错。

作为实现has-a的两种手段，组合与私有继承之间是有区别的。组合无法使用对象的私有和保护部分；而私有继承使得派生类可以在内部使用基类的protected部分（辅助函数）、基类对象甚至private部分（调用基类组件）。这是最重要的区别，如果我们想在新的类中使用已有类的protected部分或者private部分，那么不能使用组合，只能使用私有继承。相同之处是，新类都可以在内部使用旧类的public接口，而不能在外部使用新类时调用旧类接口。

## 二、类模板

模板是泛型编程的概念的体现。前面我们介绍过函数模板，它将数据类型作为参数，在通用的算法流程中实现。使用时要具体化，才在内存中得到新的函数。类模板可以用于实现ADT，一般为特殊构造的数据结构或容器：因为数据结构和容器只强调数据对象之间的逻辑关系以及需要特殊满足的要求，并不关注数据对象到底是什么类型。

1. 定义类模板

和函数模板一样，在声明和定义一个类模板前，要有关键字template。具体格式为：

Template<class T>

Class 类名{……（包含类型参数T的内容）}

关键字template说明了这是一个模板，并且用T来代表类型参数。在声明时，类模板和普通类的声明没什么区别，但是在定义的时候差异就出现了。类模板的成员函数必须在定义前加上模板前缀（template<class T>），并且函数名要用类名<T>::作限定（内联除外）。这样做的原因是：类模板的成员函数不能被单独编译，而是和实例化请求一起使用，如果不加限定，编译器并不知道这是哪个实例化对象的成员函数。除此之外，类模板的成员函数与类模板的声明都放在头文件中，而不是分开放置。

1. 使用类模板

同函数模板一样，类模板的使用要有实例化——编译器会按照模板生成一套真正存在的类声明和类定义，然后再用此套类实例化一个对象。不同的类型参数会实例化相互独立的类。这是我们就可以像使用普通类一样去使用从模板中实例化出来的类了。

但是和函数模板不同的是，函数模板在我们传入参数的时候，即使不传入类型参数<…>，编译器也可以通过实参的类型具体化一个函数，但是类模板必须要传入类型参数。

1. 类模板用作容器或数据结构时

容器或数据结构只是用来管理其中的元素，并不为元素分配内存或释放某一元素的内存（算法导论中的观点）。创建元素或释放元素的操作，是由调用函数完成的。比如，一个指针栈是用于存放指针的，它本身并不创建某类指针，也不销毁某个指针。将容器的功能界定清楚，能够帮助我们更好理解和使用他们。

1. 模板的多功能性

当我们实例化一个类模板后，就可以像常规类一样使用它：做基类、做成员变量，乃至用来实例化另一个模板（递归使用模板）。比如二维数组的形成，我们可以用指针数组加以实现，也可以使用[行][列]来直接声明一个二维数组，但容器模板使得我们可以：vector<vector<int>> Mat;来声明一个二维数组。这里Mat的每一个元素都是一个向量。此外，同函数模板一样，类模板也可以接受不止一个类型参数，也可以有非类型参数（可默认参数值），以及默认类型参数（但是不能全是默认类型参数）。

1. 类模板的具体化

具体化包括实例化和显式具体化，而实例化又包括隐式实例化和显式实例化。直接声明对象是隐式实例化，因为并没有专门在内存中创建一个新类，而是在用到对象时才创建；显式实例化就是在使用对象之前，主动将类在内存中实现，和函数模板的显式实例化类似：

Template class 类名<类型参数>；显示具体化是为某个特定类型参数创建具体的类，可能是由于某些操作不同于模板。实现方法与函数模板显式具体化很像：在通用模板声明完成后，template<> class 类名<加具体类型>{……};部分具体化是指对多类型参数的类模板指定部分类型参数。他之所以被称为部分具体化，就是因为并不是所有的类型参数都被指定了。

1. 模板的友元函数

友元函数的目的是访问类内部私有部分，他并不是成员函数，因此当友元函数需要一个类模板的对象作为参数时，在类模板内部声明原型时要加上<T>来说明这里需要的是一个具体化对象。直接这样做，被称为非模板友元函数：意为友元函数不会随类模板的具体化而产生定义，而是需要用户自己专门为友元函数提供定义。

为了解决这一问题，我们可以将友元函数也变为模板函数，随着类模板的具体化而具体化。如果我们把友元模板函数的声明和定义放在类模板的外部，而只是在类模板内部说明了友元关系，这被称作约束模板友元函数；如果放在类模板内部，称为非约束模板友元函数。两者的区别在于，放在外部的约束模板友元，参数表中需要写出类模板具体化形式；放在类模板内部的非约束模板友元函数，已经说明用到的泛型就是类模板的类，故不需要在参数表中写出类模板具体化形式。

# 友元、异常和RTTI

## 友元

类的设计中我们已经使用过友元函数：它的本质是外面的函数，但是特殊之处就是可以使用和它建立朋友关系的类的内部。类与类的关系中，我们会碰到uses-a模式，意为某一种类使用另一种类，我们可能会想到使用私有继承，但是私有继承隐含着派生类依赖基类的意思，且派生类对象并不能影响一个基类对象；如果我们想让类之间的关系不那么复杂，又能够让类与类之间保持独立性，那么我们可以借助友元类。此外，如果我们不想让友元类的全部作用域都能够访问此类内部，只需要对友元类的个别成员函数建立朋友关系，这被称为友元成员函数。

1. 友元类

如果我们想让class2使用class1的私有部分，只需要在class1内部建立朋友关系：

Friend class class2;虽然class2并没有出现过，但这样的声明是可以的：他告诉编译器，将会有一个名字为class2的类成为class1的友元类，class2可以使用class1的私有成员。Class1声明完毕后，我们便可以声明class2的内容了。这时按照先后顺序，编译器已经知道了class1的内部有些什么，所以自然而然class2在声明的内部可以使用class1。

1. 友元成员函数

为了加强访问限制，我们只在类中的某些成员函数中使用到另一个类的私有部分，这时我们可以把这些成员函数建立与另一个类的朋友关系，以实现我们的目的。但是，要特别注意类之间声明与定义的顺序。以class2中的成员函数要使用class1的私有部分为例来理解顺序的重要性。因为class2的内部要使用class1，这要求编译器首先知道class1是什么东西，所以我们选择先声明class1；但是，在class1声明时，编译器要知道class1与class2中的哪些成员函数建立朋友关系，为此我们需要在class1声明之前定义class2.看似矛盾无解的问题，实际上可以通过前向声明解决。如果我们在定义class2前，先写一句：

Class class1;那么编译器就知道，将会有一个名为class1的类出现。之后我们紧接声明class2，其中有几个成员函数用到了class1——参数表中有class1的形参。但此时无法直接在class2类内部内联成员函数定义：一是友元关系还未建立，二是编译器不知道class1到底有什么东西。最后我们紧接着声明class1的内容，并建立与class2中个别成员函数的朋友关系，此时由于是在class2的外部，所以要使用作用域解析符：

Friend class2::function(……);

最后再定义class2中的成员函数，因为此时已经知道了class1的内容，可以在友元成员函数中使用了。

1. 其他友元关系

友元关系能够互相建立：如class1可以使用class2的私有部分，class2也可以使用class1的私有部分，此时可以在两者内部都建立朋友关系。但是要注意的是声明和定义的顺序。（不过一般定义在源文件中，声明在头文件中）此外，如果某个函数要使用两个类的私有部分，那么一种是可以用友元函数的方法，在两个类中都与该函数建立朋友关系；另一种是将该函数作为一种类的成员函数，从而使用友元成员函数达到目的。

## 二、异常

如果程序运行过程中遇到一些特殊或意外情况，很有可能引发程序崩溃。为了避免程序崩溃并向用户反馈错误，我们使用异常机制。异常机制使用抛出-捕获形式，我们既可以设计抛出，也可以设置捕获。此外，C++库中有<exception>头文件，其中有多个定义好的异常类，我们可以以其为基类创建我们自己的异常类。

1. 设计抛出异常

我们要对可能出现的错误做出估计，并使用分支语句来判断，并返回错误信息给用户。在此要介绍throw机制。当异常出现，在他还未发生时，我们若是对该种情况做出了判断：比如我们做除法，已知了除数为0，但是还没有做时，我们就要throw异常，否则就太晚了。Throw()圆括号中间放置的是一个说明情况的对象，可以是我们自定义的类，也可以是异常类，总之，要在这个对象中把即将出现的问题说明。

1. 捕获异常

使用了throw，我们就要捕获它。Try{……}catch(exception handler异常处理程序&){……}就是用于捕获抛出的异常的语句块。在try部分，我们将可能出现问题的代码放入；catch接受的异常类型要和throw的抛出对象类型一致（至少能转换，因为这里形参是引用），如果在try内部真的有对象被抛出，那么catch就将其捕获，并在随后的语句块中进行处理，甚至继续抛出。因为对象中包含了错误信息，所以我们确实能够处理它。Try可以与多个catch()组合使用，因为handler可能不同，我们可以根据类型匹配，做不同处理。

Throw到catch在计算机内部，符合栈解退机制。我们知道函数调用在内存中使用的是栈模式（这就是为什么递归函数被称为压栈），后被调用的函数其中的自动变量压在前面被调用的自动变量上。栈解退机制就是，一旦抛出的异常被距离最近、并且handler类型符合的catch捕获到，那么从throw到这个catch之间的所有自动变量都被释放（调用析构函数）。这不同于return只释放将要离开的代码块中的自动变量。

Catch的handler为引用的目的，是为了将派生类也用基类引用捕获，而不是为了增加传递效率，因为在throw时会创建临时变量，但是在catch处会除去派生类的特性，只使用虚方法的基类版本；如果我们确实不知道会抛出什么类型的handler，可以使用…来代替其位置。

1. 异常类

虽然我们可以对可能出现的错误做出预测，然后用自己设计的类来抛出异常，但是这样做效果不一定好：有很多我们难以预料到的错误，并且会给用户造成困扰——到底使用什么handler来捕获异常对象呢。C++提供了异常基类——exception类，放在头文件<exception>中，我们可以由他派生出自己的异常类，用户也可以使用他来捕获；此外C++库也提供了exception的一些派生类，专门处理指定类型的错误，我们可以使用他们来对自己代码中可能出现的问题进行精确地捕获。

Exception类有一个虚方法what()，返回const char\*，下面要说明几个他的派生类。

logic\_error类是从stdexcept类派生来的，构造时需要一个const char\*成员。它主要反映的是数学运算中的逻辑错误：除数为零、函数定义域出错、索引超限等问题。

runtime\_error类也是从stdexcept中派生而来，同样需要一个const char\*成员。它反映的主要是运行时可能出现的问题：数值overflow、underflow等。数值溢出还是比较常见但又隐藏很深的问题，编程时要多加注意。

以上两者都继承自stdexcept类，我们可以在创建这个对象时将问题进行传入，之后调用what()就会把同样的字符串吐出来，以此来得到反馈。

bad\_alloc类与new。这个类在头文件<new>中。我们知道使用new分配内存，但是如果遇到可用内存不足时，程序会抛出bad\_alloc异常，我们只要用catch(exception&)去捕获就好。

1. 异常与类

在类的设计中，我们应该考虑到用户对类的非常规操作，并使用抛出异常的机制来使得用户可以使用catch进行捕获。如果只使用exception来捕获异常，由于派生类性质被剥夺，并不能传达出很有效的反馈。为此我们可以为自己定义的类设计异常类，或者在类中添加异常成员，并在可能出错的成员函数中将其抛出，以此提高我们自定义类的完善度。

如果我们使用只被类所用的异常类，为避免重名混淆，我们将异常类的作用域限定在使用它的类内部，这叫做嵌套（nest）类。嵌套类可以声明在类的public部分（如果我们想让外界使用的话）。其定义要加上类名的作用域解析符::。回到嵌套异常类，我们当然想用catch最佳匹配异常类型，所以要将其放在public部分。并且依照错误可能出现的类型，我们的异常类最好公有继承已有的异常类：logic\_error或runtime\_error。类的设计中我们关注异常的抛出，类的使用中我们关注异常的捕获。注意catch使用基类引用依然只能使用虚方法的基类版本。

1. 未捕获异常和意外异常

如果程序抛出了异常，却没有try-catch语句块将它捕获，程序会调用terminate()函数，其中有abort()用以终止程序。好在我们可以使用set\_terminate(…)来使terminate()调用我们自己定义的结束函数。这里就用到了函数指针，因为set\_terminate接受的参数就是函数指针。相似地，如果程序抛出的异常，对于我们的catch都不匹配，这将调用unexpected()函数，同样在内部有abort()来终止程序，我们可以通过set\_unexpected(…)来使得u- nexpected()函数调用我们指定的函数，同样用到函数指针。

## 三、RTTI

RTTI（runtime type identification）是在程序运行过程中对类型的识别，它只适用于存在虚方法的类层次。它的用途在于：基类指针可以指向派生类对象，但是使用它不能调用派生类特有的外部接口。RTTI帮助我们得到该对象的确切类型。RTTI三个手段相互配合，使得我们实现判断派生类具体信息的愿望。

1. dynamic\_cast<>()运算符

动态转换将基类指针或者对象，转换为派生类指针或对象。如果不能转换，则返回空指针；如果是引用不能转换，则抛出bad\_cast异常。我们能够通过其返回值或异常来判断转换是否成功。如果成功就可以调用派生类的特有接口了。

1. typeid和type\_info类

typeid是一种运算符，类似sizeof，接受类名或者变量表达式，返回的是type\_info对象的引用。Type\_info类重载了==和!=，于是我们判断一个未知的类型是否是某个确定的类型。相比dynamic\_cast<>，typeid更加严格：第三代派生和第二代派生的type\_info不能相等。但是dynamic\_cast<>可以将第三代派生转换为第二代派生。

1. 其他类型转换符

虽然不是RTTI的内容，但是说到了dynamic\_cast<>，不妨用其他的转换符与之对比，加深理解。Dynamic\_cast用于运行时的类型转化，且特别用于基类指针向派生类指针（向下）的转换；const\_cast<>只用于去除const或volatile标签，将常量转化为变量用于一时的修改，数据类型不发生转换；static\_cast<>用于相互之间可隐式转换类型的转换，如派生类可以隐式转换为基类（向上转换），由转换符的显式说明，我们能够更加清晰地知道转换在哪里发生。

要清楚的是，类型转换不会对原数据进行变动，只是将一种类型的量拷贝给另一类型。

# 标准模板库

前面我们知道了如何设计自己的类：从声明到定义、继承、友元、自己设计异常来使得类更加完善，但是同时我们也要学会使用别人的类：C++商业库或自带库。在这一部分我会强调运用部分，但是所涉及到的原理也会提到。主要为C++最为常用的数据类型string、智能指针模板、标准模板库（standard template library，STL）的使用。

## string类

在前面我们已经用过不少string的对象，并且还用它和C风格字符串做了许多对比。但在这里要详细说明的是，string向容器类的靠拢，以及更多的string类的接口使用。String类被包含在<string>头文件中，并在std名称空间下，所以使用时可以使用作用域解析符，也可以使用using声明或using编译。

1. string的多种构造

比较c风格字符串（NBTS）和string类时，我们知道可以使用字符串常量和NBTS构造string，但是除此之外，还有各种各样满足不同需求的string构造方法。需要注意的是，string有自己最多能容纳的字符个数，使用string::npos表示，在string类的许多接口中，到要用到这一常量。

用同一字符构造长度为n的字符串：string(size\_type n,char c);此方法和容器类相通；

从NBTS中截取部分形成字符串：string(const char\* s,size\_type n);n可以大于s长度

使用迭代器构造字符串：string(iter be,iter en);与容器类相通，并且可以在数组中使用

使用string对象的一部分形成新的字符串：string(const string& s,size\_type pos,size\_type n=npos);pos表示截取的开始之处，n表示截取子串的长度

使用初始化列表构造字符串：string({……})

如此多的构造函数，我们不需要背会，但需要知道有哪些形式，这样才能在需要的时候使用最方便有效的那个构造函数。

1. string的输入

这一点在第一部分输入与输出就讲过，这里还需要重新强调一下。无论是标准输入还是文件输入，我们都要用到流机制。抽取运算符对字符串的输入，截止到空白（space、tab、enter），并且不会将空白从缓冲区中丢弃，这一点对其他类型也是一样的。最为常用的对字符串的输入为std名称空间下的getline(…)函数，他接受三个输入参数：输入流、string对象、分界符（默认为\n）。getline有时会使得流关闭：当遇到eof或需要读入的字符数量大于npos的时候，流的.fail()返回true。但是我们可以进一步判断到底是因为eof引起，还是由于字符数量超限引起，如果是第二种原因，且我们想继续读入，将string对象清空，流的状态重新设置即可。分界符也很有用，只接受一个字符，流在遇到之后暂停输入，并将分界符从缓冲区中丢弃。使用这一点，我们就可以将输入格式化。如果分界符之间存在层次关系，也可以利用sstringstream来一次次分割，最终实现格式化字符串。

1. string的其他常用方法

string的比较通过重载运算符实现，此外还有字符串连接+，返回两字符串合并的结果。这里+的右操作数可以是NBTS，隐式调用转换构造函数。

为了与STL容器兼容，string提供.size()方法，返回容器中字符个数，不包括\0.

.find()用于在对象中找出目标串，返回一个位置表示第一个字符在对象中的下标，这一点和容器的find()返回目标所在的迭代器很接近，甚至和算法分析中在某一数据结构中按照key寻找对象的search方法类似。.find()有四个重载，分别接受string对象，NBTS，单字符，以及限定查找范围的参数pos和n；find是从左到右查找，找到第一个就返回，rfind从右至左查找，找到最后一个返回。此外还有find\_first\_of()，作用是在string对象中找到第一个和目标字符串中存在的字符相同的字符位置：如在”LEEBEN”的对象调用find\_first\_of(“ABDEL”)，将返回1，因为对象中的E最先与“ABDEL”匹配。Find\_last\_of()作用类似，只不过是找到对象中与目标字符串相同的最后出现的字符位置。

## 二、智能指针模板

使用指针是我们经常配合堆内存的分配，但是稍有不慎就会导致内存泄露（如在抛出异常后一系列过程中的堆内存都无法释放）。智能指针作为一个模板类，帮助我们自动释放从堆中分配来的内存，只要智能指针过期，那么它所指向的堆内存也就被自动释放，不需要我们手动delete。它被包含在头文件<memory>中，并在std名称空间下。旧版的C++提供一种auto\_ptr模板来实现智能指针，使用方法为：

auto\_ptr<Type> p(new Type);//这样我们就创建了一个指向Type对象的智能指针p

auto\_ptr<Type>(regular\_ptr);//将一个常规指针显式转化为智能指针

智能指针的使用和普通指针一样，都是用\*来解除引用，->来访问接口。

当对象被多个智能指针指向的时候，可能导致内存的重复释放，这是非常危险的，为了解决这个问题，auto\_ptr建立了所有权转移的机制，就是把对象的所有权全部转移给最新赋值的智能指针对象，但是这将导致之前的智能指针对象指向错误的数据，为了解决auto\_ptr的问题，我们可以选择使用unique\_ptr<>或shared\_ptr<>。

1. unique\_ptr<Type>

顾名思义，unique\_ptr<>在一般情况下是不允许所指的对象控制权发生转移的，即不允许unique\_ptr之间相互赋值。但有一种情况例外，即函数返回一个临时unique\_ptr对象时，可以将其赋值给调用函数的unique\_ptr对象，这是因为，该临时变量随后会被马上删除，并不会出现混乱使用的情况。

此外unique\_ptr优于auto\_ptr的一点是，可以为其分配数组：

unique\_ptr<double[]> p(new double(20));//p指向20个元素的double数组

1. shared\_ptr<Type>

shared\_ptr解决所有权问题的方法是：建立引用次数，这与浅层拷贝有些相像。Shared\_ptr之间在互相赋值时候会建立引用次数，只有当引用次数为零时，所指对象才会被真正释放；在多个shared\_ptr都指向同一对象时，他们都是有效的。

选用智能指针时，如果不需要共享，或者要数组时，最好选择unique\_ptr，事实上在函数传参时，可能会有指针共享的问题，此时如果是引用实参，那么unique\_ptr也可以使用。

## 三、STL

这是STL和我的第一次亲密接触，开始的时候有些难以理解，但到后面我越来越感受到它的方便，并在实际编程中愿意使用它。STL体现了泛型编程的思想：关注数据结构和算法的实现，而并不关注数据类型到底是什么。它的容器、迭代器、算法为我们操纵数据提供了很简便的手段：只需要调用就可以。但是由于各种容器基于不同的底层数据结构，通用的算法并不一定很好地适用不同的容器，所以有些容器自带适应本身数据结构的算法。我们在使用时，考虑到效率和空间，就要留意这些，尤其是当数据量巨大的时候，选用合适的容器和算法就很重要了。我们需要对容器有一个基础的理解：容器是同质的，意味着它的元素都是一样的数据类型：栈、队列、矢量、链表、优先队列、集合、地图等等，通过数据结构的学习，我们知道这些容器具有各自的特点，为了实现这些特点，他们的底层确实不一样。下面再来理解STL中的其他概念。

1. 迭代器

如果说容器使得数据结构独立于数据类型（因为我们可以使用任意类型来放入容器），那么迭代器就使得算法独立于容器。这意味着，算法是通过操作迭代器来实现的。理解迭代器，它是一种广义指针，可以指向容器中的元素，函数对象接受迭代器，从而访问或操作容器内的元素。每种容器都有自己的迭代器，并且迭代器各有特点。按照迭代器能够对元素进行的操作，我们可以建立这样的层次结构：

输入迭代器（只提供访问元素权限，单向）、输出迭代器（只提供写入元素权限，单向）

正向迭代器（可以对元素进行读和写，但只能递增）

双向迭代器（可以对元素进行读写，可以递增，也可以递减，但是只能逐步移动）

随机访问迭代器（可以对元素进行读、写，并且可以在容器中跳着访问）

输入迭代器的例子：普通数组的地址在算法中的表现；

输出迭代器的例子：ostream\_iterator；

不同的算法对迭代器的要求是不同的，因此这就导致了有些容器不能使用某些算法。有些迭代器比较特殊，如反向迭代器、插入迭代器，将他们与对象绑定，即可以将通用算法用在对象上。

1. 容器

容器之间有共同的特征，比如他们有自己的迭代器类型（如果我们想使用某种类型的迭代器，只需要用作用域解析符或者使用auto进行自动推断）、有begin()和end()迭代器、.size()方法返回容器内的元素个数、.swap()用于容器元素的交换等等。

对于更多更有用的容器特征，如erase用于清除容器中指定的元素、find在容器中找到指定元素位置、sort对元素进行排序、insert将元素插入、clear清空容器等等。下面介绍一些常用容器的使用。

vector<>是最常见的容器，它的迭代器种类为随机访问迭代器，所以vector支持随机访问。并且vector是一个动态数组，可以push\_back\push\_front，也可以pop。如果没有特殊要求，vector是我们首选的容器。

List<>是双向链表，它的迭代器不支持随机访问，因此没有下标操作。但是他有merge合并操作，只需要固定的时间复杂度（前提是链表已经排序），unique会将排好序的链表中相邻的相同元素除去只保留一个，splice将做参数的链表全数放入对象链表。

Set<>的底层是排序树，并且迭代器类型为输入迭代器。它的元素在进入时就有了顺序，并且不容纳相同元素（自动unique）。有算法可用于set：计算并集、差集、交集。

Map<,>的底层是红黑树，它的每一个元素都是一个pair类型，键-值对应，前一个是输入迭代器，后一个可以修改。在map中，键必须唯一，而值可以不唯一。Pair的创建有三种方法：make\_pair、显式调用构造函数以及隐式调用构造函数的列表初始化。有趣的是，可以直接用键来做下标，使用其对应的值。

multimap<,>允许键也相同，但是这就使得它的按键查找的结果与键唯一的map有些不一样：equal\_range(…)查找相同键值的元素，返回的是一个pair类型，first指向最近的那个元素，second指向最远的那个元素（即lower\_bound\upper\_bound）。

1. 函数对象

可以把函数对象理解为函数名、函数指针以及重载了()运算符的对象。函数对象可以在算法中使用，来提供用户想要的处理方式。函数对象有不接受参数的、一元的、二元的，根据算法对函数对象的要求而异：transform要求一元、sort要求二元谓词、for\_each要求一元、带\_if的要求一元谓词。函数对象我们可以自己写，也可以利用预定义好的（存储在头文件<functional>中）。一般地算术运算或关系运算都有等价的函数对象：plus、minus、multiplies、divides、modulus、equal\_to、not\_equal\_to、greater、less等等。同时，我们也可以使用适配器来将二元函数转换为一元函数，从而用于接受一元函数的算法。这里使用bind1st或bind2nd，给入参数和函数对象，直接生成一元函数。

1. 算法

算法是处理容器中元素的非成员函数，一般以迭代器和函数对象作为参数，返回类型依情况而定。算法有四种类型：非修改式序列操作（find、for\_each）、修改式序列操作（copy、transform、random\_shuffle）、排序和相关操作（sort、set\_union等）、通用数字运算。前三种都在头文件<algorithm>中，第四种在<numeric>中。根据算法完成后的结果，可分为就地（in place）算法和复制算法，前者使得原始数据位置不变，如sort，后者将结果发往了另一个位置，如copy，transform的两种形式，既可以是就地，也可以是复制。有些算法两个版本都有，如replace——replace\_copy；有些算法在谓词下工作，如remove\_if\replace\_if等算法。还有一个有趣的算法，是将容器内的元素进行各种顺序的排列：next\_permutaion(),他是一个就地算法，每次调用完都使容器内元素顺序变化，直到所有顺序都出现过，它返回false。

算法与容器类成员函数的作用有时是类似的，但是成员函数更加适用于所属容器。比如容器的remove或erase，会将指定元素删除后，缩减容器大小，节约内存，但是通用算法的remove或erase，只返回删除后指向超尾的迭代器，容器内部大小并没有变化；unique也是一样的道理。有时我们会选择成员函数，因为这样更适合，而有时为了通用性，我们选择算法。

# 写在最后

完成此篇报告的时间是2018年11月17日，距离阅读primer这本书大致一个多月，现在我正在自学《算法导论》，其中的很多内容都与编程有关，我在学习算法的时候，偶尔会使用C++来实现其中一些比较有趣和常用的例子。两者结合起来，留下了比较深的印象。

C++primer的自学让我培养了面向对象编程的思想：从接口出发设计一个类，之后将此类实例化得到对象，通过对象或者类中的静态方法实现操作。或者作为使用库的用户，先了解类中的接口然后再去使用。此外还有泛型编程的思想：容器、迭代器和算法的配合使用，来实现项目的一些操作。之后的实践，我相信还有更多锻炼的机会，编程是一个基本功，C++是一个支持在程序运行时进行决策的语言，这种特点也会逐渐成为一个编程习惯。