目录

[PART1:C++基础: 2](#_Toc462238522)

[Ch1.开始 2](#_Toc462238523)

[Ch2.变量和基本类型 3](#_Toc462238524)

[2.1算术类型： 3](#_Toc462238525)

[ch3.字符串、向量和数组 8](#_Toc462238526)

[3.1 string 8](#_Toc462238527)

[3.2 vector: 9](#_Toc462238528)

[3.3 迭代器 10](#_Toc462238529)

[ch4.表达式 11](#_Toc462238530)

[ch5.语句 12](#_Toc462238531)

[ch6.函数 12](#_Toc462238532)

[ch7.类 18](#_Toc462238533)

[7.8 构造函数： 19](#_Toc462238534)

[PART2:C++标准库: 25](#_Toc462238535)

[Ch8.IO库 25](#_Toc462238536)

[8.1 I/O类 25](#_Toc462238537)

[8.2 文件IO 26](#_Toc462238538)

[8.3 string流 27](#_Toc462238539)

[Ch9.顺序容器 27](#_Toc462238540)

[Ch10.泛型算法 31](#_Toc462238541)

[Ch11.关联容器 31](#_Toc462238542)

[Ch12.动态内存 31](#_Toc462238543)

[PART3:类设计者的工具 32](#_Toc462238544)

[Ch13.拷贝控制 32](#_Toc462238545)

[Ch14.重载运算与类型转换 32](#_Toc462238546)

[Ch15.面向对象程序设计 32](#_Toc462238547)

[Ch16.模板与泛型编程 32](#_Toc462238548)

[PART4:高级主题： 32](#_Toc462238549)

学习计划，partI~partIII，时间2月

# PART1:C++基础:

## Ch1.开始

1.数据类型，不仅定义了数据元素的内容，还定义了这类数据上可以进行的运算；

2.访问main返回值的方法依赖于系统，在UNIX和Windows中分别如下方法：

1）UNIX中：$ echo $?

2) WINZ中：$ echo %ERRORLEVEL%

3.打开编译器常用选项，GNU中使用-Wall, 微软编译器中用/W4

g++ hello.cpp -Wall --std=c++0x

或者-std=c++11

4.C++语言并未定义任何输入输出(IO)语句，而是包含了一个全面的标准库来提供I/O机制；

标准库定义了四个IO对象，cin(标准输入), cout（标准输出）, cerr（输出警告错误信息，默认不会被缓冲）, clog（用来输出一般性信息，默认被缓冲）;

5.”<<”是一个双目运算符，"<<"接受两个运算对象，左侧必须是一个ostream对象，将右侧运算对象写到给定的ostream中，并且该运算符返回其左侧运算对象；这样"<<"可以多个连用，比如cout<<"hello"<<endl;

6.C++中操纵符endl的效果是结束当前行，并将与设备关联的缓冲区内容刷新到设备中；

调试程序时添加的打印语句，应该保证“一直刷新流”。否则程序崩溃，输出可能还留在缓冲区中，从而导致关于程序崩溃为止错误判断.

7.注释通常用于概述算法，确定变量的用途，或者解释晦涩难懂的代码段；

错误的注释比完全没有注释更糟糕，因为它会误导读者。因此，修改代码时，不要忘记同时更新注释。

8.当使用istream对象作为条件时，其效果是检测流的状态；如果流是有效的，即流未遇到错误，那么检测成功，当遇到文件结束符，或遇到一个无效输入时（比如读入一个非整数给整形对象），istream对象状态会变为无效。

9.win系统中，输入文件结束符是Ctrl+Z，然后按Enter；Unix系统中，输入Ctrl+D；

10.为了使用一个类，我们不必关心它是如何实现的，只需知道类对象可以执行什么操作。一般来说，类的作者决定了类类型对象上可以使用的所有操作。

## Ch2.变量和基本类型

### 2.1算术类型：

1.内置类型

1）C++语言规定一个int至少和一个short一样大，一个long至少和一个int一样大，一个long long至少和一个long一样大，数据类型long long是在C++11中新定义的。

2）浮点型，可以表示单精度、双精度、扩展精度值；一般来说类型float和double分别有7和16个有效位；类型long double常被用于有特殊浮点需求的硬件。

3）出去布尔和扩展的字符型外，其他整形都可以被分为带符号(signed)和无符号的(unsigned);

4)字符型分为三种：char, signed char 和 unsigned char.

char和signed char并不一样，计算机表现形式只有两种：有符号，无符号的，类型char实际上会表现为两种之一，具体是哪种由编译器定。

2.如何选择数据类型：和C一样，C++的设计准则之一是尽可能的接近硬件：

1）当明确知晓数值不可能为负时，选用无符号类型；

2）使用int执行正数运算，实际应用中，short常显得太小，而long一般和int一样尺寸，当数值超过int时，用long long.

3)在算术运算中不要使用char或bool，因为类型char在一些机器上有符号，而在另一些机器中无符号，如果需要使用一个不大的正数，那么明确指定是signed char 和 unsigned char;

4)执行浮点运算，使用double。因为float通常精度不够而双精度浮点和单精度浮点数的计算代价相差无几；事实上某些机器运算double比float还快。long double提供的精度一般情况下没必要，而且它带来的运行时消耗不可忽视；

3.如果赋值给一个无符号类型超出它范围的值时，结果是初始值对无符号类型表示数值总数取模所得的余数。比如8bit的unsigned char表示0～255，把-1赋给unsigned char所得结果是255；当赋给带符号类型一个超出范围值时，结果是未定义的(可能停止工作，崩溃，或产生垃圾数据)。

4.含有无符号类型的表达式：

1）当一个有符号数和一个无符号数运算时，有符号数会转换成无符号数；比如

unsigned a = 10;

int i = -42;

std::cout << a + i <<std::endl;

输出结果：4294967264

2）当从无符号数中减去一个值时，不管这个值是不是无符号数，结果都不能是一个负值；

比如：

unsigned char i;

for (i = 10; i >= 0; i--) ;//死循环

5.默认情况下，十进制字面值是带符号的，八进制和十六进制可能是带符号，也可能是无符号的。

十进制的类型是int、long和long long中尺寸最小的，但前提是能容下当前值。

八进制和十六进制字面值类型是能容纳其数值的int,unsigned int , long, sunsigned long, long long和unsigned long long中尺寸最小的。

如果字面值超过最大数据类型范围，将出错。

6.指定字面值类型：

1）字符和字符串字面值 前缀：

u Unicode 16字符 char16\_t

U Unicode 32字符 char32\_t

L 宽字符 wchar\_t

u8 UTF-8(仅用字符串字面常量) char

2）整型字面值 后缀：

u or U unsigned

l or L long

ll or LL long long

3）浮点型字面值

f or F float

l or L long double

4)true和false是布尔类型字面值

5）nullptr是指针字面值

7.用于初始化变量的值可以是任意复杂的表达式：

double price = 109.99, discount = price \* 0.16;

double salePrice = applyDiscount(price, discount);

8.C++初始化方式：

int a = 0;

int a = {0};

int a{0}; //列表初始化，在C++11中普遍运用

int a(0);

注意：当使用列表初始化且初始值存在丢失信息的风险，则编译器报错，比如：

long double ld = 3.1415926536;

int a{ld},b = {ld};//报错，转换未执行

int c(ld),d = ld;//正确，转换执行，且确实丢失部分值。

9.任何包含了显示初始化的声明即成为定义。给extern关键字标记的变量赋一个初始值，但这样就抵消了extern的作用。

比如extern double pi = 3.14159;//此处pi是定义，而非声明了；

10.标识符规范：

1）C++为标准库保留了一些名字：

用户自定义的标志符中不能连续出现两个下划线，也不能以下划线紧连大写字母开头，定义在函数体外的标识符不能以下划线开头；

2）标识符要能体现实际含义；

3）变量名一般用小写；

4）用户自定义类名一般用大写开头；

5）如果标识符由多个单词组成，则单词间应有明显区分；

11.当第一次使用变量时再定义它：

一是更容易找到变量的定义，而是更利于给它一个比较合理的初始值；

12.在嵌套作用域中，变量名相同时，比如

int reused=42;

int main()

{

std::cout << reused<<std::endl; //输出42

int reused = 0;

std::cout << reused<<std::endl; //输出0

std::cout << ::reused<<std::endl;//显示地访问全局变量，输出42

}

注意：如果函数有可能用到某全局变量，则不宜再定义一个同名的局部变量。

13.引用：就是一个对象的别名

1)引用必须被初始化；

2)因为引用本身不是一个对象，所以不能定义引用的引用；

3)引用的类型必须和与之绑定的对象严格匹配（2.4.1和15.2.3两种情形除外）；

4)引用只能绑定在对象上，而不能与字面值或某个表达式的计算结果绑定；

变量声明：一条声明语句由一个基本数据类型和紧随其后的一个声明符列表组成；每个声明符命名了一个变量并指定该变量为与基本数据类型相关的某种类型；

14：指针：也是一种间接访问的复合类型

1）它与引用不同点：

指针本身就是一个对象，允许对指针赋值和拷贝，而且在生命周期内它可以先后指向不同的对象；

指针无须在定义时赋初值；

2）指针和引用一样，要和其指向的对象相匹配；

3）引用本身不是对象，所以指针不可以指向引用；

4）空指针赋值方式：

int \*p1 = nullptr; //C++11新标准，推荐使用

int \*p2 = 0;

int \*p3 = NULL; //NULL为预处理器变量，等同于0

15.默认情况下const被设定为文件内有效，但如果确实有变量需要在多文件之间共享，一个较好的方法是：

对于const变量不管是声明还是定义都添加extern关键字，这样只需定义一次就够了：

file\_1.cpp

extern const int bufSize = fcn();

file\_1.h

extern const int bufSize;

16.引用的类型必须和绑定的对象相匹配，但又两个例外，一个是：

初始化常量引用时，可以允许任何表达式作为初始值，只要该表达式的结果能转换成引用的类型即可。

int i = 42;

const int &r1 = i; //正确

const int &r2 = 42; //正确

const int &r3 = r1\*2;//正确：r3是一个常量引用

int &r4 = r1\*2; //错误，绑定类型不匹配

double dval = 3.14;

const int &r5 = dval;//正确，编译器构建一个临时变量来绑定r5

17.指针必须与所指对象类型匹配，但有两个例外，一个是：

一个指向常量的指针，可以指向一个非常量对象。

const double \*cptr = nullptr ;

double dval = 3.14;

cptr = &dval;

顶层const表示指针本身是个常量，底层const表示指针所指的对象是一个常量；

18.constxpr变量：让编译器验证变量的值是否是一个常量表达式。

1）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constxpr类型。

2）函数体内定义的变量一般来说并非存放在固定地址中，因此constxpr指针不能指向这样的变量；相反，定义域所有函数体之外的对象其地址固定不变，能用来初始化constxpr指针。ps:函数有效范围超出函数本身的变量，constexpr可以绑定或指向这样的变量。

3）如果constexpr声明中定义了一个指针，限定符constexpr仅对指针有效，与指针所指对象无关；

const int \*p = nullptr;//p是一个指向常量的指针

constexpr int \*q = nullptr;//q是一个常量指针

19.类型别名：

1）typedef double wages; //wages是double的同义词

typedef wages base, \*p;//base 是double的同义词，p是double \*的同义词;

wages hourly, weekly;//等价于double hourly,weekly;

using SI = Sales\_item;

SI item; //等价于Sales\_item item；

2）typedef char \* pstring;

const pstring cstr=0;//cstr是一个指向char的常量指针；

const pstring \*ps; //ps是一个指针，它的对象是：指向char的常量指针；

typedef using都不是字面的简单替换，而是一个整体的别名

const char \*cstr;//表示cstr是一个指针，指向一个const char类型，是对const pstring cstr的错误理解；

20.auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。

auto item = val1 + val2; //item初始化为val1 和val2相加的结果；

一条声明语句中，只能有一个基本数据类型；

auto i = 0, \*p = &i; //正确，i是整数,p是整型指针

auto sz = 0, pi = 3.14;//错误，sz和pi的类型不一致

//auto会忽略掉顶层的const，底层const会保留

const int ci = i, &cr = ci;

auto b = ci;//b是一个整数，顶层const忽略

auto c = cr;//c是一个整数

auto d = &i;//d是一个整型指针

auto e = &ci;//e是一个指向整型常量的指针（对常量对象取地址是一种底层const，底层const保留）

如果希望推断出的auto类型是一个顶层const，需要明确指出：

Const auto f = ci //ci推演类型是int, f是const int;

const int ci = i, &cr = ci;

auto &g=ci; //g是一个整型常量引用，绑定到ci

auto &h = 42; //错误，不能为非常量引用绑定字面值

const auto &j = 2; //正确

auto &m = ci, \*p =&ci; //m是对整形常量的引用，p是指向整形常量的指针；

设置一个类型为auto的引用时，初始值中的顶层常量属性保留；

21.decltype和引用

decltype处理顶层const和引用的方式与auto不同：

const int ci = 0, &cj = ci;

decltype(ci) x = 0; //x的类型是const int

decltype(cj) y =x; //y的类型是const int &, y绑定到x

decltype(cj) z; //错误，z是一个引用，必须初始化

引用从来都作为其所指对象的同义词出现，只有用在decltype处是一个例外。

========================================================

int i = 42,\*p = &i, &r = i;

decltype(r+0) b; //正确，b是int

decltype(\*p) c;//错误，c是 int& 引用，必须初始化

1）r是一个引用，所以decltype(r)必然是引用，但是r+0把r当作表达式一部分，结果是个具体的值，而非引用；

2）如果表达式内容是解引用操作，则decltype将得到引用类型，decltype(\*p)得到的是int &

；

3) decltype((variable))的结果永远是引用，而decltype(variable)结果只有当variable本身是引用时才是引用；

22.头文件保护

#ifndef SALES\_DATA\_H

#define SALES\_DATA\_H

...

#endif

## ch3.字符串、向量和数组

### 3.1 string

1.标准库的命名空间using声明:

1)每个名字都需要独立的using声明；

2）头文件不应该包含using声明（防止不必要的命名冲突）；

2.使用等号初始化一个变量，实际上执行的是拷贝初始化；

不使用等号，则实行的是直接初始化；

string s1 = "hello"; //拷贝初始化

string s1("heool"); // 直接初始化

3.getline:获取一行字符串，保留空白符，读入换行符（从cin读入，但不会存入string）

getline(cin, string);//从cin获得一行字符串，存入string，但不会存入换行符

4.如果一条表达式中有了size()函数，就不要再使用int ,这样可以避免int 和unsigned可能带来的问题；

int n;

s.size() < n; //n转化为unsigned，该判断结果几乎肯定是true;

string的size()函数返回一个string::size\_type，是无符号型；

ps:字符或字符串字面值与string是不同的类型；

5.在cname头文件中定义的名字从属于命名空间std,而定义在名为.h的头文件则不然；

一般来说，C++程序应该使用cname的头文件，标准库中的名字总能在命名空间std中找到。

6.范围for

string str("hello,world!");

for (auto c: str)

touper(c);

也可以使用下标迭代：下标运算符([])接收的输入参数是string::size\_type类型值，返回值是该位置上字符的引用。

string s = "hello";

for (decltype(s.size()) index = 0; index != s.size();index++)

s[index] = touper(s[index]);

7.string类型，可以用下标随机访问；

string hexdigits ="01213456789abcdef”；

hexdigits[10] = 'a';

8.C++标准并不要求标准库检测下标是否合法，一旦使用了一个超出范围的下标，就会产生不可预知的结果。

### 3.2 vector:

模板本身不是类或函数，相反可以将模板看作，为编译器生成类或函数编写的一份说明。

编译器根据模板创建类或函数的过程称为，实例化。

1.列表初始化vector对象：

1）使用拷贝初始化时，只能提供一个初始化值；

2）如果提供的是一个类内初始值，则只能使用拷贝初始化或使用花括号的形式初始化；

3）如果提供的是初始元素值的列表，则只能把初始值都放在花括号里进行列表初始化，不能用圆括号；

2.如果初始化使用了花括号的形式，但提供的值又不能用来初始化，就要考虑用这样的值来构造vector对象。比如

vector<string> v1{8};//v1有8个默认初始化的元素

vector<string> v2{8，“hi”};//v1有8个值为"hi"的元素

3.vector对象，直接初始化适用三种情况：

1）初始值已知且数量较少；

2）初始值是另一个vector对象的副本；

3）所有元素的值都一样；

一般的用法是：先创建空的vector对象，然后运行时动态的添加元素。

Ps:C++标准要求vector应该能在运行时高效快速地添加元素，那么在定义vector对象时设定其大小也就没什么必要了，事实上如果这么做性能可能更差；9.4节还介绍vector提供了允许我们进一步提升动态添加元素的性能。

4.如果循环体内包含有向vector对象添加元素的语句，则不能使用范围for循环。？？？

范围for语句体内不应改变其所遍历序列的大小。//---避免进入死循环

5.不能通过下标方式添加元素，只能对确知已存在的元素执行下标操作。

下标引发的错误不会被编译器发现，而是在运行时产生一个不可预知的值。

### 3.3 迭代器

1.所有标准库容器都可以使用迭代器，但其中只有少数几种才同时支持下标运算符。

2. 如果容器为空，则begin和end返回的是同一个迭代器，都是尾后迭代器；

3.end返回的迭代器并不实际指示某个元素，所以不能对其进行递增或解引用操作；

4.因为只有string和vector一些标准库类型有下标运算符，而所有标准库容器的迭代器都定义了==和!=, 但它们中的大多数都没有定义<运算符，**所以我们养成习惯，使用迭代器和!=。**

这种编程风格在所有标准库提供的所有容器上都有效。

for (auto it = s.begin(); it != s.end(); ++it)

\*it = toupper(\*it);

5.迭代器类型

vector<int>::iterator it;

string::iterator it2;

vector<int>::const\_iterator it3;

string::const\_iterator it4;

const\_iterator和常量指针差不多，能读取但不能修改它所指的元素值；

迭代器返回类型，由具体类型对象是否常量决定；是常量则返回const\_iterator,否则返回iterator;但有时这种默认行为并非我们所需，比如对象只需读操作无须写操作时，最好使用常量类型。

C++11中引入了两个新函数，cbegin和cend,不论vector对象（或string）本身是否是常量，返回值都是const\_iterator;

auto it5 = v.cbegin();//返回const\_iterator类型

6.vector可以动态增长，但也有一些副作用：

1）不能在范围for循环中向vector对象添加元素；

2）任何一种可能改变vector对象容量的操作，比如push\_back，都会使该vector对象的迭代器失效；（即凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素）

## 3.4数组

1.定义数组的时候必须指定数组类型，不允许使用auto关键字由初始值推断类型；

2.和vector一样，数组的元素应为对象，因此不存在引用的数组；

3.要想理解数组声明的含义，最好的办法是从数组的名字开始按照由内向外的顺序阅读；

4.使用数组下标的时候，通常将其定义为size\_t类型，size\_t是与机器相关的无符号类型，在cstddef头文件定义；

5.在大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针，比如

int ia[] = {0,1,2,3,4,5,6};

auto ia2(ia);//ia2是一个整形指针，指向ia的第一个元素

ia2 = 42;//错误，ia2是指针

decltype(ia) ia3 = {0,1,2,3,4,5,6};

6.C++11标准库提供了begin和end函数，分别返回数组的首元素指针和尾元素下一位置的指针；这样指针的使用更简单、安全；

int ia[] = {0,1,2,3,4,5,6,7,8,9};

int \*beg = begin(ia);

int \*end = end(ia);

7.内置的下标运算符不是无符号，这点与vector和string不一样；

int ia[]={1,2,3,4};

int \*p = &ia[2];

int j = p[1];

int k = p[-2]; //p[-2]是ia[0]表示的那个元素

8.使用数组初始化vector对象

int int\_arr[] = {1,2,3,4,5};

vector<int> ivec(begin(int\_arr), end(int\_arr));

ivec的两个指针参数，实际上是数组的位置；

9.**现代的C++程序应当尽量使用vector和迭代器，避免使用内置数组和指针，应该尽量使用string，避免使用C风格的基于数组的字符串；**

10.要使用范围for语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型；(为了避免数组被自动转成指针)

for (auto &row: ia)

for (auto col:row)

cout<<col<<endl;

## ch4.表达式

1.优先使用前置递增运算符，理由：前置递增，把值加1后直接返回改变了的运算对象，后置版本需将原始值存储下来，便于返回这个未修改的内容，如果我们不需要修改前的值，那么后置操作实际是一种浪费，对于整型或指针类型，编译器可能会做优化，但是对于相对复杂的迭代器类型，这种额外的工作就消耗巨大。

建议使用前置版本，这样不仅不需要担心性能问题，写出的代码也更符合编程的初衷。

2.sizeof运算符，返回size\_t类型，两种形式：

Sizeof (type); //类型

Sizeof expr； //表达式结果类型的大小

Sizeof并不实际计算运算对象的值，只取类型的大小

C++11允许我们使用作用域运算符来获取类成员的大小，无须提供一个具体的对象。

3.显示转换，P4.11

命名的强制类型转换，形式

Cast-name <type> (expression); //type 目标类型，expression要转换的值。

Cast-name是static\_cast, dynamic\_cast, const\_cast 和reinterpret\_cast一种

Const\_cast:只能改变运算对象的底层const，只有const\_cast能改变表达式的常量属性，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起编译器错误。

Const char \*cp;

Char \*q = static\_cast<char \*> (cp); //错误static\_cast不能转换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

Const\_cast常常用于有函数重载的上下文中，详见P6.4

## ch5.语句

异常处理

异常是指存在于运行时的反常行为，这些行为超出函数正常功能的范围，典型的异常包括失去数据库连接以及遇到意外输入等；

处理反常行为可能是设计所有系统最难的一部分；

xian

## ch6.函数

1.在C++中，建议使用引用类型的形参替代指针；

2.使用引用避免拷贝：函数通过引用参数访问对象，可避免拷贝；当函数无须改变引用形参的值，最好将其声明为常量引用；

3.管理数组实参，

方法一：传递指向数组首元素和尾后元素的指针。

Void print(const int \*beg, const int \*end)

{

While(beg != end)

Cout << \*beg++ << endl;

}

方法2：专门定义一个表示数组大小的形参

Void print(const int ia[], size\_t size)

{

For (size\_t I = 0; I != size; ++i){

Cout << ia[i] << endl;

}

}

4.含可变形参的函数：详见16.4节

1）如果所有的实参类型相同，可以传递一个initializer\_list的标准库类型，initializer\_list是一个模板，和vector不同的是，initializer\_list对象中的元素永远是常量值；

Initialize\_list<T> lst;

void error\_msg(initializer\_list<string> il)

{

for (auto beg = il.begin(); beg != il.end(); ++beg)

cout << \*beg << " " ;

cout << endl;

}

如果想向initializer\_list形参中传递一个值的序列，必须把序列放在花括号内：

// expected, actual are strings

if (expected != actual)

error\_msg({"functionX", expected, actual});

else

error\_msg({"functionX", "okay"});

2）省略符形参：

省略符形参仅仅用于C和C++通用的类型(仅未访问C代码设置)，特别应该注意的是，大多数类类型的对象在传递给省略符形参时都无法正确拷贝。

5.引用返回左值：调用一个返回引用的函数得到左值（当然不能对常量赋值），其他返回类型得到右值。

char &get\_val(string &str, string::size\_type ix)

{

return str[ix]; // get\_val assumes the given index is valid

}

int main()

{

string s("a value");

cout << s << endl; // prints a value

get\_val(s, 0) = 'A'; // changes s[0] to A

cout << s << endl; // prints A value

return 0;

}

6.列表初始化返回值：

C++11规定，函数可以返回花括号包围的值的列表；

vector<string> process()

{

// . . .

// expected and actual are strings

if (expected.empty())

return {}; // return an empty vector

else if (expected == actual)

return {"functionX", "okay"}; // return list-initialized vector

else

return {"functionX", expected, actual};

}

如果函数返回的是内置类型，则花括号包围的列表最多包含一个值，如果函数返回的是类类型，由类本身定义初始值如何使用。

7.使用尾置返回类型：

C++11规定任何函数的定义都能使用尾置返回，

//func接受一个int类型的实参，返回一个指针，该指针指向含有10个整数的数组

Auto func(int i) -> int (\*) [10];

8.使用decltype定义返回类型：

int odd[] = {1,3,5,7,9};

int even[] = {0,2,4,6,8};

// returns a pointer to an array of five int elements

decltype(odd) \*arrPtr(int i)

{

return (i % 2) ? &odd : &even; // returns a pointer to the array

}

Decltype并不负责把数组类型转换成对应的指针，所哟decltype的结果是个数组，要想表示arrPtr返回指针还必须在函数声明时加一个\*符号。

9.函数重载

1).如果同一作用域内的几个函数名字相同但形参列表不同，称之为重载。

void print(const char \*cp);

void print(const int \*beg, const int \*end);

void print(const int ia[], size\_t size);

main函数不能重载。

2).不允许两个函数除了返回类型，其他所有的要素都相同：

Record lookup(const Account&);

bool lookup(const Account&); // error: only the return type is different

3).形参实际相同

// each pair declares the same function

Record lookup(const Account &acct);

Record lookup(const Account&); // parameter names are ignored

4).顶层const的形参无法和另一个没有顶层const的形参区分

Record lookup(Phone);

Record lookup(const Phone); // redeclares Record lookup(Phone)

Record lookup(Phone\*);

Record lookup(Phone\* const); // redeclares Record lookup(Phone\*)

但如果形参是某种类型的指针或引用，则通过区分指向的是常量对象还是非常量对象可以实现函数重载，此时const是底层的：

// declarations for four independent, overloaded functions

Record lookup(Account&); // function that takes a reference to Account

Record lookup(const Account&); // new function that takes a const Reference

Record lookup(Account\*); // new function, takes a pointer to Account

Record lookup(const Account\*); // new function, takes a pointer to const

6.6.1节会进一步介绍，当我们传递一个非常量对象或指向非常量对象的指针时，编译器会优先选用非常量版本的函数。

何时不该重载：尽管重载能减轻我们为函数起名字，记名字负担，但是最好只重载那些确实非常相似的操作。

5)const\_cast和重载

// return a reference to the shorter of two strings

const string &shorterString(const string &s1, const string

&s2)

{

return s1.size() <= s2.size() ? s1 : s2;

}

这个函数的参数和返回类型都是const string的应用, 如果对两个非常量的string实参调用这个函数,返回结果仍然是const string的引用.

要想实现当实参不是常量,得到的结果是一个普通的引用,可以用const\_cast实现,

string &shorterString(string &s1, string &s2)

{

auto &r = shorterString(const\_cast<const string&>(s1),

const\_cast<const string&>(s2));

return const\_cast<string&>(r);

}//r实际上是一个绑定在了某个初始的非常量实参上的引用, 再强制转换成string&是安全的

10.默认实参：为了使函数既能接纳默认值，又能接受用户指定的值。

1).一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值；

2).函数调用时，实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参（靠右侧位置）；

typedef string::size\_type sz; // typedef see § 2.5.1 (p. 67)

string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');

string window;

window = screen(); // equivalent to screen(24,80,' ')

window = screen(66);// equivalent to screen(66,80,' ')

window = screen(66, 256); // screen(66,256,' ')

window = screen(66, 256, '#'); // screen(66,256,'#')

window = screen(, , '?'); // error: can omit only trailing arguments

window = screen('?'); // calls screen('?',80,' ') 字符可转换为整数

3).当设计含有默认实参的函数时，要合理设置形参的顺序，尽量让不怎么使用默认参数的形参放在前面，常使用默认值的形参放在后面；

4).默认实参声明：多次声明同一函数是合法的，但在给定的作用域中一个形参只能被赋予一次默认实参，并且该形参右侧的所有形参必须都有默认值；

// no default for the height or width parameters

string screen(sz, sz, char = ' ');

string screen(sz, sz, char = '\*'); // error: redeclaration,不能修改意见存在的默认值

string screen(sz = 24, sz = 80, char); // ok: adds default

通常，应该在函数声明中指定默认实参，并将该声明放在合适的头文件中。

5).默认实参初始值：局部变量不能作为默认实参，其他只要能转换成形参所需类型，该表达式就能作为默认实参；

// the declarations of wd, def, and ht must appear outside a function

sz wd = 80;

char def = ' ';

sz ht();

string screen(sz = ht(), sz = wd, char = def);

string window = screen(); // calls screen(ht(), 80, ' ')

void f2()

{

def = '\*'; // changes the value of a default argument

sz wd = 100; // hides the outer definition of wd but does not change the default

window = screen(); // calls screen(ht(), 80, '\*')

}

11.内敛函数和constexpr函数

1)尽量使用inline函数，原因：

a.阅读和理解shorterString函数的调用比读懂等价的条件判断更容易；

b.使用函数确保行为的统一性，每次相关操作按同样方式进行；

c.易于修改，一次修改；

d.函数重复利用，省去程序员重复编码；

e.内敛函数省去函数调用的开销；

2) 内敛函数只是向编译器发出的一个请求，编译器可以选择忽略这个请求；

3) constexpr用于常量表达式的函数，但constexpr函数不一定返回常量表达式；

4)和其他函数不一样，内敛函数和constexpr函数可以多次定义，所以通常放在头文件中；

11.调试帮助

1)assert(expr)预处理宏：定义在cassert头文件中，由预处理器处理，常用于检查“不能发生”的条件，如果expr为真，assert什么也不做，否则输出信息并终止程序。

2)NDEBUG:assert依赖于NDEBUG的预处理变量的状态，如果定义了BDEBUG，assert什么也不做，定义NDEBUG能避免检查各种条件所需的运行时开销；

$g++ -D NDEBUG main.c //相当于添加#define 语句定义NDEBUG

3)C++编译器定义了几个宏

\_\_FUNC\_\_: 文件名字符串

\_\_LINE\_\_:行号整数

\_\_TIME\_\_:编译时间字符串

\_\_DATE\_\_:编译时间字符串

12.函数匹配：

1)函数匹配过程

a.函数匹配第一步选定本次调用对应的重载函数集(每个函数称为候选函数);

b.根据本次调用提供的实参，从候选函数中选出一个可行函数；

可行函数两个特征：一是形参数量与本次调用的实参数量相等，二是每个实参类型与对应的形参类型相同，或者能转换成形参的类型；

如果函数含有默认实参，那么调用该函数时传入的实参可能少于实际使用的实参数量；

如果没找到可行函数，编译器将报告无匹配函数的错误；

c.寻找最佳匹配（如果有的话）;

如果可行函数有多个，则寻找最佳匹配，如果有且只有一个函数满足下列条件，匹配成功：

1. 该函数每个实参的匹配都不劣于其他可行函数需要的匹配；
2. 至少有一个实参的匹配由于其他可行函数提供的匹配；

调用重载函数时应尽量避免强制类型转换，如果实际应用中确实需要强制类型转换，则说明我们设计的形参集合不合理；

13.函数指针

1）函数指针指向的是函数而非对象，函数的类型由它的返回类型和形参类型共同决定，与函数名无关；

2）当把函数名作为一个值使用时，该函数自动转换成指针；

Pf = lengthCompare; //pf指向lengthCompare

Pf = &lengthCompare; //等价赋值语句，取地址符是可选的

3)可以直接使用指向函数的指针调用该函数，无须提前解引用；

bool b1 = pf("hello", "goodbye"); // calls lengthCompare

bool b2 = (\*pf)("hello", "goodbye"); // equivalent call

bool b3 = lengthCompare("hello", "goodbye"); // equivalent call

4)在指向不同函数类型的指针间不存在转换关系，但可以赋值nullptr或0；

5）重载函数的指针必须精确匹配；

6）函数指针形参

// third parameter is a function type and is automatically treated as a pointer to function

void useBigger(const string &s1, const string &s2,

bool pf(const string &, const string &));

// equivalent declaration: explicitly define the parameter as a pointer to function

void useBigger(const string &s1, const string &s2,

bool (\*pf)(const string &, const string &));

可以直接把函数作为实参使用，此时它会自动转换成指针；

// automatically converts the function lengthCompare to a pointer to function

useBigger(s1, s2, lengthCompare);

7)返回指向函数的指针

a.返回类型不会自动转换成指针，必须显示地将返回类型定义为指针；

using F = int(int \*, int) //F是函数类型，不是指针

using PF = int(\*)(int \*, int) //PF是指针类型

PF f1(int); //正确，PF是指向函数的指针，f1返回指向函数的指针

F f1(int); //错误，F是函数类型,f1不能返回一个函数

F \*f1(int); //正确，显示地指定返回类型是指向函数的指针

b.也可以直接声明f1：

int (\*f1(int)) (int \*, int); //f1是一个函数，它返回一个函数指针

可以用尾置返回类型声明：

Auto f1(int) -> int (\*)(int\*, int);

c.用decltype和auto来简化声明函数指针；

string::size\_type sumLength(const string&, const string&);

string::size\_type largerLength(const string&, const string&);

// depending on the value of its string parameter,

// getFcn returns a pointer to sumLength or to largerLength

decltype(sumLength) \*getFcn(const string &);

声明getFcn唯一注意的地方是，decltype作用于某个函数时，返回函数类型而非指针类型, 显示的加\*号表明返回指针而非函数。

## ch7.类

1.类的基本思想是数据抽象和封装。

2.当我们设计类的接口时，应该考虑如何才能使得类易于使用；而当我们使用类时，不应该顾及类的实现机理。

3.成员函数的声明必须在类的内部，它的定义既可以在类的内部也可以在类的外部；

作为接口组成部分的非成员函数，它们的定义和声明都在类的外部；

定义在类内部的函数是隐式的inline函数。

4.在成员函数内部，我们可以直接使用调用该函数的对象的成员；

任何对类成员的直接访问都被看作this的隐式引用；

5.const放在成员函数参数列表后，修饰this，被称作常量成员函数；

常量对象，以及常量对象的引用或指针都只能调用常量成员函数；

6.编译器分两步处理类：先编译成员的声明，然后才轮到成员函数（如果有的话），因此，成员函数可以随意使用类中的其他成员而无需在意这些成员出现的次序。

7.因为IO类属于不能被拷贝的类型，因此在成员函数中应用IO类型作为参数时，只能通过引用传递他们。

### 7.8 构造函数：

类通过一个或几个特殊的成员函数来控制其对象的初始化过程，这些函数叫构造函数。

无论何时，只要对象被创建，就会执行构造函数，构造函数的任务是，初始化类对象的数据成员。

1）构造函数与类同名，没有返回类型；

2）构造函数不能被声明为const，因为创建类的一个const对象时，直到构造函数执行完成初始化过程，对象才取得其“常量”属性；

3）若类没有显示地定义构造函数，那么编译器会隐式地定义一个默认构造函数，其遵循以下规则：

a.如果存在类内初始值，用它来初始化成员；

b.否则，默认初始化该成员；

4）有些类不能依赖合成的默认构造函数：

a.编译器只有在发现类不包含任何构造函数情况下，才会生出默认构造函数。一旦定义了一些其构造函数，除非我们再定义一个默认构造函数，否则不自动生成。

b.对于某些类来说，合成的默认构造函数可能是错的。比如定义在块中的内置类型或复合类型的对象被默认初始化，它们的值是未知的。

c.编译器不能为某些类合成构造函数。比如类中包含一个其他类类型的成员，且这个成员的类型没有默认构造函数。

5）// constructors

Sales\_data() = default; //无任何实参，所以是默认构造函数，default既可以和声明一起出现在类内部（内敛的），也可以和定义一起出现在类外部；

//构造函数初始化列表，

Sales\_data(const std::string &s): bookNo(s) { }//用s初始化bookNo,其他被忽略的成员，以和默认构造函数相同的方式初始化

Sales\_data(const std::string &s, unsigned n, double p):

bookNo(s), units\_sold(n), revenue(p\*n) { } //如果编译器不支持内置初始化，那么构造函数必须显式的初始化每个内置类型的成员。

Sales\_data(std::istream &);//在类外部定义构造函数

6)某些类不能依赖于合成的版本，特别是当类需要分配类对象之外的资源时（比如动态分配内存），合成会失效。

使用vector和string类能避免分配和释放内存带来的复杂性，如果类包含vector和string成员，则其拷贝、赋值和销毁的合成版本能够正常工作。

7）如果成员是const、引用或者属于某种未提供默认构造函数的类类型，必须将其初始化；

8）初始化是直接初始化成员，赋值是先初始化再赋值；除了效率问题外，更重要的是一些数据成员必须被初始化。

**建议养成使用构造函数初始化的习惯。**

9）成员的初始化顺序与它们在类定义中的出现顺序一致，而与构造函数初始值列表中初始值位置无关；

最好令构造函数初始值的顺序与成员声明的顺序保持一致，并尽量避免使用某些成员初始化其他成员；

最好用构造函数的参数作为成员的初始值；

10）如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数；

Sales\_data(std::string s = " "):bookNo(s){}

实际上就是提供了默认构造函数。

11）委托构造函数

class Sales\_data {

public:

//非委托构造函数使用对应的实参初始化成员

Sales\_data(std::string s, unsigned cnt, double price):

bookNo(s),units\_sold(cnt),revenue(cnt\*price){}

Sales\_data():Sales\_data("", 0, 0) {} //默认构造函数，委托给三参数版本构造函数

Sales\_data(std::string s):Sales\_data(s, 0, 0){}//接受一个参数s, 委托给三参数版本

Sales\_data(std::istream &is):Sales\_date() //先委托默认构造函数，继而委托三参数版本

{read(is, \*this)} //然后执行函数体，调用read函数

};

//如果委托函数有函数体的话，先执行完函数体代码，控制权才会交还给委托者的函数体

7.public访问说明符之后的成员在整个程序内可被访问，public定义类的接口；

private之后的成员可以被类的成员函数访问，但不能被使用该类的代码访问，private封装了类的实现细节；

类可以在它的第一个访问说明符之前定义成员，这种成员访问权限依赖于类定义方式，如果使用struct关键字，则第一个public之前的成员是public,

如果使用class，则是prive的。

封装的益处：

a.确保用户代码不会破环封装对象的状态；

b.被封装的类的具体实现细节可随时改变，无须调整用户级别代码；只要类的接口不变，用户代码就无须改变。

类型成员必须先定义再使用,访问控制与其他成员类同，一般出现在类开始的地方：

public:

typedef std::string::size\_type pos;

private:

pos cursor=0;

8.类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数称为它的友元。

友元声明只能出现在类定义的内部。（一般来说最好在类定义开始的地方或结束前的地方集中声明）

友元的声明仅仅制定了访问的权限，如果希望类的用户能够调用友元函数，必须在友元声明之外再专门对函数进行一次声明。

可以声明一个类的友元，还可以令一个类的成员函数为友元；

每个类负责控制自己的友元类或友元函数；

友元关系不存在传递性；

友元声明的作用是影响访问权限，它本身并非普通意义的声明（所以必须在类外部声明该函数为可见，类才可调用）

9.mutable定义成员是一个可变成员，因此任何成员函数，包括const函数都能改变它的值；

10.当提供一个类的初始值时，必须以符号=或者花括号表示。

11.类成员函数中，有const的重载：

a.基本愿望是避免多出使用同样的代码；

b.开发过程display可能变得更加复杂；

c.开发过程中,do\_display添加某些调试信息，旨在这一处添加或删除更容易；

d.这个额外的函数调用，不会增加任何开销，因为是内敛函数调用；

public:

// display overloaded on whether the object is const or not

Screen &display(std::ostream &os)

{ do\_display(os); return \*this; }

const Screen &display(std::ostream &os) const

{ do\_display(os); return \*this; }

private:

// function to do the work of displaying a Screen

void do\_display(std::ostream &os) const {os << contents;}

12.不完全类型（只声明，没定义）只能在非常有限的情景下使用，可以定义指向这种类型的指针或引用，也可以声明（但不能定义）以不完全类型

作为参数或者返回类型的函数。

所以一个类成员类型不能是该类自己，但可以含指向自身类型的引用或指针；

13.隐式的类类型转换

如果构造函数只定义一个实参，它实际上定义了转换为此类类型的隐式转换机制，叫做转换构造函数。

string null\_book = "999-99-9999";

item.combine(nul\_book);

该调用是合法的，编译器用给定的string自动创建了一个Sales\_data对象。

2）只允许一步class type转换

//错误的，需要用户定义两种转换：

//a.把"999-99-9999"转换成string

//b.把这个string转换成Sales\_data

item.combine("999-99-9999");//错误

正确用法如下：

item.combine(string("999-99-9999"));

itme.combine(Sales\_data("999-99-9999"));

14.抑制构造函数定义的隐式转换

1）

class Sales\_data{

public:

Sales\_data() = default;

Sales\_data(const std::string &s, unsigned n, double p):

bookNo(s),units\_sold(n),revenue(p\*n) {}

explicit Sales\_data(const std::string &s): bookNo(s) {}

explicit Sales\_data(std::istream &);

};

explicit只对一个实参的构造函数有效，explicit定义的函数，不再支持隐式转换；

explicit关键字只允许出现在类内的构造函数声明处；

2）class type转换不总是有效

item.combin(cin);//无效，其值不一定有效？

3）explicit构造函数只能用于直接初始化

Sales\_data item1(null\_book); //OK

Sales\_data item2 = null\_book;//error,不能将explicit构造函数用于拷贝形式的初始化过程

4）为转换显式地用构造函数

itme.combine(Sales\_data(null\_book));

item.combine(static\_cast<Sales\_data>(cin));//static\_cast可以使用explicit的构造函数

ps：标准库中含有显示构造函数的类

1. 接受一个单参数的const char\*的string构造函数，不是explicit的；
2. 接受一个容量参数的vector构造函数，是explicit的；

15.聚合类

1）聚合类满足条件：

a.所有成员都是public;

b.没有定义任何构造函数；

c.没有类内初始值；

d.没有基类，也没有virtual函数；

struct Data{

int ival;

string s;

};

Data al = {0, "Anna"};

注意：初始化顺序必须和声明的一致；初始值列表的元素绝对不能超过类的成员数量；

16.字面值常量类

数据成员都是字面值类型的聚合类，是字面值常量类。

如果一个类不是聚合类，但符合下述要求，也是字面值常量类：

a.数据成员都必须是字面值类型；

b.类必须至少含有一个constexpr构造函数；

c.如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式；或者如果成员属于某种类类型，则初始值必须使用成员自己的constexpr构造函数；

d.类必须使用析构函数的默认定义，该成员负责销毁类的对象；

为了能在编译过程中随时展开，constexpr函数被隐式地制定为内联函数；

2）尽管构造函数不能是const的，但字面值常量类的构造函数可以是constexpr函数，事实上一个字面值常量类必须至少提供一个constexpr构造函数；

constexpr构造函数体一般来说应该是空的。（既符合构造函数规则（木有返回值），又要符合constexpr规则（拥有唯一的可执行语句就是返回语句））

3）constexpr构造函数必须初始化所有数据成员，初始值或者使用constxpr构造函数，或者是一条常量表达式；

17.类的静态成员

1）静态成员只与类本身直接相关，而不是与具体对象关联；

2）静态成员函数也不与任何对象绑定，它们不包含this指针，静态成员函数不能声明成const。

虽然静态成员不属于类的某个对象，但是我们仍然可以使用类的对象，引用或指针来访问静态成员：

double r;

r = Account::rate();

Account ac1;

Account \*ac2 = &ac1;

r = ac1.rate();

r = ac2->rate();

成员函数不通过作用域运算符就能直接使用静态成员：

class Account {

public:

void calculate(){amount += amount \* interestRate;}

private:

static double interestRate;

}

3)既可以在类的内部也可以在类的外部定义静态成员函数。在类外部定义时，不能重复static，该关键字只出现在类内部；

4）一般来说，不能在类的内部初始化静态成员，而必须在类的外部定义和初始化每个静态成员，一个静态数据成员只能定义一次；

类似于全局变量，静态数据成员定义在任何函数之外；

最好是把静态数据成员的定义与其他非内敛函数定义放在同一个文件中（确保只定义一次）；

5）通常，类的静态成员不应该在类的内部初始化，然而我们可以为静态成员提供const正数类型的类内初始值，不过要求静态成员必须是字面值类型的constexpr。

即使一个常量静态数据成员在类内部初始化了，通常情况下也应该在类的外部定义一下该成员。

6）静态数据成员可以是不完全类型，特别的，静态数据成员的类型可以就是它所属的类类型，而非静态数据成员则只能声明称它所属的类的指针或引用；

class Bar {

public:

//...

private:

static Bar mem1;//静态成员可以是不完全类型

Bar \*mem2; //正确，指针成员可以是不完全类型

Bar mem3; //错误，数据成员必须是完全类型

}；

7）可以使用静态成员作为默认实参

class Screen {

public:

Screen & clear(char = bkground);//bkground表示一个在类中稍后定义的静态成员

private:

static const char bkground;

};

# PART2:C++标准库:

## Ch8.IO库

### 8.1 I/O类

1.IO对象无拷贝或赋值，也不能将形参或返回类型设置为流类型；

进行IO操作的函数，通常以引用方式传递和返回流；

读写一个IO对象会改变其状态，所以传递和返回的引用不能是const的。

2.一个流一旦发生错误，其上后续的IO操作都会失败。只有当一个流处于无错状态时，我们才可以从它读取数据，向它写入数据。

3.管理条件状态：

1）auto old\_state = cin.rdstate(); //记住cin的当前状态

cin.clear(); //使cin有效

process\_input(cin); //使用cin

cin.setstate(old\_state); //将cin还原成之前的状态

2）复位failbit和badbit，保持其他标志位不变

cin.clear(cin.rastate() & ~cin.failbit & ~cin.badbit);

3）导致缓冲区刷新：

a.程序正常结束；

b.缓冲区满时；

c.可以使用操作符，如endl来显示刷新缓冲区；

d.在每个输出操作之后，我们可以用操作符unitbuf设置流的内部状态，来清空缓冲区；

默认情况下，对cerr是设置unitbuf的，写到cerr的内容都是立即刷新的；

e.一个输出流可能被关联到另一个流，这种情况下，当读写被关联的流时，关联到的流缓冲区会被刷新；

默认，cin和cerr都关联到cout,因此读cin或写cerr都会导致cout缓冲区刷新；

4）显示刷新缓冲区

cout << "hi" <<endl; //输出hi和一个换行，然后刷新缓冲区

cout << "hi" <<flush; //输出hi，然后刷新缓冲区

cout << "hi" <<ends; //输出hi和一个空字符，然后刷新缓冲区

5）unitbuf

cout << unitbuf; //所有输出会立即刷新缓冲区

cout << nounitbuf；//恢复正常的缓冲模式

6）如果程序崩溃，缓冲区不会被刷新；

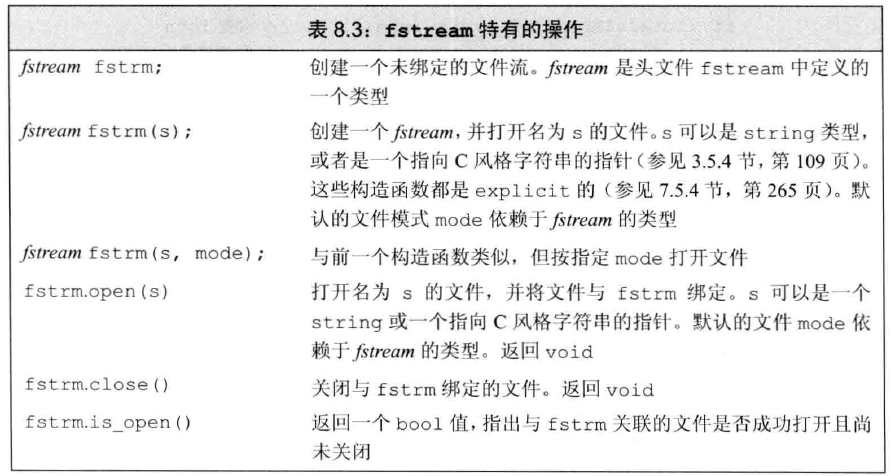
调试一个已经崩溃程序时，需要确认那些你认为已经输出的数据确实已经刷新了。

7）交互式系统通常应该关联输入流和输出流，这样所有输出都会在操作之前被打印出来。

8）每个流同时最多关联到一个流，但多个流可以同时关联到同一个ostream.

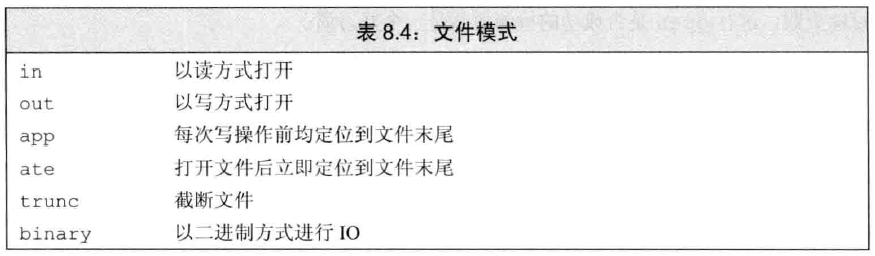
### 8.2 文件IO

1.除了继承自iostream类型的行为之外，fstream类型还增加了一些新的成员来管理与流关联的文件：



2. 当一个fstream对象离开其作用域时，与之关联的文件会自动关闭（当fstream销毁时，close会自动被调用）。

3.文件模式：



指定文件模式限制如下：

只可以对ofstream或fstream对象设定out模式；

只可以对ifstream或fstream对象设定in模式；

只有当out也被设定时，才可设定trunc模式；

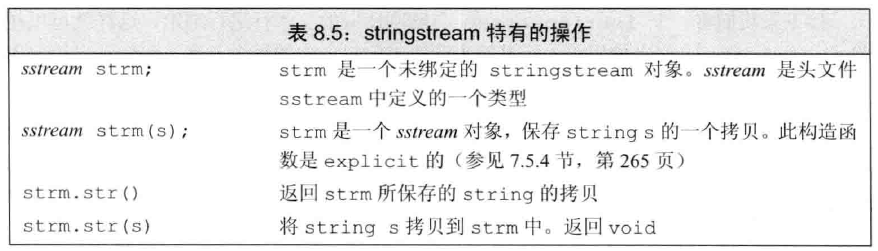
只要trunc没被设定，就可以设定app模式；

默认情况下：以out模式打开的文件会被trunc，要保留原内容，必须显示app;

Ate和binary模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用；

### 8.3 string流

1.String流除了继承自iostream的行为之外，可以对stringstream对象调用这些操作：



每个IO对象都维护一组条件状态，用来指出此对象上是否可以进行IO操作；标准库提供了一组函数，用来设置和检测这些状态。

## Ch9.顺序容器

1.顺序容器：为程序员提供了控制元素存储和访问顺序的能力，这种顺序不依赖于元素的值，而是与元素加入容器时的位置有关。

2.list和forward\_list两个容器的设计目的是，令容器任何位置的添加和删除操作都很快速；

但都不支持随机访问，为了访问一个元素，只能遍历整个容器；

而且，相比vector,deque和array,这两个容器的额外内存开销也很大；

3.array是一种更安全、更容易使用的数组类型。array不支持添加和删除元素，以及改变容器大小的操作；

forward\_list没有size操作。

新标准库的容器比旧版本快得多（原因在p470解释）；

新标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好或更好；

现代C++程序应该使用标准库容器，而不是更原始的数据结构；

4.确定使用哪个容器：通常使用vector是最好的选择，除非有很好理由选择其他容器

1）除非有很好的理由选择其他容器，否则用vector;

2)如果程序有很多小的元素，且空间的额外开销很重要，就不要使用list或forward\_list.

3)如果程序要求随机访问，应使用vector或deque.

4)如果程序要求在头尾位置插入或删除元素，但不会在中间位置进行插入或删除操作，用deque;

5)如果程序只有在读取输入时才需要在容器中间位置插入元素，随后需要随机访问元素：

a.先确定是否真的需要在容器中间位置添加元素，当处理输入时，通常可以很容易地向vector追加数据，然后用标准库的sort重排，从而避免在中间位置添加元素；

b.如果必须在中间位置插入元素，考虑输入阶段使用list，一旦输入完成，将list中的内容拷贝到一个vector中；

如果程序既需要随机访问元素，又需要在容器中间位置插入元素，那取决于在list或forward\_list中访问元素与vector或deque中插入/删除元素的相对性能；

应用中占主导地位的操作（执行访问操作多还是插入/删除更多？），决定了容器类型的选择。

如果不确定应该使用哪种容器，可以在程序中只使用vector和list的公共操作：使用迭代器，不使用下标操作，避免随机访问。

5.容器均定义为模板类，对于大多数，但不是所有容器，必须提供额外信息来生成特定的容器类型；

6.通过类型名，我们可以在不了解容器中元素类型的情况下使用它。

7.1）创建一个容器为另一个容器的拷贝，两个容器的类型及其元素类型必须匹配；

2）当传递迭代器参数来拷贝一个范围时，就不要求容器类型相同，而且新容器和原容器中的元素类型也可以不同，

只要能将要拷贝的元素转换为要初始化的容器的元素类型即可。

3）用迭代器参数来使用构造函数，可以拷贝一个容器中的子序列。

8.如果元素是内置类型或者是具有默认构造类型的类类型，可以只为构造函数提供一个容器大小参数。

如果元素类型没有默认构造函数，除了大小参数外，还必须指定一个显示的元素初始值。

9.只有顺序容器的构造函数才接受大小参数，关联容器并不支持。

10.array类型，包括元素类型和大小：

array<int, 10>::size\_type i;//正确

array<int>::size\_type j; //错误，array<int>不是一个类型

11.array的元素类型是类类型，那么该类必须有一个默认构造函数，以使值初始化能够进行；

12.内置数组类型不能进行拷贝或对象赋值操作，但array是可以的：

array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};

array<int> copy = digits;

array拷贝要求：容器类型相同，元素类型和大小也相同。

由于右边运算对象的大小可能与左边 运算对象的大小不同，因此array类型不支持assign,也不允许用花括号包围的值列表进行赋值；

13.赋值相关运算，会导致指向左边容器内部的迭代器、引用和指针失效；

而swap操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效。

14.assign(仅顺序容器)

1）顺序容器（array除外）使用assign允许我们从一个不同但相容的类型赋值，或者从容器的一个子序列赋值；

list<string> names;

vector<const char\*> oldstyle;

names = oldstyle; //错误，容器类型不同

names.assigne(oldstyle.cbegin(),oldstyle.cend()); //正确，可以将const char\*转换为string

assign的迭代器不能指向调用assign的容器本身。

2）assign另一个用法，接受一个整型值和一个元素值

list<string> slist(1); //初始化容器，含一个元素，值为空

slist1.assign(10,"hiya");//10个元素，值都是"hiya"

15.swap：交换两个相同类型容器的内容。

1）除array外，交换两个容器内容的操作保证会很快--元素本身并未交换，swap只是交换了两个容器的内部数据结构；

2）除string外，指向容器的迭代器、引用和指针都不会失效；

3）与其他容器不同，对一个string调用swap会导致迭代器、引用和指针失效；

4) 对于array，在swap操作之后，指针、引用和迭代器所绑定的元素保持不变，但元素值已经和另一个array中对应的元素进行了交换；

5）新标准中，容器既提供成员函数版本的swap,也提供非成员版本的swap;而早期版本只提供成员函数版本；统一使用非成员版本的swap是一个好习惯。

16.除了一个例外，每个容器类型都有三个与大小相关的操作：

size:返回容器中元素数目；

empty：当size为0时，返回true,否则返回false;

max\_size：返回一个大于或等于该容器所能容纳的最大元素数的值；

forward\_list支持max\_size和empty,不支持size.

17：每个容器都支持相等运算符（=和!=）,除了无序关联容器外的所有容器都支持关系运算符(>,>=,<,<=)。关系运算符要求左右两边的运算对象必须是相同类型容器，且必须保存相同类型的元素；

18.向一个vector,string或deque插入元素会使所有指向容器的迭代器、引用和指针都失效；

19.当用一个对象来初始化容器时，获将一个对象插入到容器中时，实际上放入到容器中的是对象值的一个拷贝，而不是对象本身。

20.新标准引入的emplace\_front,emplace和emplace\_back,这些操作构造而不是拷贝元素。传入的参数则是传给元素类型的构造函数。

21.对一个空容器调用front和back，就像使用一个越界的下标一样，行为将是未定义的；

如果希望确保下标是合法的，可以使用at成员函数；

vector<string> svec;//空容器

cout << svec[0]; //运行时错误

cout << svec.at[0]; //抛出一个out\_of\_range异常

22.删除迭代器p所指向的元素，返回一个指向被删元素之后元素的迭代器；若p指向尾元素，则返回尾后迭代器；若p是尾后迭代器，则函数行为未定义；

删除迭代器b和e所指定范围内的元素，返回一个指向最后一个被删元素之后的迭代器；若e本身就是尾后迭代器，则函数也返回尾后迭代器；

删除元素的成员函数并不检查其参数，在删除元素之前，程序员必须确保它是有效的；

23.在一个forward\_list中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。

24.resize操作，如果容器保存的是类类型元素，则resize向容器添加新元素，则我们必须提供初始值，或者元素类型必须提供一个默认构造函数；

25.容器操作可能使迭代器失效：

1）如果容器是vector和string，且存储空间被重新分配，则指向容器的迭代器，指针和引用都会失效；如果空间未重新分配，插入元素之后的迭代器，指针和引用失效；

2）对于deque,插入到除首尾位置之外的任何位置都会导致迭代器，指针和引用失效；

3）对于list和forward\_list，指向容器的迭代器，指针和引用仍然有效；

4）对于vector或string,指向被删元素之前的元素迭代器，指针和引用仍有效；

使用失效的迭代器，指针和引用是严重的运行时错误；

必须保证每次改变容器的操作之后都正确地重新定位迭代器，这个建议对vector,string和deque尤为重要。

26.不要保存end返回的迭代器：

添加或删除元素的循环程序必须反复调用end，而不能在循环之前保存end返回的迭代器当作容器末尾使用；C++标准库的实现中end()操作都很快，部分就是这个原因；

27.每个vector实现都可以选择自己的内存分配策略，但是必须遵循的一条原则是：只有当迫不得已时才可以分配新的内存空间；

调用shrink\_to\_fir只是一个请求，标准库并不保证退还内存。

string操作

28.当我们从一个const char \*创建string时，指针指向的数组必须以空字符结尾，拷贝操作遇到空字符时停止；

容器适配器：

29.本质上，一个适配器是一种机制，能使某种事物的行为看起来像另外一种事物一样。

一个容器适配器接受一种已有的容器类型，使其行为看起来像一种不同的类型。

30.标准库定义了三个顺序容器适配器：stack,queue和priority\_queue。

31.每个适配器都定义了两个构造函数：

1）默认构造函数创建一个空对象；

2）接受一个容器的构造函数拷贝该容器来初始化适配器；

32.默认情况下，stack和queue是基于deque实现的，priority\_queue是基于vector实现的。

33.所有适配器都要求容器具有添加和删除以及访问尾元素的能力，所以不能用array和forward\_list来构造适配器。

34.stack:只要求push\_back,pop\_back和back操作，因此可以用任何容器类型来构造（array，forward\_list除外）；

queue：要求back,push\_back,fron和push\_front,因此可以构造于list和deque之上，但不能基于vector;

priority\_queue:除了front,push\_back和pop\_back操作之外还要求随机访问能力，因此她可以构造于vector和deque，但不能基于list.

35.每个容器适配器都基于底层容器类型的操作，定义了自己的特殊操作；我们只可以使用适配器操作，而不能使用底层容器类型的操作。

## Ch10.泛型算法

1.通常不对关联容器使用泛型算法。

2.关联容器可用于只读取元素的算法，但对其使用泛型搜索算法几乎总是个坏主意。使用关联容器定义的专用find成员会比调用泛型find快很多。

实际编程中，如果真要对一个关联容器使用算法，要么是将它当作一个源序列，要么是当作一个目的位置（使用inserter将一个插入器绑定到一个关联容器）。

3.不能对一个multimap或一个multiset进行下标操作，因为这些容器中可能有多个值与一个关键字相关联。

如果关键字并不在map中，下标操作会为它创建一个元素并插入到map中，关联值将进行值初始化。

只可以对非const的map使用下标操作。

c[k]; //返回关键字为k的元素，如果k不在c中，添加一个关键字为k的元素，对其值进行初始化

c.at(k)//访问关键字为k的元素，带参数检查；若k不在c中，抛出一个out\_of\_range异常

4.解引用一个map迭代器时，会得到一个value\_type对象。

## Ch11.关联容器

## Ch12.动态内存

# PART3:类设计者的工具

## Ch13.拷贝控制

## Ch14.重载运算与类型转换

## Ch15.面向对象程序设计

## Ch16.模板与泛型编程

# PART4:高级主题：