1. 基本数据类型：

|  |
| --- |
| void, bool, char, short  Int, float, double, long |

一个char的空间应确保可以存放机器基本字符集中任意字符对应的数字值，也就是说，一个char的大小和一个机器字节一样。

类型char的表现形式有可能是unsinged char，也有可能是signed char，具体是哪种由编译器决定。

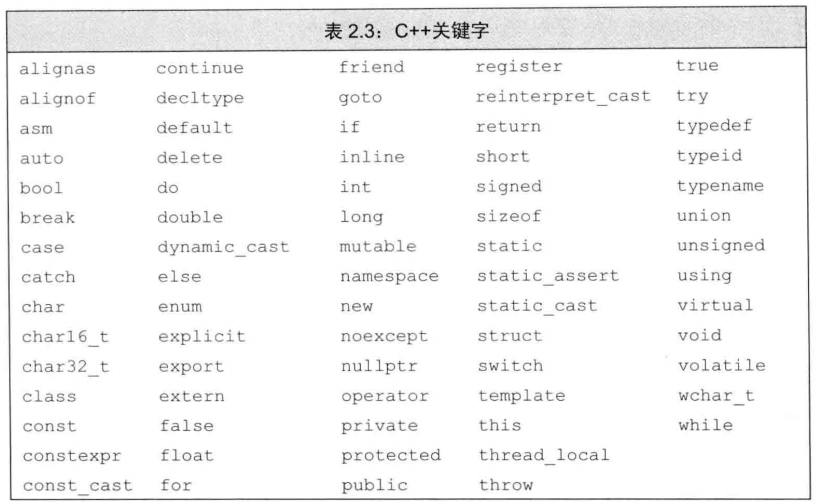
1. 变量定义

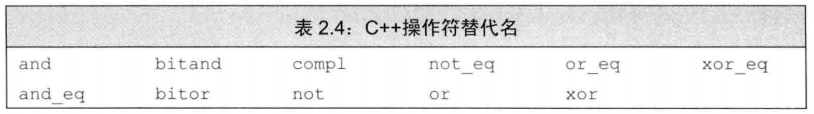
**基本数据类型** (一组)**类型声明符**

面对一条比较复杂的指针或引用的声明语句时，从右向左阅读有助于弄清楚它的真实含义。

默认情况下，类型修饰符从右向左依次绑定。

但是，如果是数组的声明，最好的办法是从数组的名字开始，由内而外(有小括号的情况)，再由右向左的顺序去阅读。





1. 变量初始化

定义于任何函数体之外的内置类型变量被初始化为0；函数体内的内置类型变量如果没有初始化，则其值未定义。

类对象如果没有显示地初始化，则其值由类确定。

建议初始化每一个内置类型的变量。

1. Void\* 指针

Void\* 指针能做的事儿比较有限：拿它和别的指针比较，作为函数的输入或者输出，赋值给另外一个void\* 指针。

以void\* 的视角来看内存空间就仅仅是内存空间，无法访问内存空间中所存的对象。

1. 常量指针、常量引用

所谓指向常量的指针或引用，不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变指向对象的值。

1. 顶层const和底层const

顶层const(top-level const)表示指针本身是个常量，而用名词底层const（low-level const）表示指针所指对象是个常量（**基本类型部分**）。

1. constexpr和常量表达式

常量表达式是指值不会改变并且在编译过程中就能得到计算结果的表达式。显然，字面值属于常量表达式，用常量表达式初始化的const对象也是常量表达式。

C++11新标准规定，允许将变量声明为constexpr类型，以便由编译器来验证变量的值是否是一个常量表达式。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：

constexpr int mf = 20;

constexpr int limit = mf + 1;

新标准还允许定义一种特殊的constexpr函数，这种函数应该能够足够简单以使得编译时就可以计算其结果，这样就能使用constexpr函数去初始化constexpr变量了。

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

1. 类型别名

有两种方法用于定义类型别名：

1. 传统方法，使用关键字**typedef**:

typedef double wages: //wage是double的同义词

1. C++11的新方法，使用别名声明(alias declaration)来定义：

using wages = double;//wage是double的同义词

如果某个类型别名指代的是复合类型或者常量，则把它用到声明语句中就会产生意想不到的后果：

typedef char \*pstring;

const pstring cstr=0; //**等价于char \* const cstr=0;不是const char \*cstr=0;**

1. auto类型说明符

C++11新标准引入了auto类型说明符，和原来那些只对应一种特定类型的说明符(比如int, double)不懂，auto让编译器通过初始值来推算变量的类型，显然，auto定义的变量必须有初始化值：

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

1. decltype类型指示符

希望从表达式的类型推断出要定义的变量的类型，但不想用该表达式的值初始化变量。C++11引入decltype，它的作用是选择并返回操作数的数据类型。编译器分析表达式并得到它的类型，却不实际计算表达式的值。

|  |
| --- |
| for(decltype(vec.size()) i= 0； i < vec.size(); ++i)  cout<<vec[i]<<endl;  for(decltype(s.size()) idx = 0; idx<s.size() && !isspace(s[idx]); ++idx)  s[idx] = toupper(s[idx]); |

切记：decltype((variable))（注意，是双层括号）的结构永远是引用，而decltype(variable)结果只有当variable本身就是一个引用时才是引用。

1. 自定义数据结构struct

struct Sales\_data{}

string bookNo;

unsigned units\_sold = 0;

double revenue = 0.0;  
}；

C++11新标准规定，可以为数据成员提供一个类内初始化值(in-class initializer)，创建对象时，类内初始化值将用于初始化数据成员，没有初始化值得成员将被默认初始化。

1. 范围for(range for)语句

这种语句遍历给定序列中的每个元素，并对序列中的每个值执行某种操作，其语法形式为：

for (declaration: expression)

statement

|  |
| --- |
| string str(“some string”)  //每行输出str中的一个字符  for(auto c: str)  cout<< c << endl; |

如果想要在上诉for语句中改变expression，则必须把declaration定义成引用类型形式。记住，所谓引用只是给定对象的一个别名，因此当使用引用作为循环控制变量时，这个变量实际上依次被绑定到序列上的每个元素。

|  |
| --- |
| string s(“Hello World!”)  //转换成大写形式  For(auto &c : s)  c = toupper(c) |

1. vector

C++标准要求vector应该能在运行时高效快速地添加元素，因此，为了vector对象能高效地增长，那么在定义vector对象的时候设定其大小就没有必要了。只有一种情况例外，那就是所有元素的值都一样。

范围for语句体内不应该改变其所遍历序列的大小(线程不安全的证据？)。

1. 数组

数组是一种复合类型。数组中元素的个数也属于数组类型的一部分，编译的时候纬度应该是已知的。也就是说，数组纬度必须是一个常量表达式。

**现代的C++程序员应当尽量使用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。**

1. 强制类型转换

**static\_cast:**

任何具有明确定义的类型转换，**只要不包含底层const，都可以使用static\_cast**。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。

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

**const\_cast:**

const\_cast只能改变运算对象的底层const。一旦我们去掉某个对象的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。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。

const\_cast常用于有函数重载的上下文中。使用const\_cast的目的,在于某些变量原本不是const的,但由于某种特殊原因,无意间被变成了const的。例如使用了一个const引用指向了一个本来不是const的对象，结果写了一些代码之后发现它实际上需要被修改。这在平时的工作中不会遇到，因为你可以直接把const引用修改成非const的，但C++中可能的情况太多，尤其考虑到很多复用的时候，有时还是会出现本不该是const的对象被const引用了这种情况。尤其是使用模板，比较复杂的情况。

**reinterpret\_cast:**

reinterpret\_cast通常为运算对象的位模式提供较低层次上的重新解释，本质上依赖于机器，要想安全使用reinterpret\_cast必须对涉及的类型和编译器实现转换的过程都非常了解。

允许将任何指针转换为任何其他指针类型。 也允许将任何整数类型转换为任何指针类型以及反向转换。reinterpret\_cast运算符是用来处理无关类型之间的转换；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。

所以总结来说：reinterpret\_cast用在**任意指针（或引用）类型之间的转换**；以及**指针与足够大的整数类型之间的转换**；从**整数类型（包括枚举类型）到指针类型，无视大小**。

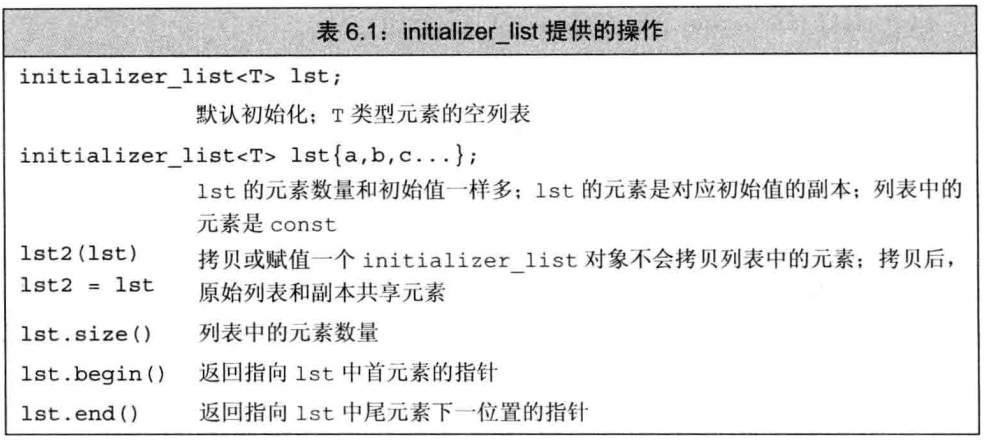
reinterpret\_cast不能像const\_cast那样去除const修饰符。

**dynamic\_cast:**

1. 含有可变形参的函数

为了能编写能处理不同数量实参的函数，C++11新标准提供了两种主要的方法：如果所有的实参类型相同，则可以传递一个名为initializer\_list的标准库类型；如果实参的类型不同，我们可以编写一种特殊的函数，也就是所谓的可变参数模板。

C++还提供了一种特殊的形参类型----省略符形参，可以用它传递可变数量的实参，这种功能一般只用于与C函数交互的接口程序。



|  |
| --- |
| Void error\_msg(ErrorCode e, initializer\_list<string> il)  {  Cout<<e.msg()<<endl;  For(const auto &elem: il)  Cout<<elem<<” “;  Cout<<endl;  }  Void error\_msg2(initializer\_list<string> il)  {  For(auto beg=il.begin(); beg != il.end(); ++beg)  Cout<<\*beg<<””;  Cout<<endl;  }  …  If(excepted != actual)  Error\_msg(ErrorCode(42), {“functionX”,expected, actual});  Else  Error\_msg(ErrorCode(0),{“functionX”,”okay”});  … |

省略符形参：

省略符形参是为了便于C++程序访问某些特殊的C代码而设置的，这些代码使用了名为varargs的C标准库功能，通常情况下省略符形参不应用于其他目的。

省略符形参只能出现在形参列表的最后一个位置，它的形式无外乎以下两种：

void foo(param\_list, …);

void foo(…);

|  |
| --- |
| 我们使用可变参数应该有以下步骤:  ⑴在程序中将用到以下这些宏:  void va\_start( va\_list arg\_ptr, prev\_param );  type va\_arg( va\_list arg\_ptr, type );  void va\_end( va\_list arg\_ptr );  va在这里是variable-argument(可变参数)的意思.  这些宏定义在stdarg.h中,所以用到可变参数的程序应该包含这个头文件. |

1. 列表初始化返回值

C++11新标准规定，函数可以返回花括号包围的值得列表。类似于其他分会结果，此处的列表也用来对表示函数返回的临时量进行初始化。如果列表为空，临时量执行值初始化；否则，返回的值由函数的返回类型决定。

1. 返回数组指针

因为数组不能拷贝，所以函数不能返回数组。不过，函数可返回数组的指针或引用。C++返回数组指针的函数形式如下：

Type (\*functionName(parameter\_list))[ArrDimenstion]

这种语法比较复杂，但是可以使用类型别名来进行简化：

typedef int art[10];

using arrT = int [10];

arrT \* func(int i);

使用尾置位返回类型：

在C++11新标准中，还有一种可以简化上述func声明的方法，那就是使用尾置返回类型(trailing return type)。任何函数的定义都能使用尾置位返回，但是这种形式对于返回类型比较复杂的函数最有效，比如返回类型是数组的指针或者数组的引用。尾置返回类型跟在形参列表后面并以一个->符号开头。为了表示函数真正的返回类型跟在形参列表之后，我们在本应该出现返回类型的地方放置一个auto。

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

使用decltype

还有一种情况，如果我们知道函数返回的指针指向那个数组，就可以使用decltype关键字声明返回类型。

|  |
| --- |
| int odd[] = {1,3,5,7,9};  int event[] = {0,2,4,6,8};  decltype(odd) \* arrPtr(int i){  return (i%2) ? &odd: &even; //返回一个指向数组的指针  } |

1. constexpr函数

constexpr函数是指能用于常量表达式的函数。定义constexpr函数的方法与其他函数类似，不过需要遵守几项约定：函数的返回类型及其所有形参的类型都得是字面值类型，而且函数体中必须有且仅有一条return语句。

为了能在编译过程中随时展开，constexpr函数被隐式地指定为内联函数。

1. 调试帮助
2. 、assert预处理宏

assert是一种预处理宏(preprocessor marco)，其实就是一个预处理变量，它的行为有点类似于内联函数。

|  |
| --- |
| assert(expr); |

首先对expr求值，如果表达式为假(即0)，assert输出信息并终止程序的执行，如果表达式为真(即非0)，assert什么也不做。

常用于检查“不能发生”的条件。

1. NDEBUG预处理变量

assert的行为依赖于一个名为NDEBUG的预处理变量的状态，如果定义了NDEBUG，则assert什么也不做。默认状态下是没有定义NDEBUG，此时assert将执行运行时检查。

我们可以使用一个#define语句定义NDEBUG，从而关闭调试状态。同时，很多编译器都提供一个命令行选项使我们可以定义预处理变量：

$ CC –D NDEBUG main.c

这条命令的作用等价于在main.c文件的一开始写#define NDEBUG。

除了用于assert外，也可以使用NDEBUG编写自己的条件调试代码。如果NDEBUG未定义，将执行#ifndef和#endif之间的代码；如果定义了NDEBUG，这些代码将被忽略掉。

|  |
| --- |
| …  #ifndef NDEBUG  …  #enddif  … |

1. 函数匹配

第一步：确定候选函数集(candidate function)。候选函数有两个特征：一是与被调用的函数同名，二是其声明在调用点可见。

第二步：考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些函数被称为可行函数(viable function)。可行函数也有两个特征：一是其形参数量与本次调用提供的实参数量相等，二是每个实参的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，则我们在调用该函数时传入的实参数量可能少于它实际使用的实参数量。

第三步：寻找最佳匹配(如果有的话)。逐一检查函数调用提供的实参，寻找形参类型与实参类型最佳匹配的可行函数，它的基本思想是：实参类型与形参类型越接近，它们匹配得越好。

**含有多个形参的函数匹配**：

如果有且仅有一个函数满足下列条件，则匹配成功：

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。

至少有一个实参的匹配由于其他可行函数提供的匹配。

如果在检查了所有实参之后没有任何一个函数脱颖而出，则该调用时错误的。编译器将报告二义性调用信息。

1. 类

默认情况下，编译器将为程序提供默认的**构造，拷贝，赋值，析构**函数。

1. 默认构造函数

C++11新标准中，如果我们需要默认的行为，那么可以通过在参数列表后面写上=default来要求编译器生成构造函数。

|  |
| --- |
| struct Sales\_data{  Sales\_data() = **default;**  Sales\_data(string &s, unsigned n ,double p):bookNo(s),units\_sold(n),revenue(p){}  string bookNo;  unsigned units\_sold = 0;  double revenue = 0.0;  }; |

上面的默认构造函数之所以对Sales\_data有效，是因为我们为内置类型的数据成员提供了初始值。

1. 访问控制和封装

使用class和struct定义类唯一的区别就是默认的访问权限，class是private，struct是public。

友元（friend）：

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成为它的友元。如果类想把一个函数作为它的友元，只需要增加一条以friend关键字开始的函数声明语句即可。

|  |
| --- |
| class Sales\_data  {  friend Sales\_data add(const Sales\_data&, const Sales\_data &);  friend std::istream &read(std::istream&, Sales\_data&);  friend std::ostream &print(std::ostream&, const Sales\_data&);  public:  Sales\_data() = defaule;  Sales\_data(string &s, unsigned n, double p):  bookNo(s),units\_sold(n),revenue(p\*n){}  private:  string bookNo;  unsigned units\_sold = 0;  double revenue = 0.0;  }； |

一般来说，最好在类定义开始或结束前的位置集中声明友元。

1. 可变数据成员(mutable)

有时候，我们希望能够修改类的某个数据成员，即使是在一个const成员函数内，可以通过在变量的声明中加入mutable关键字做到这一点。

一个可变数据成员（mutable data member）永远不会是const，即使它是const对象的成员。因此，一个const成员函数可以改变一个可变数据成员的值。

1. 构造函数

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，我们必须通过构造函数初始值列表为这些成员提供初始值。

1. 类成员初始化的顺序

成员的初始化顺序与它们在类定义中的出现顺序一致:第一个成员先被初始化，然后第二个，以此类推。构造函数初始值列表中初始值的前后位置关系不会影响实际的初始化顺序。

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。而且，如果可能的话，尽量避免使用某些成员初始化其他成员。

1. 委托构造函数

C++11新标准扩展了构造函数初始值的功能，使得我们可以定义所谓的委托构造函数(delegation constructor)。一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些(或全部)职责委托给了其他构造函数。

|  |
| --- |
| class Sales\_data{  …  public:  Sales\_data(string s, unsigned cnt, double price):  bookNo(s), units\_sold(cnt), revenue(cnt\*price) {}  //其余构造函数全部委托给另一个构造函数  Sales\_data() : Sales\_data(“”,0,0) {}  Sales\_data(string s): Sales\_data(s, 0, 0){}  Sales\_data(std::istream &is) : Sales\_data() { read(is, \*this); }  …  }; |

1. 隐式的类类型转换

如果构造函数只接受一个实参，则它实际上定义了转换为此类类型的隐式转换机制，我们有时将这种构造函数称作转换构造函数(converting constructor)。

能通过一个实参调用的构造函数定义了一条从构造函数的**参数类型**向类类型隐式转换的规则。

|  |
| --- |
| string nul\_book = “9-999-99999-9”;  //构造一个临时的Sales\_data对象：Sales\_data(nul\_book,0,0);  item.combine(nul\_book); |

但是，编译器只会自动地执行一步类型转换。下面的转换是错的：

|  |
| --- |
| //错误：需要用户定义两种转换：  //①、把“9-999-99999-9”转换成string  //②、再把这个临时的string转换成Sales\_data  item.combine(“9-999-99999-9”); |

1. 抑制构造函数定义的隐式转换

在要求隐式转换的程序上下文中，我们可以通过将构造函数声明为explicit加以阻止：

|  |
| --- |
| class Sales\_data{  …  public:  Sales\_data() = default;  Sales\_data(const string &s, unsigned n , double p):  bookNo(s), units\_sold(n), revenue(p\*n){}  explicit Sales\_data(const string &s): bookNo(s){}  explicit Sales\_data(std::istream &);  …  }; |

关键字explicit只对一个实参的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为explicit的。只能在类内部声明构造函数时使用explicit关键字，在类外部定义时不应重复。

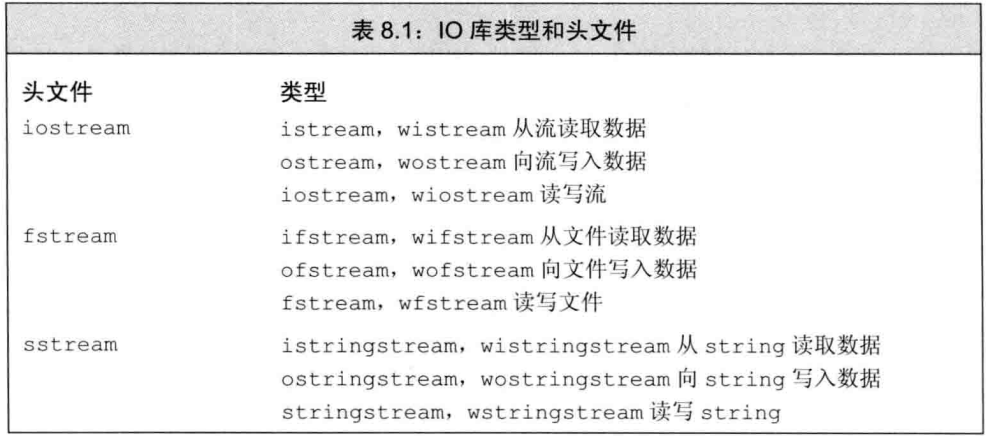
当我们使用explicit关键字声明构造函数时，它将只能以直接初始化的形式使用，而且，编译器将不会再自动转换过程中使用该构造函数。

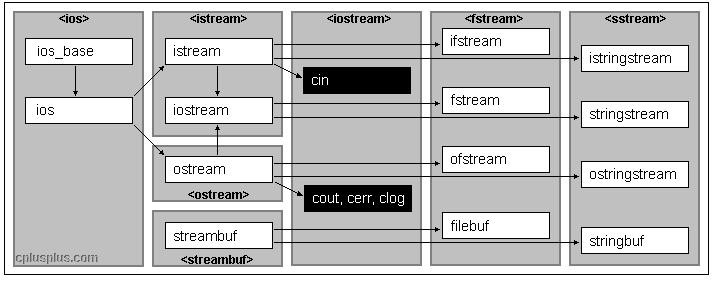
|  |
| --- |
| //正确：实参是一个显式构造的Sales\_data对象  Sales\_data item1(nul\_book);  //错误，不能讲explicit构造函数用于拷贝形式的初始化过程  Sales\_data item2 = nul\_book; |

1. 类的静态成员

和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名，static关键字则只能出现在类内部的声明语句中。

1. IO库





iostream定义了用于读写流的基本类型，fstream定义了读写命名文件的类型，sstream定义了读写内存string对象的类型。

为了支持使用宽字符的语言，标准库定义了一组类型和对象来操纵wchar\_t类型的数据，款字符版本的类型和函数的名字以一个w开始。即：

cin 🡪 wcin

cout 🡪 wcout

cerr 🡪 wcerr

istream 🡪 wistream

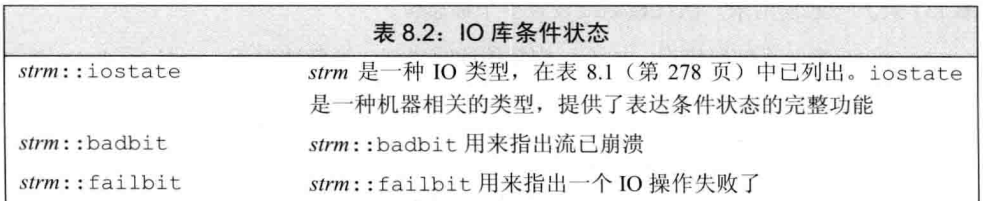
ostrem 🡪 wostream

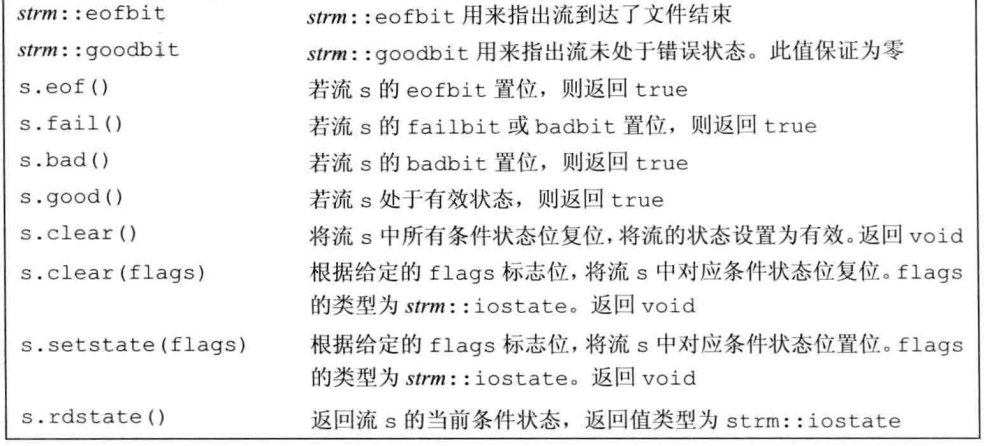
…

1. IO对象无拷贝或赋值

由于不能拷贝IO对象，因此我们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，因此传递和返回的引用不能是const的。

1. IO流的条件状态





一旦一个流发生错误，其后续的IO操作都会失败。只有当一个流处于无错误状态时，我们才可以从它读取数据，向它写入数据。在使用一个流之前，应该检查它是否处于良好的状态，最简单的方法是将它当做一个条件来使用：

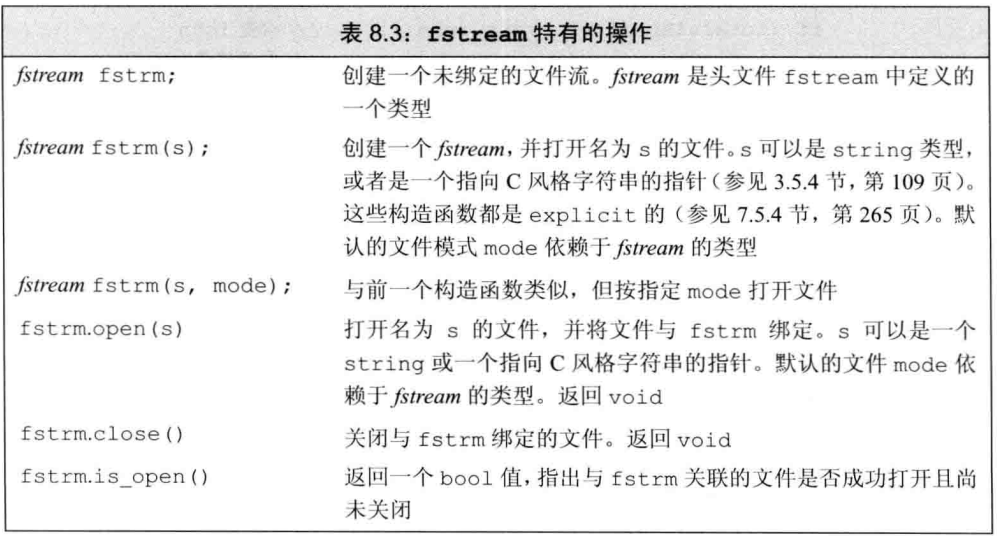
|  |
| --- |
| while(cin>>word)  //ok, 读取成功… |

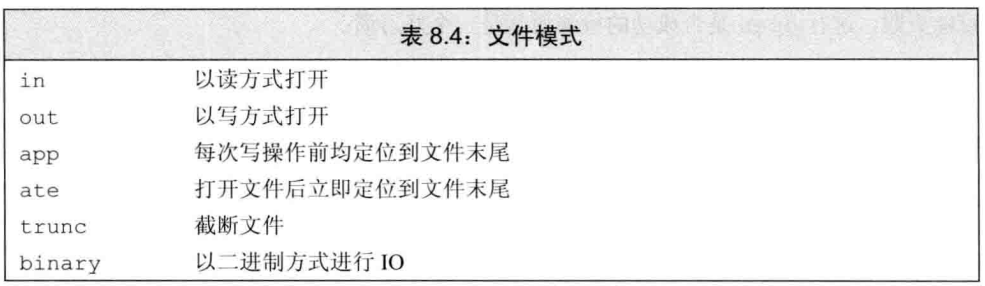
将流作为条件使用，只能告诉我们流是否有效，而无法告诉我们具体发送了什么。

1. 管理流的条件状态

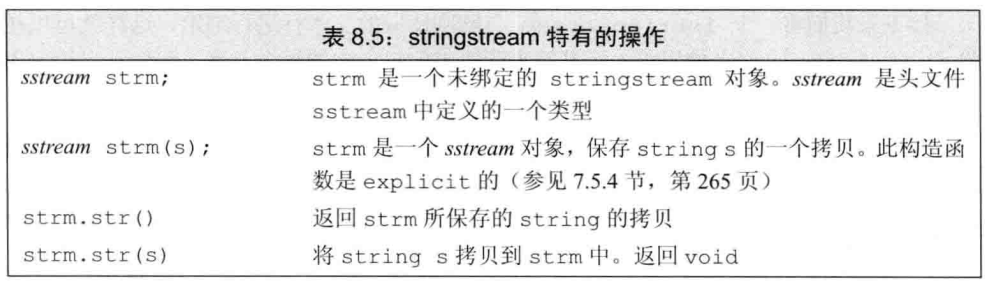
|  |
| --- |
| //记住cin的当前状态  auto odl\_state = cin.rdstate(); //记住cin的当前状态  cin.clear(); //清除所有错误标志位，使cin有效  process\_input(cin);  cin.setstate(old\_state); //将cin置为原来的状态  //复位failbit和badbit，保持其他标志位不变  cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit); |

1. 文件输入输出

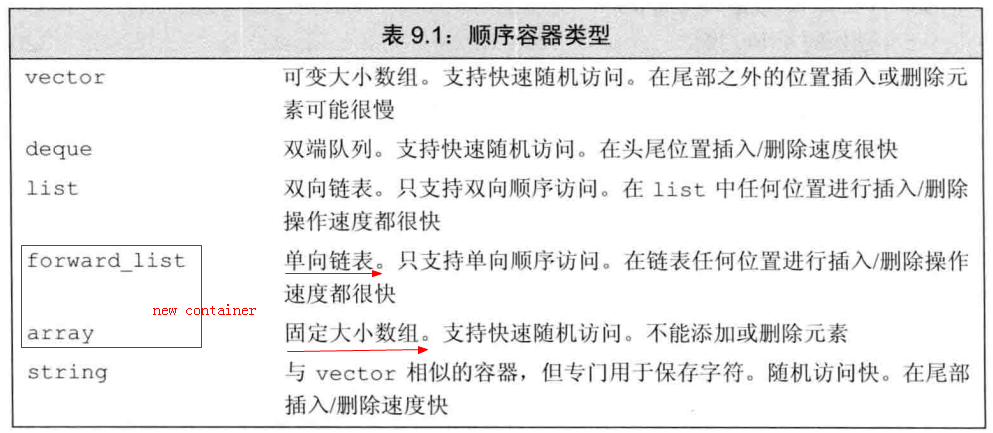




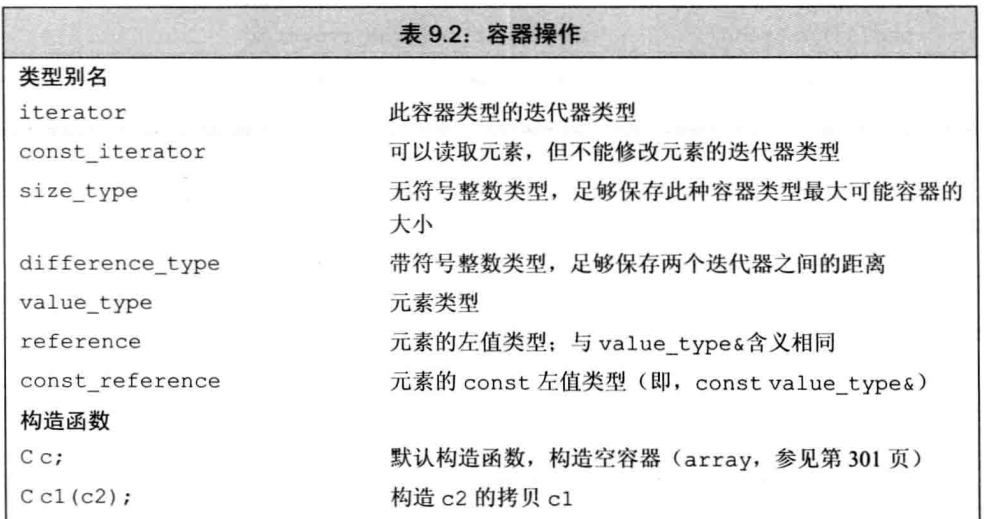
1. string流

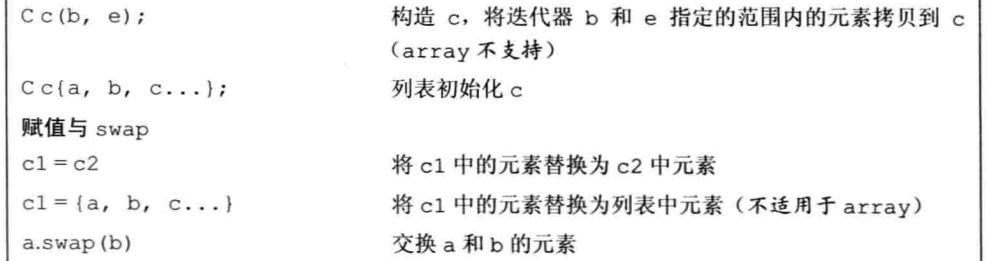


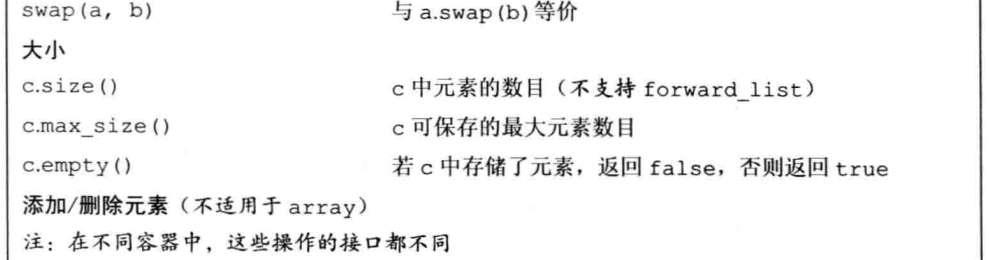
1. 顺序容器

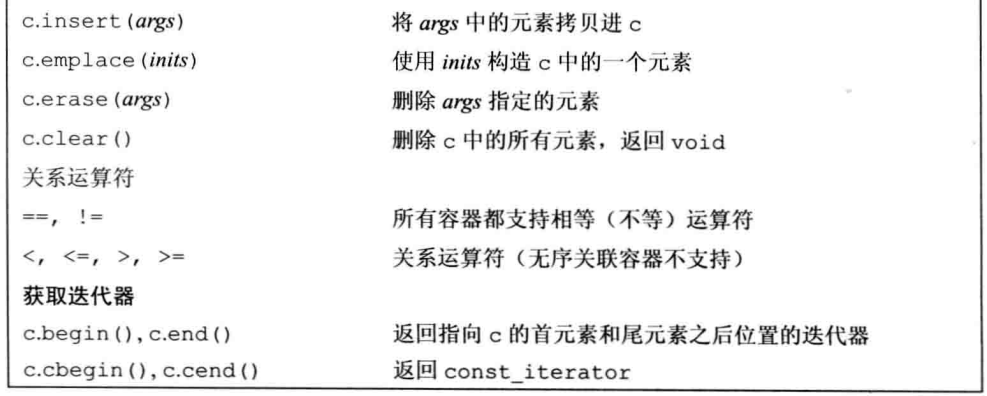


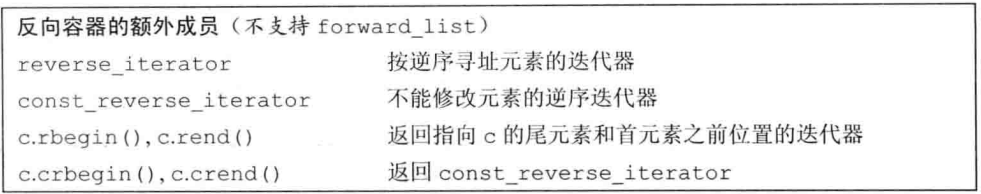
forward\_list和array是C++11新增加的类型，与内置数组相比，array是一种更安全，更容易使用的数组类型。forward\_list的设计目标是达到与最好的手写的单向链表数据结构相当的性能。因此，forward\_list没有size操作，因为保存或计算其大小就会比手写链表多出外的开销。



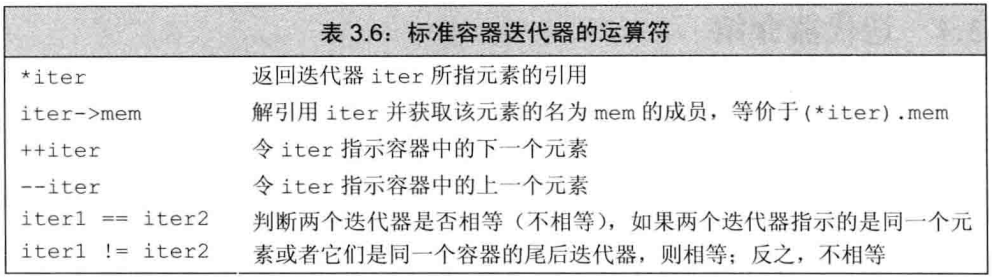




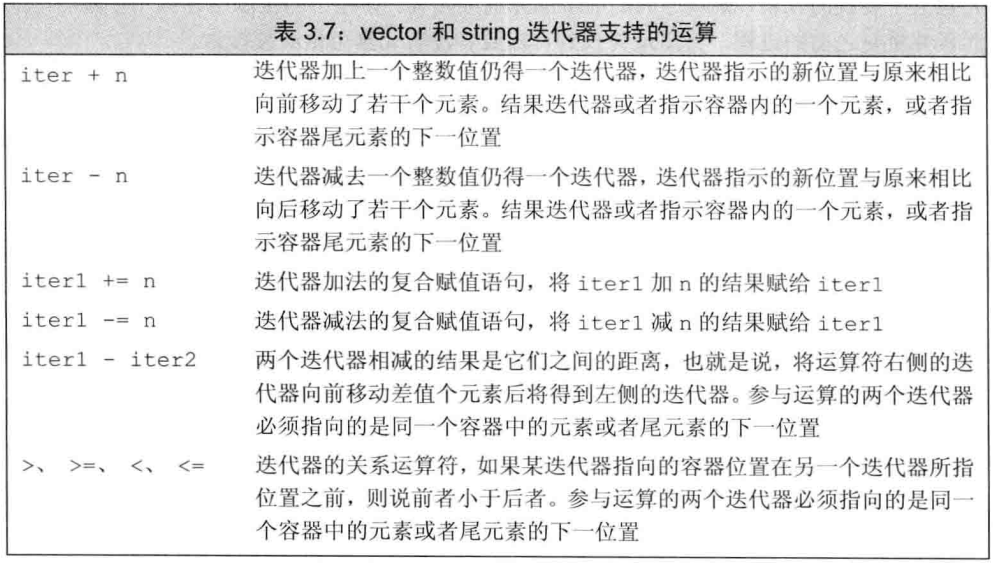




1. 迭代器



上表列出了所有容器迭代器支持的所有操作，其中，forward\_list迭代器不支持递减运算符(--)。



上表列出了迭代器支持的算术运算，这些运算只能应用于string、vector、deque和array的迭代器。

1. 容器定义和初始化