目录

第一章 入门知识 4

1.1 输入输出 4
1.2 标准命名空间 4
1.3 流作为条件 4
1.4 基本内置类型 4
1.5 带符号类型和无符号类型 4
1.6 类型转换 5
1.7 进制 5
1.8 转义序列 5
1.9 变量 6
1.10 变量声明和定义 7
1.11 命名规范 7
1.12 名字的作用域 7
1.13 引用 8
1.14 指针 8
1.15 复合类型的声明 10
1.16 const 限定符 12
1.17 类型处理 15
1.18 头文件 16
1.19 命名空间 using 17
1.20 数组 17
1.21 指针运算 20
1.22 与旧代码的接口 21
1.23 多维数组 21
1.24 数组别名 22

第二章 基础知识 23

2.1 简单语句 23

2.2 条件语句 ····· 23

第三章 真正的知识 ····· 25

第四章 现成的盛宴 ····· 26

4.1 标准库 String ····· 26

第五章 难啃的骨头 ····· 29

5.1 成员运算符的重载 ····· 29

5.2 OOP 概述 ····· 30

5.3 定义基类和派生类 ····· 31

5.4 派生类构造函数 ····· 33

5.5 继承与静态成员 ····· 34

5.6 派生类的声明 ····· 34

5.7 被用作基类的类 ····· 34

5.8 防止继承的发生 ····· 34

5.9 类型转换与继承 ····· 34

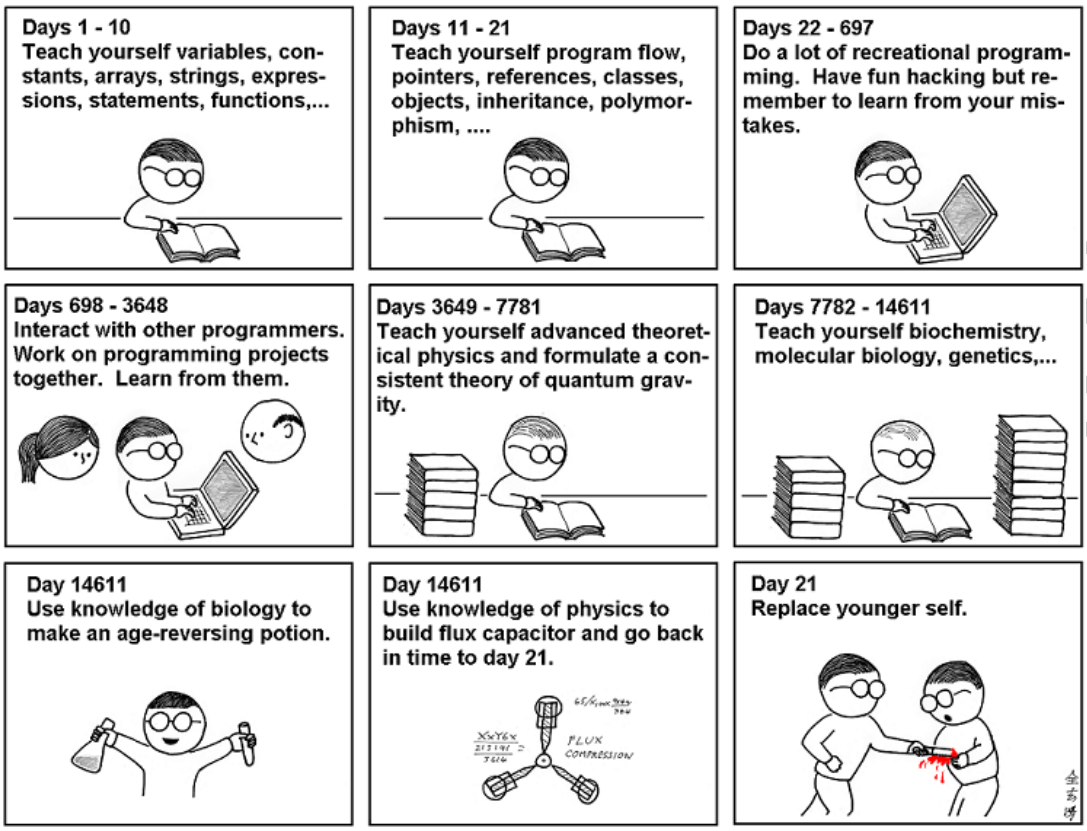
5.10 虚函数 ····· 34

5.11 抽象基类 ····· 34

5.12 访问控制与继承 ····· 36

第六章 高级货 ····· 41

第七章 实践经验 ····· 42



世界上只有两种语言，一种是没人用的，一种是被人骂的。

Bjarne Stroustrup

第 1 章　入门知识

1.1　输入输出

cin: 处理标准输入。
cout: 处理标准输出。
cerr: 输出警告和错误消息。
clog: 输出程序运行时的一般信息。

1.2　标准命名空间

标准库使用的所有名字都在命名空间 **std** 中。命名空间可以帮助我们避免不经意的名字定义冲突，以及使用库中相同名字导致的冲突。

1.3　流作为条件

当我们使用一个 **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 **EOF**，或遇到一个无效输入时，**istream** 对象的状态会变为无效。处于无效状态的 **istream** 对象会使条件为假。

1.4　基本内置类型

C++ 定义了一套包括**算术类型**和**空类型**在内的基本数据类型。其中算术类型包含了字符、整型数、布尔值和浮点型。空类型不对应具体的值，仅用于一些特殊的场合。

1.5　带符号类型和无符号类型

除去布尔值类型和扩展的字符类型之外，其他的整型可以分为**带符号的**和**无符号的**两种。带符号的类型可以表示**正数**、**负数**和 **0**。

1.6 类型转换

- 当我们把一个非布尔类型的算术值赋给布尔类型时，初始值为 0，则结果为 false，否则结果为 true。
- 当我们把一个布尔类型的值赋给非布尔类型时，初始值为 false，则结果为 0，否则结果为 1。
- 当我们把一个浮点值赋给整数时，采取的是截断处理，将仅保留小数点之前的值。
- 当我们把一个整数赋给浮点数时，小数部分为 0。此时要注意两者的所占空间的大小。
- 当我们赋给带符号类型一个超出它表示范围的值时，结果是**为定义的**。

1.7 进制

以 0 开头的整数代表八进制，以 0x 和 0X 开头的代表十六进制数。

1.8 转义序列

换行符	$\backslash n$
纵向制表符	$\backslash v$
反斜线	$\backslash\backslash$
回车符	$\backslash r$
横向制表符	$\backslash t$
退格符	$\backslash b$
问号	$\backslash ?$
进纸符	$\backslash f$
报警符	$\backslash a$
双引号	$\backslash ”$
单引号	$\backslash '$

表 1.1: C++ 转义序列

1.9　变量

变量提供了一个具名的、可供程序操作的存储空间。

1.9.1　变量定义

变量定义的基本形式是：首先是类型说明符，随后紧跟由一个或多个变量名组成的列表，其中变量名以逗号分隔，最后以分号结束。列表中每个变量名的类型都由类型说明符指定，定义时还可以为一个或多个变量赋初值。

1.9.2　初始值

当对象在创建时获得了一个特定的值，我们说这个对象被**初始化**了。初始化不是赋值，初始化的含义是创建变量时赋予一个初始值，而赋值的含义是把对象的当前值擦除，而以一个新值来替代。一个调用拷贝构造函数，一个调用 operator= 运算符来完成。

1.9.3　列表初始化

想要定义一个名为 x 的 int 变量，以下的四种方式都可以完成：

```
int x = 0;
int x = {0};
int x(0);
int x{0};
```

作为 C++11 新标准的一部分，用花括号来初始化变量得到了全面应用。无论是初始化对象还是某些时候为对象赋新值，都可以使用这样一组由花括号括起来的初始值了。当用于内置类型的变量时，这种初始化有一个重要特点：如果我们使用列表初始化且初始值存在丢失信息的风险，则编译器将报错。

1.9.4　默认初始化

如果定义变量时没有指定初值，则变量被默认初始化。默认值到底是什么由变量类型决定，同时变量定义的位置也会对此有影响。对于内置类型，定义在函数之外的变量被初始化为 0，定义在函数之内的变量未被初始化，未被初始化的变量其值是未定义的

1.10 变量声明和定义

C++ 语言支持**分离式编译**。为了支持分离式编译，C++ 语言将声明和定义区分开来。声明使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。定义负责创建与名字关联的实体，申请了存储空间。如果想声明一个变量而非定义它，就在变量名前添加关键字 **extern**，而且不要显示地初始化变量。变量能且只能被定义一次，但是可以被多次声明。

1.11 命名规范

变量命名有许多约定俗称的规范，下面的这些规范能有效提高程序的可读性。

- 标识符要能体现实际含义。
- 变量名一般小写字母。
- 用户自定义的类名一般以大写字母开头。
- 如果标识符由多个单词组成，则单词间应有明显的区分。采用驼峰式命名。

1.12 名字的作用域

变量的有效区域始于名字的声明语句，以声明语句所在的作用域末端为结束。作用域运算符为::。一般来说，在对象第一次被使用的地方附近定义它是一种好的选择，因为这样做有助于更容易地找到变量的定义。

1.12.1 作用域的嵌套

作用域可以彼此包含，被包含的作用域称为**内层作用域**，包含着的作用域被称为**外层作用域**。作用域一定那声明了某个名字，它所嵌套着的所有作用域中都可以梵文该名字。同时，允许在内层作用域中重新定义外层作用域已有的名字。

```
#include <iostream>
int reused = 42;
```

```
95  int main(int argc, char* argv[])
96  {
97    int unique = 0;
98    std::cout << reused << " " << unique << std::endl;
99    int reused = 0;
100   std::cout << reused << " " << unique << std::endl;
101   std::cout << ::reused << " " << unique << std::endl;
102   return 0;
103 }
```

当作用域运算符的左侧为空时，向全局作用域发出请求获取作用域操作符右侧名字对应的变量。

104

1.13　引用

105

106 **引用**为对象起了另外一个名字，引用类型引用另外一种类型。通过将声明
107 符写成 &x 的形式来引用类型。其中 x 是声明的变量名。一般在初始化变量
108 时，初始值会被拷贝到新建的对象中。然而定义引用时，程序把引用和它的初
109 始值绑定在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它
110 的初始值对象一直绑定在一起。因为无法令引用重新绑定到另外里格对象，因
111 此引用必须初始化。

引用并非对象，相反的，它只是为一个已经存在的对象所起的另外一个名字。因为引用本身不是一个对象，所以不能定义引用的引用。引用的类型都要和与之绑定的对象严格匹配，而且，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绑定在一起。

112

1.14　指针

113

114 指针是指向另外一种类型的复合对象。定义指针类型的方法是将声明符写
115 成 *d 的形式，其中 d 是变量名。

- 指针本身就是一个对象，允许对指针进行赋值和拷贝，而且在指针的生命周期内它可以先后指向几个不同的对象。
- 指针无须在定义时进行赋值。

指针存放某个对象的地址，要想获取该地址，需要使用**取地址符** &。

所有指针的类型都要和它所指向的对象严格匹配。(这里有例外)

1.14.1 指针值

指针值应属于下面四种之一：

* 指向一个对象。
* 指向紧邻对象所占空间的下一个位置。(迭代器哨兵)
* 空指针。
* 无效指针。

如果指针指向了一个对象，则允许使用**解引用** * 来访问对象。对指针解引用会得出所指的对象。

1.14.2 空指针

得到空指针最简单的形式就是使用字面值 **nullptr** 来初始化指针。当然也可以将指针的值初始化为字面值 0 来生成空指针。

预处理器中的 NULL 已经是过去式的东西了。

1.14.3 赋值和指针

指针和引用都能提供对其他对象的间接访问，然而在具体的实现细节上二者有很大的区别，其中最重要的一点就是引用本身不是一个对象。一旦定义了引用，就无法再将它绑定到其它的值上面，而指针却可以指向新的值。

有时候要想搞清楚一条语句到底是改变了指针的值还是改变了指针所值对象的值，最好的办法就是记住：

赋值永远改变的是等号左边的对象。

1.14.4　指针作为条件

只要指针拥有合法的值，就能将其用在条件表达式中。如果指针的值是 0，条件取 false。任何非 0 指针对应的条件值都是 true。

指针想等的状态可能对应三种：

- 都为空。
- 指向同一个对象。
- 都指向同一个对象的下一个地址。

1.14.5　void* 指针

void* 是一种特殊的指针类型，可用于存放任意对象的地址。不同的是，我们对该地址中到底是什么类型的对象并不了解。

利用 void* 指针能做的事比较有限：拿它和别的指针比较、作为函数的输入和输出，或者赋给另外一个 void* 指针，但是不能直接操作 void* 指针所指的对象。

1.15　复合类型的声明

定义的变量包括一个基本数据类型和一组声明符。在同一条定义语句中，虽然基本数据李诶笑嗯只有一个，但是声明符的形式却可以不同。

很多人容易迷惑基本数据类型和类型修饰符的关系，其实后者不过是声明符的一部分罢了。

经常有一种观点会误以为，在定义语句中，类型修饰符 * **或** & 作用于本次定义的全部变量。造成这种错误看法的原因有很多，其中之一就是我们把空格写在类型修饰符和变量名中间：

```
int* p;
```

涉及指针和引用的声明，一般有两种写法。

- 第一种把修饰符和变量标识符写在一起。
  int *p1, *p2;

- 第二种把修饰符和类型名写在一起，并且每条语句只定义一个变量。
  int* p;
  int* p2;

推荐采用第一种方式。

1.15.1 指向指针的指针

指针是内存中的对象，像其他的对象一样也有自己的地址，因此允许把指针的地址再存放到另一个指针当中。

1.15.2 指向指针的引用

引用本身不是一个对象，因此不能定义指向引用的指针。但指针是对象，所以存在对指针的引用。

```
int i = 42;
int *p;
int *&r = p;
```

要理解 r 的类型到底是什么，最简单的方式就是从右往左阅读 r 的定义。离变量名最近的符号（本例中 &r 的 &）对变量的类型有直接的影响，因此 r 是一个引用。声明符的其余部分用以确定 r 引用的类型是什么，此例中的符号 * 说明 r 引用的是一个指针。最后，声明的基本数据类型部分指出 r 引用的是一个 int 的指针。

1.16　const 限定符

有时我们希望定义这样一种变量，它的值不能被改变。为了满足这一要求，可以用关键字 **const** 对变量的类型加以限定。

```
const int bufSize = 1024;
```

因为 const 对象一旦创建后其值就不能再改变，所以 const 对象必须初始化。

1.16.1　初始化和 const

const 对象和非 const 对象的主要区别就是，只能在 const 类型的对象上执行不改变内容的操作。在不改变 const 对象的操作中还有一种是初始化，如果利用一个对象去初始化另外一个对象，则它们是不是 const 都无关紧要。如

```
int i = 123;
const int ci = i;
```

1.16.2　const 对象仅在文件内有效

编译器将在编译过程中把用到常量变量的地方都替换成相应的值。为了执行替换，编译器必须知道变量的初始值。如果程序包含多个文件，则每个用了 const 对象的文件都必须得到访问它的初始值才行。要做到这一点就必须在每一个用到变量的文件中都有它的定义。为了支持这一用法，同时避免对同一变量的重复定义。const 被限定为仅在文件内有效。

1.16.3　const 引用

可以把引用绑定到 const 对象上，就像绑定到其他对象上一样，我们称之为对常量的引用。与普通引用不同，对常量的引用不能被用作修改它所绑定的对象。

1.16.4　初始化和 const 的引用

引用的类型必须与其所引用对象的类型一致。但是有两个例外：

- 在初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果可以转换成引用的类型即可。

- 编译器生辰过的临时量。

1.16.5 指针和 const

和引用一样，也可以令指针指向常量或者非常量。类似于常量引用，指向常量的指针不能用于改变其所指的对象的值。

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所指对象的值。

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本生定义为常量。常量指针必须初始化，而且一但初始化完成，则它的值就不能再改变了。把 * 放在 const 关键字之前用以说明指针是一个常量。这样的书写形式隐含着一层意味，即不变的是指针本身的值而非指向的那个值。

```
int errNumb = 0;
int *const cerErr = &errNumb;
const double PI = 3.1415926;
const double *const pip = &pi;
```

要想弄清楚这些声明的含义，最行之有效的方法就是从右往左阅读。此例中，离 curErr 最近的符号是 const，意味着 curErr 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符中的下一个符号是 *，意味着 curerr 是一个常量指针。最后，该声明语句的基本数据类型部分确定了常量指针指向的是一个 int 对象。

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指向的对象的类型。

1.16.6 顶层和底层 const

指针本身是不是常量以及指针所指的是不是一个常量是两个相互独立的问题。用名词**顶层 const** 表示指针本身是个常量。用名字**底层 const** 表示指针所指的值是个常量。

更一般的，顶层 const 可以表示任意的对象是常量，这一点对任何数据类型都适用，如算术类型、类、指针等。底层 const 则与指针和引用等复合类型

的基本类型部分有关。比较特殊的是，指针类型既可以是顶层 const 也可以是底层 const，这一点和其它类型相比区别明显。

```
int i = 0;
int *const p1 = &i; // 不能改变p1的值，这是一个顶层const.
const int ci = 42; // 不能改变ci的值，这是一个顶层const.
const int *p2 = &ci; // 允许改变p2的值，这是一个底层const.
const int *const p3 = p2; // 靠右的const是顶层const，靠左的const是底层const.
const int &r = ci; // 用于声明引用的const都是底层const.
```

1.16.7 常量表达式和 constexpr

常量表达式是指**值不会改变并且在编译过程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。

1.16.8 constexpr 变量

在一个复杂系统中，很难分辨一个初始值到底是不是常量表达式。C++11 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时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; // 20是常量表达式
constexpr int limit = mf + 1; // mf + 1 是常量表达式
constexpr int sz = size(); // 只有当size是一个constexpr函数时才是一条正确的声明语句
```

一把来说，如果你认为变量是一个常量表达式，那就把它声明为 constexpr 类型。

1.17 类型处理

随着程序越来越复杂，程序中用到的类型也越来越复杂。这主要表现在：

- 一些类型难于拼写。
- 根本搞不清楚到底需要的类型是什么。

有两种方法可用于定义类型别名。传统的方法是使用关键字 **typedef**。

```
typedef double wages;
```

含有 typedef 的声明语句定义的不再是变量而是类型别名。
新标准规定了一种新的方法，使用**别名声明**来定义类型的别名。

```
using wages = double;
```

这种方法用关键字 using 作为别名声明的开始，其后紧跟别名和等号，其作用是把等号左侧的名字规定成等号右侧类型的别名。

1.17.1 auto 类型说明符

编程时常常需要把表达式的值赋给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而要做到这一点并非那么容易，有时甚至做不到。为了解决这个问题，C++ 新标准引入了 **auto** 类型说明符，用它就能让编译器替我们分析表达式所属的类型。显然，auto 定义的变量必须有初始值：

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

1.17.2 复合类型、常来和 auto

编译器推断出来的 auto 类型有时候和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

首先，正如我们所熟知的，使用引用其实是使用引用的对象，特别是当引用被用作初始值时，真正参与初始化的其实是引用对象的值。此时编译器以引用对象的类型作为 auto 的类型。

其次，auto 一般会忽略掉顶层 const，同时保留底层 const。如果我们希望推断出的 auto 类型是一个顶层 const，需要明确指出：

```
const auto f = ci; // ci的推演如果是int, f则是const int.
```

1.17.3　decltype 类型指示符

有时我们遇到这种情况：希望从表达式的类型推断出要定义的变量的类型。但是不想用该表达式的值初始化变量。为了满足这一要求，C++11 引入了第二种类型说明符 **decltype**，它的作用是选择并返回操作数的数据类型。在此过程中，编译器分析表达式并得到它的数据类型，却不实际计算表达式的值：

```
decltype(function()) sum = x; // sum的类型就是函数function的返回类型。
```

decltype 处理顶层 const 和引用的方式与 auto 有写不同。如果 decltype 使用的表达式是一个变量，则 decltype 返回该变量的类型 (包括顶层 const 和引用在内)：

```
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绑定到变量x
decltype(cj) z; // 错误
```

1.18　头文件

为了确保各个文件中类的定义一致，类通常定义在头文件中，而且类所在的头文件的名字应与类名一致，头文件通常包含那些只被定义一次的实体。

头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

C++ 程序还会用到的一项预处理功能是**头文件保护符**，头文件保护符依赖于预处理变量。预处理变量有两种状态：已定义和未定义。**#define** 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量

是否已经定义：**#ifdef** 当且仅当变量已定义时为真，**#ifndef** 当且仅当变量未定义时为真。一旦检测结果为真，则会执行后续操作直到遇到 **#endif** 指令为止。一般把预处理变量的名字全部大写。

1.19 命名空间 using

形如 using namespace::xxxx，表示编译器应从操作符左侧名字对应的作用域中寻找右侧那个名字。因此，XIAOHAI::addChain(par) 的意思就是要使用命名空间 XIAOHAI 中的名字 addChain 的函数。

1.19.1 每个名字都需要独立的 using 声明

每个 using 声明引入命名空间中的一个成员。例如：

```
using std::cin;
using std::cout;
using std::vector;
```

用到的每个名字都必须有自己的声明语句，而且每句话都以分号结束。

1.19.2 头文件不应包含 using 声明

位于头文件的代码，一般来说不应该使用 using 声明。如果使用了，反而可能由于不经意的包含了一些名字，造成可能产生始料未及的名字冲突。

1.20 数组

数组是存放类型相同的对象的容器，这些对象本身没有名字，只能通过位置进行访问。数组是一种复合类型。数组的声明形式为 a[d]，其中 a 是数组的名字，d 是数组的维度。维度说明了数组中元素的个数，因此必须大于 0，数组中元素的个数也属于数组类型的一部分，编译的时候维度必须是已知的。也就是说，维度必须是一个常量表达式 (constexpr)。

```
unsigned int cnt = 42; // 不是常量表达式。
constexpr unsigned sz = 42; // 是常量表达式
int arr[10]; // OK
int *parr[sz]; // 含有42个整型指针的数组
```

```
string bad[cnt]; // errooooooooooooooooor
```

和内置类型的变量一样，如果在函数内部定义了某种类型的数组，那么默认初始化会令数组含有未定义的值。

定义数组的时候必须指定数组的类型，不允许使用 auto 关键字由初始值的列表推断类型。

1.20.1　显式初始化数组的元素

可以对数组的元素进行列表初始化，此时允许忽略数组的维度。如果在声明时没有指明数组的维度，编译器会根据初始值的数量计算并推测出来。相反，如果指明了数组的维度，那么初始值的总数量不应该超出指定的大小。

1.20.2　字符数组的特殊性

字符数组有一种额外的初始化形式，我们可以用字符串字面值对此类数组进行初始化。当使用这种方式时，一定要注意字符串字面值的结尾处还有一个空字符。

1.20.3　不允许拷贝和赋值

不能将数组的内容拷贝给其它的数组作为初始值。

1.20.4　理解复杂的数组声明

和 vector 一样，数组能存放大多数类型的对象，例如，可以定义一个存放指针的数组。又因为数组本身就是对象，所以允许定义数组的指针和数组的引用。

```
int *ptrs[10]; /// ptrs是含有十个整型指针的数组。
int &refs[10]; /// 错误，不存在引用的数组。
int (*Parray)[10] = &arr; /// Parray指向一个含有10个整
    数的数组。
int (&arrRef)[10] = arr; /// arrRef引用一个含有10个整
    数的数组。
```

默认情况下，类型修饰符从右往左进行一次绑定。对于 Ptrs 来说，从右往左理解其含义比较简单：首先知道我们定义的是一个大小为 10 的数组，它的名字是 Ptrs，然后知道数组中存放的是指向 int 的指针。但是对于 Parray 来说，从右往左理解就不太合理了。因为数组的维度是紧跟着被声明的名字，所以就数组而言，由内向外阅读要比从右往左阅读好多了。由内向外阅读可以帮助我们更好的理解 Parray 的含义：首先是圆括号括起来的部分，*Parray 意味着 Parray 是个指针，接下来观察右边，可知道 Parray 是个指向大小为 10 的数组的指针，最后观察左边，知道数组中的元素是 int。这样最终的含义就明白无误了，Parray 是一个指针，它指向一个 int 数组，数组中包含 10 元素。同理，(&arrRef) 表示 arrRef 是一个引用，它引用的对象的是一个大小为 10 的数组，数组中的元素的类型是 int。

要理解数组声明的含义，最好的方式就是从数组的名字开始按照由内向外的顺序阅读。

1.20.5 访问数组元素

在使用数组下标的时候，通常将其定义为 **size_t** 类型。**size_t** 是一种机器无关的无符号类型，它被设计得足够大以便能表示内存中任意对象的大小。在 **cstddef** 头文件中定义了 size_t 类型。

与 vector 和 string 一样，当需要遍历数组的所有元素时，最好的办法也是使用范围 for 语句。

```
for(auto i : scores)
{
  std::cout << i << std::endl;
}
```

1.20.6 检查下标的值

数组的下标是否在合理的范围之内由程序员负责进行检查。

大多数常见的安全问题都源于缓冲区溢出错误。当数组或其他类似数据结构的下标越界并试图访问非法内存区域时，就会产生此类错误。

1.20.7　指针和数组

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。其中当使用数组作为一个 auto 变量的初始值时，推断得到的类型是指针而非数组。然而，必须指出的是，当使用关键字 decltype 时上述转换不会发生，decltype(ia) 返回的类型是同维度的数组。

1.20.8　指针也是迭代器

指向数组元素的指针拥有更多功能，vector 和 string 的迭代器支持的元素，数组的指针全部支持。就像使用迭代器遍历 vector 对象中的元素一样，使用指针也能遍历数组的元素。当然，这样做的前提是先得获取得到指向数组第一个元素的指针和指向数组尾元素的下一位置的指针。C++11 新标准引入了两个名为 begin() 和 end() 的函数，begin(ia) 返回指向 ia 首元素的指针，end(ia) 返回指向 ia 尾元素下一位置的指针，这两个函数定义在头文件 **iterator** 中。

1.21　指针运算

从 (给) 一个指针加上 (减去) 某个整数值，其结果仍是指针。新指针指向的元素与原来的指针相比前进了 (后退了) 该整数值个位置。

给指针加上一个整数，得到的新指针仍需要指向同一个数组的其他元素，或者指向同一数组的尾元素的下一位置。如果计算所得的指针超出了上述范围就将产生错误，而且这种错误编译器一般发现不了。

两个指针相减的结果的类型是一种名为 **ptrdiff_t** 的标准库类型，和 size_t 一样，ptrdiff_t 也是一种定义在 cstddef 头文件中的类型，因为差值可能为负值。所以 ptrdiff_t 是一种带符号的类型。

1.21.1 指针比较

只要两个指针指向同一个数组的元素，或者指向该数组的尾元素的下一个位置，就能利用关系运算符对其进行比较。例如：

```
int *b = arr;
int *e = arr + sz;
while(b < e)
{
  ++b;
}
```

1.22 与旧代码的接口

1.22.1 混用 string 对象和 C 风格字符串

更一般的情况是，任何出现字符串字面值的地方都可以用以空字符结束的字符数组来替代：

1. 允许使用空字符串结束的字符数组来初始化 string 对象或为 string 对象赋值。

2. 在 string 对象的加法运算中允许使用空字符串结束的字符数组作为其中一个运算对象；在 string 对象复合赋值运算中允许使用以空字符结束的字符数组作为右侧的运算对象。

1.22.2 string 转 C 风格字符串 c_str()

故名思议，c_str 函数的返回值是一个 C 风格的字符串。也就是说，函数的返回结果是一个指针，指针指向一个以空字符结束的字符数组。

1.23 多维数组

严格说，C++ 语言中没有多维数组，通常所说的多维数组其实就是数组的数组。

1.23.1　使用范围 fro 语句处理多维数组

```
size_t cnt = 0;
for(auto &row : ia)
{
  for(auto &col : row)
  {
    ++cnt;
    balabala.....
  }
}
```

要使用范围 for 语句处理多维数组，除了最内层的循环外，其他所有的循环的控制变量都应该是引用类型。

1.24　数组别名

第 2 章　基础知识

2.1　简单语句

表达式语句的作用是执行表达式并丢弃掉求值结果。

2.1.1　复合语句

复合语句是指花括号括起来的语句和声明的序列，复合语句也被称为块。一个块就是一个作用域，在块中引入的名字只能在块内部以及嵌套在块内的子块中进行使用。通常，名字在有限的作用域中可见，该区域从名字定义处开始，到名字所在的块的结束为止。

2.1.2　语句作用域

可以在 if、switch、while 和 for 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用域了。

2.2　条件语句

C++ 提供了两种条件执行的语句。一种是 if 语句，它根据条件决定控制流。另一种是 switch 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。

2.2.1　if 语句

if 语句的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。if 语句包含两种形式：一种包含 else 分支，另外一种没有。常见的语法形式为：

```
if(condition)
```

```
462     {
463       statement
464     }
```

465 **if else 语句的形式是**

```
466     if(condition)
467     {
468       statement;
469     }
470     else
471     {
472       statement2;
473     }
```

474 条件检测部分必须用括号括起来。

第3章　真正的知识

第 4 章　现成的盛宴

4.1　标准库 String

标准库类型 string 表示可变长的字符序列，使用 string 类型必须首先包含 string 头文件，且打开 std 命名空间，因为 string 位于 std 命名空间中。

```
#include <string>
using std::string;
```

初始化 string 对象的方式	
stirng s1	默认初始化，s1 是空串。
string s2(s1)	拷贝初始化。
sting s2 = s1	拷贝初始化。
string s3(“value”)	s3 是字面值的副本，除了字面值最后的那个空字符串。
stirng s3 = “value”	等价于上面的那种初始化方法
sting s4(n, ‘c’)	把 s4 初始化为由连续 n 个字符 c 组成的串。

4.1.1　string 支持的操作

4.1.2　读取 string 对象

在执行读取操作时，string 对象会自忽略开头的空白并从第一个真正的字符开始读起，直到遇到下一个空白为止。

4.1.3　使用 getline 读取一整行

getline 函数的参数是一个输入流和一个 string 对象，函数从给定的输入流中读取内容，直到遇到换行符为止 (注意换行符也被读了进来)，然后把所读的内容存入到那个 string 对象中去 (注意不存在符)。和输入运算符一样，getline 也会返回它的流参数。

string 支持的操作	
os << s	输出
is >> s	读取
getline(is, s)	从 is 中读取一行赋给 s，并返回 is。
s.empty()	判断是否为空
s.size()	返回 s 中字符的个数
s[n]	返回 s 中第 n 个字符的引用，从 0 开始计数。
s1+s2	
s1=s2	
s1==s2	
s1!=s2	
<, <=, >, >=	

出发 getline 函数返回的那个换行符实际上被丢掉了，得到的 string 对象中并不包含该换行符。

4.1.4 string::size_type 类型

string 的 size() 函数返回的是一个 string::size_type 类型的值，这是 string 的一个配套类型，类似下面的情状（自己歪歪的，而且明显知道自己歪歪的不对）：

```
class string
{
  typedef unsigned int size_type;
};
```

4.1.5 string 的加法

两个 string 对象相加得到一个新的 string 对象，其内容是把左侧的运算对象与右侧的运算对象串接而成。

4.1.6　字面值和 string 对象相加

当把 string 对象和字符字面值及字符串字面值混在一条语句中使用时，必须确保每个加法运算 (+) 的两侧的运算对象至少有一个是 string，这里包括加法的左结合性。

4.1.7　范围 for 语句

如果想对 string 对象中的每个字符做点儿什么操作，目前最好的办法就是使用**范围 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。

```
for(declaration : expression)
{
  statement;
}
```

其中 expression 表示一个**序列**。declaration 部分负责定义一个变量，该变量将被用于访问序列中的每个元素。每次迭代，declaration 部分的变量会被初始化为 expression 部分的下一个元素值。一般情况下，序列中的每个原子元素会被拷贝到 declaration。如果想要改变序列中的原子元素，则必须把循环变量定义成引用类型。

4.1.8　只改变一部分值

要想访问 string 对象的单个字符有两种方式：

- 使用下标。
- 使用迭代器。

下标运算符 [ ]接收的输入参数是 string::size_type 类型的值，这个参数表示要访问的字符的位置。返回值是该位置上字符的引用。

任何表达式只要它的值是一个整型值就能作为索引。使用超出索引范围的下标操作将引发不可预知的结果。

第 5 章　难啃的骨头

5.1　成员运算符的重载

在迭代器类及智能指针类中常常用到解引用运算符 (*) 和箭头运算符 (− >)。我们以如下形式向 StrBlobPtr 类添加这两种运算符:

```
class StrBlobPtr
{
  public:
  std::string& operator*() const
  {
    auto p = check(curr, "dereference past end");
    return (*p)[curr];
  }
  std::string* operator->() const
  {
    return & this->operator*();
  }
};
```

解引用运算符返回所指元素的一个引用。箭头运算符不执行任何自己的操作，而是调用解引用运算符并返回解引用结果元素的地址。

箭头运算符必须是类的成员。解引用运算符通常也是类的成员，尽管并非必须如此。

值得注意的是，我们将这两个运算符定义成了 const 成员，这是因为获取一个元素并不会改变 StrBlobPtr 对象的状态。同时，它们的返回值分别是非常量 string 的引用和指针，因为 StrBlobPtr 只能绑定到非常量的 StrBlobPtr 对象。

5.1.1　对箭头运算符返回值的限定

和大多数其它运算符一样，我们能令 operator* 完成任何我们指定的操作。换句话说，我们可以让 operator* 返回一个固定值 42，或者打印对象的内容，或者其它。箭头运算符则不是这样，它永远不能丢掉成员访问这个最基本的含义。当我们重载箭头时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不同。

对于形如 $point->mem$ 的表达式来说，point 必须是指向类对象的指针或者是一个重载了 operator-> 的类的对象。根据 point 类型的不同，$point->mem$ 分别等价于

```
(*point).mem; // point是一个内置的指针类型
point.operator()->mem;  // point是类的一个对象
```

除此之外，代码将发生错误。point->mem 的执行过程如下所是：

- 如果 point 是指针，则我们应用内置的箭头运算符，表达式等价于 (*point).mem。首先解引用该指针，然后从所得的对象中获取指定的成员。如果 point 所指的类型没有名为 mem 的成员，程序会发生错误。
- 如果 point 是定义了 operator 的类的一个对象，则我们使用 $point.operator->()$ 的结果来获取 mem。其中，如果该结果是一个指针，则执行第一步。如果该结果本身含有重载的 $operator->()$，则重复调用当前步骤。最终，当这一过程结束时程序或者返回了所需要的内容，或者返回了一些表示程序错误的信息。

重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对像。

5.2　OOP 概述

面向对象程序设计 (object-oriented programming) 的核心思想是**数据抽象、继承和动态绑定**。

1. 通过使用数据抽象，我们可以使接口与实现分离。
2. 使用继承可以定义相似的类型并对其相似关系建模。

3. 通过使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

5.2.1 继承

通过继承联系在一起的类构成一种层次关系。通常在层次关系的根部由一个基类，其他类则直接或间接地从基类继承而来，这些继承得到的类称为派生类。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

在 C++ 语言中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。对于某些函数，基类希望派生类各自定义适合自身的版本，此时基类就将这些函数声明称虚函数。

派生类必须在其内部对所有的重新定义的虚函数进行声明，派生类可以在这样的函数之前加上关键字 virtual，但是并不是必须这样做的。C++11 规定允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施是在该函数的形参列表之后增加一个 override 关键字 (在对象的 const 属性之后)。

5.2.2 动态绑定

在 C++ 语言中，当我们使用基类的引用或指针调用一个虚函数时将发生动态绑定。

5.3 定义基类和派生类

5.3.1 定义基类

```
class Quote{
  public:
  Quote() = default;
  Quote(const std::string &book, double sales_price) :
    bookNo(book), price(sales_price){ }
```

```
597     std::string isbn() const
598     {
599       return bookNo;
600     }
601     virtual double net_price(std::size_t n) const
602     {
603       return (n * price);
604     }
605     virtual ~Quote() = default;
606
607     private:
608       std::string bookNo;
609
610     protected:
611       double price = 0.;
612   };
```

基类通常都应该定义一个徐析构函数，即使该函数不执行任何实际操作也是如此。

613

5.3.1.1 成员函数与继承

614

615 在 C++ 语言中，基类必须将他的两种成员函数区分对待：

616 • 一种是基类希望派生类进行覆盖的函数。

617 • 另外一种是基类希望派生类直接继承而不要改变的函数。

618 对于前者，基类通常将其定义为虚函数。基类通过在其成员函数的声明语句之
619 前将上关键字 virtual 使得该函数执行动态绑定。任何构造函数之外的非静态
620 函数都可以是虚函数。关键字 virtual 只能出现在类内部的声明语句之前而不
621 能出现用于类外部的函数定义。如果基类把一个函数声明成虚函数，则该函数
622 在派生类中隐式地也是虚函数。

5.3.1.2 访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有访问权限访问从基类继承而来的成员。和其他使用基类的代码一样，派生类能访问公有成员，而不能访问私有成员。不过在某些时候基类中还有这样一种成员，基类希望它的派生类有权限访问该成员，同时禁止其他用户访问。我们用**受保护的**访问运算符说明这样的成员。

5.4 派生类构造函数

尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也需要使用基类的构造函数来初始化它的基类部分。

每个类控制它自己的成员如何进行初始化。

派生类对象的基类部分与派生类对象自己的数据成员都是在构造函数的初始化阶段执行初始化操作的。类似于我们初始化成员的过程，派生类构造函数同样是通过初始化列表来将实参传递给基类的构造函数的。例如：

```
Bulk_quote(const std::string &book, double p, std::
    size_t qty, double disc) :
  Quote(book, p), min_qty(qty), discount(disc) { }
```

首先初始化基类的部分，然后按照声明的顺序依次来初始化派生类的成员。

5.5　继承与静态成员

5.6　派生类的声明

5.7　被用作基类的类

5.8　防止继承的发生

5.9　类型转换与继承

5.9.1　静态类型与动态类型

5.9.2　不存在从基类向派生类的隐式类型转换

5.9.3　在对象之间不存在类型转换

5.10　虚函数

5.10.1　对需要函数的调用只能在运行时才被解析

5.10.2　派生类中的虚函数

5.10.3　find 和 override 说明符

5.10.4　虚函数与默认实参

5.10.5　回避虚函数机制

5.11　抽象基类

5.11.1　纯虚函数

和普通的虚函数不一样，一个纯虚函数无需定义。我们通过在函数体的位置（即在声明语句的分号之前）书写 =0 就可以将一个虚函数说明为纯虚函数了。其中，=0 只能出现在类内部的虚函数声明语句处:

```
660  class Quote{
661    public:
662    Quote() = default;
663    Quote(const std::string &book, double sales_price) :
664    bookNo(book), price(sales_price) { }
665    virtual ~Quote() = default;
666    std::string isbn() const {
667      return bookNo;
668    }
669
670    virtual double net_price(std::size_t n) const{
671      return n * price;
672    }
673
674    private:
675    std::string bookNo;
676    protected:
677    double price = 0.0;
678  };
679
680  class Disc_quote : public Quote{
681    public:
682    Disc_quote() = default;
683    Disc_quote(const std::string &book, double price,
684       std::size_t qty, double disc)
685    : Quote(book, price), quantity(qty), discount(disc)
686       { }
687    double net_price(std::size_t) const = 0;
688
689    protected:
690    std::size_t quantity = 0;
691    double discount = 0.0;
```

```
};
```

和我们之前定义的 Bulk_quote 类一样，Disc_quote 也分别定义了一个默认构造函数和一个接受四个参数的构造函数。尽管我们不能定义这个类的对象，但是 Disc_quote 的派生类构造函数将会使用 Disc_quote 的构造函数来构建各个派生类对象的 Disc_quote 部分。其中，接受四个参数的构造函数将前两个参数传递给 Quote 的构造函数，然后直接初始化自己的成员 discount 和 quantity。默认构造函数则对这些成员进行默认初始化。

值的注意的是，我们也可以为纯虚函数提供定义，不过函数体必须定义在类的外部。也就是说，我们不能在类的内部为一个 =0 的函数提供函数体。

5.11.2　含有纯虚函数的类是抽象基类

含有纯虚函数的类是**抽象基类**。抽象基类负责定义接口，而后续的其他类可以覆盖该接口。我们不能创建一个抽象基类的对象。

我们不能创建抽象基类的对象。

5.11.3　派生类构造函数只初始化它的直接基类

每个类各自控制其对象的初始化过程，派生类构造函数只初始化它的直接基类。

5.12　访问控制与继承

每个类分别控制自己的成员初始化过程。与之类似，每个类还控制着其成员对于派生类来说是否可访问。

5.12.1　受保护的成员

1. 和私有成员类似，受保护的成员对于类的用户来说是不可访问的。

2. 和共有成员类似，受保护的成员对于派生类的成员和友元来说是可以访问的。

3. 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象的受保护成员没有任何访问权限。

例如：

```
class Base{
  protected:
  int prot_mem;
};

class Sneaky : public Base{
  friend void clobber(Sneaky&);
  friend void clobber(Base&);
  int j;
};
/// 正确
void clobber(Sneaky &s) { s.j = s.prot_mem = 0;}
/// 错误
void clobber(Base &b) { b.prot_mem = 0;}
```

派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员，对于普通的基类对象中的成员不具有特殊的访问权限。

5.12.2 公有、私有和受保护继承

某个类对其继承而来的成员的访问权限受到两个因素影响：

- 基类中该成员的访问说明符。
- 在派生类的派生列表中的访问说明符。

```
736   class Base{
737     public:
738         void pub_mem();
739     protected:
740         int prot_mem;
741     private:
742         char priv_mem;
743   };
744
745   struct Pub_Derv : public Base{
746     /// 正确，派生类能访问受保护的成员。
747     int f() { return prot_mem;}
748     /// 错误，私有成员对于派生类而言是不可访问的。
749     char g() {return priv_meme;}
750   };
751
752   struct Priv_Derv : private Base{
753     /// 正确，private不影响派生类的访问权限。
754     int f1() const { return prot_mem;}
755   };
```

派生访问说明符对于派生类的成员能否访问其直接基类的成员没什么影响。对基类成员的访问权限只与基类中的访问说明符有关。Pub_Derv 和 Priv_Derv 都能访问受保护的成员 prot_mem，同时它们都不能访问私有成员 priv_mem。

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

```
Pub_Derv d1; /// 继承自Base的成员是public的
Priv_Derv d2; /// 继承自Base的成员是private的
d1.pub_mem(); /// 正确
d2.pub_mem(); /// 错误，pub_mem在派生类中是private的
```

Pub_Derv 和 Priv_Derv 都继承了 pub_mem 函数。如果继承是公有的，则成员将遵循其原有的访问说明符，此时 d1 可以调用 pub_mem。在 Priv_Derv

中，Base 的成员是私有的，因此类的用户不能调用 pub_mem。

派生访问说明符还可以控制继承自派生类的新类的访问权限：

```
struct Derived_from_Public : public Pub_Derv{
  /// 正确，prot_mem在Pub_Derv中仍是受保护的。
  int use_base() {return prot_mem;}
};
struct Derived_from_Private : public Priv_Derv{
  /// 错误
  int use_base() {return prot_mem;}
};
```

Pub_Derv 的派生类之所以能访问 Base 的 prot_mem 成员是因为该成员在 Pub_Derv 中仍然是受保护的。相反，Priv_Derv 的派生类无法执行类的访问，对于它们来说，Priv_Derv 继承自 Base 的所有成员都是私有的。

假设我们之前还定义了一个名为 Prot_Derv 的类，它采用受保护继承，则 Base 的所有公有成员在新定义的类中都是受保护的。Prot_Derv 的用户不能访问 pub_mem，但是 Prot_Derv 的成员和友元可以访问那些继承而来的成员。

5.12.3 派生类向基类转换的可访问性

派生类向基类转换是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响。假定 D 继承自 B：

1. 只有当 D 公有地继承 B 时，用户代码才能使用派生类向基类的转换；如果 D 继承 B 的方式是保护的或者私有的，则用户代码不能使用该转换。

2. 不论 D 以什么方式继承 B，D 的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。

3. 如果 D 继承 B 的方式是公有的或者受保护的，则 D 的派生类的成员和友元可以使用 D 向 B 的类型转换，反之，如果 D 继承 B 的方式是私有的，则不能使用。

不考虑继承的话，我们可以认为一个类有两种不同的用户：普通用户和类的实现着。其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有成员；实现着则负责编写类的成员和友元的代码，成员和友元即能访问类的公有部分，也能访问类的私有实现部分。如果进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的那部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及其基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

795

第6章　高级货

第 7 章　实践经验