C++ 学习笔记

于程洋

2021 年 7 月 19 日

目 录

第1章 复合类型

1.1 指针、数组

用于分配内存的方法，C++有3种管理数据内存的方式

1 自动存储在函数内部定义的常规变量使用自动存储空间，被称为自动变量，这意味着它们在所属的函数被调用时自动产生，在该函数结束时消亡。

实际上，自动变量是一个局部变量，其作用域为包含它的代码块。代码块是被包含在花括号中的一段代码。自动变量通常存储在栈中。这意味着执行代码块时，其中的变量将依次加入到栈中，而在离开代码块时，将按相反的顺序释放这些变量，这被称为后进先出（LIFO）。因此，在程序执行过程中，栈将不断地增大和缩小。

2 静态存储静态存储是整个程序执行期间都存在地存储方式。使变量成为静态的方式有两种：

- 在函数外面定义它
- 在声明变量中使用关键字 static。如

```
1                    static double fee = 56.50;
```

在 K&R C 中只能初始化静态数组和静态结构，而 C++ Release 2.0 和 ANSI C 中，也可以初始化自动数组和自动结构。

3 动态存储 new 和 delete 运算符提供了一种比自动变量和静态变量更灵活的方法。它们管理了一个内存池，这在 C++ 中被称为自由存储空间（free store）或堆（heap）。该内存池同用于静态变量和自动变量的内存是分开的。new 和 delete 让你能够在一个函数中分配内存，在另一个函数中释放它。因此，数据的生命周期不完全受程序或函数的生存时间控制。与使用常规变量相比，使用 new 和 delete 让程序员对程序如何使用内存有更大的控制权。然而，内存管理也更复杂。在栈中，自动添加和删除机制使得占用的内存总是连续的，但 new 和 delete 的相互影响可能导致占用的自由存储区不连续，这使得跟踪新分配内存的位置更困难。

栈、堆和内存泄露

如果使用 new 运算符在自由存储空间（或堆）上创建变量后，没有调用 delete，即使包含指针的内存由于作用域规则和对象生命周期的原因而被释放，在自由存储空间上动态分配的变量或结构也将继续存在。实际上，将会无法访问自由存储空间中的结构，因为指向这些内存的指针无效。这将导致内存泄漏。被泄漏的内存将在程序的整个生命周期内都不可使用；这些内存被分配出去，但无法收回。极端情况（不过不常见）是，内存泄漏可能会非常严重，以致于应用程序可用的内存被耗尽，出现内存耗尽错误，导致程序崩溃。另外，这种泄漏还会给一些操作系统或在相同的内存空间中运行的应用程序带来负面影响，导致它们崩溃。

要避免内存泄漏，最好是同时使用 new 和 delete 运算符，在自由存储空间上动态分配内存，随后便释放它。

1.2 数组的替代品

模板类 vector 和 array 是数组的替代品。

1.2.1 模板类 vector

模板类 vector 类似于 string 类，也是一种动态数组。可以在运行阶段设置 vector 对象的长度，可在末尾附加新数据，还可在中间插入新数据。基本上，它是使用 new 创建动态数组的替代品。vector 类使用 new 和 delete 来管理内存，但这种工作是自动完成的。

要使用 vector 对象，必须包含头文件 vector。其次，vector 包含在名称空间 std 中。模板使用不同的语法来指出它存储的数据类型。vector 类使用不同的语法来指定元素数。

```
#include <vector>
using namespace std;
vector<int> vi;      //create a zero-size array of int
int n;
cin >> n;
vector<double> vd(n);  //create an array of n doubles
// vector<typename> variable(n_elem);
```

1.2.2 模板类 array (C++11)

vector 类的功能比数组强大，但付出的代价是效率稍低。如果需要的是固定长度的数组，使用数组是更佳的选择，但代价是不那么方便和安全。C++11 新增了模板类 array，它也位于名称空间 std 中。与数组一样，array 对象的长度也是固定的，也使用栈（静态内存分配），而不是自由存储区，因此其效率和数组相同，但更方便，更安全。

```
#include <array>
using namespace std;
array<int, 5> ai; //create array object of 5 ints
array<double, 4> ad = {1.2, 2.1, 3.43, 4.3};
// array<typename, n_elem> variable;
```

在 C++11 中，可将列表初始化用于 vector 和 array 对象，但在 C++98 中，不能堆 vector 对象这样做。

第2章 循环

2.1 for 循环

for 循环为执行重复的操作提供了循环渐进的步骤。

1. 设置初始值
2. 执行测试，看看循环是否应当继续进行
3. 执行循环操作
4. 更新用于测试的值

```
for(initialization; test-expression; update-expression)
        body
```

2.1.1 副作用和顺序点

C++ 就递增运算符生效做了一些规定。首先，副作用（side effect）指的是在计算表达式时对某些东西（如存储在变量中的值）进行了修改；顺序点（sequence point）是程序执行过程中的一个点，在这里，进入下一步之前将确保对所有的副作用都进行了评估。在 C++ 中，语句的分号是一个顺序点，这意味着程序处理下一条语句之前，赋值运算符、递增运算符和递减运算符执行的所有修改都必须完成。任何完整的表达式末尾都是一个顺序点。

```
while (guest++ < 10)
        cout << guest << endl;
```

这里表达式 guest++ ¡ 10 是一个完整表达式，C++ 将确保副作用（将 guest 加 1）在程序进入 cout 之前完成。而下面的语句中

```
y = (4 + x++) + (6 + x++);
```

表达式 4 + x++ 不是一个完整的表达式，因此，C++ 不保证 x 的值在计算子表达式 4 + x++ 后立刻增加 1。在这个例子中，整条赋值语句是一个完整的表达式，而分号标注了顺序点。C++ 没有规定是在计算每个子表达式之后将 x 的值递增，还是在整个表达式计算完毕后才将 x 的值递增，所以应避免使用。

2.1.2 前缀格式和后缀格式

选择前缀格式还是使用后缀格式可能对程序的行为没有影响（for 循环中的条件语句），但是执行速度可能有细微的差别。对于内置类型和当代的编译器而言，这可能不是什么问题。然而，C++ 允许您针对类定义这些运算符，在这种情况下，用户这样定义前缀函数：将值加 1，然后返回结果；但后缀版本首先复制一个副本，将其加 1，然后将复制的副本返回。因此，对于类而言，前缀版本的效率比后缀版本高。

2.1.3 递增/递减运算符和指针

可将递增运算符用于指针和基本变量。将递增运算符用于指针时，将把指针的值增加其值相的数据类型占用的字节数，这种规则适用于对指针递增和递减

```
1          double arr[5] = {21.1, 11.2, 23.4, 23.1, 43.2};
2          double* pt = arr;  //pt points to arr[0]
3          ++pt;              //pt points to arr[1]
```

也可以结合使用这些运算符和 * 运算符来修改指针指向的值。前缀递增、前缀递减和解除引用运算符的优先级相同，以从右到左的方式进行结合。后缀递增和后缀递减的优先级相同，但比前缀运算符的优先级高，这两个运算符以从左到右的方式进行结合

```
1          double x = *++pt;  //increment pointer, pt points to arr[2]
2          ++*pt;
3  //increment the pointed to value, change 23.4 to 24.4
4          (*pt)++;   //increment pointed-to value
5          x = *pt++;
6  //dereference original location, then increment pointer
```

后缀运算符++的优先级更高，这意味着将运算符用于 pt，而不是 *pt，因此对指针递增。然后后缀运算符意味着将对原来的地址（&arr[2]）而不是递增后的新地址解除引用，因此 *pt++ 的值为 arr[2]，但该语句执行完毕后，pt 的值将为 arr[3] 的地址。

复合语句有一种有趣的特性。如果在语句块中定义一个新的变量，则仅当程序执行该语句块中的语句时，该变量才存在。执行完该语句块后，变量将被释放，即此变量仅在该语句块中才是可用的

在使用语句块时，如果在外部语句块中声明一个变量，而外部语句块中也有一个这种名称的变量，在声明位置到内部语句块结束的范围之内，新变量将隐藏旧变量；然后旧变量再次可见。

```
#include<iostream>
int main()
{
        using std::cout;
        using std::endl;
        int x = 20;     //original x
        {
                cout << x << endl;  //use original x
                int x = 100;        // new x
                cout << x << endl;  // use new x
        }
        cout << x << endl;     // use original x
        return 0;
}
```

2.2 while 循环

while 循环没有初始化和更新部分的 for 循环，它只有测试条件和循环体：

```
        while (test-condition)
                body
```

2.2.1 类型别名

C++ 为类型建立别名的方式有两种。一种是使用预处理器：

```
#define BYTE char //reprocessor replace BYTE with char
```

这样，预处理器将在编译程序时用 char 替换所有的 BYTE，从而使 BYTE 成为 char 的别名。

第二种方法是使用 C++（和 C）的关键字 typedef 来创建别名。例如，要将 byte 作为 char 的别名，可以这样做：

```
typedef char byte; //make byte an alias for char
//typedef typename aliasname;
```

要让 byte_pointer 成为 char 指针的别名，可将 byte_pointer 声明为 char 指针，然后在前面加上 typedef：

```
typedef char * byte_pointer;
// or
#define float_pointer float *
float_pointer pa, pb;
// preprocessor convert to
float * pa, pb; //pa a pointer, pb just a float
```

对于一系列变量的声明，使用 #define 可能并不适用，如上所示的第 6 行，预处理器置换声明时对后面的变量不一致。typedef 方法不会出现这样的问题。它能够处理更复杂的类型别名。

2. 3 do while 循环

```
        do
                body
        while (test-expression);
```

2. 4 基于范围的 for 循环（C++11）

C++11 新增了一种循环：基于范围（range-based）的 for 循环。这简化了一种常见的循环任务：对数组（或容器类，如 vector 和 array）的每个元素执行相同的操作

```
        double price[5] = {4.55, 12.11, 23.12, 33.12, 3.11};
        for (double x : price)
                cout << x << std::endl;
```

要修改数组的元素，需要使用不同的循环变量语法：

```
        for (double &x : price)
                x = x * 0.8;
```

符号 & 表明 x 是一个引用变量（我认为这是直接对数组的地址进行引用，从而 x 就是真实的数组元素，而不是复制的副本，这时对 x 的改变就是对地址中存储值的改变）。

2.5 循环和文本输入

cin 对象支持 3 种不同模式的单字符输入，其用户接口各不相同。

2.5.1 使用原始的 cin 进行输入

如果程序要使用循环来读取来自键盘的文本输入，必须要有办法知道何时停止读取。一种方法是选择某个特殊字符——有时被称为哨兵字符，将其作为停止标记。

```
        //...
        cin >> ch;
        while (ch != '#')
        {
                cout << ch;
                ++count;
                cin >> ch;
        }
//input
see ken run#really fast
//output
seekenrun
```

程序输出时省略了空格，原因在 cin。读取 char 值时，与读取其他基本类型一样，cin 将忽略空格和换行符。因此，输入中的空格没有被回显，也没有被包括在计数内。

而且发送给 cin 的输入被缓冲。这表示只有用户按下回车键后，输入的内容才会被发送给程序。这就是在运行程序时，可以在 # 后输入字符的原因。

2.5.2 使用 cin.get(cahr) 进行补救

cin 所属的 istream 类中包含一个能够满足这种要求的成员函数。具体来说，成员函数 cin.get(ch) 读取输入中的下一个字符（即使它是空格），并将其赋给变量。不过该输入仍被缓冲。但是在 C 语言中，要修改变量的值，必须将变量地址传递给函数。而调用 cin.get(ch) 传递的是 ch，而不是 &ch。在 C 语言中，这样的代码无效。而在 C++ 中有效，只要函数将参数声明为引用即可。

2.5.3 文件尾条件

使用诸如 # 等符号来表示输入结束很难令人满意，这样的符号可能就是合法输入的组成部分。如果输入来自文件，则可以使用一种功能强大的技术——检测文件尾（EOF）。C++输入工具和操作系统协同工作，来检测文件尾并将这种信息告知程序。

读取文件中的信息似乎同 cin 和键盘输入没什么关系，但其实存在两个相关的地方。首先，很多操作系统（包括 Unix、Linux 和 Windows 命令提示符模式）都支持重定向，允许用文件替换键盘输入。其次，很多操作系统都允许通过键盘来模拟文件尾条件。在 Unix 中，可以在行首按下 Ctrl+D 来实现；在 Windows 命令提示符模式下，可以在任意位置按 Ctrl+Z 和 Enter。总之，很多 PC 编程环境都将 Ctrl+Z 视为模拟的 EOF，但具体细节（必须在行首还是可以在任何位置，是否必须按下回车键等）各不相同。

检测到 EOF 后，cin 将两位（eofbit 和 failbit）都设置为 1。可以通过成员函数 eof() 来查看 eofbit 是否被设置；如果检测到 EOF，则 cin.eof() 将返回 bool 值 true，否则返回 false。同样，如果 failbit 被设置为 1，则 fail() 成员函数返回 true，否则返回 false。注意，eof() 和 fail() 方法报告最近读取的结果；也就是说，它们在事后报告，而不是预先报告。因此应将 cin.eof() 或 cin.fail() 测试放在读取后。

第3章 分支语句和逻辑运算符

3.1 字符函数库 cctype

C++从 C 语言继承了一个与字符相关的、非常方便的函数软件包，它可以简化诸如确定字符是否为大写字母、数字、标点符号等工作，这些函数的原型是在头文件 cctype（老式的风格中为 ctype.h）中定义的。

表 3-1 cctype 中的字符函数

函数名称	返回值
isalnum()	如果参数是字母数字，即字母或数字，该函数返回 true
isalpha()	如果参数是字母，该函数返回 true
iscntrl()	如果参数是控制字符，该函数返回 true
isdigit()	如果参数是数字（0 ～ 9），该函数返回 true
isgraph()	如果参数是除空格之外的打印字符，该函数返回 true
islower()	如果参数是小写字母，该函数返回 true
isprint()	如果参数是打印字符（包括空格），该函数返回 true
ispunct()	如果参数是标点符号，该函数返回 true
isspace()	如果参数是标准空白字符，如空格、进纸、换行符、回车、水平制表符或者垂直制表符，该函数返回 true
isupper()	如果参数是大写字母，该函数返回 true
isxdigit()	如果参数是十六进制数字，该函数返回 true
tolower()	如果参数是大写字符，则返回其小写，否则返回该参数
toupper()	如果参数是小写字符，则返回其大写，否则返回该参数

3.2 ?: 运算符

?: 运算符也被称为条件运算符，它是 C++ 中唯一一个需要 3 个操作数的运算符。通用格式为

```
1        expression1 ? expression2 : expression3
2        5 > 3 ? 10 : 12 //5>3 is true, so expression value is 10
```

3.3 简单文件输入/输出

3.3.1 写入到文本文件中

对于文件输入，C++ 使用了类似于 cout 的东西。下面列出有关将 cout 用于控制台输出的基本事实

- 必须包含头文件 iostream。
- 头文件 iostream 定义了一个处理输出的 ostream 类。
- 头文件 iostream 声明了一个名为 cout 的 ostream 对象。
- 必须指明名称空间 std；例如，为引用元素 cout 和 endl，必须使用编译指令 using 或前缀 std::。
- 可以结合使用 cout 和运算符¡¡来显示各种类型的数据。

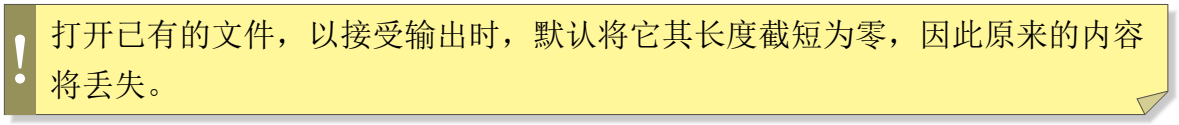
文件输出与此极为相似。

- 必须包含头文件 fstream。
- 头文件 fstream 定义了一个用于处理输出的 ofstream 类。
- 需要声明一个或多个 ofstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std；例如，为引用元素 ofstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ofstream 对象与文件关联起来。为此，方法之一是使用 open() 方法。
- 使用完文件后，应使用 close() 将其关闭。
- 可结合使用 ofstream 对象和运算符¡¡来输出各种类型的数据。

```
1        ofstream outFile  //outFile an ofstream object
2        ofstream fout  //fout an ofstream object
3        outFile.open("fish.txt");
4        char filename[50];
5        cin >> filename;
6        fout.open(filename);
7        double wt = 125.8;
8        outFile << wt; //write a number to fish.txt
9        char line[81] = "Object are closer than they appear.";
10       fout << line << endl;
```

声明一个 ofstream 对象并将其同文件关联起来后，便可以像使用 cout 那样使用它。所有可用于 cout 的操作和方法都可用于 ofstream 对象。

默认情况下，open() 将首先截断该文件，即将其长度截短到零——丢其原有的内容，然后将新的输出加入到该文件中。



3.3.2 读取文本文件

回顾文本文件输入，它是基于控制台输入的。控制台输入涉及多个方面，下面首先总结这些方面。

- 必须包含头文件 iostream。
- 头文件 iostream 定义了一个用处理输入的 istream 类。
- 头文件 iostream 声明了一个名为 cin 的 istream 变量（对象）。
- 必须指明名称空间 std；例如，为引用元素 cin，必须使用编译指令 using 或前缀 std::。
- 可以结合使用 cin 和运算符¿¿来读取各种类型的数据。
- 可以使用 cin 和 get() 方法来读取一个字符，使用cin 和 getline() 来读取一行字符。
- 可以结合使用 cin 和 eof()、fail()方法来判断输入是否成功。
- 对象 cin 本身被用作测试条件时，如果最后一个读取操作成功，它将被转换为布尔值 true，否则被转换为 false。

文件输入与此极其相似：

- 必须包含文件头 fstream。
- 头文件 fstream 定义了一个用于处理输入的 ifstream 类。
- 需要声明一个或多个 ifstream 变量（对象），并以自己喜欢的方式对其进行命名，条件是遵守常用的命名规则。
- 必须指明名称空间 std；例如，为引用元素 ifstream，必须使用编译指令 using 或前缀 std::。
- 需要将 ifstream 对象与文件关联起来。为此，方法之一是使用 open()方法。
- 使用完文件后，应使用 close() 方法将其关闭。
- 可结合使用 ifstream 对象和运算符¿¿来读取各种类型的数据。
- 可以使用 ifstream 对象和 get()方法来读取一个字符，使用 ifstream 对象和 getline()来读取一行字符。
- 可以结合使用 ifstream 和 eof()、fail()等方法来判断输入是否成功。
- ifstream 对象本身被用作测试条件时，如果最后一个读取操作成功，它将被转换为布尔值 true，否则被转换为 false。

```
1         ifstream inFile; // inFile an ifstream object
2         ifstream fin; //fin an ifstream object
3         inFile.open("bowling.txt");
4         char filename[50];
5         cin >> filename;
6         fin.open(filename);
7         double wt;
8         inFile >> wt; //read a number from bowling.txt
9         char line[81];
10        fin.getline(line, 81); //read a line of text
```

声明一个 ifstream 对象并将其同文件关联起来后，便可以像使用 cin 那样使用它。所有可用于 cin 的操作和方法都可用于 ifstream 对象

! Windows 文本文件的每行都以回车字符和换尾符结尾；通常情况下，C++在读取文件时将这两个字符转换为换行符，并在写入文件时执行相反的转换。有些文本编辑器不会自动在最后一行末尾加上换行符。此时需要在最后按下回车键，然后再保存文件。

第4章 函数——C++的编程模块

4.1 函数基础知识

要使用 C++函数，必须要完成如下工作：

- 提供函数定义
- 提供函数原型
- 调用函数

4.1.1 定义函数

可以将函数分为两类：没有返回值的函数和有返回值的函数。没有返回值的函数被称为 void 函数，其通用格式为：

```
void functionName (parameterList)
{
        statements;
        return;  //optional
}
```

有返回值的函数将生成一个值，并返回给调用函数。通用格式为：

```
typeName functionName (parameterList)
{
        statements;
        return value;
}
```

有返回值的函数，必须使用返回语句，以便将值返回给调用函数。值本身可以是常量、变量，也可以是表达式，只是其结果的类型必须为 typeName 类型或可以被转换为 typeName（例如，如果声明是返回类型为 double，而函数返回一个 int 表达式，则该 int 值将被强制转换为 double 类型）。C++对于返回值的类型有一定的限制：不能是数组，但是可以是其它任何类型——整数、浮点数、指针，甚至可以是结构和对象！

4.1.2 函数原型和函数调用

```
//using prototype and function calls
#include<iostream>
void cheers(int); //prototype;no return value
double cube(double x); //prototype; return a double
int main()
{
        ...
}
void cheers(int n)
{
        ...
}
double cube(double x)
{
        return x*x*x;
}
```

1. 为什么需要函数原型

   原型描述了函数到编译器的接口，它将函数返回值的类型（如果有的话）以及参数的类型和数量告诉编译器。

2. 原型的语法

   函数原型是一条语句，必须以分号结束，如上述代码所示，对于 cheer()的原型，只提供了参数类型。通常在原型的参数列表中，可以包括变量名，也可以不包括变量名。原型中的变量名相当于占位符，因此不必与函数定义中的变量名相同。

4.2 函数参数和按值传递

4.2.1 数组和函数

将数组做为函数参数时，我们知道如下恒等式

```
arr == &arr[0];
arr[i] == *(arr + i);
&arr[i] == arr + i;
```

传递常规变量时，函数将使用该变量的拷贝；但传递数组时，由于传递的是数组的地址，函数将使用原来的数组。为防止函数无意中修改数组的内容，可在声明形参时使用关键字 const，如

```
void show_array(const double ar[], int n);
```

该声明表明，指针 ar 指向的时常量数据。这意味着不能使用 ar 修改该数据，可以使用其值。这并不意味着原始数组必须是常量，而只是意味着不能在 show_array()函数中使用 ar 来修改这些数据，因此，此函数将数组视为只读数据。

4.2.2 指针和 const

可以用不同的方式将 const 关键字用于指针，第一种方法是让指针指向一个常量对象，这样可以防止使用该指针来修改所指向的值，第二种方法是将指针本身声明为常量，这样可以防止改变指针指向的位置。

```
//声明一个指向常量的指针
int age = 13;
const int * pt = &age;
```

这里将常规变量的地址赋给指向 const 的指针。还有两种可能：将 const 的变量的地址赋给指向 const 的指针（可行）、将 const 的地址赋给常规指针（不可行，表面上可以修改指针指向的值来修改变量的值，这使得解释变得模糊）

> **!** 如果数据类型本身并不是指针，则可以将 const 数据或非 const 数据的地址赋给指向 const 的指针，但非 const 指针只能指向非 const 数据的地址。

尽可能使用 const

将指针参数声明为指向常量数据的指针有两条理由：

- 这样可以避免由于无意间修改数据而导致的编程错误；
- 使用 const 使得函数能够处理 const 和非 const 实参，否则将只能接受非 const 数据。

如果条件允许，则应将指针形参声明为指向 const 的指针。

第二种使用 const 的方式使得无法修改指针的值：

```
//first method
int age = 23;
const int * pt = &age;
int sage = 32;
pt = &sage; //okay to point to another location
//second method
int sloth = 3;
const int * ps = &sloth; //a pointer to const int
int * const finger = &sloth; //a const pointer to int
```

第二种方法中，这种声明格式使得 finger 只能指向 sloth，但允许使用 finger 来修改 sloth 的值。而不能使用 ps 来修改 sloth 的值，但允许将 ps 指向另一个位置。即 finger 和*ps 都是 const，而*finger 和 ps 不是。

4.3 函数和二维数组

4.4 函数和 C-风格字符串

4.5 函数和结构

使用结构编程时，最直接的方式是像处理基本类型那样来处理结构；将结构作为参数传递，并在需要时将结构用作返回值使用。但如果结构非常大，则复制结构将增大内存要求，降低系统运行的速度。出于这些原因，许多程序员倾向于传递结构的地址，然后使用指针来访问结构的内容。

4.5.1 传递方式

- 传递和返回结构
  当结构比较小时，按值传递结构最合理

```
struct travel_time
{
        int hours;
        int mins;
};
travel_time sum(travel_time t1, travel_time t2);
```

- 传递结构的地址
  假设要传递结构的地址而不是整个结构以节省时间和空间，使用指向结构的指针。

```
travel_time sum(travel_time * t1, travel_time * t2);
```

4.6 函数与对象

4.7 函数指针

与数据项相似，函数也有地址。函数的地址是存储其机器语言代码的内存的开始地址。可以编写将另一个函数地址作为参数的函数。这样第一个函数将能够找到第二个函数，并运行它。与直接调用另一个函数相比，这种方法很笨拙。

4.7.1 函数指针的基础知识

1. 获取函数的地址

   只要使用函数名即可。即如果 think()是一个函数，则 think 就是该函数的地址。

2. 声明函数指针

   声明指向函数的指针时，必须指定指针指向的函数类型。即声明应指定函数的返回类型以及函数的特征标（参数列表）。

```
double pam(int);
double (*pf)(int); //pf points to a function that takes
//one int arg and return type double
pf = pam; //pf points to pam() function
double *pf(int); //pf() a function that return a pointer-to-double
double ned(double);
int ted(int);
pf = ned; //invalid -- mismatched signatures
pf = ted; //invalid -- mismatched return types
```

3. 使用指针来调用函数

   即使用指针来调用被指向的函数，如(*pf)(5)。

```
const double *(*(*pd)[3])(const double *,int) = &pa;
//pd = &pa;
```

如上代码所示，对于指针的初始化，是对指针变量的初始化，而不是对指针形式的初始化。即上述代码是 pd=&pa，而非*(*(*pd)[3]=&pa)。即初始化的是指针，而不是指针指向的值。

第5章 函数探幽

5.1 内联函数

内联函数是 C++为提高程序运行速度所做的一项改进。常规函数和内联函数之间的主要区别不在于编写方式，而在于 C++编译器如何将它们组合到程序中。常规函数调用使程序跳到另一个地址，并在函数结束时返回。而内联函数的编译代码与其他程序代码“内联”起来了。编译器将使用相应的函数代码替换函数调用。对于内联代码，程序无需跳到另一个位置处执行代码，再跳回来。因此，内联函数的运行速度比常规函数稍快，但代价是需要占用更多内存。

应有选择地使用内联函数。如果执行函数代码的时间比处理函数调用机制的时间长，则节省的时间将只占整个过程的很小一部分。如果代码执行时间很短，则内联调用就可以节省非内联调用使用的大部分时间。不过由于这个过程相当快，尽管节省了该过程的大部分时间，但节省的时间绝对值并不大，除非该函数经常被调用。使用方式：

- 在函数声明前加上关键字 inline；
- 在函数定义前加上关键字 inline。

通常的做法是省略原型，将整个定义（函数头和所有函数代码）放在本应提供原型的地方。

> ! 将函数作为内联函数时，编译器并不一定会满足这种要求。它可能认为该函数过大或注意到函数调用了自己（内联函数不能递归），因此不将其作为内联函数；而有些编译器没有启用或实现这种特性。

内联函数和常规函数一样，也是按值来传递参数的。如果参数为表达式，则函数将传递表达式的值。这使得 C++的内联功能远胜过 C 语言的宏定义。

内联与宏

inline 工具是 C++新增的特性。C 语言使用预处理其语句#define 来提供宏——内联代码的原始实现。例如，下面是一个计算平方的宏：
#define SQUARE(X) X*X
这并不是通过传递参数实现的，而是通过文本替换来实现的——X是“参数”的符号标记。

```
a = SQUARE(5.0); //replaced by a = 5.0*5.0;
b = SQUARE(4.5+7.5);//replaced by b = 4.5+7.5*4.5+7.5;
d = SQUARE(c++);//replaced by d = c++ * c++;
```

上述示例只有第一个能正常工作。可通过使用括号进行改进：#define SQUARE(X) ((X)*(X)) 这样仍存在这样的问题，即宏不按值传递。即新的定义中，SQUARE(c++) 仍将 c 递增两次。

5.2 引用变量

引用是已定义的变量的别名（另一个名称）。引用变量的主要用途是用作函数的形参。通过将引用变量用作参数，函数将使用原始数据，而不是副本。

5.2.1 创建引用变量

C++给&符号赋予了另一个含义，将其用来声明引用。例如

```
int rats;
int & rodents = rats; //makes rodents an alias for rats
```

其中，&不是地址运算符，而是类型标识符的一部分。像指针一样，int &指的是指向 int 的引用。上述引用声明允许将 rats 和 rodents 互换——它们指向相同的值和内存单元。

引用和指针有一定的区别。必须在声明引用是将其初始化，而不能像指针那样，先声明，再赋值。引用更接近 const 指针，必须在创建时进行初始化，一旦与某个变量关联起来，就将一直效忠于它。

5.2.2 将引用用作函数参数

引用经常被用作函数参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为按引用传递。按引用传递允许被调用的函数能够访问调用函数中的变量。按值传递导致被调用函数使用调用程序的值的拷贝。

5.3 引用的属性和特别之处

如果想让函数使用传递给它的信息，而不对这些信息进行修改，同时又想使用引用，则应使用常量引用，即使用 const：

```
1 double refcube(const double &ra);
```

按值传递的函数，可使用多种类型的实参。但是传递引用的限制将比较严格。例如，refcube(x+3.0)，在现代C++中，这是错误的，因此表达式并不是变量。而有些较老的编译器将发出警告。有些情况下允许这样做，此时程序将创建一个临时的无名变量，并将其初始化为表达式的值。

如果实参与引用参数不匹配，C++将生成临时变量。当前，仅当参数为 const 引用时，C++才允许这样做。如果引用参数是 const，则编译器将在下面情况下生成临时变量：

- 实参的类型正确，但不是左值
- 实参的类型不正确，但可以转换为正确的类型

左值参数是可被引用的数据对象，例如，变量、数组元素、结构成员、引用和解除引用的指针都是左值。非左值包括字面常量（用引号括起的字符串除外，它们由其地址表示）和包含多项的表达式。在C中，左值最初指的是可出现在赋值语句左边的实体，但是这是引入关键字 const 之前的情况。现在，常规变量和 const 变量都可以视为左值，因为可以通过地址访问它们。

```
1 double refcube(const double &ra);
2 double side = 3.0;
3 double * pd = &side;
4 double & rd = side;
5 long edge = 5L;
6 double len[4] = {2.0, 5.0, 10.0, 12.0};
7 double c1 = refcube(side);//ra is side
8 double c2 = refcube(lens[2]);//ra is lens[2]
9 double c3 = refcube(rd);//ra is rd
10 double c4 = refcube(*pd);//ra is *pd
```

```
double c5 = refcube(edge);//ra is temporary variable
double c6 = refcube(7.0);//ra is temporary
double c7 = refcube(side+10.0);//ra is temporary
```

上述代码中将生成一个临时匿名变量，并让 ra 指向它。这些临时变量只在函数调用期间存在，此后编译器便可以随意将其删除。

> ! 尽量避免临时变量的创建

5.3.1 将引用用于结构

引用非常适合用于结构和类。使用结构引用参数的方式与使用基本变量引用相同。

返回引用需要注意的一点是，应避免返回函数终止时不再存在的内存单元引用。如：

```
const free_throws & clone(free_throws & ft)
{
        free_throws newguy;
        newguy = ft;
        retunr newguy;
}
```

该函数返回一个指向临时变量的引用，函数运行完毕后它将不再存在。同样，也应避免返回指向临时变量的指针。

5.3.2 将引用用于类对象

将类对象传递给函数时，C++通常的做法是使用引用。

5.3.3 对象、继承和引用

5.4 函数重载

默认参数使得能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）使得能够使用多个同名的函数。可可以通过函数重载来设计一系列函数——它们完成相同的工作，但使用不同的参数列表。

函数重载的关键是函数的参数列表——也称函数特征标。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是

无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。

```
void print(const char * str, int width); //#1
void print(double d, int width);         //#2
void print(long l, int width);           //#3
void print(int i, int width);            //#4
void print(const char * str);            //#5
```

使用 print()函数时，编译器将根据所采取得用法使用相应特征标得原型：

```
print("Panda", 6);     //use #1
print("Syrup");        //use #5
print(1999.0, 10);     //use #2
print(1999, 12);       //use #4
print(1999L, 15);      //use #3
```

当函数调用时未使用正确得参数类型。如：

```
unsigned int year = 3210;
print(year, 6);
```

上述的函数调用不与任何原型匹配。此时C++将尝试使用标准类型转换强制进行匹配。如果#2时是 print()唯一的原型，则函数调用将把 year 转换为 double 类型。但是上述“符合条件”的有三种类型。在这种情况下，C++将拒绝这种函数调用，视其为错误。

编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标，以避免使用混乱。

是特征标，而不是函数类型使得可以对函数进行重载。

```
long gronk(int n, float m);
double gronk(int n, float m);
```

C++不允许以这种方式重载 gronk()。返回类型可以不同，但特征标也必须不同，因为是根据特征标识别调用的。

重载引用参数

```
1 void sink(double & r1);
2 void sink(const double & r2);
3 void sink(double && r3);
```

左值引用参数 r1 与可修改的左值参数匹配；const 左值引用参数 r2 与可修改的左值参数、const 左值参数和右值参数匹配；右值引用参数 r3 与右值匹配。如果重载使用这三种参数的函数，将调用最匹配的版本：

```
1 void staff(double & rs); //matches modifiable lvalue
2 void staff(const double & rc); //matched rvalue,const lvalue
3 void stove(double & r1); //modifiable lvalue
4 void stove(const double & r2); //const lvalue
5 void stove(double && r3) //rvalue
```

名片修饰

C++给重载函数指定了秘密身份。使用C++开发工具中的编辑器编写和编译程序时，C++编译器将执行一些神奇的操作——名称修饰或名称矫正，它根据函数原型中指定的形参类型对每个函数名进行加密。如下述未经修饰的函数原型：

long MyFunctionFoo(int, float);

这种格式对于人类来说很合适；而编译器将名称转换为不太好看的内部表示，来描述该接口，如下所示：

?MyFunctionFoo@@YAXH

对原始名称进行的表面看来无意义的修饰将对参数数目和类型进行编码。添加的一组符号随函数特征标而异，而修饰时使用的约定随编译器而异。

5.5 函数模板

函数模板是通用的函数描述，它们使用泛型来定义函数，其中的泛型可用具体的类型替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。由于模板允许以泛型的方式编写程序，因此有时也被称为通用编程。由于类型是用参数表示的，因此模板特性有时也被称为参数化类型。

函数模板允许以任意类型的方式来定义函数。例如，可以这样建立一个交换模

板：

```
template <typename AnyType>
void swap(AnyType & a, AnyType & b)
{
        AnyType temp;
        temp = a;
        a = b;
        b = temp;
}
```

第一行指出要建立一个模板，并将其类型命名为 AnyType。关键字 template 和 typename 是必需的。在 C++98 添加关键字 typename 之前，C++使用关键字 class 来创建模板。另外尖括号也是必须的。类型名可以任意选择，只要遵守 C++命名规则即可。模板并不创建任何函数，只是告诉编译器如何定义函数。如上诉代码，需要交换 int 的函数时，编译器将按模板模式创建这样的函数，并用 int 代替 AnyType。

> ! 函数模板不能缩短可执行程序。在程序执行时，最终仍由多个独立的函数定义，就像以手工的方式定义了这些函数一样。最终的代码不包含任何模板，而只包含为程序生成的实际函数。使用模板的好处是，它使生成多个函数定义更简单，更可靠。

5.5.1 重载的模板及模板的局限性

可以像重载常规函数定义那样重载模板定义，而且和常规函数重载一样，被重载的模板的函数特征标必须不同。

不过编写的模板函数很可能无法处理某些类型。即如果 T 为数组、指针或结构，T 类型的变量就不能直接进行乘除运算。

第6章 内存模型和函数空间