C++语言程序设计

# 第一部分 基础篇

## 一、什么是C++

### 1.1 C++ 简介

C++ 是一门非常经典的高级编程语言。顾名思义，C++可以看做是C语言的增强版，在C的基础上扩展了更多的功能；最主要的扩展，就是面向对象和泛型编程。

因此C++融合了多种不同的编程方式：以C语言为代表的面向过程编程；面向对象编程；以及模板化的泛型编程。

可以说，C++一门“大而全”的编程语言，你可以用它实现想要的任何功能；与此同时，学习C++需要掌握的内容也会比较多。

#### 1.1.1 C和C++

20世纪70年代，贝尔实验室的Dennis Ritchie为了开发UNIX操作系统，专门设计了一门结构化的高级语言，这就是大名鼎鼎的C语言。因为是为操作系统设计的语言，它本身是比较底层的，所以C具有低级语言的高运行效率、硬件访问能力，此外又融合了高级语言的通用性。

C语言语法清晰，具有非常好的结构化编程的特性。于是C语言快速地统治了底层的系统级编程，并成为了之后几十年内经典的教学语言。

C语言编程的整体思路是“过程式”的，也就是说，我们把想让计算机执行的操作按照步骤一步步定义好，然后用C语言写出来；所以我们写的代码，就是一个处理流程的描述。这种方式很容易理解，也可以非常方便地翻译成计算机能懂的机器语言；但是在面对大型项目、代码量非常大时，就会显得杂乱无章，代码的可读性就大大降低了。

于是另一种编程方式应运而生，这就是面向对象编程。这种方式的主要思路是先构建“对象”，然后通过定义好的对象行为，实现我们想要的操作。

贝尔实验室的 Bjarne Stroustrup（比雅尼·斯特劳斯特鲁普），在20世纪80年代创建了一个新的面向对象语言——C++。

名字一目了然，它是基于C的，扩展了C的功能；所以C++是C语言的超集，所有C语言程序都可以在C++的环境下运行。而扩展的部分，主要就是引入了面向对象的特性，并实现了对C的泛型编程支持。

C++的出现极大地扩充了C的应用场景，为C语言的长盛不衰提供了很大的助力。所以我们平常看招聘要求的技术栈描述，往往是把C/C++放在一起说的。

#### 1.1.2 C++ 的应用场景

C++完全兼容C，具有C面向硬件的特性；此外还拥有面向对象和泛型编程的扩展。所以C++编写的程序运行效率高、功能强大，特别适合用在系统级应用场景上。所以我们经常可以看到，偏向底层、系统的开发，一般用的语言都是C++。

* 底层硬件，系统编程：JVM的底层，Python解释器的底层，都离不开C/C++的身影；人工智能核心库的代码，也大多是C++写的
* 嵌入式开发
* 游戏开发

当然，除了这些实际应用场景外，由于C/C++是经典的教学语言，因此计算机专业考研、考级、竞赛等场合往往也是把C++作为第一语言的。无论学习还是工作，C++都是一门非常有用的编程语言。

### 1.2 C++ 标准

C++作为一门高级编程语言，在不同的硬件平台上有着良好的可移植性。这意味着我们不需要改动代码，写出来的程序就可以在不同的平台“翻译”成机器能读懂的语言。要实现这个目标，就必须对C++编写的程序设定一些规范，这就是C++的标准。

C++之父Stroustrup写过一本《C++编程语言》（The C++ Programming Language），里面有一个参考手册，专门介绍了这门语言的特性和用法。这其实就是最初的C++事实标准。

不过真正意义上的标准，还需要专门的组织认证。ANSI（American National Standards Institute，美国国家标准局）在制定了C语言标准之后，在90年代专门设了一个委员会来制定C++的标准，并和ISO（国际标准化组织）一起创建了联合组织ANSI/ISO。1998年，第一个C++国际标准终于出炉了；这个标准在2003年又做了一次技术修订。因此我们一般所说的C++标准，第一版往往被叫做C++ 98/03。

跟大多数语言一样，C++也在不停地发展更新。ISO在2011年批准了C++新标准，这可以认为是C++的2.0版本，一般被叫做C++ 11。C ++ 11新增了很多新特性，极大地扩展了C++的语言表达能力。此后在2014年和2017年，又出了两个新版本C++标准，一般叫做C++ 14和C++ 17，不过这两个版本增加的内容并不多；真正意义上的下一个大版本是2020年的C++ 20，它再一次给C++带来了大量的新特性。

### 1.3 C++ 代码如何运行

我们用C++写好的代码，其实就是符合特定语法规则的一些文字和符号。计算机是怎样识别出我们想要做的操作、并正确执行呢？

这就需要一个专门的翻译程序，把我们写的源代码，翻译成计算机能理解的机器语言。这个翻译的过程就叫做“编译”，而这个“翻译官”就叫做编译器。所以C++是一门编译型的编程语言，这一点和C是一致的。

事实上，C++代码的运行过程跟C程序代码也是一样的，大致可以分为下面几步：

1. 首先编写C++程序，保存到文件中，这就是我们的源代码；
2. 编译。用C++编译器将源代码编译成机器语言，得到的这个结果叫做目标代码；
3. 链接。C/C++程序一般都会用到库（library），这些库是已经实现好的目标代码，可以实现特定的功能（比如在屏幕上把信息打印显示出来）。这时我们就需要把之前编译好的目标代码，和所用到的库里的目标代码，组合成一个真正能运行的机器代码。这个过程叫做“链接”，得到的结果叫做可执行代码；
4. 运行。可执行代码就是可以直接运行的程序，运行它就可以执行我们想要的操作了。

## 二、简单上手——Hello World

### 2.1 开发环境和工具（Visual Studio）

写C++程序其实很简单，直接用记事本写好代码，然后用一个编译器做编译运行就可以了；不过这意味这我们得自己保证语法正确，严重影响开发效率。所以实际应用中我们一般都会使用功能更强大的工具，除了提供编译器外，还可以给我们做语法检查和提醒，方便我们调试程序——这就是所谓的“集成开发环境”（IDE）。

Windows系统环境下，最普遍、最好用的IDE就是Visual Studio了，这是微软官方的开发工具，功能非常强大。

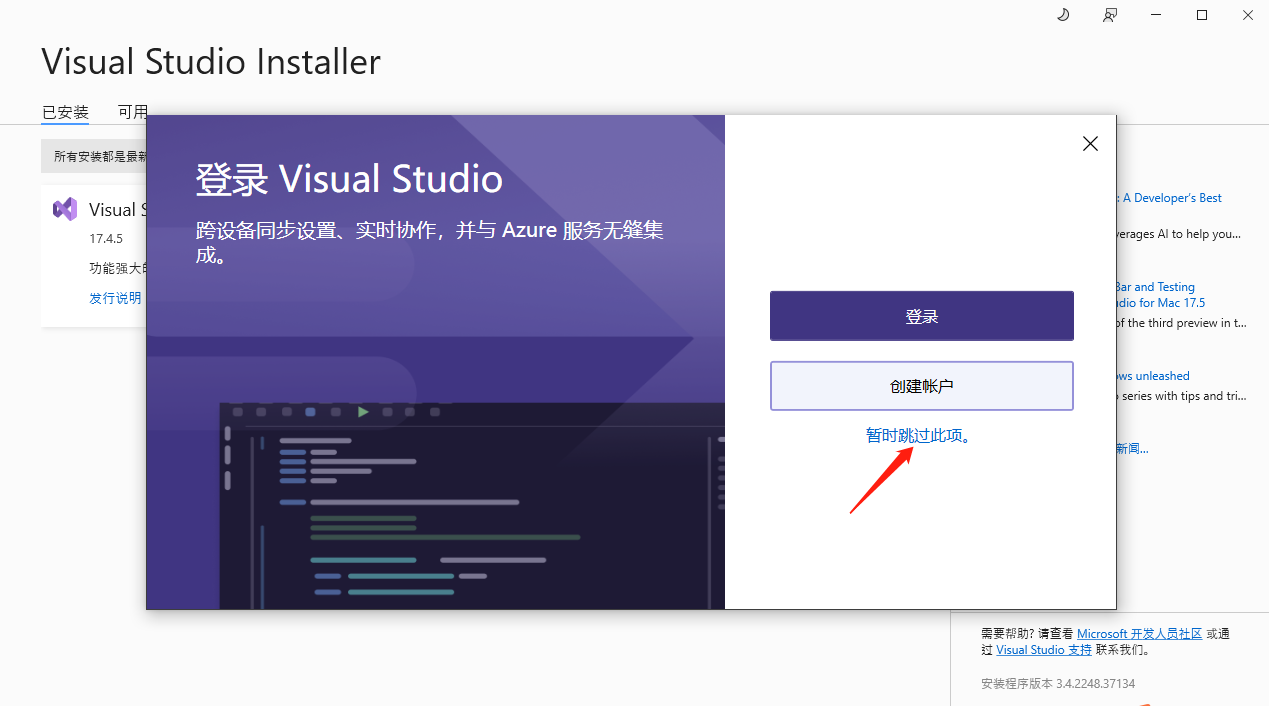
打开Visual Studio的中文版官方网站 <https://visualstudio.microsoft.com/zh-hans/>，点击“下载Visual Studio”按钮，选择最新的免费社区版Community 2022。然后双击运行安装程序VisualStudioSetup.exe。

在安装引导程序中，选择自己需要的组件。我们直接选择“使用C++的桌面开发”即可，这个选项会打包安装Windows下C++开发的所有组件。注意不需要选“通用Windows平台开发”，这个还包含了.net平台，是针对C#开发的。

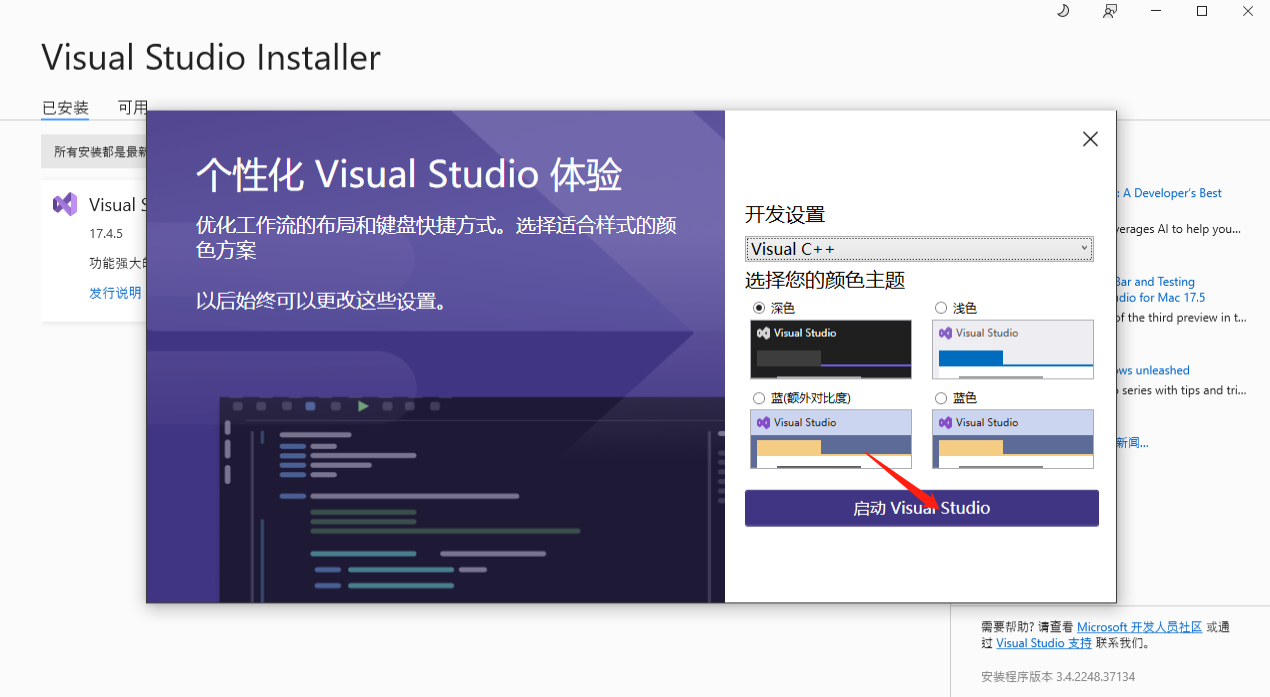


点击“安装”，引导程序会自动帮我们下载和安装所有需要的组件，这个过程可能需要花费一些时间。

如果选择了“安装后启动”，那么安装完成就会自动运行。开始的界面是登录微软账号，我们可以直接跳过。



选择开发设置为“Visual C++”，选择自己喜欢的界面主题色，然后点击启动。

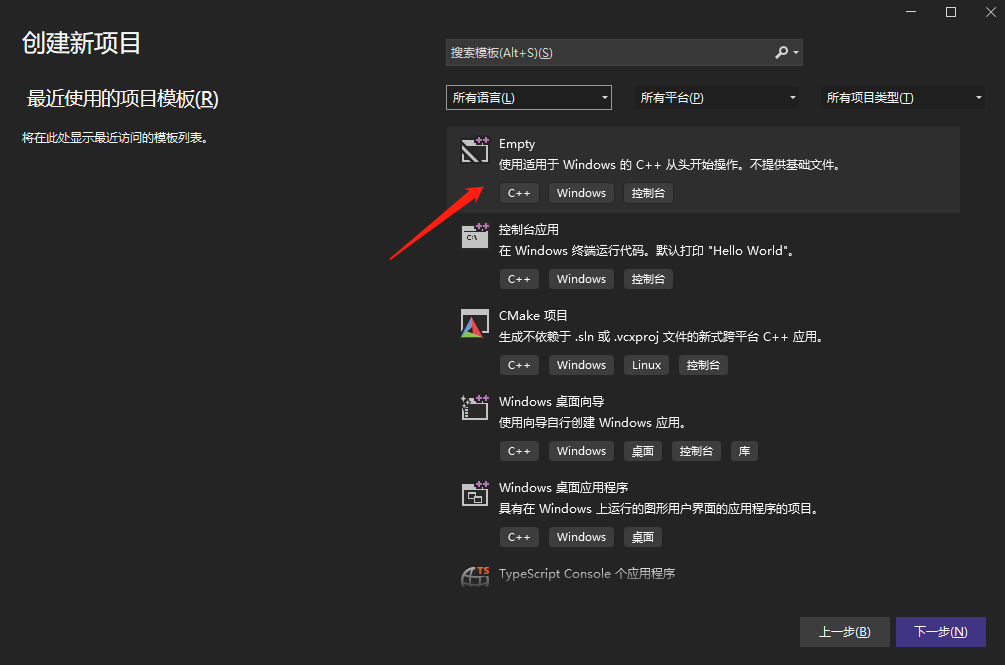


### 2.2 写一个Hello World

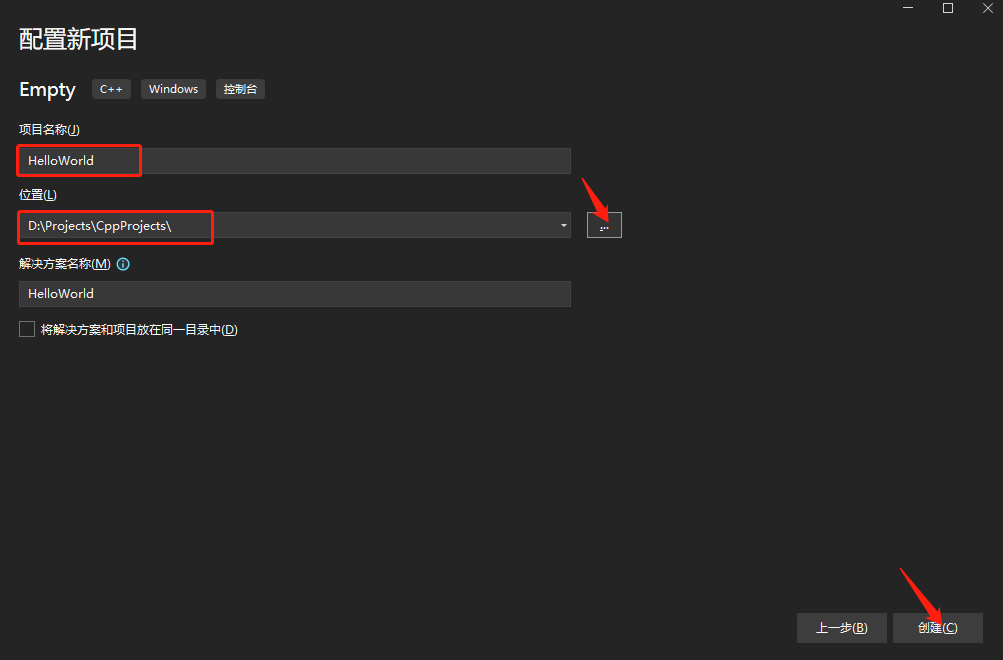
Visual Studio启动之后，我们首先应该创建一个项目。所谓“项目”，就是一个工作任务，需要实现相应的需求。点击“创建新项目”。



直接选择一个空项目。



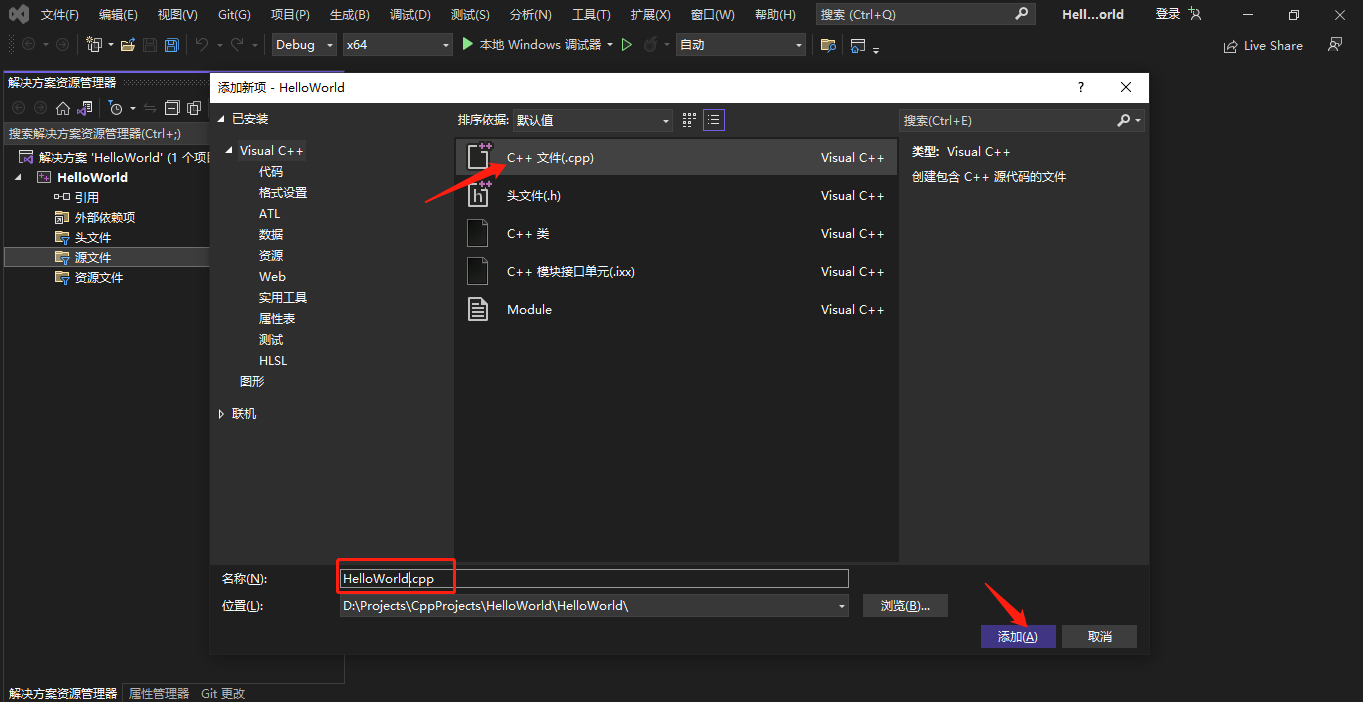
指定项目名称和保存位置。



这里还有一个“解决方案”（Solution）的概念，其实就是一组有关联的项目，共同合作解决一个需求。

#### 2.2.1 代码编写

在打开的解决方案界面里，右键点击“源文件”文件夹图标，添加一个新建项。我们要添加的是一个C++文件，命名为HelloWorld，后缀名是.cpp。



接下来我们就可以写代码了。

下面就是一段最简单的代码，我们在屏幕上输出Hello World。

#include<iostream>

int main()

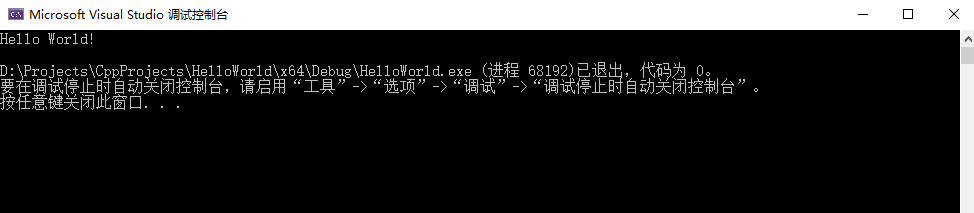
{

std::cout << "Hello World!" << std::endl;

}

我们可以点击工具栏的按钮 （快捷键F5），用一个本地的调试器来“调试”代码；所谓的调试，就是查看具体的运行过程，我们可以用它来解决出现的问题。当然也可以点它旁边的三角按钮，这是不调试直接运行（快捷键Ctrl+F5）。

结果如下：



界面上弹出了一个窗口，显示出了我们想要的信息“Hello World！”。后面还跟着一串信息，这是调试控制台告诉我们，程序已经执行完毕正常退出了。随便一个键，就可以关闭这个窗口。

#### 2.2.2 代码解读

这个简单的程序里，主要包括了这样几部分。

1. 第一行 #include<iostream>

这是一个预处理指令，告诉编译器我们需要使用一个叫做iostream的库。因为我们需要输出信息，而系统的标准库提供了这样的功能，所以要用#include做一个引入的预处理。

1. 主函数 main()

接下来的主体，是一个“主函数”。

所谓的函数，就是包装好的一系列要执行的操作，可以返回一个结果。一个C++程序可以包含很多函数，其中一个必须叫做main，它是执行程序的入口。也就是说，当我们运行这个程序的时候，操作系统就会找到这个“主函数”开始执行。

main() 的定义形式如下：

int main()

{

*statements*

return 0;

}

具体细分，第一行int main()叫做函数头，下面的花括号扩起来的部分叫函数体。函数头定义了函数的名字叫main，前面的int表示返回值是整数类型（integer）；后面的括号里面本应该写传入的参数列表，这里是空的。花括号包围的部分就是函数体，里面就是我们要执行的操作。

1. 语句

函数体里，每一步操作都是一个“语句”（statement），用分号结尾。我们这里的语句，执行的就是输出Hello World的操作。

std::cout << "Hello World!" << std::endl;

这是一个“表达式”。所谓表达式，一般由多个运算的对象和运算符组成，执行运算之后会得到一个计算结果。在这里，两个连在一起的小于号“<<”就是一个用来输出的运算符。它的使用规则是：左边需要一个“输出流”的对象，也就是输出到哪里；右边是要输出的内容，最简单的就是一个“字符串”，需要用双引号引起来。

所以 std::cout << "Hello World!" 的意思就是：将“Hello World！”这串信息，输出到cout这个对象。cout就是一个输出流对象，iostream库里定义了它的功能，接收到信息之后就可以输出显示了。而cout前面的std是所谓的“命名空间”（namespace），主要是为了避免还有别的cout对象重名起冲突。这里的双冒号“::”也是一个运算符，叫做作用域运算符，专门指明了我们用的cout是标准库std中的。如果不想总用双冒号，也可以直接加上一句：

using namespace std;

这样就可以直接用cout，不需要加std::了。

输出运算符 << 得到的计算结果，还是它左边的那个输出流对象cout。这样一来，我们就可以在后面继续写入信息信息了。所以后面的 << endl ，其实就是把 endl 这个内容，又写入到cout中输出了。这个endl是一个“操作符”，表示结束一行，并把缓冲区的内容都刷到输出设备。

1. 返回值

最后一行语句就是返回一个值。大多数系统中，main的返回值是用来指示状态的。返回0表示成功，非0表示出错，具体值可以用来表示错误类型，这是由系统定义的。

我们这里写了return 0，其实不写也是可以的，默认正常运行结束就会返回0。

#### 2.2.3 注释

可以看到，纯粹的代码还是比较抽象的；特别是当代码越来越多、越来越复杂之后，就会变得越来越难理解。所以我们一般会插入一些解释说明的文字，这叫做“注释”。注释不会被执行，对代码的功能没有任何影响。

在C++中，有两种注释的表示。一种是单行注释，用双斜线“//”，表示以它开始的当前行是注释内容；另一种是多行注释，使用一对“界定符”（/\* 和 \*/）,在它们之间的所有内容都是注释。

#include<iostream>

/\*

\* 主函数

\* Hello World

\*/

int main()

{

// 输出一行信息

std::cout << "Hello World!" << std::endl;

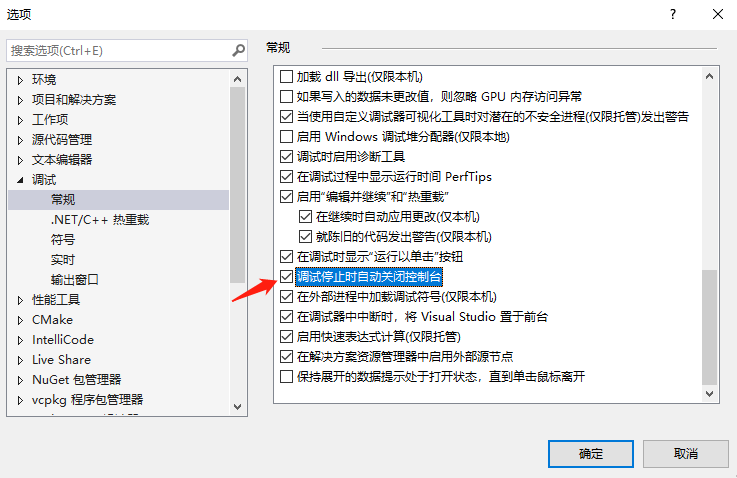
return 0;

}

#### 2.2.4 代码的改进——简单的输入输出

我们之前写的代码非常简单，实现了输出Hello World的功能。不过输出显示用的是“调试控制台”，运行完成总会显示一行额外信息，能不能让它更纯粹地运行、不显示多余内容呢？

当然可以，调试台输出的信息本身就有提示，只要更改一下VS的设置。要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。



不过出现了新的问题：再次运行的时候，窗口一闪而过，根本看不清输出了什么。为了查看输出结果，我们还是希望把窗口保持住、不要直接退出，这可以通过在main()函数中增加一句输入语句来实现：

int main()

{

// 输出一行信息

std::cout << "Hello World!" << std::endl;

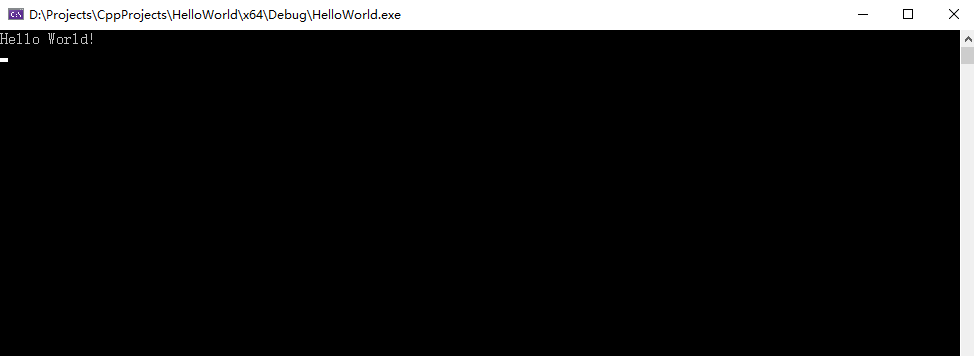
// 等待键盘输入

std::cin.get();

return 0;

}

这里的cin跟cout刚好相反，它是一个输入流对象。调用它内部的函数get()，就可以读取键盘的输入；等待键盘输入的时候，窗口就会一直开着。这里的键盘输入是以回车作为结束标志的，所以运行看到结果之后，直接敲回车就可以退出了。

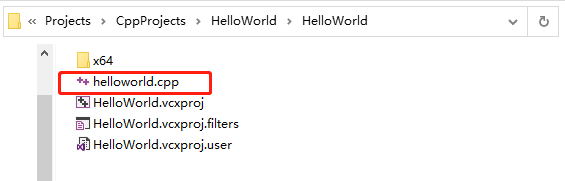


### 2.3 编译、链接和运行

我们之前写好C++代码之后，是直接在Visual Studio里借助“本地windows调试器”运行的；而如果真正开发一个软件，显然不能总是依赖VS的调试器运行。真正应用中，我们最终要得到一个“可执行文件”，一般以.exe作为扩展名，双击就可以运行程序了。

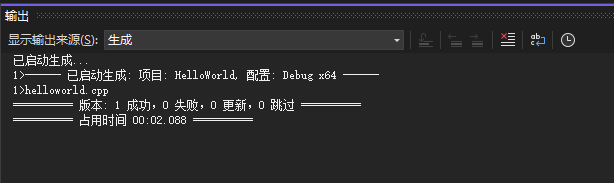
怎样转换得到可执行文件呢？之前已经提到，C++是一种编译型语言，在运行之前需要进行编译和链接。我们现在就用上节写好的Hello World代码，把这个过程具体说明一下。

首先我们可以在Visual Studio左侧的“解决方案资源管理器”里，右键点击创建的项目HelloWorld，选择“在文件资源管理器中打开文件夹”，就会进入保存项目的文件夹。

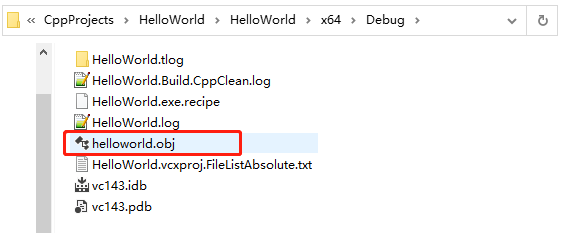


这里看到的helloworld.cpp，就是我们写好的C++源代码文件。其它的文件都是VS生成的项目文件。另外还有一个x64文件夹，是之前我们在本地进行调试运行时生成的，里面有一个Debug子文件夹，保存了调试运行的相关信息和日志。如果我们右键HelloWorld项目名，然后选择“清理”，Debug里面就只剩下一些日志和空文件了。

源代码首先需要编译（compile），得到目标代码。编译器当然是由Visual Studio提供的。我们首先点击一下源代码文件，然后在VS的菜单栏中选择“生成”-> “编译”（快捷键Ctrl+F7），就可以进行编译了。在下方的“输出”窗口内，可以看到编译的结果信息。

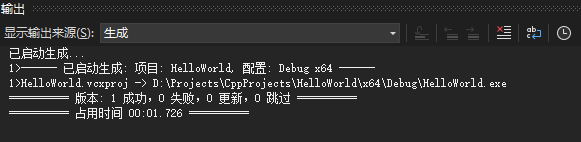


编译完成之后，再回到之前打开的项目文件夹，找到x64下的Debug目录，点进去之后就会发现多了几个文件，除了一些调试工具外，最重要的就是一个helloworld.obj，这就是编译生成的目标代码文件。



目标文件就是计算机能够直接运行的机器码。但是仅有helloworld.cpp源代码转换的机器码还不够。因为我们用到了iostream中的cout和cin对象进行输入输出操作，这就需要把iostream中对应的目标代码也提出来，组合成一个完整的、能直接运行的机器代码。这就是所谓的“链接”（link）过程，结果就会生成一个可执行文件。

在VS中，我们可以点击工具栏“生成” –> “生成HelloWorld”（快捷键Ctrl+B）；也可以直接右键HelloWorld项目名选择“生成”。在“输出”窗口可以清楚地看到，扩展名为.exe的可执行文件已经生成了。



在对应的目录找到这个文件，双击运行，我们会发现跟之前在调试器中的运行结果是一样的，可以直接在窗口中显示“Hello World！”，回车就会退出。这个.exe文件可以复制到任何位置，直接双击运行程序。

### 2.4 初步认识函数

通过一个最简单的Hello World程序，我们已经了解了C++基本的代码风格、简单的输入输出操作，以及程序编译运行的完整过程。利用这些知识我们可以为这个程序增加更多的功能，比如提示用户输入自己的名字XXX，然后显示“Hello， XXX”。

代码如下：

#include<iostream>

using namespace std;

int main()

{

// 输出一行信息

cout << "Hello World!" << endl;

// 提示输入姓名

cout << "请输入您的大名：" << endl;

// 用一个变量接收键盘输入

string name;

cin >> name;

// 输出欢迎信息

cout << "Hello, " << name << endl;

// 等待键盘输入

cin.get();

cin.get();

// 这里写两次是因为之前输入信息时敲回车确认，会由第一个get捕捉到

return 0;

}

但是这样代码就比较多了，可读性会变差。解决办法是，我们可以把中间一部分代码“包装”成函数，就像主函数一样。只不过这种函数不是启动直接调用的，而是需要在程序中明确地写出来什么时候调用。

代码如下：

#include<iostream>

using namespace std;

// 定义一个函数

void welcome()

{

cout << "Hello World!" << endl;

cout << "请输入您的大名：" << endl;

string name;

cin >> name;

cout << "Hello, " << name << endl;

}

int main()

{

// 调用函数

welcome();

// 等待键盘输入

cin.get();

cin.get();

return 0;

}

这样每一部分处理逻辑都可以分块包装成函数，主函数的执行过程看起来就简单多了。当然，如果认为一个文件中有太多函数也会影响可读性，我们还可以把它们分开。比如新建一个叫做welcom.cpp的源文件，专门放刚才的welcome函数。而在主函数中，需要额外对它做一个“声明”，表示有这样一个函数，它的实现在另外的文件里。

#include<iostream>

// 声明一个函数

void welcome();

int main()

{

// 调用函数

welcome();

cin.get();

cin.get();

return 0;

}

函数是C++中基本的编程单元，也是“模块化编程”的核心思想，我们还会在后面的章节详细展开。

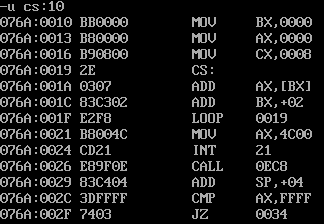
## 三、变量和数据类型

一段程序的核心有两个方面：一个是要处理的信息，另一个就是处理的计算流程。计算机所处理的信息一般叫做“数据”（data）。

对计算机来说，需要明确地知道把数据存放在哪里、以及需要多大的存储空间。在机器语言和汇编语言中，我们可能需要充分了解计算机底层的存储空间，这非常麻烦；而在C++程序中，我们可以通过“声明变量”的方式来实现这些。

### 3.1 变量和常量

为了区分不同的数据，在程序中一般会给它们起个唯一的名字，这就是所谓的“变量”。在C++中，“变量”其实就是记录了计算机内存中的一个位置标签，可以表示存放的数据对象。

#### 3.1.1 变量的声明和赋值

想要使用变量，必须先做“声明”，也就是告诉计算机要用到的数据叫什么名字，同时还要指明保存数据所需要的空间大小。比如：

int a;

这里包含两个信息：一个是变量的名字，叫做“a”，它对应着计算机内存中的一个位置；另一个是变量占据的空间大小，这是通过前面的“int”来指明的，表示我们需要足够的空间来存放一个“整数类型”（integer）数据。

所以变量声明的标准语法可以写成：

*数据类型 变量名;*

变量名也可以有多个，用逗号分隔就可以。

在C++中，可以处理各种不同类型的数据，这里的int就是最基本的一种“数据类型”（data type），表示一般的整数。

当然，如果我们直接在代码中声明一个变量，然后打印输出的话就会报错，因为这个变量没有被“初始化”。也就是说，a这个变量现在可以表示内存中一个位置了，但是里面的数据是什么？这就需要让a有一个“初始值”：

int a = 1;

这个操作叫做“赋值”。需要说明的是，这里等号“=”表示的是赋值操作，并不是数学上的“等于”。换句话说，我们还可以继续给a赋别的值：

int a = 1;

a = 2;

现在a的值就是2了。a的值可以改变，所以它叫做“变量”。

扩展知识：

C++是一种静态类型（statically typed）语言，需要在编译阶段做类型检查（type checking）。也就是说所有变量在创建的时候必须指明类型，而且之后不能更改。对于复杂的大型程序来说，这种方式更有助于提前发现问题、提高运行效率。

代码如下：

#include<iostream>

using namespace std;

int main()

{

int a = 1;

cout << "a = " << a << endl;

a = 2;

cout << "现在 a = " << a << endl;

cin.get();

}

要运行的话，可以右键项目名 -> 设为启动项目，或者右键解决方案 -> 设置启动项目。

注意，如果不给初始值，后面再赋值、再使用也是合法的；但一般不能不赋值、直接使用。因为在函数中定义的变量不被初始化，而在函数外部定义的变量会被默认初始化为0值。

#### 3.1.2 标识符

每个变量都有一个名字，就是所谓的“变量名”。在C++中，变量、函数、类都可以有自己专门的名字，这些名字被叫做“标识符”。

标识符由字母、数字和下划线组成；不能以数字开头；标识符是大小写敏感的，长度不限。

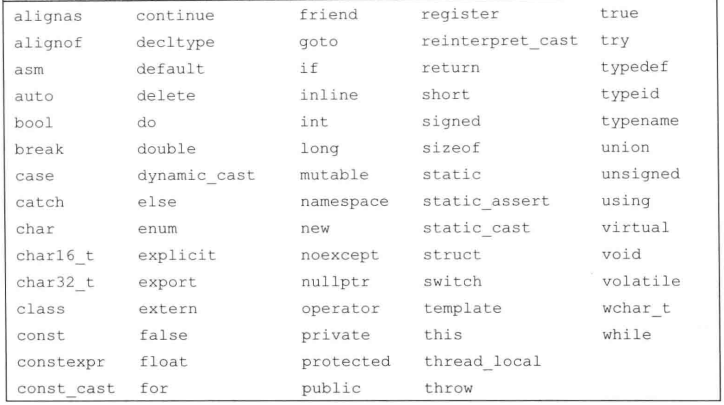
所以下面的变量名都是合法而且不同的：

int b, B, B2, a1\_B2;

此外，C++中还对变量命名有一些要求和约定俗成的规范：

* 不能使用C++关键字；
* 不能用连续两个下划线开头，也不能以下划线加大写字母开头，这些被C++保留给标准库使用；
* 函数体外的标识符，不能以下划线开头；
* 要尽量有实际意义（不要定义a、b，而要定义name、age）；
* 变量名一般使用小写字母；
* 自定义类名一般以大写字母开头；
* 如果包含多个单词，一般用下划线分隔，或者将后面的单词首字母大写；

所谓的“关键字”，就是C++保留的一些单词，供语言本身的语法使用。包括：



以及C++中使用的一些运算操作符的替代名：



#### 3.1.3 作用域

变量有了名字，那只要用这个名字就可以指代对应的数据。但是如果出现“重名”怎么办呢？

在C++中，有“作用域”（scope）的概念，就是指程序中的某一段、某一部分。一般作用域都是以花括号{}作为分隔的，就像之前我们看到的函数体那样。

同一个名字在不同的作用域中，可以指代不同的实体（变量、函数、类等等）。

定义在所有花括号外的名字具有“全局作用域”（global scope），而在某个花括号内定义的名字具有“块作用域”。一般把具有全局作用域的变量叫做“全局变量”，具有块作用域的变量叫做“局部变量”。

测试代码如下：

#include<iostream>

using namespace std;

// 全局作用域，全局变量

int number = 0;

int main()

{

// 块作用域，局部变量

int number = 1;

// 访问局部变量

cout << "number = " << number << endl;

// 访问全局变量

cout << "number = " << ::number << endl;

cin.get();

}

如果在嵌套作用域里出现重名，一般范围更小的局部变量会覆盖全局变量。如果要特意访问全局变量，需要加上双冒号:: ，指明是默认命名空间。

#### 3.1.4 常量

用变量可以灵活地保存数据、访问数据。不过有的时候，我们希望保存的数据不能更改，这种特殊的变量就被叫做“常量”。在C++中，有两种方式可以定义常量：

（1）使用符号常量

这种方式是在文件头用 #define 来定义常量，也叫作“宏定义”。

#define ZERO 0

跟#include一样，井号“#”开头的语句都是“预处理语句”，在编译之前，预处理器会查找程序中所有的“ZERO”，并把它替换成0。这种宏定义的方式是保留的C语言特性，在C++中一般不推荐。

（2）使用const限定符

这种方式跟定义一个变量是一样的，只需要在变量的数据类型前再加上一个const关键字，这被称为“限定符”。

// 定义常量

const int Zero = 0;

// 不能修改常量值

//Zero = 10;

const修饰的对象一旦创建就不能改变，所以必须初始化。

跟使用 #define定义宏常量相比，const定义的常量有详细的数据类型，而且会在编译阶段进行安全检查，在运行时才完成替换，所以会更加安全和方便。

### 3.2 基本数据类型

定义变量时，不可或缺的一个要素就是数据类型。本质上讲，这就是为了实现计算需求，我们必须先定义好数据的样式，告诉计算机这些数据占多大空间，这就是所谓“数据类型”的含义。

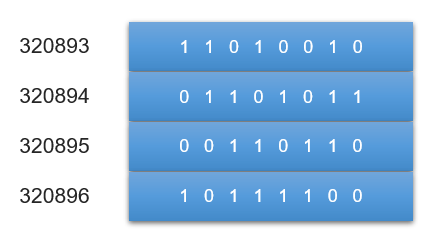
C++支持丰富的数据类型，它内置了一套基本数据类型，也为我们提供了自定义类型的机制。

接下来我们先介绍基本数据类型，主要包括算术类型和空类型（void）。其中算术类型又包含了整型和浮点型；而空类型不对应具体的值，只用在一些特定的场合，比如一个函数如果不返回任何值，我们可以让void作为它的返回类型。

#### 3.2.1 整型

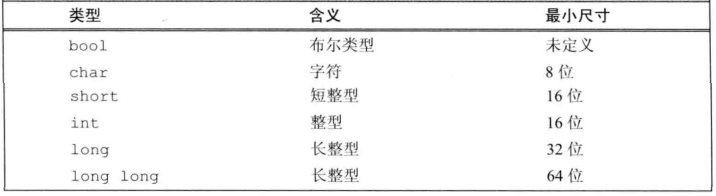
整型（integral type）本质上来讲就是表示整数的类型。

我们知道在计算机中，所有数据都是以二进制“0”“1”来表示的，每个叫做一位（bit）；计算机可寻址的内存最小单元是8位，也就是一个字节（Byte）。所以我们要访问的数据，都是保存在内存的一个个字节里的。



一个字节能表示的最大数是28 = 256，这对于很多应用来讲显然是不够的。不同的需求可能要表示的数的范围也不一样，所以C++中定义了多个整数类型，它们的区别就在于每种类型占据的内存空间大小不同。

C++定义的基本整型包括char、short、int、long，和C++ 11新增的long long类型，此外特殊的布尔类型bool本质上也是整型。



在C++中对它们占据的长度定义比较灵活，这样不同的计算机平台就可以有自己的实现了（这跟C是一样的）。由于char和bool相对特殊，我们先介绍其它四种。C++标准中对它们有最小长度的要求，比如：

* short类型至少为16位（2字节）
* int至少2字节，而且不能比short短
* long至少4字节，而且不能比int短
* long long至少8字节，而且不能比long短

现在一般系统中，short和long都选择最小长度，也就是short为16位、long为32位、long long为64位；而int则有不同选择。我们一般使用的电脑操作系统，比如Windows 7、Windows 10、Mac OS等等的实现中，int都是32位的。

所以short能表示的数有216 = 65536 个，考虑正负，能表示的范围就是-32768 ~ 32767；而int表示的数范围则为 - 231 ~ 231 - 1。（大概是正负20亿，足够用了）

#include<iostream>

using namespace std;

int main()

{

short a = 1;

cout << "a = " << a << endl;

cout << "a的长度为：" << sizeof(a) << endl;

int b;

cout << "b的长度为：" << sizeof(b) << endl;

long c;

cout << "c的长度为：" << sizeof(c) << endl;

long long d;

cout << "d的长度为：" << sizeof(d) << endl;

cin.get();

}

这里我们用到了sizeof，这是一个运算符，可以返回某个变量占用的字节数。我们可以看到，变量占用的空间大小只跟类型有关，跟变量具体的值无关。

#### 3.2.2 无符号整型

整型默认是可正可负的，如果我们只想表示正数和0，那么所能表示的范围就又会增大一倍。以16位的short为例，本来表示的范围是-32768 ~ 32767，如果不考虑负数，那么就可以表示0 ~ 65535。C++中，short、int、long、long long都有各自的“无符号”版本的类型，只要定义时在类型前加上unsigned就可以。

short a = 32768;

cout << "a = " << a << endl;

cout << "a的长度为：" << sizeof a << endl;

unsigned short a2 = 32768;

cout << "a2 = " << a2 << endl;

cout << "a2的长度为：" << sizeof a2 << endl;

上面的代码可以测试无符号数表示的范围。需要注意，当数值超出了整型能表示的范围，程序本身并不会报错，而是会让数值回到能表示的最小值；这种情况叫做“数据溢出”（或者“算术溢出”），写程序时一定要避免。

由于类型太多，在实际应用中使用整型可以只考虑三个原则：

* 一般的整数计算，全部用int；
* 如果数值超过了int的表示范围，用long long；
* 确定数值不可能为负，用无符号类型（比如统计人数、销售额等）；

#### 3.2.3 char类型

如果我们只需要处理很小的整数，也可以用另外一种特殊的整型类型——char，它通常只占一个字节（8位）。不过char类型一般并不用在整数计算，它更重要的用途是表示字符（character）。

计算机底层的数据都是二进制位表示的，这用来表示一个整数当然没有问题，可怎么表示字母呢？这就需要将常用的字母、以及一些特殊符号对应到一个个的数字上，然后保存下来，这就是“编码”的过程 。

最常用的字符编码集就是ASCII码，它用0~127表示了128个字符，这包括了所有的大小写字母、数字、标点符号、特殊符号以及一些计算机的控制符。比如字母“A”的编码是65，数字字符“0”的编码是48。

在程序中如果使用char类型的变量，我们会发现，打印出来就是一个字符；而它的底层是一个整数，也可以做整数计算。

char ch = 65;

cout << "65对应的字符为：" << ch << endl;

char ch2 = ch + 1;

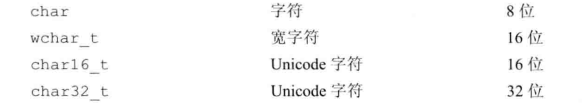
cout << "66对应的字符为：" << ch2 << endl;

char类型用来表示整数时，到底是有符号还是无符号呢？之前的所有整型，默认都是有符号的，而char并没有默认类型，而是需要C++编译器根据需要自己决定。

所以把char当做小整数时，有两种显式的定义方式：signed char 和 unsigned char；至于char定义出来的到底带不带符号，就看编译器的具体实现了。

另外， C++还对字符类型进行了“扩容”，提供了一种“宽字符”类型wchar\_t。wchar\_t会在底层对应另一种整型（比如short或者int），具体占几个字节要看系统中的实现。

wchar\_t会随着具体实现而变化，不够稳定；所以在C++11新标准中，还为Unicode字符集提供了专门的扩展字符类型：char16\_t和char32\_t，分别长16位和32位。



#### 3.2.4 bool类型

在程序中，往往需要针对某个条件做判断，结果只有两种：“成立”和“不成立”；如果用逻辑语言来描述，就是“真”和“假”。真值判断是二元的，所以在C语言中，可以很简单地用“1”表示“真”，“0”表示“假”。

C++支持C语言中的这种定义，同时为了让代码更容易理解，引入了一种新的数据类型——布尔类型bool。bool类型只有两个取值：true和false，这样就可以非常明确地表示逻辑真假了。bool类型通常占用8位（1个字节）。

bool bl = true;

cout << "bl = " << bl << endl;

cout << "bool类型长度为：" << sizeof bl << endl;

我们可以看到，true和false可以直接赋值给bool类型的变量，打印输出的时候，true就是1，false就是0，这跟C语言里的表示其实是一样的。

#### 3.2.5 浮点类型

跟整数对应，浮点数用来表示小数，主要有单精度float和双精度double两种类型，double的长度不会小于float。通常，float会占用4个字节（32位），而double会占用8个字节（64位）。此外，C++还提供了一种扩展的高精度类型long double，一般会占12或16个字节。

除了一般的小数，在C++中，还提供了另外一种浮点数的表示法，那就是科学计数法，也叫作“E表示法”。比如：5.98E24表示5.98×1024；9.11e-31表示9.11×10-31。

// 浮点类型

float f = 3.14;

double pi = 5.2e-3;

cout << "f = " << f << endl;

cout << "pi = " << pi << endl;

这就极大地扩展了我们能表示的数的范围。一般来讲，float至少有6位有效数字，double至少有15位有效数字。所以浮点类型不仅能表示小数，还可以表示（绝对值）非常大的整数。

（float和double具体能表示的范围，可以查找float.h这个头文件）

#### 3.2.6 字面值常量

我们在给一个变量赋值的时候，会直接写一个整数或者小数，这个数据就是显式定义的常量值，叫做“字面值常量”。每个字面值常量也需要计算机进行保存和处理，所以也都是有数据类型的。字面值的写法形式和具体值，就决定了它的类型。

（1）整型字面值

整型字面值就是我们直接写的一个整数，比如30。这是一个十进制数。而计算机底层是二进制的，所以还支持我们把一个数写成八进制和十六进制的形式。以0开头的整数表示八进制数；以0x或者0X开头的代表十六进制数。例如：

* 30 十进制数
* 036 八进制数
* 0x1E 十六进制数

这几个数本质上都是十进制的30，在计算机底层都是一样的。

在C++中，一个整型字面值，默认就是int类型，前提是数值在int能表示的范围内。如果超出int范围，那么就需要选择能够表示这个数的、长度最小的那个类型。

具体来说，对于十进制整型字面值，如果int不够那么选择long；还不够，就选择long long（不考虑无符号类型）；而八进制和十六进制字面值，则会优先用无符号类型unsigned int，不够的话再选择long，之后依次是unsigned long、long long和unsigned long long。

这看起来非常复杂，很容易出现莫名其妙的错误。所以一般我们在定义整型字面值时，会给它加上一个后缀，明确地告诉计算机这个字面值是什么类型。

* 默认什么都不加，是int类型；
* l或者L，表示long类型；
* ll或者LL，表示long long类型；
* u或者U，表示unsigned无符号类型；

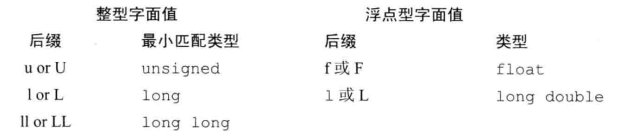
我们一般会用大写L，避免跟数字1混淆；而u可以和L或LL组合使用。例如9527uLL就表示这个数是unsigned long long类型。

（2）浮点型字面值

前面已经提到，可以用一般的小数或者科学计数法表示的数，来给浮点类型赋值，这样的数就都是“浮点型字面值”。浮点型字面值默认的类型是double。如果我们希望明确指定类型，也可以加上相应的后缀：

* f或者F，表示float类型
* l或者L，表示long double类型

这里因为本身数值是小数或者科学计数法表示，所以L不会跟long类型混淆。



（3）字符和字符串字面值

字符就是我们所说的字母、单个数字或者符号，字面值用单引号引起来表示。字符字面值默认的类型就是char，底层存储也是整型。

而多个字符组合在一起，就构成了“字符串”。字符串字面值是一串字符，用双引号引起来表示。

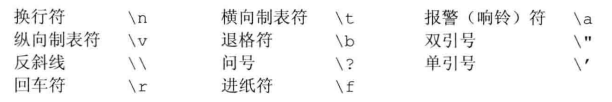
* ‘ A ’ 字符字面值
* “Hello World!” 字符串字面值

字符串是字符的组合，所以字符串字面值的类型，本质上是char类型构成的“数组”（array）。关于数组的介绍，我们会在后面章节详细展开。

* 转义字符

有一类比较特殊的字符字面值，我们是不能直接使用的。在ASCII码中我们看到，除去字母、数字外还有很多符号，其中有一些本身在C++语法中有特殊的用途，比如单引号和双引号；另外还有一些控制字符。如果我们想要使用它们，就需要进行“转义”，这就是“转义字符”。

C++中规定的转义字符有：



其中，经常用到的就是符号中的问号、双引号、单引号、反斜线，还有换行符和制表符。

// 转义字符

char tchar = '\n';

cout << "tchar = " << tchar << endl;

cout << "Hello World!\t\"Hello C++!\"" << endl;

（4）布尔字面值

布尔字面值非常简单，只有两个：true和false。

#### 3.2.7 类型转换

我们在使用字面值常量给变量赋值时会有一个问题，如果常量的值超出了变量类型能表示的范围，或者把一个浮点数赋值给整型变量，会发生什么？

这时程序会进行自动类型转换。也就是说，程序会自动将一个常量值，转换成变量的数据类型，然后赋值给变量。

// 1. 整数值赋给bool类型

bool b = 25; // b值为true，打印为1

// 2. bool类型赋值给算术整型

short s = false; // s值为0

// 3. 浮点数赋给整数类型

int i = 3.14; // i值为3

// 4. 整数值赋给浮点类型

float f = 10; // f值为10.0，打印为10

// 5. 赋值超出整型范围

unsigned short us = 65536; // us值为0

s = 32768; // s值为-32768

转换规则可以总结如下：

* 非布尔类型的算术值赋给布尔类型，初始值为0则结果为 false , 否则结果为true ；
* 布尔值赋给非布尔类型，初始值为 false 则结果为0，初始值为 true 则结果为1；
* 浮点数赋给整数类型，只保留浮点数中的整数部分，会带来精度丢失；
* 整数值赋给浮点类型，小数部分记为0。如果保存整数需要的空间超过了浮点类型的容量，可能会有精度丢失。
* 给无符号类型赋值，如果超出它表示范围，结果是初始值对无符号类型能表示的数值总数取模后的余数。
* 给有符号类型赋值，如果超出它表示范围，结果是未定义的（ undefined ）。此时，程序可能继续工作，也可能崩溃。

C++中的数据类型转换，是一个比较复杂的话题。我们这里先了解一下变量赋值时的自动类型转换，关于更加复杂的转换，我们会在下一章继续介绍。

## 四、运算符

有了数据之后，就可以对数据对象进行各种计算了。在编程语言中，可以通过“运算符”来表示想要进行的计算。

### 4.1 表达式和运算符

#### 4.1.1 基本概念

在程序中，一个或多个运算对象的组合叫做“表达式”（expression），我们可以把它看成用来做计算的“式子”。对一个表达式进行计算，可以得到一个结果，有时也把它叫做表达式的值。

前面讲到的字面值常量和变量，就是最简单的表达式；表达式的结果就是字面值和变量的值。而多个字面值和变量，可以通过一些符号连接组合在一起，表示进行相应的计算，这就可以得到更加复杂的表达式，比如 a + 1。像“+”这些符号就被叫做“运算符”（operator）。

C++中定义的运算符，可以是像“+”这样连接两个对象，称为“二元运算符”；也可以只作用于一个对象，称为“一元运算符”。另外，还有一个比较特殊的运算符可以作用于三个对象，那就是三元运算符了。

#### 4.1.2 运算优先级和结合律

如果在一个表达式中，使用多个运算符组合了多个运算对象，就构成了更加复杂的“复合表达式”，比如 a + 1 - b。对于复合表达式，很显然我们应该分步来做计算；而计算顺序，是由所谓的“优先级”和“结合律”确定的。

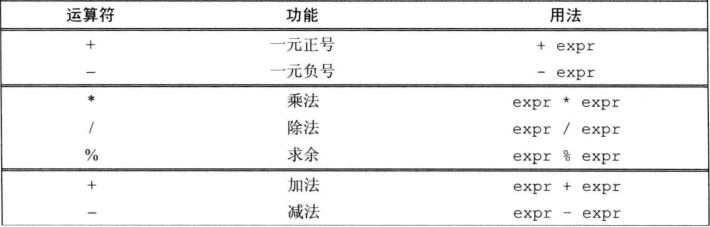
简单来说，就是对不同的运算符赋予不同的“优先级”，我们会优先执行高优先级的运算、再执行低优先级的运算。如果优先级相同，就按照“结合律”来决定执行顺序。这其实跟数学的综合算式是一样的，我们会定义乘除的优先级要高于加减，平级运算从左往右，所以对于算式：

1 + 2 – 3 × 4

我们会先计算高优先级的 3×4，然后按照从左到右的结合顺序计算1+2，最后做减法。另外，如果有括号，那就要先把括起来的部分当成一个整体先做计算，然后再考虑括号外的结合顺序，这一点在C++表达式中同样适用。

### 4.2 算术运算

最简单的运算符，就是表示算术计算的加减乘除，这一类被称为“算术运算符”。C++支持的算术运算符如下：



这里需要注意的是，同一个运算符，在不同的场合可能表达不同的含义。比如“-”，可以是“减号”也可以是“负号”：如果直接放在一个表达式前面，就是对表达式的结果取负数，这是一元运算符；如果连接两个表达式，就是两者结果相减，是二元运算符。

算术运算符相关规则如下：

* 一元运算符（正负号）优先级最高；接下来是乘、除和取余；最后是加减；
* 算术运算符满足左结合律，也就是说相同优先级的运算符，将从左到右按顺序进行组合；
* 算术运算符可以用来处理任意算术类型的数据对象；
* 不同类型的数据对象进行计算时，较小的整数类型会被“提升”为较大的类型，最终转换成同一类型进行计算；
* 对于除法运算“/”，执行计算的结果跟操作数的类型有关。如果它的两个操作数（也就是被除数和除数）都是整数，那么得到的结果也只能是整数，小数部分会直接舍弃，这叫“整数除法”；当至少有一个操作数是浮点数时，结果就会是浮点数，保留小数部分；
* 对于取余运算“%”（或者叫“取模”），两个操作数必须是整数类型；

// 除法

int a = 20, b = 6;

cout << " a / b = " << a / b << endl;

cout << " -a / b = " << -a / b << endl; // 负数向0取整

float a2 = 20;

cout << " a2 / b = " << a2 / b << endl;

// 取模

cout << " a % b = " << a % b << endl;

cout << " -a % b = " << -a % b << endl;

在这里，同样是除法运算符“/”,针对不同类型的数据对象，其实会做不同的处理。使用相同的符号、根据上下文来执行不同操作，这是C++提供的一大特色功能，叫做“运算符重载”（operator overloading）。

### 4.3 赋值

将一个表达式的结果，传递给某个数据对象保存起来，这个过程叫做“赋值”。

#### 4.3.1 赋值运算符

在C++中，用等号“=”表示一个赋值操作，这里的“=”就是赋值运算符。需要注意的是，赋值运算符的左边，必须是一个可修改的数据对象，比如假设我们已经定义了一个int类型的变量a，那么

a = 1;

这样赋值是对的，但

1 = a;

就是错误的。因为a是一个变量，可以赋值；而 1只是一个字面值常量，不能再对它赋值。

int a, b;

a = 1;

//1 = a; // 错误：表达式必须是可修改的左值

a = b + 5;

//b + 5 = a; // 错误：表达式必须是可修改的左值

const int c = 10;

//c = a + b; // 错误：表达式必须是可修改的左值

所以像变量a这样的可以赋值的运算对象，在C++中被叫做“左值”（lvalue）；对应的，放在赋值语句右面的表达式就是“右值”（rvalue）。

赋值运算有以下一些规则：

* 赋值运算的结果，就是它左侧的运算对象；结果的类型就是左侧运算对象的类型；
* 如果赋值运算符两侧对象类型不同，就把右侧的对象转换成左侧对象的类型；
* C++ 11新标准提供了一种新的语法：用花括号{}括起来的数值列表，可以作为赋值右侧对象。这样就可以非常方便地对一个数组赋值了；
* 赋值运算满足右结合律。也就是说可以在一条语句中连续赋值，结合顺序是从右到左；
* 赋值运算符优先级较低，一般都会先执行其它运算符，最后做赋值；

a = {2};

int arr[] = {1,2,3,4,5}; // 用花括号对数组赋值

a = b = 20; // 连续赋值

#### 4.3.2 复合赋值运算符

实际应用中，我们经常需要把一次计算的结果，再赋值给参与运算的某一个变量。最简单的例子就是多个数求和，比如我们要计算a、b、c的和，那么可以专门定义一个变量sum，用来保存求和结果：

int sum = a; // 初始值是a

sum = sum + b; // 叠加b

sum = sum + c; // 叠加c

要注意赋值运算符“=”完全不是数学上“等于”的意思，所以上面的赋值语句sum = sum + b; 说的是“计算sum + b的结果，然后把它再赋值给sum”。

为了更加简洁，C++提供了一类特殊的赋值运算符，可以把要执行的算术运算“+”跟赋值“=”结合在一起，用一个运算符“+=”来表示；这就是“复合赋值运算符”。

复合赋值一般结合的是算术运算符或者位运算符。每种运算符都有对应的组合形式：



关于位运算符，我们会在稍后介绍。

这样上面的代码可以改写为：

int sum = a; // 初始值是a

sum += b; // 完全等价于 sum = sum + b;

sum += c;

#### 4.3.3 递增递减运算符

C++为数据对象的“加一”“减一”操作，提供了更加简洁的表达方式，这就是递增和递减运算符（也叫“自增”“自减”运算符）。“递增”用两个加号“++”表示，表示“对象值加一，再赋值给原对象”；“递减”则用两个减号“--”表示。

++a; // a递增，相当于 a += 1;

--b; // b递减，相当于 b -= 1;

递增递减运算符各自有两种形式：“前置”和“后置”，也就是说写成“++a”和“a++”都是可以的。它们都表示“a = a + 1”，区别在于表达式返回的结果不同：

前置时，对象先加1，再将更新之后的对象值作为结果返回；

后置时，对象先将原始值作为结果返回，再加1；

这要特别注意：如果我们单独使用递增递减运算符，那前置后置效果都一样；但如果运算结果还要进一步做计算，两者就有明显不同了。

int i = 0, j;

j = ++i; // i = 1，j = 1

j = i--; // i = 0, j = 1

在实际应用中，一般都是希望用改变之后的对象值；所以为了避免混淆，我们通常会统一使用前置的写法。

### 4.4 关系和逻辑运算

在程序中，不可缺少的一类运算就是逻辑和关系运算，因为我们往往需要定义“在某种条件发生时，执行某种操作”。判断条件是否发生，这就是一个典型的逻辑判断；得到的结果或者为“真”（true），或者为“假”。很显然，这类运算的结果应该是布尔类型。

#### 4.4.1 关系运算符

最简单的一种条件，就是判断两个算术对象的大小关系，对应的运算符称为“关系运算符”。包括：大于“>”、小于“<”、等于“==”、不等于“!=”、大于等于“>=”、小于等于“<=”。



这里要注意区分的是，在C++语法中一个等号“=”表示的是赋值，两个等号“==”才是真正的“等于”。

1 < 2; // true

3 >= 5; // false

10 == 4 + 6; // true

(10 != 4) + 6; // 7

关系运算符的相关规则：

* 算术运算符的优先级高于关系运算符，而如果加上括号就可以调整计算顺序；
* 关系运算符的返回值为布尔类型，如果参与算术计算，true的值为1，false的值为0；

#### 4.4.2 逻辑运算符

一个关系运算符的结果是一个布尔类型（ture或者false），就可以表示一个条件的判断；如果需要多个条件的叠加，就可以用逻辑“与或非”将这些布尔类型组合起来。这样的运算符叫做“逻辑运算符”。

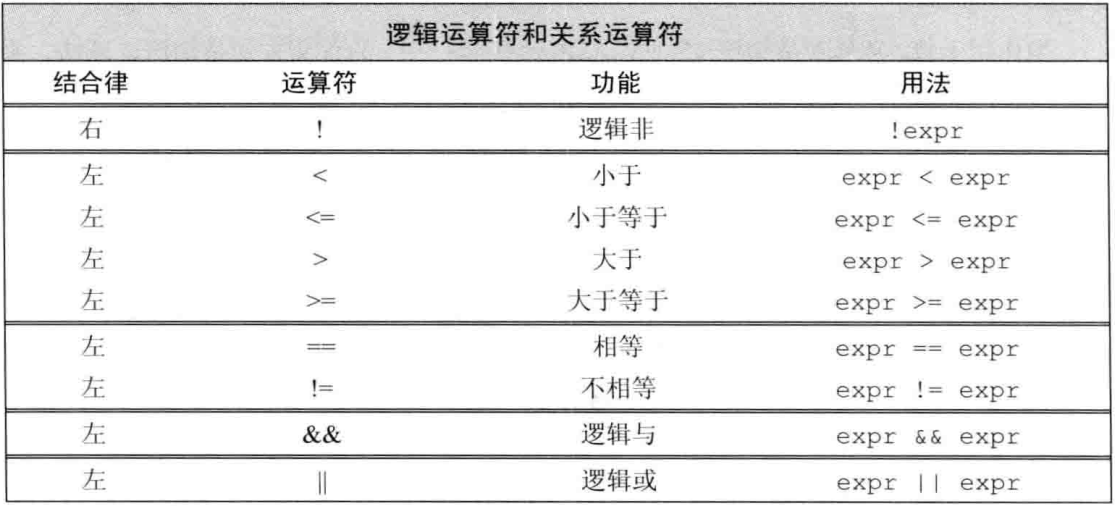
* 逻辑非（！）：一元运算符，将运算对象的值取反后返回，真值反转；
* 逻辑与（&&）：二元运算符，两个运算对象都为true时结果为true，否则结果为false；
* 逻辑或（||）：二元运算符，两个运算对象只要有一个为true结果就为true，都为false则结果为false；

1 < 2 && 3 >= 5; // false

1 < 2 || 3 >= 5; // true

!(1 < 2 || 3 >= 5); // false

我们可以把逻辑运算符和关系运算符的用法、优先级和结合律总结如下（从上到下优先级递减）：



这里需要注意的规则有：

* 如果将一个算术类型的对象作为逻辑运算符的操作数，那么值为0表示false，非0值表示true；
* 逻辑与和逻辑或有两个运算对象，在计算时都是先求左侧对象的值，再求右侧对象的值；如果左侧对象的值已经能决定最终结果，那么右侧就不会执行计算：这种策略叫做“**短路求值**”；

i = -1;

1 < 2 && ++i; // false

cout << " i = " << i << endl; // i = 0

1 < 2 || ++i; // true

cout << " i = " << i << endl; // i = 0

#### 4.4.3 条件运算符

C++还从C语言继承了一个特殊的运算符，叫做“条件运算符”。它由“?”和“:”两个符号组成，需要三个运算表达式，形式如下：

*条件判断表达式 ? 表达式1 : 表达式2*

它的含义是：计算条件判断表达式的值，如果为true就执行表达式1，返回求值结果；如果为false则跳过表达式1，执行表达式2，返回求值结果。这也是C++中唯一的一个三元运算符。

i = 0;

cout << ((1 < 2 && ++i) ? "true" : "false") << endl;

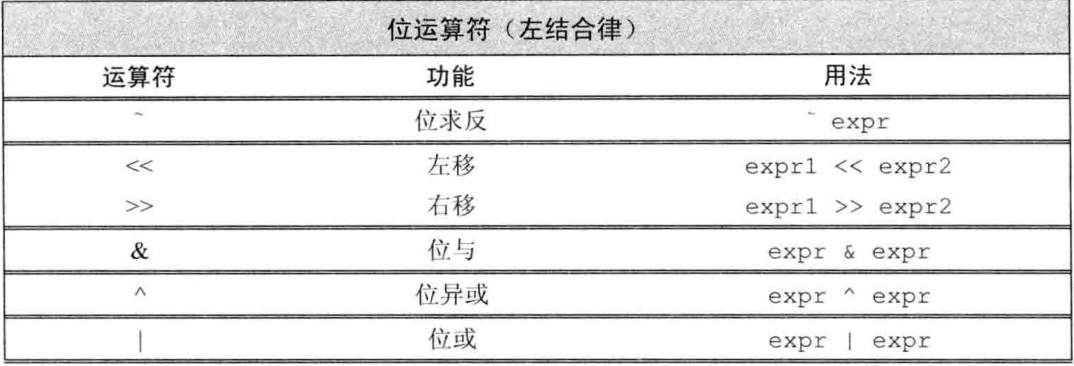
* 条件运算符的优先级比较低，所以输出的时候需要加上括号
* 条件运算符满足右结合律

事实上，条件运算符等同于流程控制中的分支语句if...else...，只用一条语句就可以实现按条件分支处理，这就让代码更加简洁。关于分支语句，我们会在后面详细介绍。

### 4.5 位运算符

之前介绍的所有运算符，主要都是针对算术类型的数据对象进行操作的；所有的算术类型，占用的空间都是以字节（byte，8位）作为单位来衡量的。在C++中，还有一类非常底层的运算符，可以直接操作到具体的每一位（bit）数据，这就是“位运算符”。

位运算符可以分为两大类：移位运算符，和位逻辑运算符。下面列出了所有位运算符的优先级和用法。



#### 4.5.1移位运算符

算术类型的数据对象，都可以看做是一组“位”的集合。那么利用“移位运算符”，就可以让运算对象的所有位，整体移动指定的位数。

移位运算符有两种：左移运算符“<<”和右移运算符“>>”。这个符号我们并不陌生，之前做输入输出操作的时候用的就是它，不过那是标准IO库里定义的运算符重载版本。

下面是移位运算符的一个具体案例：



* 较小的整数类型（char、short以及bool）会自动提升成int类型再做移位，得到的结果也是int类型
* 左移运算符“<<”将操作数左移之后，在右侧补0；
* 右移运算符“>>”将操作数右移之后，对于无符号数就在左侧补0；对于有符号数的操作则要看运行的机器环境，有可能补符号位，也有可能直接补0；
* 由于有符号数右移结果不确定，一般只对无符号数执行位移操作；

unsigned char bits = 0xb5; // 181

cout << hex; // 以十六进制显示

cout << "0xb5 左移2位：" << (bits << 2) << endl; // 0x 0000 02d4

cout << "0xb5 左移8位：" << (bits << 8) << endl; // 0x 0000 b500

cout << "0xb5 左移31位：" << (bits << 31) << endl; // 0x 8000 0000

cout << "0xb5 右移3位：" << (bits >> 3) << endl; // 0x 0000 0016

cout << dec;

cout << (200 << 3) << endl; // 乘8操作

cout << (-100 >> 2) << endl; // 除4操作，一般右移是补符号位

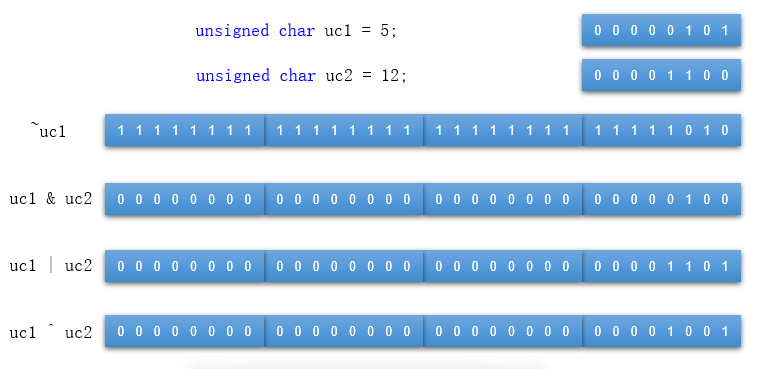
#### 4.5.2 位逻辑运算符

计算机存储的每一个“位”（bit）都是二进制的，有0和1两种取值，这跟布尔类型的真值表达非常类似。于是自然可以想到，两个位上的“0”或“1”都可以执行类似逻辑运算的操作。

位逻辑运算符有：按位取反“~”，位与“&”，位或“|”和位异或“^”。

* 按位取反“~”：一元运算符，类似逻辑非。对每个位取反值，也就是把1置为0、0置为1；
* 位与“&”：二元运算符，类似逻辑与。两个数对应位上都为1，结果对应位为1；否则结果对应位为0；
* 位或“|”：二元运算符，类似逻辑或。两个数对应位上只要有1，结果对应位就为1；如果全为0则结果对应位为0；
* 位异或“^”：两个数对应位相同，则结果对应位为0；不同则结果对应位为0；

下面是位逻辑运算符的一个具体案例：



// 位逻辑运算

cout << (~5) << endl; // ~ (0... 0000 0101) = 1... 1111 1010, -6

cout << (5 & 12) << endl; // 0101 & 1100 = 0100, 4

cout << (5 | 12) << endl; // 0101 | 1100 = 1101, 13

cout << (5 ^ 12) << endl; // 0101 & 1100 = 1001, 9

### 4.5 类型转换

在C++中，不同类型的数据对象，是可以放在一起做计算的。这就要求必须有一个机制，能让有关联的两种类型可以互相转换。在上一章已经介绍过变量赋值时的自动类型转换，接下来我们会对类型转换做更详细的展开。

#### 4.5.1 隐式类型转换

大多数情况，C++编译器可以自动对类型进行转换，不需要我们干涉，这种方式叫做“隐式类型转换”。

隐式类型转换主要发生在算术类型之间，基本思路就是将长度较小的类型转换成较大的类型，这样可以避免丢失精度。隐式类型转换不仅可以在变量赋值时发生，也可以在运算表达式中出现。例如：

short s = 15.2 + 20;

cout << " s = " << s << endl; // s = 35

cout << " 15.2 + 20 结果长度为：" << sizeof(15.2 + 20) << endl;

cout << " s 长度为：" << sizeof(s) << endl;

对于这条赋值语句，右侧是两个字面值常量相加，而且类型不同：15.2是double类型，20是int类型。当它们相加时，会将int类型的20转换为double类型，然后执行double的加法操作，得到35.2。

这个结果用来初始化变量s，由于s是short类型，所以还会把double类型的结果35.2再去掉小数部分，转换成short类型的35。所以s最终的值为35。

隐式类型转换的一般规则可以总结如下：

* 在大多数算术运算中，较小的整数类型(如bool、char、short)都会转换成int类型。这叫做“整数提升”；（而对于wchar\_t等较大的扩展字符类型，则根据需要转换成int、unsigned int、long、unsigned long、long long、unsigned long long中能容纳它的最小类型）
* 当表达式中有整型也有浮点型时，整数值会转换成相应的浮点类型；
* 在条件判断语句中，其它整数类型会转换成布尔类型，即0为false、非0为true；
* 初始化变量时，初始值转换成变量的类型；
* 在赋值语句中，右侧对象的值会转换成左侧对象的类型；

此外，要尽量避免将较大类型的值赋给较小类型的变量，这样很容易出现精度丢失或者数据溢出。

s = 32767;

cout << " s + 1 = " << s + 1 << endl;

short s2 = s + 1;

cout << " s2 = " << s2 << endl;

另外还要注意，如果希望判断一个整型变量a是否在某个范围（0， 100）内，不能直接写：0 < a < 100;

由于小于运算符“<”满足左结合律，要先计算0 < a，得到一个布尔类型的结果，再跟后面的100进行比较。此时布尔类型做整数提升，不管值是真（1）还是假（0），都会满足 < 100 的判断，因此最终结果一定是true。

要想得到正确的结果，需要将两次关系判断拆开，写成逻辑与的关系。

a = -1;

0 < a < 100; // 不论a取什么值，总是true

0 < a && a < 100; // false

#### 4.5.2 强制类型转换

除去自动进行的隐式类型转换，我们也可以显式地要求编译器对数据对象的类型进行更改。这种转换叫做“强制类型转换”（cast）。

比如对于除法运算，我们知道整数除法和浮点数除法是不同的。如果希望对一组整数求一个平均数，直接相加后除以个数是无法得到想要的结果的：

// 求平均数

int total = 20, num = 6;

double avg = total / num;

cout << " avg = " << avg << endl; // avg = 3

因为两个int类型的数相除，执行的是整数除法，得到3；再转换成double类型对avg做初始化，得到是3.0。如果想要更准确的结果，就必须将int类型强制转换成double，做浮点数除法。

C++中可以使用不同的方式进行强制类型转换。

（1）C语言风格

最经典的强转方式来自C语言，格式如下：

(*类型名称*) *值*

把要强制转成的类型，用一个小括号括起来，放到要转换的对象值前面就可以了。

（2）C++函数调用风格

这种方式跟C语言的强转类似，只不过看起来更像是调用了一个函数：

*类型名称* (*值*)

要转成的类型名就像是一个函数，调用的时候，后面小括号里是传递给它的参数。

（3）C++强制类型转换运算符

C++还引入了4个强制类型转换运算符，这种新的转换方式比前两种传统方式要求更为严格。通常在类型转换中用到的运算符是static\_cast，用法如下：

static\_cast<*类型名称*> (*值*)

static\_cast运算符后要跟一个尖括号，里面是要转换成的类型。

有了这些强转的方式，就可以解决之前求平均数的问题了：

// C语言风格

cout << " avg = " << (double) total / num << endl;

// C++函数风格

cout << " avg = " << double (total) / num << endl;

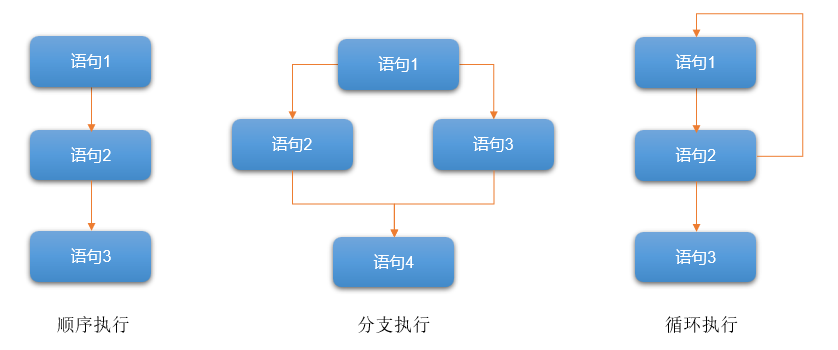
// C++强转运算符

cout << " avg = " << static\_cast<double>(total) / num << endl;

强制类型转换会干扰正常的类型检查，带来很多风险，所以通常要尽量避免使用强制类型转换。

## 五、流程控制语句

C++程序执行的流程结构可以有三种：顺序、分支和循环。除了最简单的顺序结构是默认的，分支和循环都需要使用专门的“流程控制语句”来定义。



### 5.1 语句

C++中表示一步操作的一句代码，就叫做“语句”（statement），大多数语句都是以分号“;”结尾的。C++程序运行的过程，其实就是找到主函数，然后从上到下顺序执行每条语句的过程。

#### 5.1.1 简单语句

使用各种运算符，作用到数据对象上，就得到了“表达式”；一个表达式末尾加上分号，就构成了“表达式语句”（expression statement）。

表达式语句表示，要执行表达式的计算过程，并且丢弃最终返回的结果。

int a = 0; // 变量定义并初始化语句

a + 1; // 算术表达式语句，无意义

++a; // 递增语句，a的值变为1

cout << " a = " << a << endl; // 输出语句

其中第二行a + 1; 是没什么意义的，因为它只是执行了加法操作，却没有把结果保存下来（赋值给别的变量），a的值也没有改变，也没有任何附带效果（比如最后一句的输出）。

最简单的语句，其实是“空语句”，就是只有一个分号的语句：

; // 空语句

这看起来好像没什么用。不过有时候，可能程序在语法上需要有一条语句，而逻辑上什么都不用做；这时就应该用一条空语句来填充。

初学C++，一定不要忘记语句末尾的分号；当然，对于不需要分号的场景，也尽量避免多写分号。

#### 5.1.2 复合语句

简单语句从上到下按顺序依次执行，这非常符合我们对计算机运行的预期。但是很多场景下，简单的顺序结构远远不能满足逻辑需要：比如我们可能需要按照条件判断，做程序的分支执行；也可能需要将一段代码循环执行多次。这就需要一些“流程控制语句”（比如if、while、for等）来表达更加复杂的操作了。

而对于流程控制语句，逻辑上来说只是一条语句；事实上却可能包含了多条语句、复杂的操作。这就需要用一个花括号“{}”，把这一组语句序列包成一个整体，叫做“复合语句”（compound statement），也叫做“块”（block）。

int i = 0;

while (i < 5) {

int a = i;

++i;

}

这里的while表示一个循环，后面只能跟要循环执行的一条语句；如果我们想写两条语句，就要用花括号括起来，构成“块”。

对于复合语句（块）需要注意：

* 花括号后面不需要再加分号，块本身就是一条语句；
* 块内可以声明变量，变量的作用域仅限于块内部；
* 只有一对花括号、内部没有任何语句的块叫做“空块”，等价于空语句；

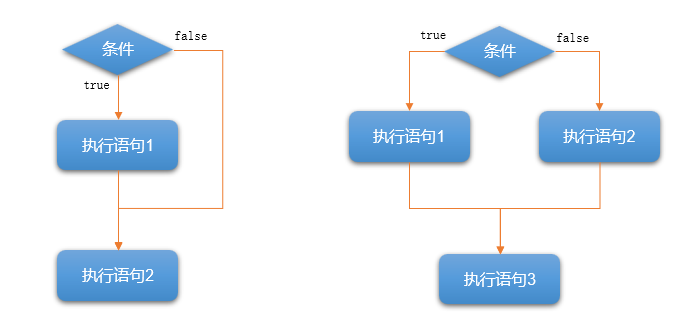
### 5.2 条件分支

很多情况下，我们为程序的执行会提供“岔路”的选择机会。一般都是：满足某种条件就执行A操作，满足另一种条件就执行B操作……这样的程序结构叫做“条件分支”。

C++提供了两种按条件分支执行的控制语句：if和switch。

#### 5.2.1 if

if语句主要就是判断一个条件是否为真（true），如果为真就执行下面的语句，如果为假则跳过。具体形式可以分为两种：一种是单独一个if，一般称为“单分支”；另一种是if … else …，称为“双分支”。



（1）单分支

单分支是最简单的if用法，判断的条件用小括号括起来跟在if后面，然后是如果条件为真要执行的语句。基本形式为：

if (*条件判断*)

*语句*

如果条件为假，那么这段代码就会被完全跳过。

我们可以举一个简单示例，判断输入的年龄数值，然后输出一句欢迎词：

#include<iostream>

using namespace std;

int main()

{

cout << "请输入您的芳龄：" << endl;

int age;

cin >> age;

if ( age >= 18 )

{

cout << "欢迎您，成年人！" << endl;

}

cin.get();

cin.get();

}

通常会用一个花括号将if后面的语句括起来，成为一个“块”。现在块里只有一条语句，所以花括号是可以省略的：

if ( age >= 18 )

cout << "欢迎您，成年人！" << endl;

如果要执行的是多条语句，花括号就不能省略；否则if后面其实就只有第一条语句。为了避免漏掉括号出现错误，一般if后面都会使用花括号。

（2）双分支

双分支就是在if分支的基础上，加了else分支；条件为真就执行if后面的语句，条件为假就执行else后面的语句。基本形势如下：

if (*条件判断*)

*语句1*

else

*语句2*

if分支和else分支，两者肯定会选择一个执行。

我们可以在之前程序的基础上，增加一个else分支：

if ( age >= 18 )

{

cout << "欢迎您，成年人！" << endl;

}

else

{

cout << "本程序不欢迎未成年人！" << endl;

}

我们可以回忆起来，之前介绍过的唯一一个三元运算符——条件运算符，其实就可以实现类似的功能。所以条件运算符可以认为是if … else 的一个语法糖。

以下两条语句跟上面的if…else是等价的：

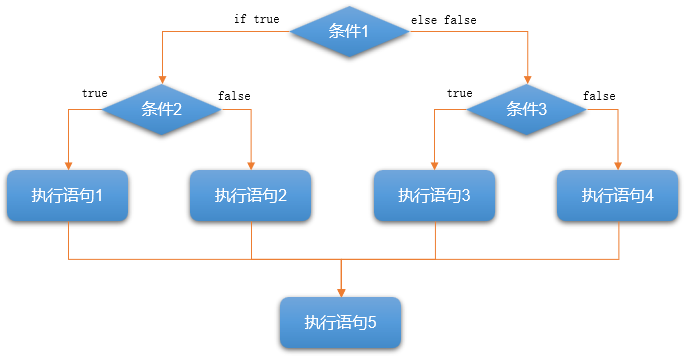
// 条件运算符的等价写法

age >= 18 ? cout << "欢迎您，成年人！" << endl : cout << "本程序不欢迎未成年人！" << endl;

cout << (age >= 18 ? "欢迎您，成年人！" : "本程序不欢迎未成年人！") << endl;

（3）嵌套分支（多分支）

程序中的分支有可能不只两个，这时就需要对if分支或者else分支再做条件判断和拆分了，这就是“嵌套分支”。



简单来说，就是if或者else分支的语句块里，继续使用if或者if…else按条件进行分支。这是一种“分层”的条件判断。

if ( age >= 18 )

{

cout << "欢迎您，成年人！" << endl;

if (age < 35)

{

cout << "加油，年轻人！" << endl;

}

}

else

{

cout << "本程序不欢迎未成年人！" << endl;

if (age >= 12)

{

cout << "少年，好好学习！" << endl;

}

else

{

cout << "小朋友，别玩电脑！" << endl;

}

}

嵌套分支如果比较多，代码的可读性会大幅降低。所以还有一种更加简单的嵌套分支写法，那就是if … else if …，具体形式如下：

if (*条件判断1*)

*语句1*

else if (*条件判断2*)

*语句2*

else if (*条件判断3*)

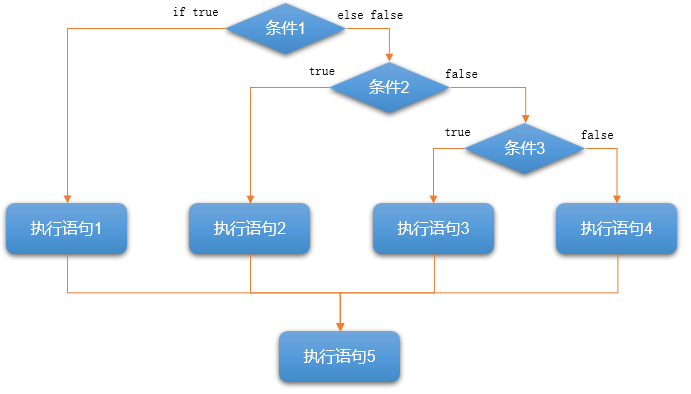
*语句3*

*…*

else

*语句n*

这种分支的嵌套，本质上只能对else分支进行，而且只能在最底层的分支中才能执行语句。



测试代码如下：

if (age < 12) {

cout << "小朋友，别玩电脑！" << endl;

}

else if (age < 18)

{

cout << "少年，好好学习！" << endl;

}

else if (age < 35)

{

cout << "加油，年轻人！" << endl;

}

else if (age < 60)

{

cout << "加油，中年人！" << endl;

}

else

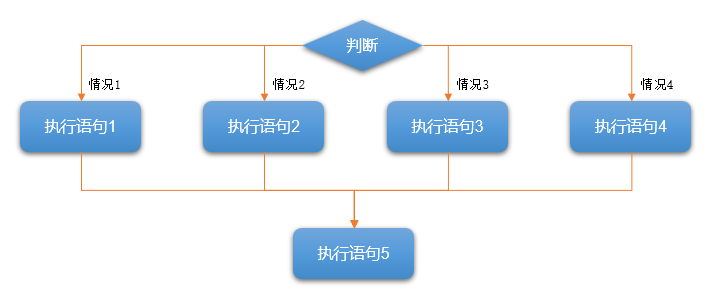
{

cout << "好好休息，老年人！" << endl;

}

#### 5.2.2 switch

在一些应用场景中，要判断的条件可能不是范围，而是固定的几个值。比如考试成绩只分“A”“B”“C”“D”四个档位，分别代表“优秀”“良好”“及格”“不及格”。



这个时候如果用if … else会显得非常繁琐，而swith语句就是专门为了这种分支场景设计的。

switch语法基本形式如下：

swith (*表达式*){

case *值1*:

*语句1*

break;

case *值2*:

*语句2*

break;

…

default:

*语句n*

break;

}

这里switch后面的括号里是一个表达式，对它求值，然后转换成整数类型跟下面每个case后面的值做比较；如果相等，就进入这个case指定的分支，执行后面的语句，直到swith语句结束或者遇到break退出。需要注意的是：

* case关键字和后面对应的值，合起来叫做一个“case标签”；case标签必须是一个**整型**的常量表达式；
* 任何两个case标签不能相同；
* break语句的作用是“中断”，会直接跳转到switch语句结构的外面；
* 如果没有break语句，那么匹配某个case标签之后，程序会从上到下一直执行下去；这会执行多个标签下面的语句，可能发生错误；
* 如果没有匹配上任何case标签的值，程序会执行default标签后面的语句；default是可选的，表示“默认要执行的操作”。

我们可以利用swith写一个判断考试成绩档位，输入一句相应的话：

#include<iostream>

using namespace std;

int main()

{

cout << "请输入您的成绩：" << endl;

char score;

cin >> score;

switch (score)

{

case 'A':

cout << "成绩优秀！" << endl;

break;

case 'B':

cout << "成绩良好！" << endl;

break;

case 'C':

cout << "恭喜！及格了！" << endl;

break;

case 'D':

cout << "欢迎下次再来！" << endl;

break;

default:

cout << "错误的成绩输入！" << endl;

break;

}

cin.get();

cin.get();

}

### 5.3 循环

可以重复执行一组操作的语句叫做“循环”，有时也叫作“迭代”。循环一般不能无限进行下去，所以会设置一个终止的判断条件。

C++中的循环语句，有while、do while和for三种。

#### 5.3.1 while

while只需要给定一个判断条件，只要条件为真，就重复地执行语句。形式如下：

while (*条件*)

*语句*

要执行的语句往往会有多条，这就需要用花括号将它们括起来。这个块一般被称为“循环体”。

一般来说，用来控制while循环的条件中一定会包含变量，通常叫做“循环变量”；而它或者在条件中变化，或者在循环体中变化，这样才能保证循环能够终止退出。

比如我们可以用一个循环输出10次“Hello World”，并且打印出当前循环次数：

#include<iostream>

using namespace std;

int main()

{

cout << "循环开始...\n" << endl;

int i = 1;

while (i <= 10)

{

cout << "Hello World!" << endl;

cout << "现在是第" << i << "次循环\n" << endl;

++i;

}

cout << "循环结束!" << endl;

cin.get();

}

这里需要注意，循环体最后的 ++i一定不能漏掉。如果没有这条语句，i的值就不会更改，循环就永远不会退出。

#### 5.3.2 do while

do while和while非常类似，区别在于do while是先执行循环体中的语句，然后再检查条件是否满足。所以do while至少会执行一次循环体。

do while语法形式如下：

do

*语句*

while (*条件*)

我们可以接着之前while循环的代码继续测试：

do

{

cout << "现在是倒数第" << --i << "次循环" << endl;

cout << "GoodBye World!\n" << endl;

} while (i > 1);

由于之前的变量i已经做了10次递增，因此do wihle开始时i的值为11。进入循环体直接输出内容，每次i递减1，直到i = 1时退出循环。

#### 5.3.3 for

通过while和do while可以总结出来，一个循环主要有这样几个要素：

* 一个条件，用来控制循环退出；
* 一个循环体，用来定义循环要执行的操作；

而一般情况下，我们都是通过一个循环变量来控制条件的，这个变量需要随着循环迭代次数的增加而变化。while和do while的循环变量，都是在循环体外单独定义的。

for是用法更加明确的循环语句。它可以把循环变量的定义、循环条件以及循环变量的改变都放在一起，统一声明出来。

（1）经典for循环

for循环的经典语法形式是：

for (*初始化语句*; *条件*; *表达式*)

*语句*

关键字for和它后面括号里的部分，叫做“for语句头”。

for语句头中有三部分，用分号分隔，主要作用是：

* 初始化语句负责初始化一个变量，这个变量值会随着循环迭代而改变，一般就是“循环变量”；
* 中间的条件是控制循环执行的关键，为真则执行下面的循环体语句，为假则退出。条件一般会以循环变量作为判断标准；
* 最后的表达式会在本次循环完成之后再执行，一般会对循环变量进行更改；

这三个部分并不是必要的，根据需要都可以进行省略。如果省略某个部分，需要保留分号表示这是一个空语句。

我们可以用for循环语句，实现之前输出10次“Hello World”的需求：

#include<iostream>

using namespace std;

int main()

{

for (int i = 1; i <= 10; i++)

{

cout << "Hello World!" << endl;

cout << "现在是第" << i << "次循环！\n" << endl;

}

cin.get();

}

（2）范围for循环

C++ 11新标准引入了一种更加简单的for循环，这种语句可以直接遍历一个序列的所有元素。这种for循环叫做“范围for循环”。语法形式如下：

for (声明: *序列表达式*)

*语句*

这里for语句头中的内容就很简单了，只需要声明一个变量，后面跟上一个冒号（注意不是分号），再跟上一个序列的表达式就可以了。所谓“序列”，其实就是一组相同类型的数据对象排成了一列来统一处理；所以这个声明的意思，其实就是从序列中依次取出所有元素，每次都赋值给这个变量。

所以范围for循环的特点就是，不需要循环变量，直接就可以访问序列中的所有元素。

// 范围for循环

for (int num : {3, 6, 8, 10})

{

cout << "序列中现在的数据是：" << num << endl;

}

这里用花括号把一组数括起来，就构成了最简单的序列：{3, 6, 8, 10}。后面将要介绍的数组，以及vector、string等类型的对象，也都是序列。

#### 5.3.4 循环嵌套

循环语句和分支语句一样，也是可以进行嵌套的。具体可以while循环中嵌套while，可以for循环中嵌套for，也可以while、do while和for混合嵌套。因为for的循环变量定义更明确，所以一般用for的循环嵌套会多一些。

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 5; j++)

{

cout << "Hello World! i = " << i << ", j = " << j << endl;

}

}

循环嵌套之后，内层语句执行的次数，将是外层循环次数和内层循环次数的乘积。这会带来大量的时间消耗，使程序运行变慢，所以使用嵌套循环要非常谨慎。

下面是一个使用双重for循环，打印输出“九九乘法表”的例子。

#include<iostream>

using namespace std;

int main()

{

// i表示行数，j表示列数

for (int i = 1; i < 10; i++)

{

for (int j = 1; j <= i; j++) {

cout << j << " × " << i << " = " << i \* j << "\t";

}

cout << endl;

}

cin.get();

}

这里使用内外两层for循环，实现了一个二维“表”的输出。后面我们会看到，循环嵌套对于处理多维数据非常有用。

### 5.4 跳转

在流程控制语句中还有一类“跳转语句”，主要用来中断当前的执行过程。C++中有四种跳转语句：break，continue，goto以及return。

#### 5.4.1 break

break语句表示要“跳出”当前的流程控制语句，它只能出现在switch或者循环语句（while、do while、for）中。当代码中遇到break时，会直接中断距离最近的switch或者循环，跳转到外部继续执行。

int i = 0;

while (true)

{

cout << " Hello World! " << endl;

cout << " 这是第" << ++i << "次输出\n" << endl;

if (i >= 5)

{

break;

}

}

如果循环条件永远为真，那么循环体中一定要有break，保证在某种情况下程序可以退出循环。

#### 5.4.2 continue

continue语句表示“继续”执行循环，也就是中断循环中的本次迭代、并开始执行下一次迭代。很明显，continue只能用在循环语句中，同样针对最近的一层循环有效。

continue非常适合处理需要“跳过”某些情况的场合。

// 逢7过

for (int num = 1; num < 100; num++)

{

cout << "\t";

// 如果是7的倍数，或者数字中有7，则跳过

if (num % 7 == 0 || num % 10 == 7 || num / 10 == 7)

continue;

cout << num;

// 如果是10的倍数，则换行

if (num % 10 == 0)

cout << endl << endl;

}

上面模拟了一个经典的小游戏“逢7过”，如果遇到7的倍数比如7、14、21，或者数字中有7比如17、27、71，都要跳过。

#### 5.4.3 goto

goto语句表示无条件地跳转到程序中的另一条语句。goto的语法形式为：

goto *标签*;

这里的“标签”可以认为是一条语句的“名字”，跟变量类似，只不过它是指代一条语句的标识符。定义标签也非常简单，只要在一条语句前写出标识符，然后跟上冒号就可以了，比如：

begin: int a = 0;

下面是一个具体的例子：

int x = 0;

cout << "程序开始..." << endl;

begin:

do

{

cout << " x = " << ++x << endl;

} while (x < 10);

if (x < 15) {

cout << "回到原点！" << endl;

goto begin;

}

cout << "程序结束！" << endl;

由于goto可以任意跳转，所以它非常灵活，也非常危险。一般在代码中不要使用goto。

#### 5.4.4 return

return是用来终止函数运行并返回结果的。之前的Hello World程序中就曾经介绍，主函数最后的那句 return 0; 就是结束主函数并返回结果，一般这句可以省略。

而在自定义的函数中，同样可以用return来返回。

### 5.5 应用案例

综合利用分支和循环语句，就可以实现很多有趣的功能。

#### 5.5.1 判断质数

质数也叫素数，是指一个大于1的自然数，因数只有1和它自身。质数是数论中一个经典的概念，很多著名定理和猜想都跟它有关；质数也是现代密码学的基础。

判断一个数是否为质数没有什么规律可言，我们可以通过验证小于它的每个数能否整除，来做暴力求解。下面是一段判断质数、并输出0~100内所有质数的程序：

#include<iostream>

using namespace std;

// 定义一个判断质数的函数，用return返回判断结果

bool isPrime(int num)

{

int i = 2;

while (i < num)

{

if (num % i == 0) return false;

++i;

}

return true;

}

int main()

{

cout << "请输入一个自然数（不超过20亿）：" << endl;

int num;

cin >> num;

if (isPrime(num))

{

cout << num << "是质数！" << endl;

}

else

{

cout << num << "不是质数！" << endl;

}

cout << "\n=========================\n" << endl;

cout << "0 ~ 100 内的质数有：" << endl;

for (int i = 2; i <= 100; i++)

{

if (isPrime(i))

cout << i << "\t";

}

cout << endl;

cin.get();

cin.get();

}

#### 5.5.2 猜数字

猜数字是一个经典的小游戏，程序随机生成一个0~100的数字，然后由用户输入来猜测。如果猜对，输出结果并退出；如果不对，则提示偏大还是偏小。我们可以对猜的次数做限制，比如一共5次机会。

#include<iostream>

using namespace std;

int main()

{

cout << "=========猜数字========" << endl;

cout << "规则：输入0~100的整数，有5次机会\n" << endl;

// 以当前时间为随机数种子，生成一个0~100的伪随机数

srand(time(0));

int target = rand() % 100;

int n = 0; // 猜的次数

while (n < 5)

{

cout << "请输入0~100的整数：" << endl;

int num;

cin >> num;

if (num == target)

{

cout << "恭喜你，猜对了！ 幸运数字是：" << target << endl;

break;

}

else if (num > target)

cout << "数字太大了！再猜一遍！" << endl;

else

cout << "数字太小了！再猜一遍！" << endl;

++n;

}

if (n == 5)

cout << "已经猜过5遍，没有猜中！欢迎下次再来！" << endl;

cin.get();

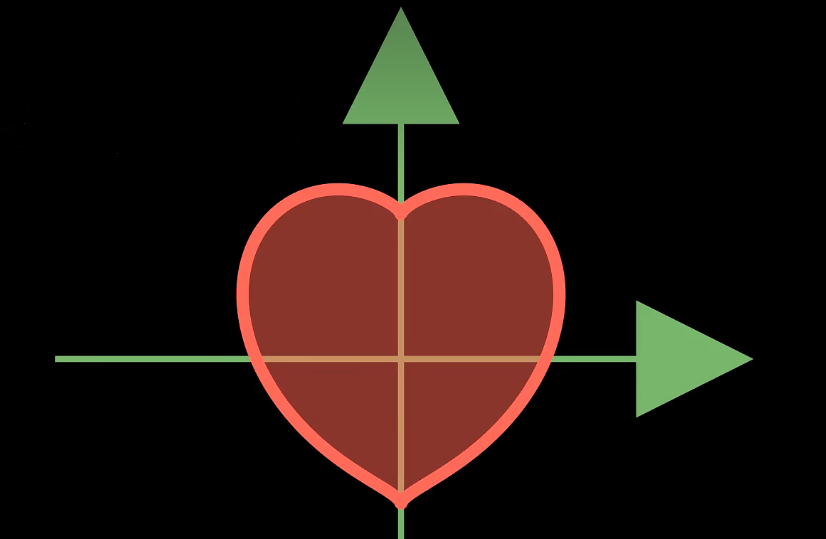
cin.get();

}

#### 5.5.3 爱心曲线

利用流程控制语句也可以绘制二维图形。只要知道函数表达式，就可以画出相应的曲线了。

我们可以尝试绘制传说中的“爱心曲线”。一个典型的爱心曲线函数如下：



曲线是一个封闭图形，与坐标轴的四个交点坐标为和，我们知道坐标(x, y)满足 的点都在“爱心”内部，而满足 的点都在“爱心”外部。

所以我们可以取边长为的正方形区域作为“画板”，扫描范围内所有点；在曲线内部的点用“\*”填充，外部的点则用空格填充。

#include<iostream>

using namespace std;

int main()

{

// 爱心曲线方程 (x^2+y^2-a)^3 - x^2 y^3 = 0

int a = 1;

// 定义绘图边界

double bound = 1.3 \* sqrt(a);

// x、y坐标变化步长

double step = 0.05;

for ( double y = bound; y >= -bound; y -= step)

{

for (double x = -bound; x <= bound; x += step)

{

double result = pow((pow(x, 2) + pow(y, 2) - a), 3) - pow(x, 2) \* pow(y, 3);

if (result <= 0)

cout << "\*";

else

cout << " ";

}

cout << endl;

}

cin.get();

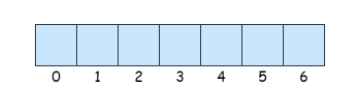
}

## 六、复合数据类型

C++中不仅有基本数据类型，还提供了更加灵活和丰富的复合数据类型。

### 6.1 数组

在程序中为了处理方便，常常需要把具有相同类型的数据对象按有序的形式排列起来，形成“一组”数据，这就是“数组”（array）。



数组中的数据，在内存中是连续存放的，每个元素占据相同大小的空间，就像排好队一样。

#### 6.1.1 数组的定义

数组的定义形式如下：

*数据类型* *数组名*[*元素个数*];

* 首先需要声明类型，数组中所有元素必须具有相同的数据类型；
* 数组名是一个标识符；后面跟着中括号，里面定义了数组中元素的个数，也就是数组的“长度”；
* 元素个数也是类型的一部分，所以必须是确定的；

int a1[10]; // 定义一个数组a1，元素类型为int，个数为10

const int n = 4;

double a2[n]; // 元素个数可以是常量表达式

int i = 5;

//int a3[i]; // 错误，元素个数不能为变量

需要注意，并没有通用的“数组”类型，所以上面的a1、a2的类型分别是“int数组”和“double数组”。这也是为什么我们把数组叫做“复合数据类型”。

#### 6.1.2 数组的初始化

之前在讲到for循环时，提到过使用范围for循环可以遍历一个“序列”，用花括号括起来的一组数就是一个序列。所以在给数组赋值时，也可以使用这样的序列。

int a3[4] = {1,2,3,4};

float a4[] = {2.5, 3.8, 10.1}; // 正确，初始值说明了元素个数是3

short a5[10] = {3,6,9}; // 正确，指定了前三个元素，其余都为0

//long a6[2] = {3,6,9}; // 错误，初始值太多

//int a6[4] = a3; // 错误，不能用另一个数组对数组赋值

需要注意的是：

* 对数组做初始化，要使用花括号{}括起来的数值序列；
* 如果做了初始化，数组定义时的元素个数可以省略，编译器可以根据初始化列表自动推断出来；
* 初始值的个数，不能超过指定的元素个数；
* 初始值的个数，如果小于元素个数，那么会用列表中的值初始化靠前的元素；剩余元素用默认值填充，整型的默认值就是0；
* 如果没有做初始化，数组中元素的值都是未定义的；这一点和普通的局部变量一致；

#### 6.1.3 数组的访问

（1）访问数组元素

数组元素在内存中是连续存放的，它们排好了队之后就会有一个队伍中的编号，称为“索引”，也叫“下标”；通过下标就可以快速访问每个元素了，具体形式为：

*数组名*[*元素下标*]

这里也是用了中括号来表示元素下标位置，被称为“下标运算符”。比如a[2]就表示数组a中下标为2的元素，可以取它的值输出，也可以对它赋值。

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

cout << "a[2] = " << a[2] << endl; // a[2] = 3

a[2] = 36;

cout << "a[2] = " << a[2] << endl; // a[2] = 36

需要注意的是：

* 数组的下标从0开始；
* 因此a[2]访问的并不是数组a的第2个元素，而是第三个元素；一个长度为10的数组，下标范围是0~9，而不是1~10；
* 合理的下标，不能小于0，也不能大于 (数组长度 - 1)；否则就会出现数组下标越界；

（2）数组的大小

所有的变量，都会在内存中占据一定大小的空间；而数据类型就决定了它具体的大小。而对于数组这样的“复合类型”，由于每个元素类型相同，因此占据空间大小的计算遵循下面的简单公式：

数组所占空间 = 数据类型所占空间大小 \* 元素个数

这样一来，即使定义的时候没有指定数组元素个数，现在也可以计算得出了：

// a是已定义的数组

cout << "a所占空间大小：" << sizeof(a) << endl;

cout << "每个元素所占空间大小：" << sizeof(a[0]) << endl;

// 获取数组长度

int aSize = sizeof(a) / sizeof(a[0]);

cout << "数组a的元素个数：" << aSize << endl;

这里为了获取数组的长度，我们使用了sizeof运算符，它可以返回一个数据对象在内存中占用的大小（以字节为单位）；数组总大小，除以每个数据元素的大小，就是元素个数。

（3）遍历数组

如果想要依次访问数组中所有的元素，就叫做“遍历数组”。我们当然可以用下标去挨个读取：

cout << "a[0] = " << a[0] << endl;

cout << "a[1] = " << a[1] << endl;

…

但这样显然太麻烦了。更好的方式是使用for循环：

// 获取数组长度

int aSize = sizeof(a) / sizeof(a[0]);

for (int i = 0; i < aSize; i++ )

{

cout << "a[" << i << "] = " << a[i] << endl;

}

循环条件如果写一个具体的数，很容易出现下标越界的情况；而如果知道了数组长度，直接让循环变量i小于它就可以了。

当然，这种写法还是稍显麻烦。C++ 11标准给我们提供了更简单的写法，就是之前介绍过的范围for循环：

for (int num: a )

{

cout << num << endl;

}

当然，这种情况下就无法获取元素对应的下标了。

#### 6.1.4 多维数组

之前介绍的数组只是数据最简单的排列方式。如果数据对象排列成的不是“一队”，而是一个“方阵”，那显然就不能只用一个下标来表示了。我们可以对数组进行扩展，让它从“一维”变成“二维”甚至“多维”。

int arr[3][4]; // 二维数组,有三个元素，每个元素是一个长度为4的int数组

int arr2[2][5][10]; // 三维数组

C++中本质上没有“多维数组”这种东西，所谓的“多维数组”，其实就是“数组的数组”。

* 二维数组int arr[3][4]表示：arr是一个有三个元素的数组，其中的每个元素都是一个int数组，包含4个元素；
* 三维数组int arr2[2][5][10]表示：arr2是一个长度为2的数组，其中每个元素都是一个二维数组；这个二维数组有5个元素，每个元素都是一个长度为10的int数组；

一般最常见的就是二维数组。它有两个“维度”，第一个维度表示数组本身的长度，第二个表示每个元素的长度；一般分别把它们叫做“行”和“列”。

（1）多维数组的初始化

和普通的“一维”数组一样，多维数组初始化时，也可以用花括号括起来的一组数。使用嵌套的花括号可以让不同的维度更清晰：

*数据类型 数组名*[*行数*][列数] = {数据1, 数据2, 数据3, …};

*数据类型 数组名*[*行数*][列数] = {

{数据11, 数据12, 数据13, …},

{数据21, 数据22, 数据23, …},

…

};

需要注意：

* 内嵌的花括号不是必需的，因为数组中的元素在内存中连续存放，可以用一个花括号将所有数据括在一起；
* 初始值的个数，可以小于数组定义的长度，其它元素初始化为0值；这一点对整个二维数组和每一行的一维数组都适用；
* 如果省略嵌套的花括号，当初始值个数小于总元素个数时，会按照顺序依次填充（填满第一行，才填第二行）；其它元素初始化为0值；
* 多维数组的维度，可以省略第一个，由编译器自动推断；即二维数组可以省略行数，但不能省略列数。

// 嵌套的花括号的初始化

int ia[3][4] = {

{1,2,3,4},

{5,6,7,8},

{9,10,11,12}

};

// 只有一层花括号的初始化

int ia2[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12 };

// 部分初始化，其余补0

int ia3[3][4] = {

{1,2,3},

{5,6}

};

int ia4[3][4] = {1,2,3,4,5,6};

// 省略行数，自动推断

int ia5[][4] = {1,2,3,4,5};

（2）访问数据

也可以用下标运算符来访问多维数组中的数据，数组的每一个维度，都应该有一个对应的下标。对于二维数组来说，就是需要指明“行号”“列号”，这相当于数据元素在二维矩阵中的坐标。

// 访问ia的第二行、第三个数据

cout << "ia[1][2] = " << ia[1][2] << endl;

// 修改ia的第一行、第二个数据

ia[0][1] = 19;

同样需要注意，行号和列号都是从0开始、到 (元素个数 - 1) 结束。

（3）遍历数组

要想遍历数组，当然需要使用for循环，而且要扫描每一个维度。对于二维数组，我们需要对行和列分别进行扫描，这是一个双重for循环：

cout << "二维数组总大小：" << sizeof(ia) << endl;

cout << "二维数组每行大小：" << sizeof(ia[0]) << endl;

cout << "二维数组每个元素大小：" << sizeof(ia[0][0]) << endl;

// 二维数组行数

int rowCnt = sizeof(ia) / sizeof(ia[0]);

// 二维数组列数

int colCnt = sizeof(ia[0]) / sizeof(ia[0][0]);

for (int i = 0; i < rowCnt; i++)

{

for (int j = 0; j < colCnt; j++)

{

cout << ia[i][j] << "\t";

}

cout << endl;

}

同样，这里利用了sizeof运算符：

* 行数 = 二维数组总大小 / 每行大小
* 列数 = 每行大小 / 每个元素大小

当然，也可以使用范围for循环：

for (auto & row : ia)

{

for (auto num : row)

{

cout << num << "\t";

}

cout << endl;

}

这里的外层循环使用了auto关键字，这也是C++ 11新引入的特性，它可以自动推断变量的类型；后面的&是定义了一个“引用”。关于这部分内容，会在后面继续介绍。

#### 6.1.5 数组的简单排序算法

数组排序指的是给定一个数组，要求把其中的元素按照从小到大（或从大到小）顺序排列。

这是一个非常经典的需求，有各种不同的算法可以实现。我们这里介绍两种最基本、最简单的排序算法。

（1）选择排序

选择排序是一种简单直观的排序算法。

它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后追加到已排序序列的末尾。以此类推，直到所有元素均排序完毕。



选择排序可以使用双重for循环很容易地实现：

#include<iostream>

using namespace std;

int main()

{

int arr[] = {5, 9, 2, 7, 4, 3, 12, 6, 1, 5, 7};

int size = sizeof(arr) / sizeof(arr[0]);

// 选择排序

for (int i = 0; i < size; i++)

{

for (int j = i + 1; j < size; j++)

{

if (arr[j] < arr[i])

{

// 如果arr[j]更小，就和arr[i]交换位置

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

// 输出

for (int num : arr)

cout << num << "\t";

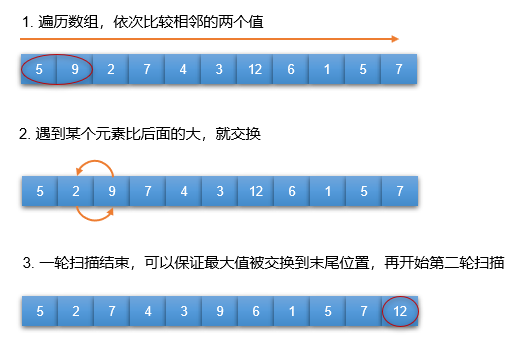
cin.get();

}

（2）冒泡排序

冒泡排序也是一种简单的排序算法。

它的基本原理是：重复地扫描要排序的数列，一次比较两个元素，如果它们的大小顺序错误，就把它们交换过来。这样，一次扫描结束，我们可以确保最大（小）的值被移动到序列末尾。这个算法的名字由来，就是因为越小的元素会经由交换，慢慢“浮”到数列的顶端。



冒泡排序的代码实现也非常简单，同样是使用双重for循环：

// 冒泡排序

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size - i - 1; j++)

{

if (arr[j] > arr[j+1])

{

int temp = arr[j+1];

arr[j+1] = arr[j];

arr[j] = temp;

}

}

}

### 6.2 模板类vector简介

数组尽管很灵活，但使用起来还是很多不方便。为此，C++语言定义了扩展的“抽象数据类型”（Abstract Data Type， ADT），放在“标准库”中。

对数组功能进行扩展的一个标准库类型，就是“容器”vector。顾名思义，vector“容纳”着一堆数据对象，其实就是一组类型相同的数据对象的集合。

#### 6.2.1 头文件和命名空间

vector是标准库的一部分。要想使用vector，必须在程序中包含<vector>头文件，并使用std命名空间。

#include<vector>

using namespace std;

在vector头文件中，对vector这种类型做了定义；使用#include引入它之后，并指定命名空间std之后，我们就可以在代码中直接使用vector了。

#### 6.2.2 vector的基本用法

vector其实是C++中的一个“类模板”，是用来创建类的“模子”。所以在使用时还必须提供具体的类型信息，也就是说，这个容器中到底要容纳什么类型的数据对象；具体的形式是在vector后面跟一个尖括号<>，里面填入具体类型信息。

vector<int> v;

（1）初始化

跟数组相比，vector的初始化更加灵活方便，可以应对各种不同的需求。

// 默认初始化，不含任何元素

vector<int> v1;

// 列表初始化

vector<char> v2 = {'a', 'b', 'c'};

// 省略等号的列表初始化

vector<short> v3{1,2,3,4,5};

// 只定义长度，元素初值默认初始化，容器中有5个0

vector<int> v4(5);

// 定义长度和初始值，容器中有5个100

vector<long> v5(5, 100);

这里有几种不同的初始化方式：

1. 默认初始化一个vector对象，就是一个空容器，里面不含任何元素；
2. C++ 11之后可以用花括号括起来的列表，对vector做初始化；等号可以省略；这种方式是把一个列表拷贝给了vector，称为“拷贝初始化”
3. 可以用小括号表示初始化vector的长度，并且可以给所有元素指定相同的初始值；这种方式叫做“直接初始化”

（2）访问元素

vector是包含了数据对象的“容器”，在这个容器集合中，每个数据对象都会有一个编号，用来做方便快速的访问；这个编号就是“索引”（index）。同样可以用下标操作符来获取对应索引的元素，这一点跟数组非常相似。

cout << "v5[2] = " << v5[2] << endl;

v5[4] = 32;

//v5[5] = 16; // 严重错误！不能越界访问索引

需要注意：

* vector内元素的索引，也是从0开始；
* vector索引最大值为 (vector长度 - 1)，不能越界访问；如果直接越界访问并赋值，有可能导致非常严重的后果，出现安全问题

（3）遍历所有元素

vector中有一个可以调用的函数size()，只要调用它就能直接得到vector的长度（即元素个数）：

// 获取vector的长度

cout << v5.size() << endl;

调用的方式是一个vector对象后面跟上一个点，再跟上size()。这种基于对象来调用的函数叫做“成员函数”。

这样我们就可以非常方便地用for循环遍历元素了：

for (int i = 0; i < v5.size(); i++)

{

cout << v5[i] << "\t";

}

当然，用范围for循环同样非常简单：

for (int num: v5)

{

cout << num << "\t";

}

（3）添加元素

vector的长度并不是固定的，所以可以向一个定义好的vector添加元素。

// 在定义好的vector中添加元素

v5.push\_back(69);

for (int num : v5)

{

cout << num << "\t";

}

这里的push\_back同样是一个成员函数，调用它的时候在小括号里传入想要添加的数值，就可以让vector对象中增加一个元素了。

这就使得我们在创建vector对象时不需要知道元素个数，使用更加灵活，避免了数组中的缺陷。

下面的代码创建了一个空vector，并使用添加元素的方式给它赋值为倒序的10~1：

vector<int> vec;

for (int i = 10; i > 0; i--)

{

vec.push\_back(i);

}

#### 6.2.3 vector和数组的区别

* 数组是更加底层的数据类型；长度固定，功能较少，安全性没有保证；但性能更好，运行更高效；
* vector是模板类，是数组的上层抽象；长度不定，功能强大；缺点是运行效率较低；

除了vector之外，C++ 11 还新增了一个array模板类，它跟数组更加类似，长度是固定的，但更加方便、更加安全。所以在实际应用中，一般推荐对于固定长度的数组使用array，不固定长度的数组使用vector。

### 6.3 字符串

字符串我们并不陌生。之前已经介绍过，一串字符连在一起就是一个“字符串”，比如用双引号引起来的“Hello World！”就是一个字符串字面值。

字符串其实就是所谓的“纯文本”，就是各种文字、数字、符号在一起表达的一串信息；所以字符串就是C++中用来表达和处理文本信息的数据类型。

#### 6.3.1 标准库类型string

C++的标准库中，提供了一种用来表示字符串的数据类型string，这种类型能够表示长度可变的字符序列。和vector类似，string类型也定义在命名空间std中，使用它必须包含string头文件。

#include<string>

using namespace std;

（1）定义和初始化string

我们已经接触过C++中几种不同的初始化方式，string也是一个标准库类型，它的初始化与vector非常相似。

// 默认初始化，空字符串

string s1;

// 用另一个字符串变量，做拷贝初始化

string s2 = s1;

// 用一个字符串字面值，做拷贝初始化

string s3 = "Hello World!";

// 用一个字符串字面值，做直接初始化

string s4("hello world");

// 定义字符和重复的次数，做直接初始化，得到 hhhhhhhh

string s5(8, 'h');

初始化方式主要有：

1. 默认初始化，得到的就是一个空字符串；
2. 拷贝初始化，用赋值运算符（等号“=”）表示；可以使用另一个string对象，也可以使用字符串字面值常量；
3. 直接初始化，用括号表示；可以在括号中传入一个字符串，也可以传入字符和重复的次数

可以发现，字符串也可以看做数据元素的集合；它里面的元素，就是字符。

（2）处理字符串中的字符

通过初始化已经可以看出，string的行为与vector非常类似。string同样也可以通过下标运算符访问内部的每个字符。字符的“索引”，就是在字符串中的位置。

string str = "hello world";

// 获取第3个字符

cout << "str[2] = " << str[2] << endl;

// 将第1个字符改为'H'

str[0] = 'H';

// 将最后一个字符改为'D'

str[str.size() - 1] = 'D';

cout << "str = " << str << endl;

字符串内字符的访问，跟vector内元素的访问类似，需要注意：

* string内字符的索引，也是从0开始；
* string同样有一个成员函数size，可以获取字符串的长度；
* 索引最大值为 (字符串长度 - 1)，不能越界访问；如果直接越界访问并赋值，有可能导致非常严重的后果，出现安全问题；
* 如果希望遍历字符串的元素，也可以使用普通for循环和范围for循环，依次获取每个字符

比如，我们可以考虑遍历所有字符，将小写字母换成大写：

// 遍历字符串中字符，将小写字母变成大写

for (int i = 0; i < str.size(); i++)

{

str[i] = toupper(str[i]);

}

这里又调用了string的一个函数toupper，可以把传入的字符转换成大写并返回。

（3）字符串相加

string本身的长度是不定的，可以通过“相加”的方式扩展一个字符串。

// 字符串相加

string str1 = "hello", str2("world");

string str3 = str1 + str2; // str3 = "helloworld"

string str4 = str1 + ", " + str2 + "!"; // str4 = "hello, world!"

//string str5 = "hello, " + "world!"; // 错误，不能将两个字符串字面值相加

需要注意：

* 字符串相加使用加号“+”来表示，这是算术运算符“+”的运算符重载，含义是“字符串拼接”；
* 两个string对象，可以直接进行字符串相加；结果是将两个字符串拼接在一起，得到一个新的string对象返回；
* 一个string对象和一个字符串字面值常量，可以进行字符串相加，同样是得到一个拼接后的string对象返回；
* 两个字符串字面值常量，不能相加；
* 多个string对象和多个字符串字面值常量，可以连续相加；前提是按照左结合律，每次相加必须保证至少有一个string对象；

（4）比较字符串

string类还提供几种用来做字符串比较的运算符，“==”和“!=”用来判断两个字符串是否完全一样；而“<”“>”“<=”“>=”则用来比较两个字符串的大小。这些都是关系型运算符的重载。

str1 = "hello";

str2 = "hello world!";

str3 = "hehehe";

str1 == str2; // false

str1 < str2; // true

str1 >= str3; // true

字符串比较的规则为：

* 如果两个字符串长度相同，每个位置包含的字符也都相同，那么两者“相等”；否则“不相等”；
* 如果两个字符串长度不同，而较短的字符串每个字符都跟较长字符串对应位置字符相同，那么较短字符串“小于”较长字符串；
* 如果两个字符串在某一位置上开始不同，那么就比较这两个字符的ASCII码，比较结果就代表两个字符串的大小关系

#### 6.3.2 字符数组（C风格字符串）

通过对string的介绍可以发现，字符串就是一串字符的集合，本质上其实就是一个“字符的数组”。

在C语言中，确实是用char[]类型来表示字符串的；不过为了区分纯粹的“字符数组”和“字符串”，C语言规定：字符串必须以空字符结束。空字符的ASCII码为0，专门用来标记字符串的结尾，在程序中写作’\0’。

// str1没有结尾空字符，并不是一个字符串

char str1[5] = {'h','e','l','l','o'};

// str2是一个字符串

char str2[6] = { 'h','e','l','l','o','\0'};

cout << "str1 = " << str1 << endl;

cout << "str2 = " << str2 << endl;

如果每次用到字符串都要这样定义，对程序员来说就非常不友好了。所以字符串可以用另一种更方便的形式定义出来，那就是使用双引号：

char str3[] = "hello";

//char str3[5] = "hello"; // 错误，"hello"的长度为6

cout << "str3 = " << str3 << endl;

这就是我们所熟悉的字符串“字面值常量”。这里需要注意的是，我们不需要再考虑末尾的空字符，编译器会自动帮我们补全；但真实的字符串的长度，依然要包含空字符，所以上面的字符串“hello”长度不是5、而是6。

所以，C++中的字符串字面值常量，为了兼容C依然定义为字符数组（char[]）类型，这和string是两种不同类型；两者的区别，跟数组和vector的区别类似，char[]是更底层的类型。一般情况下，使用string会带来更多方便，也会更加安全。

#### 6.3.3 读取输入的字符串

程序中往往需要一些交互操作，如果想获取从键盘输入的字符串，可以使用多种方法。

（1） 使用输入操作符读取单词

标准库中提供了iostream，可以使用内置的cin对象，调用重载的输入操作符>>来读取键盘输入。

string str;

// 读取键盘输入，遇到空白符停止

cin >> str;

cout << str;

这种方式的特点是：忽略开始的空白符，遇到下一个空白符（空格、回车、制表等）就会停止。所以如果我们输入“hello world”，那么读取给str的只有“hello”：这相当于读取了一个“单词”。

剩下的内容“world”其实也没有丢，而是保存在了输入流的“输入队列”里。如果我们想读取更多的输入信息，就需要使用更多的string对象来获取：

string str1, str2;

cin >> str1 >> str2;

cout << str1 << str2 << endl;

这样，如果输入“hello world”，就可以输出“helloworld”。

（2）使用getline读取一行

如果希望直接读取一整行输入信息，可以使用getline函数来替代输入操作符。

string str3;

getline(cin, str3);

cout << "str3 = " << str3 << endl;

getline函数有两个参数：一个是输入流对象cin，另一个是保存字符串的string对象；它会一直读取输入流中的内容，直到遇到换行符为止，然后把所有内容保存到string对象中。所以现在可以完整读取一整行信息了。

（3）使用get读取字符

还有一种方法，是调用cin的get函数读取一个字符。

char ch;

ch = cin.get(); // 将捕获到的字符赋值给ch

cin.get(ch); // 直接将ch作为参数传给get

有两种方式：

* 调用cin.get()函数，不传参数，得到一个字符赋给char类型变量；
* 将char类型变量作为参数传入，将捕获的字符赋值给它，返回的是istream对象

get函数还可以读取一行内容。这种方式跟getline很相似，也可以读取一整行内容，以回车结束。主要区别在于，它需要把信息保存在一个char[]类型的字符数组中，调用的是cin的成员函数：

// get读取一整行

char str4[20];

cin.get(str4, 20);

cout << "str4 = " << str4 << endl;

// get读取一个字符

cin.get(); // 先读取之前留下的回车符

cin.get(); // 再等待下一次输入

get函数同样需要传入两个参数：一个是保存信息的字符数组，另一个是字符数组的长度。

这里还要注意跟getline的另一个区别：键盘输入总是以回车作为结束的；getline会把最后的回车符丢弃，而get会将回车符保留在输入队列中。

这样的效果是，下次再调用get试图读取一行数据时，会因为直接读到了回车符而返回空行。这就需要再次调用get函数，捕获下一个字符：

cin.get(); // 先读取之前留下的回车符

cin.get(); // 再等待下一次输入

这样就可以将之前的回车符捕获，从而为读取下一行做好准备。这也就解释了之前为什么要写两个cin.get()：第一个用来处理之前保留在输入队列的回车符；第二个用来等待下一次输入，让窗口保持开启状态。

#### 6.3.4 简单读写文件

实际应用中，我们往往会遇到读写文件的需求，这也是一种IO操作，整体用法跟命令行的输入输出非常类似。

C++的IO库中提供了专门用于文件输入的ifstream类和用于文件输出的ofstream类，要使用它们需要引入头文件fstream。ifstream用于读取文件内容，跟istream的用法类似；也可以通过输入操作符>>来读“单词”（空格分隔），通过getline函数来读取一行，通过get函数来读取一个字符：

ifstream input("input.txt");

// 逐词读取

string word;

while (input >> word)

cout << word << endl;

// 逐行读取

string line;

while (getline(input, line))

cout << line << endl;

// 逐字符读取

char ch;

while (input.get(ch))

cout << ch << endl;

类似地，写入文件也可以通过使用输出运算符 << 来实现：

ofstream output("output.txt");

output << word << endl;

### 6.4 结构体

实际应用中，我们往往希望把很多不同的信息组合起来，“打包”存储在一个单元中。比如一个学生的信息，可能包含了姓名、年龄、班级、成绩…这些信息的数据类型可能是不同的，所以数组和vector都无法完成这样的功能。

C/C++中提供了另一种更加灵活的数据结构——结构体。结构体是用户自定义的复合数据结构，里面可以包含多个不同类型的数据对象。

#### 6.4.1 结构体的声明

声明一个结构体需要使用struct关键字，具体形式如下：

struct *结构体名*

{

*类型1 数据对象1*;

*类型2 数据对象2*;

*类型3 数据对象3*;

…

};

结构体中数据对象的类型和个数都可以自定义，这为数据表达提供了极大的灵活性。结构体可以说是迈向面向对象世界中“类”概念的第一步。

我们可以尝试定义这样一个“学生信息”结构体：

struct studentInfo

{

string name;

int age;

double score;

};

这个结构体中包含了三个数据对象：string类型的名字name，int类型的年龄age，以及double类型的成绩score。一般会把结构体定义在主函数外面，称为“外部定义”，这样可以方便外部访问。

#### 6.4.2 结构体初始化

定义好结构之后，就产生了一个新的类型，叫做“studentInfo”。接下来就可以创建这种类型的对象，并做初始化了。

// 创建对象并初始化

studentInfo stu = {"张三", 20, 60.0};

结构体对象的初始化非常简单，跟数组完全一样：只要按照对应顺序一次赋值，逗号分隔，最后用花括号括起来就可以了。

结构体还支持其它一些初始化方式：

struct studentInfo

{

string name;

int age;

double score;

}stu1, stu2 = {"小明", 18, 75.0}; // 定义结构体之后立即创建对象

// 使用列表初始化

studentInfo stu3{"李四", 22, 87};

// 使用另一结构体对象进行赋值

studentInfo stu4 = stu2;

需要注意：

* 创建结构体变量对象时，可以直接用定义好的结构体名作为类型；相比C语言中的定义，这里省略了关键字struct
* 不同的初始化方式效果相同，在不同位置定义的对象作用域不同；
* 如果没有赋初始值，那么所有数据将被初始化为默认值；算术类型的默认值就是0；
* 一般在代码中，会将结构体的定义和对象的创建分开，便于理解和管理

#### 6.4.3 访问结构体中数据

访问结构体变量中的数据成员，可以使用成员运算符（点号.），后面跟上数据成员的名称。例如stu.name就可以访问stu对象的name成员。

cout << "学生姓名：" << stu.name << "\t年龄：" << stu.age << "\t成绩：" << stu.score << endl;

这种访问内部成员的方式非常经典，后面要讲到的类的操作中，也会用这种方式访问自己的成员函数。

#### 6.4.4 结构体数组

可以把结构体和数组结合起来，创建结构体的数组。顾名思义，结构体数组就是元素为结构体的数组，它的定义和访问跟普通的数组完全一样。

// 结构体数组

studentInfo s[2] = {

{"小红", 18, 92},

{"小白", 20, 82}

};

cout << "学生姓名：" << s[0].name << "\t年龄：" << s[0].age << "\t成绩：" << s[0].score << endl;

cout << "学生姓名：" << s[1].name << "\t年龄：" << s[1].age << "\t成绩：" << s[1].score << endl;

### 6.5 枚举

实际应用中，经常会遇到某个数据对象只能取有限个常量值的情况，比如一周有7天，一副扑克牌有4种花色等等。对于这种情况，C++提供了另一种批量创建符号常量的方式，可以替代const。这就是“枚举”类型enum。

#### 6.5.1 枚举类型定义

枚举类型的定义和结构体非常像，需要使用enum关键字。

// 定义枚举类型

enum week

{

Mon, Tue, Wed, Thu, Fri, Sat, Sun

};

与结构体不同的是，枚举类型内只有有限个名字，它们都各自代表一个常量，被称为“枚举量”。

需要注意的是：

* 默认情况下，会将整数值赋给枚举量；
* 枚举量默认从0开始，每个枚举量依次加1；所以上面week枚举类型中，一周七天枚举量分别对应着0~6的常量值；
* 可以通过对枚举量赋值，显式地设置每个枚举量的值

#### 6.5.2 使用枚举类型

使用枚举类型也很简单，创建枚举类型的对象后，只能将对应类型的枚举量赋值给它；如果打印它的值，将会得到对应的整数。

week w1 = Mon;

week w2 = Tue;

//week w3 = 2; // 错误，类型不匹配

week w3 = week(3); // int类型强转为week类型后赋值

cout << "w1 = " << w1 << endl;

cout << "w2 = " << w2 << endl;

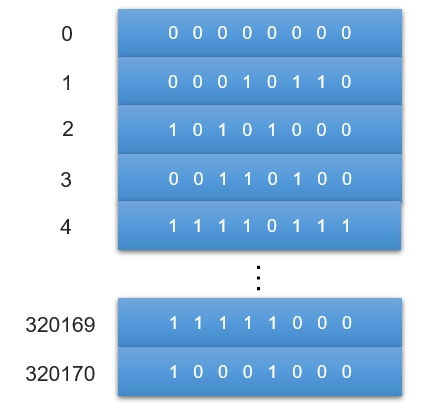
cout << "w3 = " << w3 << endl;

这里需要注意：

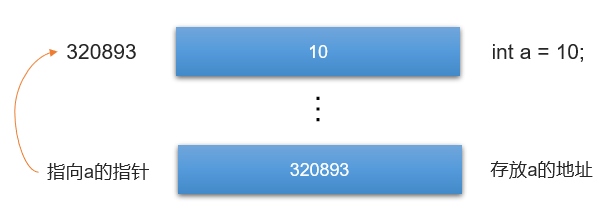
* 如果直接用一个整型值对枚举类型赋值，将会报错，因为类型不匹配；
* 可以通过强制类型转换，将一个整型值赋值给枚举对象；
* 最初的枚举类型只有列出的值是有效的；而现在C++通过强制类型转换，允许扩大枚举类型合法值的范围。不过一般使用枚举类型要避免直接强转赋值。

### 6.6 指针

计算机中的数据都存放在内存中，访问内存的最小单元是“字节”（byte）。所有的数据，就保存在内存中具有连续编号的一串字节里。



指针顾名思义，是“指向”另外一种数据类型的复合类型。指针是C/C++中一种特殊的数据类型，它所保存的信息，其实是另外一个数据对象在内存中的“地址”。通过指针可以访问到指向的那个数据对象，所以这是一种间接访问对象的方法。



#### 6.6.1 指针的定义

指针的定义语法形式为：

*类型* \* *指针变量*;

这里的类型就是指针所指向的数据类型，后面加上星号“\*”，然后跟指针变量的名称。指针在定义的时候可以不做初始化。相比一般的变量声明，看起来指针只是多了一个星号“\*”而已。例如：

int\* p1; // p1是指向int类型数据的指针

long\* p2; // p2是指向long类型数据的指针

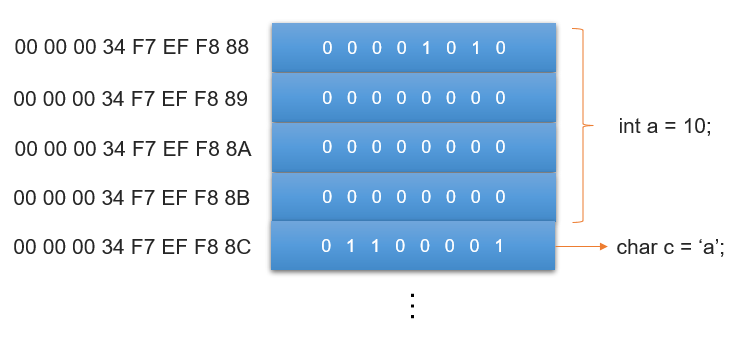
cout << "p1在内存中长度为：" << sizeof(p1) << endl;

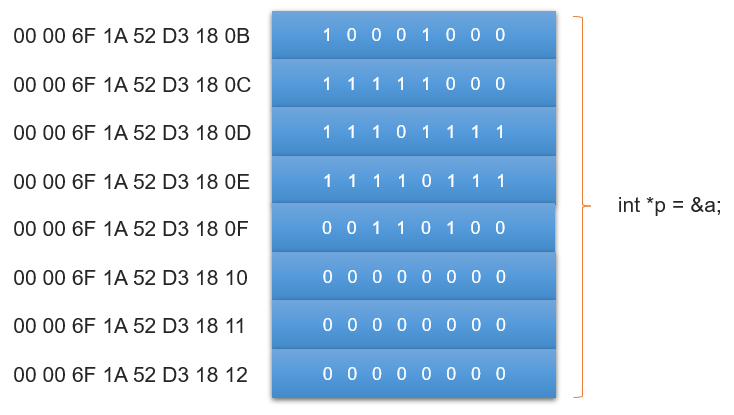
cout << "p2在内存中长度为：" << sizeof(p2) << endl;

p1、p2就是两个指针，分别指向int类型和long类型的数据对象。

指针的本质，其实就是一个整数表示的内存地址，它本身在内存中所占大小跟系统环境有关，而跟指向的数据类型无关。64位编译环境中，指针统一占8个字节；若是32位系统则占4字节。

#### 6.6.2 指针的用法





（1）获取对象地址给指针赋值

指针保存的是数据对象的内存地址，所以可以用地址给指针赋值；获取对象地址的方式是使用“取地址操作符”（&）。

int a = 12;

int b = 100;

cout << "a = " << a << endl;

cout << "a的地址为：" << &a << endl;

cout << "b的地址为：" << &b << endl;

int\* p = &b; // p是指向b的指针

p = &a; // p指向了a

cout << "p = " << p << endl;

把指针当做一个变量，可以先指向一个对象，再指向另一个不同的对象。

（2）通过指针访问对象

指针指向数据对象后，可以通过指针来访问对象。访问方式是使用“解引用操作符”（\*）：

p = &a; // p是指向a的指针

cout << "p指向的内存中，存放的值为：" << \*p << endl;

\*p = 25; // 将p所指向的对象（a），修改为25

cout << "a = " << a << endl;

在这里由于p指向了a，所以\*p可以等同于a。

#### 6.6.3 无效指针、空指针和void\*指针

（1）无效指针

定义一个指针之后，如果不进行初始化，那么它的内容是不确定的（比如0xcccc）。如果这时把它的内容当成一个地址去访问，就可能访问的是不存在的对象；更可怕的是，如果访问到的是系统核心内存区域，修改其中内容会导致系统崩溃。这样的指针就是“无效指针”，也被叫做“野指针”。

int\* p1;

//\*p1 = 100; // 危险！指针没有初始化，是无效指针

指针非常灵活非常强大，但野指针非常危险。所以建议使用指针的时候，一定要先初始化，让它指向真实的对象。

（2）空指针

如果先定义了一个指针，但确实还不知道它要指向哪个对象，这时可以把它初始化为“空指针”。空指针不指向任何对象。

int\* np = nullptr; // 空指针字面值

np = NULL; // 预处理变量

np = 0; // 0值

int zero = 0;

//np = zero; // 错误，int变量不能赋值给指针

cout << "np = " << np << endl; // 输出0地址

//cout << "\*np = " << \*np << endl; // 错误，不能访问0地址的内容

空指针有几种定义方式：

* 使用字面值nullptr，这是C++ 11 引入的方式，推荐使用；
* 使用预处理变量NULL，这是老版本的方式；
* 直接使用0值；
* 另外注意，不能直接用整型变量给指针赋值，即使值为0也不行

所以可以看出，空指针所保存的其实就是0值，一般把它叫做“0地址”；这个地址也是内存中真实存在的，所以也不允许访问。

空指针一般在程序中用来做判断，看一个指针是否指向了数据对象。

（3）void \* 指针

一般来说，指针的类型必须和指向的对象类型匹配，否则就会报错。不过有一种指针比较特殊，可以用来存放任意对象的地址，这种指针的类型是void\*。

int i = 10;

string s = "hello";

void\* vp = &i;

vp = &s;

cout << "vp = " << vp << endl;

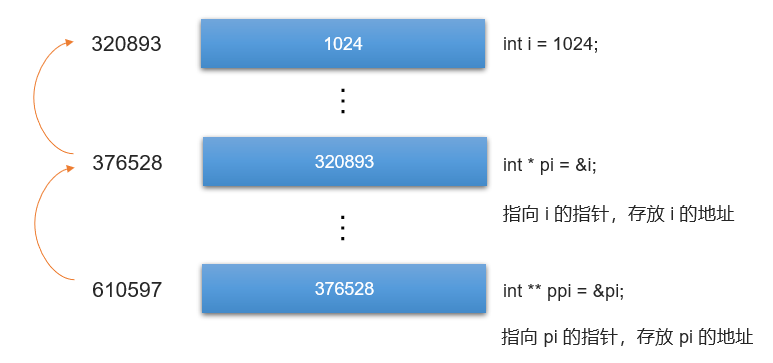
cout << "vp的长度为： " << sizeof(vp) << endl;

//cout << "\*vp = " << \*vp << endl; // 错误，不能通过void \*指针访问对象

void\* 指针表示只知道“保存了一个地址”，至于这个地址对应的数据对象是什么类型并不清楚。所以不能通过 void\* 指针访问对象；一般 void\* 指针只用来比较地址、或者作为函数的输入输出。

#### 6.6.4 指向指针的指针

指针本身也是一个数据对象，也有自己的内存地址。所以可以让一个指针保存另一个指针的地址，这就是“指向指针的指针”，有时也叫“二级指针”；形式上可以用连续两个的星号\*\*来表示。类似地，如果是三级指针就是\*\*\*，表示“指向二级指针的指针”。



int i = 1024;

int\* pi = &i; // pi是一个指针，指向int类型的数据

int\*\* ppi = &pi; // ppi是一个二级指针，指向一个int\* 类型的指针

cout << "pi = " << pi << endl;

cout << "\* pi = " << \* pi << endl;

cout << "ppi = " << ppi << endl;

cout << "\* ppi = " << \* ppi << endl;

cout << "\*\* ppi = " << \*\* ppi << endl;

如果需要访问二级指针所指向的最原始的那个数据，应该做两次解引用操作。

#### 6.6.5 指针和const

指针可以和const修饰符结合，这可以有两种形式：一种是指针指向的是一个常量；另一种是指针本身是一个常量。

（1）指向常量的指针

指针指向的是一个常量，所以只能访问数据，不能通过指针对数据进行修改。不过指针本身是变量，可以指向另外的数据对象。这时应该把const加在类型前。

const int c = 10, c2 = 56;

//int\* pc = &c; // 错误，类型不匹配

const int\* pc = &c; // 正确，pc是指向常量的指针，类型为const int \*

pc = &c2; // pc可以指向另一个常量

int i = 1024;

pc = &i; // pc也可以指向变量

\*pc = 1000; // 错误，不能通过pc更改数据对象

这里发现，pc是一个指向常量的指针，但其实把一个变量i的地址赋给它也是可以的；编译器只是不允许通过指针pc去间接更改数据对象。

（2）指针常量（const指针）

指针本身是一个数据对象，所以也可以区分变量和常量。如果指针本身是一个常量，就意味它保存的地址不能更改，也就是它永远指向同一个对象；而数据对象的内容是可以通过指针改变的。这种指针一般叫做“指针常量”。

指针常量在定义的时候，需要在星号\*后、标识符前加上const。

int\* const cp = &i;

\*cp = 2048; // 通过指针修改对象的值

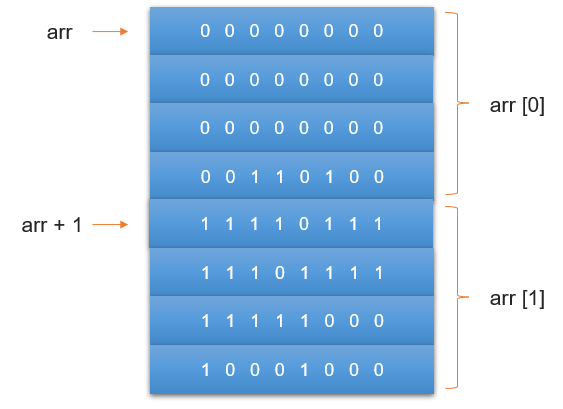
cout << "i = " << i << endl;

//cp = &c; // 错误，不可以更改cp的指向

const int\* const ccp = &c; // ccp是一个指向常量的常量指针

这里也可以使用两个const，定义的是“指向常量的常量指针”。也就是说，ccp指向的是常量，值不能改变；而且它本身也是一个常量，指向的对象也不能改变。

#### 6.6.6 指针和数组



（1）数组名

用到数组名时，编译器一般都会把它转换成指针，这个指针就指向数组的第一个元素。所以我们也可以用数组名来给指针赋值。

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

cout << "arr = " << arr << endl;

cout << "&arr[0] = " << &arr[0] << endl;

int\* pia = arr; // 可以直接用数组名给指针赋值

cout << "\* pia = " << \*pia << endl; // 指针指向的数据，就是arr[0]

也正是因为数组名被认为是指针，所以不能直接使用数组名对另一个数组赋值，数组也不允许这样的直接拷贝：

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

//int arr2[5] = arr; // 错误，数组不能直接拷贝

（2）指针运算

如果对指针pia做加1操作，我们会发现它保存的地址直接加了4，这其实是指向了下一个int类型数据对象：

pia + 1; // pia + 1 指向的是arr[1]

\*(pia + 1); // 访问 arr[1]

所谓的“指针运算”，就是直接对一个指针加/减一个整数值，得到的结果仍然是指针。新指针指向的数据元素，跟原指针指向的相比移动了对应个数据单位。

（3）指针和数组下标

我们知道，数组名arr其实就是指针。这就带来了非常有趣的访问方式：

\* arr; // arr[0]

\*(arr + 1); // arr[1]

这是通过指针来访问数组元素，效果跟使用下标运算符arr[0]、arr[1]是一样的。进而我们也可以发现，遍历元素所谓的“范围for循环”，其实就是让指针不停地向后移动依次访问元素。

（4）指针数组和数组指针

指针和数组这两种类型可以结合在一起，这就是“指针数组”和“数组指针”。

* 指针数组：一个数组，它的所有元素都是相同类型的指针；
* 数组指针：一个指针，指向一个数组的指针；

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

int\* pa[5]; // 指针数组，里面有5个元素，每个元素都是一个int指针

int(\* ap)[5]; // 数组指针，指向一个int数组，数组包含5个元素

cout << "指针数组pr的大小为：" << sizeof(pa) << endl; // 40

cout << "数组指针ap的大小为：" << sizeof(ap) << endl; // 8

pa[0] = arr; // pa中第一个元素，指向arr的第一个元素

pa[1] = arr + 1; // pa中第二个元素，指向arr的第二个元素

ap = &arr; // ap指向了arr整个数组

cout << "arr =" << arr << endl;

cout << "\* arr =" << \*arr << endl; // arr解引用，得到arr[0]

cout << "arr + 1 =" << arr + 1 << endl;

cout << "ap =" << ap << endl;

cout << "\* ap =" << \*ap << endl; // ap解引用，得到的是arr数组

cout << "ap + 1 =" << ap + 1 << endl;

这里可以看到，指向数组arr的指针ap，其实保存的也是arr第一个元素的地址。arr类型是int \*，指向的就是arr[0]；而ap类型是int (\*) [5]，指向的是整个arr数组。所以arr + 1，得到的是arr[1]的地址；而ap + 1，就会跨过整个arr数组。

### 6.7 引用

我们可以在C++中为数据对象另外起一个名字，这叫做“引用”（reference）。

#### 6.7.1 引用的用法

在做声明时，我们可以在变量名前加上“&”符号，表示它是另一个变量的引用。引用必须被初始化。

int a = 10;

int& ref = a; // ref是a的引用

//int& ref2; // 错误，引用必须初始化

cout << "ref = " << ref << endl; // ref等于a的值

cout << "a的地址为：" << &a << endl;

cout << "ref的地址为：" << &ref << endl; // ref和a的地址完全一样

引用本质上就是一个“别名”，它本身不是数据对象，所以本身不会存储数据，而是和初始值“绑定”（bind）在一起，绑定之后就不能再绑定别的对象了。

定义了应用之后，对引用做的所有操作，就像直接操作绑定的原始变量一样。所以，引用也是一种间接访问数据对象的方式。

ref = 20; // 更改ref相当于更改a

cout << "a = " << a << endl;

int b = 26;

ref = b; // ref没有绑定b，而是把b的值赋给了ref绑定的a

cout << "a的地址为：" << &a << endl;

cout << "b的地址为：" << &b << endl;

cout << "ref的地址为：" << &ref << endl;

cout << "a = " << a << endl;

当然，既然是别名，那么根据这个别名再另起一个别名也是可以的：

// 引用的引用

int& rref = ref;

cout << "rref = " << rref << endl;

cout << "a的地址为：" << &a << endl;

cout << "ref的地址为：" << &ref << endl;

cout << "rref的地址为：" << &rref << endl;

“引用的引用”，是把引用作为另一个引用的初始值，其实就是给原来绑定的对象又绑定了一个别名，这两个引用绑定的是同一个对象。

要注意，引用只能绑定到对象上，而不能跟字面值常量绑定；也就是说，不能把一个字面值直接作为初始值赋给一个引用。而且，引用本身的类型必须跟绑定的对象类型一致。

//int& ref2 = 10; // 错误，不能创建字面值的引用

double d = 3.14;

//int& ref3 = d; // 错误，引用类型和原数据对象类型必须一致

#### 6.7.2 对常量的引用

可以把引用绑定到一个常量上，这就是“对常量的引用”。很显然，对常量的引用是常量的别名，绑定的对象不能修改，所以也不能做赋值操作：

const int zero = 0;

//int& cref = zero; // 错误，不能用普通引用去绑定常量

const int& cref = zero; // 常量的引用

//cref = 10; // 错误，不能对常量赋值

对常量的引用有时也会直接简称“常量引用”。因为引用只是别名，本身不是数据对象；所以这只能代表“对一个常量的引用”，而不会像“常量指针”那样引起混淆。

常量引用和普通变量的引用不同，它的初始化要求宽松很多，只要是可以转换成它指定类型的所有表达式，都可以用来做初始化。

const int& cref2 = 10; // 正确，可以用字面值常量做初始化

int i = 35;

const int& cref3 = i; // 正确，可以用一个变量做初始化

double d = 3.14;

const int& cref4 = d; // 正确，d会先转成int类型，引用绑定的是一个“临时量”

这样一来，常量引用和对变量的引用，都可以作为一个变量的“别名”，区别在于不能用常量引用去修改对象的值。

int var = 10;

int& r1 = var;

const int& r2 = var;

r1 = 25;

//r2 = 35; // 错误，不能通过const引用修改对象值

#### 6.7.3 指针和引用

从上一节中可以看到，常量引用和指向常量的指针，有很类似的地方：它们都可以绑定/指向一个常量，也可以绑定/指向一个变量；但不可以去修改对应的变量对象。所以很明显，指针和引用有很多联系。

（1）引用和指针常量

事实上，引用的行为，非常类似于“指针常量”，也就是只能指向唯一的对象、不能更改的指针。

int a = 10;

// 引用的行为，和指针常量非常类似

int& r = a;

int\* const p = &a;

r = 20;

\*p = 30;

cout << "a = " << a << endl;

cout << "a的地址为：" << &a << endl;

cout << "r = " << r << endl;

cout << "r的地址为：" << &r << endl;

cout << "\*p = " << \*p << endl;

cout << "p = " << p << endl;

可以看到，所有用到引用r的地方，都可以用\*p替换；所有需要获取地址&r的地方，也都可以用p替换。这也就是为什么把操作符\*，叫做“解引用”操作符。

（2）指针的引用

指针本身也是一个数据对象，所以当然也可以给它起别名，用一个引用来绑定它。

int i = 56, j = 28;;

int\* ptr = &i; // ptr是一个指针，指向int类型对象

int\*& pref = ptr; // pref是一个引用，绑定指针ptr

pref = &j; // 将指针ptr指向j

\*pref = 20; // 将j的值变为20

pref是指针ptr的引用，所以下面所有的操作，pref就等同于ptr。

可以有指针的引用、引用的引用，也可以有指向指针的指针；但由于引用只是一个“别名”，不是实体对象，所以不存在指向引用的指针。

int& ref = i;

//int&\* rptr = &ref; // 错误，不允许使用指向引用的指针

int\* rptr = &ref; // 事实上就是指向了i

（3）引用的本质

引用类似于指针常量，但不等同于指针常量。

指针常量本身还是一个数据对象，它保存着另一个对象的地址，而且不能更改；而引用就是“别名”，它会被编译器直接翻译成所绑定的原始变量；所以我们会看到，引用和原始对象的地址是一样，引用并没有额外占用内存空间。这也是为什么不会有“指向引用的指针”。

引用的本质，只是C++引入的一种语法糖，它是对指针的一种伪装。

指针是C语言中最灵活、最强大的特性；引用所能做的，其实指针全都可以做。但是指针同时又令人费解、充满危险性，所以C++中通过引用来代替一些指针的用法。后面在函数部分，我们会对此有更深刻的理解。

### 6.8 应用案例

#### 6.8.1 翻转数组

翻转数组，就是要把数组中元素的顺序全部反过来。比如一个数组{1,2,3,4,5,6,7,8}，翻转之后就是{8,7,6,5,4,3,2,1}。

（1）另外创建数组，反向填入元素

数组是将元素按照顺序依次存放的，长度固定。所以如果想要让数组“翻转”，一种简单的思路是：直接创建一个相同长度的新数组，然后遍历所有元素，从末尾开始依次反向填入就可以了。

#include<iostream>

using namespace std;

int main()

{

const int n = 8;

int arr[n] = { 1,2,3,4,5,6,7,8 };

// 1. 直接创建一个新数组，遍历元素反向填入

int newArr[n];

for (int i = 0; i < n; i++)

{

newArr[n-i-1] = arr[i];

}

// 打印数组

for (int i = 0; i < n; i++)

{

cout << newArr[i] << "\t";

}

cout << endl;

cin.get();

}

需要注意原数组下标为i的元素，对应翻转后的新数组下标为n-i-1（n为数组长度）。

（2）基于原数组翻转

另建数组的方式很容易实现，但有明显的缺点：需要额外创建一个数组，占用更多的内存。最好的方式是，不要另开空间，就在原数组上调整位置。

这种思路的核心在于：我们应该有两个类似“指针”的东西，每次找到头尾两个元素，将它们调换位置；而后指针分别向中间逼近，再找两个元素对调。由于数组中下标是确定的，因此可以直接用下标代替“指针”。

// 2. 双指针分别指向数组头尾，元素对调

int head = 0, tail = n - 1;

while(head < tail)

{

int temp = arr[head];

arr[head] = arr[tail];

arr[tail] = temp;

// 指针向中间移动

++head;

--tail;

}

// 打印数组

for (int i = 0; i < n; i++)

{

cout << arr[i] << "\t";

}

cout << endl;

#### 6.8.2 检验幻方

“幻方”是数学上一个有趣的问题，它让一组不同的数字构成一个方阵，并且每行、每列、每个对角线的所有数之和相等。比如最简单的三阶幻方，就是把1~9的数字填到九宫格里，要求横看、竖看、斜着看和都是15。

口诀：二四为肩，六八为足，左三右七，戴九履一，五居中央。



我们可以给定一个n×n的矩阵，也就是二维数组，然后判断它是否是一个幻方：

#include<iostream>

using namespace std;

int main()

{

const int n = 3;

int arr[n][n] = {

{4, 9, 2},

{3, 5, 7},

{8, 1, 6}

};

// 目标和

int target = (1 + n \* n) \* n / 2;

bool isMagic = true;

// 检验每一行

for (int i = 0; i < n; i++)

{

int sum = 0;

for (int j = 0; j < n; j++)

{

sum += arr[i][j];

}

// 如果和不是target，说明不是幻方

if (sum != target)

{

isMagic = false;

break;

}

}

// 检验每一列

for (int j = 0; j < n; j++)

{

int sum = 0;

for (int i = 0; i < n; i++)

{

sum += arr[i][j];

}

if (sum != target)

{

isMagic = false;

break;

}

}

// 检验两个对角线

int sumDiag1 = 0;

int sumDiag2 = 0;

for (int i = 0; i < n; i++)

{

sumDiag1 += arr[i][i];

sumDiag2 += arr[i][n-i-1];

}

if (sumDiag1 != target || sumDiag2 != target)

{

isMagic = false;

}

// 判断结果

cout << "给定的矩阵arr" << (isMagic ? "是" : "不是") << n << "阶幻方！" << endl;

cin.get();

}

#### 6.8.3 大整数相加

实际应用中，有时会遇到非常大的整数，可能会超过long、甚至long long的范围。这时就需要用不限长度的字符串保存数据，然后进行计算。

最简单的需求就是“大整数相加”，即给定两个字符串形式的非负大整数 num1 和num2 ，计算它们的和。

我们可以把字符串按每个字符一一拆开，相当于遍历整数上的每一个数位，然后通过“乘10叠加”的方式，就可以整合起来了。这相当于算术中的“竖式加法”。

#include<iostream>

using namespace std;

int main()

{

string num1 = "32535943020935527435432875";

string num2 = "9323298429842985843509";

// 用一个空字符串保存结果

string result;

// 获取两数个位的索引

int p1 = num1.size() - 1;

int p2 = num2.size() - 1;

// 设置一个进位标志

int carry = 0;

while (p1 >= 0 || p2 >= 0 || carry > 0)

{

int x = (p1 >= 0) ? (num1[p1] - '0') : 0;

int y = (p2 >= 0) ? (num2[p2] - '0') : 0;

int sum = x + y + carry;

result += (sum % 10 + '0'); // 和的个位写入结果

carry = sum / 10; // 和的十位保存在进位上

// 继续遍历下一位

--p1;

--p2;

}

// 结果需要做翻转

int i = 0, j = result.size() - 1;

while (i < j)

{

char temp = result[j];

result[j] = result[i];

result[i] = temp;

++i;

--j;

}

cout << num1 << " + " << num2 << endl << endl;

cout << " = " << result;

cin.get();

}

#### 6.8.4 旋转图像

旋转图像的需求，在图片处理的过程中非常常见。我们知道对于计算机而言，图像其实就是一组像素点的集合，所以图像旋转的问题，本质上就是一个二维数组的旋转问题。

我们可以给定一个二维数组，用来表示一个图像，然后将它顺时针旋转90°。例如，对于4×4的矩阵：

{

{ 5, 1, 9, 11},

{ 2, 4, 8, 10},

{ 13, 3, 6, 7},

{ 15, 14, 12, 16}

}

旋转之后变为：

{

{ 15, 13, 2, 5},

{ 14, 3, 4, 1},

{ 12, 6, 8, 9},

{ 16, 7, 10, 11}

]

根据数学上矩阵的特性，可以把矩阵A先做转置得到AT，然后再翻转每一行就可以了。

#include<iostream>

using namespace std;

int main()

{

const int n = 4;

int image[n][n] = {

{ 5, 1, 9, 11},

{ 2, 4, 8, 10},

{ 13, 3, 6, 7},

{ 15, 14, 12, 16}

};

// 矩阵转置

for (int i = 0; i < n; i++)

{

for (int j = 0; j <= i; j++)

{

// 以对角线为对称轴，两边互换

int temp = image[i][j];

image[i][j] = image[j][i];

image[j][i] = temp;

}

}

// 每一行翻转

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n / 2; j++)

{

int temp = image[i][j];

image[i][j] = image[i][n-j-1];

image[i][n - j - 1] = temp;

}

}

// 打印输出

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

cout << image[i][j] << "\t";

}

cout << endl;

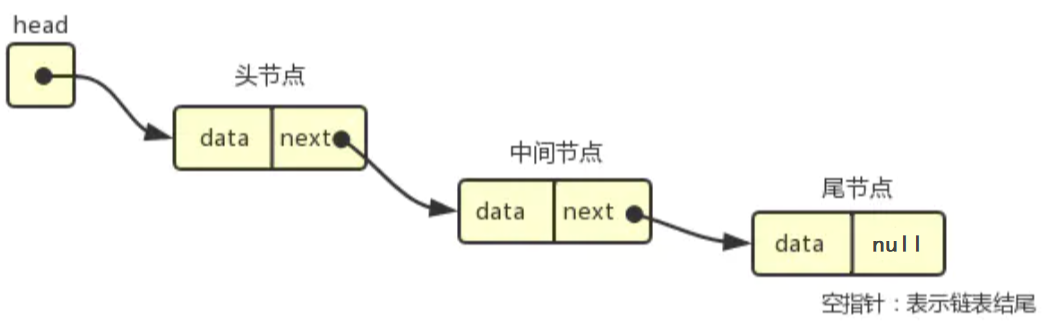
}

cin.get();

}

#### 6.8.5 翻转链表

链表（Linked List）是一种常见的基础数据结构，它是一种线性表，但是并不会像数组那样按顺序存储数据，而是在每一个节点里存指向下一个节点的指针。



对于数组，可以通过下标访问每个元素；如果想要翻转一个数组，只要不停地把头尾元素互换就可以了。

而链表并没有“下标”，所以想要访问元素只能依次遍历；如果要翻转一个链表，关键就在于“next”指针需要反向。

我们可以先定义一个结构体类型ListNode，用来表示链表中的每个节点：

#pragma once

struct ListNode

{

int value;

ListNode\* next;

};

自定义一个头文件list\_node.h，将结构体TreeNode的定义放在里面，这样之后如果需要使用它，就可以直接引入：

#include "list\_node.h"

这里的#prama once是一条预处理指令，表示头文件的内容只被解析一次，不会重复处理。

接下来就可以实现翻转链表的过程了。

#include<iostream>

#include "list\_node.h"

using namespace std;

int main()

{

// 定义一个链表 1->2->3->4->5->NULL

ListNode node5 = { 5, nullptr };

ListNode node4 = { 4, &node5 };

ListNode node3 = { 3, &node4 };

ListNode node2 = { 2, &node3 };

ListNode node1 = { 1, &node2 };

ListNode\* list = &node1;

ListNode\* curr = list;

ListNode\* prev = nullptr;

// 翻转链表

while (curr)

{

ListNode\* temp = curr->next;

curr->next = prev;

prev = curr;

curr = temp;

}

ListNode\* newList = prev;

// 打印链表

ListNode\* np = newList;

while (np)

{

cout << np->value << "\t->\t";

np = np->next;

}

cout << "null" << endl;

cin.get();

}

这里对指向结构体对象的curr指针，需要先解引用，然后取它所指向ListNode对象里的next指针。这个过程本应该写作：

(\*curr).next;

这个写法比较麻烦，所以一般会用另一种简化写法：

curr->next;

这两个写法完全等价。这里的“->”叫做“箭头运算符”，它是解引用和访问成员两个操作的结合；这样就可以很方便地表示“取指针所指向内容的成员”。

## 七、函数

函数其实就是封装好的代码块，并且指定一个名字，调用这个名字就可以执行代码并返回一个结果。

### 7.1 函数基本知识

#### 7.1.1 函数定义

一个完整的函数定义主要包括以下部分：

* 返回类型：调用函数之后，返回结果的数据类型；
* 函数名：用来命名代码块的标识符，在当前作用域内唯一；
* 参数列表：参数表示函数调用时需要传入的数据，一般叫做“形参”；放在函数名后的小括号里，可以有0个或多个，用逗号隔开；
* 函数体：函数要执行的语句块，用花括号括起来。

函数一般都是一个实现了固定功能的模块，把参数看成“输入”，返回结果看成“输出”，函数就是一个输入到输出的映射关系。

我们可以定义一个非常简单的平方函数：

// 平方函数 y = f(x) = x ^ 2

int square(int x)

{

int y = x \* x;

return y;

}

使用流程控制语句return，就可以返回结果。

#### 7.1.2 函数调用

调用函数时，使用的是“调用运算符”，就是跟在函数名后面的一对小括号；括号内是用逗号隔开的参数列表。

这里的参数不是定义时的形参，而是为了初始化形参传入的具体值；为了跟函数定义时的形参列表区分，把它叫作“实参”。

调用表达式的类型就是函数的返回类型，值就是函数执行返回的结果。

#include<iostream>

using namespace std;

// 平方函数 y = f(x^2)

int square(int x)

{

return x \* x;

}

int main()

{

int n = 6;

cout << n << "的平方是：" << square(n) << endl;

cin.get();

}

这里需要注意：

* 实参是形参的初始值，所以函数调用时传入实参，相当于执行了int x = 6的初始化操作；实参的类型必须跟形参类型匹配；
* 实参的个数必须跟形参一致；如果有多个形参，要按照位置顺序一一对应；
* 如果函数本身没有参数，参数列表可以为空，但空括号不能省；
* 形参列表中多个参数用逗号分隔，每个都要带上类型，类型相同也不能省略；
* 如果函数不需要返回值，可以定义返回类型为void；
* 函数返回类型不能是数组或者函数

#### 7.1.3 案例练习

下面几个案例可以作为函数的基本练习。

（1）求两个数的立方和

定义一个函数，输入两个整型参数x、y，返回x3 + y3。

int cubeSum(int x, int y)

{

return pow(x, 3) + pow(y, 3);

}

（2）求阶乘

阶乘的计算公式n! = 1 × 2 × 3 ×…× n，可以用一个循环来实现。定义一个求阶乘的函数，传入一个整数n，返回n!。

// 求阶乘

int factorial(int n)

{

int result = 1;

for (int i = 1; i <= n; i++)

result \*= i;

return result;

}

（3）复制字符串

定义一个函数，传入一个字符串str和一个整数n，将字符串str复制n次后返回。

// 复制字符串

string copyStr(string str, int n)

{

string result;

while (n > 0)

{

result += str;

--n;

}

return result;

}

#### 7.1.4 局部变量的生命周期

之前介绍过变量的作用域，对于花括号内定义的变量，具有“块作用域”，在花括号外就不可见了。函数体都是语句块，而主函数main本身也是一个函数；所以在main中定义的所有变量、所有函数形参和在函数体内部定义的变量，都具有块作用域，统称为“局部变量”。局部变量仅在函数作用域内部可见。

// 函数形参x是局部变量，作用域为函数内部

void f(int x)

{

// 函数内部定义的变量a是局部变量，作用域为函数内部

int a = 10;

}

int main()

{

// 主函数中定义的变量b也是局部变量，作用域为主函数内

int b = 0;

}

在C++中，作用域指的是变量名字的可见范围；变量不可见，并不代表变量所指代的数据对象就销毁了。这是两个不同的概念：

* 作用域：针对名字而言，是程序文本中的一部分，名字在这部分可见；
* 生命周期：针对数据对象而言，是程序在执行过程中，对象从创建到销毁的时间段

基于作用域，变量可以分为“局部变量”和“全局变量”。对于全局变量而言，名字全局可见，对象也只有在程序结束时才销毁。

而对于局部变量代表的数据对象，基于生命周期，又可以分为“自动对象”和“静态对象”。

（1）自动对象

平常代码中定义的普通局部变量，生命周期为：在程序执行到变量定义语句时创建，在程序运行到当前块末尾时销毁。这样的对象称为“自动对象”。

形参也是一种自动对象。形参定义在函数体作用域内，一旦函数终止，形参也就被销毁了。

对于自动对象来说，它的生命周期和作用域是一致的。

（2）静态对象

如果希望延长一个局部变量的生命周期，让它在作用域外依然保留，可以在定义局部变量时加上static关键字；这样的对象叫做“局部静态对象”。

局部静态对象只有局部的作用域，在块外依然是不可见的；但是它的生命周期贯穿整个程序运行过程，只有在程序结束时才被销毁，这一点与全局变量类似。

// 显示自身被调用多少次的函数

int callCount()

{

static int cnt = 0; // 静态对象只会创建一次

cout << "我被调用了" << ++cnt << "次！" << endl;

return cnt;

}

int main()

{

//cout << cnt << endl; // 错误，局部变量在作用域外不可见

callCount();

callCount();

callCount();

}

可以发现，静态对象只在第一次执行到定义语句时创建出来，之后即使函数执行结束，它的值依然保持；下一次函数调用时，不会再次创建、也不会重新赋值，而是直接在之前的值基础上继续叠加。

静态对象和自动对象应用的场景不同，所以它们存放的内存区域也是不一样的。静态对象如果不在代码中做初始化，基本类型会被默认初始化为0值。

#### 7.1.5 函数声明

如果我们将一个函数放在主函数后面，就会出现运行错误：找不到标识符。这是因为函数和变量一样，使用之前必须要做声明。函数只有一个定义，可以定义在任何地方；如果需要调用函数，只需要在调用前做一个声明，告诉编译器“存在这个函数”就可以了。

函数声明的方式，和函数的定义非常相似；区别在于声明时不需要把函数体写出来，用一个分号替代就可以了。

#include<iostream>

using namespace std;

// 声明函数

int square(int x);

int main()

{

int n = 6;

cout << n << "的平方是：" << square(n) << endl;

cin.get();

}

// 定义函数

int square(int x)

{

int y = x \* x;

return y;

return x \* x;

}

事实上，由于没有函数体的执行过程，所以形参的名字也完全不需要，可以省略。可以直接这样声明一个函数：

int square(int);

函数声明中包含了返回类型、函数名和形参类型，这就说明了调用这个函数所需要的所有信息。函数声明也被叫做“函数原型”。

一般情况下，把函数声明放在头文件中会更加方便。

#### 7.1.6 分离式编译和头文件

（1）分离式编译

当程序越来越复杂，我们就会希望代码分散到不同的文件中来做管理。C++支持分离式编译，这就可以把函数单独放在一个文件，独立编译之后链接运行。

比如可以把复制字符串的函数单独保存成一个文件copy\_string.cpp：

#include<string>

using namespace std;

// 复制字符串

string copyStr(string str, int n)

{

string result;

while (n > 0)

{

result += str;

--n;

}

return result;

}

然后只要在主函数调用之前做声明就可以了：

#include<iostream>

using namespace std;

// 声明函数

string copyStr(string, int);

int main()

{

int n = 6;

cout << copyStr("hello ", n) << endl;

cin.get();

}

（2）编写头文件

对于一个项目而言，有些定义可能是所有文件共用的，比如一些常量、结构体/类，以及功能性的函数。于是每次需要引入时，都得做一堆声明——这显然太麻烦了。

一个好方法是，把它们定义在同一个文件中，需要时用一句#include统一引入就可以了，就像使用库一样。这样的文件以.h作为后缀，被称为“头文件”。

比如我们可以把之前的一些功能性的函数（比如求平方、阶乘、复制字符串等），放在一个叫做utils.h的头文件中：

#pragma once

#include<string>

// 平方函数 y = f(x^2)

int square(int x)

{

int y = x \* x;

return y;

return x \* x;

}

// 求立方和

int cubeSum(int x, int y)

{

return pow(x, 3) + pow(y, 3);

}

// 求阶乘

int factorial(int n)

{

int result = 1;

for (int i = 1; i <= n; i++)

result \*= i;

return result;

}

// 复制字符串

std::string copyStr(std::string str, int n);

这里有两点需要说明：

* #pragma once是一条预处理指令，表示这个头文件的内容只会被编译一次，这就避免了多次引入头文件时的重复定义；
* 复制字符串函数copyStr已经在别的文件单独做了定义，这里只要声明就可以；

如果想要使用这些函数，只要在文件中引入头文件即可：

#include "utils.h"

这里文件名没有使用尖括号<>，而是使用了引号；这表示要在当前项目的根目录下寻找文件，而不是到编译器默认的库目录下去找。

### 7.2 参数传递

函数在每次调用时，都会重新创建形参，并且用传入的实参对它进行初始化。形参的类型，决定了形参和实参交互的方式；也决定了函数的不同功能。

可以先回忆一下对变量的初始化：对一个变量做初始化，如果用另一个变量给它赋初值，意味着值的拷贝；也就是说，此后这两个变量各自一份数据，各自管理，互不影响。而如果是定义一个引用，绑定另一个变量做初始化，并不会引发值的拷贝；引用和原变量管理的是同一个数据对象。

int i1 = 0;

int i2 = i1;

i2 = 1; // i1的值仍然是0

int& i3 = i1;

i3 = 10; // i1的值也变为10

参数传递和变量的初始化类似，根据形参的类型可以分为两种方式：传值（value）和传引用（reference）。

#### 7.2.1 传值参数

直接将一个实参的值，拷贝给形参做初始化的传参方式，就被称为“值传递”，这样的参数被称为“传值参数”。之前我们练习过的所有函数，都是采用这种传值调用的方式。

int square(int x)

{

return x \* x;

}

int main()

{

int n = 6;

cout << n << "的平方是：" << square(n) << endl;

}

在上面平方函数的调用中，实参n的值（6）被拷贝给了形参x。

（1）传值的困扰

值传递这种方式非常简单，但是面对这样的需求会有些麻烦：传入一个数据对象，让它经过函数处理之后发生改变。例如，传入一个整数x，调用之后它自己的值要加1。这看起来很简单，但如果直接：

void increase(int x)

{

++x;

}

int main()

{

int n = 6;

increase(n); // n的值不会增加

}

这样做并不能实现需求。因为实参n的值是拷贝给形参x的，之后x的任何操作，都不会改变n。

（2）指针形参

使用指针形参可以解决这个问题。如果我们把指向数据对象的指针作为形参，那么初始化时拷贝的就是指针的值；复制之后的指针，依然指向原始数据对象，这样就可以保留它的更改了。

// 指针形参

void increase(int\* p)

{

++(\*p);

}

int main()

{

int n = 0;

increase( &n ); // 传入n的地址，调用函数后n的值会加1

}

#### 7.2.2 传引用参数

使用指针形参可以解决值传递的问题，不过这种方式函数定义显得有些繁琐，每次调用还需要记住传入变量的地址，使用起来不够方便。

（1）传引用方便函数调用

C++新增了引用的概念，可以替换必须使用指针的场景。采用引用作为函数形参，可以使函数调用更加方便。这种传参方式叫做“传引用参数”。之前的例子就可以改写成：

// 传引用

void increase(int& x)

{

++x;

}

int main()

{

int n = 0;

increase( n ); // 调用函数后n的值会加1

}

由于使用了引用作为形参，函数调用时就可以直接传入n的值，而不用传地址了；x只是n的一个别名，修改x就修改了n。对比可以发现，这段代码相比最初尝试写出的传值实现，只是多了一个引用声明&而已。

（2）传引用避免拷贝

使用引用还有一个非常重要的场景，就是不希望进行值拷贝的时候。实际应用中，很多时候函数要操作的对象可能非常庞大，如果做值拷贝会使得效率大大降低；这时使用引用就是一个好方法。

比如，想要定义一个函数比较两个字符串的长度，需要将两个字符串作为参数传入。因为字符串有可能非常长，直接做值拷贝并不是一个好选择，最好的方式就是传递引用：

// 比较两个字符串的长度

bool isLonger(const string & str1, const string & str2)

{

return str1.size() > str2.size();

}

（3）使用常量引用做形参

在上面的例子中，比较两个字符串长度，并不会更改字符串本身的内容，所以可以把形参定义为常量引用。

这样的好处是，既避免了对数据对象可能的更改，也扩大了调用时能传的实参的范围。因为之前讨论过常量引用的特点，可以用字面值常量对它做初始化，也可以用变量做初始化。

所以在代码中，一般要尽量使用常量引用作为形参。

#### 7.2.3 数组形参

之前已经介绍过，数组是不允许做直接拷贝的，所以如果想要把数组作为函数的形参，使用值传递的方式是不可行的。与此同时，数组名可以解析成一个指针，所以可以用传递指针的方式来处理数组。

比如一个简单的函数，需要遍历int类型数组所有元素并输出，就可以这样声明：

void printArray(const int\*); // 指向int类型常量的指针

void printArray(const int[]);

void printArray(const int[5]);

由于只是遍历输出，不需要修改数组内容，所以这里使用了const。

以上三种声明方式，本质上是一样的，形参的类型都是const int \*；虽然第三种方式指定了数组长度，但由于编译器会把传入的数组名解析成指针，事实上的数组长度还是无法确定的。

这就带来另一个问题：在函数中，遍历元素时怎样确定数组的结束？

（1）规定结束标记

一种简单思路是，规定一个特殊的“结束标记”，遇到这个标记就代表当前数组已经遍历完了。典型代表就是C语言风格的字符串，是以空字符’\0’为结束标志的char数组。

这种方式比较麻烦，而且太多特殊规定也不适合像int这样的数据类型。

（2）把数组长度作为形参

除指向数组的指针外，可以再增加一个形参，专门表示数组的长度，这样就可以方便地遍历数组了。

void printArray(const int\* arr, int size)

{

for (int i = 0; i < size; i++)

cout << arr[i] << "\t";

cout << endl;

}

int main()

{

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

printArray(arr, 6);

}

在C语言和老式的C++程序中，经常使用这种方法来处理数组。

（3）使用数组引用作为形参

之前的方法依赖指针，所以都显得比较麻烦。更加方便的做法，还是用引用来替代指针的功能。

C++允许使用数组的引用作为函数形参，这样一来，引用作为别名绑定在数组上，使用引用就可以直接遍历数组了。

// 使用数组引用作为形参

void printArray(const int(&arr)[6])

{

for (int num : arr)

cout << num << "\t";

cout << endl;

}

int main()

{

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

printArray(arr);

}

这里需要注意的是，定义一个数组引用时需要用括号将&和引用名括起来：

int(&arr)[6] // 正确，arr是一个引用，绑定的是长度为6的int数组

// int & arr[6] // 错误，这是引用的数组，不允许使用

使用数组引用之后，调用函数直接传入数组名就可以了。

#### 7.2.4 可变形参

有时候我们并不确定函数中应该有几个形参，这时就需要使用“可变形参”来表达。

C++中表示可变形参的方式主要有三种：

* 省略符（…）：兼容C语言的用法，只能出现在形参列表的最后一个位置；
* 初始化列表initializer\_list：跟vector类似，也是一种标准库模板类型；initializer\_list对象中的元素只能是常量值，不能更改；
* 可变参数模板：这是一种特殊的函数，后面会详细介绍。

### 7.3 返回类型

函数可以通过return语句，终止函数的执行并“返回”函数调用的地方；并且可以给定返回值。返回值的类型由函数声明时的“返回类型”决定。

return语句可以有两种形式：

return; // 直接返回，无返回值

return *表达式*; // 返回表达式的值

#### 7.3.1 无返回值

当函数返回类型为void时，表示函数没有返回值。可以在函数中需要返回时直接执行 return语句，也可以不写。因为返回类型为void的函数执行完最后一句，会自动加上return返回。

例如，可以将之前“两元素值互换”的代码，包装成一个函数。可以先做一个判断，如果两者相等就直接返回，这样可以提高运行效率。

// 元素互换

void swap(int& x, int& y)

{

if (x == y)

return; // 不需要交换，直接返回

int temp = x;

x = y;

y = temp;

}

这里判断如果元素相等就直接返回，有些类似于流程控制中的break。最后一句代码后面省略了return。

#### 7.3.2 有返回值

如果函数返回类型不为void，那么函数必须执行return，并且每条return必须返回一个值。返回值的类型应该跟函数返回类型一致，或者可以隐式转换为一致。

（1）函数返回值的原理

函数在调用点会创建一个“临时量”，用来保存函数调用的结果。当使用return语句返回时，就会用返回值去初始化这个临时量。所以返回值的相关规则，跟变量或者形参的初始化是一致的。

之前写过一个“比较字符串长度”的isLonger函数，我们可以稍作修改，让它可以返回较长的那个字符串：

// 字符串比较长度，返回较长的

string longerStr(const string& str1, const string& str2)

{

return str1.size() > str2.size() ? str1 : str2;

}

int main()

{

string str1 = "hello world!", str2 = "c++ is interesting!";

cout << longerStr(str1, str2) << endl;

}

调用这个函数，经过判断发现str2较长，这时执行return将返回str2。由于返回类型是string，所以将用str2对一个string临时量做初始化，执行的是值拷贝。最终返回的值，是str2的一个副本。

（2）返回引用类型

对于string对象，显然做值拷贝并不高效。所以我们依然可以借鉴之前的经验，使用引用类型来做返回值的传递，这样就可以避免值拷贝。

// 返回一个string常量对象的引用，不做值拷贝

const string & longerStr(const string& str1, const string& str2)

{

return str1.size() > str2.size() ? str1 : str2;

}

这里我们同样把返回值定义成了常量引用，方式和作用跟形参完全一样。

上面函数返回的是形参str1或者str2的引用；而函数中的形参本身又是引用类型，所以最终是实参对象的引用。

而如果返回的是一个函数内局部变量的引用，比如：

const string & f()

{

string str = "test";

return str;

}

这样做是不安全的：因为str是函数内部的局部对象，函数执行完成后就销毁了；而返回值是它的引用，相当于引用了一个不存在的对象，这可能会导致无法预料的问题。

所以，函数返回引用类型时，不能返回局部对象的引用；同样道理，也不应该返回指向局部对象的指针。

（3）返回类对象后连续调用

如果函数返回一个类的对象，那么我们可以继续调用这个对象的成员函数，这样就形成了“链式调用”。例如：

longerStr(str1, str2).size();

调用运算符，和访问对象成员的点运算符优先级相同，并且满足左结合律。所以链式调用就是从左向右依次调用，代码可读性会更高。

#### 7.3.3 主函数的返回值

主函数main是一个特殊函数，它是我们执行程序的入口。所以C++中对主函数的返回值也有特殊的规定：即使返回类型不是void，主函数也可以省略return语句。如果主函数执行到结尾都没有return语句，编译器就会自动插入一条：

return 0;

主函数的返回值可以看做程序运行的状态指示器：返回0表示运行成功；返回非0值则表示失败。非0值具体的含义依赖机器决定。

这也是为什么之前我们在主函数中都可以不写return。

#### 7.3.3 返回数组指针

与形参的讨论类似，由于数组“不能拷贝”的特点，函数也无法直接返回一个数组。同样的，我们可以使用指针或者引用来实现返回数组的目标；通常会返回一个数组指针。

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

int\* pa[5]; // 指针数组，pa是包含5个int指针的数组

int(\*ap)[5] = &arr; // 数组指针，ap是一个指针，指向长度为5的int数组

int(\*fun(int x))[5]; // 函数声明，fun返回值类型为数组指针

这里对于函数fun的声明，我们可以进行层层解析：

* fun(int x) ：函数名为fun，形参为int类型的x；
* ( \* fun(int x) )：函数返回的结果，可以执行解引用操作，说明是一个指针；
* ( \* fun(int x) )[5]：函数返回结果解引用之后是一个长度为5的数组，说明返回类型是数组指针；
* int ( \* fun(int x) )[5]：数组中元素类型为int

数组指针的定义比较繁琐，为了简化这个定义，我们可以使用关键字typedef来定义一个类型的别名：

typedef int arrayT[5]; // 类型别名，arrayT代表长度为5的int数组

arrayT\* fun2(int x); // fun2的返回类型是指向arrayT的指针

C++ 11新标准还提供了另一种简化方式，用一个->符号跟在形参列表后面，再把类型单独提出来放到最后。这种方式叫做“尾置返回类型”。

auto fun3(int x) -> int(\*)[5]; // 尾置返回类型

因为返回类型放到了末尾，所以前面的类型用了自动推断的auto。

### 7.4 递归

如果一个函数调用了自身，这样的函数就叫做“递归函数”（recursive function）。

#### 7.4.1 递归的实现

递归是调用自身，如果不加限制，这个过程是不会结束的；函数永远调用自己下去，最终会导致程序栈空间耗尽。所以在递归函数中，一定会有某种“基准情况”，这个时候不会调用自身，而是直接返回结果。基准情况的处理保证了递归能够结束。

递归是不断地自我重复，这一点和循环有相似之处。事实上，递归和循环往往可以实现同样的功能。

比如之前求阶乘的函数，我们可以用递归的方式重新实现：

#include<iostream>

using namespace std;

// 递归方式求阶乘

int factorial(int n)

{

if (n == 1)

return 1;

else

return factorial(n - 1) \* n;

}

int main()

{

cout << "5! = " << factorial(5) << endl;

cin.get();

}

这里我们的基准情况是n == 1，也就是当n不断减小，直到1时就结束递归直接返回。5的阶乘具体计算流程如下：



因为递归至少需要额外的栈空间开销，所以递归的效率往往会比循环低一些。不过在很多数学问题上，递归可以让代码非常简洁。

#### 7.4.2 经典递归——斐波那契数列

斐波那契数列（Fibonacci sequence），又称[黄金分割](https://baike.baidu.com/item/%E9%BB%84%E9%87%91%E5%88%86%E5%89%B2/115896)数列，指的是这样一个数列：

1, 1, 2, 3, 5, 8, 13, 21, 34, …

它的规律是：当前数字，是之前两个数字之和。在数学上，斐波那契数列被以递推的方法定义：

F(0)=1，F(1)=1, F(n) = F(n - 1) + F(n - 2)（n ≥ 2，n ∈ N\*）

这天然适合使用递归实现：

#include<iostream>

using namespace std;

int fib(int n)

{

if (n == 1 || n == 2)

return 1;

return fib(n - 2) + fib(n - 1);

}

int main()

{

cout << "fib(9) = " << fib(9) << endl;

cin.get();

}

### 7.5 应用案例

#### 7.5.1 二分查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法，前提是数据对象必须先排好序。二分查找事实上采用的是一种“分治”策略，它充分利用了元素间的次序关系。

#include<iostream>

using namespace std;

// 可以递归调用的二分查找

int search(const int(&a)[10], int start, int end, int target)

{

// 基准情况：目标值超出范围，或者start > end，说明没有找到

if ( target < a[start] || target > a[end] || start > end)

return -1;

// 取二分的中间坐标

int mid = (start + end) / 2;

// 比较中间值和目标值的大小

if (a[mid] == target)

return mid; // 找到了

else if (a[mid] > target)

return search(a, start, mid - 1, target); // 比目标值大，在更小的部分找

else

return search(a, mid + 1, end, target); // 比目标值小，在更大的部分找

}

int main()

{

int arr[10] = { 1,2,3,4,5,6,9,12,25,38 };

int key = 25;

int size = sizeof(arr) / sizeof(arr[0]);

int result = search(arr, 0, size - 1, key);

result == -1

? cout << "在数组中没有找到" << key << "!" << endl

: cout << "在数组中找到" << key << "，索引下标为：" << result << endl;

cin.get();

}

#### 7.5.2 快速排序

之前介绍过两种对一组数据进行排序的算法：选择排序和冒泡排序，它们都需要使用两层for循环遍历数组，效率较低。一种巧妙的改进思路是：通过一次扫描，将待排记录分隔成独立的两部分，其中一部分的值全比另一部分的小；接下来分别对这两部分继续进行排序，最终全部排完。这种算法更加高效，被称为“快速排序”。



可以看出，快排也应用了分治思想，一般会用递归来实现。

#include<iostream>

using namespace std;

void quickSort(int(&)[10], int, int);

int partition(int(&)[10], int, int);

void printArr(const int(&)[10]);

void swap(int(&)[10], int, int);

int main()

{

int arr[10] = { 23, 45, 18, 6, 11, 19, 22, 18, 12, 9 };

printArr(arr);

int size = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, size - 1);

printArr(arr);

cin.get();

}

// 快速排序

void quickSort(int(&a)[10], int start, int end)

{

// 基准情况

if (start >= end)

return;

// 分区，返回分区点下标

int mid = partition(a, start, end);

// 递归调用，分别对两部分继续排序

quickSort(a, start, mid - 1);

quickSort(a, mid + 1, end);

}

// 按照pivot分区的函数

int partition(int(&a)[10], int start, int end)

{

// 选取一个分区的“支点”

int pivot = a[start];

int left = start, right = end;

while (left < right)

{

// 分别从左右两边遍历数组

while (a[left] <= pivot && left < right)

++left;

while (a[right] >= pivot && left < right)

--right;

// 左右互换

swap(a, left, right);

}

if (a[left] < pivot) {

swap(a, start, left);

return left;

}

else if (a[left] > pivot)

{

swap(a, start, left - 1);

return left - 1;

}

}

// 数组中两元素互换的函数

void swap(int(&a)[10], int i, int j)

{

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

// 打印输出一个数组

void printArr(const int(&a)[10])

{

for (int num : a)

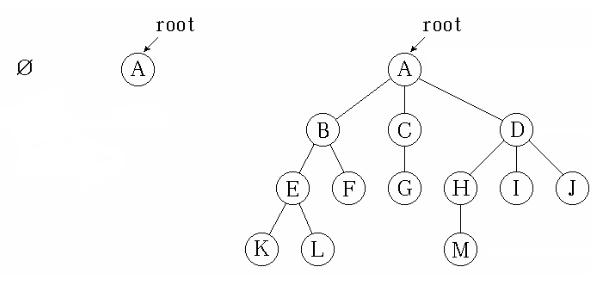
cout << num << "\t";

cout << endl;

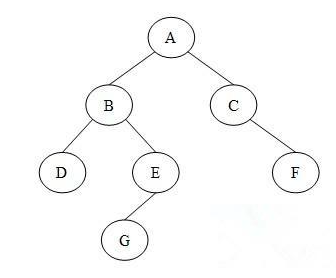
}

#### 7.5.3 遍历二叉树

跟数组不同，树是一种非线性的数据结构，是由n（n >=0）个节点组成的有限集合。如果n==0，树为空树。如果n>0，树有一个特定的节点，叫做根节点（root）。



对于树这种数据结构，使用最频繁的是二叉树。每个节点最多只有2个子节点的树，叫做二叉树。二叉树中，每个节点的子节点作为根的两个子树，一般叫做节点的左子树和右子树。



我们可以为树的节点定义一种结构体类型，而且为了方便以后在不同的文件中使用，还可以自定义一个头文件tree\_node.h，将结构体TreeNode的定义放在里面：

#pragma once

#include<string>

using namespace std;

struct TreeNode

{

string name;

TreeNode\* left;

TreeNode\* right;

};

在别的文件中，如果想要使用TreeNode这个结构体，我们只要引入就可以：

#include "TreeNode.h"

对于树的遍历，主要有这样三种方式：

* 先序遍历：先访问根节点，再访问左子树，最后访问右子树；
* 中序遍历：先访问左子树，再访问根节点，最后访问右子树；
* 后序遍历：先访问左子树，再访问右子树，最后访问根节点。

这种遍历方式就隐含了“递归”的思路：左右子树本身又是一棵树，同样需要按照对应的规则来遍历。

我们可以先单独创建一个文件print\_tree.cpp，实现二叉树的遍历方法：

#include<iostream>

#include "tree\_node.h"

// 先序遍历打印二叉树

void printTreePreOrder(TreeNode\* root)

{

// 基准情况，如果是空树，直接返回

if (root == nullptr) return;

//cout << (\*root).name << "\t";

cout << root->name << "\t";

// 递归打印左右子树

printTreePreOrder(root->left);

printTreePreOrder(root->right);

}

// 中序遍历打印二叉树

void printTreeInOrder(TreeNode\* root)

{

// 基准情况，如果是空树，直接返回

if (root == nullptr) return;

printTreeInOrder(root->left);

cout << root->name << "\t";

printTreeInOrder(root->right);

}

// 后序遍历打印二叉树

void printTreePostOrder(TreeNode\* root)

{

// 基准情况，如果是空树，直接返回

if (root == nullptr) return;

printTreePostOrder(root->left);

printTreePostOrder(root->right);

cout << root->name << "\t";

}

然后将这些函数的声明也放到头文件tree\_node.h中：

void printTreePreOrder(TreeNode\* root);

void printTreeInOrder(TreeNode\* root);

void printTreePostOrder(TreeNode\* root);

接下来就可以在代码中实现具体的功能了：

#include<iostream>

#include "tree\_node.h"

int main()

{

// 定义一棵二叉树

TreeNode nodeG = {"G", nullptr, nullptr};

TreeNode nodeF = { "F", nullptr, nullptr };

TreeNode nodeE = { "E", &nodeG, nullptr };

TreeNode nodeD = { "D", nullptr, nullptr };

TreeNode nodeC = { "C", nullptr, &nodeF};

TreeNode nodeB = { "B", &nodeD, &nodeE };

TreeNode nodeA = { "A", &nodeB, &nodeC };

TreeNode\* tree = &nodeA;

printTreePreOrder(tree);

cout << endl << endl;

printTreeInOrder(tree);

cout << endl << endl;

printTreePostOrder(tree);

cin.get();

}

## 八、函数高阶

函数是模块化编程思想的重要体现，相对于传统的C语言，C++还提供了很多新的函数特性。这一章我们就来深入探讨一下函数的高级用法以及在C++中的新特性。

### 8.1 内联函数

内联函数是C++为了提高运行速度做的一项优化。

函数让代码更加模块化，可重用性、可读性大大提高；不过函数也有一个缺点：函数调用需要执行一系列额外操作，会降低程序运行效率。

为了解决这个问题，C++引入了“内联函数”的概念。使用内联函数时，编译器不再去做常规的函数调用，而是把它在调用点上“内联”展开，也就是直接用函数代码替换了函数调用。

#### 8.1.1 内联函数的定义

定义内联函数，只需要在函数声明或者函数定义前加上inline关键字。

例如之前写过的函数：比较两个字符串、并返回较长的那个，就可以重写为内联函数：

inline const string& longerStr(const string& str1, const string& str2)

{

return str1.size() > str2.size() ? str1 : str2;

}

当我们试图打印输出调用结果时：

cout << longerStr(str1, str2) << endl;

编译器会自动把它展开为：

cout << (str1.size() > str2.size() ? str1 : str2) << endl;

这样就大大提高了运行效率。

#### 8.1.2 内联函数和宏

内联函数是C++新增的特性。在C语言中，类似功能是通过预处理语句#define定义“宏”来实现的。

然而C中的宏本身并不是函数，无法进行值传递；它的本质是文本替换，我们一般只用宏来定义常量。用宏实现函数的功能会比较麻烦，而且可读性较差。所以在C++中，一般都会用内联函数来取代C中的宏。

### 8.2 默认实参

在有些场景中，当调用一个函数时它的某些形参一般都会被赋一个固定的值。为了简单起见，我们可以给它设置一个“默认值”，这样就不用每次都传同样的值了。

这种会反复出现的默认值，称为函数的“默认实参”。当调用一个有默认实参的函数时，这个实参可以省略。

#### 8.2.1 定义带默认实参的函数

我们用一个string对象表示学生基本信息，调用函数时应传入学生的姓名、年龄和平均成绩。对于这些参数，我们可以指定默认实参：

// 默认实参

string stuInfo(string name = "", int age = 18, double score = 60)

{

string info = "学生姓名：" + name + "\t年龄：" +

to\_string(age) + "\t平均成绩：" + to\_string(score);

return info;

}

定义默认实参，形式上就是给形参做初始化。这里在整合学生信息时，使用了运算符+进行字符串拼接，并且调用to\_string函数将age和score转换成了string。

这里需要注意，一旦某个形参被定义了默认实参，那它后面的所有形参都必须有默认实参。也就是说，所有默认实参的指定，应该在形参列表的末尾。

// 错误，默认实参不在形参列表末尾

//string stuInfo(string name = "", int age = 18, double score);

// 正确，可以前面的形参没有默认实参

string stuInfo(string name, int age = 18, double score = 60);

#### 8.2.2 使用默认实参调用函数

函数调用时，如果对某个形参不传实参，那么它初始化时用的就是默认实参的值。由于之前所有形参都定义了默认实参，因此可以用不同的传参方式调用函数：

cout << stuInfo() << endl; // ""，18, 60.0

cout << stuInfo("张三") << endl; // "张三"，18, 60.0

cout << stuInfo("李四", 20) << endl; // "李四"，20, 60.0

cout << stuInfo("王五", 22, 85.5) << endl; // "王五"，22, 85.5

//cout << stuInfo(19, 92.5) << endl; // 错误，不能跳过前面的形参给后面传值

//cout << stuInfo(, , 59.5) << endl; // 错误，只能省略末尾的形参

可以看到，默认实参定义时要优先放到形参列表的尾部；而调用时，只能省略尾部的参数，不能跳过前面的形参给后面传值。

### 8.3 函数重载

在C++中，同一作用域下，同一个函数名是可以定义多次的，前提是形参列表不同。这种名字相同但形参列表不同的函数，叫做“重载函数”。这是C++相对C语言的重大改进，也是面向对象的基础。

#### 8.3.1 定义重载函数

在上一章数组形参部分，我们曾经实现过几个不同的打印数组的函数，它们是可以同时存在的：

// 使用指针和长度作为形参

void printArray(const int\* arr, int size)

{

for (int i = 0; i < size; i++)

cout << arr[i] << "\t";

cout << endl;

}

// 使用数组引用作为形参

void printArray(const int(&arr)[6])

{

for (int num : arr)

cout << num << "\t";

cout << endl;

}

int main()

{

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

printArray(arr, 6); // 传入两个参数，调用第一种实现

printArray(arr); // 传入一个参数，调用第二种实现

}

这里需要注意：

* 重载的函数，应该在形参的数量或者类型上有所不同；
* 形参的名称在类型中可以省略，所以只有形参名不同的函数是一样的；
* 调用函数时，编译器会根据传递的实参个数和类型，自动推断使用哪个函数；
* 主函数不能重载

#### 8.3.2 有const形参时的重载

当形参有const修饰时，要区分它对于实参的要求到底是什么，是否要进行值的拷贝。如果是传值参数，传入实参时会发生值的拷贝，那么实参是变量还是常量其实是没有区别的：

void fun(int x);

void fun(const int x); // int常量做形参，跟不加const等价

void fun2(int\* p);

void fun2(int\* const p); // 指针常量做形参，也跟不加const等价

这种情况下，const不会影响传入函数的实参类型，所以跟不加const的定义是一样的；这叫做“顶层const”。这时两个函数相同，无法进行函数重载。

另一种情况则不同，那就是传引用参数。这时如果有const修饰，就成了“常量的引用”；对于一个常量，只能用常量引用来绑定，而不能使用普通引用。

类似地，对于一个常量的地址，只能由“指向常量的指针”来指向它，而不能用普通指针。

void fun3(int &x);

void fun3(const int & x); // 形参类型是常量引用，这是一个新函数

void fun4(int\* p);

void fun4(const int\* p); // 形参类型是指向常量的指针，这是一个新函数

这种情况下，const限制了间接访问的数据对象是常量，这叫做“底层const”。当实参是常量时，不能对不带const的引用进行初始化，所以只能调用常量引用做形参的函数；而如果实参是变量，就会优先匹配不带const的普通引用：这就实现了函数重载。

#### 8.3.3 函数匹配

如果传入的实参跟形参类型不同，只要能通过隐式类型转换变成需要类型，函数也可以正确调用。那假如有几个不同的重载函数，它们的形参类型可以进行自动转换，这时传入实参应该调用哪个函数呢？例如：

void f();

void f(int x);

void f(int x, int y);

void f(double x, double y = 1.5);

f(3.14); // 应该调用哪个函数？

确定到底调用哪个函数的过程，叫做“函数匹配”。

（1）候选函数

函数匹配的第一步，就是确定“候选函数”，也就是先找到对应的重载函数集。候选函数有两个要求：

* 与调用的函数同名
* 函数的声明，在函数的调用点是可见的

所以上面的例子中，一共有4个叫做f的函数，它们都是候选函数。

（2）可行函数

接下来需要从候选函数中，选出跟传入的实参匹配的函数。这些函数叫做“可行函数”。可行函数也有两个要求：

* 形参个数与调用传入的实参数量相等
* 每个实参的类型与对应形参的类型相同，或者可以转换成形参的类型

上面的例子中，传入的实参只有一个，是一个double类型的字面值常量，所以可以排除 f() 和 f(int, int) 。而剩下的 f(int) 和 f(double, double = 1.5) 都是匹配的，所以有2个可行函数。

（3）寻找最佳匹配

最后就是在可行函数中，选择最佳匹配。简单来说，实参类型与形参类型越接近，它们就匹配得越好。所以，能不进行转换就实际匹配的，要优于需要转换的。

上面的例子中，f(int) 必须要将double类型的实参转换成int，而f(double, double = 1.5) 不需要，所以后者是最佳匹配，最终调用的就是它。第二个参数会由默认实参1.5来填补。

（4）多参数的函数匹配

如果实参的数量不止一个，那么就需要逐个比较每个参数；同样，类型能够精确匹配的要优于需要转换的。这时寻找最佳匹配的原则如下：

* 如果可行函数的所有形参都能精确匹配实参，那么它就是最佳匹配
* 如果没有全部精确匹配，那么当一个可行函数所有参数的匹配，都不比别的可行函数差、并且至少有一个参数要更优，那它就是最佳匹配

（5）二义性调用

如果检查所有实参之后，有多个可行函数不分优劣、无法找到一个最佳匹配，那么编译器会报错，这被称为“二义性调用”。例如：

f(10, 3.14); // 二义性调用

这时的可行函数为 f(int, int) 和 f(double, double = 1.5)。第一个实参为int类型，f(int, int)占优；而第二个实参为double类型，f(double, double = 1.5)占优。这时两个可行函数分不出胜负，于是就会报二义性调用错误。

#### 8.3.4 重载与作用域

重载是否生效，跟作用域是有关系的。如果在内层、外层作用域分别声明了同名的函数，那么内层作用域中的函数会覆盖外层的同名实体，让它隐藏起来。

不同的作用域中，是无法重载函数名的。

#include<iostream>

using namespace std;

void print(double d)

{

cout << "d: " << d << endl;

}

void print(string s)

{

cout << "s: " << s << endl;

}

int main()

{

// 调用之前做函数声明

void print(int i);

print(10);

print(3.14); // 将3.14转换为3，然后调用

//print("hello"); // 错误，找不到对应参数的函数定义

cin.get();

}

void print(int i)

{

cout << "i: " << i << endl;

}

如果想让函数正确地重载，应该把函数声明放到同一作用域下：

#include<iostream>

using namespace std;

// 作用域和重载测试

void print(int i)

{

cout << "i: " << i << endl;

}

void print(double d)

{

cout << "d: " << d << endl;

}

void print(string s)

{

cout << "s: " << s << endl;

}

int main()

{

print(10);

print(3.14);

print("hello");

cin.get();

}

### 8.5 函数指针

一类特殊的指针，指向的不是数据对象而是函数，这就是“函数指针”。

#### 8.5.1 声明函数指针

函数指针本质还是指针，它的类型和所指向的对象类型有关。现在指向的是函数，函数的类型是由它的返回类型和形参类型共同决定的，跟函数名、形参名都没有关系。

例如之前写过的函数：

string stuInfo(string name = "", int age = 18, double score = 60)

{

string info = "学生姓名：" + name + "\t年龄：" +

to\_string(age) + "\t平均成绩：" + to\_string(score);

return info;

}

它的类型就是：string(string, int, double)。

如果要声明一个指向它的指针，只要把原先函数名的位置填上指针就可以了：

string (\* fp) (string, int, double); // 一个函数指针

这里要注意，指针两侧的括号必不可少。如果去掉括号，

string \*fp(string, int, double); // 这是一个函数，返回值为指向string的指针

这就变成了一个返回string \*类型的函数。

更加复杂的例子也是一样，例如之前写过的比较字符串长度的函数：

const string& longerStr(const string& str1, const string& str2)

{

return str1.size() > str2.size() ? str1 : str2;

}

对应类型的函数指针就是：

const string &(\*fp) (const string &, const string &);

#### 8.5.2 使用函数指针

当一个函数名后面跟调用操作符（小括号），表示函数调用；而单独使用函数名作为一个值时，函数会自动转换成指针。这一点跟数组名类似。

所以我们可以直接使用函数名给函数指针赋值：

fp = longerStr; // 直接将函数名作为指针赋给fp

fp = &longerStr; // 取地址符是可选的，和上面没有区别

也可以加上取地址符&，这和不加&是等价的。

赋值之后，就可以通过fp调用函数了。fp做解引用可以得到函数，而这里解引用符\*也是可选的，不做解引用同样可以直接表示函数。

cout << fp("hello", "world") << endl;

cout << (\*fp)("C++", "is good") << endl;

所以这里能够看出，函数指针完全可以当做函数来使用。

在对函数指针赋值时，函数的类型必须精确匹配。当然，函数指针也可以赋nullptr，表示空指针，没有指向任何一个函数。

#### 8.5.3 函数指针作为形参

有了指向函数的指针，就给函数带来了更加丰富灵活的用法。比如，可以将函数指针作为形参，定义在另一个函数中。也就是说，可以定义一个函数，它以另一个函数类型作为形参。当然，函数本身不能作为形参，不过函数指针完美地填补了这个空缺。这一点上，函数跟数组非常类似。

void selectStr(const string& s1, const string& s2, const string & fp(const string&, const string&));

void selectStr(const string& s1, const string& s2, const string & (\*fp) (const string&, const string&));

同样，上面两种形式是等价的，\*是可选的。

很明显，对于函数类型和函数指针类型来说，这样的定义太过复杂，所以有必要使用typedef做一个类型别名的声明。

// 类型别名

typedef const string& Func(const string&, const string&); // 函数类型

typedef const string& (\*FuncP)(const string&, const string&); // 函数指针类型

当然，还可以用C++ 11提供的decltype函数直接获取类型，更加简洁：

typedef decltype(longerStr) Func2;

typedef decltype(longerStr) \*FuncP2;

这样一来，声明函数指针做形参的新函数，就非常方便了：

void selectStr(const string&, const string&, Func);

#### 8.5.4 函数指针作为返回值

类似地，函数不能直接返回另一个函数，但是可以返回函数指针。所以可以将函数指针作为另一个函数的返回值。

这里需要注意的是，这种场景下，函数的返回类型必须是函数指针，而不能是函数类型。

// 函数指针作为返回值

FuncP fun(int);

//Func fun2(int); // 错误，不能直接返回函数

Func\* fun2(int);

// 尾置返回类型

auto fun3(int) -> FuncP;

另外也可以使用尾置返回类型的方式，指定返回函数指针类型。