

# 成为 ACM/ICPC 的好基友

## New Guys In ACM/ICPC

石博天      张怀文      李政霖      薛嘉宾      杜志浩

2013 年 8 月



# 前言

## ACM 与 ICPC

ACM/ICPC 是有美国计算机协会（ACM）主办的“ACM 国际大学生程序设计竞赛”。这是一场展示大学生创新能力、团队精神和在压力下编写程序、分析和解决问题能力的年度竞赛，已经有 30 多年的历史。是计算机领域大学生竞赛中颇具含金量的一项赛事！

ACM/ICPC 以团队的形式代表各学校参加比赛，每队由 3 名队员组成。比赛期间，每队使用 1 台电脑，在规定的时间内使用规定的编程语言完成规定的问题。程序编写完成后提交给裁判处运行，运行结果将会随时返回给队员。期间如果提交答案被判定为错误，会被罚在总时间上增加 20 分钟罚时。若该题最终回答正确，则对题数加 1，同时罚时计入总时间。如果直至比赛结束也未能最对，则该题不计时。

比赛最终的获胜者为解答题目数量最多的队伍。当数目一样多的时候，则按照完成问题所用时间排名。

与 IOI 等其他的计算机程序设计竞赛相比，ACM/ICPC 的特点就是题量大，队员多，电脑少。所以除了扎实的专业水平，良好的团队协作和优秀的心理素质也是非常重要的。

参赛队需要通过预选赛层层选拔，过五关斩六将，才有机会代表参加全球总决赛。并且获得丰厚的奖励。

ACM/ICPC 应该算是大学生所能参加比赛中含氧量最高的比赛了，因为它真正考验了参赛选手的各方面的综合素质，而不仅仅是背书的能力。只有通过大量的训练、锻炼出优秀的计算思维才能取得优秀的成绩。

## 关于本书

本书是为内蒙古大学计算机学院 2013 级 ACM/ICPC 新手训练营所编写的系列教程。编写者是一群来自于内蒙古大学精英学生开发者联盟的靠谱青年。

本教程力求用“不靠谱”的方式去介绍很多“靠谱”的事情。所以我们会在不影响大家阅读兴趣的情况下，尽量保证内容的严谨性！

关于书中内容：第零章的开始之前，为大家介绍了一些学习计算机程序设计所必备的基础知识。第一章的主要内容是 C++ 程序设计语言的快速入门。第二章将会为大家讲解一些非常简单，但不失趣味性的入门算法。第三章主要是各种常见、常用的排序算法。第四章将会为大家详细介绍几种数据结构——用来对现实问题进行抽象描述的工具。第五章将会讲解一些参加 ACM/ICPC 这样类型比赛的一些基本算法。第六章将会讲解一些诸如动态规划、图论等的高级算法。第七章将主要会围绕着计算机算法领域的很多数学算法进行讲解。

当然，写书并不是一件简单的事情，尤其是对一帮平均年龄大 (bù) 约 (dào) 20 岁的大学生来说。所以书中难 (kǎn) 免 (dìng) 会出现各种错误。希望大家在看到书中错误的时候能够及时指出！

本书的编写者有（按照章节编写顺序）：石博天、张怀文、李政霖、薛家斌、杜志浩。本书勘误请联系邮箱：[contact@imudges.com](mailto:contact@imudges.com)。希望本书能够打开大家走向高手之路的大门！

# 目录

<b>第零章 开始之前</b>	<b>1</b>
0.1 从语言的发展了解计算机 . . . . .	1
0.2 内存、程序与数据 . . . . .	5
0.3 算法与算法的复杂性 . . . . .	7
0.4 数据结构 . . . . .	10
0.5 二进制 . . . . .	11
0.6 习题 . . . . .	11
<b>第一章 码中一窥 C++</b>	<b>13</b>
1.1 Helloworld++ . . . . .	15
1.1.1 Code . . . . .	15
1.1.2 Analyze . . . . .	16
1.1.3 Result . . . . .	21
1.1.4 Try . . . . .	23
1.2 使用变量与数据类型 . . . . .	23
1.2.1 Code . . . . .	24
1.2.2 Analyze . . . . .	26
1.2.3 Result . . . . .	27
1.2.4 Try . . . . .	31
1.3 字符类型 . . . . .	32
1.3.1 Code . . . . .	32
1.3.2 Analyze . . . . .	33
1.3.3 Result . . . . .	34

1.3.4	Try . . . . .	35
1.4	布尔类型 . . . . .	35
1.4.1	Code . . . . .	37
1.4.2	Analyze . . . . .	38
1.4.3	Result . . . . .	39
1.4.4	Try . . . . .	40
1.5	运算符与表达式 . . . . .	41
1.5.1	Code . . . . .	42
1.5.2	Analyze . . . . .	44
1.5.3	Result . . . . .	45
1.5.4	Try . . . . .	47
1.6	语句与控制 . . . . .	47
1.6.1	Code . . . . .	48
1.6.2	Analyze . . . . .	48
1.6.3	Result . . . . .	48
1.6.4	Try . . . . .	48

# 第零章 开始之前

## 0.1 从语言的发展了解计算机

计算机，顾名思义，就是一台可以计算的机器。一般来说，作为一台冷冰冰的机器，它是没有智能的。即“只能根据外部的指令做出预先设定好的几种动作”。这些动作实际上是对计算机的五大基本部件进行操作。

这五大部件有“输入设备”、“输出设备”、“运算器”、“控制器”、“存储器”。输入设备和输出设备顾名思义，就是将数据输入到计算机和从计算机中得到结果的设备。而运算器、控制器和存储器我们稍后会谈到。

利用这五大部件组成的计算机，我们称之为基于“冯诺依曼”结构。这五大部件可不是凭空捏造，而是来之有据的！

早在 1930 年代，人就开始构思一种还不存在的、能执行命令的机器——图灵机。这种机器的发明来源是模仿人类利用纸和笔进行数学运算而得来的。

想要进行数学运算，我们需要一些工具：首先是答题纸，我们需要用答题纸来记述每一步的结果。然后需要草稿纸来暂存计算的中间步骤。然后我们需要笔在纸上写字，需要眼睛来阅读纸上的内容。然后我们需要一系列的基本运算规则例如四则运算、公式定理。只需要这些东西我们就可以构建出无穷的数学世界！

图灵机就是仿照上面描述而虚构出的一种设备。它有一张无限长的纸带（如同纸）：纸带被划分出连续的一串格子，每个格子都用唯一的编号来组织在一起。然后还有一个读写头（笔和眼）：通过读写头我们可以阅读或者修改纸带上的数据。同时还需要一套控制规则（公式定理运算律）：根据当前机器所处状态和当前读写头所指的格子上的符号来确定读写头的下一

步动作，并且改变状态寄存器的值。它还有一个状态寄存器（草稿纸）：用来保存图灵机当前所处的状态，例如：状态 -正常运行；状态 -停机。等等。

图灵机就是这样一台机器，图灵认为它能模拟人类所能进行的任何计算过程。

1945 年，冯·诺依曼发表文章——“关于 EDVAC 的报告草案”。该方案描述了现代电子计算机由“运算器”、“控制器”、“存储器”、“输入设备”和“输出设备”五大部分组成，并且描述了这五大部分之间的关系。并且设计了这台基于二进制的 EDVAC。它的功率为 56KW，占地面积近 50 平方米，重近 8 吨，需要三十个技术人员同时操作。

后来各大公司生产的通用计算机，速度越来越快、功能越来越强大、编程方式越来越简单，但是其根本结构还是未能突破这种几十年前冯诺依曼的设计。

那么，向机器发号施令、协同五大部件工作的人就是程序员。。

而计算机程序设计其实就是程序员将现实的问题转换成计算机可以理解的方式，并且让计算机去执行自己的指令，然后得到结果。

也就是说，计算机程序其实就是一种能够被计算机所识别的指令的序列。计算机只能根据程序中所规定的内容一步一步地进行运转。

那么这种指令是如何被识别被执行的呢？这就要说起计算机的核心部件——CPU 了。CPU 是 Central Processing Unit 的缩写，中文名叫做“中央处理器”。CPU 内包含了“运算器”和“控制器”两大部分。它是计算机的核心部件。我们可以将它看作是计算机系统的大脑。它被计算机指令所控制（控制器的功能）、根据指令进行运算（运算器的功能）。

而程序员所编制的程序就会被 CPU 去执行。那 CPU 究竟会识别什么样的程序呢？是这样的程序：

代码 1: 机器代码

1	0110	1001	0110	1010	0101	1001	0101	1010
2	1101	1101	1111	0001	0100	1011	0101	0111
3	1001	0110	1101	0100	1001	0011	1111	1101
4	1001	1110	1000	0111	0001	1011	0100	0000
5	0001	0000	0011	0001	0110	1001	1110	0110



上面的代码 1 就是计算机能识别的一种语言。对于人来说它是非常的晦涩难懂，以至于我都不知道上面的代码究竟是什么含义。但是对于计算机来说，处理这种纯数字的代码是计算机非常擅长的事情。我们将其称为“机器语言”

我们再来探索一下上面的代码。每一行是由 32 个 0 或者 1 的字符组成的。CPU 可以轻松识别这些字符的含义。假设识别上面这种语言的计算机每一条指令长度是 32 个 0 或 1 字符组成，我们称这台计算机的字长为 32 位。

我们再假设，这台计算机的每条指令分为两个部分，前半部分的 16 位是“操作码”，CPU 在看到这 16 位的内容时就能识别出来自己所需要进行的工作。

然后指令后半部分的 16 位称为“操作数”。这部分表明该条指令所进行的操作的目标是哪里，或者携带一些关键的信息以协助该操作。

在厂商设计并且生产号一台 CPU 之后，会随着它附带一份上百或者上百页的“技术手册”。这本手册上写明了该型号 CPU 的各种参数、工作方式。最重要的是，它告诉了读者“应该用怎样一种语言来命令 CPU”。

假设你现在是工地的包工头，你在指挥来自世界各地的工人（而且他们都足够笨，笨到你需要非常详细地教他们该如何办事）。你让每个工人都给你拿来一块砖。对中国工人应该说：“从地上捡起一块砖，走到我面前，然后放下”。对美国工人应该说：“Pick up a piece of brick, come up to me, and then put down”。可见对于相同的工作，不通的人需要不通的理解方式。而且当工人太笨的时候，你的“指导”工作大量增加。

机器语言也是类似。不同厂商甚至仅仅是不通型号的 CPU 能接受的指令格式都不太相同。也就是说，上面的机器语言在这台机器上能运行，但是换了另一台机器就不一定能用了。而且如果只是为了让工人去搬一块砖还要费那么多口舌（指令），其实不是非常麻烦！要是能发明一种简单通用的“世界语”，命令只需要用世界语下达一次，无论是亚洲人还是非洲人，都能听懂并且去工作。

所以，你召集了手下一大群工人开会。你规定：“当我举起红色旗子时，你们就立刻从地上拿起一块砖。当我举起黄色旗子时，就都向我走过来。当我举起绿色旗子时，就都把砖放在地上”。

这下方便了，进行同样的事情，你只需要依次举起红色、黄色、绿色旗子就够了，不用多费口舌。而你所付出的开销就是将原来冗长的命令换成一个旗子。作为工人，他们所付出的开销就是当看到某种颜色的旗子，就把它转换成对应的指令。就是这么简单！

计算机厂商发现，如果用上面这种“替换”的方式，很容易做出简单、可读性强、并且有可能在多种不同机器上通用的语言，于是他们就发明了一种语言，叫做“汇编语言”。

汇编语言其实就是对机器语言进行了简单的替换。用几个简短的字母替代了原来 16 位的操作码。用简短的标号替代了原来 16 位的操作数。

#### 代码 2: 汇编语言

1	0110 1001 0110 1010 0101 1001 0101 1010	LDI #100
2	1101 1101 1111 0001 0100 1011 0101 0111	ADD R1,R2,R3

见上面的代码 2，LDI 和 ADD 是操作码，空格后面的 #100 和 R1,R2,R3 是操作数。我们把（0100 1001 0110 1010）这么长而且难以记忆的操作码变成了像 LDI 这么简洁的方式表达。

计算机还是无法识别像汇编语言这样的语言，所以我们还是需要一种叫做“汇编工具”的程序，将汇编语言转换成真正的机器语言才能被计算机所识别。那也就是说，对于同一条汇编指令，我们可以在“转换”的时候，根据程序将会运行的目标平台，转换成不同的机器语言。只要我们做的事情一样，即便机器语言不一样，通过使用不同的转换规则，我们就能够达到同样的效果！汇编语言就这么出现并且持续发展了几十年。

虽然汇编语言看起来那么的美好，但是它毕竟离我们人类的认知方式有些遥远。我们在计算数学题的时候可以说：“将上述过程重复 100 次”（循环结构）。或者有分段函数：“如果  $x < 0$  则  $y = -1$ ；如果  $x > 0$  则  $y = 1$ ”（选择结构）。像这样的表述，我们利用汇编语言很难“优雅”地实现。这时候如果能有一种语言能够把上面这几种结构轻松表达出来就好了！

同时，汇编语言通过统一的操作符带来了一定的通用性（可移植性）。但是像操作数是对寄存器进行操作，而不同的计算机系统的寄存器命名方式各有不同，所以它的通用性还是不是很强。这时候如果有一种语言，能够完全忽略寄存器，仅仅像数学运算那样对“变量”进行操作就好了！

于是，高级语言出现了。

高级语言分为很多种，我们在这里只论述一种叫做“命令式语言”的高级语言。这种语言的语义基础是要模拟“数据存储/数据操作”的图灵机模型。十分符合现代计算机的体系结构。

如果说冰冷的机器与人类之间有一道鸿沟，汇编语言、高级语言的出现，都在不断地缩小这一种鸿沟。

高级语言的好处非常多：易学易懂、能够将大量的精力放在算法的实现而非计算机系统底层的研究中。而且由于它的抽象程度高更高，所以可移植性就会更好。

## 0.2 内存、程序与数据

现在我们来通过程序解释一下冯诺依曼结构计算机的工作原理。上面我们曾经提到过，计算机是被程序控制的。那么程序的执行流程到底是什么样呢？

首先，我们要理解程序的物理结构。一般来说，不论我们用多么高级的语言编写程序，最后都要被转化成当前计算机所能识别的机器语言。这些机器语言按照严格的先后顺序存储在计算机的外存（如硬盘）上。当需要执行的时候，则将其调入主存（内存）。然后由 CPU 控制，按照严格的顺序逐条执行指令。

那么主存（内存）到底是怎样的一种结构？

正如之前我们所讲述的图灵机上的纸带。我们可以将内存想象成类似的形式。在这里我喜欢用“抽屉”作为举例。

假设你有一个非常非常高的柜子，上面从上到下依次排满了抽屉。为了方便定位，你在每个抽屉都编上了号。例如，最上面的抽屉是 0 号，往下一层的抽屉是 1 号，以此类推。既然是抽屉，那在其中肯定能放一定量的东西。由于每个抽屉尺寸规格是完全一致的，所以抽屉中能存放的东西也是一致的。这种柜子其实就是内存最简单的抽象。

总结一下，内存的特性就是这个柜子的特性：有编号、编号是顺序排列的（由 0、1、2 递增）、在每个编号中有一块空间。好了，这就是内存！

当然关于内存还有一些你必须知道的细节：内存的编号从 0 开始到整

个内存的结束称为寻址空间。一般地，每个编号下对应的存储空间为 8 位，我们称其为一字节。也就是说：每一个地址对应一个能存储 1 字节数据的空间

一般来说，在有操作系统的计算机上，程序的执行是受到操作系统控制的，程序执行时的变量等数据也是由操作系统所管理的。对于程序员，我们必须要了解自己程序在操作系统的管理下的运行机制，才能让我们编写出更好的程序。

与“程序”遥相呼应的就是“数据”。数据可以指在程序运算过程中临时存储在内存上的内容。不严格地说，我们可以将这种临时的数据称为变量。如何理解“变量”？变量就如同在数学中的  $x, y, z$ 。在计算机中，变量是一种能够被赋值，能够被取值的元素。它对应着内存上的某个编号，在该编号下以一定的长度存储着数据。

变量是内存上某段数据的逻辑表现，是组成程序的重要元素。例如，假设我们在编号为 A 的杯子中加满牛奶，在编号为 B 的杯子中加满果汁。现在我们想要让两个杯子中的东西交换一下，即在 B 中装满牛奶，在 A 中装满果汁。这时候我们都会想到解决办法：先找到一个杯子 C，将 A 中的牛奶倒入 C 中。再将 B 中的果汁倒入 A 中，再将 C 中的牛奶倒回到 B 中。如此一来，我们就实现了 A 与 B 中内容的交换。在这里 A、B 和 C 都是变量，这个编号就是内存地址，杯子中的空间就是在这个地址下内存中的空间。而牛奶和果汁就是数据。这种利用杯子 C 来让 A 和 B 中的内容互换的流程就称之为“算法”。

对于高级语言来说，可以用如下的伪代码表示：

代码 3: 两个变量交换数据

```
1 a = 1; // 让 a 等于 1
2 b = 2; // 让 b 等于 2
3 c = a; // 让 c 等于 a
4 a = b; // 让 a 等于 b
5 b = c; // 让 b 等于 c
```

我们把上面这段代码 3 称为伪代码。是因为这段代码并不是某种计算机能识别的代码，而是人类用来表达想法的工具。我们将会在后来的课程

学习到真正的计算机语言。

## 0.3 算法与算法的复杂性

高级语言的出现，让硬件设计与软件设计工作逐渐分离。作为程序员，我们所关注的内容主要是控制机器运转的一系列规则。而机器怎么理解这些规则的描述是硬件工程师的工作，与我们没有太大的关系。所以我们工作的重点在于设计一套解决问题的逻辑流程。这套流程的核心就是算法。

算法其实就是完成某种事情的步骤，也就是将某件复杂的事情拆分成很多道简单的工序。例如把 A 杯与 B 杯中的内容互换：1, 从 A 到 C; 2, 从 B 到 A; 3, 从 C 到 B。这样的工序就是一种算法。而简单地算法相互组合又能实现出复杂的算法。例如我们有 5 个高矮各不相同的人排成一列，现在我想让长得最高的人排到第一个位置上，该如何去设计这个算法？

这里为了方便，我们用 `queue` 表示这个队列，用 `queue[i]` 表示第 *i* 个人。

例如 `queue[1]` 就是指排头，`queue[5]` 就是队尾。我们的算法可以这样简单的描述：

代码 4: 按身高排队

```
1 如果 queue[5] 的身高 > queue[4] 的身高则这两人交换位置
2 如果 queue[4] 的身高 > queue[3] 的身高则这两人交换位置
3 如果 queue[3] 的身高 > queue[2] 的身高则这两人交换位置
4 如果 queue[2] 的身高 > queue[1] 的身高则这两人交换位置
```

代码 4 中的每一个小步骤（元素交换），其实都可以看做是程序 3 的一段变体。

我们现在成功地实现了从一条队伍中找到最高的人并且排到最前面。但如果我想让整个队伍从大到小排列该如何做呢？其实如果你明白了上面两个例子的相关性，这道题一定不会难到你！

在执行第一遍代码 4 后，排在第一个的人是队伍里最高的，如果把上面的代码再执行一次，一定会让第二高的人排在第二次！也就是说，如果有 *n* 个人，我就把上面的代码执行 *n* 次，最终得到的结果肯定是已经按顺序排列完毕的队伍！很神奇吧？

这里我常常用中国传统道家文化来类比。道家文化认为万物都是由阴阳两种物质组成，所谓两仪生四象，就是  $2 \times 2 = 4$ ；四象生八卦就是  $4 \times 2 = 8$ 。如此一来，世间万物皆可由太极所产生，无穷无尽。

可见，很对复杂的算法其实都是通过一些简单的算法所组成的。而我们后续课程将会为大家从简单算法讲到复杂的算法！

我们再来讨论讨论算法效率的问题。还是上面道排序问题。

如果对于 5 个人的队伍而言，代码 4 中一共需要进行 4 次操作（我们将比较和交换看作是一个操作）。然后我们需要执行五次代码 4，这样就一共需要  $5 \times 4 = 20$  次操作。也就是说如果对于  $n$  个人的队伍而言，共需要进行  $n \times (n - 1)$  次操作。

这样的是不是有些多呢？有没有更好的方法？我们考虑一下对于上面算法的一个小小的改进。如果一条队伍已经执行了一次代码 4，则排在第一位的人肯定是最高的！则执行第二次代码 4 时，第四步就完全没必要了（因为 `queue[2]` 永远不可能大于 `queue[1]`）。那只执行前三步就足够了，依次类推，我们的计算量是  $4 + 3 + 2 + 1 = 10$  次操作。对于  $n$  个人的队伍而言，共需要进行  $(n - 1) \times n / 2$  次运算。可见这个修改后的算法计算步骤还是会比原来的算法要快一些。

解决同一个问题有多种方法，那么我们该如何衡量某种方法的优劣呢？那我们就要看看程序运行所需要的开销了。显然，程序运行的开销主要是“对存储器使用的开销”，包括内存使用量，外存使用量等等。我们称这种开销为“空间复杂度”。空间复杂度越高代表它所占用的空间越大。

还有一个开销是时间上的开销，计算机在进行一般操作的时候都需要 CPU 花费一些计算时间，而这种计算时间往往反映在操作“次数”上。操作的次数往往跟运算的规模有很大的关系，当数据规模是  $n$ （类比队列中有  $n$  个人）时，时间复杂度就是  $n$  的函数（即时间复杂度是  $T(n)$ ）。

例如上面的排序，原始方法  $T(5) = 20$ ，改进后的方法  $T(5) = 10$ ，当然这个数值越小，就说明它的时间复杂度越小，即它所需要的时间也就越少。

但我们是不是就能说改进方法比原始方法快两倍？（因为  $20/10 = 2$ ）。当我们的运算量  $n$  非常大的时候。会发生什么？我们进行一点儿数学上的推导：

表 1: 几种常见的复杂度

$T(n)$	别称
$T(1) = O(1)$	常数复杂度
$T(n) = O(\log_2 n)$	对数复杂度
$T(n) = O(n)$	线性复杂度
$T(n) = O(n \log_2 n)$	$n \log_2 n$ 复杂度
$T(n) = O(n^2)$	平方阶复杂度
$T(n) = O(n^3)$	立方阶复杂度
$T(n) = O(n!)$	阶乘阶复杂度
$T(n) = O(2^n)$	指数阶复杂度
$T(n) = O(n^n)$	写出这种代码就去死吧复杂度

对于原始方法:

$$f_1(n) = \lim_{n \rightarrow \infty} (n \times (n - 1)) = \lim_{n \rightarrow \infty} (n^2 - n) = n^2 \quad (1)$$

设:

$$T_1(n) = O_1(f_1(n)) = O_1(n^2) \quad (2)$$

而对于改进方法:

$$f_2(n) = \lim_{n \rightarrow \infty} \left( \frac{n \times (n - 1)}{2} \right) = \lim_{n \rightarrow \infty} \left( \frac{1}{2}n^2 - \frac{1}{2}n \right) = n^2 \quad (3)$$

设:

$$T_2(n) = O_2(f_2(n)) = O_2(n^2) \quad (4)$$

可见, 当  $n \rightarrow \infty$  时,  $T_1(n) = T_2(n)$ 。也就是说即便改进过的算法有一定的优势, 但是当数据量足够大的时候, 这种优势体现得就不再明显了。一般情况下我们用  $O()$  的方式来表示算法的时间复杂度。

实际上,  $T(n)$  有很多种常见的情况。

上面的复杂度在  $n$  很大的情况下, 可以从小到大排列为:  $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(n!) < O(n^n)$

通常来说, 当我们的程序的复杂度大于等于  $O(n^2)$  的时候就要考虑考虑能否通过改善算法来减小复杂度了。而我们上面曾经讨论过的排序算法,

经过理论证明，它所能达到的最小时间复杂度是  $O(\log_2 n)$ 。而排序算法作为算法的基础，我们也将会在后续的课程中为大家介绍。

## 0.4 数据结构

在说完算法之后，还有一个不得不提到的东西，那就是“数据结构”。

如果说算法的目的是通过操纵一系列数据来完成特定的功能。那么什么是数据？我们可以将“数据”定义为“计算机所能够识别、存储和处理的所有符号的集合”。

“1”可以是数据，“IMUDGES”也可以是数据。甚至一段声音、一段视频都可以是数据。而计算机所处理的数据，必定需要按照一种方式来组织、存储，这就是数据结构。不同的数据结构可能会对算法的空间、时间复杂度有着明显的影响！

数据结构又可以分为两个部分，一个是“数据的逻辑结构”，另一个是“数据的存储结构”。

所谓数据的逻辑结构，就是指数据跟数据“看起来”的关系。例如我们让一组数据从小到大依次排列，所以我们可以断言：后面的数据元素“看起来”应该大于等于前面的数据元素。而这种递增的关系仅仅是“看起来”，并不与数据在内存中的排布方式有着直接的关系。

所谓数据的存储结构，就是指数据在计算机存储器中的排布方式。常见方式有：顺序方式——将逻辑上相邻的数据元素也存储在相邻的存储单元中。例如一串连续储存在内存中的数组；链接方式——多个数据元素被作为一个独立的部分遍布存储器的各个地方，每个部分都应该有能够表示相邻关系的指针。例如链表；索引方式——按照数据元素序号建立索引表、索引表中第  $i$  项的值为第  $i$  个数据元素的存储地址。例如数据库中的索引表；散列方式——数据元素的地址与标识该数据元素的关键字之间有一定的对应关系。例如哈希表；

而对于数据的运算无非就是“插入”、“删除”、“查找”、“排序”、“更新”这几种。Pascal 之父——Nicklaus Wirth 曾经提出过一个公式：

$$\text{程序} = \text{算法} + \text{数据结构}$$



这个公式对于计算机科学的影响程度可以类似于爱因斯坦的质能方程—— $E = MC^2$  对物理学的贡献——这个公式揭示了程序设计的本质。而在后续章节中，我们将会始终围绕着算法和数据结构，让大家逐渐形成这种针对特定问题求解的计算思维。

## 0.5 二进制

在上面的例子中，我们似乎也看到了，计算机所喜欢的机器语言是由 0 和 1 组成的。除此之外没有其他数字。我们不严谨地称这种只有 0 和 1 参加的运算为二进制运算。我们常用的十进制运算是“逢十进一”。而二进制则是“逢二进一”。也就是说，当两个加数的结果等于 2 的时候，就应该向前进位。十进制可以进行的运算在二进制中也基本存在。例如加减乘除，都是可以在二进制的情况下进行。即便你在程序中使用的是十进制运算，但是编译器最终会将这些十进制数编码成二进制数再参与运算。在十进制中，我们有“乘 10”和“除 10”，可以当做是在末尾“补零”和“删除”。我们统计将这种运算成为位移运算。如果说我们做 R 进制运算，则左移一位就是将原数字乘 R。右移则是将原数字除 R 并删掉余数。

## 0.6 习题

**习题 1** 解释“冯诺依曼”计算机系统由哪五大部件组成？分别阐述其功能。

**习题 2** “存储器”包括哪些部分？内存和硬盘，哪个属于存储器？哪个不属于存储器？

**习题 3** 举例说明输入设备和输出设备。

**习题 4** 举一个例子，来描述内存。（不要再用柜子举例了）

**习题 5** 用语言直接描述算法：在 5 个数种选择第二大的数。（把自己想象成一台简单的图灵机）、列出步骤

**习题 6** 查阅资料、了解二进制的加法与减法、了解 2 进制与 10 进制相互转换的方法。并按要求转换：

$$1011011010111101B \rightarrow \_\_D$$

249D → \_\_\_\_\_B

(B 表示二进制, D 表示十进制)

# 第一章 码中一窥 C++

凡是语言，都一定有着自己的规则。而不同于英语、汉语这种自然语言，作为计算机所能识别的语言，它是需要在严格约束下避免二义性的，所以说我们将学到的 C++ 语言，是一门语法规则非常严格，但又“灵活多变”的语言。

所谓非常严格，是指 C++ 中不允许出现歧义，这种没有歧义的文法是经过严格的文法规则的限制、和一些约定来约束的。所以，如果在你的代码中出现了自己都会觉得有歧义的地方，往往就是程序中容易出错的地方。

而所谓灵活多变，就是指在掌握 C++ 严格的语法之后，可以写出无穷无尽充满创意的程序。从黑框 (控制台程序)，到带有图形界面 (GUI) 的程序，再到游戏，都可以利用 C++ 的语法加上你脑中的算法搞定！而所有的这些灵活多变的内容，都是在严格的文法约束下实现的。所以说 C++ 语言，是一门语法规则非常严格，但又灵活多变的语言！

C++ 是一门高级语言，所谓高级语言正如你们在前面章节中看到的，是不同于机器语言、汇编语言；能更加贴近人类思维的一门语言。C++ 是在 C 语言的基础上，增加了例如面向对象编程等诸多特性的程序设计语言。在程序设计竞赛中被广泛采用。

非常不严格地讲：我们常常所说的“C++”，包含了这门语言和一系列的工具。语言顾名思义就不多阐述。而工具实际上包含了很多内容：

我们将 C++ 语言所编写的代码叫做“源代码”。而源代码是不能直接被计算机执行的（回想我们在上一章中讲述的内容）。而是应该利用编译程序，将源代码编译成计算机能够识别的机器代码然后再执行。而执行编译工作的工具就叫做“编译器”。

除此之外，仅仅有 C++ 语言，我们连基本的功能（例如屏幕输出）都

实现不了。即便是高级语言，想要与硬件设备交互，一般都需要直接编写一定的机器语言或者汇编语言，所以才能让我们的程序操作硬件：屏幕打印、屏幕输入、文件操作等等。而我们想要通过一己之力实现这些最底层的操作是很困难的。好在 C++ 提供了一组可以被调用的工具。这些工具其实就是别人写好的代码，通过 C++ 语法能够接受的形式所调用，最终辅助我们完成底层调用的功能，让我们的精力能够集中在更重要的算法设计上而不是千篇一律的与底层的通讯上。我们称这种“别人的代码”叫做“库”。就如同仓库一样，我们可以调用仓库中的内容来实现我们的意图。

那么我们的程序是怎么与别人的程序产生关联的呢？这有很多种情况，我只讲述最常见的一种：静态链接。我们的程序在调用库的时候，编译器可以仅仅在相应的位置上做个标记，表明这里的程序是调用了库而不是我自己的。这么做的好处就是可以充分地提高编译效率。因为库中的代码往往经过严格的编写、测试，基本能保证没有问题，所以我们只需要保存该库编译完毕的状态，没有必要在编译自己的程序时还要将别人的库再编译一遍。我们只需要编译自己的程序就可以了。所以我们的程序与库之间是并列关系，而不是包含关系。但是想要让自己的程序能够变成可执行程序，仅仅是这种“做标记”的方法是不可接受的。因为你所需要的库文件在你的电脑上会有，但并不意味着别人的电脑上也会有。再加上可执行程序的复杂原理，我们必须将这些静态库与我们的程序绑定在一起。而将编译完的“目标代码 (Object Code)”与库文件链接形成“可执行文件”的工具叫做“链接器”。

也就是说，C++ 程序设计流程是：

1. 编写代码
2. 编译成目标代码
3. 链接成可执行文件

一般来说，想要实现上面的流程，需要利用操作系统提供的黑框 (Linux 下的终端、Windows 下的控制台)，再加上一大堆非常复杂的命令才能实现。编译命令甚至有可能比源代码还要复杂。如果需要对自己的代码进行调试，那可是更加麻烦。所以聪明人们就为懒人发明了又一件工具：集成开发环境 (Integrated Development Environment, IDE)。所谓开发环境，可以将其

看作是写代码的一个工具。我们可以在这个开发环境中编写代码、代码着色、自动排查错误等。所谓“集成”就是指将“编译器”、“连接器”都集中在 IDE 中。程序员可以简简单单地点击按钮，就可以实现整套编译链接运行调试等功能。

在 ACM/ICPC 的比赛中，通常使用的编译器是 GCC(GNU Compiler Collection)。GCC 是一个支持很多种语言的开源的编译器的集合。它可以安装在 Windows（不推荐）和 Linux 等多种操作系统上。而在比赛中普遍使用的是安装了 Linux 操作系统和 GCC 编译器的开发环境，并且配置了某些 IDE 来让学生进行开发。

关于 IDE：对于 Windows 来说，我们推荐使用 Dev-C++。对于 Linux，我们推荐使用 Code-Blocks。

本章的行文基于“CART”步骤，即 (C)ode, (A)nalyze, (R)esult, (T)ry。四步。

Code 是指展示代码，Analyze 是针对代码分析其中的语法元素，Result 是根据程序的运行结果来分析程序的逻辑，Try 则是举一反三，完成与该代码相关的另一段代码的编写。

## 1.1 Helloworld++

其实吧，作为一名编程初学者，第一个编写的程序应该就是传说中的“Helloworld”。这个程序意图就是在屏幕上简简单单地打印出一个“Helloworld”的文字。目的是为了表示当前的开发环境没啥问题，可以继续工作了。

但是我觉得一个 Helloworld 太简单了，各种书籍都把这个例子用烂了，那我在这里就用一个稍微复杂的例子来实现这个“Helloworld”吧！

### 1.1.1 Code

代码 1.1: Helloworld++

```
1 #include <iostream> ❶  
2 #include <cstring>
```

```
3 using namespace std; ❷
4
5 int main(){ ❸
6
7     /* 字符串最大长度 */ ❹
8     const int trunk_size = 100; ❺
9
10    /* 保存名字的字符串指针 */
11    char *name = (char*)malloc(sizeof(char)*trunk_size);
12
13    /* 保存性别的字符串指针 */
14    char *sex_str;
15    sex_str = (char*)malloc(sizeof(char)*trunk_size);
16
17    cout<<" Who are you?"<<endl;
18    cout<<">>";
19    cin>>name;
20    cout<<" Hello, "<<name<<"!"<<endl;
21    cout<<" Are you a boy or girl?"<<endl;
22    cout<<">>";
23    cin>>sex_str;
24    if(!strcmp("boy",sex_str)❽){ //判断这里是"boy"还是"girl"
25        cout<<"OH! NO! We have a tons of boys!"<<endl❾;
26    }else{
27        cout<<"Hey! Welcome to IMUDGES!!!!!!!"<<endl;
28    }
29    return 0;❿
30 }
```

### 1.1.2 Analyze

代码 1.1 是加强版的 Helloworld，所以我管它叫做 Helloworld++。代码首先通过提示依次输入你的姓名和性别，再根据不同的性别输出不同的“问候语”。对它的部分解读如下：

❶ 导入库文件，当你想要使用某些系统库提供的功能时，必

须 include 相应的库文件

- ❷ 使用命名空间，这个涉及到 C++ 对模块的组织规划，已超出本书的范围。但一般情况下这句话是必不可少的！
- ❸ 主函数
- ❹ 代码注释
- ❺ 变量定义
- ❻ 标准输出流，在控制台程序下控制输出
- ❼ 标准输入流，可以将输入到计算机中的内容读入程序
- ❽ 调用字符串比较函数
- ❾ endl 表示换行。cout、cin、endl 等功能必须在 using namespace std; 后才能使用
- ❿ 主函数的返回值

我们再来逐行精读这段代码。

首先，代码 1.1 的前两行中使用了“#include<xxxx>”。这部分内容叫做“预编译指令<sup>1</sup>”。所谓预编译，就是指在编译之前要“告诉”编译器的“话”。编译器收到这些指令之后就会按照指令上的内容提前进行一些预处理。

在 C++ 中常见的预处理指令主要有“#include”、“#define”等。当然还有一些诸如“#ifndef”等在本书中不是很常用的预处理指令。

“#include xxx”用来表明该段程序编译前应该包含哪些其他文件。由于操作系统会将一些常用的操作封装成函数库，并写在某些自带的文件中，所以当我们使用这些函数库的时候，一般都需要通过该命令将其包含。命令中的 xxx 则是用“<>”或者双引号括起来的文件名。用“<xxx>”则表示这个文件在系统的搜索路径下保存，一般是系统库。而用双引号括起来的则表示这个文件跟当前程序代码文件存放在同一个文件夹。

而“#define A B”则是表示强行将 B 替换成 A。也就是说在以后的代码中凡是出现过“B”的地方都将会无条件地被替换成“A”。在 C 语言<sup>2</sup>时

---

<sup>1</sup>也叫做预处理指令。

<sup>2</sup>C++ 的前身，C++ 就是对 C 语言扩展而成的语言

代，常常用这种方法来避免一些硬编码<sup>3</sup>。例如有一个运算式频繁使用了 `PI=3.14` 这个数值。但是突然想要让 `PI=3.1415`。这时候需要把所有的 `3.14` 替换成 `3.1415`。但如果很早之前我们的程序就是用 `PI` 来表示这个值，在此时也就只需要将 `#define PI 3.14` 改成 `#define PI 3.1415` 即可。

由于 `#define` 是在“编译前”执行，所以它带来的开销不会在程序运行时体现出来，而且还能进行很多技巧性的操作。但是由于这种“技巧性”的操作稍有不慎就可能产生很麻烦的后果，所以不建议初学者滥用这种方法。而它所提供的功能可以通过类似但更加安全的方法实现。

代码 1.1 的第三行用来“使用命名空间”。所谓命名空间，是一种 C++ 对多文件多函数的组织工具。我们常常把功能相近的函数库放在一个模块中，为这个模块可以取一个好记的名字。当以后使用这些函数库时，就可以使用 `using namespace xxx;` 这样的语句来使用。

在代码下面的诸如 `“cin”`、`“cout”`、`“endl”` 等内容，以及以后大家将会接触到的一些函数库都需要 `“std”` 命名空间，大家记住在每次编写代码时都写上这一句就好了。有关命名空间的详细主题已经超过本书“速成”的要旨，就不多介绍了。

程序的第五行是定义了主函数。所谓函数，暂时可以理解为对一组操作的包装<sup>4</sup>。而主函数则是整个程序的入口。可执行文件被打开后，操作系统将会从主函数开始依次执行。在主函数中又调用了各种各样其他的程序，以此来完成整个运行逻辑。

之所以称之为主函数是因为它的结构。`int main()` 表明这是一个返回 `int` 类型<sup>5</sup>的函数，而函数名 `“main”` 则表明了这个函数是一个主函数。后面紧跟着一个大括号。主函数内部的东西都写在了这个大括号中。

一般来说，大括号表示一段语句，而其中的语句隶属于这个大括号，所以在在大括号中的代码应该比外面的代码多缩进一截。这么做对编译器没有任何影响，只是为了让代码层次感明确、易读。所以希望大家都遵守这个惯例！

---

<sup>3</sup>即“直接将常数值分布在程序的各处”，这种硬编码导致的程序冗余性会让代码修改起来非常困难。

<sup>4</sup>术语叫做“封装”

<sup>5</sup>什么叫做“返回 `int` 类型”，我们稍后再谈



在大括号中的内容就是系统会执行的代码。第七行的内容称为注释。注释就是一种会自动被编译器所屏蔽的“非代码”。注释能够辅助程序员解释代码功能，增加代码可读性。同时由于“被注释后会被编译器忽略”的特性，使得注释可以用来“临时屏蔽”一些代码。

在 C++ 中添加注释有两种方法。一种是行注释：“//”——两条正斜杠后面的内容都是注释。还有一种是块注释：“/\*”到“\*/”中间的所有内容，无论经过多少行，都会被看做是注释。正如同代码的第 7、10、13 行。

在代码的第 8 行，声明了一个常量。此处的常量与数学意义上的常量非常类似。常量的定义通常由两部分。

一部分是定义部分，形如“`const int trunk_size`”。其中“`const`”是定义常量的关键字<sup>6</sup>。“`int`”表示这个常量的类型是“整型”。关于数据类型将会在后面介绍。“`trunk_size`”就是变量名咯，是这个常量的名字。

另一部分是赋值<sup>7</sup>部分，形如“`trunk_size=100`”。这部分内容就是指令常量“`trunk_size`”等于常量值“100”。（跟我们学过的数学完全一样是吧！）

既然名叫常量，则就说明这个量一经定义不许再在程序中改变。所以常量的定义部分和赋值部分是合在一起的，形如第 8 行代码。此之于“变量”来说是略有不同的，我们将稍后再说。

在第 11 行和 14、15 行有两条非常相似的语句。这就是传说中的变量定义。其实变量定义跟常量定义非常相似，反倒是常量定义麻烦了一些——变量定义不需要 `const` 修饰。在 11 行，定义了变量，并且直接在等号右边赋值。第 14 行则是先定义变量，再在第 15 行赋值。

变量定义部分的“`char *`”表示了这是个“字符指针类型<sup>8</sup>”。“`name`”和“`sex_str`”是变量名。而等号后面的赋值部分，实际上是在进行“在堆中申请一段连续存储空间”的操作。上面这堆文字具体的意义我们后面会详细提到，目前你只需要知道这就是声明变量的一种方法就好了！

在第 17，18 行，有一种“`cout<<xxxx<<endl;`”的格式。其中 `cout` 是输出流对象。也就是说需要输出的时候就靠它。而在第 19 行的 `cin` 则是代表输

---

<sup>6</sup>一般与“保留字”等价。他们是一种特殊的单词，这些单词不能作为常量、变量名等用户自定义的内容。因为它们本身已经被 C++ 编译器赋予了神圣的意义。

<sup>7</sup>赋值就如同数学中的等式，将等号右边的值赋予等号左边的“名字”，从此这个“名字”将会与该值绑定

<sup>8</sup>指针，即“保存内存地址”的变量，该主题将会在后面谈到

入流对象。这两种对象对应着不同的符号进行控制。如果说这种大于号和小于号类似于箭头，则可以将“cout”和“cin”分别看作是屏幕和键盘。数据从箭头起点处流向目标。而且 cout 和 cin 必须要写在开头。于是就有了例子中的样式：第 17 行的“Who are you”作为数据，被传送到了 cout 这个“屏幕”。然后又将“endl”这个换行符传递给了 cout。在第 19 行则表示将键盘的内容输出到“name”这个变量中。

代码的第 17-23 行都是在进行输出和输入。而在 24 行到 28 行则是一种特殊的结构——选择结构。

所谓选择结构，就是指根据某个条件进行选择。如果这个条件成立则做事情 A，如果另一个条件成立则做事情 B，以此类推，直到什么都不成立的时候，则做另一个事情。

这五行代码是选择结构中的 if 语句。形式大约是：

```
if(< 事件 A>){  
    若 A 事件成立则执行此块  
}else if(< 事件 B>){  
    若 B 事件成立则执行此块  
}else{  
    若都不满足则执行此块  
}
```

值得注意的是在第 24 行的“strcmp(“boy”,sex\_str)”。这是一个系统库“<cstring>”中的函数“strcmp(strA,strB)”。这个函数的功能是比较字符串。若 strA 与 strB 的内容完全相等，则它的值等于 0。在这个函数调用前的感叹号是“取反”符号，功能就是让 1 变 0、0 变 1。所以这段代码的逻辑就是：如果 sex\_str 等于“boy”，则执行第 25 行。否则执行 27 行。

在第 29 行，“return 0”表示这个函数返回“0”。由于这个函数是主函数，则它的返回值将会被操作系统所接收<sup>9</sup>。

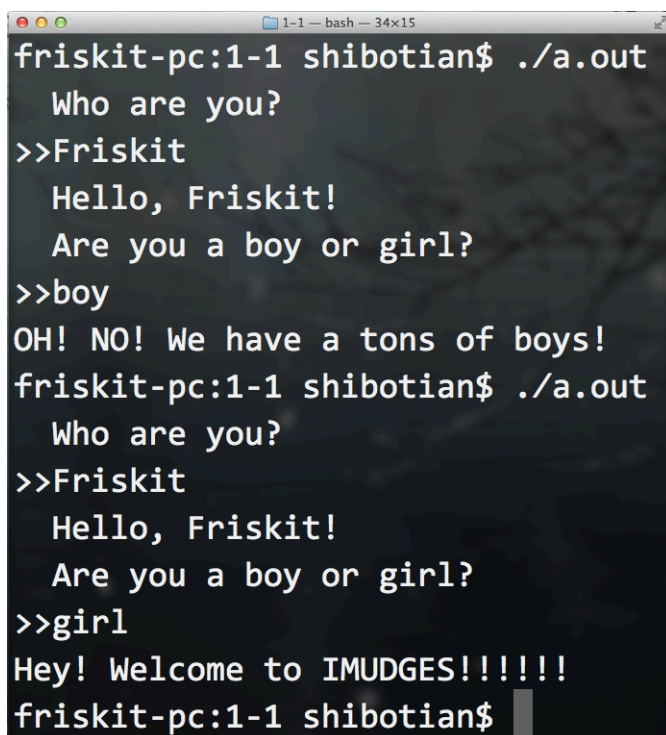
至此我们便大致分析了这个“Helloworld++”程序。接下来我们执行以下看看结果！

---

<sup>9</sup>有关函数和返回值的相关主题将会在后面介绍

### 1.1.3 Result

根据程序逻辑，这段代码执行时要先输入一个名字，然后输入 boy 或者 girl，屏幕则会根据这些不同的输入打印出不同的响应文字。



```
friskit-pc:1-1 shibotian$ ./a.out
Who are you?
>>Friskit
Hello, Friskit!
Are you a boy or girl?
>>boy
OH! NO! We have a tons of boys!
friskit-pc:1-1 shibotian$ ./a.out
Who are you?
>>Friskit
Hello, Friskit!
Are you a boy or girl?
>>girl
Hey! Welcome to IMUDGES!!!!!!
friskit-pc:1-1 shibotian$
```

图 1.1: Helloworld++

我们再来分析一下这个程序的运行逻辑。

**步骤 1»** 在程序的 1-2 行，通过预编译指令，包含了两个函数库文件。

**步骤 2»** 在程序的第 5 行找到程序的入口、主函数。

**步骤 3»** 在程序的第 8 15 行定义了一系列常量和变量。

**步骤 4»** 在程序 17 18 行输出提示信息，让用户输入名字。

**步骤 5»** 等待用户输入名字，并将输入内容存储至变量 “name”。

**步骤 6»** 再提示用户输入性别。

**步骤 7»** 等待用户输入性别，是“boy”或者“girl”。并且将输入内容存储至变量 `sex_str`。

**步骤 8»** 判断 `sex_str` 里的内容。

**步骤 9»** 如果内容是“boy”则执行第 25 行的输出。

**步骤 10»** 如果内容不是“boy”则执行第 27 行的输出。

**步骤 11»** 主函数返回，程序结束。

至此就是这个程序的运行流程。

我们再来小结一下上面的程序所遇到的内容！希望大家在看到这些名词之前能努力回忆一下它们的具体含义。然后再看后面的解释内容！

**预编译指令** 被编译器在编译前执行的命令。包括“`#include`”、“`#define`”等等。

**命名空间** 命名空间是 C++ 用来高效地组织各种代码的方法。通常我们使用的命名空间是“`std`”。用法是：`using namespace std;`

**主函数** 整个程序运行的入口。它的返回值将会被操作系统所接收，你可以通过返回值告诉操作系统这个程序结束执行时的状态。例如 0 表示正常，-1 表示错误。

**变量** 变量就如同数学中的概念一样，是一种可以变化的量。变量就是内存中固定大小的一块存储空间，其大小由其类型决定。变量使用前需要预先定义！

**常量** 常量是一种一经定义就不能改变的量，也正是因为这个特性，它必须在定义的时候立刻赋值。将一般不会改变的数存储成常量而不是变量，一来能防止不小心地修改这些值，另一方面能够通过系统的内存管理机制得到优化。

**标准输入输出** 即“`cout`”和“`cin`”。就是一种用来控制“输出到屏幕”和“从键盘读取”的工具。大于号或者小于号表示输出方向。将“`cout`”看作是屏幕、将“`cin`”看作是键盘，将会比较容易加深记忆。

**if 语句** 它就是一种选择结构的语句。就如同数学中的分段函数一样，根据不同的条件进行不同的操作。大家再回忆一下它的形式。

**语句** 语句就是 C++ 程序中的基本组成单位。变量定义可以使一个语句、赋值操作可以是语句、函数调用可以是语句，还有一些程序控制语句（例如 if 语句等）。总之，我们的程序是由一个个语句组成的。在语句之中可以嵌套其他语句。

**关于分号** 一般来说，除了预编译指令以外，我们需要在每一个语句完成之后加上分号。

在这一节中，大家了解到了一个完整的不算大也不算小的 C++ 程序的样子。我们今后所编写的代码也基本都长这个样子，区别就是主函数里头的东西可能更丰富、算法更高深等等。但是这个整体的框架是万变不离其宗的。所以希望大家能够根据上面的例子“改造”出自己的代码来。

### 1.1.4 Try

在了解完上面的例子，我们可以用类似的代码“创作”出一些自己的程序！

**Try 1** 制作一个程序，在屏幕上输出一行“Hello World!”。

**Try 2** 制作一个加法运算器，根据提示分别输入加数和被加数，并且输出两者相加的结果。

**Try 3** 制作一个程序：用户输入数字 1-7，输出汉语“星期 x”，（使用连续的 if-else if 语句）

## 1.2 使用变量与数据类型

我们在前面曾经讲过，现在的计算机都是一种“存储计算模型”。“存储”在程序中的体现就是“变量”。变量其实就是在程序运行时用来存储数据的容器。它的物理结构就是内存中的某一块固定大小的区域。尽管都是

表 1.1: 基本数据类型

数据类型	长度	能表示的数据范围
char	1	$-128 \sim 127$
bool	1	true or false
short	2	$-32768 \sim 32767$
unsigned short	2	$0 \sim 65535$
int	4	$-2147483648 \sim 2147483647$
unsigned int	4	$0 \sim 4294967295$
long	$8^{10}$	$-9223372036854775808 \sim 9223372036854775807$
unsigned long	8	$0 \sim 18446744073709551615$
float	4	$-3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	8	$-1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$

用“0”、“1”将内存区域填满，但不同类型的变量所占用内存单元的数量会有所不同，也会导致这种类型所表示的数据范围的不同。同时，一般的数据类型还有“有符号”和“无符号”的区别。这是由于计算机利用数据单元中的 1 位来表示正负，所以即便是能够表示数字的数量相同，但能表示的范围仍有所区别。

在 C++ 中有一些内置的基本数据类型，用户也可以定义自己的数据类型。不同的数据类型以不同的排布方式放置在内存中。数据相同的内存单元，用“不同类型的角度”来看都有不同的取值。常见的基本内置数据类型如下：

了解了上述的数据类型之后，让我们来看看这一节的代码吧。

### 1.2.1 Code

代码 1.2: 数据类型

```
1 #include <iostream>
2 #include <bitset>①
```

<sup>10</sup>在 32 位计算机上长度可能为 4 字节，功能和取值范围与 int 完全一致

```

3
4 using namespace std;
5
6 int main(){
7     //测试不同数据类型占用内存大小
8     cout<<"Size of char:\t"②<<sizeof(char)③<<endl;
9     cout<<"Size of short:\t"<<sizeof(short)<<endl;
10    cout<<"Size of int:\t"<<sizeof(int)<<endl;
11    cout<<"Size of long:\t"<<sizeof(long)<<endl;
12    cout<<"Size of float:\t"<<sizeof(float)<<endl;
13    cout<<"Size of double\t"<<sizeof(double)<<endl;
14
15    //测试数据范围
16    short short_number = 32767; //有符号short类型的最大值
17    cout<<"short_number = "<<short_number<<endl;
18    short_number = short_number + 1; //最大值+1会发生溢出11,
        变成最小值。
19    cout<<"short_number + 1 = "<<short_number<<endl;
20
21    unsigned short ushort_number = 65535;
22    cout<<"ushort_number = "<<ushort_number<<endl;
23    ushort_number = ushort_number + 1;
24    cout<<"ushort_number + 1 = "<<ushort_number<<endl;
25
26    //测试内存内容
27    int int_number = 123;
28    cout<<"Binary format of 'int_number' is: "<<bitset<sizeof
        (unsigned int)*8>(int_number)④<<endl;
29
30    float float_number = 123;
31    cout<<"Binary format of 'float_number' is: "<<bitset<
        sizeof(float)*8>((int &)float_number)<<endl;
32
33    //测试类型转换
34    int convert_int = 1.2;⑤
35    cout<<"'1.2' saved in an int variable is: "<<convert_int

```

<sup>11</sup>溢出就是指当运算数字过大, 超过上限或者下限, 变成了另一个方向的极值。例如最大值+1变成了最小值, 最小值-1变成了最大值

```
35         <<endl;
36
37     float float_a=1.2;
38     float float_b=1.3;
39     float float_c;
40
41     float_c = float_a + float_b;
42     cout<<"float_a + float_b = "<<float_c<<endl;
43
44     float_c = float_a + 1⑥;
45     cout<<"float_a + 1 = "<<float_c<<endl;
46
47     float_c = float_a + (int)⑦float_b;
48     cout<<"float_a + (int)float_b = "<<float_c<<endl;
49     return 0;
50 }
```

### 1.2.2 Analyze

首先我们看看上面代码的一部分注解：

- ① bitset 中包含了用来输出二进制的功能。在一些竞赛题中会出现类似“将十进制转换为二进制”的问题<sup>12</sup>，所以可以善用此工具。
- ② “\t”表示了一个制表符中的 tab 键，经常作对齐之用。（什么是制表符？）
- ③ sizeof 是 C 语言中的一个关键字，用来计算内存大小。括号中可以填入数据类型、结构体、甚至是变量或常量。返回结果是一个 unsigned long 类型的数值，表示括号中内容在内存中占用了多少个字节 (byte) 的空间。这个关键字在 C++ 中非常常用！
- ④ bitset 的用法是：bitset<二进制位长度（是位数不是字节数，用整型表示）>(要表示的数字)，例如 bitset<32>(12345) 表示将 12345 用二进制表示，不足在前面补零，凑够 32 位。

<sup>12</sup>例如将点分十进制的 IP 地址转换为二进制形式



- ⑤ 此处进行了隐式强制类型转换。就是说在没有告知的情况下，赋值的数字发生了精度的变化。由于 `int` 类型的先天不足，所以当将等号右边的一系列运算完成后的结果转换成为 `int` 类型时，会将小数部分删掉（绝非四舍五入）。
- ⑥ 此处进行了隐含类型转换。`int` 类型的 1 会被自动转换成 `float` 类型
- ⑦ 在变量前放置一个用括号括起来的数据类型，这种操作叫做显式强制类型转换。所谓显式强制类型转换，即由用户强制地、将某种类型的内容不顾精度的损失而将其转换成为特定类型。基本数据类型通常是可以转换的，但用户自定义的数据类型并不那么容易转换。由于这种转换是强制的，能够绕过编译器的一些限制，所以虽然它非常容易出错，但仍然非常的实用！

上面的程序进行了很多关于数据类型的试验。

在 8 至 13 行，分别测试不同的基本数据类型在内存中所占空间的大小。在第 16 至 24 行进行了一些数据的“边界”操作。

在第 28 行，我们尝试输出了 `int` 类型变量“`int_number`”在内存中的表示。而在第 31 行又输出了 `float` 类型变量“`float_number`”在内存中的表示情况。

在 33 行以后，程序又测试了有关类型转换的内容。

### 1.2.3 Result

我们再来看看这段程序运行后的输出：



的整体运算结果必须要转换成为 `int` 类型。所以系统会自动地将 “1.2” 中的小数部分抹去，以此来适应 `int` 类型的范围。所以当这行代码运行完成之后内存中会有一个值为 “1” 的 `int` 类型的变量。

在代码 41、44、47 行，我们进行了一些 “看似” 简单的运算操作，我们仔细来分析一下其中的详细步骤。在 41 行，程序进行了 “`float_c = float_a+float_b`” 的操作。此时系统进行了如下的操作：

**步骤 1** 根据符号优先级规则，判断复杂表达式的执行顺序。此处发现应该先进行 “+” 操作。

**步骤 2** 判断 “+” 号两边操作数类型，发现都是 `float` 类型，则可以进行运算，得到两者运算结果 2.5，该结果也是 `float` 类型。

**步骤 3** 判断 “=” 号两边操作数类型，发现都是 `float` 类型，可以直接赋值。进行赋值操作，此时 `float_c=2.5`

再看看第 44 行代码进行的操作：

**步骤 1** 根据符号优先级规则，判断表达式的执行顺序。此处发现应该进行 “+” 操作。

**步骤 2** 判断 “+” 号两边操作数类型：“+” 号左边是 `float` 类型，右边是 `int` 类型的字面常量。则需要先进行 “隐含类型转换” 再运算。

**步骤 3** 系统将 `int` 类型的 “1” 转换成为 `float` 类型，则该表达式变成了 `float_a + 1.0f`<sup>14</sup>。

**步骤 4** 系统运算得到 2.2、类型为 `float`。

**步骤 5** 判断 “=” 号两边的操作数类型，发现都是 `float` 类型，可以直接赋值。进行赋值操作，此时 `float_c=2.2`

还有第 47 行代码：

**步骤 1** 根据符号优先级规则，判断表达式的执行顺序。此处发现最先应进行对 `float_b` 的强制类型转换。即 `(int)float_b`。

---

<sup>14</sup>此处用 `1.0f` 表示是 `float` 类型的 1

**步骤 2** 将 `float_b` 强制转换成 `int` 类型，即讲 1.3 的小数部分抹去，变成一个 `int` 类型的量“1”。

**步骤 3** 根据符号优先级规则，发现此处应该执行“+”操作。

**步骤 4** 判断“+”号两边操作数类型：“+”号左边是 `float` 类型，右边是 `int` 类型的字面常量。则需要先进行“隐含强制类型转换”再运算。

**步骤 5** 系统将 `int` 类型的“1”转换成为 `float` 类型，则表达式变成了 `float_a+1.0f`

**步骤 6** 系统运算得到 2.2、类型为 `float`。

**步骤 7** 判断“=”号两边的操作数类型，发现都是 `float` 类型，则可以直接赋值。进行赋值操作，此时 `float_c=2.2`

过上面几个例子中包含了几种类型转换。在 47 行出现的强制类型转换，通常优先级比较高，但是当优先级关系不明确（就是没记住）的情况下，建议用括号将它们整体扩起来，如“`float_c = float_a + ((int)float_b)`”。此时会将原类型在不顾精度损失的情况下进行强制的类型转换。

而另一种“隐含类型转换”也经常在程序中出现。该转换是在用户“不知情”的情况下自动进行的转换。之所以能够自动进行，是因为“多数情况”下，这种转换不会损失精度，对运算结果不产生直接的影响。但是还有一些例外情况，就是在不同类型赋值时，还是有可能产生精度损失的。隐含类型转换的目的是为了能让不同类型的数据进行运算。其核心就是在尽量保证“不损失精度<sup>15</sup>”的情况下使得某种运算的操作数类型一致。

隐含类型转换常发生在这些情况下：

**涉及混合类型的表达式运算** 在这种运算中，系统务必要保证运算数类型趋于相同，并且尽量不损失精度。所以 C++ 中有一种“向上转型”的概念。“下方”的类型比较简单、表示范围小。“上方”的类型比较复杂、表示范围大。

---

<sup>15</sup>此处的不损失精度只是宏观来说。例如之所以类型转换会从 `int` 到 `float` 而不是从 `float` 到 `int`，是因为从 `float` 到 `int` 会导致小数部分直接丢失。而 `int` 到 `float` 类型的转换，可能会丢失精度（`int` 数量级太大），但精度损失导致的误差不会太大（很大的数据下有一个很小的误差）。

**涉及函数调用时参数传递** 关于函数与参数的问题，我们将会在以后提到。在此大家只需要知道当日后我们遇到了“函数参数”时，可能会出现隐含类型转换的情况。

隐含类型转换是一把双刃剑。即便是“不损失精度”，也可能因自动数据类型的变化导致不可预知的错误。尤其是非基本数据类型的隐含转换。这种操作甚至在一些语言（例如 Java）中是被当做编译错误的<sup>16</sup>。但同时它还能为程序员省去很多核对类型的繁琐工作。所以想要用好这个特性，还是需要大家有着比较多的积累才能游刃有余。

总结一下，前面出现了两种主要的类型转换：一是隐含类型转换。这种转换往往是系统自动进行的，系统也会尽量进行处理，尽量让精度不会损失。另一种称为强制类型转换。强制类型转换又分为显式和隐式。所谓显式强制类型转换就是指由用户主动发起的、不顾精度会不会有影响都会进行的。而隐式强制类型转换则是用户不自觉地、由系统自动进行的、有可能会让精度降低的转换。常发生在赋值操作、函数返回值传参等情景。

### 1.2.4 Try

上面说了那么多有关数据类型的主题，下面我们来试一试！

**Try 1** 假设有变量定义

```
“short a = 32768; cout<<a<<endl;”
```

猜想一下应该输出什么？再用程序验证你的猜想。

**Try 2** 为表 1.1 中的每一种数据类型都创建一个变量，并用 sizeof 关键字测试表格中哪项与你的结果不同。如果有不同，为什么会产生不同？请大家查阅资料解决这个问题。

**Try 3** 代码：

---

<sup>16</sup>在 Java 语言中，不同的非基本类型之间的运算会被当做是编译错误看待。必须要通过强制类型转换来实现该功能，可以将这种限制看做是 Java 编译器对于程序员操作的一种“提示”

“cout<<1/2;”

的输出会是多少？如何才能让这个结果变成正确的“0.5”？试试你能找到多少种方法。

## 1.3 字符类型

我们在上一节中讨论了几种基本数据类型。但是有两个类型一直没有提到——char 型与 bool 型，它们的用法也比较特殊。

char 型也叫做“字符型”，它用来表示一个字符。它在内存中占用 1 字节的空间，也就是说它实际上能表示  $2^8 = 256$  个不同的数字。很早很早以前，计算机科学家们将这 256 个数字编制成了一张如同词典一样的表，给一些常见的西欧字符编号。例如我们让 97 这个数字对应字符类型的‘a’。让 48 这个数字代表字符类型的‘0’<sup>17</sup>。这个如同词典一样的映射表被称为“ASCII”码表。最早的 ASCII 码表有 128 个数字组成，后来发展成为了占用 1 字节的“扩展 ASCII 码表”。

通常在我们使用 cout 等对象的时候，当系统判断出来接下来要处理的数据是 char 类型时，不会将它对应的整数值输出出来，而是将其对应的文字字形打印在屏幕上。当然，既然它也是内存中一种存储数据的结构，我们也可以使用“非 char”的方法输出它的值。例如将其转换成 int 类型输出，将会得到它对应的编号。我们可以如同 int 一样对 char 类型的数据进行处理甚至运算。下面的这些程序将解释这些现象。

### 1.3.1 Code

代码 1.3: 字符类型

```
1 #include <iostream>
2 using namespace std;
3
```

<sup>17</sup>字符类型的‘0’与数字 0 不同。字符类型通常只作为文字的表示，一般不参与运算。而数字则表示程序中用来进行运算的一个数学概念

```
4 int main(){
5     char char_A = 'a'❶;
6     cout<<"char_A: "<<char_A<<endl;
7     cout<<"int of char_A: "<<(int)char_A❷<<endl;
8
9     char_A = char_A + 1;❸;
10    cout<<"char_A + 1: "<<char_A<<endl;
11    cout<<"int of char_A + 1: "<<(int)char_A<<endl;
12
13    cout<<"char of int '48': "<<(char)48❹<<endl;
14    cout<<"char of int '7': "<<(char)7❺<<endl;
15    cout<<"int of '\\n': "<<(int)'\n'❻<<endl;
16    return 0;
17 }
```

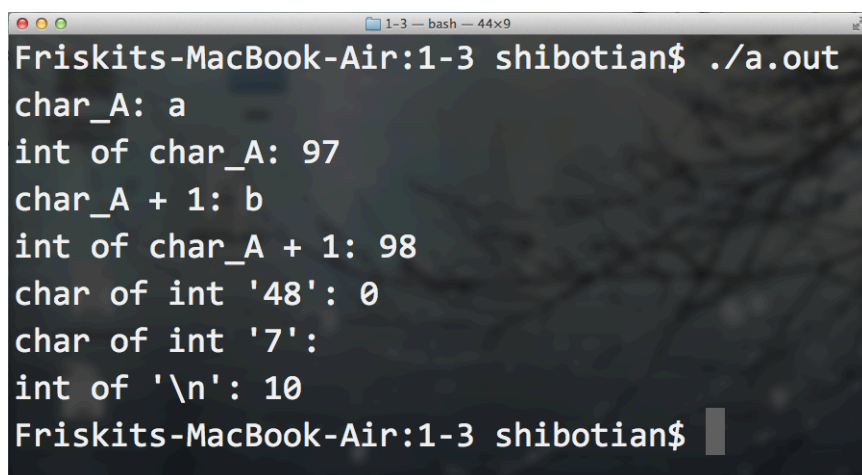
### 1.3.2 Analyze

在这段代码中，我们测试了一些关于字符串的功能。下面是对代码的一些解释：

- ❶ 就如同”int a = 1;”中的 1 是整型字面常量一样，这行代码中的’a’被称为字符型字面常量。因为它的值也就体现在它的字面意义上。
- ❷ 用”int”的方式去窥探 char 类型
- ❸ char 类型如同 int 一样，是可以进行四则运算的。
- ❹ 当然也可以用 char 的方式去窥探 int 类型！
- ❺ 有些 ascii 字符是能够被人看得到的文字，但有些字符并非是文字，而是代表了一些操作，例如 ascii 中的 7 表示了”响铃”
- ❻ 像’\n’这样的字符，其实也是 ascii 码中的一种，我们可以看看其具体的值是多少！

### 1.3.3 Result

然后我们看看代码的执行效果：



```
Friskits-MacBook-Air:1-3 shibotian$ ./a.out
char_A: a
int of char_A: 97
char_A + 1: b
int of char_A + 1: 98
char of int '48': 0
char of int '7':
int of '\n': 10
Friskits-MacBook-Air:1-3 shibotian$
```

图 1.3: 字符类型

在上面的输出中，`char_A` 的内容是字符 ‘a’。用强制类型转换的方法可以知道它的 “int” 值。所以说 ‘a’ 的 int 值是 97。然后我们如同对 int 进行操作一样，对 char 类型的 `char_A` 进行 +1 的操作，得到的内容即是数值  $97+1=98$ ，同时也是 ascii 码为 98 的字符 ‘b’。

后面我们以 “char” 的方式去输出了一个 int 类型的 “48”。这时候输出结果是 “0”。此处的 0 是打印在屏幕上的字符，而非能够直接计算的数字，因为它在内存中的数据是 48。紧接着我们以 “char” 的方式输出了 int 类型的 “7”。但是屏幕上并未显示任何内容。这是因为在 ASCII 不但有那些 “看得见” 的字符（如英文字母、数字），还有很多 “看不见” 但是有各种功能的字符。例如换行符 ‘\n’、制表符等一些控制字符。例如 ASCII 值为 7 的字符，实际上是指计算机的蜂鸣器鸣叫一声。大家可以再重新执行一下这段代码，听听是不是有这样的声音。后面又输出了作为换行符的 “\n”，它的 ASCII 值为 10。



### 1.3.4 Try

本节内容主要讲解有关字符类型的相关主题，大家可以用前面所学过的知识完成以下的小练习。

**Try 1** 输出一个“可见字符”的 ASCII 码表。输出字符和其对应的 ASCII 码。

**Try 2** 根据 Try 1 的结果，找到一种判断给字符分类的方法。分成：大写字母、小写字母、数字、标点符号等。编写一个程序，让用户输入一个字符，屏幕输出这个字符所属的分类。

## 1.4 布尔类型

布尔是英国的一名数学家，早在 18xx 年就发明了一系列处理二值运算的逻辑数学方法。所谓二值运算就是“非 1 即 0”。他成功建立了逻辑演算领域，从此，人类可以利用“公式推到”的方法来定量地解决逻辑问题。这一逻辑理论也被人称为布尔代数。

C++ 中用来表示“二值化”逻辑值的类型就叫做布尔类型。布尔类型只有两种取值：`true` 或 `false`，在内存中只占用 1 字节空间，但由于它只表示两种可能性，所以即便仅用 1 字节 (8 位) 空间存储，也浪费了八分之七 (7 位) 的空间。但在一些嵌入式设备中对内存管理有着极端的需求，所以可以通过一些位运算的技巧实现 1 字节存储 8 个二值化数据。

在这里我们了解一下逻辑运算。逻辑运算不同于算术运算，它的运算符主要是“与”、“或”、“非”。C++ 中相似的运算一共有两种。一种是普通的“与或非”，另一种则是“位与”、“位或”、“位非”。对于布尔类型而言，由于它在内存中只有最低位表示一个 1 或者 0，所以这两种逻辑运算效果是相同的，但对于那些一般的数值来说，位运算就有自己的神奇之处了！

对于普通的逻辑运算，C++ 中使用“`&&`”表示与，“`||`”表示或，“`!`”表示非。对于这三种运算有如下的真值表：

表 1.2: 与运算

<b>&amp;&amp;</b>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

表 1.3: 或运算

<b>  </b>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>

表 1.4: 非运算

	<b>!</b>
<b>true</b>	<b>false</b>
<b>false</b>	<b>true</b>

从表格中我们可以看出逻辑运算的规则<sup>18</sup>:

**与运算** 两个运算数全为 **true**，结果才为 **true**。只要任何一个为 **false**，则结果就是 **false**。

**或运算** 两个运算数只要有一个 **true**，结果就是 **true**。两个全为 **false** 是，加过才为 **false**。

**非运算** 操作数为 **true**，则结果为 **false**。操作数为 **false**，则结果为 **true**。

而另一种逻辑运算就是“位逻辑运算”。它们的参与运算的不再是操作数整体的布尔值，而是操作数的每一个二进制位。位与运算的运算符是“&”，位或运算的运算符是“|”，位非运算的运算符是“~”例如有 `short a = 385(00000001 10000001)` 和 `short b = 128(00000000 10000000)`。

**a&b** `385 & 130 = (00000001 10000001) & (00000000 10000010) = (00000000 10000000) = 128。`

**a|b** `385 | 130 = (00000001 10000001) | (00000000 10000010) = (00000001 10000011) = 387`

<sup>18</sup>与运算在 C++ 中具体的语法规则会在后面的代码中提到

**a** (00000001 10000001) = (11111110 01111110) = -386<sup>19</sup>

**b** (00000000 10000010) = (11111111 01111101) = -131

位逻辑运算就是按照每一个对应位进行的逻辑运算。前面我们曾经说到过用 1 个字节表示 8 个不同的逻辑值，就是用位逻辑运算实现的。通过位与 0 来让某一位置 0，通过位或 1 来让某一位置 1。再通过位与 1 来获得某一位上的值。

既然 `bool` 类型也是内存中占用 1 字节的一种数据类型，在内存中它有 256 种不同的取值，但对于布尔类型来说只要 `true` 和 `false` 就足够了。0 是 `false`，1 是 `true`，那么其他数值呢？

C++ 规定，对于 `bool` 类型来说，只有 0 表示 `false`，剩下所有数都表示 `true`。

### 1.4.1 Code

代码 1.4: 布尔类型

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      bool bool_a = true❶;
6      bool bool_b = 2❷;
7      bool bool_c = 0❸;
8
9      cout<<"bool_a:"<<bool_a<<endl;
10     cout<<"bool_b:"<<bool_b<<endl;
11     cout<<"bool_c:"<<bool_c<<endl;
12
13     cout<<"int value of bool_a:"<<(int)bool_a<<endl;
14     cout<<"int value of bool_b:"<<(int)bool_b<<endl;
15     cout<<"int value of bool_c:"<<(int)bool_c<<endl;
```

<sup>19</sup>这里的 -386 和后面的 -131 都是有符号类型

```
16
17     bool_c = bool_c + 1; ❷
18     cout<<"bool_c + 1 = "<<bool_c<<endl;
19
20     bool_c = bool_c - 1;
21     cout<<"bool_c - 1 = "<<bool_c<<endl;
22
23
24     if(bool_c❸){
25         cout<<"bool_c is true or something like"<<endl;
26     }else{
27         cout<<"bool_c is false or something like"<<endl;
28     }
29
30     short a = 385;
31     short b = 130;
32     cout<<"a & b = "<<(a&b)❹<<endl;
33     cout<<"a | b = "<<(a|b)<<endl;
34     cout<<"~a = "<<(~a)<<endl;
35     cout<<"~b = "<<(~b)<<endl;
36     return 0;
37 }
```

### 1.4.2 Analyze

- ❶ true 是 C++ 中的关键字，表示“真”。“假”为 false
- ❷ 布尔类型可以赋值数字
- ❸ 0 就代表 false
- ❹ 布尔类型也是可以进行一些运算的！但此处的运算时经过隐含类型转换之后进行的
- ❺ 布尔类型能够用来作为 if 的判断条件
- ❻ 用括号括起来是因为“<<”的优先级要比“&”高

这段代码主要是进行与布尔类型有关的操作。

从代码中我们了解到 `bool` 类型可以接受 “true”、“false” 甚至是数字等等作为值。在代码的第 6 行可以看到 `bool_b` 被赋值为 2，但在第 14 行输出的时候，发现 `bool_b` 的输出值实际上是 1。这是因为在赋值的时候，系统进行“隐含类型转换”——将数字 2 转换为 `bool` 类型的 `true`，而这个 `true` 实际上是数字 “1”，再将这个 1 赋值给 `bool_b`。

而在后面直接输出 `bool` 类型时，不难发现，输出的 `bool` 类型实际上就是非 1 即 0 的数值。这是因为 `cout` 会在输出布尔值时自动展开它的整数形式。所以输出的布尔类型的值与将其显示强制类型转换成 `int` 之后的值完全一致。

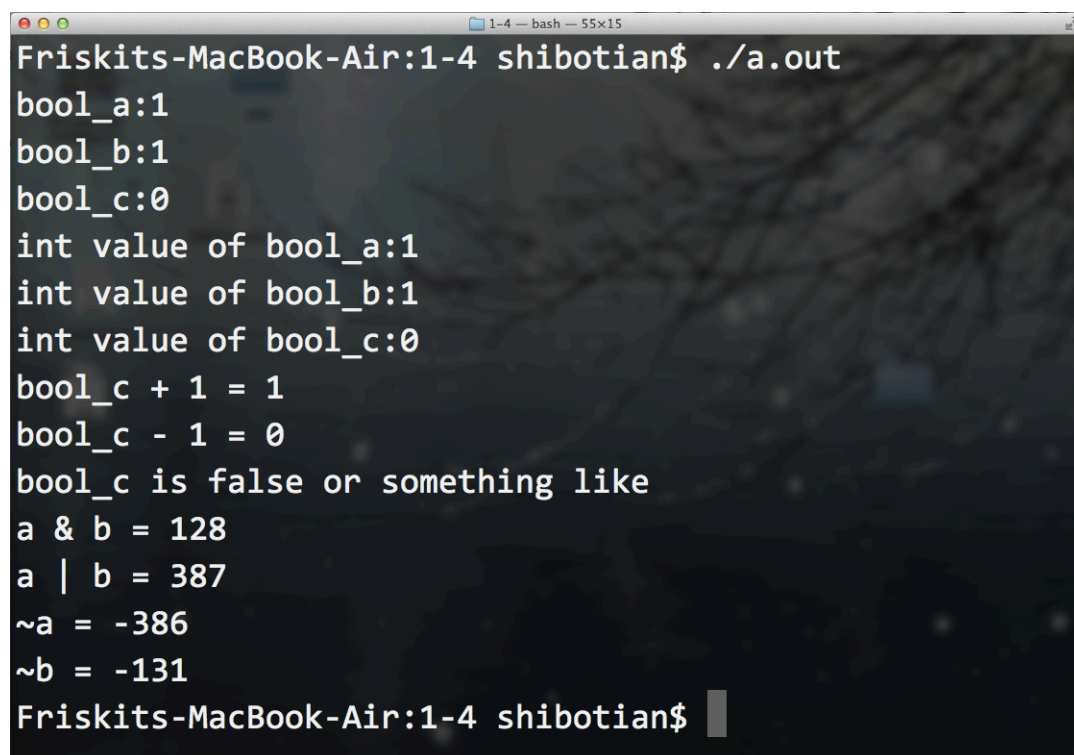
代码的第 17、20 行，我们测试了对 `bool` 类型变量的运算。也不难发现，这样的运算实际上就是对 `bool` 类型变量内部的整数值进行的运算。

在代码的 24 行至 28 行，程序使用 `if` 语句进行不同的操作，可见 `bool` 类型能够作为 `if` 中进行判断的条件！

在 30 行以后的代码印证了前面我们见到的位逻辑运算的功能。

### 1.4.3 Result

代码执行后效果如图：

A terminal window titled '1-4 -- bash -- 55x15' on a 'Friskits-MacBook-Air:1-4 shibotian\$' prompt. The output of './a.out' shows the values of boolean variables and the results of various operations. The output is as follows:

```
Friskits-MacBook-Air:1-4 shibotian$ ./a.out
bool_a:1
bool_b:1
bool_c:0
int value of bool_a:1
int value of bool_b:1
int value of bool_c:0
bool_c + 1 = 1
bool_c - 1 = 0
bool_c is false or something like
a & b = 128
a | b = 387
~a = -386
~b = -131
Friskits-MacBook-Air:1-4 shibotian$
```

图 1.4: 布尔类型

在这一节中我们了解到了有关“逻辑运算”的主题。包括了普通的“与或非”和“按位的与或非”。逻辑运算是后面很多内容的基础，在 C++ 中的地位非常重要。位运算代码的字里行间之中充满了小技巧，熟练地使用能够很方便地解决很多复杂的问题！

#### 1.4.4 Try

**Try 1** 用程序验证本节中出现的逻辑运算真值表

**Try 2** 实现用一个字节长度的变量存储 8 个逻辑类型的程序段。依次给每一位赋逻辑值、并依次输出每一位的逻辑值。

## 1.5 运算符与表达式

其实前面代码的字里行间处处都有表达式的存在。表达式是程序组成的基本部分它们通常是由运算符与操作数组成的式子。由于运算符的种类有很多，而且每种运算符又可以与很多种类型的操作数甚至是表达式组成更加复杂的表达式，所以表达式的种类也有很多。最常见的表达式有“算术表达式”、“赋值表达式”、“关系表达式”、“逻辑表达式”等等。

简单的表达式通常由一个运算符和若干个操作数构成，例如数学中的四则运算、逻辑运算中的与或。它们通常用来描述算法中的最基本的操作。多个简单的表达式能够复合成一个复杂的表达式，用以完成复杂的操作。任何一个表达式在运算完成之后都会有一个结果，这个结果包含了某种数据类型的值，称为表达式的值。表达式的值和类型与操作数和运算符的类型有关。

表示内存中某个能够被用户访问的“名字”被称为左值。通常是用户能够读写的变量，并且左值表示的是这个“变量”而不是计算的结果。左值通常出现在赋值符号（等号）的左边，故名左值。左值是一个内存地址，通过左值就可以对该地址下的数据进行处理。与左值对应的则是“右值”。当一个符号或者常量放在赋值符号（等号）右边时，计算机负责读取他们的值，也就是它们所代表的真实值。这个值可能是通过表达式计算而来，也可能是变量、常量等。左值可以看作是“目的地址”，右值相当于“要放入目的地址的数据值”。

在我们的算法中，很少出现只包含两个操作数一个运算符的表达式，更多的是像“`int a = 1 * 2 + 3 / 4 * (5 + 6 / 7)`”。所以说表达式的值不但与表达式内运算符的种类及操作数的类型有关，更与表达式内操作的执行顺序有关。

高级语言都有“隐含”的运算顺序，其中一部分就是数学上的严格规定：例如“先乘除后加减”、“括号最先计算”等等。还有一部分特殊的运算符的特殊的规定：例如“逻辑运算先与后或”等等。这些特殊运算的顺序不再是我们熟悉的顺序，尽管我们可以通过一张详细的“运算符顺序表”来描述清楚，但是我还是建议大家在遇到优先级不明确的运算时一定要用括号进行辅助。因为不论是数学运算还是逻辑运算，“括号内的内容先运

算”是亘古不变的道理。

表达式由众多的基本操作组成。常用的操作主要有赋值操作、算术操作、自增操作、关系逻辑操作、条件运算符等多种。涉及包括单目、双目、甚至三目<sup>20</sup>等多种运算符。其中的一些操作及它们的具体功能会在下面的代码中一一展示！

### 1.5.1 Code

代码 1.5: 运算符与表达式

```

1  #include <iostream>
2  using namespace std;
3  int main(){
4      cout<<"----- 赋值与复合赋值 -----"<<endl;
5      int a = 1;
6      cout<<"a+=9 = "<<(a+=19)<<endl;
7      cout<<"a*=10 = "<<(a*=10)<<endl;
8      cout<<"a/=25 = "<<(a/=25)<<endl;
9
10     cout<<"--- 特殊数学运算（求模） ---"<<endl;
11     int b = 123;
12     cout<<"b%100 = "<<(b%2100)<<endl;
13
14     cout<<"----- 后缀自增自减操作 -----"<<endl;
15     int c = 10;
16     cout<<"before d=c--, c = "<<c<<endl;
17     int d = c--3;
18     cout<<"after d=c--, c = "<<c<<endl;
19     cout<<"after d=c--, d = "<<d<<endl;
20
21     cout<<"----- 前缀自增自减操作 -----"<<endl;
22     int e = 10;

```

<sup>20</sup>所谓“目”就是指操作数的数量，单目运算符就是一个只针对一个操作数进行操作的运算符，例如取反。双目则是操纵两个操作数的运算符，如同数学上的加减乘除等。三目运算符同理。



```

23     cout<<"before f=--e, e = "<<e<<endl;
24     int f = --4e;
25     cout<<"after f=--e, e = "<<e<<endl;
26     cout<<"after f=--e, f = "<<f<<endl;
27
28     cout<<"-----关系运算-----"<<endl;
29     int g = 10;
30     int h = 11;
31     cout<<"g == h? :"<<(g==h)<<endl;
32     cout<<"g > h? :"<<(g>h)<<endl;
33     cout<<"g < h? :"<<(g<h)<<endl;
34
35     cout<<"-----逻辑运算-----"<<endl;
36     bool i = true;
37     bool j = false;
38     cout<<"i&& j = "<<(i&&j)<<endl;
39     cout<<"i||j = "<<(i||j)<<endl;
40     cout<<"!i = "<<(i)<<endl;
41     cout<<"!j = "<<(!j)<<endl;
42
43     cout<<"-----条件运算符-----"<<endl;
44     int k = g>h?g:h5;
45     cout<<"Bigger number between "<<g<<" and "<<h<<" is: "<<k
    <<endl;
46
47     cout<<"-----左运算符与右运算符-----"<<endl;
48     cout<<"10<<1 = "<<(10<<61)<<endl;
49     cout<<"10<<2 = "<<(10<<2)<<endl;
50     cout<<"8>>1 = "<<(8>>1)<<endl;
51     cout<<"8>>2 = "<<(8>>2)<<endl;
52
53     cout<<"-----复杂表达式例子-----"<<endl;
54     int _a = 10;
55     int _b = 10;
56     bool _c = (!(_a==_b))?false:true;
57     cout<<"_c = "<<_c<<endl;
58     return 0;
59 }

```

### 1.5.2 Analyze

- ❶ 这种“+=9”的方式就是复合赋值，其功能就是  $a=a+9$ ，下面的“\*= 功能也一致”
- ❷ “%”运算符是“取模”的意思，所谓取模就是计算余数，例如  $123\%100$ ，就是指 123 除 100 后的余数。
- ❸ “-”是一种一种自减操作，如同“c”的形式则是后缀自减操作，功能是“先用 c 的值，再  $c+=1$ ”
- ❹ 形如“-e”的形式是前缀自减操作，功能是“先  $e-=1$ ，再用 e 的值”
- ❺ 条件运算符是一个三目运算符（三个操作数组成）。形式是：“A?B:C”，即：当 A 成立则其值为 B，不成立则为 C。这种运算与我们后面讲到的 if 语句非常相似。在一些简单的情况下也可以相互替代。
- ❻ 对于数值类型，左右运算符的功能是“位移”

在代码 1.5 中，罗列了很多有关表达式的操作。

第 5 行是前面曾经提到过的基本的赋值表达式。等号右边会进行计算，并将最终的计算结果以变量 a 类型，即 int 的方式<sup>21</sup>赋值给 a。

在后面的几行是几种“复合赋值”表达式。其功能就不再是单一的赋值，而通常是“先进行某种运算，然后再将结果赋值”。例如“+=”就是指“先加后赋值”。“a+=b”就是“ $a=a+b$ ”。这样的写法不但更加易读<sup>22</sup>，同时还能提高效率<sup>23</sup>。其他诸如此类的还有“-=”、“\*=”、“/=”等等非常多的运算符。

代码的第 12 行进行了一种名为“求模”的运算。其功能是“求余数”。求模运算在日常编程中非常有用！它可以与乘除 10 配合，用来截取某一位数字。或者用来产生一个“轮回”的效果等等。

<sup>21</sup>如果不是 int 则转换到 int 类型

<sup>22</sup>只是在 a 的基础上增加一个增量

<sup>23</sup>因为 a 会被调用两次。虽然这种情况效率提升不明显，但是当“a”是个复杂的内容（例如函数调用）的时候，它对效率的影响将会非常明显！

从 14 行开始时自增自减操作，自增自减分为两种：一种是前缀自增自减操作，一种是后缀自增自减操作。这两种操作的区别，大家可以从代码的执行之中推敲出来<sup>24</sup>。

第 28 行开始是“关系运算”。所谓关系运算就是指判断两个操作数的关系。常见的有“>”、“<”、“==<sup>25</sup>”、“!=(不等于)”、“>=”、“<=”等等。关系运算的操作数通常是两个可比较的数，常见的基本数据类型都可以作为它的操作数。比较的结果实际上是 bool 类型。成立则是 1，不成立则是 0。

第 35 行开始是“逻辑运算”。在前面的章节我们已经提到过，逻辑运算就是将“逻辑值”作为操作数的一种运算。逻辑运算主要包括“与——&&”、“或——||”、“非——!”

第 47 行开始介绍了“左运算符(‘<<’)和右运算符(‘>>’)”。这两种运算符是 C++ 中附带的运算符，具体的功能会根据操作数的不同而有所不同。我们也可以根据需要来实现自定义的功能。例子中的运算符的功能是“位移”。“a>>b”就是指将 a 的二进制位右移 b 位，“<<”则表示左移。

程序的最后（56 行）是一个复杂表达式的例子。在这个例子中大家可以看到很多操作其实都是可以放在同一个表达式中执行的。程序首先执行“\_a==\_b”返回值是 true，再对其取反则为 false，所以 \_c 的值应为冒号右边的“true”。面对这种复杂的表达式，应该从运算优先级入手，先计算优先级最高的部分（通常用小括号括起来）。然后再逐层计算得到结果。

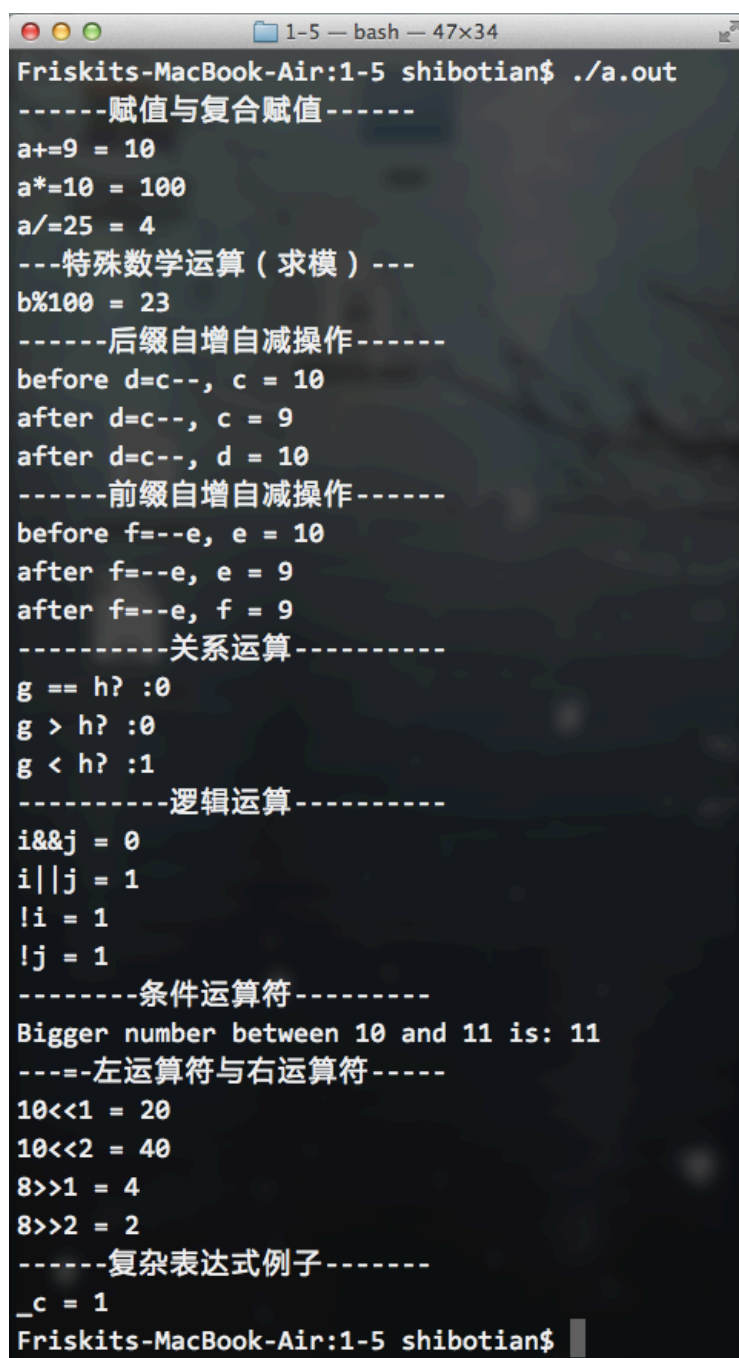
### 1.5.3 Result

代码执行后的输出结果如图：

---

<sup>24</sup>前缀运算是“先运算，后用值”，后缀运算是“先用值后运算”。这里的“后”是指在整个表达式运算完毕之后。例如 `int a = 0; int b = (a++)+(a++)`，实际上是 `b = 0 + 0` 然后 a 再经过两次 ++

<sup>25</sup>关系运算中判断是否相等是两个等号!! 一个等号表示的是赋值符号

A terminal window titled '1-5 - bash - 47x34' on a Mac. The prompt is 'Friskits-MacBook-Air:1-5 shibotian\$'. The user has run './a.out', which displays various C++ operator examples categorized by dashed lines. The examples include assignment and compound assignment, special mathematical operations (modulus), postfix and prefix increment/decrement, relational operations, logical operations, conditional operator, left and right shift operators, and a complex expression example. The terminal output is as follows:

```
Friskits-MacBook-Air:1-5 shibotian$ ./a.out
-----赋值与复合赋值-----
a+=9 = 10
a*=10 = 100
a/=25 = 4
---特殊数学运算（求模）---
b%100 = 23
-----后缀自增自减操作-----
before d=c--, c = 10
after d=c--, c = 9
after d=c--, d = 10
-----前缀自增自减操作-----
before f=--e, e = 10
after f=--e, e = 9
after f=--e, f = 9
-----关系运算-----
g == h? :0
g > h? :0
g < h? :1
-----逻辑运算-----
i&&j = 0
i||j = 1
!i = 1
!j = 1
-----条件运算符-----
Bigger number between 10 and 11 is: 11
----左运算符与右运算符----
10<<1 = 20
10<<2 = 40
8>>1 = 4
8>>2 = 2
-----复杂表达式例子-----
_c = 1
Friskits-MacBook-Air:1-5 shibotian$
```

图 1.5: 运算符与表达式

本节的内容是表达式与运算符。学到这里，你应该已经了解到各种运算符在 C++ 中的使用方法了。

### 1.5.4 Try

**Try 1** 用程序运算  $\frac{(2 \times 3) + 4}{2} \times 0.5$  的结果

**Try 2** 编写一个程序，提示用户输入半径  $r$ ，并且计算以  $r$  为半径的圆的面积和周长，( $\pi = 3.14$ )

**Try 3** 编写一个程序，让用户输入一个 5 位数，然后将这五位数的反过来输出，例如输入 12345，输出 54321。（提示，利用求模和除法）

## 1.6 语句与控制

语句是构成程序的基本单元，通常由简单的语句表示一个简单的动作、或者由复合语句来表示一组复杂的操作。语句是由按照一定语法规则排列的单词组成，单词之间由运算符、分隔符或空格分割。在 C++ 中的语句以分号作为结束的标志。之所以使用一个分号分割语句，是因为 C++ 编译器通常不将“换行”等操作看作是分割语句的标志，甚至我们也可以将多个语句由分号分割后写在同一行。在 C++ 中主要包含四种语句类型：

1. 表达式语句
2. 复合语句
3. 选择语句
4. 循环语句

一般来说，在程序中语句逐行罗列，程序也依次执行。这一种上下两条语句紧密相连的结构被成为“顺序结构”。除此之外，我们在前面的小节中也遇到过“当 A 则 B 否则 C”的执行方式。这种方式由一个条件去控制究竟是 B 被执行，还是 C 被执行？这种结构我们称为“选择结构”。除此之外，当我们要重复处理大量近似的操作时，使用顺序结构是非常麻烦的，所以这时候就有了“循环结构”。利用循环结构，我们可以用简洁的描述实现复杂的功能。这些算法中的抽象的控制结构，在 C++ 中使用“选

择语句”、“循环语句”来体现<sup>26</sup>。在这几种控制语句之中，还穿插着诸如“break”、“continue”等特殊的语句用来进行一些“精巧”的控制。

### **1.6.1 Code**

### **1.6.2 Analyze**

### **1.6.3 Result**

### **1.6.4 Try**

---

<sup>26</sup>顺序语句在哪里？其实除了这两种结构以外我们所逐行书写的代码都属于顺序语句。