从零开始编程  
暨  
从入门到劝退指南

Start in Exploration  
End in Perfect  
止於至善

Last edit: klhsag 2018/9/30 凌晨

当前参与编辑：（暂时掩饰敏感信息）

klh

沈\*\*\*\*

孙\*\*\*\*

目录

[前言 3](#_Toc526030989)

[Chapter 1 准备 5](#_Toc526030990)

[Section 1-1简史 5](#_Toc526030991)

[Section 1-2模型 11](#_Toc526030992)

[Section 1-3语言 13](#_Toc526030993)

[Section 1-4 提升 23](#_Toc526030994)

[Section 1-5 理由 24](#_Toc526030995)

[Chapter 2 基础概念 25](#_Toc526030996)

[Section 2-1 Hello, world! 25](#_Toc526030997)

[Section 2-2 A+B Problem, or more 26](#_Toc526030998)

[Chapter 3 结构化设计 30](#_Toc526030999)

[Chapter 4 过程化设计 32](#_Toc526031000)

[Chapter 5 数据观念 33](#_Toc526031001)

[Chapter 6 面向对象设计 34](#_Toc526031002)

# 前言

编写的主要动力来源于混乱低效的教学。

虽然程序设计乃是当今大学生的必修内容之一（由于考试的要求这又一般特指为C/C++的程序设计，如果能称为程序设计的话），但是从比较广一点的视角来看，大部分人的掌握都最多可以用勉强来形容。这种勉强还是局限在比较好的工科院校的电类专业里可以得出的结论。相当多的人若干项课程结束后依然是处于几乎完全云里雾里的状态。

这种尴尬现象大致有以下几个原因：

1. 课堂缺乏内容。作为大部分人此时需要从零开始的课程，课堂的教学方式往往并不匹配这种形势，往往还谈不上开始就已经让听者进入了听天书状态；后续内容也经常是杂乱无章，主次错乱，轻重不显，难易不分；教学方法常常忽略学生此时迫切的详尽指导的需求，采用照本宣科或任务导向的形式而不顾接受程度。
2. 基础不牢。这里的基础不是指编程方面的基础，我们一般并不能指望这个。但是，更明显的问题是，许多人看似与各种形式的计算机日常打交道，对其工作原理甚至缺乏底线上的认知。由于在一些看起来甚至可笑（至少在教学者看来）的地方没有默认的基础共识，导致一些从入门开始就存在的根本性障碍。因此首先扫除计算机知识上不可原谅的盲点是首要的工作
3. 练习量低。错用中学一般科目的思想来对待编程是非常错误的做法。编程必须以实践为导向，但是在现实的学习中除了上机时间不足这种问题外，更为甚者，还存在进行大量训练纸面题目的方法，这是根本性的错误。如果能够较好地完成上机实践的内容，则纸面内容根本无需大量的练习，不过稍微做一些好的样卷就能基本通关的事。
4. 主次不清。这种现象尤其在老师和学生都采用“祖传典籍”和“上古编译器”时严重，在一些无关紧要乃至错误的地方研究，对于自己在学什么则根本没有把握。比较好的一种测试方法是询问他们对于问题或需求如何在计算机上通过编程进行解决的思路。如果只记住种种样例而不能理解各种形式之所以存在的逻辑，恐怕无法有灵活的运用。
5. 缺乏榜样学习。在这里我们专指优秀的程序设计。列夫·托尔斯泰的名言道：“幸福的家庭彼此相似，不幸的家庭则各有各的不幸。”相似的，对于给定的入门问题，错误的做法多至已经有基础的人不能穷尽想象，而好的做法不过那么几种。然而许多人对此并不重视，以至于开始时糊涂，糊涂地做完，结束后仍然是糊涂的。编程当然以能完成给定的任务为关键目标，但是尤其在初学时不重视规范（以至于优化）自己的做法恐怕会遗患无穷。

以上内容也可以视作对初学者的建议。不过，既然写了这些，我们就要提到另一个理由，那就是缺乏真正引人入门的完整资料，它应当不对读者的水平做出过分的期待，应当足够简明易懂来激发兴趣和信心，应当条理清晰以便作为可以随时翻阅的工具资料，应当着重于思想的传达来指引正确的方向，应当主从协调来增强学习的效率，应当涉足广泛来拓展视界和提升深度，总而言之就是清除那些本不应当存在的沼泽地。

关于这本书应该具体包括一些什么样的内容，这则是一个值得仔细斟酌的问题。C语言当然是一个有打基础作用的语言，但是C++有时则不一定（它是很容易把人带进坑的语言）；java则可能让你一开始就呆在坑里面；但是我们真的就有必要从这几门“经典”语言开始么？python也可能是更适合初学者的选择，MATLAB也未必就不能作为开始，Haskell这种函数式语言或许也能发掘它的作用。

除此之外，这部书应不应该变成某个特定语言的讲解书籍？这确实就是不少书存在的定位，但我认为这样就不存在这本的意义了。这也可以视为教学的又一混乱根源——陷入语言的沼泽而看不清真正的内容是很危险的。因此本书会竭力让你意识到：语言的存在目的是为了方便（而不是作为某种考试的工具等等机械的东西），语言的选择依赖于需求的变化，而如果你能真正掌握核心的实质，这种语言间的转换也并不困难。本书绝不仅仅是为了让你通过所谓的“计算机二级考试”或者可以顺利毕业而写，我们致力于发掘隐藏的真正可能——从一些面向考试的教学到获得属于自我的真知，以及如何真正开始一些开发的工作。

/\*\*\*\*\*\*\*\*\*\*\*\*拟定语言流程大纲\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*比较容易混乱\*\*\*\*\*\*\*交错程度比较高\*\*\*\*\*\*\*\*\*\*

P, C/C++ 基础语言规范，结构化过程化面向对象化

Java设计模式，面向对象的一些问题

MFC, Qt

JavaScript, PHP: ajax

多线程， Modern C++

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Chapter 1 准备

## Section 1-1简史

如何准备一个能尽快让基础不牢的同学尽快进入状态的基础知识大杂烩是一个很难完美解决的问题。不能篇幅过长，讲完所有该讲不该讲的东西，也应该让绝大部分人可以顺利接受之后的内容，最好还要让稍有基础的同学补足一些疏忽的盲点。

故此，我们依据惯例从发展历史展开我们的知识普及。不过，希望你接受我们认为你至少了解中学里必须掌握的那一些常识性知识点。

#### 理论的准备

“众所周知”的是，世界上第一台电子计算机是1946年的ENIAC，不过，根据另一种说法，这是媒体的错误传播，ENIAC不过是第一台有通用能力的计算机，而1942年的ABC样机已能求解线性方程组。

不过，纠结这个第一对我们不是很重要。有些发明是天才的作品，有些则早已为人所预言，只等待技术上的实现；而计算机属于后者。

早在17世纪，欧洲一批数学家就已开始设计和制造以数字形式进行基本运算的数字计算机。1642年，法国数学家帕斯卡采用与钟表类似的齿轮传动装置，制成了最早的十进制加法器。1678年，德国数学家莱布尼兹制成的计算机，进一步解决了十进制数的乘、除运算。

英国数学家巴贝奇在1822年制作差分机模型时提出一个设想，每次完成一次算术运算将发展为自动完成某个特定的完整运算过程。1884年，巴贝奇设计了一种程序控制的通用分析机。这台分析机虽然已经描绘出有关程序控制方式计算机的雏型，但限于当时的技术条件而未能实现。

诗人拜伦的女儿Ada Augusta Byron(1815－1852)（其母则是数学家Anna Isabella Milbanke）突破了当时人对用机器计算数字的设想，提出了对有分析能力机器的构想，对这样的“分析机”如何进行编程做出了详尽的描述。她设计了巴贝奇分析机上解伯努利方程的一个程序，并证明当时的19世纪计算机狂人巴贝奇的分析器可以用于许多问题的求解。她甚至还建立了循环和子程序的概念。由于她在程序设计上的开创性工作，Ada Lovelace被称为世界上第一位程序员。

巴贝奇的设想提出以后的一百多年期间，电磁学、电工学、电子学不断取得重大进展，在元件、器件方面接连发明了真空二极管和真空三极管；在系统技术方面，相继发明了无线电报、电视和雷达……。所有这些成就为现代计算机的发展准备了技术和物质条件。

与此同时，数学、物理也相应地蓬勃发展。到了20世纪30年代，物理学的各个领域经历着定量化的阶段，描述各种物理过程的数学方程，其中有的用经典的分析方法已根难解决。于是，数值分析受到了重视，研究出各种数值积分，数值微分，以及微分方程数值解法，把计算过程归结为巨量的基本运算，从而奠定了现代计算机的数值算法基础。

社会上对先进计算工具多方面迫切的需要，是促使现代计算机诞生的根本动力。20世纪以后，各个科学领域和技术部门的计算困难堆积如山，已经阻碍了学科的继续发展。特别是第二次世界大战爆发前后，军事科学技术对高速计算工具的需要尤为迫切。在此期间，德国、美国、英国部在进行计算机的开拓工作，几乎同时开始了机电式计算机和电子计算机的研究。

1936年，英国数学家阿诺·图灵发表了具有里程碑意义的论文《论数字计算在解决策略问题中的应用》，在文中提出了“图灵机”的构想。图灵机不是一台具体的机器，而是一种抽象计算模型。他证明了任何可计算的函数都存在相应的图灵机，反之，不存在相应图灵机的函数就是不具有可计算其函数值算法的函数。这意味着，任何复杂的可解的问题都可以划归为一些列简单重复的运算。

1938年克劳德·艾尔伍德·香农（Claude Elwood Shannon, 1916—2001）在MIT获得电气工程硕士学位，硕士论文题目是《A Symbolic Analysis of Relay and Switching Circuits》（继电器与开关电路的符号分析）。当时他已经注意到电话交换电路与布尔代数之间的类似性，即把布尔代数的“真”与“假”和电路系统的“开”与“关”对应起来，并用1和0表示。于是他用布尔代数分析并优化开关电路，奠定了数字电路的理论基础。

香农于1940年在普林斯顿高级研究所期间开始思考信息论与有效通信系统的问题。经过8年的努力，香农在1948年6月和10月在《贝尔系统技术杂志》上连载发表了具有深远影响的论文《通讯的数学原理》。1949年，香农又在该杂志上发表了另一著名论文《噪声下的通信》。在这两篇论文中，香农阐明了通信的基本问题，给出了通信系统的模型，提出了信息量的数学表达式，并解决了信道容量、信源统计特性、信源编码、信道编码等一系列基本技术问题。两篇论文成为了信息论的奠基性著作。

冯·诺依曼总结了ENIAC等初代计算机的缺陷，提出了“存储程序”的观念，确立冯·诺依曼体系结构，沿用至今。

#### 硬件的改良

今时不比往昔，我们一直走在全面压榨计算机性能的路上，只不过，最近这些年在想办法利用性能，而在过去的很长一段时间里，最大限度地节约不仅仅是道德操守，实际上还是能产生有用程序的底线。了解一些历史不会让我们在研究一些东西时陷入奇怪的质疑当中。

1981年的IBM PC采用了主频为4.77MHz的Intel 8088CPU，内存64KB, 160KB软驱，操作系统是Microsoft提供的MS-DOS。不难想象那时以至于那之前的计算机程序设计必须有何等的效率。今天我们任何人所看见的一个最小的程序，不，哪怕只是一张图片，都可以撑爆它的内存。有理由庆幸这个时代已经过去，如今我们往往不需要再替硬件斤斤计较，而是专注于我们的软件逻辑，哪怕性能上不是极致压榨的完美。

1965年，英特尔联合创始人戈登·摩尔(Gordon Moore)预测，计算机芯片的处理能力每两年就会翻一番。这一定律持续有效超过四十多年。

由于高纯硅的独特性，集成度越高，晶体管的价格越便宜，这样也就引出了摩尔定律的经济学效益，在20世纪60年代初，一个晶体管要10美元左右，但随着晶体管越来越小，直小到一根头发丝上可以放1000个晶体管时，每个晶体管的价格只有千分之一美分。据有关统计，按运算10万次乘法的价格算，IBM704计算机为85美分，IBM709降到17美分，而60年代中期IBM耗资50亿研制的IBM360系统计算机已变为3.0美分。

从硬件开始，这一行业的发展扩张速度就远超传统。硬件的发展与软件之间形成相互刺激的作用，这一行业的内容因此瞬息万变。

#### 程序设计语言的发展

囿于篇幅所限，以及读者此时的理解程度，这里仅对计算机发展开始时的一些已经比较有定论的内容做必要的陈述。

在电子管计算机时期，一些计算机配置了汇编语言和子程序库，科学计算用的高级语言FORTRAN初露头角。在晶体管计算机阶段，事务处理的COBOL语言、科学计算机用的ALGOL语言，和符号处理用的LISP等高级语言开始进入实用阶段。操作系统初步成型，使计算机的使用方式由手工操作改变为自动作业管理。

原始的计算机程序设计是完全面向硬件的。基于数字电路给出的抽象，我们得以用一定的简单的规则通过某种方式向计算机输入信息。在相当一段时间内，一个成熟的方法是使用卡带。通过在穿孔纸带或卡片上有规律地打孔，就可以把二进制信息输入到计算机内。当然后来这又被磁带所逐渐取代。原始的程序编写就在这上面进行。由于这时计算机性能也不高，所以程序也简易。这个时候我们只有这种被称为“机器码”的编码可以使用。

随着产业发展，记住每个1和0当然是痛苦和不人性的。因此汇编语言应运而生。汇编语言是直接面向处理器（Processor）的程序设计语言。处理器是在指令的控制下工作的，处理器可以识别的每一条指令称为机器指令。每一种处理器都有自己可以识别的一整套指令，称为指令集。处理器执行指令时，根据不同的指令采取不同的动作，完成不同的功能，既可以改变自己内部的工作状态，也能控制其它外围电路的工作状态。本质上来说，汇编语言是面向特定类型乃至型号的机器的，它本质上就是用易记的缩写来代替难以记忆的机器码。

高级语言在汇编的基础上形成了高度封装，语法更加自然，远离了对硬件的直接操作，从而使我们有机会考虑更上层的结构。

在初始的一些尝试后，第一个得到广泛使用的公认的高级语言是FORTRAN(1954)，其发明之初是为了解决科学数值计算，是少数内置矩阵运算的语言，广泛应用于科学和工程计算领域。

LISP语言（1958）由约翰·麦卡锡在1960年左右创造的一种基于λ演算的函数式编程语言，它代表着编程语言体系的另一个分支，尽管看起来并非主流，仍有极大价值和相当范围的运用。

COBOL（1959）是数据处理领域最为广泛的程序设计语言，是第一个广泛使用的高级编程语言。主要应用于数值计算并不复杂，但数据处理信息量却很大的商业领域。

Simula 67（1962）被认为是最早的面向对象程序设计语言，

BASIC（1964）的意思就是“初学者的全方位符号指令代码”，是一种设计给初学者使用的程序设计语言。BASIC是一种解释语言，在完成编写后不须经过编译及连结即可执行，但如果要单独执行仍然需要编译成可执行文件。

Pascal（1968）语言语法严谨，层次分明，程序易写，具有很强的可读性，是第一个结构化的编程语言。Pascal的取名是为了纪念十七世纪法国著名哲学家和数学家Blaise Pascal，它由瑞士Niklaus Wirth教授于六十年代末设计并创立。

C语言（1969）既具有高级语言的特点，又具有汇编语言的特点，由美国贝尔研究所的D.M.Ritchie于1972年推出。1978后，C语言已先后被移植到大、中、小及微型机上。

SQL（1975）是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统。

C++（1983）是一种静态数据类型检查的、支持多重编程范式的通用程序设计语言。它支持过程化程序设计、数据抽象、面向对象程序设计、泛型程序设计等多种程序设计风格。

这之后的语言演化流程则不断更加复杂，虽然百花齐放，但是对于初学者而言，到此为止概念已经足够混乱了。语言的特性之区别不是在这里能够阐释清楚的，更不必说到今天为止一切仍处于争议之中了。切记，在这一领域，很少会有标准答案。

#### 系统

想要学好程序设计而不是处处迷糊陷入迷雾，那么必须对我们的工作环境有一个真正的基本认识。这里我们精要地提及一些关于系统的基本知识。

从1946年诞生第一台电子计算机以来，它的每一代进化都以减少成本、缩小体积、降低功耗、增大容量和提高性能为目标，随着计算机硬件的发展，同时也加速了操作系统(简称OS)的形成和发展。

最初的电脑并没有操作系统，人们通过各种操作按钮来控制计算机，后来出现了汇编语言，操作人员通过有孔的纸带将程序输入电脑进行编译。这些将语言内置的电脑只能由操作人员自己编写程序来运行，不利于设备、程序的共用。为了解决这种问题，就出现了操作系统，这样就很好实现了程序的共用，以及对计算机硬件资源的管理。

操作系统的出现，进一步把一般软件的运行环境与硬件分离开来。在现代操作系统对计算机的全面接管下，各种其他程序并不能直接对硬盘进行操作，所有的请求在底层上都要由操作系统来分配。由此我们可以从针对每种不同的硬件进行特定的编程这种苦海中脱离出来（这是操作系统和硬件驱动的工作），而是针对操作系统给出的形式比较统一的接口来完成我们的工作。

操作系统位于底层硬件与用户之间，是两者沟通的桥梁。用户可以通过操作系统的用户界面，输入命令。操作系统则对命令进行解释，驱动硬件设备，实现用户要求。以现代观点而言，一个标准[个人电脑](https://baike.baidu.com/item/%E4%B8%AA%E4%BA%BA%E7%94%B5%E8%84%91" \t "_blank)的OS应该提供以下的功能：

* [进程管理](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E7%AE%A1%E7%90%86)（Processing management）
* 内存管理（Memory management）
* 文件系统（File system）
* 网络通讯（Networking）
* 安全机制（Security）
* 用户界面（User interface）
* [驱动程序](https://baike.baidu.com/item/%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F)（Device drivers）

操作系统的发展历程详细展开也需要相当的篇幅。不过，其中一个沿用至今的最重要的发展就是从单任务的处理向多道程序并行处理的演变。

在早些时候，人们发现按顺序处理命令的计算机操作系统有一个很大的缺陷：各部分操作速度并不一致。CPU内部的运算可以达到相当快的速度，然而与存储设备交互则是有延迟的，比较起CPU的运算速度这个时间相当长。因此，开发出了“多道技术”，允许CPU在这个等待时间内执行其他任务。

很快，操作系统就从只能单用户同时间执行一个任务向支持多用户多任务处理的转变。这时我们也不单只是为了最大化利用CPU。现今的操作系统一般为不同的程序运行分别建立进程，程序各自可以顺序的执行自己的指令，而操作系统能够为不同的进程分配时间，同时也限制它们各自的权限，在不同的任务当中轮流切换。微观上的快速切换在宏观上就表现为任务同时进行。

进入多核时代使并行的实际作用进一步提高。互相不依赖的操作由此得以真正同时进行，极大地加快里计算机的工作效率。

#### 架构

冯·诺依曼体系结构是一直以来绝大多数计算机的基本组成原理。这方面的基础内容我们希望可以做一个假设，读者已经在基本的学习中已经对这个方面的内容有基础性的了解，否则也就不必再往下看了，应当尽快去补充基本常识。

五大基本组成要素是：运算器、控制器、存储器、输入设备、输出设备。

输入设备与输出设备是用于进行交互的设施，这两部分与计算机的核心功能距离最远，主要目标是为实现人机交互和机器与外部交互提供条件。我们的编程环境一般会隔离具体的设备，而是根据相应配套的驱动程序去在软件的层面上进行交互。一般来讲，具体的输入输出已经确定的内容这一过程不在我们密切关注的点上。不过，如何识别和转化获取的输入信息以及如何生成合适的输出信息有时我们也需要关注，不过核心问题的解决也一般不在具体的设备上而是在计算机内部做处理。因此I/O的具体设备一般不是我们密切关注的点，我们会把问题抽象化，直接切中要害，而不是在怎样美观地输出几个数字之类的问题上作不必要的过多关注。

从CPU（中央处理器）面世以来，控制器与运算器这两个计算机中最核心的部件就是合成到一个芯片上的。事实上，这是很自然的，因为控制与运算实际上有相似的形式（都是因果的推导）。计算机的核心工作就在CPU之中。同时它也是和其他部分密切联系并下达指令的。

存储器同样是核心组成部分。根据冯·诺依曼提出的概念，在这一体系结构下，数据和指令的存储形式统一的，这就完成了一个根本性的进步——程序也可以被存储。在现代的体系下存储器又分为多个层级。

大类上有内存与外存之分。外存是长期存储信息的结构，典型代表是软盘（当今基本被淘汰）、硬盘、U盘与固态硬盘等，顾名思义，它们实际上在计算机内部结构之外，运行时CPU必须把数据从外存拷贝到内存中，然后在内存的基础上操作。

内存可简单分为ROM和RAM。ROM是一般不可修改（往往必须使用物理方法才能修改），但是可以长期保存的存储器，也就是只可读不可写，它们通常刻录了计算机运行的必备指令。RAM则是随机存取存储器，可读可写，但是断电失去数据，CPU通常就在这个区域上对数据进行处理，只在需要持久化保存的时候才把它们重新覆写到外存之上。CPU对内存的访问比对外存要快的多得多。

为了满足高速运算的需要，内存还有更精细的分类。一种有普遍应用的是“寄存器——高速缓冲存储器（Cache）——主存储器”的三级结构。寄存器是CPU的一个核心组成部分，是有限存贮容量的高速存贮部件，用来暂存指令、数据和地址。高速缓存是介于之间的一种存储器，容量较小但是速度接近CPU。

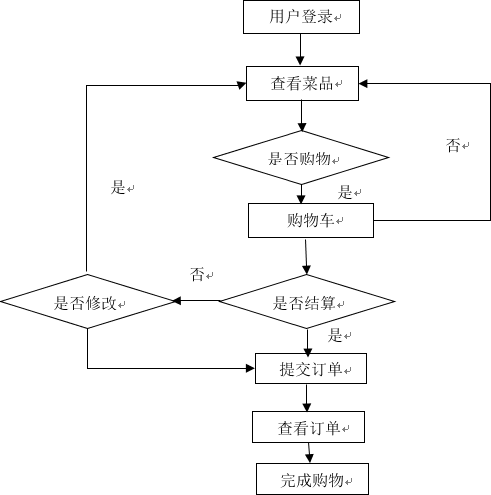
现代CPU的工作机制已经十分复杂，不是单纯的按照顺序执行固定的命令。为了最高效的完成任务，CPU会将“将要被使用多次的”数据复制到高速缓存中，在执行指令时将数据传入寄存器根据指令操作，并且会打乱没有顺序依赖关系的指令来尽可能让电路满负荷工作。

不过，这并不意味着我们的编程工作会变得过于复杂。事实上，这些复杂的机制都是为了保证我们能够用依然简单的程序语言描述出想要实现的内容而无须关心具体工作的细节，并且效率更高。但是，想要有深刻的理解，对计算机内部的工作原理不能没有认识。今后在具体的学习中你还可以更进一步加深这些认识。

## Section 1-2模型

#### 流程图

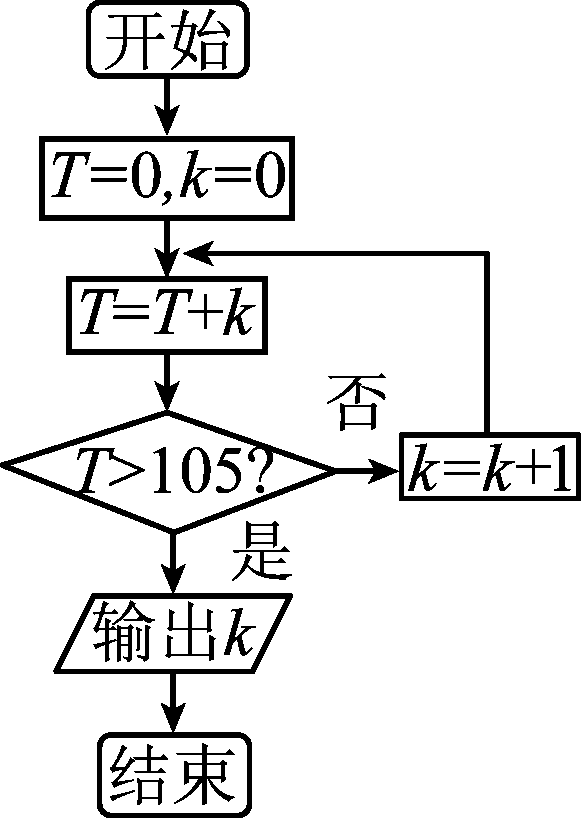
应该可以做这样的假设，此刻，基本上所有的读者都至少对一种被称为“流程图”的东西有所了解。



上面是一个简单的流程图的例子。顾名思义，它可以描述我们进行事务的基本流程，能够形象直观，降低歧义，增强理解。

流程图有助于把我们从一般对话和文章中过分不明确的信息明确出来，清晰地描述开始，进行，转折，结束的具体情形。现在起你也应当纠正一种不合适的想法：计算机编程是为了让计算机能听懂我们而做的妥协。不尽然如此。事实上，更有必要的是，我们有必要审视我们一般的日常语言描述存在的缺陷，正视它们不能有效率地给出确定的指令的现实。如果在学习的过程中不断地感到难以理解各种做法的缘由，那么就有必要审视，你有没有采取正确的思维方式。

自然语言存在随意性和多义性，还容易产生歧义。因此有必要采取规范化、抽象化、形式化的方法。



在这样的典型的流程图中，形式化的语言取代了自然语言。通过使用数学上的一些变量，并运用“赋值”“比较”等操作，我们能完成一些基本的功能。如果你对数学上的“映射”概念有一定的理解，相信你也能清楚，这种方法看起来抽象，却足以完成我们需要的逻辑。

算法

以特定的图形符号加上说明，表示算法的图，称为流程图。

算法（Algorithm）是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。

算法中的指令描述的是一个计算，当其运行时能从一个初始状态和（可能为空的）初始输入开始，经过一系列有限而清晰定义的状态，最终产生输出并停止于一个终态。一个状态到另一个状态的转移不一定是确定的。

算法最基本的要素：

* 有穷性
* 确定性
* 有效性
* 输入与输出

流程图的语言再作进一步抽象就是原始的高级语言的形式了。

#### 程序的生成

高级语言是为了直观展现我们采用的处理流程而出现的，与汇编语言本身就是机器码的缩写不同，高级语言必须经过一定的“翻译”流程才能生成真正为计算机所能理解的可执行代码。

编程语言在这一路径上大致可分为两类，即编译型和解释型，如今这两者界限并不明显。

编译型语言在早期占据主流地位，通过被称为“编译（compile）”的流程，源代码（Source Coding）经过一系列处理后（通常会有汇编语言作为中介之一）被翻译为机器语言的代码，随后运行时只需要运行生成的可执行文件(executable file)就能运行程序，这个过程保障了最大化的效率。执行编译过程的程序被称为“编译器（compiler）”

解释型语言通常依靠一个专门的解释器来进行工作，在执行代码时才对程序进行翻译工作。起初这是方便初学者的考虑。在计算机性能大大提高的今天，解释型语言（如今更流行的一种说法称为脚本语言）凭借极大的便利性赢得广泛的受众。不必编译的特点能够极大的方便程序员测试自己的代码。

编译与翻译的区别非常薄弱，更多是一种根据需要的选择。如今不少语言同时支持两者，赋予开发者更大的自由；此外还有不少语言选择一种中间方案，首先把代码翻译成一种中间格式，然后由专门的程序来执行这些中间代码，这些语言是基于虚拟机语言（现在详细地讨论它还为时尚早）。

然而，只要是在一个有基本的操作系统的环境下做这些工作，最好不要忘记，你的行为实际上受到操作系统的全面监控与接管。（形象地说，最终解释权归操作系统所有——除非你能发现并攻击操作系统的漏洞）

## Section 1-3语言

#### 语言流派

编程语言的派系实际上非常繁多，包含了各种并不相同的思想流派。从哪一种编程语言开始学习，从什么侧重点开始切入，实际上应当是非常审慎的行为。

不幸的是，许多人就是从这个时候开始无法进行学习的。所以这一节我们有必要讨论一下这个重中之重的话题。

通常的取舍在Pascal、C/C++、Python、Java等几种语言中间。

PASCAL语言是第一个达成了整饬的形式的高级语言，在当时的时代独树一帜。不幸的是，当时的作者对其的开发期待是使其成为一种教学语言。毫无疑问，这门严谨规范的语言的教学作用不可低估，但是由于极其严格的规范要求对编程者的自由有着不少限制，后来这一语言逐渐脱离潮流。

C语言功能空前强大，设计之初就是为编写操作系统而生。C语言假设程序员明确自己的行为并充分信任他们进行自由的操作，与此同时C语言又高度贴近汇编的语言结构，同时C语言也是第一个充分考虑如何建立大型软件系统的语言。C语言一直是强大的基石语言。

C++设计之初是为了拓展C语言。在随后的岁月里，它不断地吸收各方面的新的编程思想（虽然这也常常是为人所诟病的一面），更加适用于复杂的软件工程，影响深远。

Python是一种脚本语言，致力于为让使用者把精力集中于解决问题而不是语言内容。Python的一个格言是“用一种方法，最好是只有一种方法来做一件事”。

Java是面向对象的语言，开创了虚拟机类语言的先河，具有极强的跨平台支持能力，规范严格，广泛应用于大规模软件工程开发。

作为良好编程习惯的开端，Pascal实际上具有不容忽视的作用。不过由于近些年来的越发衰落（很大程度上与作为主要支持者的公司衰落有关），大规模基于其进行教学已经基本没有现实基础了。但是，不容忽视的是，Pascal曾一度为相当流行的语言，有不容忽视的地位，许多遗留的文档和规范源自PASCAL。本书虽然不会重点介绍该语言，但是并不会回避，将会有基本的介绍。

C/C++是功能强大的语言，但是错误的学习往往要把学习者引往不归路。C的哲学是程序员同时具有极大的自由，也对自己的自由负责。如果达成真正的理解，你能够以这一媒介获得极深刻的认识，获得触类旁通的秘诀；如果不能有一定理解，恐怕这个语言多半是先给人带来痛苦。

Python进一步隔离程序员与计算机内部细节的联系。对于非专业人员，或者进行研究者，以及希望快捷开发自己所需功能者，这一语言十分适合。

Java恐怕是这之中最不适合初学的语言。Java以组织大型软件工程见长，以繁冗的规范进行协调，用虚拟机隔离计算机模型，从多个角度看从Java开始基础性的学习没有什么优点（除非你打算只学习Java）。

本书根据现今的教学实际，以C++语言的使用作为主要内容，同时也会兼顾PASCAL和C乃至现代语言的做法，力求开拓广阔的视野。

#### 环境搭建

有一种不负责任的说法试图灌输这样一种错误的观念：优秀的程序员可以只用记事本（windows下的notepad）编程。这是一个极其扭曲的认识。记事本最多做一些最简单的工作，它不支持绝大部分的高级编辑功能，更不支持编译，只能用于最简单的程序编写，想要进行复杂的工作恐怕会令人十分痛苦。

学习编程，首先你必须有可以随时可以进行代码编写与测试的机器。某种程度上这不是完全必备的条件，但这要求你的学习更加严谨细致。

随时记住，你最好的合作伙伴，除了当前的这本书以及其他手上的入门书籍，还有如下的一些知识来源：google.com, baidu.com, cn.bing.com, CSDN, etc.. ——拥抱新时代的学习方法吧。

一个比windows下的notepad更合适编程的软件是nodepad++，这个软件是一个精巧的升级版记事本，支持辨识代码格式、语法高亮、代码自动补全等功能，并且能够拓展更多自定义的插件。比起notepad最重要的改进是，notepad++支持无·次数限制的撤销/恢复功能——以避免你的代码过于容易地被破坏，notepad++也支持更强大的在文档中进行查找/替换的功能。然而，notepad++的缺点是，比起其他一些更专业化的的工具，它还是显得简陋。

Sublime Text与eclipse是更加专业性的工具，它们支持更高级的代码组织功能，甚至是加入编译插件。事实上你现在就可以尝试用它们做一些简单的工作，不过想要配置好它们有些繁琐，初学时在这些上面花时间有些偏离重点了，这是我们学习一定深度后更适合考虑的内容。

Vim是为专业程序员定制的文本编辑器，不幸的是对于新手来讲它太难入手了，也不切合入门的实际需要。

较为适合初学者使用与较大规模软件开发的是集成开发环境（IDE）。在IDE中，集成了代码编写环境，编译器，调试器，各种工程管理功能等等为一体，既方便初学者快速上手，也适应于工程上的开发。

Free Pascal IDE于信息学竞赛中有广泛使用，在传统PASCAL上有不少适应时代的改进。距离较近的稳定的推荐的版本是FPC2.4.0。（值得注意的是常见的2.6.2版本有难以忽视的崩溃问题；实话说这个语言没什么必要跟进版本）

Dev-C++是一个传统的竞赛中使用的C++IDE，但是其多方面功能的孱弱使其难堪大用，初学者可以很快用它构建自己的前几个程序，随后我们不建议使用它。需要指出的是它采用GNU C++系列编译器。这是C++语言的一个与其他几乎所有语言都不相同的独特之处——由于C++堪称最复杂的语言，有多家不同的公司和组织都在开发自己的C++产品，C++是一个只有“标准”的语言，而几乎所有编译器都不能（或者不打算）支持全部已制定的的C++标准。不同系列甚至不同版本的C++编译器对C++代码有不同的处理方法。GNU是一系列开源编译器的总称，“开源”意味着编译程序的代码是公开的，这使得GNU C++（常常又简写为G++）成为众多领域的重要标准。

CodeBlocks是一个轻量级的IDE，在不大的空间内集成了大量实用的功能，是大学生程序设计竞赛的标准环境之一。想要使用CodeBlocks学习C++，最好选择下载自带GNU C++编译器的版本，避免一开始就陷入配置的麻烦。

Microsoft Visual Studio 是功能强大的windows下集成开发环境。初学C++可以相当方便地从此入门。目前推荐安装不低于VS2010的版本。其缺点是占用很大空间，安装配置需要腾出不短的时间，启动速度相对较慢。其C++编译器是微软开发的MSVC（Microsoft Visual C++）。

必须针对提出警告的是一款上古IDE：VC6.0。这一编译器目前还在各大高校的各个不密切相关专业中被广泛应用。除去所谓的计算机二级考试这个理由，在当下看来VC6.0的缺陷可谓罄竹难书。当然，考虑到还有人不得不使用它（此处应该表示默哀），我们会适当辨析一些VC6.0独有的问题。

本书并不打算像一些已经比较过时的书籍那样，规定一个标准的参考编译器，不对读者使用不同版本的编译器产生的问题负责——这同样是不合理的。有另一种时常存在的误区（不太清楚其产生来源），有人认为，想要研究一些搞不清的编程问题，只要敲出对应的一些代码进行编译，查看程序运行情况或者翻译出来的汇编代码就可以把情况搞清楚。这种做法某种程度上可以帮助你确定存在一种问题，但是，如果认为用这样的方法就一定能理解问题的原因与原理，则往往是流于表面的，对于C++语言尤其是如此。编译器如何对一段代码进行翻译，通常是遵循一定的标准，而一段对标准有偏差理解而写出来的代码存在的问题经常是隐晦的。有时，由于代码编写的不好，会导致有随机性甚至多义性不确定性；由于编译器的不完善，随着版本更新，原本就糟糕的代码很可能表现出不同的结果。现代的编译器在打开优化开关之后都能够智能地优化生成的代码，如果编写看似没有问题的代码（实则并不规范），可能遭到编译器错误的处理，而版本迭代又容易导致细微之处的不同。想要搞清楚根本性的问题，无论如何还是要了解语言本身规定的标准、编译器实现的标准、以及什么样的做法规范什么样的做法并非规范。

相反，本书的讲解将紧密结合常用的多种编译器，但是并不依赖。我们以跨平台为一个基本点来讲授我们的内容。读者也可以尝试不拘泥于单独的一种环境完成学习。

#### Hello, world!

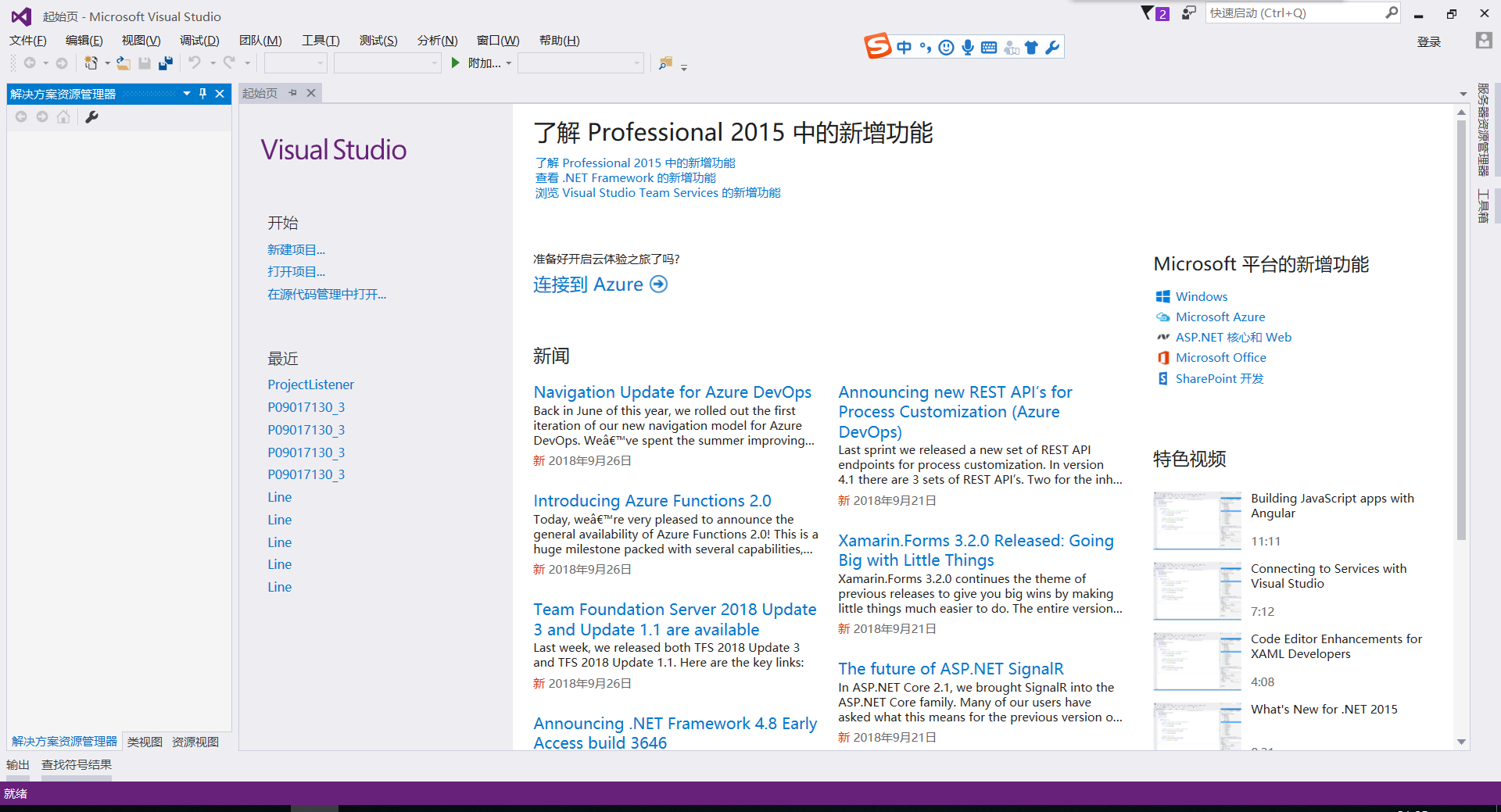
以Visual Studio 2015为例，我们来具体做一个开始编程的指引。

根据此前经历过的一些奇葩遭遇，我想，还是有必要指出，开始一切工作前，我们最好先确认，您已经把系统调整到一个看起来稍微专业点的程度——比如说，您已经在“设置”（或者“控制面板”）中设置好不隐藏所有文件的拓展名。文件的拓展名实际上只是windows为了方便标识应当自动地采用那一种程序来打开某一类文件，但是同时这也意味着导致一些文件名实际不同的文件看起来相同。您应当让系统支持您自行修改文件名后缀，这会让工作更加便利。

如果甚至不清楚如何设置，我们建议您自行搜索相关的内容或询问身边的人。这里的篇幅恐怕不太够用了了。

如果你正确地依照安装程序的指示安装好了带有Visual C++2015组件的VS2015，并完成了诸如激活等麻烦事以后，你就可以进入一个开始界面，准备好开始写第一个C++程序了。

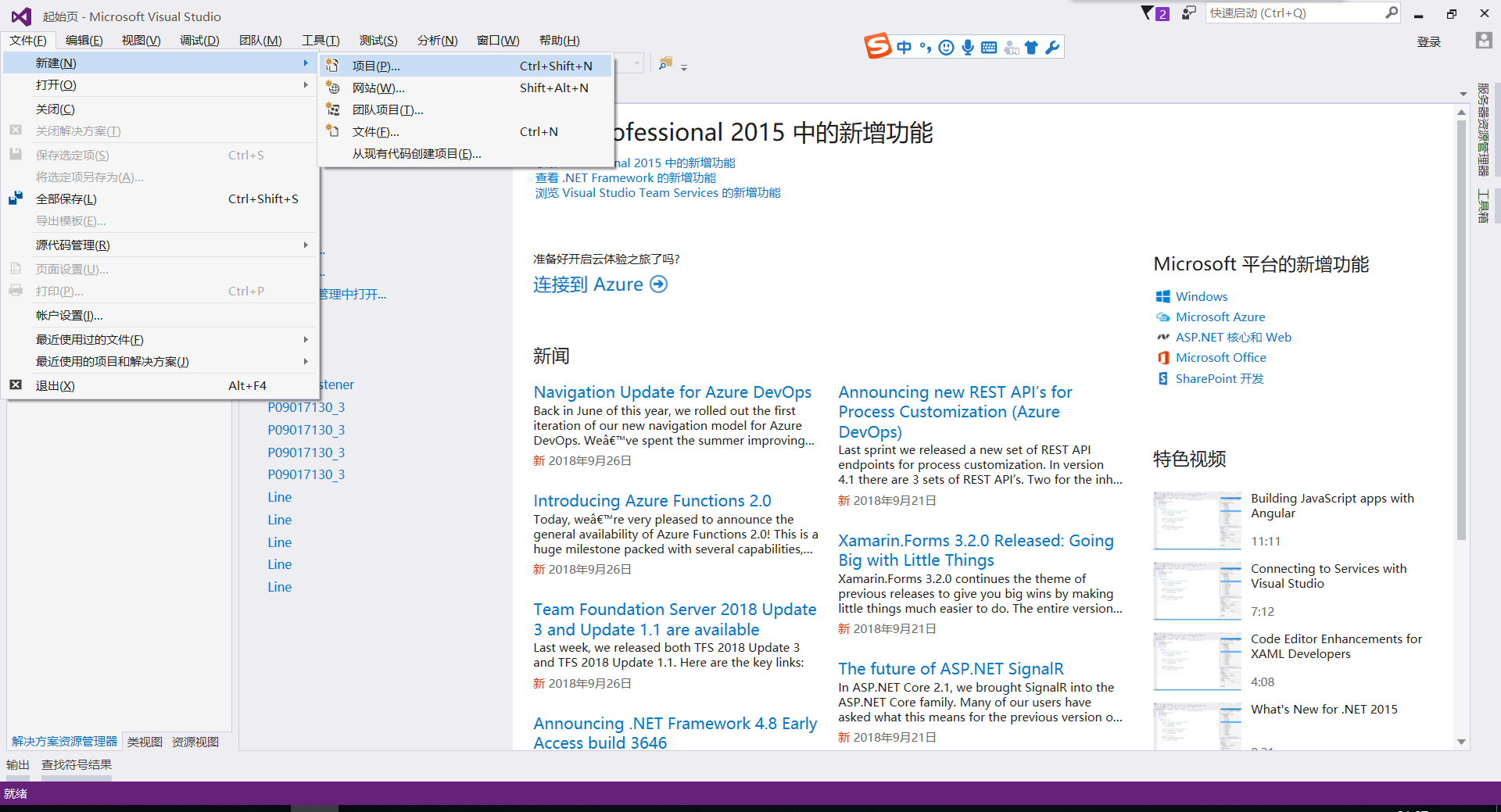
鉴于您可能是第一次进入这个界面，我们来做一点详细的介绍。

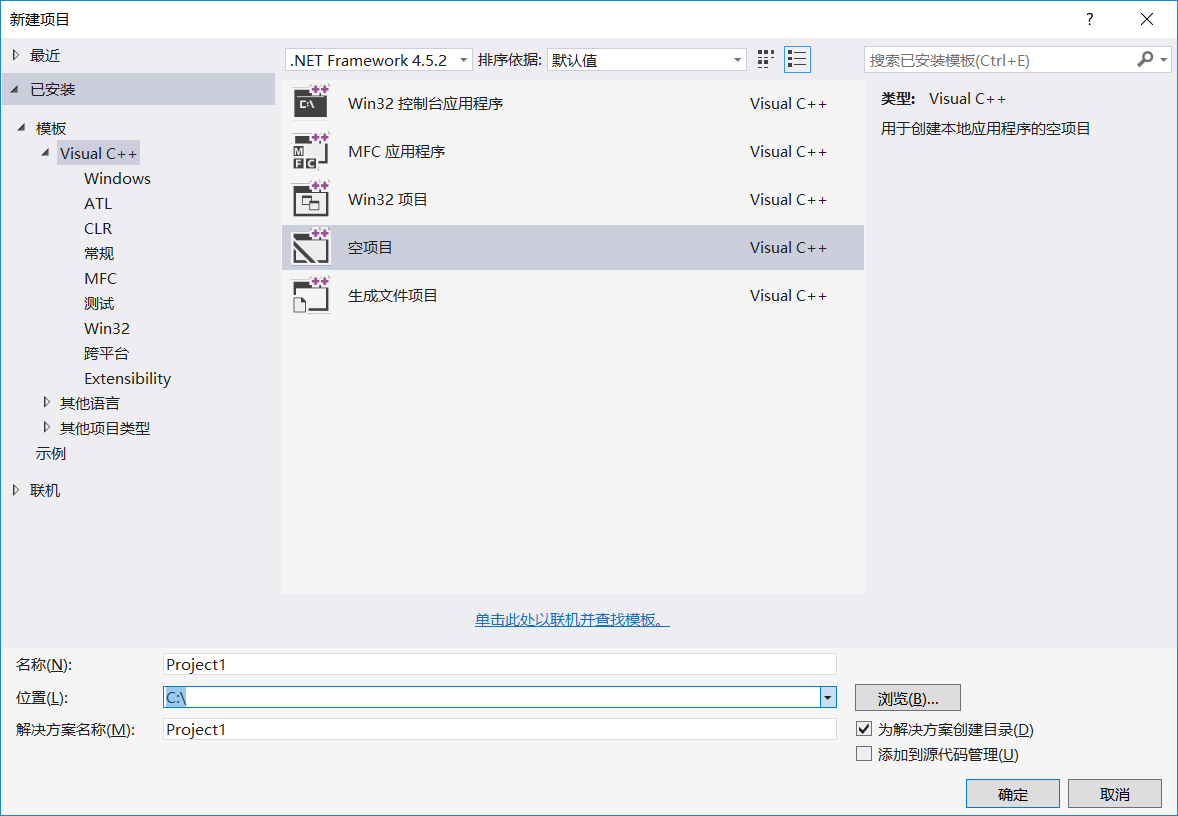


开始第一个程序的编写，您需要点击“文件-新建-项目”，在接下来显示的这个对话框中，选择“模板-Visual C++-常规”中的“空项目”。

值得指出的是，VS作为一个现代的编译器，不再支持您编译简单的单独的源代码——即使只有一个源代码文件，也必须用一个“项目（project）”的形式来组织源码，最终一个或多个项目组织成一个大的单位——“解决方案（solution）”。从源代码生成可执行文件是以解决方案为单位的。

我们建议您选择新建一个便于查看、修改、上传等操作的专用的文件夹来保存源代码，而不是使用默认的设置——这有可能让你在积累一段时间后无法从一堆文件里理清头绪。



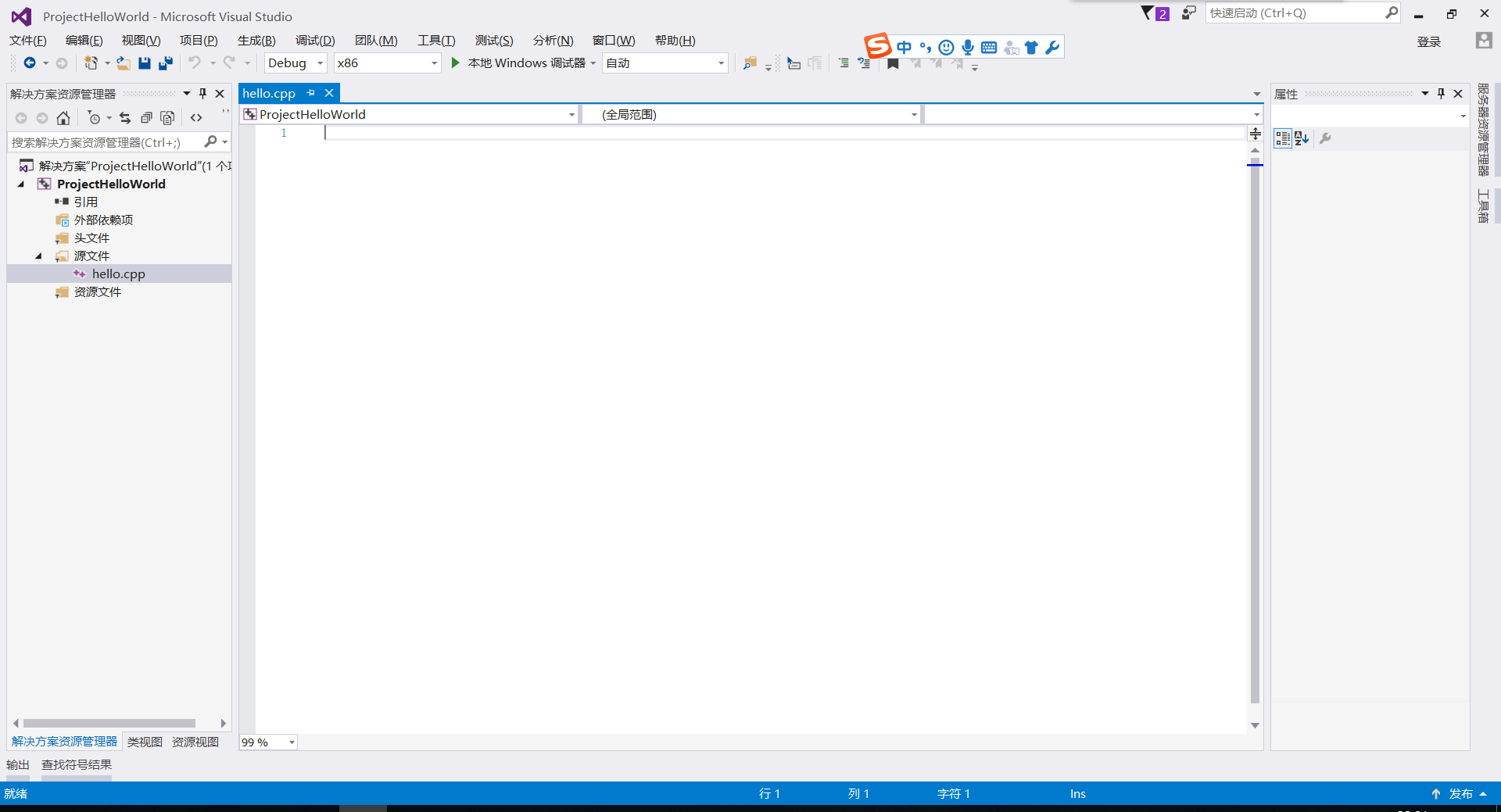


我们不妨取Project名为”ProjectHelloWorld”。然后点击确定来创建项目。

在“解决方案资源管理器”中，可以看到这个新的解决方案（只包含一个项目）已经被打开，不过它还是空的。

右击项目名或子目录名“源文件”，在弹出菜单中选择“添加->新建项”，选择使用.cpp格式，我们不妨命名这个文件为hello.cpp，随后点击“添加”。

这样我们就添加了一个“hello.cpp”，它是项目中最关键最核心的源代码部分。



现在你可以从键盘上输入下面的这段程序。

#include <iostream>

**using** **namespace** std**;**

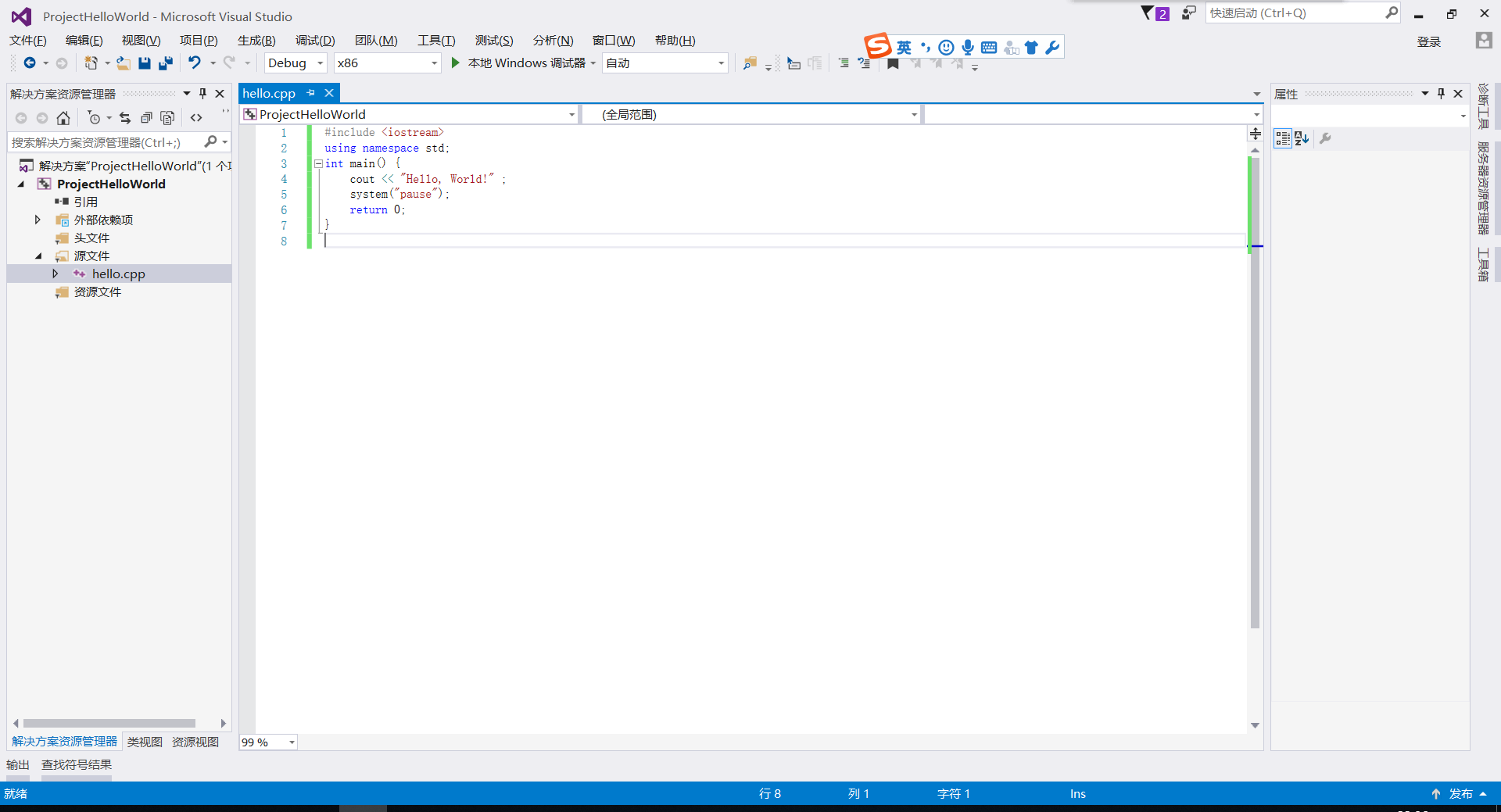
int main**()** **{**

cout **<<** "Hello, World!" **;**

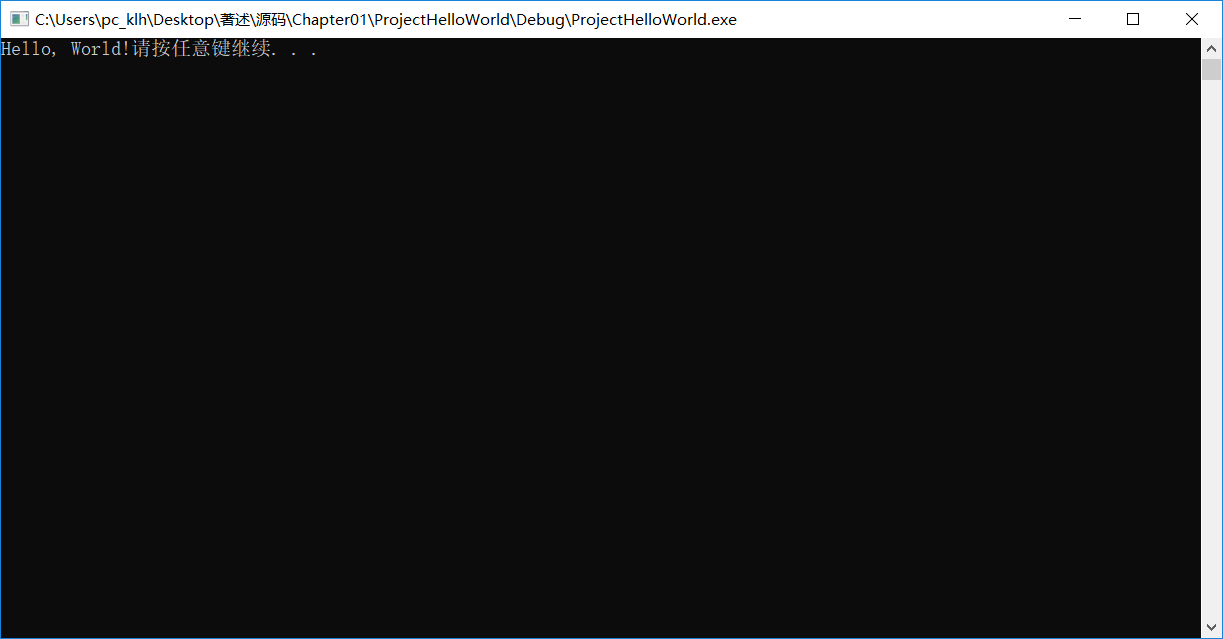
system**(**"pause"**);**

**return** 0**;**

**}**



随后，我们可以按下快捷键F5（请记住这个按键）来尝试编译并运行这段代码。典型的状况应该是下图所示。



我们可以简单地讲一讲为什么会有这样的运行结果。

在这一段标准的C++程序的开头，为一个C++程序的开始做了一些准备工作，在此暂时不做一些详细的介绍。程序的主要执行过程是在int main(){...}当中完成的。

cout << ... ;这一句简单地向屏幕输出内容，根据C++规范，每一条语句都必须用分号（注意区分中英文符号）作为结尾。这看起来还是清晰明了的。

system(“pause”);下达了一个指令，它的作用也表现的很明显：请求一个暂停。之所要有这一句，是因为这个简单程序在想要实现的功能——显示一句”hello, world”执行完毕之后，即将结束，糟糕的是，接下来VS就会在程序执行结束时立刻把这个对话框关掉——这将导致我们什么都看不到。暂停可以防止VS在我们来得及确认之前把窗口关掉。你也可以不局限于这种方法，更多方法此处限于篇幅暂不展开。

随后的return 0; 标志着程序的正常结束。这句话一般也可以不写（并不非常推荐这么做），因为当代编译器基本都可以自动为你补全这一句固定的话。

为什么是这样的界面？

相信许多人一开始对编程的希望并非如此。你可能希望做出一些看起来炫酷的东西，你可能希望能够很快地去满足自己以前的种种幻想——这一切很合理，但是尝试最快速度得到它们而不计较背后的原理，你很快就会陷入各种莫名其妙不可逾越的障碍。

也许这听起来有些令人沮丧，但是请面对面对这样的事实：虽然，利用一些玩具般的东西也可以做出图形化的界面，可是那通常也意味着已经有自动化的工具在背后为你生成了海量的代码——这可不是合理的学习途径。我们的编程学习必须从一些基本的内容开始，不能在基础未牢之时就试图踩在可视化界面这个巨人的肩膀上。

很快，如果你领会了接下来要学习的内容，你也将意识到，一个看起来可视化的界面也不过是一个包装。而我们，最需要的能力之一就是拆开包装，能看清楚事实上发生了什么事。一个美观的界面可能是对于非专业性的用户是友好的，然而，专业地，我们应该关注真正的核心——很多时候与界面无关。

早在这门科学建立之时，还没有如今的建立可视化桌面的需求。因此，传统上，我们不依赖一个个复杂的图形，而是输出一行又一行的字符，同时也可以接受字符的输入，很多时候它们都仅仅只是ASCII字符。虽然简陋，这样使我们能把事务集中于我们真正要解决的问题。这之后发展出的图形化界面实际上都是额外附加在这之上的内容，你可以比喻其为“胶囊的糖衣”（糖这个概念是时常出现的比喻，这提示我们总是更应该关注本质而不是似是而非的外在）。

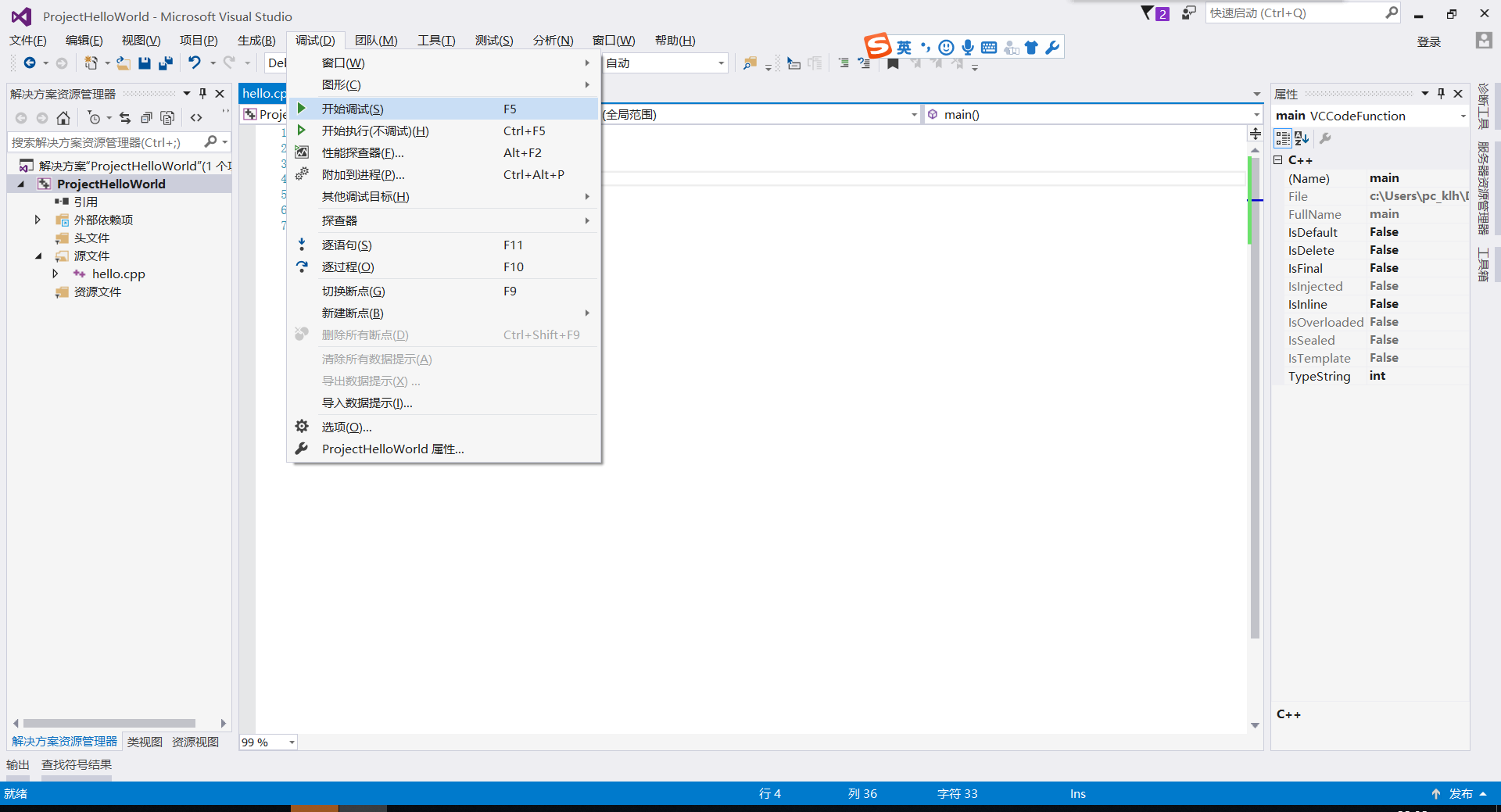
即使在当今广泛流行的桌面操作系统，如Windows、Ubuntu Linux等等，也都保留有一个命令行模式的界面，一般统称为“控制台”(console。在这个界面字符提示逐行输出，你需要从键盘按一定规则输入命令，一般而言，你需要按下回车键来确认，这才能让计算机开始读取你的输入。

如果想变得专业一些，你需要打破自己对GUI的依赖。图形界面在你不清楚要干什么时可以更好的提示你，但是移动鼠标其实并不是多有效率的做法；当你明确自己要做什么，你也许会发现熟练从键盘传达指令实际上更加有效率。

我们再看一下上方的菜单，熟悉编程后我们可以很容易的区分每一栏包括的大致内容。

在调试栏下，可以看到我们按下的F5键对应的其实是“开始调试”命令。

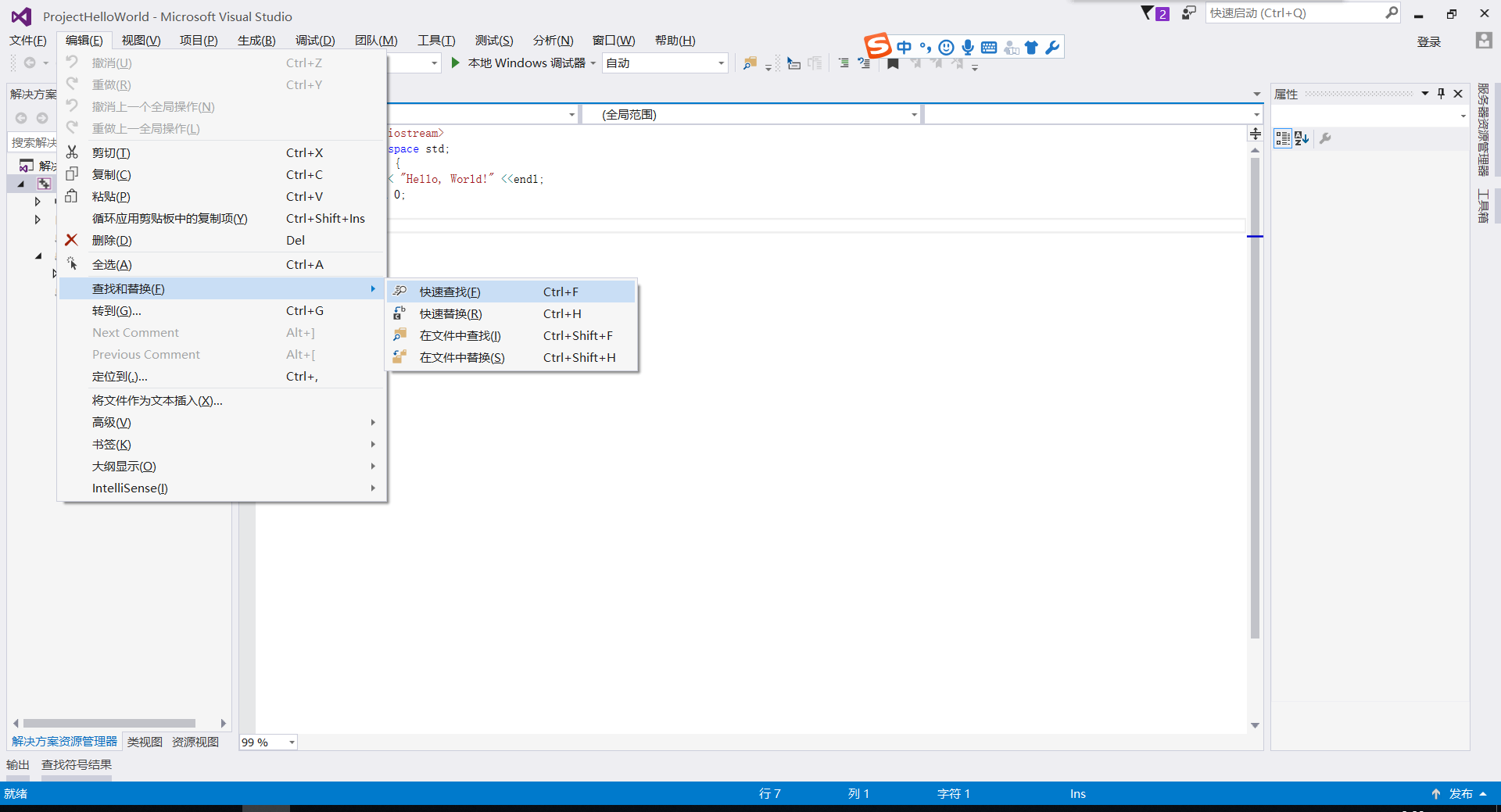
注意到下面另一个按钮“开始执行（不调试）”



这涉及到另一些工程上的概念。调试（debug）和发行（release）是一组非常重要的概念。在开发过程中，有时我们希望软件的发行版本具有最优的性能，但是在测试阶段，我们希望一旦发生错误能够比较好的定位出错误来源——这两者有点矛盾。选择不同的编译类型会让编译器作出适应的不同选择。

值得一提的是，release版本会自动在上面这个程序末尾添加system(“pause”)指令。因此你为了手动调试而添加的这条语句不宜在发行的时候保留。另外，许多其他编译器有不同的准则，多数时候都不需要这一句命令。

此外编辑菜单的内容也是你需要多看几眼的——对于高效地编写与修改代码十分有帮助。



记住如下一些快捷操作键将会有极大的帮助。它们之中的大部分不仅可用于VS，并且是广泛适用于各个编辑器。

剪切 Ctrl + X

复制 Ctrl + C

粘贴 Ctrl + V

撤销 Ctrl + Z

恢复 Ctrl + Y

查找 Ctrl + F

替换 Ctrl + H （有些编辑器如CodeBlocks对此例外）

调试 F5 （许多编译器则是使用F9）

## Section 1-4 提升

不少常规性的教学有一个很大的缺点，就是评价标准极其不明确。这使得不少人在不断勉强地完成作业之后越来越积累不良的做法。如果你致力于切实提升自己的水平，有一些更好的方法。

在线评测系统能够为你解决许多问题。在一个在线评测系统中，一般你只需要提交符合要求的源代码，然后由系统来运行并根据运行结果来评价——通常是根据一组或多组给定的测试数据检验你的程序的输出是否正确。这种方法有助于你养成正确的解决问题的思路——按照一定的规约协约编写实现要求的功能的程序，而不是养成这样一种糟糕的模式——勉勉强强完成一些任务，寄希望于不严格的检查，也不讲究能否总是给出正确的结果，然后可能还有很多精力投入用一些蹩脚的文档或报告来糊弄。

比较适合一般水准的一个在线评测网站是LeetCode，它涵盖多方面的编程知识，可以比较全面的检查你的基础能力。我们不建议你按照随意的顺序做题——这会导致你经常遇到与你的精力集中位置不匹配的题目；有必要事先按照一定的研究做一些分类。

牛客网是国内一个比较好的选择。

如果希望提升自己的算法能力，或者通过超量学习来真正的提升效率，你可以选择poj.org和codeforces等网站（你可能需要相当一定的基础才能开始）。

书籍方面阅览各个相关方面的的必读书目也是非常重要的基础，随时应当开始。尽管如此，我们仍优先你推荐先读完这本书的相关内容——许多书都实际上不够面向缺乏基础的本科以上学生而写。

一个简单的书单：（均有中文版）

《C++ Primer Plus》/《C++ Primer》

《Effective C++》

《Learn Python The Hard Way》

《Python基础教程》

《Head First Java》

《算法导论》

《算法竞赛入门经典》&训练指南

《挑战程序设计竞赛》

## Section 1-5 理由

“编程就是用另一种语言来造物啊，上帝创造宇宙，你创造代码的世界。

“我觉得编程很amazing。”

——孙\*\*\*\*

# Chapter 2 基础概念

## Section 2-1 Hello, world!

在上一章的简单例子中，我们使用了一个被称为”hello world”的简单程序。事实上，绝大部分语言都可以用着这样的形式来做一个精炼的介绍。通常当你转移到一个新的环境，你可以用这个简单的程序来测试当前的环境，观察是否可以得到预期的结果。

一般地，在不同语言中”Hello World”有大体上一致而细节上并不相同的写法，下面是一些经典的示范。

======hello.pas======

{PASCAL语言, 这是多行注释 。

注意不要有尝试嵌套多行注释的想法，在右大括号出现前注释不会被认为结束}

**program** helloworld**;**

**begin**

writeln**(**'Hello, world!'**);**

**end.**

======hello.c / hello.cpp======

/\* C语言,

这是多行注释 \*/

//VC的C编译器 和 所有C++也接受这种风格的单行注释

#include <stdio.h>

int main**() {**

printf**(**"Hello, World!"**);**

**return** 0**;**

**}**

======hello.java======

// Java语言, 这是一行注释

public class hello **{**

public static void main**(**String**[]** args**)** **{**

System**.**out**.**println**(** "Hello, World!"**);**

**}**

**}**

即使还未有系统的学习，相信看出这几段代码的含义还是比较简单的。实际上，大部分语言的helloworld只会更简单。

为何执行这样简单功能的程序包含一些看起来不相关的很多内容？

像python等脚本语言旨在为程序员提供便利，因此这个简单的功能只需要输入一句话，然后回车就能执行。但是对于编译型语言就不一样了。我们源代码文件有一定的组织结构。因此编译时要解决一个问题：程序的运行从何处开始？

虽然有时实际的实现随实际情况的千变万化而有所不同，不过各个主要的编译型语言都提供了默认的机制。除此之外，这些语言在每个程序的开始处还要做一些必要的准备工作。

为什么会有注释？

根据标准的百科式的解释：“注释就是对代码的解释和说明，其目的是让人们能够更加轻松地了解代码。注释是编写程序时，写程序的人给一个语句、程序段、函数等的解释或提示，能提高程序代码的可读性。”

注释中的内容在编译时会被忽略，它们存在的意义正如名称所言，能够起到提示的作用，这能避免抽象的代码不被人理解，并且你可以总结一大段代码的总体功能方便自己和其他人后来的理解，这样就不必每次都要详细地一行行读代码才能理解某段代码实际上做了什么。

也有一种说法认为，“应当让代码自己注释自己”。一段写的足够好的代码能容易地让人理解其意图，而依赖注释不能特别地改变糟糕的代码有多难阅读和修改；不过必要时使用注释仍然是明智的。

“Hello, world!”这一句显示在命令行上的输出看起来有些简陋和僵硬——事实上在真正的工程中，我们往往不会用这种方法来给出我们的输出，有时是接入到一个GUI界面，有时是转换成一个文件或数据包用来进行数据的传递。然而，大部分语言都能提供这样简单的输出功能以提供便利。这样，你在初学阶段和某些需要做测试的时候，就能免去为了能显示一些简单的文字来检验自己的程序有没有正确运行这些小事也要大动干戈的麻烦。

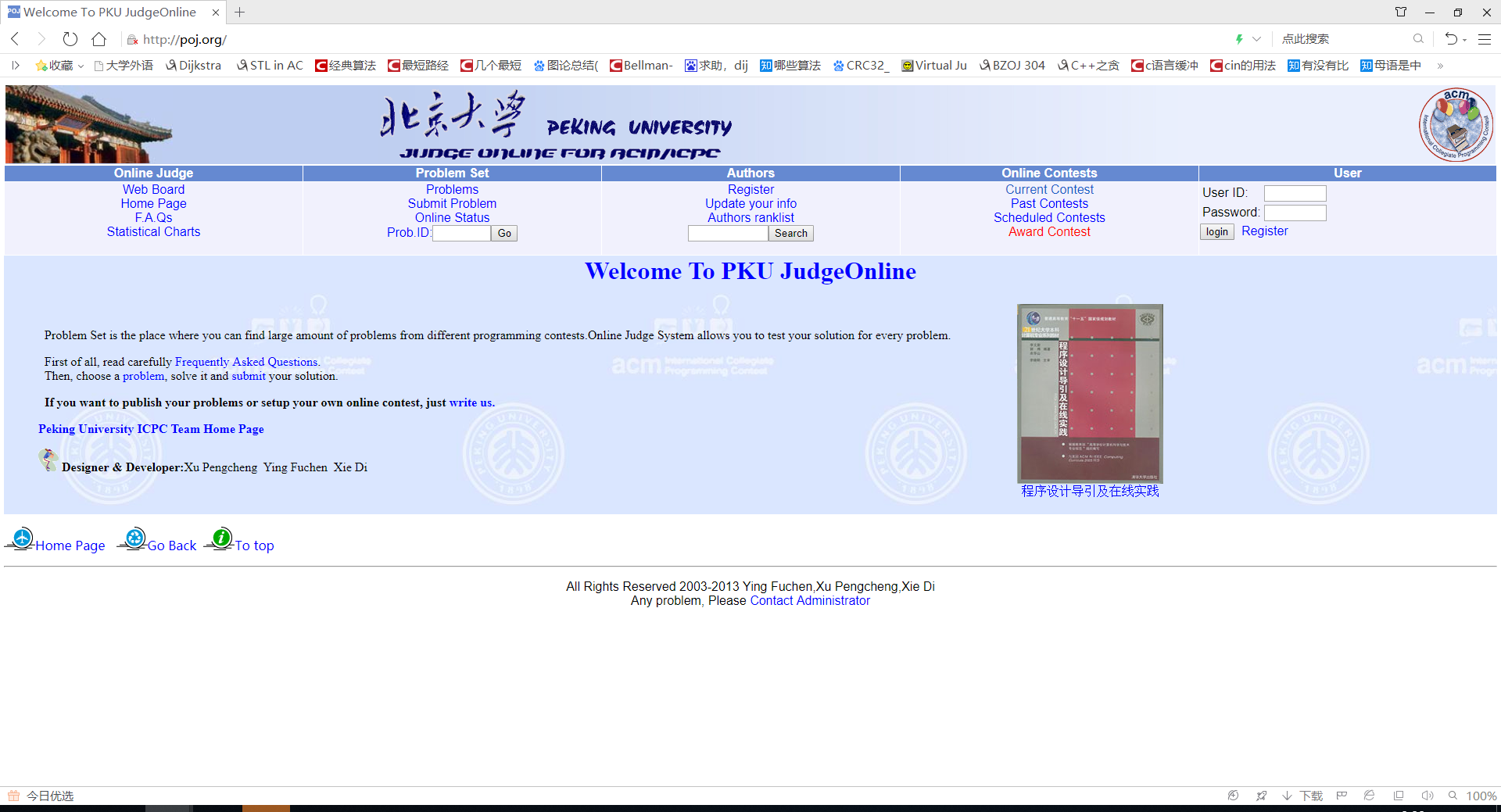
当然，只能输出一些事先指定的句子是远远不满足我们的需求的。接下来我们来可以看一看怎样做一个简单的计算工作。

## Section 2-2 A+B Problem, or more

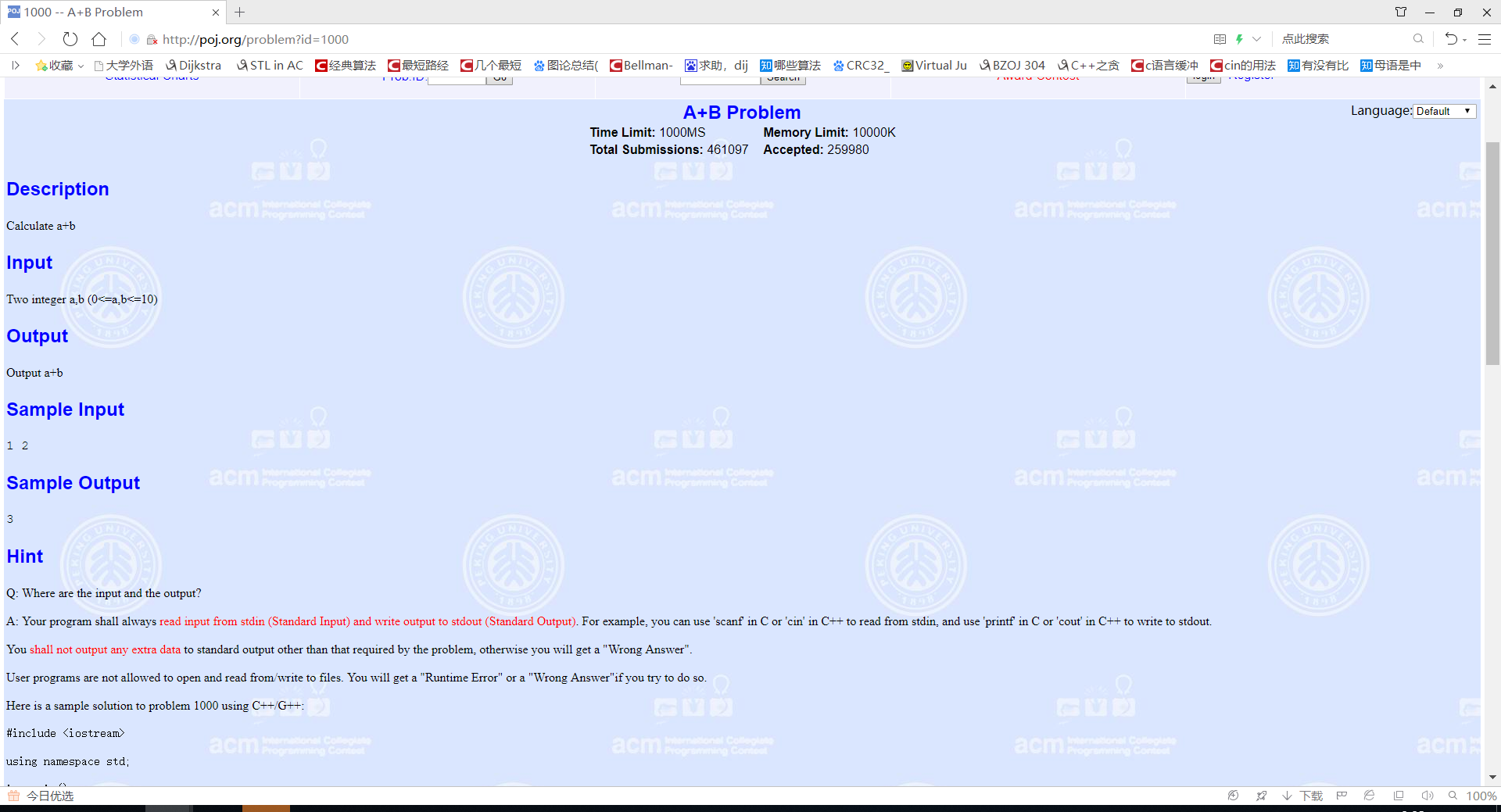
现在我们来看一看另一个入门程序的例子。

登陆poj.org，尝试开始提交一个入门级的程序：A+B Problem

当然你首先要注册一个账号。注册方法就不赘述了。



输入Problem ID为1000——poj题库当中的第一题。



一般来讲题面是纯英文的，不过通常不构成阅读障碍。第一题的页面提供了非常详细的信息来帮助你提交人生中的第一题——甚至包括了标准程序（标程，std）。现在你可以复制其中第一段程序，并把它提交（submit）。

提交时注意选择所使用的语言，这里第一段程序是C++的，你可以选择G++（GNU C++编译器，建议选择）或C++（MSVC）提交。

随后就会自动跳转到status list页面，通常会快就能评测出你的代码的正确性。完全通过会表示为Accepted。

#include <iostream>

**using** **namespace** std**;**

int main**()** **{**

int a**,**b**;**

cin **>>** a **>>** b**;**

cout **<<** a**+**b **<<** endl**;**

**return** 0**;**

**}**

现在我们来考虑一下这道题目是如何被解决的。（虽然很简单，但是这样能解释得比较简单一些）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*有待修改\*\*\*\*\*\*\*\*\*\*\*\*

其中#include是一个编译指令，在这里指C/C++中包含头文件命令，用于调用编译器的类库路径里面的头文件。iostream则是指iostream库，它指的是输入输出流，它能够为下列cin和cout等的使用提供必要的条件。这个程序从main函数开始执行，int说明了这个函数返回一个整数值。其中int a,b; int sum;都为声明语句，int说明了变量的类型，a和b都是变量的名称，注意每个语句都必须以分号结尾，否则程序就会进行报错。cin这条语句在执行时等待着用户输入变量的值，>>为流提取运算符，可以提取用户输入的数据。=为赋值运算符，赋值运算符的左边为左值，左值必须为一个可修改的值，不能是常量，赋值运算符的右边为右值，往往是一个值或者是一个表达式。cout这条语句在执行提取了输出流，<<为流插入运算符，此运算符右边的值会插入到输出流中，endl为换行符，是end line的缩写，endl输出一个换行符。下面我们再来看一个简单的例子来熟悉一下我们所介绍的内容。

这是一个求圆的周长和面积的程序，这个程序需要你输出圆的半径从而求得圆的周长和面积。

#include <iostream>

using namespace std;

int main() {

double r,c,s; //定义圆的半径,周长以及面积

const double Pi = 3.1415; //定义圆周率为一个恒常量

cin >> r; //输入圆的半径

c = 2 \* Pi\*r; //计算圆的周长

s = Pi\*r\*r; //计算圆的面积

cout << "The C of the Circle is " << c << endl; //输出圆的周长

cout << "The S of the Circle is " << s << endl; //输出圆的面积

return 0;

}

double是双精度浮点数类型（一种较为精确的浮点数类型，另一种浮点数类型为float），在这里使用double就意味着r,c,s这些变量的类型都为双精度浮点数类型。const是一个关键字，const修饰的数据类型为常类型，常类型的变量或者是对象的值是不能够被改变的，这意味着我们所定义的Pi为一个常量。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Chapter 3 结构化设计

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*有待修改\*\*\*\*\*\*\*\*\*\*\*

计算机除了存储数据以外还能够对数据进行分析、合并、抽取、修改、推断，合成以及其他操作。为了防止在进行操作的过程中发生数据的丢失和破坏，程序需要有执行重复的操作和进行决策的工具。在C++中我们将采用结构化程式设计的方式来提高计算机运行的效率。结构化程式设计由一些简单、有层次的程式流程架构所组成，大致上可分为顺序、选择及循环这三种结构。下面我们首先来介绍一下循环这部分。在很多时候程序需要重复执行某一项任务，如将止于至善这句话打印10遍、求n的阶乘。这时候就需要用到C++中的循环。在C++中循环可分为for循环，while循环以及do while循环这三种。我们来通过下面这个程序来看一下for循环是如何工作的。

#include <iostream>

using namespace std;

int main()

{

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

cout << "The loops is begin." << endl;

cout << "The loops is end." << endl;

return 0;

}

下面是该程序的输出：

The loops is begin.

The loops is begin.

The loops is begin.

The loops is begin.

The loops is begin.

The loops is end.

由输出可知for循环是按如下步骤进行工作的

1. 设置初始值
2. 判断是否满足循环条件，决定循环是否继续执行
3. 执行循环操作
4. 更新用于测试的值

在该循环中，程序首先将初始值整数变量i设置为0,其次循环部分开始判断变量i是否满足循环条件i<=5，如果满足条件，那么将会执行循环体cout<<”The loops is begin.”<<endl;，在执行完循环体之后，将对循环体设定的整数变量进行更新i++，在这里我们使用了++运算符——递增运算符，它会将操作数的值加1，如i的值本身为3，在执行完这条语句之后i的值就会在循环体完成后增加到4，递增运算符不只限用于for循环，在后面的内容中，我们还要详细地介绍它。到此for循环便执行了一个周期。下面我们来看看for循环的组成。

for(表达式1; 表达式2; 表达式3)

循环体

在整个for循环中初始化只会进行一次，通常表达式1将被用来进行初始化，在后续的程序中将对初始化所设置的变量进行操作。

表达式2通常决定了循环体是否被执行，通常来说，这个表达式是关系表达式，即对两个值进行比较。如上述程序中我们将i的值与5进行了比较，如果比较的结果为真，则程序将执行循环体，如果比较的结果为假，则for循环就会结束。实际上，这个表达式可以为任意表达式，C++会将表达式的结果强制转换成bool类型，表达式的值只会是true或者是false，如果是true则循环继续执行，如果是false则循环结束。

表达式3为更新表达式，在每次循环结束时执行，此时循环体已经执行完毕，通常来说，表达式3都是通过对初始化时设置的变量进行增减，它可以是任意的有效的C++表达式。

for循环的循环体由一条语句组成，也可以由一个语句块组成，在后面的使用中我们将进行声明。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Chapter 4 过程化设计

# Chapter 5 数据观念

# Chapter 6 面向对象设计