内容简介

本书主要是汇编各书和参考资料而成，比较系统地介绍了计算机文化、和程序设计。通过这两部分有机的结合（前者占 1/3，后者占 2/3），即理论与实践结合，使学生理解和掌握有关计算机和信息技术的基本概念和基本原理，对计算机学科有全局性的认识；学会使用计算机进行信息处理，熟练掌握 C++语言编程技术，为后续相关课程的学习打好基础。本书层次分明，由浅入深，具有学习和实用双重意义。

本书可作为高等院校各专业一、二年级学生的教学参考书和技术资料，对广大从事计算机相关研究和应用开发的科技人员也有很大的参考价值。
《计算概论》是普通高校面向理工科低年级学生开设的计算机基础教育课。课程前 1/3 部分为计算机文化，后 2/3 部分为程序设计。

任教此课两年来，发现没有合适的教材，因此根据授课经验，编成此书。
目录

前言

第1章 引论

1.1 计算机科学 ................................................................................................................. 2
1.2 摩尔定律 ......................................................................................................................... 3
1.3 SCOPE OF PROBLEMS ........................................................................................................... 5

上篇 计算机文化 ............................................................................................................. 8

第2章 计算机系统

2.1 COMPUTER INTRODUCTION ................................................................................................. 9
  2.1.1 TURING MODEL ........................................................................................................... 10
  2.1.2 VON NEUMANN MODEL ............................................................................................. 15
  2.1.3 Computer components ................................................................................................... 17
  2.1.4 History ............................................................................................................................. 18
  2.1.5 Practice set ......................................................................................................................... 21

2.2 计算机系统漫游 ................................................................................................................. 22
  2.1.1 Information is Bits + Context .............................................................................................. 24
  2.1.2 Programs Are Translated by Other Programs into Different Forms ................................... 26
  2.1.3 It Pays to Understand How Compilation Systems Work ..................................................... 28
  2.1.4 Processors Read and Interpret Instructions Stored in Memory ......................................... 30
  2.1.5 Caches Matter ...................................................................................................................... 36
  2.1.6 Storage Devices Form a Hierarchy ....................................................................................... 37
  2.1.7 The Operating System Manages the Hardware ................................................................. 38
  2.1.8 Systems Communicate With Other Systems Using Networks ......................................... 45
  2.1.9 The Next Step ......................................................................................................................... 47
  2.1.10 Summary ........................................................................................................................ 47

第3章 数据和数的表示 ............................................................................................................. 50

3.1 数据的表示 ......................................................................................................................... 50
3.2 数的表示 .......................................................................................................................... 64
第 4 章 程序设计语言和开发环境.................................................................................. 68
4.1 程序设计语言........................................................................................................ 68
4.2 开发环境................................................................................................................ 78

下篇 程序设计............................................................................................................. 84

第 5 章 C++基础.......................................................................................................... 85
5.1 GETTING STARTED............................................................................................... 85
5.2 FUNDAMENTAL TYPES....................................................................................... 89
5.3 ARITHMETIC OPERATOR....................................................................................... 91
5.4 CONTROL STRUCTURES....................................................................................... 93

第 6 章 数组和结构..................................................................................................... 104
6.1 数组....................................................................................................................... 104
   6.1.1 Initializing arrays.............................................................................................. 105
   6.1.2 Accessing the values of an array....................................................................... 106
   6.1.3 Multidimensional arrays .................................................................................. 107
6.2 结构....................................................................................................................... 115
   6.2.1 Data structures.................................................................................................. 115
   6.2.2 Pointers to structures....................................................................................... 119
   6.2.3 Nesting structures............................................................................................ 122
Quiz: Structures........................................................................................................ 122

第 7 章 C++标准库..................................................................................................... 124
7.1 C LANGUAGE LIBRARY....................................................................................... 125
7.2 INPUT/OUTPUT STREAM LIBRARY................................................................. 126
7.3 STRING LIBRARY.................................................................................................. 127
7.4 STL: STANDARD TEMPLATE LIBRARY........................................................... 127

第 8 章 函数和递归..................................................................................................... 130
8.1 FUNCTIONS WITH NO TYPE. THE USE OF VOID.............................................. 135
8.2 ARGUMENTS PASSED BY VALUE AND BY REFERENCE..................................... 137
8.3 DEFAULT VALUES IN PARAMETERS.................................................................. 139
8.4 OVERLOADED FUNCTIONS.............................................................................. 140
8.5 INLINE FUNCTIONS............................................................................................. 142
8.6 RECURSIVITY...................................................................................................... 142
8.7 DECLARING FUNCTIONS.................................................................................... 146
第1章 引论

计算机文化这个词的出现到被广泛认可的时间并无确切的考证，但基本上是在20世纪80年代后期。计算机开始是一种装置，进而到一门学科，再发展成为一种“文化”，它对人类的影响力之大令人惊叹。计算机文化是指能够理解计算机是什么，以及它如何被作为资源使用的。简单地说，计算机文化不但是要知道如何使用计算机，更重要的是知道什么时候使用计算机。

在当今世界，几乎所有专业都与计算机息息相关。但是，只有某些特定职业和学科才会深入研究计算机本身的制造、编程和使用技术。用来诠释计算机学科内不同研究领域的各个学术名词的涵义不断发生变化，同时新学科也层出不穷。五个主要的计算机学科（disipline of computing）包括1：

- 计算机工程学（Computer Engineering），是电子工程的一个分支，主要研究计算机软硬件和二者间的彼此联系。
- 计算机科学（Computer Science），是对计算机进行学术研究的传统称谓。主要研究计算技术和执行特定任务的高效算法。该门学科为我们解决确定一个问题在计算机领域内是否可解，如可解其效率如何，以及如何形成更加高效率的程序。时至今日，在计算机科学内已经派生了许多分支，每一个分支都针对不同类别的问题进行深入研究。
- 软件工程学（Software Engineering），着重于研究开发高质量软件系统的方法学和实践方式，并试图压缩并预测开发成本及开发周期。
- 信息系统（Information Systems），研究计算机在一个广泛的有组织环境（商业为主）中的计算机应用。
- 信息技术（Information Technology），指计算机相关的管理和维护。

《计算概论》课程关注的是计算机学科。较大规模的致力于计算机科学的组织有：美国计算机协会（Association of Computing Machinery，简称 ACM）；美国电气电子工程师协会（Institute of Electrical and Electronics Engineers，简称为 IEEE）。

---

第 1 章 引论

1.1 计算机科学

计算机科学是一门包含各种各样与计算和信息处理相关主题的系统学科，从抽象的算法分析、形式化语法等等，到更具体的主题如编程语言、程序设计、软件和硬件等。作为一门学科，它与数学、计算机程序设计、软件工程和计算机工程有显著的不同，却通常被混淆，尽管这些学科之间存在不同程度的交叉和覆盖。2

计算机科学研究的课题是：
- 计算机程序能做什么和不能做什么（可计算性）；
- 如何使程序更高效的执行特定任务（算法和复杂性理论）；
- 程序如何存取不同类型的数据（数据结构和数据库）；
- 程序如何显得更具有智能（人工智能）；
- 人类如何与程序沟通（人机互动和人机界面）。

计算机科学的大部分研究是基于“冯·诺依曼计算机”和“图灵机”的，它们是绝大多数实际机器的计算模型。为此模型的开山鼻祖，邱奇-图灵论题（Church-Turing Thesis）表明，尽管在计算的时间，空间效率上可能有所差异，现有的各种计算设备在计算的能力上是等同的。尽管这个理论通常被认为是计算机科学的基础，可是科学家也研究其它种类的机器，如在实际层面上的并行计算机和在理论上概率计算机、oracle 计算机和量子计算机。在这个意义上讲，计算机只是一种计算的工具；著名的计算机科学家 Dijkstra 有一句名言“计算机科学之关注于计算机并不甚于天文学之关注于望远镜。”。

计算机科学根植于电子工程、数学和语言学，是科学、工程和艺术的结晶。它在 20 世纪最后的三十年间兴起成为一门独立的学科，并发展出自己的方法与术语。

早期，虽然英国的剑桥大学和其他大学已经开始教授计算机科学课程，但它只被视为数学或工程学的一个分支，并非独立的学科。剑桥大学声称有世界上第一个传授计算机的资格。世界上第一个计算机科学系是由美国的普渡大学在 1962 年设立，第一个计算机学院于 1980 年由美国的东北大学设立。现在，多数大学都把计算机科学系列为独立的部门，一部分将它与工程系、应用数学系或其他学科联合。

计算机科学领域的最高荣誉是 ACM 设立的图灵奖，被誉为是计算机科学的诺贝尔奖。它的获得者都是本领域最为出色的科学家和先驱。华人中首获图灵奖的是姚期智博士。他于 2000 年以其对计算理论做出的诸多“根本性的、意义重大的”贡献而获得这一崇高荣誉。

2 http://zh.wikipedia.org/wiki/计算机科学
1.2 摩尔定律

Moore's law describes a long-term trend in the history of computing hardware. Since the invention of the integrated circuit in 1958, the number of transistors that can be placed inexpensively on an integrated circuit has increased exponentially, doubling approximately every two years. The trend was first observed by Intel co-founder Gordon E. Moore in a 1965 paper. It has continued for almost half of a century and is not expected to stop for another decade at least and perhaps much longer.

Almost every measure of the capabilities of digital electronic devices is linked to Moore's law: processing speed, memory capacity, even the number and size of pixels in digital cameras. All of these are improving at (roughly) exponential rates as well. This has dramatically increased the usefulness of digital electronics in nearly every segment of the world economy. Moore's law describes this driving force of technological and social change in the late 20th and early 21st centuries.
计算机第一定律——摩尔定律

归纳起来，主要有以下三种“版本”:

- 集成电路芯片上所集成的电路的数目，每隔 18 个月就翻一番。
- 微处理器的性能每隔 18 个月提高一倍，而价格下降一倍。
- 用一个美元所能买到的电脑性能，每隔 18 个月翻两番。

![图 1-2 Computer Speedup](http://baike.baidu.com/view/17904.htm)

Moore's Law: “The density of transistors on a chip doubles every 18 months, for the same cost” (1965)

半导体集成电路的密度或容量每 18 个月翻一番

Moore's Law is still valid. His law has nothing to do with the speed of the processor. It has to do with the number of transistors which is still doubleing every couple of years. Case in point there is now multiple cores in the same space instead of one core.

戈登·摩尔 (Gordon Moore)，CPU 生产商 Intel 公司的创始人之一。1965 年提出“摩尔定律”，1968 年创办 Intel 公司。摩尔 1929 年出生在美国加州的旧金山
第 1 章 引论

曾获得加州大学伯克利分校的化学学士学位，并且在加州理工大学 (CIT) 获得物理和化学两个博士学位。50 年代中期他和集成电路的发明者罗伯特·诺伊斯 (Robert Noyce) 一起，在威廉·肖克利半导体公司工作。后来，诺伊斯和摩尔等 8 人集体辞职创办了半导体工业史上有名的仙童半导体公司 (Fairchild Semiconductor)。仙童成为现在的 Intel 和 AMD 之父。1968 年，摩尔和诺伊斯一起退出仙童公司，创办了 Intel。Intel 初期致力于开发当时计算机工业尚未开发的数据存储领域，后来，Intel 进行战略转移，专攻微型计算机的 “心脏” 部件 -- CPU。

1.3 Scope of Problems

What can you do with 1 computer?
What can you do with 100 computers?
What can you do with an entire data center?

http://en.wikipedia.org/wiki/Distributed_computing#Projects

Projects:

A variety of distributed computing projects have grown up in recent years. Many are run on a volunteer basis, and involve users donating their unused computational power to work on interesting computational problems. Examples of such projects include the Stanford University Chemistry Department Folding@home project, which is focused on simulations of protein folding to find disease cures and to understand biophysical systems; World Community Grid, an effort to create the world's largest public computing grid to tackle scientific research projects that benefit humanity, run and funded by IBM; SETI@home, which is focused on analyzing radio-telescope data to find evidence of intelligent signals from space, hosted by the Space Sciences Laboratory at the University of California, Berkeley (the Berkeley Open Infrastructure for Network Computing (BOINC), was originally developed to support this project); LHC@home, which is used to help design and tune the Large Hadron Collider, hosted by CERN in Geneva; and distributed.net, which is focused on finding optimal Golomb rulers and breaking various cryptographic ciphers.

http://folding.stanford.edu/English/Main
http://zh.wikipedia.org/wiki/Folding@home

http://www.equn.com/folding/
Folding@home 是如何工作的呢？
Folding@home 是一个研究研究蛋白质折叠，误折，聚合及由此引起的相关疾病的分布式计算工程。使用联网式的计算方式和大量的分布式计算能力来模拟蛋白质折叠的过程，并指引我们近期对由折叠引起的疾病的一系列研究。

图 1-3 Folding@home

图 1-4 Shrek © Dreamworks Animation, rendering multiple frames of high-quality animation
Indexing the web (Google)


Simulating an Internet-sized network for networking experiments (PlanetLab)
http://www.planet-lab.org/

PlanetLab 是一个全球研究网络，支持新网络服务的开发。自 2003 年初以来，超过 1,000 名顶尖院校和工业研究实验室的研究人员使用 PlanetLab 开发新的技术，如分布式存储、网络映射、P2P 系统、分布式哈希表和查询处理。PlanetLab 目前有 1128 个节点，分布在 511 个地点。

Speeding up content delivery (Akamai)

美国 Akamai 是国际上最大的 CDN 服务商，它的巨大网络分发能力在峰值时可达到 15Gbps。Akamai 公司是为数不多的旨在消除 Internet 瓶颈和提高下载速度的几家新公司之一，是一个致力于网络交通提速的“内容发布”公司，是波士顿高技术区最卓越的新兴企业之一。Akamai 公司向全球企业提供发送互联网内容，汇聚媒体应用程序的服务（目前，该公司为 15 个国家的企业管理着 8000 多台服务器）。1998 年，丹尼尔·L 和麻省理工学院的一些研究人员一起创立了这家公司，他在麻省理工学院的硕士论文构成了 Akamai 公司最初的“自由流”（Freeflow）技术的核心。
上篇  计算机文化

上篇的主要目的是向读者介绍有关计算机和信息技术的基本概念和基本原理，使读者能够对计算机学科有全局性的认识。
2.1 Computer Introduction

The phrase computer science has a very broad meaning today. However, in this book, we define the phrase as "issues related to the computer". This introductory chapter first tries to find out what a computer is, then investigates other issues directly related to computers. We look first at the Turing model as a mathematical and philosophical definition of computation. We then show how today's computers are based on the von Neumann model. The chapter ends with a brief history of this culture-changing device...the computer.

Objectives
After studying this chapter, the students should be able to:

- Define the Turing model of a computer.
- Define the von Neumann model of a computer.
- Describe the three components of a computer: hardware, data, and software.
- List topics related to computer hardware.
- List topics related to data.
- List topics related to software.
- Discuss some social and ethical issues related to the use of computers.
- Give a short history of computers.
2.1.1 TURING MODEL

The idea of a universal computational device was first described by Alan Turing in 1937. He proposed that all computation could be performed by a special kind of machine, now called a Turing machine. Although Turing presented a mathematical description of such a machine, he was more interested in the philosophical definition of computation than in building the actual machine. He based the model on the actions that people perform when involved in computation. He abstracted these actions into a model for a computational machine that has really changed the world.

Perceptual knowledge (感性认识)
计算机组成部分
http://net.pku.edu.cn/~course/cs101/2008/video/computer_components.flv
Introduction to Computer Hardware
http://net.pku.edu.cn/~course/cs101/2008/video/intro2computer_hardware.flv

Install http://net.pku.edu.cn/~course/cs101/2008/video/flvplayer_setup.exe, if your computer can not show videos.

图 2-1 Mother board (主板：集成多个部件、适配器，提供它们之间的互联)
主板(Main Board)又名主机板、系统板、母板，是 PC 机的核心部件。PC 机的主板包括 CPU、芯片组 (Chipset)、高速缓存 (Cache)、ROM_BIOS 芯片、CMOS 芯片、内存 RAM、总线通道、软硬磁盘接口、串行和并行接口、USB 接口、扩展槽 (Slots)、直流电源插座、可充电电池以及各种条线。

图中从上到下，左到右：内存条，磁盘、光驱等的数据线接口；CPU 风扇（一般下面是散热器，和 CPU）；棕色 AGP 槽：只能接显卡；白色 PCI 槽：能接显卡、网卡、声卡等。

图 2-2 CPU = 运算器+控制器

图 2-3 Alan Turing, founder of computer science, and artificial intelligence


图灵是举世罕见的天数学家和计算机科学家，仅仅在世 42 年。他的英年
早逝,像他横溢的才华一样，令世界吃惊与难以置信。生命虽然短暂，但那传奇的
人生，丰富多彩的创造力和智慧而深邃的思想，使他犹如一颗耀眼的明星，持续
地照耀着人间后世在科学的浩瀚太空里探索未来的人们。

自上个世纪 60 年代以来，计算机技术飞速发展，信息产业逐渐成为影响人类社
会的最重要的工业之一。支持技术与工业发展的理论基础是计算机科学。众所周知，“诺贝尔奖”是世界上最负盛名的奖项，但仅用于奖励那些在物理、化学、文
学、医学、经济学与促进世界和平等方面做出开拓性重大贡献的人士。“图灵奖”则
是计算机科学领域的最高奖项，有“计算机界诺贝尔奖”之称。设立这个大奖，既
是为了促进计算机科学的进一步发展，也是为了纪念一位天才数学家、计算机
科学的奠基人艾兰・图灵。

http://zh.wikipedia.org/wiki/图灵

图灵被视为计算机科学之父。1931 年进入剑桥大学国王学院，毕业后到美国
普林斯顿大学攻读博士学位，二战爆发后回到剑桥，后曾协助军方破解德国的著名密码系统 Enigma，帮助盟军取得了二战的胜利。

图灵对于人工智能的发展有诸多贡献，例如图灵曾写过一篇名为《机器会思
考吗？》（Can Machine Think?）的论文，其中提出了一种用于判定机器是否具有
智能的试验方法，即图灵试验。至今，每年都有试验的比赛。

此外，图灵提出的著名的图灵机模型为现代计算机的逻辑工作方式奠定了基
础。

http://net.pku.edu.cn/~course/cs101/2008/video/alan_turing.flv

A short video describing the life and unfortunate death of Alan Turing.
http://zh.wikipedia.org/wiki/姚期智

姚期智，美籍华人，计算机科学家，2000 年图灵奖得主，是目前唯一一位获得此
奖项的华人及亚洲人。目前是清华大学理论计算机科学研究中心教授。

因为对计算理论，包括伪随机数生成，密码学与通信复杂度的诸多贡献，美国计
算机协会（ACM）决定把该年度的图灵奖授予他。

Data processors

![Figure 1.1 A single purpose computing machine](image)

Figure 1.1 A single purpose computing machine
Before discussing the Turing model, let us define a computer as a **data processor**. Using this definition, a computer acts a black box that accepts input data, processes the data, and creates output data (Figure 1.1). Although this model can define the functionality of a computer today, it is too general. In this model, a pocket calculator is also a computer (which it is, in a literal sense).

Another problem with this model is that it does not specify the type of processing, or whether more than one type of processing is possible. In other words, it is not clear how many types or sets of operations a machine based on this model can perform. Is it a specific-purpose machine or a general-purpose machine?

This model could represent a specific-purpose computer (or processor) that is designed to do a single job, such as controlling the temperature of a building or controlling the fuel usages in a car. However, computers, as the term is used today, are **general-purpose** machines. They can do many different types of tasks. This implies that we need to change this model into the Turing model to be able to reflect the actual computers of today.

**Programmable data processors**

The Turing model is a better model for a general-purpose computer. This model adds an extra element to the specific computing machine: the program. A **program** is a set of instructions that tells the computer what to do with data. Figure 1.2 shows the Turing model.

In the Turing model, the **output data** depends on the combination of two factors: the **input data** and the program. With the same input, we can generate different outputs if we change the program. Similarly, with the same program, we can generate different outputs if we change the input data. Finally, if the input data and the program remain the same, the output should be the same. Let us look at three cases.

---

![Figure 1.2 A computer based on the Turing model: programmable data processor](image-url)
Figure 1.3 shows the same sorting program with different input data, although the program is the same, the outputs are different, because different input data is processed.

Figure 1.3 The same program, different data

Figure 1.4 shows the same input data with different programs. Each program makes the computer perform different operations on the input data. The first program sorts the data, the second adds the data, and the third finds the smallest number.

Figure 1.4 The same input, different program

We expect the same result each time if both input data and the program are the same, of course. In other words, when the same program is run with the same input
data, we expect the same output.

The universal Turing machine
A universal Turing machine, a machine that can do any computation if the appropriate program is provided, was the first description of a modern computer. It can be proved that a very powerful computer and a universal Turing machine can compute the same thing. We need only provide the data and the program -- the description of how to do the computation -- to either machine. In fact, a universal Turing machine is capable of computing anything that is computable.

A computer is a machine that manipulates data according to a list of instructions.

2.1.2 VON NEUMANN MODEL

Computers built on the Turing universal machine store data in their memory. Around 1944-1945, John von Neumann proposed that, since program and data are logically the same, programs should also be stored in the memory of a computer.

Four subsystems
Computers built on the von Neumann model divide the computer hardware into four subsystems: memory, arithmetic logic unit, control unit, and input/output (Figure 1.5).

Figure 1.5 von Neumann model
Memory is the storage area. This is where programs and data are stored during processing. We discuss the reasons for storing programs and data later in the chapter.

The arithmetic logic unit (ALU) is where calculation and logical operations take place. For a computer to act as a data processor, it must be able to do arithmetic operations on data (such as adding a list of numbers). It should also be able to do logical operations on data.

The control unit controls the operations of the memory, ALU, and the input/output subsystems.

The input subsystem accepts input data and the program from outside the computer, while the output subsystem sends results of processing to the outside world. The definition of the input/output subsystem is very broad: it also includes secondary storage devices such as disk or tape that store data and programs for processing. When a disk stores data that results from processing, it is considered an output device; when data is read from the disk, it is considered as an input device.

The stored program concept
The von Neumann model states that the program must be stored in memory. This is totally different from the architecture of early computers in which only the data was stored in memory; the programs for their tasks implemented by manipulating a set of switches or by changing the wiring system.

The memory of modern computers hosts both a program and its corresponding data. This implies that both the data and programs should have the same format, because they are stored in memory. In fact, they are stored as binary patterns in memory -- a sequence of 0s and 1s.

Sequential execution of instructions
A program in the von Neumann model is made of a finite number of instructions. In this model, the control unit fetches one instruction from memory, decodes it, and then executes it. In other words, the instructions are executed one after another. Of course, one instruction may request the control unit to jump to some previous or following instructions, but this does not mean that the instructions are not executed sequentially. Sequential execution of a program was the initial requirement of a
computer based on the von Neumann model. Today's computers execute programs in the order that is most efficient.

### 2.1.3 Computer components

We can think of a computer as being made up of three components: computer hardware, data, and computer software.

**Computer hardware**

Computer hardware today has four components under the von Neumann model, although we can have different types of memory, different types of input/output subsystems, and so on.

**Data**

The von Neumann model clearly defines a computer as a data processing machine that accepts the input data, processes it, and outputs the result.

The von Neumann model does not define how data must be stored in a computer. If a computer is an electronic device, the best way to store data is in the form of an electrical signal, specifically its presence or absence. This implies that a computer can store data in one of two states.

Obviously, the data we use in daily life is not just in one of two states. For example, our numbering system uses digits that can take one of ten states (0 to 9). We cannot (as yes) store this type of information in a computer; it needs to be changed to another system that uses only two states (0 and 1). We also need to be able to process other types of data (text, image, audio, and video). These also cannot be stored in a computer directly, but need to be changed to the appropriate form (0s and 1s).

In Chapter 3, we will learn how to store different types of data as a binary pattern, a sequence of 0s and 1s. In Chapter 4, we show how data is manipulated, as a binary pattern, inside a computer.

Although data should be stored in only one form inside a computer, a binary pattern, data outside a computer can take many forms. In addition, computers (and the notion of data processing) have created a new field of study known as data organization, which asks the question: can we organize our data into different entities and formats before
storing it inside a computer? Today, data is not treated as a flat sequence of information. Instead, data is organized into small units, small units are organized into larger units, and so on. We will look at data from this point of view in Chapters 11–14.

**Computer software**

Computer software is a general term used to describe a collection of computer programs, procedures and documentation that perform some tasks on a computer system. The term includes application software such as word processors which perform productive tasks for users, system software such as operating systems, which interface with hardware to provide the necessary services for application software, and middleware which controls and co-ordinates distributed systems. Software includes websites, programs, video games etc. that are coded by programming languages like C, C++, etc.

### 2.1.4 History

机械计算机器（1930 年以前）

1645 年，法国 Pascal 发明了齿轮式加减法器
1673 年，德国数学家 Leibniz 发明了乘除器

第一台现代意义的计算机

1821 年，英国数学家 C. Babbage 设计了差分机，这是第一台可自动进行数学变换的机器，由于条件所限没有实现。他被誉为“计算机之父”。
图 2-3 国数学家 C. Babbage 设计了差分机

电子计算机的诞生（1930 - 1950）

1945 年，ENIAC（Electronic Numerical Integrator and Computer）在宾夕法尼亚大学诞生。ENIAC 用了近 18000 个真空管，重达 30 吨，耗电 150 千瓦，长 30 米，宽 1 米，高 2.4 米，每秒 5000 次加法运算。

图 2-4 ENIAC
迈向现代计算机

Alan Turing (1912-1954) 1936 年上研究生时提出了图灵机 (Turing Machine)，奠定了计算机的理论基础。

ACM Turing Award: the “Nobel Prize of computing”

John von Neumann (1903-1957) 1946 年发表了一篇关于如何用数字来表示逻辑操作的论文， von Neumann 体系结构为现代计算机普遍采用。

计算机的诞生（1950 - 现在）

基于 von Neumann model，改进主要体现在硬件或软件方面（而不是模型），如表 2-1 所示。第一代，真空管计算机，始于 20 世纪 40 年代末。第二代，晶体管计算机，始于 20 世纪 50 年代末。第三代，集成电路计算机，始于 20 世纪 60 年代中期。第四代，微处理器计算机，始于 20 世纪 70 年代早期。

2008 年我国首款超百万亿次超级计算机曙光 5000A 在天津高新区曙光产业基地正式下线。成为继美国之后第二个能自主研制超百万亿次高性能计算机的国家。它的运算速度超过每秒 160 万亿次，内存超过 100TB，存储容量超过 700TB。

性能：峰值运算速度达到每秒 230 万亿次浮点运算 (230TFLOPS); 单机柜性能 7.5 万亿次，单机柜耗电 20KW，百万亿次计算仅需要约 14 个机柜，占地约 15 平方米。

表 2-1 Modern von Neumann machine

<table>
<thead>
<tr>
<th>代号</th>
<th>起止年代</th>
<th>主要元件</th>
<th>主要元件图例</th>
<th>速度（次/秒）</th>
<th>特点与应用领域</th>
</tr>
</thead>
<tbody>
<tr>
<td>一</td>
<td>40 年代末至 50 年代末</td>
<td>电子管</td>
<td>![电子管图片]</td>
<td>5 千-1 万次</td>
<td>计算机发展的初期阶段，体积巨大，运算速度较低，耗电量大，存储容量小，主要用于进行科学计算。</td>
</tr>
<tr>
<td>二</td>
<td>50 年代末至 60 年代末</td>
<td>晶体管</td>
<td>![晶体管图片]</td>
<td>几万-几十万次</td>
<td>体积减少，耗电减少，运算速度较快，价格下降，不仅用于科学计算，还用于数据处理和事务管理，逐渐用于工业控制。</td>
</tr>
<tr>
<td>三</td>
<td>60 年代中期开始</td>
<td>中、小规模集成电路</td>
<td>![中、小规模集成电路图片]</td>
<td>几十万-几百次</td>
<td>体积、功耗进一步减少，可靠性及速度进一步提高，应用领域进一步拓展到文字处理、企业管理、自动控制、城市交通管理等方面。</td>
</tr>
<tr>
<td>四</td>
<td>70 年代初开始</td>
<td>大规模超大规模集成电路</td>
<td>![大规模超大规模集成电路图片]</td>
<td>几千万-几千次</td>
<td>性能大幅度提高，价格大幅度下降，广泛应用于社会生活的各个领域，进入办公室和家庭。在办公室自动化、电子编译程序、数据库管理、图像识别、语音识别、专家系统等领域中大显身手。</td>
</tr>
</tbody>
</table>
威力：可在 30 秒内完成上海证券所 10 年的 1000 多支股票交易信息的 200 种证券指数的计算。可在 3 分钟内，可以同时完成 4 次 36 小时的中国周边、北方大部、北京周边、北京市的 2008 年奥运会需要的气象预报计算，包括风向、风速、温度、湿度等，精度 1 公里，即精确到每个奥运会场馆。

图 2-5 曙光 5000

2.1.5 Practice set

Multi-Choice Questions

12. 现在的计算机是基于 ____ 模型

13. 在冯·诺伊曼模型中， ____ 子系统存储数据和程序
   a. ALU b. 输入/输出 c. 存储器 d. 控制单元

14. 在冯·诺伊曼模型中， ____ 子系统执行计算和逻辑运算
   a. ALU b. 输入/输出 c. 存储器 d. 控制单元

15. 在冯·诺伊曼模型中， ____ 子系统接收数据和程序并将处理结果传给输出设备
   a. ALU b. 输入/输出 c. 存储器 d. 控制单元
16. 在冯·诺伊曼模型中，____子系统是其他子系统的管理者
   a. ALU b. 输入/输出 c. 存储器 d. 控制单元

17. 根据冯·诺伊曼模型，____被存在存储器中
   a. 只有数据 b. 只有程序 c. 数据和程序 d. 以上都不是

18. 问题的逐步解决方案被称为____
   a. 硬件 b. 操作系统 c. 计算机语言 d. 算法

19. FORTRAN 和 COBOL 是____的例子
   a. 硬件 b. 操作系统 c. 计算机语言 d. 算法

20. 在 17 世纪能执行加法和减法的计算机是____

21. 在计算机语言中，____是告诉计算机怎么处理数据的一系列指令
    a. 操作系统 b. 算法 c. 数据处理器 d. 程序

22. ____是以结构化的形式来设计和编写程序
    a. 软件工程 b. 硬件工程 c. 算法开发 d. 教育体系

23. 第一台特殊用途的电子计算机被称为____
    a. Pascal b. Pascaline c. ABC d. EDVAC

24. 第一台基于冯·诺伊曼模型的计算机有一个被称为____
    a. Pascal b. Pascaline c. ABC d. EDVAC

25. 第一台使用存储和编程概念的计算机器被称为____

26. ____将程序设计任务从计算机操作任务中分离出来
    a. 算法 b. 数据处理器 c. 高级程序设计语言 d. 操作系统

### 2.2 计算机系统漫游

本节内容取自下面这本书的 A Tour of Computer Systems 章。等号线之间内容是我加的。

Computer Systems: A Programmer's Perspective (CS:APP)
A computer system consists of hardware and systems software that work together to run application programs. Specific implementations of systems change over time, but the underlying concepts do not. All computer systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to get better at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare “power programmer,” enlightened by an understanding of the underlying computer system and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical errors caused by the way that computers represent numbers. You will learn how to optimize your C code by using clever tricks that exploit the designs of modern processors and memory systems. You will learn how the compiler implements procedure calls and how to use this knowledge to avoid the security holes from buffer overflow bugs that plague network and Internet software. You will learn how to recognize and avoid the nasty errors during linking that confound the
average programmer. You will learn how to write your own Unix shell, your own
dynamic storage allocation package, and even your own Web server!

In their classic text on the C programming language [40], Kernighan and
Ritchie introduce readers to C using the hello program shown in Figure 1.1.
Although hello is a very simple program, every major part of the system must
work in
code/intro/hello.c
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("hello, world\n");
6 }

code/intro/hello.c

Figure 1.1: The hello program.

concert in order for it to run to completion. In a sense, the goal of this book is to
help you understand what happens and why, when you run hello on your system.

We begin our study of systems by tracing the lifetime of the hello program,
from the time it is created by a programmer, until it runs on a system, prints its
simple message, and terminates. As we follow the lifetime of the program, we will
briefly introduce the key concepts, terminology, and components that come into
play. Later chapters will expand on these ideas.

2.1.1 Information is Bits + Context

Our hello program begins life as a source program (or source file) that the
programmer creates with an editor and saves in a text file called hello.c. The
source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit
chunks called bytes. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that
represents each character with a unique byte-sized integer value. For example,
Figure 1.2 shows the ASCII representation of the hello.c program.

#include <stdio.h>
The hello.c program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character ‘#’. The second byte has the integer value 105, which corresponds to the character ‘i’, and so on. Notice that each text line is terminated by the invisible newline character ‘\n’, which is represented by the integer value 10. Files such as hello.c that consist exclusively of ASCII characters are known as text files. All other files are known as binary files.

The representation of hello.c illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network— is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

Aside: The C programming language.

C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the C standard library. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as “K&R” [40]. In Ritchie’s words [64], C is “quirky, flawed, and an enormous success.” So why the success?

- C was closely tied with the Unix operating system. C was developed from the
beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.

- **C is a small, simple language.** The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.

- **C was designed for a practical purpose.** C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed base of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs.

End Aside.

### 2.1.2 Programs Are Translated by Other Programs into Different Forms

The hello program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run hello.c on the system, the individual C statements must be translated by other programs into a sequence of low-level machine-language instructions. These instructions are then packaged in a form called an executable object program and stored as a binary disk file. Object programs are also referred to as executable object files.

On a Unix system, the translation from source file to object file is performed by a compiler driver:

```
unix> gcc -o hello hello.c
```
Here, the GCC compiler driver reads the source file hello.c and translates it into an executable object file hello. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (preprocessor, compiler, assembler, and linker) are known collectively as the compilation system.

**Figure 1.3: The compilation system.**

- **Preprocessing phase.** The preprocessor (cpp) modifies the original C program according to directives that begin with the # character. For example, the `#include <stdio.h>` command in line 1 of hello.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text. The result is another C program, typically with the .i suffix.

- **Compilation phase.** The compiler (cc1) translates the text file hello.i into the text file hello.s, which contains an assembly-language program. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- **Assembly phase.** Next, the assembler (as) translates hello.s into machine-language instructions, packages them in a form known as a relocatable object program, and stores the result in the object file hello.o. The hello.o file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view hello.o with a text editor, it would appear to be gibberish.

- **Linking phase.** Notice that our hello program calls the printf function, which is part of the standard C library provided by every C compiler. The
The printf function resides in a separate precompiled object file called printf.o, which must somehow be merged with our hello.o program. The linker (ld) handles this merging. The result is the hello file, which is an executable object file (or simply executable) that is ready to be loaded into memory and executed by the system.

Aside: The GNU project.

GCC is one of many useful tools developed by the GNU (short for GNU’s Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open-source movement (commonly associated with Linux) owes its intellectual origins to the GNU project’s notion of free software (“free” as in “free speech” not “free beer”). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel.

End Aside.

2.1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as hello.c, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- **Optimizing program performance.** Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a switch statement always more efficient than a sequence of if-then-else statements? Just how expensive is a function call? Is a while loop more
efficient than a do loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times?

In Chapter 3, we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.

- **Understanding link-time errors.** In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in Chapter 7.

- **Avoiding security holes.** For many years now, buffer overflow bugs have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 as part of our study of assembly language.
2.1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our hello.c source program has been translated by the compilation system into an executable object file called hello that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a shell:

unix> ./hello
hello, world
unix>

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the hello program and then waits for it to terminate. The hello program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

2.1.4.1 Hardware Organization of a System

To understand what happens to our hello program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel. Don’t worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Buses

Running throughout the system is a collection of electrical conduits called buses that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as words. The number of bytes in a word (the word size) is a fundamental system parameter that varies across systems. For example, Intel Pentium systems have a word size of 4 bytes, while serverclass systems such as Intel Itaniums and high-end Sun SPARCS
have word sizes of 8 bytes. Smaller systems that are used as embedded controllers in automobiles and factories can have word sizes of 1 or 2 bytes. For simplicity, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

Figure 1.4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

I/O Devices

Input/output (I/O) devices are the system’s connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system’s main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.
Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 11, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

**Main Memory**

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of Dynamic Random Access Memory (DRAM) chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an Intel machine running Linux, data of type *short* requires two bytes, types *int*, *float*, and *long* four bytes, and type *double* eight bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

**Processor**

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.\(^1\)

From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task, over and over again: It reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple operation dictated by the instruction, and then updates the PC to point to the next instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register file is a

---

\(^1\) PC is also a commonly used acronym for “personal computer”. However, the distinction between the two should be clear from the context.
small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- **Load:** Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- **Store:** Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- **Update:** Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register, overwriting the previous contents of that register.
- **I/O Read:** Copy a byte or a word from an I/O device into a register.
- **I/O Write:** Copy a byte or a word from a register to an I/O device.
- **Jump:** Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

Chapter 4 has much more to say about how processors work.

2.1.4.2 Running the **hello** Program

Given this simple view of a system’s hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters “./hello” at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable **hello** file by executing a sequence of instructions that copies the code and data in the **hello** object file from disk to main memory. The data include the string of characters “hello, world
” that will eventually be printed out.

Using a technique known as direct memory access (DMA, discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the **hello** object file are loaded into memory, the processor begins executing the machine-language instructions in the **hello**
program’s main routine. These instruction copy the bytes in the “hello, world\n” string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

Figure 1.5: Reading the hello command from the keyboard.
Figure 1.6: **Loading the executable from disk into main memory.**

Figure 1.7: **Writing the output string from memory to the display.**
2.1.5 Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. The machine instructions in the hello program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string “hello,world\n”, originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer’s perspective, much of this copying is overhead that slows down the “real work” of the program. Thus, a major goal for system designers is make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to millions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this processor-memory gap continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller faster storage devices called cache memories (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 1.8 shows the cache memories in a typical system. An L1 cache on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger L2 cache with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as Static Random Access Memory (SRAM).
One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in Chapter 6.

### 2.1.6 Storage Devices Form a Hierarchy

This notion of inserting a smaller, faster storage device (e.g., cache memory) between the processor and a larger slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in every computer system are organized as a *memory hierarchy* similar to Figure 1.9. As we move from the top of the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file occupies the top level in the hierarchy, which is known as level 0 or L0. The L1 cache occupies level 1 (hence the term L1). The L2 cache occupies level 2. Main memory occupies level 3, and so on.

The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache, which is a cache for the L2 cache, which is a cache for the main memory, which is a cache for the disk. On some networked systems with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the L1 and L2 caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. Chapter 6 will have much more to say about this.
2.1.7 The Operating System Manages the Hardware

Back to our hello example. When the shell loaded and ran the hello program, and when the hello program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the operating system. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

![Figure 1.9: An example of a memory hierarchy.](image)

![Figure 1.10: Layered view of a computer system.](image)
The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications, and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: processes, virtual memory, and files. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

*Figure 1.11: Abstractions provided by an operating system.*

**Aside: Unix and Posix.**
The 1960s was an era of huge, complex operating systems, such as IBM’s OS/360 and Honeywell’s Multics systems. While OS/360 was one of the most successful software projects in history, Multics dragged on for years and never achieved wide-scale use. Bell Laboratories was an original partner in the Multics project, but dropped out in 1969 because of concern over the complexity of the project and the lack of progress. In reaction to their unpleasant Multics experience, a group of Bell Labs researchers — Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna — began work in 1969 on a simpler operating system for a DEC PDP-7 computer, written entirely in machine language. Many of the ideas in the new system, such as the hierarchical file system and the notion of a shell as a user-level process, were borrowed from Multics but implemented in a smaller, simpler package. In 1970, Brian Kernighan dubbed the new system “Unix” as a pun on the complexity of “Multics.” The kernel was rewritten in C in 1973, and Unix was announced to the outside world in 1974 [65].

Because Bell Labs made the source code available to schools with generous terms, Unix developed a large following at universities. The most influential work was done at the University of California at Berkeley in the late 1970s and early 1980s, with Berkeley researchers adding virtual...
memory and the Internet protocols in a series of releases called Unix 4.xBSD (Berkeley Software Distribution). Concurrently, Bell Labs was releasing their own versions, which become known as System V Unix. Versions from other vendors, such as the Sun Microsystems Solaris system, were derived from these original BSD and System V versions. Trouble arose in the mid 1980s as Unix vendors tried to differentiate themselves by adding new and often incompatible features. To combat this trend, IEEE (Institute for Electrical and Electronics Engineers) sponsored an effort to standardize Unix, later dubbed “Posix” by Richard Stallman. The result was a family of standards, known as the Posix standards, that cover such issues as the C language interface for Unix system calls, shell programs and utilities, threads, and network programming. As more systems comply more fully with the Posix standards, the differences between Unix versions are gradually disappearing.

End Aside.

2.1.7.1 Processes

When a program such as hello runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of the processor, main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system’s memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A process is the operating system’s abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By concurrently, we mean that the instructions of one process are interleaved with the instructions of another process. The operating system performs this interleaving with a mechanism known as context switching.

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the context, includes information such as the current values of the PC, the register file, and the contents of main memory. At any point in time, exactly one process is running on the system. When the operating system decides to transfer control from the current process to a new process, it performs a context switch by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example hello scenario.
There are two concurrent processes in our example scenario: the shell process and the hello process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the hello program, the shell carries out our request by invoking a special function known as a system call that passes control to the operating system. The operating system saves the shell’s context, creates a new hello process and its context, and then passes control to the new hello process. After hello terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command line input.

![Process context switching](image)

Figure 1.12: Process context switching.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in Chapter 8.

One of the implications of the process abstraction is that by interleaving different processes, it distorts the notion of time, making it difficult for programmers to obtain accurate and repeatable measurements of running time. Chapter 9 discusses the various notions of time in a modern system and describes techniques for obtaining accurate measurements.

### 2.1.7.2 Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called threads, each running in the context of the process and sharing the same code and
global data. Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. You will learn the basic concepts of concurrency, including threading, in Chapter 13.

2.1.7.3 Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its virtual address space. The virtual address space for Linux processes is shown in Figure 1.13. (Other Unix systems use a similar layout.) In Linux, the topmost one-fourth of the address space is reserved for code and data in the operating system that is common to all processes. The bottommost three-quarters of the address space holds the code and data defined by the user’s process. Note that addresses in the figure increase from bottom to the top.

![Figure 1.13: Process virtual address space.](image-url)
The virtual address space seen by each process consists of a number of well-defined areas, each with a specific purpose. You will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- **Program code and data.** Code begins at the same fixed address, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file, in our case the `hello` executable. You will learn more about this part of the address space when we study linking and loading in Chapter 7.

- **Heap.** The code and data areas are followed immediately by the run-time heap. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 10.

- **Shared libraries.** Near the middle of the address space is an area that holds the code and data for shared libraries such as the C standard library and the math library. The notion of a shared library is a powerful, but somewhat difficult concept. You will learn how they work when we study dynamic linking in Chapter 7.

- **Stack.** At the top of the user’s virtual address space is the user stack that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in Chapter 3.

- **Kernel virtual memory.** The kernel is the part of the operating system that is always resident in memory. The top 1/4 of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents of a process’s virtual memory on disk, and then use the main memory as a cache.
for the disk. Chapter 10 explains how this works and why it is so important to the operation of modern systems.

2.1.7.4 Files
A file is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file. All input and output in the system is performed by reading and writing files, using a small set of system calls known as Unix I/O.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all of the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies. You will learn about Unix I/O in Chapter 11.

Aside: The Linux project.
In August, 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT
Hello everybody out there using minix -
I’m doing a (free) operating system (just a hobby, won’t be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I’d like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I’ve currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I’ll get something practical within a few months, and I’d like to know what features most people would want. Any suggestions are welcome, but I won’t promise I’ll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
The rest, as they say, is history. Linux has evolved into a technical and cultural phenomenon. By combining forces with the GNU project, the Linux project has developed a complete, osix-compliant version of the Unix operating system, including the kernel and all of the supporting infrastructure. Linux is available on a wide array of computers, from hand-held devices to mainframe computers. A group at IBM has even ported Linux to a wristwatch!

End Aside.

2.1.8 Systems Communicate With Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of view of an individual system, the network can be viewed as just another I/O device, as shown in Figure 1.14. When the system copies a sequence of bytes from main memory to the network adapter, the data flows across the network to another machine, instead of, say, to a local disk drive. Similarly, the system can read data sent from other machines and copy this data to its main memory.
Figure 1.14: A network is another I/O device.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

Returning to our hello example, we could use the familiar telnet application to run hello on a remote machine. Suppose we use a telnet client running on our local machine to connect to a telnet server on a remote machine. After we log in to the remote machine and run a shell, the remote shell is waiting to receive an input command. From this point, running the hello program remotely involves the five basic steps shown in Figure 1.15.

After we type the "hello" string to the telnet client and hit the enter key, the client sends the string to the telnet server. After the telnet server receives the string from the network, it passes it along to the remote shell program. Next, the remote shell runs the hello program, and passes the output line back to the telnet
server. Finally, the telnet server forwards the output string across the network to the telnet client, which prints the output string on our local terminal.

This type of exchange between clients and servers is typical of all network applications. In Chapter 12 you will learn how to build network applications, and apply this knowledge to build a simple Web server.

![Figure 1.15: Using telnet to run hello remotely over a network.](image)

### 2.1.9 The Next Step

This concludes our initial whirlwind tour of systems. An important idea to take away from this discussion is that a system is more than just hardware. It is a collection of intertwined hardware and systems software that must cooperate in order to achieve the ultimate goal of running application programs. The rest of this book will expand on this theme.

### 2.1.10 Summary

A computer system consists of hardware and systems software that cooperate to run application programs. Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context. Programs are translated by other programs into different forms, beginning as ASCII text and then translated by compilers and linkers into binary executable files.

Processors read and interpret binary instructions that are stored in main memory. Since computers spend most of their time copying data between memory, I/O devices, and the CPU registers, the storage devices in a system are arranged in a hierarchy, with the CPU registers at the top, followed by multiple levels of
hardware cache memories, DRAM main memory, and disk storage. Storage devices that are higher in the hierarchy are faster and more costly per bit than those lower in the hierarchy. Storage devices that are higher in the hierarchy serve as caches for devices that are lower in the hierarchy. Programmers can optimize the performance of their C programs by understanding and exploiting the memory hierarchy.

The operating system kernel serves an intermediary between the application and the hardware. It provides three fundamental abstractions: (1) Files are abstractions for I/O devices. (2) Virtual memory is an abstraction for both main memory and disks. (3) Processes are abstractions for the processor, main memory, and I/O devices.

Finally, networks provide ways for computer systems to communicate with one another. From the viewpoint of a particular system, the network is just another I/O device.

**Bibliographic Notes**

Ritchie has written interesting first hand accounts of the early days of C and Unix [63, 64]. Ritchie and Thompson presented the first published account of Unix [65]. Silberschatz and Gavin [70] provide a comprehensive history of the different flavors of Unix. The GNU(www.gnu.org) and Linux (www.linux.org) Web pages have loads of current and historical information. Unfortunately, the Posix standards are not available online. They must be ordered for a fee from IEEE (standards.ieee.org).

**Bibliography**


第3章 数据和数的表示


3.1 数据的表示

计算机就是数据处理器。但是在讨论数据处理之前，首先我们应该知道数据的本质。我们将讨论不同类型的数据，以及它们是如何在计算机中表示的。

3.1.1 数据的类型

今天，数据以不同的形式出现，如：数字、文字、图像、音频和视频（图 3-1）。人们需要处理所有这些数据类型。

- 工程程序使用计算机的主要目的是处理数字：进行算术运算、解代数或三角方程、找出微分方程的根等等。
- 与工程程序不同的是，文字处理程序使用计算机的主要目的是处理文本：调整对齐、移动、删除等等。
- 程序处理程序使用计算机则主要用来处理图像：创建、缩小、放大、旋转等等。
- 计算机也处理音频数据。可以使用计算机播放音乐，并且可以把声音作为数据输入到计算机中。
- 最后，计算机不仅能用来播放电影，还能创建我们在电影中所看见的特技效
The computer industry uses the term multimedia to define information that contains numbers, text, images, audio, and video.

### 3.1.2 计算机内部的数据

现在的问题：怎样处理所有这些数据类型？是否用不同的计算机处理不同类型的数？也就是说，是否有专门一类计算机只处理数字，或者是否有专门一类计算机只处理文本？

这种用不同计算机处理不同数据类型的方法既不经济也不切合实际，因为数据往往是多种类型的混合。例如，尽管银行主要是处理数字，但它也需要以文本形式存储客户名字。再如，图像通常是图形和文本的混合。

最有效的解决方法就是采用统一的数据表示法。所有计算机外的数据类型都采用统一的数据表示法，经过转换后存入计算机，当数据从计算机输出时再还原回来。这种通用的格式被称为位模式。

#### 3.1.2.1 位

在进一步讨论为模式之前，首先必须给出位的定义。位（bit, binary digit 的缩写，二进制数字）是存储在计算机中的最小数据单位：它是 0 或 1。位代表设备的某一个状态，这些设备只能处于两种状态中的某一种状态。例如，开关要么合上要么断开。惯例是用 1 表示合上状态，用 0 表示断开状态。电子开关能表示位，换句话说会所，开关能存储一个位的信息。现在计算机使用各种各样的两态设备来存储数据。

#### 3.1.2.2 位模式

单个的位并不能解决数据表示问题。如果每一个数据块都能被表示成 0 或 1，那么紧紧需要单个位就行了。然而，我们需要存储更大的数字、文本、图形等等。

为了表示数据的不同类型，应该使用位模式，它是一个序列，有时也被称为位流。图 3-2 展示了由 16 个位组成的位模式。它是 0 和 1 的组合。这就意味着，如果要存储一个由 16 个位组成的位模式，那么就需要 16 个电子开关。如果要存储 1000 个位模式，每个 16 位，那么就需要 16000 个开关。
现在问题是：计算机存储器是怎么知道它所存储的位模式表示哪种类型的数
据。实际上它并不知道。计算机存储器仅仅将数据以位模式存储。至于解释位模
式是数字类型、文本类型或其他的数据类型则是由输入/输出设备或程序来完成。
换句话说，当数据输入计算机时，它们被编码，当呈现给用户时，它们被解码（图
3-3）

图 3-2 位模式

图 3-3 位模式的例子

3.1.2.3 字节

通常长度为 8 的位模式被称为字节。这个术语同样被用于测量内存或其他存
储设备的大小。例如，一台能存储 800 万位信息的计算机我们称其有 1 兆自己的
内存。

3.1.3 表示数据

现在我们能解释为什么能用位模式表示不同类型的数
据。

3.1.3.1 文本

在任何语言中，文本的片断是用来表示语言中某个意思的一系列的符号。例
如，在英语中使用 26 个符号（A, B, C, ..., Z）来表示大写字母，26 个符号（a, b,
第3章 数据和数的表示

c, ..., z）表示小写字母，9个符号（0, 1, 2, ..., 9）来表示数字字符（不是数字；后面将看到它们的不同之处），以及符号（, ?, :, ..., !）来表示标点。另外一些符号如空格、换行和制表符被用于文本的对齐和可读性。

位模式可以表示任何一个符号。换句话说，例如由4个符号组成的“BYTE”文本可采用4个位模式表示，每一个模式定义一个符号（图3-4）。

现在问题是：在一种语言中，位模式到底需要多少位来表示一个符号？这取决于该语言集中到底有多少不同的符号。例如，如果要创建某个虚构的语言，它仅仅使用大写的英文字母，则只需要26个符号。相应的这种语言的位模式则至少需要表示26个符号。对另一种语言，如中文，可能需要更多的符号。在一种语言中，表示符号的位模式的长度取决于该语言中所使用的符号的数量。更多的符号意味着更长的位模式。

尽管位模式的长度取决于符号的数量，但是它们的关系并不是线性的；而是对数关系。如果需要2个符号，位模式长度是1位（log2 2 = 1），如果需要4个符号，长度是2位（log2 4 = 2）。从表3-1中可以很容易地看出它们之间的关系。2位的位模式能表示四种不同的形式：00, 01, 10 和 11。这些形式中的任何一种都可用来代表一个字符。同样，3位的位模式有八种不同的形式：000, 001, 010, 011, 100, 101, 110 和 111。

<table>
<thead>
<tr>
<th>符号数目</th>
<th>位模式的长度</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
</tr>
</tbody>
</table>
代码

不同的位模式集合被设计用于表示文本符号。每一个集合被称为代码。表示符号的过程被称为编码。下面将介绍常用代码。

ASCII 美国国家标准协会（ANSI）开发了一个被称为美国信息交换信息标准码（ASCII）的代码。此代码使用 7 位表示每个符号。即此代码可以定义 128（2^7）种不同的符号。用于表示 ASCII 的完整位模式如表 3-2 所示。图 3-5 展示了 ASCII 码中 “BYTE” 是如何表示的。

### 表 3-2 ASCII Table

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Octal</th>
<th>Char</th>
<th>ASCII</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>020</td>
<td>00</td>
<td>null</td>
<td>32</td>
<td>20 0 020</td>
</tr>
<tr>
<td>1</td>
<td>021</td>
<td>021</td>
<td>start of heading</td>
<td>33</td>
<td>21 0 021</td>
</tr>
<tr>
<td>2</td>
<td>022</td>
<td>022</td>
<td>start of text</td>
<td>34</td>
<td>22 0 022</td>
</tr>
<tr>
<td>3</td>
<td>023</td>
<td>023</td>
<td>end of text</td>
<td>35</td>
<td>23 0 023</td>
</tr>
<tr>
<td>4</td>
<td>024</td>
<td>024</td>
<td>end of transmission</td>
<td>36</td>
<td>24 0 024</td>
</tr>
<tr>
<td>5</td>
<td>025</td>
<td>025</td>
<td>escape</td>
<td>37</td>
<td>25 0 025</td>
</tr>
<tr>
<td>6</td>
<td>026</td>
<td>026</td>
<td>acknowledge</td>
<td>38</td>
<td>26 0 026</td>
</tr>
<tr>
<td>7</td>
<td>027</td>
<td>027</td>
<td>bell</td>
<td>39</td>
<td>27 0 027</td>
</tr>
<tr>
<td>8</td>
<td>028</td>
<td>028</td>
<td>backspace</td>
<td>40</td>
<td>28 0 028</td>
</tr>
<tr>
<td>9</td>
<td>029</td>
<td>029</td>
<td>horizontal tab</td>
<td>41</td>
<td>29 0 029</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>010</td>
<td>LF line feed, new line</td>
<td>42</td>
<td>2A 0 010</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>011</td>
<td>VT vertical tab</td>
<td>43</td>
<td>2B 0 011</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>012</td>
<td>FF form feed, new page</td>
<td>44</td>
<td>2C 0 012</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>013</td>
<td>carriage return</td>
<td>45</td>
<td>2D 0 013</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>014</td>
<td>shift out</td>
<td>46</td>
<td>2E 0 014</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>015</td>
<td>shift in</td>
<td>47</td>
<td>2F 0 015</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>000</td>
<td>data link escape</td>
<td>50</td>
<td>30 0 016</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>001</td>
<td>device control 1</td>
<td>51</td>
<td>31 0 017</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>002</td>
<td>device control 2</td>
<td>52</td>
<td>32 0 018</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>003</td>
<td>device control 3</td>
<td>53</td>
<td>33 0 019</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>004</td>
<td>device control 4</td>
<td>54</td>
<td>34 0 020</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>005</td>
<td>negative acknowledgment</td>
<td>55</td>
<td>35 0 021</td>
</tr>
<tr>
<td>22</td>
<td>6</td>
<td>006</td>
<td>synchronous idle</td>
<td>56</td>
<td>36 0 022</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>007</td>
<td>end of media</td>
<td>57</td>
<td>37 0 023</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>000</td>
<td>cancel</td>
<td>58</td>
<td>38 0 024</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>011</td>
<td>end of record</td>
<td>59</td>
<td>39 0 025</td>
</tr>
<tr>
<td>26</td>
<td>A</td>
<td>012</td>
<td>sub</td>
<td>60</td>
<td>3A 0 026</td>
</tr>
<tr>
<td>27</td>
<td>B</td>
<td>013</td>
<td>escape</td>
<td>61</td>
<td>3B 0 027</td>
</tr>
<tr>
<td>28</td>
<td>C</td>
<td>014</td>
<td>file separator</td>
<td>62</td>
<td>3C 0 028</td>
</tr>
<tr>
<td>29</td>
<td>D</td>
<td>015</td>
<td>group separator</td>
<td>63</td>
<td>3D 0 029</td>
</tr>
<tr>
<td>30</td>
<td>E</td>
<td>016</td>
<td>record separator</td>
<td>64</td>
<td>3E 0 030</td>
</tr>
<tr>
<td>31</td>
<td>F</td>
<td>017</td>
<td>unit separator</td>
<td>65</td>
<td>3F 0 031</td>
</tr>
</tbody>
</table>

Source: [Leslie Tables.com](http://www.LeslieTables.com)
usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'.

<table>
<thead>
<tr>
<th>B</th>
<th>Y</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000010</td>
<td>1011001</td>
<td>1010100</td>
<td>1000101</td>
</tr>
</tbody>
</table>

图 3-5 在 ASCII 码中如何表示“BYTE”单词

以下列举了这个代码的一些突出特点:

- ASCII 码使用 7 位模式，范围从 0000000 到 1111111。
- 第一个模式（0000000）表示空字符（没有字符）。
- 最后一个模式（1111111）表示删除字符。
- 有 31 种控制（不可打印的）字符。
- 数字符（0 到 9）编码在字母字符之前。
- 有一些专用的打印字符。
- 大写字母（A…Z）编码在小写字母（a…z）之前。
- 大小写字符仅用 1 位来区分。例如，A 的模式是 1000001，a 的模式是 1100001。唯一的不同是从右数第 6 个位上。
- 在大小写字母之间，有 6 种特殊的字符。

扩展 ASCII 码 为了使每个位模式大小统一为 1 字节（8 位），ASCII 位模式通过在左边增加额外的 0 来进行扩充。现在每一个模式都能很容易地恰好存入 1 字节大小的内存中。换句话说，在扩展 ASCII 码中，第一个模式是 00000000，最后一个模式是 01111111。

一些制造商曾经决定使用额外的位创建额外附加的 128 个符号。然而，由于各个制造商最终未能制定出一个统一标准的代码集，这种尝试并未获得成功。

EBCDIC 在计算机早期时代，IBM 公司开发了一种名为扩充的二进制编码的十进制交换码（EBCDIC）。这种代码使用 8 位模式，因此它能最多表示 256 个符号。然而，这种代码除了在 IBM 大型机外并未在其他计算机中得到普遍的应用。

Unicode 前面所述的两种代码所表示的符号仅仅属于英语。为此，需要更大容量的代码。于是硬件和软件制造商联合起来共同设计了一种名为 Unicode 的代码，这种代码使用 16 位并能表达多达 65536（2^16）个符号。代码的不同部分被分配用于表示世界上不同语言的符号。其中有部分代码被用于表示图形和特殊符
号。Java™语言使用这种代码来表示字符。微软 Windows 使用了前 256 个字符的一个变化版本。Unicode 符号的简写集见附录 B。

ISO 国际标准化组织，通常称为 ISO，设计了一种使用 32 位模式的代码。这种代码能表示 4,294,967,296（2^{32}）个符号，足以表示现今世界上的任何符号。

### 3.1.3.2 数

在计算机中，数是使用二进制系统来表示的。在这种系统中，位模式（一系列的 0 和 1）被用来表示数。然而，像 ASCII 码的代码并没有用来表示数据。其中原因和有关数表示的讨论将在第 3 章阐述。

### 3.1.3.3 图像

图像在计算机中有两种表示方式：位图图像或者矢量图形（图 3-6）。

1. **位图图像** (bitmap graphics).
   在这种方法中，图像被分成像素矩阵，每个像素是一个小点。像素大小取决于分辨率。例如，图像可以分成 1,000 像素或者 10,000 个像素。第二种情况，尽管有更好的图像显示（较高的分辨率），但是需要更多的内存来存储图像。

   在把图像分成像素之后，每一个像素被赋值为位模式。模式的尺寸和值取决于图像。对于仅由黑白组成的图像（例如棋盘），1 位模式已足够表示像素。0 模式表示黑像素，1 模式表示白像素。然后，模式被一个接一个记录并存储在计算机中。图 3-7 显示了这种图像以及它的表示。

   如果一副图像不是由纯黑、纯白像素组成，那么可以增加位模式的长度来表示灰色度。例如，可以使用 2 位模式来显示四重灰度级。黑色像素被表示成 00，深灰色像素被表示成 01，浅灰色像素被表示成 10，白色像素被表示成 11。
如果是表示彩色图像，则每一种彩色像素被分解为三种主色：红、绿、蓝 (RGB)。然后测出每一种颜色的强度，并把位模式（通常是 8 位）分配给它。换句话说，每一个像素有 3 个位模式：一个用于表示红色的强度，一个用于表示绿色的强度，一个用于表示蓝色的强度。例如图 3-8 显示采用 4 个位模式表示彩色图像中的像素。

常见位图图像格式：GIF, JPEG, PNG, TIFF, BMP, and PCX 等等。

2. 矢量图像 (vector graphics)

位图图像的问题是改变图形大小的时会产生失真。而矢量图表示方法并不存储位模式，它是将图像分解成曲线和直线的组合，其中每一曲线或直线由数学公式表示。例如，一条直线可以通过它的端点坐标来作图，圆则可以通过它的圆心坐标和半径长度来作图。这些公式的组合被存储在计算机中。当要显示或打印图像时，将图像的尺寸作为输入传给系统，系统根据新的尺寸重新设计图像并用相同的公式画出图像。在这种情况下，每画一次图像，公式也将重新估算一次。
3.1.3.1 音频

音频表示声音和音乐。虽然没有存储声音和音乐的标准，但是有一种想法即将音频转换成数字数据，并使用位模式存储它们。音频本质上是模拟数据。它是连续的（模拟的），而不是离散的（数字的）。图 3-9 显示了把音频数据转化成位模式的步骤。具体步骤如下：

1) 对模拟信号进行采样。采样就是以相等的间隔来测量信号的值。
2) 量化采样值。量化就是给采样值分配值（从值集中）。例如，如果一采样值为 29.2，而值集为 0 到 63 的整数值，因此量化该采样值即是给采样值赋值 29。
3) 将量化值转化成位模式。例如，把 25 转换为位模式为 00011001。
4) 存储位模式。

3.1.3.1 视频

视频是图像（帧）在时间上的表示。电影就是一系列的帧，一张接一张地播放而形成的运动图像。所以，如果知道如何将图像存储在计算机中，也就知道了如何存储视频；每一副图像或帧转化成一系列位模式并储存。这些图像组合起来就可以表示视频。需要注意现在视频通常是被压缩存储的。第 15 章将讨论 MPEG，这是一种常见的视频压缩技术。
3.1.4 十六进制表示法

当数据存储在计算机中时是用位模式表示的。然而，人们发现处理位模式很困难。写一长串 0 和 1 非常乏味而且容易出错。而采用十六进制可以帮助我们。

十六进制法是以 16（hexadec 在希腊语中表示 16）为基础的。这意味着有 16 个符号（十六进制数字）：0、1、2、3、4、5、6、7、8、9、A、B、C、D、E 和 F。当把位模式转换成十六进制的时候，十六进制的重要性就变得很明显了。

每个十六进制数字能表示成 4 个位，4 个位也能被十六进制数字表示。表 3-3 给出了位模式和十六进制数字间的关系。

<table>
<thead>
<tr>
<th>位模式</th>
<th>十六进制数字</th>
<th>位模式</th>
<th>十六进制数字</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

转换

将位模式转换成十六进制数是通过将模式组成每 4 个一组，找到与每 4 位相对应的十六进制数字。十六进制数转换成位模式即将每个十六进制数字转化为相对应的 4 位模式（图 3-10）。

![图 3-10 二进制到十六进制和十六进制到二进制的转换](image)

十六进制有两种写法，第一种，在数的前面加小写（或大写）的 x 来代表当前表示方法为十六进制。例如，在这种约定表示法中 xA34 表示一个十六进制数。另一种写法，将数字基数（16）作为表示法的下标。例如，在第二种表示法中 A34_{16} 表示与前面同样的值。
3.1.5 八进制表示法

另一种将位模式分组的表示法为八进制法。八进制法以 8（oct 在希腊语中表示 8）为基础。这意味着有 8 个符号（八进制数字）: 0、1、2、3、4、5、6、7。

当你学会如何将位模式转换成八进制数时，八进制法的重要性就变得明显了。每个八进制数字能表示成 3 个位，同样 3 个位也能表示成八进制数字，表 3-4 显示了位模式和八进制数字间的关系。

3 位模式可以通过八进制来表示，反之亦然。

<table>
<thead>
<tr>
<th>位模式</th>
<th>十六进制数字</th>
<th>位模式</th>
<th>十六进制数字</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

转换

将位模式转换成八进制数是通过将模式组织成每 3 个一组，找到与每个 3 位组相对应的八进制值。把八进制数转换成位模式，即将每个八进制数字转换成相对应的 3 位模式（图 3-11）。

八进制也有两种写法。第一种，在数的前面加 0 来代表八进制表示（有时用一小写的 o 表示）。例如，0634 表示一个八进制数。另一种写法，将数字基数（8）作为表示法的下标。例如，634_8 表示与前面同样的值。

3.1.6 Multi-Choice Questions
第 3 章 数据和数的表示

11. 下面哪些属于数据？
   a. 数  b. 视频  c. 音频  d. 以上全是

12. 储存一个字节，需____个电子开关。
   a. 1  b. 2  c. 4  d. 8

13. 一个字节有____位。
   a. 2  b. 4  c. 8  d. 16

14. 在一个有 64 个符号的集合中，每个符号需要用长度为____位的位模式来表示。
   a. 4  b. 5  c. 6  d. 7

15. 10 位的位模式可表示多少符号？
   a. 128  b. 256  c. 512  d. 1024

16. 在扩展 ASCⅡ中，每个符号为____位。
   a. 7  b. 8  c. 9  d. 10

17. 如果 ASCⅡ中 E 的编码是 1000101，则 e 的 ASCⅡ编码是____。
   a. 1000110  b. 1000111  c. 0000110  d. 1100101

18. 扩展 ASCⅡ，就是正常的 ASCⅡ编码位模式将____而得到的。
   a. 0 位加在左边  b. 0 位加在右边  c. 1 位加在左边  d. 1 位加在右边

19. ____是用于 IBM 大型机的编码。
   a. ASCⅡ  b. 扩展 ASCⅡ  c. EBCDIC  d. Unicode

20. ____是 16 位编码，并可表示除了英语外的其他语言的符号。
   a. ASCⅡ  b. 扩展 ASCⅡ  c. EBCDIC  d. Unicode

21. ____是 Java 语言中用于表示字符的编码。
   a. ASCⅡ  b. 扩展 ASCⅡ  c. EBCDIC  d. Unicode

22. 32 位编码是由____开发，能表示所有语言中的符号。
   a. ASCII  b. ISO  c. EBCDIC  d. Hamming

23. 图像在计算机中通过以下____方法表示。
   a. 位图图形  b. 矢量  c. 矩阵图形  d. a 或 b
第3章 数据和数的表示

24. 位图图形和矢量图的表示方法在计算机中是用于____的表示方法。
   a. 音频  b. 视频  c. 图像  d. 数字

25. 在计算机的____图像表示方法中，每个像素由一个或多个位模式表示。
   a. 位图  b. 矢量  c. 量化  d. 二进制

26. 在计算机的____图像表示方法中，是通过将图像分解成由曲线和直线组成。
   a. 位图  b. 矢量  c. 量化  d. 二进制

27. 采用____图形表示方法在计算机中表示图像，重新调节图像会产生波纹状或颗粒状图像。
   a. 位图  b. 矢量  c. 量化  d. 二进制

28. 当相要在计算机上下载音乐时，音频信号必须____。
   a. 采样  b. 量化  c. 编码  d. 上面的全是

--------------------------------------------------------------------------------------------------------

下面例子取自李晓明、张丽的《计算机系统平台》一书的第3章

十进制转换成二进制

我们一般采用下面的竖式来表示上面的过程。

例：求$89_{10}$的二进制表示。

\[
\begin{array}{c|cccccc}
2 & 89 & 1 & p_1 \\
2 & 44 & 0 & p_2 \\
2 & 22 & 0 & p_3 \\
2 & 11 & 1 & p_4 \\
2 & 5 & 1 & p_5 \\
2 & 2 & 0 & p_6 \\
2 & 1 & 1 & p_7 \\
0 & & & & & & \\
\end{array}
\]

则$89_{10} = p_7 p_6 p_5 p_4 p_3 p_2 p_1 = 1011001_2$

小数的转换也有两种办法，一种是减权定位法。与前面整数的情况一样。第二种办法略有不同，小数的转换不再采用除法，而是采用乘法，称为乘基取整法。具体的做法是：乘（以）基取整，先整为高，后整为低。也就是用十进制小数部
分不断地乘以 2，直到乘积的小数部分为 0，然后把每次乘积的整数部分按照先后顺序排列在一起。我们来看一个例子。

例：求 0.8125\text{_{10}} 的二进制表示。

则 0.8125\text{_{10}} = p_1 p_2 p_3 p_4 = 0.1101_2

如果要转换的十进制数既有整数部分又有小数部分，那么要将这两部分分开，分别用除基取余法和乘基取整法求得整数和小数部分，最后再把它们合在一起。

- 八进制转换成二进制

八进制数和十六进制数转换成二进制数则比较简单。因为 2^3=8，所以三位二进制数正好等于一位八进制数。我们来看看八进制的 8 个数码对应的二进制数。

<table>
<thead>
<tr>
<th>八进制数码</th>
<th>二进制数码</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

把八进制数转换成二进制数时，直接把这些数码对换成相应的三位二进制数就可以了。例如，

765_8 = 111 110 101_2
0.765_8 = 0.111 110 101_2

- 十六进制数转换成二进制数

因为 2^4=16，所以一位十六进制数正好等于四位二进制数，它们的关系也是一一对应的。
第 3 章 数据和数的表示

16 个十六进制数码对应的二进制数码如下:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0111</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>

把十六进制数转换成二进制数非常简单，只要把每位上的十六进制数码替换成 4 位二进制数码即可。如:

A13F\textsubscript{16} = 1010 0001 0011 1111\textsubscript{2}
0.A13F\textsubscript{16} = 0.1010 0001 0011 1111\textsubscript{2}

当然和八进制转换成二进制数一样，这需要你熟记这些数码的对应关系。因为十六进制数和二进制数有这样简单的对应关系，而且十六进制数比较简短、比二进制数更容易读写（二进制数那么长的 0/1 串，人看了不晕才怪呢，除非你是计算机），所以，我们平时都用十六进制计数法描述计算机中的二进制数。例如，需要显示计算机中寄存器的内容时，通常都是以十六进制表示。

3.2 数的表示

3.2.1 二进制和十进制转换

3.2.1.1 二进制系统

3.2.2 十进制向二进制转换

3.3 整数的表示法

3.3.1 无符号整数格式

3.3.2 符号加绝对值格式

3.3.3 二进制补码格式

3.3.4 二进制反码格式

3.3.5 整数表示法小结

3.4 EXCESS 系统

3.5 浮点数表示法

3.5.1 转换成二进制; 3.5.2 规范化; 3.5.3 符号、幂和尾数; 3.5.4 IEEE 标准

3.6 十六进制表示法

3.7 关键术语

3.8 小结

3.9 Multi-Choice Questions
20. 在____系统中只使用0和1。
   a. 十进制  b. 八进制  c. 二进制  d. 十六进制

21. 将十进制数转换成二进制数，需要不断用____来除这个数。
   a. 2  b. 8  c. 10  d. 16

22. 以下三种整数表示法中哪种既可以处理正数又可以处理负数？
   a. 符号值表示法  b. 二进制反码表示法  c. 二进制补码表示法  d. 以上都是

23. 在无符号整数中，4位地址分配单元可以表示____个非负数。
   a. 7  b. 8  c. 15  d. 16

24. 在所有的符号整数表示法中，4位地址分配单元可以表示____个非负数。
   a. 7  b. 8  c. 15  d. 16

25. 在____表示法中，内存中存储的1111表示-0。
   a. 无符号整数  b. 符号加绝对值  c. 二进制反码  d. 二进制补码

26. 表示0中存储1111表示-1。
   a. 无符号整数  b. 符号加绝对值  c. 二进制反码  d. 二进制补码

27. 在____表示法中，0有两种表示法。
   a. 符号加绝对值  b. 二进制反码  c. 二进制补码  d. a和b

28. 在____表示法中，0只有一种表示法。
   a. 符号加绝对值  b. 二进制反码  c. 二进制补码  d. a和c

29. 如果最左边一位为0，在____整数表示法中，其表示的十进制数是正的。
   a. 符号加绝对值  b. 二进制反码  c. 二进制补码  d. 以上都是

30. 如果最左边一位为1，在____整数表示法中，其表示的十进制数是正的。
   a. 符号加绝对值  b. 二进制反码  c. 二进制补码  d. 以上都不是

31. 现在的计算机中用于存储数值使用最广泛的表示方法是____。
   a. 符号加绝对值  b. 二进制反码  c. 二进制补码  d. 无符号整数

32. ____表示法经常用于将模拟信号转换为数字信号。
   a. 无符号整数  b. 符号加绝对值  c. 二进制反码  d. b和c
33. 无符号整数可以用于____。
   a. 计数 b. 寻址 c. 信号处理 d. a 和 b

34. ____ 表示法经常用于存储浮点数的指数值。
   a. 无符号整数 b. 二进制反码 c. 二进制补码 d. Excess-X

35. 在 Excess-X 转换中，我们需要将要转换的数____ 幻数 X。
   a. 加 b. 减 c. 乘 d. 除

36. 在 Excess-X 数表示法中，X，N 和位数分配之间的关系是什么？
   a. X=2^N-1 b. X=2^N+1 c. X=2^(N-1) - 1 d. a 或 c

37. 当计算机存储小数（分数）时，____ 通常表示为 2 的幂。
   a. 分子 b. 分母 c. 整数 d. a 或 b

38. 如果分数的分母是 1024，那么分子的长度是____ 位。
   a. 2 b. 8 c. 10 d. 16

39. 如果分数的分子长度为 3 个位，那么分母是____。
   a. 2 b. 8 c. 10 d. 16

40. 5 在 Excess-128 中表示为____。
   a. 00000101 b. 10000100 c. 10000101 d. 10000001

41. 当小数规范化之后，在小数点的左边还有____ 位。
   a. 0 b. 1 c. 随机位顺序 d. a 或 b

42. 将规范的数乘以____，表示将二进制中的小数点移动 e 位。
   a. 2 e b. e/2 c. e^2 d. 2^e

43. 当小数规范化后，计算机存储了____。
   a. 符号 b. 幂 c. 尾数 d. 以上都是

44. 在计算机中分数的精度是由____ 定义。
   a. 符号 b. 幂 c. 尾数 d. 以上都不是

45. 尾数是如何存储在计算机中的？
   a. 以二进制反码 b. 以二进制补码 c. 以无符号整数格式
   d. 以符号加绝对值
46. 八进制数转换成二进制数有____位.
   a.2    b.3    c.4    d.8
第4章 程序设计语言和开发环境

本章的第一节取自 Book1-Ch9。


4.1 程序设计语言

9.1 演化

9.1.1 机器语言 9.1.2 汇编语言 9.1.3 高级语言
9.2 翻译

9.2.1 编译 9.2.2 解释 9.2.3 翻译过程
9.3 编程模式

9.3.1 过程式模式 9.3.2 面向对象模式 9.3.3 函数式模式 9.3.4 说明式模式
9.4 共同概念

9.4.1 标识符 9.4.2 数据类型 9.4.3 变量 9.4.4 字面值 9.4.5 常量 9.4.6 输入和输出 9.4.7 表达式 9.4.8 语句 9.4.9 子程序
9.5 推荐读物
9.6 关键术语 9.7 小结

对程序设计语言作说明，目的是使大家对不同的语言及其使用场合有个初步了解。首先讨论语言的演化。

http://zh.wikipedia.org/wiki/演化

英文中的「evolution」一词，起源于拉丁文的「evolvere」，原本的意思是将一个卷在起的东西打开，也可以指任何事物的生长、变化或发展。包括恒星的演变，化学的演变，文化的演变或者观念的演变。自从 19 世纪以后，演化通常用来指生物学上，不同世代之间外表特征与基因频率的改变。
第 4 章 程序设计语言和开发环境

9.1 计算机语言的演化

机器语言：指令由 0 和 1 串组成，程序可直接执行

符号语言：用符号表示不同的机器语言指令，必须被汇编（assembler）为机器语言

高级语言：从关注计算机转移到关注问题本身。用语句（statement）构造应用逻辑。通常不能直接执行，要编译（compile）或解释（interpret）执行。


Computer Language: The term computer language includes a large variety of artificial languages used to communicate with computers. 1) It is broader than the more commonly-used term programming language. 2) Programming languages are a subset of computer languages. For example, HTML is a markup language and a computer language, but it is not traditionally considered a programming language. 3) Machine code is a computer language. It can technically be used for programming, and has been (e.g. the original bootstrapper for Altair BASIC), though most would not consider it a programming language.

Types of Computer Languages

Programming languages are the primary means by which developers of
computing systems instruct a machine to organize or manipulate information or control physical devices. Most software is written using one or more programming languages. Common examples include C, C++, Java, BASIC, assembly languages, and many others.

Scripting languages are designed to control other software or to coordinate the actions of multiple software applications.

Machine code is a non human-readable binary computer language which corresponds to the available instructions for a microprocessor.

What problem gave birth to programming languages?
Video, "The Machine that Changed the World", PBS

Before high level programming languages existed, computers were programmed one instruction at a time using binary or hex. This was a tedious job and there were a lot of errors. Programs were difficult to read, and modification was extremely difficult because all programs had to be written using absolute addressing. Obviously, this job did not attract many people, so there was a shortage of programmers. Expensive computers sat idle for long periods of time while software was being developed. Software often cost two to four times as much as the computer.

This led to the development of assemblers and assembly languages. Programming became somewhat easier, but many users still wanted floating point numbers and array indexing. Since these capabilities were not supported in hardware, high level languages had to be developed to support them.

What Makes a Language Hot?
It seems that there are many factors in what languages are hot. We can boil them all down to one statement. A language is hot because many programmers are interested in programming in it. From this definition, we can also look at the factors that make a language desirable to program in. Also, the factors that make a language hot may not be the same that keep it hot. For example, both Fortran and COBOL became hot languages for ease of use reasons. They were simply easier to use that their alternatives. However, they stayed hot languages because of experienced programmers and legacy code.
Pascal

Language是由著名瑞士计算机科学家 N. Wirth（1984 年图灵奖得主）设计的一种语言，1968 年提出后被全世界广泛接受，成为一种对计算机科学技术发展有巨大影响的语言。这个语言的名字是为了纪念历史上著名的数学家和计算学科的先驱 Blaise Pascal（帕斯卡）。Pascal 语言把许多好的东西结合在一个很简练的语言里，被计算机教育界广泛采用。从七十年代末往后的很长一段时间里，Pascal 成为世界范围的计算机专业教学语言。

http://en.wikipedia.org/wiki/Niklaus_Wirth

Niklaus Wirth

In 1984 he won the Turing Award for developing a sequence of innovative computer languages. Euler, Algol W, Pascal, Modula, Modula-2 and Oberon. His article Program Development by Stepwise Refinement, about the teaching of programming, is considered to be a classic text in software engineering. In 1975 he wrote the book Algorithms + Data Structures = Programs, which
gained wide recognition and is still useful today.

**Software engineering** is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

![Figure Niklaus Wirth & Ada](http://baike.baidu.com/view/1107373.htm)

http://en.wikipedia.org/wiki/Ada_Lovelace


C

C programming language was devised by Bell Labs of AT & T as a system implementation language for the nascent Unix operating system.

**Contribution: Efficiency, Casting, Flexible array Library, and Good portability**

C 是由美国贝尔实验室的 Dennis Ritchie 在 1972 年设计开发的，开发目的是想成为一种编制“系统程序”的工具语言。Ritchie 等人首先用自己发明的 C 语言编写了 Unix 操作系统。以后 C 语言逐步发展成为开发系统软件的主要语言。C
语言已成为最重要的软件系统开发语言，由此可见 C 语言在计算机领域地位之重要。

SIMULA
Although never widely used, SIMULA is historically important. Developed from 1964-67, SIMULA was designed as a language to aid in creating simulations. SIMULA I was introduced in 1964, its sole purpose was system simulation. Immediately after SIMULA I was finished, the designers decided to work on a new language which would contain many more features so that it could be used for more general purpose areas. The result was SIMLUA 67.

To help make simulation easier, SIMULA 67 introduced the concept of classes, including instancing and coroutine. This was the beginning of data abstraction.

C++
C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. It was intended to deliver that to real projects.

Contribution: One line comment, Inline function, Multiple inheritance, Overloading operators and functions

C++ 是在 C 语言基础上发展出的一种“面向 对象”语言。它是由 Bjarne Stroustrup 在美国贝尔实验室开发的 (1979)。C++ 是 C 语言的一个扩充，修正了 C 的一些弱点和不足，使用起来更方便可靠：以支持“面向对象”(Object-Oriented, 简称为 OO) 的程序设计方法为基本目标，提供了一套支持面向对象程序设计的机制，如“类”(class)、“对象”(object) 等等。面向对象的方法被认为是开发复杂软件系统的一种有效途径，OO 程序设计语言也已经被广泛接受。C++ 是目前使用最广泛的一种面向对象的程序设计语言。
Notations
Programming languages: A notation for describing algorithms (算法, 即步骤) and data structure (数据结构, 即数据或者是信息)
Program: A sentence of a programming language

Ease of use
This seems to be very important in the choice of a new language. Is this language going to be faster and easier to use than other languages. This should be viewed in an historical perspective. Is COBOL easy to use? Was it in 1960? New languages and programming paradigms change what we consider to be easy to use.

Language Features
A language may also be chosen because it has a particular feature. You would write in Java if you wanted to write an application that ran over the Internet. You would not use COBOL if you wanted to write scientific programs.

Performance
In some applications performance is a big issue. This alone will keep Fortran and C alive for a long time to come. Other applications don't need a high performance language to get good responses on some hardware. Why do we need to write interactive software in a very fast language?

Corporate Support
Is there a large corporation or organization that is pushing the language? Would C have become so
popular if Unix had not been written in it? Also, the reappearance of Basic may be solely due to Microsoft's support of it.

**Experienced Programmers**

This is especially important on long projects. Are future programmers going to be able to understand the code and continue development? Also, you are more likely to program an application in a language you know well instead of a language you will need to learn.

**Legacy Code**

The amount of legacy code dictates the need for programmers to understand a language. This in itself can make a language hot. The prime example is COBOL. Long after this language should have faded away, it was still in heavy use. The only reason was the large amount of code written in COBOL that was still necessary to support.

9.8 Multi-Choice Questions

31. 计算机硬件唯一可理解语言。
   a. 机器 b. 符号 c. 高级 d. 自然

32. ____语言又被称为汇编语言。
   a. 机器 b. 符号 c. 高级 d. 自然

33. 挪威语、波斯语、和俄语被归类于 ____语言。
   a. 机器 b. 符号 c. 高级 d. 自然

34. C、C++和 Java 可归类于 ____语言。
   a. 机器 b. 符号 c. 高级 d. 自然

35. 用来编程的软件称为 ____。
   a. 预处理程序 b. 文本编辑器 c. 翻译程序 d. 源文件

36. ____把不同来源的预编译单元汇编到一个可执行程序中。
   a. 预处理程序 b. 文本编辑器 c. 链接器 d. 载入程序

37. 编译器由 ____和 ____组成。
   a. 预处理程序、载入程序 b. 文本编辑器、载入程序
c. 预处理程序、翻译程序 d. 链接器、预处理程序
38. ____是机器语言代码。
   a.翻译单元  b.目标模块  c.源文件  d.子程序

39. 操作系统程序通过调用____来把程序载入内存。
   a.载入程序  b.链接器  c.翻译程序  d.处理器

40. ____语言使用传统方法编程并且被归入强制语言。
   a.过程化  b.函数型  c.说明性  d.面向对象

41. FORTRAN 是一种____语言。
   a.过程化  b.函数型  c.说明性  d.面向对象

42. PASCAL 是一种____语言。
   a.过程化  b.函数型  c.说明性  d.面向对象

43. C++是一种____语言。
   a.过程化  b.函数型  c.说明性  d.面向对象

44. LISP 是一种____语言。
   a.过程化  b.函数型  c.说明性  d.面向对象

45. ____是在商业环境中广泛使用的语言。
   a.FORTRAN b.C++ c.C d.COBOL

46. ____是第一种高级语言，至今仍广泛使用于科学和工程界。
   a.FORTRAN b.C++ c.C d.COBOL

47. ____是一种通过强调结构化编程方法来教初学者编程而设计的语言。
   a.C++ b.C c.Pascal d.Scheme

48. UNIX 操作系统是用____语言编写的。
   a.C++ b.C c.Pascal d.LISP

49. 在 DoD 中流行的一种过程化语言是____。
   a.Ada b.Java c.C++ d.LISP

50. ____是很受欢迎的面向对象语言。
   a.FORTRAN b.COBOL c.C++ d.LISP
51. 在 C++中，____使数据和操作对用户不可见。
   a. 封装 b. 继承 c. 多态 d. 模块化

52. ____程序可以是应用程序或者是 applet。
   a. FORTRAN b. COBOL c. C++ d. Java

53. LISP 和 Scheme 都是____语言。
   a. 过程化 b. 函数型 c. 说明性 d. 面向对象

54. Prolog 是____语言的例子。
   a. 过程化 b. 函数型 c. 说明性 d. 面向对象

55. HTML、PERL 和 SQL 同属于____语言。
   a. 现代 b. 专用 c. 说明性 d. 面向对象

56. C 语言中的标准数据类型是____。
   a. int b. char c. float d. 以上皆是

57. 标准数据类型____可以描述带小数部分的数。
   a. int b. char c. float d. 以上皆是

58. 标准数据类型____能描述不带小数部分的数。
   a. int b. char c. float d. 以上皆是

59. 标准数据类型____可以描述计算机字母表中的任何值。
   a. int b. char c. float d. 以上皆是
4.2 开发环境

本章题。

Figure Building a Program

Figure Program Execution
The GNU Project is a free software, mass collaboration project, announced in 1983 by Richard Stallman. It initiated the GNU operating system, software development for which began in January 1984. The founding goal of the project was, in the words of its initial announcement, to develop "a sufficient body of free software [...] to get along without any software that is not free."

独立开发工具：编辑、编译、调试等是单独的工具，通常是命令行界面

编辑工具：vi
编译工具：g++
调试工具：gdb

CodeBlocks 编程环境
http://www.codeblocks.org/
基本概念
Project (工程)：编写程序的工作是以 Project 为单位。在开始一个新程序时，要先建立一个 Project，之后在程序编写过程中所有与这个程序有关的文件都会包含在这个 Project 中。

编制的程序可以有各种不同类型，编程环境为每种类型的程序准备了一个模板，用来生成程序的最初框架。在门课程里只介绍编写 Console Application 类型的程序。

这类程序的特点是：程序运行中会打开一个类似于 DOS 操作系统的界面，所有键盘输入都是通过 DOS 界面进行的，而所有输出都是输出到 DOS 窗口中。

Setting Up Code::Blocks and the MINGW Compiler on Windows
http://www.cprogramming.com/code_blocks/
This tutorial will provide you with detailed instructions for a standard installation of Code::Blocks and the MINGW compiler, a free development environment for C and C++.

Step 1: Download Code::Blocks

- Go to this website: http://www.codeblocks.org/downloads
- Find subsection Code::Blocks IDE (Windows binaries)
- Click on the link Code::Blocks IDE, with MINGW compiler
- Save the file to your desktop. It is roughly 14 megabytes.

Step 2: Install Code::Blocks

- Double click the installer.
- Hit next several times. Other setup tutorials will assume you have installed in C:\Program Files\CodeBlocks (the default install location), but you may install elsewhere if you like
- Do a Full Installation
- Launch Code::Blocks

Step 3: Running in Code::Blocks
You will be prompted with a Compilers auto-detection window:
When you get the compiler auto-detection window, just hit close. Click on the File menu, hit "New project". The following window will come up:
Click on "Console Application" and hit the "Create" button.

You will now be prompted with where you'd like to save the console application. I'd recommend you put it in its own folder, as it may create several files (this is especially true if you create other types of projects).

You can now open the main.cpp file on the left:

At this point, you will have your main.cpp file, which you can modify if you like. For now, it just says "Hello World!", so we can run it as is. Hit F9, which will first compile it and then run it.
You now have a running program! You can simply edit main.cpp and then hit F9 to compile it and run it again.
下篇  程序设计

下篇的主要目的是使读者学会使用计算机进行信息处理，熟练掌握 C++语言编程技术。
5.1 Getting Started

C++ is a programming language of many different dialects, similar to the way that each spoken language has many different dialects. In C++, dialects are not because the speakers live in the North or South. Instead, it is because there are many different compilers that support slightly different features. There are several common compilers: in particular, Borland C++, Microsoft C++, and GNU C++. There are also many front-end environments for the different compilers—the most common is Dev-C++ around GNU’s G++ compiler. Some, such as G++, are free, while others are not. Please see the compiler listing for more information on how to get a compiler and set it up.

Each of these compilers is slightly different. Each one should support the ANSI/ISO standard C++ functions, but each compiler will also have nonstandard functions (these functions are similar to slang spoken in different parts of a country). Sometimes the use of nonstandard functions will cause problems when you attempt to compile source code (the actual C++ written by a programmer and saved as a text file) with a different compiler. These tutorials use ANSI/ISO standard C++ and should not suffer from this problem (with sufficiently modern compilers). Note that if you are using an older compiler, such as TCLite, you should read check out some compatibility issues.

If you don't have a compiler, I strongly suggest that you get one. A simple compiler is sufficient for our use, but make sure that you do get one in order to get the most from these tutorials.

C++ is a different breed of programming language. A C++ program begins with a function, a collection of commands that do "something". The function that begins a C++ program is called main; this function is always called when the program first executes. From main, we can also call other functions whether they be written by us or by others. To access a standard function that comes with the compiler, you include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:
#include <iostream>
using namespace std;

int main()
{
    cout<"HEY, you, I'm alive! Oh, and Hello World!\n"
    cin.get();
}

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called iostream into our program before actually creating the executable. By including header files, you can gain access to many different functions. For example, the cout function requires iostream. Following the include is the statement, "using namespace std;". This line tells the compiler to use a group of functions that are part of the standard library (std). By including this line at the top of a file, you allow the program to use functions such as cout. The semicolon is part of the syntax of C and C++. It tells the compiler that you're at the end of a command. You will see later that the semicolon is used to end most commands in C++.

The next important line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as BEGIN and END. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

The next line of the program may seem strange. If you have programmed in another language, you might expect that print would be the function used to display text. In C++, however, the cout object is used to display text. It uses the "<<" symbols, known as "insertion operators", to indicate what to output. cout<< results in a function call with the ensuing text as an argument to the function. The quotes tell the compiler that you want to output the literal string as-is. The "\n" sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail). It moves the cursor on your screen to the next line. Again, notice the semicolon: it is added onto the end of all, such as function calls, in C++.

The next command is cin.get(). This is another function call: it reads in input and expects the user to hit the return key. Many compiler environments will open a new console window, run the program, and then close the window. This command keeps
that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Upon reaching the end of main, the closing brace, our program will return the value of 0 (and integer, hence why we told main to return an int) to the operating system. This return value is important as it can be used to tell the OS whether our program succeeded or not. A return value of 0 means success and is returned automatically (but only for main, other functions require you to manually return a value), but if we wanted to return something else, such as 1, we would have to do it with a return statement:

```c++
#include <iostream>
using namespace std;
int main()
{
    cout<<"HEY, you, I'm alive! Oh, and Hello World!\n";
    cin.get();

    return 1;
}
```

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .cpp (or whatever extension your compiler requires) file. If you are using a command-line compiler, such as Borland C++ 5.5, you should read the compiler instructions for information on how to compile. Otherwise compiling and running should be as simple as clicking a button with your mouse.

You might start playing around with the cout function and get used to writing C++.

Comments are critical for all but the most trivial programs and this tutorial will often use them to explain sections of code. When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment use either //, which tells the compiler that the rest of the line is a comment, or /* and then */ to block off everything between as a comment. Certain compiler environments will change the color of a commented area, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program. When you are learning to program, it is useful to be able to comment out sections of
code in order to see how the output is affected.

Comments: Explain programs to other programmers, Improve program readability, Ignored by compiler.
Single-line comment: Begin with //. Multi-line comment: Start with /*, End with */

Preprocessor directives & White space
Processed by preprocessor before compiling
Begin with #
#include <iostream> tells preprocessor to include the input/output stream header file <iostream>

White space, blank lines, space characters and tabs used to make programs easier to read. They are ignored by the compiler.

Function main & Statements
Function main is a part of every C++ program. Exactly one function in a program must be main, which can “return” a value. This main function, int main(), returns an integer. Body is delimited by braces ({}).

Statements: Instruct the program to perform an action. All statements end with a semicolon (;).

Namespace & Standard output stream object
Namespace std::, species using a name that belongs to “namespace” std. Can be removed through use of using statements.

Standard output stream object std::cout, “Connected” to screen. Defined in input/output stream header file <iostream>

Stream insertion operator & Escape characters
Stream insertion operator <<, Value to right (right operand) inserted into left operand. std::cout << "Hello"; Inserts the string "Hello" into the standard output. Displays to the screen

Escape characters, A character preceded by "\". "\n", Cursor moves to beginning of next line on the screen

Return statement
One of several means to exit a function

When used at the end of main, the value 0 indicates the program terminated successfully. Only for main, other functions require you to manually return a value, but if we wanted to return something else, such as 1, we would have to do it with a return statement.

Escape sequence (转义序列)

\n Newline. Position the screen cursor to the beginning of the next line.
\t Horizontal tab. Move the screen cursor to the next tab stop.
\r Carriage return. Position the screen to the beginning of the current line; do not advance to the next line.
\a Alert. Sound the system bell.
\\ Backslash. Used to print a backslash character.
\' Single quote. Used to print a single quote character.
\" Double quote. Used to print a double quote characters.

------------------------

5.2 Fundamental Types

So far you should be able to write a simple program to display information typed in by you, the programmer and to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input. The function you use is known as cin, and is followed by the extraction operator >>.

Of course, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables; when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char, int, and float.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you declare a variable.

Sometimes it can be confusing to have multiple variable types when it seems like some variable types are redundant. Using the right variable size can be important for
making your code readable and for efficiency—some variables require more memory than others. For now, suffice it to say that the different variable types will almost all be used!

To declare a variable you use the syntax type `<name>`. It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma. The declaration of a variable or set of variables should be followed by a semicolon (Note that this is the same procedure used when you call a function). If you attempt to use an undefined variable, your program will not run, and you will receive an error message informing you that you have made a mistake. Don't forget that variables, just like keywords, are case-sensitive, so it's best to use a consistent capitalization scheme to avoid these errors.

Here are some variable declaration examples:

```cpp
int x;
int a, b, c, d;
char letter;
float the_float;
```

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

Here is a sample program demonstrating the use of a variable:

```cpp
#include <iostream>
using namespace std;
int main()
{
    int thisisanumber;
    cout<<"Please enter a number: ";
    cin>> thisisanumber;
    cin.ignore();
    cout<<"You entered: "<< thisisanumber <<"n";
    cin.get();
}
```
Let's break apart this program and examine it line by line. The keyword int declares thisisanumber to be an integer. The function cin>> reads a value into thisisanumber; the user must press enter before the number is read by the program. cin.ignore() is another function that reads and discards a character. Remember that when you type input into a program, it takes the enter key too. We don't need this, so we throw it away. Keep in mind that the variable was declared an integer; if the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored). Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty. Notice that when printing out a variable quotation marks are not used. Were there quotation marks, the output would be "You Entered: thisisanumber." The lack of quotation marks informs the compiler that there is a variable, and therefore that the program should check the value of the variable in order to replace the variable name with the variable when executing the output function. Do not be confused by the inclusion of two separate insertion operators on one line. Including multiple insertion operators on one line is perfectly acceptable and all of the output will go to the same place. In fact, you must separate string literals (strings enclosed in quotation marks) and variables by giving each its own insertion operators (<<). Trying to put two variables together with only one << will give you an error message, do not try it. Do not forget to end functions and declarations with a semicolon. If you forget the semicolon, the compiler will give you an error message when you attempt to compile the program.

5.3 Arithmetic Operator

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C++ == is used for that task. The equal sign is still extremely useful. It sets the left input to the equal sign, which must be one, and only one, variable equal to the value on the right side of the equal sign. The operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:
a = 4 * 6; // (Note use of comments and of semicolon) a is 24
a = a + 5; // a equals the original value of a with five added to it
a == 5     // Does NOT assign five to a. Rather, it checks to see if a equals 5.

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is useful in other areas of C++; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

a < 5  // Checks to see if a is less than five
a > 5  // Checks to see if a is greater than five
a == 5 // Checks to see if a equals five, for good measure

Quiz: The basics of C++

1. What is the correct value to return to the operating system upon the successful completion of a program?
   A. -1  B. 1  C. 0  D. Programs do not return a value.

2. What is the only function all C++ programs must contain?
   A. start()  B. system()  C. main()  D. program()

3. What punctuation is used to signal the beginning and end of code blocks?
   A. {}  B. -> and <-  C. BEGIN and END  D. ( and )

4. What punctuation ends most lines of C++ code?
   A. .  B. ;  C. :  D. '

5. Which of the following is a correct comment?
   A. /* Comments */  B. ** Comment **  C. /* Comment */  D. { Comment }

6. Which of the following is not a correct variable type?
   A. float  B. real  C. int  D. double
7. Which of the following is the correct operator to compare two variables?
A. :=  B. =  C. equal  D. ==

5.4 Control Structures

http://www.cplusplus.com/doc/tutorial/control.html

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the compound-statement or block. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: {

{ statement1; statement2; statement3; }

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

if (condition) statement

Where condition is the expression that is being evaluated. If this condition is
true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints \textit{x is 100} only if the value stored in the \texttt{x} variable is indeed 100:

\begin{verbatim}
if (x == 100)
    cout << "x is 100";
\end{verbatim}

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces \{ \}:

\begin{verbatim}
if (x == 100)
    { 
        cout << "x is ";
        cout << x;
    }
\end{verbatim}

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword \texttt{else}. Its form used in conjunction with \texttt{if} is:

\begin{verbatim}
if (condition) statement1 else statement2
\end{verbatim}

For example:

\begin{verbatim}
if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";
\end{verbatim}

prints on the screen \textit{x is 100} if indeed \texttt{x} has a value of 100, but if it has not - and only if not - it prints out \textit{x is not 100}.

The \texttt{if + else} structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in \texttt{x} is positive, negative or none of them (i.e. zero):

\begin{verbatim}
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
\end{verbatim}
else
    cout << "x is 0";

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

**Iteration structures (loops)**

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

**The while loop**

Its format is:

```c++
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.

For example, we are going to make a program to countdown using a while-loop:

```c++
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
    int n;
    cout << "Enter the starting number > " ; Enter the starting number > 8
    cin >> n;         8, 7, 6, 5, 4, 3, 2, 1, FIRE!

    while (n>0) {
        cout << n << "", ";
        --n;
    }
}
```
cout << "FIRE!\n";
return 0;
}

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition \( n > 0 \) (that \( n \) is greater than zero) the block that follows the condition will be executed and repeated while the condition \( (n > 0) \) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to \( n \)
2. The while condition is checked \( (n > 0) \). At this point there are two possibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   
   ```cpp
   cout << n << " , ";
   --n;
   ```
   
   (prints the value of \( n \) on the screen and decreases \( n \) by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included \( --n; \) that decreases the value of the variable that is being evaluated in the condition \( (n) \) by one - this will eventually make the condition \( (n > 0) \) to become false after a certain number of loop iterations: to be more specific, when \( n \) becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

**The do-while loop**

Its format is:
do statement while (condition);

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
using namespace std;

int main ()
{
    unsigned long n;
    do {
        cout << "Enter number (0 to end): ";
        cin >> n;
        cout << "You entered: " << n << "\n";
    } while (n != 0);
    return 0;
}
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

**The for loop**

Its format is:
```
for (initialization; condition; increase) statement;
```
and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased
on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces \{ \}.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```cpp
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
    for (int n=10; n>0; n--)
    { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
        cout << n << ", ";
    }
    cout << "FIRE!\n";
    return 0;
}
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for ( ;n<10; ) if we wanted to specify no initialization and no increase; or for ( ;n<10;n++ ) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (, ) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (, ) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}

This loop will execute for 50 times if neither \texttt{n} or \texttt{i} are modified within the loop:

\begin{center}
\begin{tikzcd}
\text{Initialization} \arrow[draw=none]{r}
\text{Condition} \arrow[draw=none]{r}
\text{Increase}
\end{tikzcd}
\end{center}

\texttt{n} starts with a value of 0, and \texttt{i} with 100, the condition is \texttt{n!=i} (that \texttt{n} is not equal to \texttt{i}). Because \texttt{n} is increased by one and \texttt{i} decreased by one, the loop's condition will become false after the 50th loop, when both \texttt{n} and \texttt{i} will be equal to 50.

\textbf{Jump statements.}

\textbf{The break statement}

Using \texttt{break} we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

\begin{verbatim}
// break loop example
#include <iostream>
using namespace std;

int main ()
{
   int n;
   for (n=10; n>0; n--)
   {
      cout << n << ", ";
      if (n==3)
         break;
      cout << n << ", ";
   }

   10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

   return 0;
}
\end{verbatim}
cout << "countdown aborted!";
break;
}
}
return 0;
}

**The continue statement**

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```cpp
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
    for (int n=10; n>0; n--)
    {
        if (n==5) continue;
        cout << n << " , ";
    }
    cout << "FIRE!\n";
    return 0;
}
```

**The goto statement**

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object
oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

```
// goto loop example
#include <iostream>
using namespace std;
int main ()
{
    int n=10;
    loop:
    cout << n << "", ";
    n--;
    if (n>0) goto loop;
    cout << "FIRE!\n";
    return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The exit function

`exit` is a function defined in the `cstdlib` library.

The purpose of `exit` is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several `if` and `else if` instructions. Its form is the following:

```
switch (expression)
{
    case constant1:
```
group of statements 1;
break;
case constant2:
  group of statements 2;
  break;
.
.
.
default:
  default group of statements
}

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
}

if-else equivalent
if (x == 1) {
  cout << "x is 1";
}
else if (x == 2) {
  cout << "x is 2";
}
else {
  cout << "value of x unknown";
}
The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put `break` statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a `break` statement is reached.

For example, if we did not include a `break` statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a `break` instruction or the end of the switch selective block. This makes unnecessary to include braces `{ }` surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```cpp
switch (x) {
    case 1:
    case 2:
    case 3:
        cout << "x is 1, 2 or 3";
        break;
    default:
        cout << "x is not 1, 2 nor 3";
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3) :`) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of `if` and `else if` statements.
第6章 数组和结构

6.1 数组

http://cplusplus.com/doc/tutorial/arrays.html

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this:

```
  0   1   2   3   4
billy ------------
     |   |   |   |   |
     int
```

where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where `type` is a valid type (like `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

```
int billy [5];
```

**NOTE:** The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of
non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

6.1.1 Initializing arrays

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces {}.

For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

This declaration would have created an array like this:

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>billy</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-----</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
|       | 16| 2 | 77| 40| 12071
```

The amount of values between braces {} must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array `billy` we have declared that it has 5 elements and in the list of initial values within braces {} we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces {}:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array `billy` would be 5 ints long, since we have provided 5 initialization values.
6.1.2 Accessing the values of an array

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

\[
\text{name[index]}
\]

Following the previous examples in which \text{billy} had 5 elements and each of those elements was of type \text{int}, the name which we can use to refer to each element is the following:

\[
\begin{array}{cccccc}
\text{billy[0]} & \text{billy[1]} & \text{billy[2]} & \text{billy[3]} & \text{billy[4]} \\
\end{array}
\]

For example, to store the value 75 in the third element of \text{billy}, we could write the following statement:

\[
\text{billy[2]} = 75;
\]

and, for example, to pass the value of the third element of \text{billy} to a variable called \text{a}, we could write:

\[
\text{a} = \text{billy[2]};
\]

Therefore, the expression \text{billy[2]} is for all purposes like a variable of type \text{int}. Notice that the third element of \text{billy} is specified \text{billy[2]}, since the first one is \text{billy[0]}, the second one is \text{billy[1]}, and therefore, the third one is \text{billy[2]}. By this same reason, its last element is \text{billy[4]}. Therefore, if we write \text{billy[5]}, we would be accessing the sixth element of \text{billy} and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets \[ \] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets \[ \] with arrays.

\[
\text{int billy[5];} \quad \text{// declaration of a new array}
\]
billy[2] = 75;  // access to an element of the array.

If you read carefully, you will see that a type specifier always precedes a variable
or array declaration, while it never precedes an access.

Some other valid operations with arrays:

billy[0] = a;
billy[a] = 75;
b = billy[a+2];
billy[billy[a]] = billy[2] + 5;

// arrays example
#include <iostream>
using namespace std;
int billy[] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
    for ( n=0 ; n<5 ; n++ )  {
        result += billy[n];
    }
    cout << result;
    return 0;
}

6.1.3 Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a
bidimensional array can be imagined as a bidimensional table made of elements, all of
them of a same uniform data type.
jimmy represents a bidimensional array of 3 per 5 elements of type \texttt{int}. The way to declare this array in C++ would be:
\begin{verbatim}
int jimmy[3][5];
\end{verbatim}
and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:
\begin{verbatim}
jimmy[1][3]
\end{verbatim}

(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:
\begin{verbatim}
char century[100][365][24][60][60];
\end{verbatim}
declares an array with a \texttt{char} element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory! Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:
\begin{verbatim}
int jimmy[3][5]; // is equivalent to
int jimmy[15]; // (3 * 5 = 15)
\end{verbatim}
With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:
\begin{verbatim}
multidimensional array pseudo-multidimensional array
#define WIDTH 5 #define WIDTH 5
#define HEIGHT 3 #define HEIGHT 3

int jimmy [HEIGHT][WIDTH]; int jimmy [HEIGHT * WIDTH];
int n,m; int n,m;
\end{verbatim}
None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

We have used "defined constants" (\#define) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

\#define HEIGHT 3
to:
\#define HEIGHT 4
with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets [ ]. For example, the following function:
void procedure (int arg[])

accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

    int myarray [40];

it would be enough to write a call like this:

    procedure (myarray);

Here you have a complete example:

    // arrays as parameters
    #include <iostream>
    using namespace std;

    void printarray (int arg[], int length) {
        for (int n=0; n<length; n++)
            cout << arg[n] << " ";
        cout << "\n";
    }

    int main ()
    {
        int firstarray[] = {5, 10, 15};
        int secondarray[] = {2, 4, 6, 8, 10};
        printarray (firstarray,3);
        printarray (secondarray,5);
        return 0;
    }

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:
for example, a function with a multidimensional array as argument could be:

```c
void procedure (int myarray[][3][4]
```

Notice that the first brackets `[]` are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate.


Arrays are useful critters because they can be used in many ways. For example, a tic-tac-toe board can be held in an array. Arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

Think about arrays like this:

```
[ ][ ][ ][ ]
```

Each of the bracket pairs is a slot(element) in the array, and you can put information into each one of them. It is almost like having a group of variables side by side.

Let's look at the syntax for declaring an array.

```c
int examplearray[100]; // This declares an array
```

This would make an integer array with 100 slots, or places to store values(also called elements). To access a specific part element of the array, you merely put the array name and, in brackets, an index number. This corresponds to a specific element of the array. The one trick is that the first index number, and thus the first element, is zero, and the last is the number of elements minus one. 0-99 in a 100 element array, for example.

What can you do with this simple knowledge? Let's say you want to store a string,
because C had no built-in datatype for strings, it was common to use arrays of characters to simulate strings. (C++ now has a \texttt{string} type as part of the standard library.)

For example:

\begin{Verbatim}
char astring[100];
\end{Verbatim}

will allow you to declare a char array of 100 elements, or slots. Then you can receive input into it from the user, and if the user types in a long string, it will go in the array. The neat thing is that it is very easy to work with strings in this way, and there is even a header file called cstring. There is another lesson on the uses of strings, so it's not necessary to discuss here.

The most useful aspect of arrays is multidimensional arrays. How I think about multi-dimensional arrays:

\begin{verbatim}
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
[ ][ ][ ][ ]
\end{verbatim}

This is a graphic of what a two-dimensional array looks like when I visualize it. For example:

\begin{Verbatim}
int twodimensionalarray[8][8];
\end{Verbatim}

declares an array that has two dimensions. Think of it as a chessboard. You can easily use this to store information about some kind of game or to write something like tic-tac-toe. To access it, all you need are two variables, one that goes in the first slot and one that goes in the second slot. You can even make a three-dimensional array, though you probably won't need to. In fact, you could make a four-hundred dimensional array. It would be confusing to visualize, however. Arrays are treated like any other variable in most ways. You can modify one value in it by putting:

\begin{Verbatim}
arrayname[arrayindexnumber] = whatever;
\end{Verbatim}

or, for two-dimensional arrays
arrayname[arrayindexnumber1][arrayindexnumber2] = whatever;

However, you should never attempt to write data past the last element of the array, such as when you have a 10 element array, and you try to write to the [10] element. The memory for the array that was allocated for it will only be ten locations in memory, but the next location could be anything, which could crash your computer.

You will find lots of useful things to do with arrays, from storing information about certain things under one name, to making games like tic-tac-toe. One suggestion I have is to use for loops when access arrays.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int x;
    int y;
    int array[8][8]; // Declares an array like a chessboard
    for ( x = 0; x < 8; x++ ) {
        for ( y = 0; y < 8; y++ )
            array[x][y] = x * y; // Set each element to a value
    }
    cout<<"Array Indices:\n";
    for ( x = 0; x < 8; x++ ) {
        for ( y = 0; y < 8; y++ )
            cout<<"["<<x<<"]["<<y<<"]="<< array[x][y] <<" ";
        cout<<"\n";
    }
    cin.get();
}
```

Here you see that the loops work well because they increment the variable for you, and you only need to increment by one. Its the easiest loop to read, and you access the entire array.

One thing that arrays don't require that other variables do, is a reference operator when you want to have a pointer to the string. For example:

```cpp
char *ptr;
char str[40];
```
ptr = str;  // Gives the memory address without a reference operator(&)

As opposed to
int *ptr;
int num;
ptr = &num; // Requires & to give the memory address to the ptr

The reason for this is that when an array name is used as an expression, it refers to a pointer to the first element, not the entire array. This rule causes a great deal of confusion, for more information please see our Frequently Asked Questions.

### 6.1.4 Quiz: Arrays

1. Which of the following correctly declares an array?
   - A. int anarray[10];
   - B. int anarray;
   - C. anarray{10};
   - D. array anarray[10];

2. What is the index number of the last element of an array with 29 elements?
   - A. 29
   - B. 28
   - C. 0
   - D. Programmer-defined

3. Which of the following is a two-dimensional array?
   - A. array anarray[20][20];
   - B. int anarray[20][20];
   - C. int array[20, 20];
   - D. char array[20];

4. Which of the following correctly accesses the seventh element stored in foo, an array with 100 elements?
   - A. foo[6];
   - B. foo[7];
   - C. foo(7);
   - D. foo;
5. Which of the following gives the memory address of the first element in array foo, an array with 100 elements?
A. foo[0];  
B. foo;  
C. &foo;  
D. foo[1];

6.2 结构

http://cplusplus.com/doc/tutorial/structures.html

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types.

6.2.1 Data structures

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```cpp
struct structure_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    ...
} object_names;
```

where `structure_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{ }` there is a list with the data members, each one is specified with a type and a valid
The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as \texttt{structure\_name} is created and can be used in the rest of the program as if it was any other type. For example:

```c
struct product {
    int weight;
    float price;
} ;

product apple;
product banana, melon;
```

We have first declared a structure type called \texttt{product} with two members: \texttt{weight} and \texttt{price}, each of a different fundamental type. We have then used this name of the structure type (\texttt{product}) to declare three objects of that type: \texttt{apple}, \texttt{banana} and \texttt{melon} as we would have done with any fundamental data type. Once declared, \texttt{product} has become a new valid type name like the fundamental ones \texttt{int}, \texttt{char} or \texttt{short} and from that point on we are able to declare objects (variables) of this compound new type, like we have done with \texttt{apple}, \texttt{banana} and \texttt{melon}.

Right at the end of the \texttt{struct} declaration, and before the ending semicolon, we can use the optional field \texttt{object\_name} to directly declare objects of the structure type. For example, we can also declare the structure objects \texttt{apple}, \texttt{banana} and \texttt{melon} at the moment we define the data structure type this way:

```c
struct product {
    int weight;
    float price;
} apple, banana, melon;
```

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like \texttt{apple}, \texttt{banana} and \texttt{melon}) from a single structure type (\texttt{product}).

Once we have declared our three objects of a determined structure type (\texttt{apple}, \texttt{banana} and \texttt{melon}) we can operate directly with their members. To do that we use a dot (\texttt{.}) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their
respective types:

- apple.weight
- apple.price
- banana.weight
- banana.price
- melon.weight
- melon.price

Each one of these has the data type corresponding to the member they refer to:

- apple.weight, banana.weight and melon.weight are of type int, while
- apple.price, banana.price and melon.price are of type float.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

```c++
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
    string title;
    int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
    string mystr;

    mine.title = "2001 A Space Odyssey";
    mine.year = 1968;

    cout << "Enter title: ";
    getline (cin,yours.title);
    cout << "Enter year: ";
```

Enter title: Alien
Enter year: 1979

My favorite movie is:
2001 A Space Odyssey (1968)
And yours is:
Alien (1979)
getline (cin, mystr);
stringstream(mystr) >> yours.year;

cout << "My favorite movie is:\n ";
printmovie (mine);
cout << "And yours is:\n ";
printmovie (yours);
return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year << ")\n";
}

The example shows how we can use the members of an object as regular variables. For example, the member yours.year is a valid variable of type int, and mine.title is a valid variable of type string.

The objects mine and yours can also be treated as valid variables of type movies_t, for example we have passed them to the function printmovie as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.

Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them:

```cpp
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
#define N_MOVIES 3

struct movies_t {
    string title;
};

int main() {
    string mystr;
    for (int i = 0; i < N_MOVIES; i++) {
        cout << "Enter title: ";
        getline (cin, mystr);
        stringstream(mystr) >> yours.year;
        cout << "Enter year: ";
        getline (cin, mystr);
        stringstream(mystr) >> yours.year;
        cout << "You have entered these movies:
";
        printmovie (mine);
        cout << "Blade Runner (1982)\n";
        printmovie (yours);
        cout << "Matrix (1999)\n";
        cout << "Enter title: ";
        getline (cin, mystr);
        stringstream(mystr) >> yours.year;
        cout << "Enter year: ";
        getline (cin, mystr);
        stringstream(mystr) >> yours.year;
    }
    return 0;
}
```
void printmovie (movies_t movie);

int main ()
{
    string mystr;
    int n;

    for (n=0; n<N_MOVIES; n++)
    {
        cout << "Enter title: ";
        getline (cin,films[n].title);
        cout << "Enter year: ";
        getline (cin,mystr);
        stringstream(mystr) >> films[n].year;
    }

    cout << "You have entered these movies:\n"
    for (n=0; n<N_MOVIES; n++)
        printmovie (films[n]);
    return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year << ")\n";
}

6.2.2 Pointers to structures

Like any other type, structures can be pointed by its own type of pointers:
struct movies_t {
        string title;
        int year;
};

movies_t amovie;
movies_t * pmovie;

Here amovie is an object of structure type movies_t, and pmovie is a pointer to point to objects of structure type movies_t. So, the following code would also be valid:

pmovie = &amovie;

The value of the pointer pmovie would be assigned to a reference to the object amovie (its memory address).

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (\rightarrow):

// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
        string title;
        int year;
};

int main ()
{
        string mystr;

        movies_t amovie;
movies_t * pmovie;
pmovie = &amovie;

        cout << "Enter title: ";
        Enter title: Invasion of the body snatchers
        Enter year: 1978
        You have entered:
        Invasion of the body snatchers (1978)
getline (cin, pmovie->title);
cout << "Enter year: ";
generate (cin, mystr);
(stringstream) mystr >> pmovie->year;

cout << "\nYou have entered:\n";
cout << pmovie->title;
cout << " (" << pmovie->year << ")\n";

return 0;
}

The previous code includes an important introduction: the arrow operator (->). This is a
dereference operator that is used exclusively with pointers to objects with members.
This operator serves to access a member of an object to which we have a reference. In
the example we used:

pmovie->title

Which is for all purposes equivalent to:

(*pmovie).title

Both expressions pmovie->title and (*pmovie).title are valid and both mean
that we are evaluating the member title of the data structure pointed by a pointer
called pmovie. It must be clearly differentiated from:

*pmovie.title

which is equivalent to:

*(pmovie.title)

And that would access the value pointed by a hypothetical pointer member called
title of the structure object pmovie (which in this case would not be a pointer). The
following panel summarizes possible combinations of pointers and structure members:

<table>
<thead>
<tr>
<th>Expression</th>
<th>What is evaluated</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.b</td>
<td>Member b of object a</td>
<td>*(a).b</td>
</tr>
<tr>
<td>a-&gt;b</td>
<td>Member b of object pointed by a</td>
<td>(*a).b</td>
</tr>
<tr>
<td>*a.b</td>
<td>Value pointed by member b of object a</td>
<td>*(a.b)</td>
</tr>
</tbody>
</table>
6.2.3 Nesting structures

Structures can also be nested so that a valid element of a structure can also be in its turn another structure.

```c
struct movies_t {
    string title;
    int year;
};
```

```c
struct friends_t {
    string name;
    string email;
    movies_t favorite_movie;
} charlie, maria;
```

```c
friends_t * pfriends = &charlie;
```

After the previous declaration we could use any of the following expressions:

- charlie.name
- maria.favorite_movie.title
- charlie.favorite_movie.year
- pfriends->favorite_movie.year

(where, by the way, the last two expressions refer to the same member).

6.2.4 Quiz : Structures

1. Which of the following accesses a variable in structure b?
   A. b->var; B. b.var; C. b-var; D. b>var;

2. Which of the following accesses a variable in structure *b?
   A. b->var; B. b.var; C. b-var; D. b>var;

3. Which of the following is a properly defined struct?
   A. struct {int a;} B. struct a_struct {int a;}

• 122 •
C. struct a_struct int a;  D. struct a_struct {int a;};

4. Which properly declares a variable of struct foo?
A. struct foo;  B. foo var;  C. foo;  D. int foo;
第7章C++标准库

http://www.cplusplus.com/reference/
http://www.gamedev.net/reference/articles/article2320.asp
http://net.pku.edu.cn/~yhf/tutorial/STL_doc/index.html

The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed. Much of the power and productivity of the C++ language comes from its standard library.

C++标准库很大，在现在的情况下，C++标准库确实越来越好，因为大的库会包含大量的功能。标准库中的功能越多，开发自己的应用程序时能借助的功能就越多，C++库并非提供一切（很明显的是没有提供开发和图形用户接口的支持），但确实提供了很多。标准 C++库中主要有以下主要组件:

C Language library
The elements of the C language library are also included as a subset of the C++ Standard library. These cover many aspects, from general utility functions and macros to input/output functions and dynamic memory management functions.

Input/Output Stream library
Provides functionality to use an abstraction called streams specially designed to perform input and output operations on sequences of character, like files or strings. This functionality is provided through several related classes, as shown in the following relationship map, with the corresponding header file names: <ios>, <istream>, <iostream>, <fstream>, and <sstream>.

String library
The C++ strings library provides the definitions of the basic_string class, which is a class template specifically designed to manipulate strings of characters of any character type. It also include two specific instantiations: string and wstring, which respectively use char and wchar_t as character types.

STL: Standard Template Library
The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures
of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

---

A Note About Standards

C++ is a constantly evolving language, so it's important to know what version a given article is talking about. The C++ language was originally created by Bjarne Stroustrup, and was released in a series of versions along with the compiler CFront. The C++ Standards committee was formed in 1989 and the first version of the standard was released in 1998 in the document ISO/IEC 14882:1998. In 2003, a technical corregenda was released in the form of the document ISO/IEC 14882:2003. This is an essentially an errata version, no major changes were introduced in 14882:2003, only clarifications and corrections. The standards committee is currently working on the next version of the standard, which is anticipated to be released in 2009 and will have major changes. In the meantime, the standard library committee is working on a series of additions to the library that are being released in a documents called technical reports. At this time, Technical Report 1 has been released, and Technical Report 2 is still in the process of accepting proposals.

Namespace

标准库中东西很多，程序员所选择的类名或函数名很有可能和标准库中的某个名字相同。为了避免这种情况所造成的名字冲突，标准库中的一切被放到了命名空间 std 中。

A namespace groups different identifiers in a named scope. By defining all identifiers in a namespace, the name of the namespace is the only global identifier that might conflict with other global symbols.

---

7.1 C Language library

http://www.cplusplus.com/reference/clibrary/

The C++ library includes the same definitions as the C language library organized in the same structure of header files, with the following differences:
Each header file has the same name as the C language version but with a "c" prefix and no extension. For example, the C++ equivalent for the C language header file `<stdlib.h>` is `<cstdlib>`.

Every element of the library is defined within the `std` namespace.

Nevertheless, for compatibility with C, the traditional header names `name.h` (like `stdlib.h`) are also provided with the same definitions within the global namespace. In the examples provided in this reference, this version is used so that the examples are fully C-compatible, although its use is deprecated in C++.

cctype (ctype.h)

cctype, character handling functions

This header declares a set of functions to classify and transform individual characters.

All these functions take as parameter the `int` equivalent of one character and return an `int`, that can either be another character or a value representing a boolean value: an `int` value of 0 means false, and an `int` value different from 0 represents true.

There are two sets of functions:

First a set of classifying functions that check whether the character passed as parameter belongs to a certain category.

And secondly, two functions to convert between letter cases

### 7.2 Input/Output Stream library


The iostream library is an object-oriented library that provides input and output functionality using streams.

A stream is an abstraction that represents a device on which input and output operations are performed. A stream can basically be represented as a source or destination of characters of indefinite length.
Streams are generally associated to a physical source or destination of characters, like a disk file, the keyboard, or the console, so the characters gotten or written to/from our abstraction called stream are physically input/output to the physical device. For example, file streams are C++ objects to manipulate and interact with files; Once a file stream is used to open a file, any input or output operation performed on that stream is physically reflected in the file.

The library and its hierarchy of classes is split in different files:

- `<ios>,<istream>,<ostream>,<streambuf>` and `<iosfwd>` aren't usually included directly in most C++ programs. They describe the base classes of the hierarchy and are automatically included by other header files of the library that contain derived classes.
- `<iostream>` declares the objects used to communicate through the standard input and output (including `cin` and `cout`).
- `<fstream>` defines the file stream classes (like the template `basic_ifstream` or the class `ofstream`) as well as the internal buffer objects used with these (`basic_filebuf`). These classes are used to manipulate files using streams.
- `<sstream>`: The classes defined in this file are used to manipulate string objects as if they were streams.
- `<iomanip>` declares some standard manipulators with parameters to be used with extraction and insertion operators to modify internal flags and formatting options.

### 7.3 String library


### 7.4 STL: Standard Template Library

Standard Template Library: Containers

A container is a holder object that stores a collection other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

stack, queue and priority_queue are implemented as container adaptors. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the container class independently of the underlying container class used.

http://www.cplusplus.com/reference/algorithm/

Standard Template Library: Algorithms

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or
pointers, such as an array or an instance of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).
Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type specifier of the data returned by the function.
- `name` is the identifier by which it will be possible to call the function.
- `parameters` (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`.

Here you have the first function example:

```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
    int r;
    r=a+b;
    return (r);
}
```

The result is 8
int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
    return 0;
}

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

\[
\text{int addition (int a, int b)}
\]

\[
\text{z = addition (5, 3);} 
\]

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares another local variable (int r), and by means of the expression \( r = a + b \), it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

\[
\text{return (r);} 
\]

finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows it regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This
value becomes the value of evaluating the function call.

```c
int addition (int a, int b)
{
    z = addition ( 5 , 3 );
}
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to the value returned by `addition (5, 3)`, that is 8. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (8).

The following line of code in main is:

```c
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.
Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables \( a \), \( b \) or \( r \) directly in function \( \text{main} \) since they were variables local to function \( \text{addition} \). Also, it would have been impossible to use the variable \( z \) directly within function \( \text{addition} \), since this was a variable local to the function \( \text{main} \).

```cpp
#include <iostream.h>

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

main ()
{
    unsigned short Age;
    float ANumber, AnotherOne;

    cout << "Enter your age:
    cin >> Age;
    ...
}
```

Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; these are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

```cpp
// function example
```

The first result
```cpp
#include <iostream>
using namespace std;

int subtraction (int a, int b) {
    int r;
    r=a-b;
    return (r);
}

int main () {
    int x=5, y=3, z;
    z = subtraction (7,2);
    cout << "The first result is " << z << '\n';
    cout << "The second result is " << subtraction (7,2) << '\n';
    cout << "The third result is " << subtraction (x,y) << '\n';
    z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n';
    return 0;
}
```

In this case we have created a function called `subtraction`. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function main we will see that we have made several calls to function `subtraction`. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:
z = 5;
cout << "The first result is " << z;

As well as
cout << "The second result is " << subtraction (7,2);
has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:
cout << "The second result is " << 5;
since 5 is the value returned by subtraction (7,2).
In the case of:
cout << "The third result is " << subtraction (x,y);

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result.
The fourth case is more of the same. Simply note that instead of:
z = 4 + subtraction (x,y);
we could have written:
z = subtraction (x,y) + 4;
with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:
z = 4 + 2;
z = 2 + 4;

8.1 Functions with no type. The use of void.

If you remember the syntax of a function declaration:

type name ( argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return
statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the \texttt{void} type specifier for the function. This is a special specifier that indicates absence of type.

\begin{verbatim}
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
    cout << "I'm a function!";
} I'm a function!

int main ()
{
    printmessage ();
    return 0;
}
\end{verbatim}

\texttt{void} can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function \texttt{printmessage} could have been declared as:

\begin{verbatim}
void printmessage (void)
{
    cout << "I'm a function!";
}
\end{verbatim}

Although it is optional to specify \texttt{void} in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to \texttt{printmessage} is:

\begin{verbatim}
printmessage ();
\end{verbatim}

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:
8.2 Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```c
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of $x$ and $y$, i.e. 5 and 3 respectively, but not the variables $x$ and $y$ themselves.

```c
int addition (int a, int b)
{
    z = addition ( 5 , 3 );
}
```

This way, when the function addition is called, the value of its local variables $a$ and $b$ become 5 and 3 respectively, but any modification to either $a$ or $b$ within the function addition will not have any effect in the values of $x$ and $y$ outside it, because variables $x$ and $y$ were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```c
// passing parameters by reference
#include <iostream>
using namespace std;
x=2, y=6, z=14
void duplicate (int& a, int& b, int& c)
{
    a*=2;
}
```
b*=2;
c*=2;
}

int main()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << " y=" << y << " z=" << z;
    return 0;
}

The first thing that should call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (`&`). This ampersand is what specifies that their corresponding arguments are to be passed by reference instead of by value.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

```c
void duplicate (int& a, int& b, int& c)
```

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to `duplicate`, shows the values of all the three variables of `main` doubled.

If when declaring the following function:

```c
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```c
void duplicate (int a, int b, int c)
```
i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```cpp
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
    prev = x-1;
    next = x+1;
}

int main ()
{
    int x=100, y, z;
    prevnext (x, y, z);
    cout << "Previous=\n" << y << ", Next=" << z;
    return 0;
}
```

Previous=99, Next=101

### 8.3 Default values in parameters

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```cpp
// default values in functions
#include <iostream>
using namespace std;

void prevnext (int x=100, int& prev, int& next)
{
    prev = x-1;
    next = x+1;
}
```

...
int divide (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}

int main ()
{
    cout << divide (12);
    cout << endl;
    cout << divide (20,4);
    return 0;
}

As we can see in the body of the program there are two calls to function divide. In the first one:
divide (12)
we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).
In the second call:
divide (20,4)
there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

8.4 Overloaded functions

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

// overloaded function 10
#include <iostream>

using namespace std;

int operate (int a, int b)
{
    return (a*b);
}

float operate (float a, float b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << operate (x,y);
    cout << "n";
    cout << operate (n,m);
    cout << "n";
    return 0;
}

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate
depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

### 8.5 Inline Functions

The *inline* specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```c++
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the *inline* keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

### 8.6 Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number \((n!)\) the mathematical formula would be:

\[
 n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 1 \\
\]

more concretely, \(5!\) (factorial of 5) would be:

\[
5! = 5 \times 4 \times 3 \times 2 \times 1 = 120
\]

and a recursive function to calculate this in C++ could be:

```c++
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a) {
    // Please type a number: 9
    if (a == 0) return 1;
    return a * factorial(a-1); // 9! = 362880
}
```

```c++
Please type a number: 9
9! = 362880
long factorial (long a)
```
{ 
    if (a > 1) 
        return (a * factorial (a-1)); 
    else 
        return (1); 
}

int main ()
{
    long number;
    cout << "Please type a number: ";
    cin >> number;
    cout << number << "! = " << factorial (number);
    return 0;
}

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.


Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C++, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an
extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

A simple example of recursion would be:

```c
void recurse()
{
    recurse(); //Function calls itself
}

int main()
{
    recurse(); //Sets off the recursion
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

```c
#include <iostream>

using namespace std;

void recurse ( int count ) // Each call gets its own count
{
    cout<< count <<"\n";
    // It is not necessary to increment count since each function's
    // variables are separate (so each count will be initialized one greater)
    recurse ( count + 1 );
}

int main()
{
    recurse ( 1 ); //First function call, so it starts at one
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it
was previous by passing in count + 1. Keep in mind, it is not a function restarting itself, it is hundreds of functions that are each unfinished with the last one calling a new recurse function.

It can be thought of like the Russian dolls that always have a smaller doll inside. Each doll calls another doll, and you can think of the size being a counter variable that is being decremented by one.

Think of a really tiny doll, the size of a few atoms. You can't get any smaller than that, so there are no more dolls. Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit. The condition where the function will not call itself is termed the base case of the function. Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true and only then allow the function to call itself).

A quick example:

```c
void doll ( int size )
{
    if ( size == 0 )    // No doll can be smaller than 1 atom (10^0==1) so doesn't call itself
        return;          // Return does not have to return something, it can be used
    doll ( size - 1 );  // Decrements the size variable so the next doll will be smaller.
}
int main()
{
    doll ( 10 ); //Starts off with a large doll (its a logarithmic scale)
}
```

This program ends when size equals one. This is a good base case, but if it is not properly set up, it is possible to have an base case that is always true (or always false).

Once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable...twice, once before the function recurses, and once after...

```c
void printnum ( int begin )
```
{ 
    cout<< begin;
    if ( begin < 9 ) // The base case is when begin is greater than 9
    {
        // for it will not recurse after the if-statement
        printnum ( begin + 1);
    }
    cout<< begin; // Outputs the second begin, after the program has
                   // gone through and output
}

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin.

This is just the beginning of the usefulness of recursion. Heres a little challenge, use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number*number-1*number-2...*1).

Hint: Recursively find the factorial of the smaller numbers first, ie, it takes a number, finds the factorial of the previous number, and multiplies the number times that factorial...have fun. :-)

8.7 Declaring functions

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters. Its form is:

type name ( argument_type1, argument_type2, ...);
It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called \texttt{protofunction} with two \texttt{int} parameters with any of the following declarations:

\begin{verbatim}
int protofunction (int first, int second);
int protofunction (int, int);
\end{verbatim}

Anyway, including a name for each variable makes the prototype more legible.

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int a);
void even (int a);

int main ()
{
    int i;
    do {
        cout << "Type a number (0 to exit): ";
        cin >> i;
        odd (i);
    } while (i!=0);
    return  0;
}

void odd (int a)
{
    if ((a%2)!=0) cout << "Number is odd.\n";
    else even (a);
}
```
void even (int a)
{
    if ((a%2)==0) cout << "Number is even.\n";
    else odd (a);
}

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:
void odd (int a);
void even (int a);

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.
9.1 Pointers

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

9.1.1 Reference operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.
The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```c
    ted = &andy;
```

This would assign to `ted` the address of variable `andy`, since when preceding the name of the variable `andy` with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that `andy` is placed during runtime in the memory address 1776. This number (1776) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```c
    andy = 25;
    fred = andy;
    ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:

```
<table>
<thead>
<tr>
<th>andy</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1775</td>
<td>1776</td>
</tr>
</tbody>
</table>
```

![Diagram](https://via.placeholder.com/150)

First, we have assigned the value 25 to `andy` (a variable whose address in memory we have assumed to be 1776).

The second statement copied to `fred` the content of variable `andy` (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to `ted` not the value contained in `andy` but a reference to it (i.e., its address, which we have assumed to be 1776). The reason is that
in this third assignment operation we have preceded the identifier Andy with the reference operator (&), so we were no longer referring to the value of Andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like Ted in the previous example) is what we call a pointer. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

9.1.2 Dereference operator (*)

We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```c
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since Ted is 1776, and the value pointed by 1776 is 25.

You must clearly differentiate that the expression Ted refers to the value 1776, while *Ted (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the
dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

\[
beth = ted; \quad // \text{beth equal to ted (1776)} \\
beth = *ted; \quad // \text{beth equal to value pointed by ted (25)}
\]

Notice the difference between the reference and dereference operators:

- & is the reference operator and can be read as "address of"
- * is the dereference operator and can be read as "value pointed by"

Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *.

Earlier we performed the following two assignment operations:

\[
andy = 25; \\
ted = &andy;
\]

Right after these two statements, all of the following expressions would give true as result:

\[
andy == 25 \\
&andy == 1776 \\
ted == 1776 \\
*ted == 25
\]

The first expression is quite clear considering that the assignment operation performed on andy was andy=25. The second one uses the reference operator (&), which returns the address of variable andy, which we assumed it to have a value of 1776. The third one is somewhat obvious since the second expression was true and the assignment operation performed on téd was téd=&andy. The fourth expression uses the dereference operator (*) that, as we have just seen, can be read as "value pointed by", and the value pointed by téd is indeed 25.

So, after all that, you may also infer that for as long as the address pointed by téd remains unchanged the following expression will also be true:

\[
*ted == andy
\]

### 9.1.3 Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes
necessary to specify in its declaration which data type a pointer is going point to. It is not the same thing to point to a `char` than to point to an `int` or a `float`.

The declaration of pointers follows this format:

```cpp
type * name;
```

where `type` is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:

```cpp
int * number;
char * character;
float * greatnumber;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an `int`, the second one to a `char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (`*`) that we use when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator that we have seen a bit earlier, but which is also written with an asterisk (`*`). They are simply two different things represented with the same sign.

Now have a look at this code:

```cpp
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
    int firstvalue, secondvalue;
    int * mypointer;

    firstvalue is 10
    secondvalue is 20
}
```
mypointer = &firstvalue;
*mypointer = 10;
mypointer = &secondvalue;
*mypointer = 20;
cout << "firstvalue is " << firstvalue << endl;
cout << "secondvalue is " << secondvalue << endl;
return 0;
}

Notice that even though we have never directly set a value to either firstvalue or secondvalue, both end up with a value set indirectly through the use of mypointer. This is the procedure:

First, we have assigned as value of mypointer a reference to firstvalue using the reference operator (&). And then we have assigned the value 10 to the memory location pointed by mypointer, that because at this moment is pointing to the memory location of firstvalue, this in fact modifies the value of firstvalue.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with secondvalue and that same pointer, mypointer.

Here is an example a little bit more elaborated:

// more pointers
#include <iostream>
using namespace std;

int main ()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;

    p1 = &firstvalue;  // p1 = address of firstvalue
    p2 = &secondvalue; // p2 = address of secondvalue

    *p1 = 10;          // value pointed by p1 = 10
    *p2 = *p1;         // value pointed by p2 = value
pointed by p1
    p1 = p2;       // p1 = p2 (value of pointer
is copied)
    *p1 = 20;     // value pointed by p1 = 20

    cout << "firstvalue is " << firstvalue << endl;
    cout << "secondvalue is " << secondvalue <<
end1;
    return 0;
}

I have included as a comment on each line how the code can be read: ampersand (&) as "address of" and asterisk (*) as "value pointed by".

Notice that there are expressions with pointers p1 and p2, both with and without dereference operator (*). The meaning of an expression using the dereference operator (*) is very different from one that does not: When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

Another thing that may call your attention is the line:

int * p1, * p2;

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type int* (pointer to int).

Otherwise, the type for the second variable declared in that line would have been int (and not int*) because of precedence relationships. If we had written:

int * p1, p2;

p1 would indeed have int* type, but p2 would have type int (spaces do not matter at all for this purpose). This is due to operator precedence rules. But anyway, simply remembering that you have to put one asterisk per pointer is enough for most pointer users.

\subsection{9.1.4 Pointers and arrays}

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the
address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

```c
int numbers[20];
int * p;
```

The following assignment operation would be valid:

```c
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a constant pointer. Therefore, the following allocation would not be valid:

```c
numbers = p;
```

Because `numbers` is an array, so it operates as a constant pointer, and we cannot assign values to constants.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

```c
#include <iostream>
using namespace std;

int main ()
{
    int numbers[5];
    int * p;
    p = numbers;  *p = 10;       10, 20, 30, 40, 50,
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << "", ";
    return 0;
}
```

In the chapter about arrays we used brackets ([ ]) several times in order to specify the
index of an element of the array to which we wanted to refer. Well, these bracket sign operators [] are also a dereference operator known as offset operator. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

```c
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

These two expressions are equivalent and valid both if `a` is a pointer or if `a` is an array.

### 9.1.5 Pointer initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:

```c
int number;
int *tommy = &number;
```

The behavior of this code is equivalent to:

```c
int number;
int *tommy;
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (`tommy`), never the value being pointed (`*tommy`). You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

```c
int number;
int *tommy;
*tommy = &number;
```

that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```c
char * terry = "hello";
```

In this case, memory space is reserved to contain "hello" and then a pointer to the
first character of this memory block is assigned to `terry`. If we imagine that "hello" is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:

![Diagram showing memory locations]

It is important to indicate that `terry` contains the value 1702, and not 'h' nor "hello", although 1702 indeed is the address of both of these.

The pointer `terry` points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these two expression:

\[
*\text{(terry} + 4)\\
\text{terry}[4]
\]

Both expressions have a value of 'o' (the fifth element of the array).

### 9.1.6 Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, `char` takes 1 byte, `short` takes 2 bytes and `long` takes 4.

Suppose that we define three pointers in this compiler:

```c
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000
respectively.  
So if we write:
```c
mychar++;  
myshort++;  
mylong++;  
```

mychar, as you may expect, would contain the value 1001. But not so obviously, 
myshort would contain the value 2002, and mylong would contain 3004, even 
though they have each been increased only once. The reason is that when adding one to 
a pointer we are making it to point to the following element of the same type with 
which it has been defined, and therefore the size in bytes of the type pointed is added to 
the pointer.

This is applicable both when adding and subtracting any number to a pointer. It would 
happen exactly the same if we write:
```c
mychar = mychar + 1;  
myshort = myshort + 1;  
mylong = mylong + 1;  
```

Both the increase (++) and decrease (--) operators have greater operator precedence 
than the dereference operator (*), but both have a special behavior when used as suffix 
(the expression is evaluated with the value it had before being increased). Therefore, 
the following expression may lead to confusion:
*p++

Because ++ has greater precedence than *, this expression is equivalent to *(p++). Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

(*p)++

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

*p++ = *q++;

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. And then both are increased. It would be roughly equivalent to:

*p = *q;
++p;
++q;

Like always, I recommend you to use parentheses () in order to avoid unexpected results and to give more legibility to the code.

9.1.7 Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```cpp
char a;
char * b;
char ** c;
a = 'z';
```
b = &a;
c = &b;

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'z'</td>
<td>7230</td>
<td>8092</td>
</tr>
<tr>
<td>7230</td>
<td>8092</td>
<td>10502</td>
<td></td>
</tr>
</tbody>
</table>

The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

### 9.1.8 void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

One of its uses may be to pass generic parameters to a function:

```cpp
#include <iostream>
using namespace std;

// increaser
```
```cpp
void increase (void* data, int psize)
{
    if ( psize == sizeof(char) )
    { char* pchar; pchar=(char*)data; ++(*pchar); }
    else if (psize == sizeof(int) )
    { int* pint; pint=(int*)data; ++(*pint); }
}

t int main ()
{
    char a = 'x';
    int b = 1602;
    increase (&a,sizeof(a));
    increase (&b,sizeof(b));
    cout << a << " ", " << b << endl;
    return 0;
}
```

`sizeof` is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non-dynamic data types this value is a constant. Therefore, for example, `sizeof(char)` is 1, because `char` type is one byte long.

### 9.1.9 Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;  // p has a null pointer value
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.
9.1.10 Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```cpp
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()
{
    int m,n;
    int (*minus)(int,int) = subtraction;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout <<n;
    return 0;
}
```
In the example, \texttt{minus} is a pointer to a function that has two parameters of type \texttt{int}. It is immediately assigned to point to the function \texttt{subtraction}, all in a single line:

\begin{verbatim}
int (* minus)(int,int) = subtraction;
\end{verbatim}

### 9.2 Dynamic Memory

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is \textit{dynamic memory}, for which C++ integrates the operators \texttt{new} and \texttt{delete}.

#### 9.2.1 Operators \texttt{new} and \texttt{new[]} 

In order to request dynamic memory we use the operator \texttt{new}. \texttt{new} is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets \texttt{[\ldots\texttt{]}}. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

\begin{verbatim}
pointer = new type
pointer = new type [number_of_elements]
\end{verbatim}

The first expression is used to allocate memory to contain one single element of type \texttt{type}. The second one is used to assign a block (an array) of elements of type \texttt{type}, \texttt{where number_of_elements is an integer value representing the amount of these.}

For example:

\begin{verbatim}
int * bobby;
bobby = new int [5];
\end{verbatim}

In this case, the system dynamically assigns space for five elements of type \texttt{int} and returns a pointer to the first element of the sequence, which is assigned to \texttt{bobby}. Therefore, now, \texttt{bobby} points to a valid block of memory with space for five elements of type \texttt{int}. 

• 164 •
The first element pointed by bobby can be accessed either with the expression `bobby[0]` or the expression `*bobby`. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with `bobby[1]` or `*(bobby+1)` and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type `bad_alloc` is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the default method used by `new`, and is the one used in a declaration like:

```cpp
bobby = new int [5]; // if it fails an exception is thrown
```

The other method is known as `nothrow`, and what happens when it is used is that when a memory allocation fails, instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a null pointer, and the program continues its execution.

This method can be specified by using a special object called `nothrow`, declared in header `<new>`, as argument for `new`:
bobby = new (nothrow) int [5];

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if `bobby` took a null pointer value:

```c++
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
    // error assigning memory. Take measures.
}
```

This `nothrow` method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

### 9.2.2 Operators delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator `delete`, whose format is:

```c++
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to `delete` must be either a pointer to a memory block previously allocated with `new`, or a null pointer (in the case of a null pointer, `delete` produces no effect).

```c++
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
    cout << "How many numbers would you like to type? 5\n";
    cout << "Enter number : 75\n";
    cout << "Enter number : 436\n";
    cout << "Enter number : 1067\n";
    cout << "Enter number : 8\n";
    cout << "Enter number : 32\n";

    return 0;
}
```
int i,n;

int * p;

You have entered: 75, 436, 1067, 8, 32,

cout << "How many numbers would you like to type? ";
cin >> i;
p= new (nothrow) int[i];
if (p == 0)
   cout << "Error: memory could not be allocated";
else
{
   for (n=0; n<i; n++)
   {
      cout << "Enter number: ";
cin >> p[n];
   }
   cout << "You have entered: ";
   for (n=0; n<i; n++)
   {
      cout << p[n] << ", ";
   }
delete[] p;
}
return 0;

Notice how the value within brackets in the `new` statement is a variable value entered by the user (i), not a constant value:
P= new (nothrow) int[i];

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the
value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

9.2.3 Dynamic memory in ANSI-C

Operators `new` and `delete` are exclusive of C++. They are not available in the C language. But using pure C language and its library, dynamic memory can also be used through the functions `malloc`, `calloc`, `realloc` and `free`, which are also available in C++ including the `<cstdlib>` header file (see `cstdlib` for more info).

The memory blocks allocated by these functions are not necessarily compatible with those returned by `new`, so each one should be manipulated with its own set of functions or operators.
10.1 Memory organization

How memory is organized for a running program

When a program is loaded into memory, it is organized into three areas of memory:

- **text segment** (sometimes also called the code segment) is where the compiled code of the program itself resides.
- **stack segment** is where memory is allocated for automatic variables within functions.
- memory allocated in the **heap segment** remains in existence for the duration of a program.

http://en.wikipedia.org/wiki/Virtual_memory

**Virtual memory**

The program thinks it has a large range of contiguous addresses; but in reality the parts it is currently using are scattered around RAM, and the inactive parts are saved in a disk file.
计算机系统的分层视图
操作系统（Operating system）主要用途是防止硬件被失控的应用程序滥用，和在控制复杂而又广泛不同的低级硬件设备方面，为应用程序提供简单一致的方法。

<table>
<thead>
<tr>
<th>应用程序</th>
</tr>
</thead>
<tbody>
<tr>
<td>操作系统</td>
</tr>
<tr>
<td>处理器</td>
</tr>
</tbody>
</table>

操作系统提供的抽象表示

进程是操作系统对运行程序的一种抽象
虚拟存储器为每个进程提供一个假象，好像每个进程都在度占地使用主存。
每个进程看到的存储器都是一致的，成为虚拟存储器。
文件是字节序列

Process virtual address space
10.2 Variable scope

Storage duration @ running view
is the property of an object that defines the minimum potential lifetime of the storage containing the object. There are three storage durations: static, automatic, and dynamic. Automatic storage is also called stack memory. Heap memory is for dynamic storage.

The storage duration that a variable has depends on how you create the variable.

An object with static storage duration has storage allocated at program startup.
The memory remains in place for the duration of program execution.

An object with automatic storage duration has storage allocated by a function call, and deallocated by the corresponding function return.
An object with dynamic storage duration has storage allocated by a call to an allocation function (operator new) and deallocated by a corresponding call to a deallocation function (operator delete).

Variables have scope @ code view
The scope of a variable is simply that part of your program in which the variable name is valid.
Within a variable’s scope, you can legally refer to it, set its value, or use it in an expression.

Automatic Variables @ code view
between a pair of curly braces. These are called automatic variables and are said to have local scope or block scope.
“in scope” from the point at which it is declared until the end of the block containing its declaration.
“born” when it’s declared and automatically ceases to exist at the end of the block containing the declaration.

Global variables @ code view
Variables declared outside of all blocks and classes are called globals and have global scope.
Globals have static storage duration by default.

10.3 Understanding pointers

Every pointer has a type. This type indicates what kind of object the pointer points to.

<table>
<thead>
<tr>
<th>Pointer Type</th>
<th>Object Type</th>
<th>Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *</td>
<td>int</td>
<td>xp, ip[0], ip[1]</td>
</tr>
<tr>
<td>union uni *</td>
<td>union uni</td>
<td>up</td>
</tr>
</tbody>
</table>

Every pointer has a value. This value is an address of some object of the designated type.

• 172 •
Union data type

```cpp
union union_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    .
    .
} object_names;
```

Unions allow one same portion of memory to be accessed as different data types, .
Its declaration and use is similar to the one of structures but its functionality is totally different:

http://lkml.indiana.edu/hypermail/linux/kernel/0302.2/0485.html
The decision to do so is made by having mk_elfconfig look at the elf machine-type.

Test host endian
```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    union {
        short s;
        char c[2];
    } endian_test;

    endian_test.s = 0x0102;
    if (memcpy(endian_test.c, "\x01\x02", 2) == 0)
        cout << "host_big_endian" << endl;
    else if (memcpy(endian_test.c, "\x02\x01", 2) == 0)
        cout << "host_little_endian" << endl;
    else
        return(-1);
}```
Code Illustrating Use of Pointers in C

...
An algorithm is just the outline or idea behind a program. We express algorithms in *pseudo-code*: something resembling C or Pascal, but with some statements in English rather than within the programming language. It is expected that one could translate each pseudo-code statement to a small number of lines of actual code, easily and mechanically.

The purpose of design of algorithms is obvious: one needs an algorithm in order to write a program.

Analysis of algorithms is less obviously necessary, but has several purposes:

- Analysis can be more reliable than experimentation. If we experiment, we only know the behavior of a program on certain specific test cases, while analysis can give us guarantees about the performance on all inputs.
- It helps one choose among different solutions to problems. As we will see, there can be many different solutions to the same problem. A careful analysis and comparison can help us decide which one would be the best for our purpose, without requiring that all be implemented and tested.
- We can predict the performance of a program before we take the time to write code. In a large project, if we waited until after all the code was written to discover that something runs very slowly, it could be a major disaster, but if we do the analysis first we have time to discover speed problems and work around them.
- By analyzing an algorithm, we gain a better understanding of where the fast and slow parts are, and what to work on or work around in order to speed it up.

11.1 节取自下面这本书的-Ch1。


11.1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? In this chapter, we will answer these questions.

11.1.1 Algorithms

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, one might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the sorting problem:

- **Input:** A sequence of $n$ numbers $\langle a_1, a_2, ..., a_n \rangle$.
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$.

For example, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is called an instance of the sorting problem. In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step), and as a result a large number of good sorting algorithms have been developed. Which algorithm is best for a given application depends on-among
other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, and the kind of storage device to be used: main memory, disks, or tapes.

An algorithm is said to be **correct** if, for every input instance, it halts with the correct output. We say that a correct algorithm **solves** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one. Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled. We shall see an example of this in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we shall be concerned only with correct algorithms.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

**What kinds of problems are solved by algorithms?**

Sorting is by no means the only computational problem for which algorithms have been developed. (You probably suspected as much when you saw the size of this book.) Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. While the solutions to the various problems involved are beyond the scope of this book, ideas from many of the chapters in this book are used in the solution of these biological problems, thereby enabling scientists to accomplish tasks while using resources efficiently. The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.

- The Internet enables people all around the world to quickly access and retrieve large amounts of information. In order to do so, clever algorithms are employed to manage and manipulate this large volume of data. Examples of problems which must be solved include finding good routes on which the data will travel (techniques for solving such problems appear
Electronic commerce enables goods and services to be negotiated and exchanged electronically. The ability to keep information such as credit card numbers, passwords, and bank statements private is essential if electronic commerce is to be used widely. Public-key cryptography and digital signatures (covered in Chapter 31) are among the core technologies used and are based on numerical algorithms and number theory.

In manufacturing and other commercial settings, it is often important to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A candidate for the presidency of the United States may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming, which we shall study in Chapter 29.

While some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas. We also show how to solve many concrete problems in this book, including the following:

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and our goal is to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if we disallow routes that cross over themselves. How do we choose which of all possible routes is the shortest? Here, we model the road map (which is itself a model of the actual roads) as a graph (which we will meet in Chapter 10 and Appendix B), and we wish to find the shortest path from one vertex to another in the graph. We shall see how to solve this problem efficiently in Chapter 24.
We are given a sequence \( \langle A_1, A_2, \ldots, A_n \rangle \) of \( n \) matrices, and we wish to determine their product \( A_1 A_2 A_n \). Because matrix multiplication is associative, there are several legal multiplication orders. For example, if \( n = 4 \), we could perform the matrix multiplications as if the product were parenthesized in any of the following orders: \((A_1(A_2(A_3A_4))), (A_1((A_2A_3)A_4)), ((A_1A_2)(A_3A_4)), ((A_1(A_2A_3))A_4), \) or \(((A_1A_2)A_3)A_4)\). If these matrices are all square (and hence the same size), the multiplication order will not affect how long the matrix multiplications take. If, however, these matrices are of differing sizes (yet their sizes are compatible for matrix multiplication), then the multiplication order can make a very big difference. The number of possible multiplication orders is exponential in \( n \), and so trying all possible orders may take a very long time. We shall see in Chapter 15 how to use a general technique known as dynamic programming to solve this problem much more efficiently.

We are given an equation \( ax \equiv b \pmod{n} \), where \( a, b, \) and \( n \) are integers, and we wish to find all the integers \( x \), modulo \( n \), that satisfy the equation. There may be zero, one, or more than one such solution. We can simply try \( x = 0, 1, \ldots, n - 1 \) in order, but Chapter 31 shows a more efficient method.

We are given \( n \) points in the plane, and we wish to find the convex hull of these points. The convex hull is the smallest convex polygon containing the points. Intuitively, we can think of each point as being represented by a nail sticking out from a board. The convex hull would be represented by a tight rubber band that surrounds all the nails. Each nail around which the rubber band makes a turn is a vertex of the convex hull. (See Figure 33.6 on page 948 for an example.) Any of the \( 2^n \) subsets of the points might be the vertices of the convex hull. Knowing which points are vertices of the convex hull is not quite enough, either, since we also need to know the order in which they appear. There are many choices, therefore, for the vertices of the convex hull. Chapter 33 gives two good methods for finding the convex hull.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but exhibit two characteristics that are common to many interesting algorithms.
There are many candidate solutions, most of which are not what we want. Finding one that we do want can present quite a challenge.

There are practical applications. Of the problems in the above list, shortest paths provides the easiest examples. A transportation firm, such as a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor and fuel costs. Or a routing node on the Internet may need to find the shortest path through the network in order to route a message quickly.

Data structures

This book also contains several data structures. A **data structure** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

Technique

Although you can use this book as a "cookbook" for algorithms, you may someday encounter a problem for which you cannot readily find a published algorithm (many of the exercises and problems in this book, for example!). This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and understand their efficiency.

Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, it is unknown whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete
problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. A small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

It is valuable to know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If you can show that the problem is NP-complete, you can instead spend your time developing an efficient algorithm that gives a good, but not the best possible, solution.

As a concrete example, consider a trucking company with a central warehouse. Each day, it loads up the truck at the warehouse and sends it around to several locations to make deliveries. At the end of the day, the truck must end up back at the warehouse so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by the truck. This problem is the well-known "traveling-salesman problem," and it is NP-complete. It has no known efficient algorithm. Under certain assumptions, however, there are efficient algorithms that give an overall distance that is not too far above the smallest possible. Chapter 35 discusses such "approximation algorithms."

### 11.1.2 Algorithms as a technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (i.e., well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. And memory may be cheap, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. These resources should be used wisely, and algorithms that are
efficient in terms of time or space will help you do so.

**Efficiency**

Algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as **insertion sort**, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, **merge sort**, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort usually has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can be far less significant in the running time than the dependence on the input size $n$. Where merge sort has a factor of $\lg n$ in its running time, insertion sort has a factor of $n$, which is much larger. Although insertion sort is usually faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort's advantage of $\lg n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of one million numbers. Suppose that computer A executes one billion instructions per second and computer B executes only ten million instructions per second, so that computer A is 100 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort $n$ numbers. (Here, $c_1 = 2$.) Merge sort, on the other hand, is programmed for computer B by an average programmer using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions (so that $c_2 = 50$). To sort one million numbers, computer A takes

$$\frac{2 \times (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds},$$

while computer B takes

...
\[
\frac{50 \times 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 100 \text{ seconds}.
\]

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs 20 times faster than computer A! The advantage of merge sort is even more pronounced when we sort ten million numbers: where insertion sort takes approximately 2.3 days, merge sort takes under 20 minutes. In general, as the problem size increases, so does the relative advantage of merge sort.

**Algorithms and other technologies**

The example above shows that algorithms, like computer hardware, are a technology. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary computers in light of other advanced technologies, such as

- hardware with high clock rates, pipelining, and superscalar architectures,
- easy-to-use, intuitive graphical user interfaces (GUIs),
- object-oriented systems, and
- local-area and wide-area networking.

The answer is yes. Although there are some applications that do not explicitly require algorithmic content at the application level (e.g., some simple web-based applications), most also require a degree of algorithmic content on their own. For example, consider a web-based service that determines how to travel from one location to another. (Several such services existed at the time of this writing.) Its implementation would rely on fast hardware, a graphical user interface, wide-area networking, and also possibly on object orientation. However, it would also require algorithms for certain operations, such as finding routes (probably using a shortest-path algorithm), rendering maps, and interpolating addresses.

Moreover, even an application that does not require algorithmic content at the application level relies heavily upon algorithms. Does the application rely on fast hardware? The hardware design used algorithms. Does the application rely on graphical user interfaces? The design of any GUI relies on algorithms. Does the
application rely on networking? Routing in networks relies heavily on algorithms. Was the application written in a language other than machine code? Then it was processed by a compiler, interpreter, or assembler, all of which make extensive use of algorithms. Algorithms are at the core of most technologies used in contemporary computers.

Furthermore, with the ever-increasing capacities of computers, we use them to solve larger problems than ever before. As we saw in the above comparison between insertion sort and merge sort, it is at larger problem sizes that the differences in efficiencies between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

11.2 算法的概念

计算机只是一个计算工具，它本身不能主动帮助我们做任何事情，需要我们告诉它如何进行计算。

程序设计就是要告诉计算机如何进行计算的。这与我们中学时代的数学解题过程是一样的，只不过描述的手段有所变化而已。

算法 is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

具有如下性质:
通用性：对于符合输入数据类型的任意输入数据，都能根据算法进行问题求解，并保证计算结果的正确性。
能行性：算法中每条指令都是基本的，至少能够被人或机器所确定执行。
确定性：每条指令必须有确切的定义，即无二义性。执行一步后，关于下一步如何执行，应该有明确的指示。
有穷性：对于任意一个合法的输入，算法应在有限多步内结束，并给出计算结果。

11.3 算法的三种基本结构

只使用如下三种结构，就可以描述任何算法，且算法结构优良
顺序结构（Sequence）
分支结构（Decision）
循环结构（Repetition）
每一种基本结构分别只有一个入口和一个出口

11.4 算法的表示

算法是让人来理解的，因此需要有效的算法表示机制
流程图（Flowchart）表示法
伪代码（Pseudocode）表达法

11.5 介绍几种基本算法

数值算法
加减乘除、最大最小值、解方程、求微积分
非数值算法
排序、查找、文本处理、流程处理

11.6 迭代与递归

一般来说，有两种途径用于编写解决问题的算法。一种使用迭代，另一种使用递归。递归是算法自我调用的过程。
第12章 程序设计

计算机解决问题的基本思想和方法
POJ 练习题分类

基本技能
基本输入输出，算术逻辑运算，循环，数组，指针，函数

基本应用
进制转换，字符串处理，时间和日期处理，高精度计算

数据结构
链表，二叉树

基本算法
简单计算题，排序，枚举，递归，计算机模拟，动态规划

12.1 简单计算题

基本过程包括
- 将一个用自然语言描述的实际问题抽象成一个计算问题
- 给出计算过程，并编程实现
- 将计算结果还原成对原来问题的解答

关键是读懂问题
- 搞清输入和输出数据的含义及给出的格式
通过输入输出样例验证自己的理解是否正确

棋盘走子步数

12.2 模拟

现实中的有些问题，难以找到公式或规律来进行解决，只能按照一定步骤，不停地做下去，最后才能得到答案。

这样的问题，用计算机来解决十分合适，只要让计算机模拟人在解决问题的
行为即可。
猴子选大王(约瑟夫问题)

12.3 可模型化的问题

现实问题=>（模型）=>程序
例题：某国为了防御敌国的导弹袭击，开发出一种导弹拦截系统。它有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭，并观测到导弹依次飞来的高度。
问：这套系统最多能拦截多少导弹。
约束：拦截来袭导弹时，必须按来袭导弹袭击的时间顺序，不许先拦截后面的导弹，再拦截前面的。

拦截导弹计数

12.4 动态规划

基本思想：划分成子问题，保留中间结果
应用条件
最优子结构性质：问题的最优解所包含的子问题的解也是最优的，
子问题重叠性质：每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。保存中间计算结果而非重复计算。

动态规划的实质
记忆化搜索，就是用搜索解决问题时，将会重复计算的值，算好一次后就存起来，以后不必重新计算，用空间换时间。

Dynamic Programming: From novice to advanced, By Dumitru
http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg
An important part of given problems can be solved with the help of dynamic programming (DP for short). Being able to tackle problems of this type would greatly increase your skill. I will try to help you in understanding how to solve problems using DP. The article is based on examples, because a raw theory is very hard to understand.
Note: If you're bored reading one section and you already know what's being discussed in it - skip it and go to the next one.
Introduction (Beginner)

What is a dynamic programming, how can it be described?

A DP is an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

Now let's see the base of DP with the help of an example:

Given a list of N coins, their values (V_1, V_2, ..., V_N), and the total sum S. Find the minimum number of coins the sum of which is S (we can use as many coins of one type as we want), or report that it's not possible to select coins in such a way that they sum up to S.

Now let's start constructing a DP solution:

First of all we need to find a state for which an optimal solution is found and with the help of which we can find the optimal solution for the next state.

What does a "state" stand for?

It's a way to describe a situation, a sub-solution for the problem. For example a state would be the solution for sum \( i \), where \( i \leq S \). A smaller state than state \( i \) would be the solution for any sum \( j \), where \( j < i \). For finding a state \( i \), we need to first find all smaller states \( j \) (\( j < i \)). Having found the minimum number of coins which sum up to \( i \), we can easily find the next state - the solution for \( i+1 \).

How can we find it?

It is simple - for each coin \( j \), \( V_j \leq i \), look at the minimum number of coins found for the \( i-V_j \) sum (we have already found it previously). Let this number be \( m \). If \( m+1 \) is less than the minimum number of coins already found for current sum \( i \), then we write the new result for it.

For a better understanding let's take this example:

Given coins with values 1, 3, and 5.
And the sum \( S \) is set to be 11.
First of all we mark that for state 0 (sum 0) we have found a solution with a minimum number of 0 coins. We then go to sum 1. First, we mark that we haven't yet found a solution for this one (a value of Infinity would be fine). Then we see that only coin 1 is less than or equal to the current sum. Analyzing it, we see that for sum 1-V₁= 0 we have a solution with 0 coins. Because we add one coin to this solution, we'll have a solution with 1 coin for sum 1. It's the only solution yet found for this sum. We write (save) it. Then we proceed to the next state - sum 2. We again see that the only coin which is less or equal to this sum is the first coin, having a value of 1. The optimal solution found for sum (2-1) = 1 is coin 1. This coin 1 plus the first coin will sum up to 2, and thus make a sum of 2 with the help of only 2 coins. This is the best and only solution for sum 2. Now we proceed to sum 3. We now have 2 coins which are to be analyzed - first and second one, having values of 1 and 3. Let's see the first one. There exists a solution for sum 2 (3 - 1) and therefore we can construct from it a solution for sum 3 by adding the first coin to it. Because the best solution for sum 2 that we found has 2 coins, the new solution for sum 3 will have 3 coins. Now let's take the second coin with value equal to 3. The sum for which this coin needs to be added to make 3, is 0. We know that sum 0 is made up of 0 coins. Thus we can make a sum of 3 with only one coin - 3. We see that it's better than the previous found solution for sum 3, which was composed of 3 coins. We update it and mark it as having only 1 coin. The same we do for sum 4, and get a solution of 2 coins - 1+3. And so on.

**Pseudocode:**

Set Min[i] equal to Infinity for all of i
Min[0]=0

For i = 1 to S
For j = 0 to N - 1
  If (Vj<=i AND Min[i-Vj]+1<Min[i])
  Then Min[i]=Min[i-Vj]+1

Output Min[S]

Here are the solutions found for all sums:

<table>
<thead>
<tr>
<th>Sum</th>
<th>Min. nr. of coins</th>
<th>Coin value added to a smaller sum to</th>
</tr>
</thead>
</table>

• 189 •
obtain this sum (it is displayed in brackets)

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 (0)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1 (1)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3 (0)</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1 (3)</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5 (0)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3 (3)</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1 (6)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3 (5)</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1 (8)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>5 (5)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1 (10)</td>
</tr>
</tbody>
</table>

As a result we have found a solution of 3 coins which sum up to 11.

Additionally, by tracking data about how we got to a certain sum from a previous one, we can find what coins were used in building it. For example: to sum 11 we got by adding the coin with value 1 to a sum of 10. To sum 10 we got from 5. To 5 - from 0. This way we find the coins used: 1, 5 and 5.

Having understood the basic way a DP is used, we may now see a slightly different approach to it. It involves the change (update) of best solution yet found for a sum i, whenever a better solution for this sum was found. In this case the states aren't calculated consecutively. Let's consider the problem above. Start with having a solution of 0 coins for sum 0. Now let's try to add first coin (with value 1) to all sums already found. If the resulting sum will be composed of fewer coins than the one previously found - we'll update the solution for it. Then we do the same thing for the second coin, third coin, and so on for the rest of them. For example, we first add coin 1 to sum 0 and get sum 1. Because we haven't yet found a possible way to make a sum of 1 - this is the best solution yet found, and we mark $S[1]=1$. By adding the same coin to sum 1, we'll
get sum 2, thus making $S[2]=2$. And so on for the first coin. After the first coin is processed, take coin 2 (having a value of 3) and consecutively try to add it to each of the sums already found. Adding it to 0, a sum 3 made up of 1 coin will result. Till now, $S[3]$ has been equal to 3, thus the new solution is better than the previously found one. We update it and mark $S[3]=1$. After adding the same coin to sum 1, we'll get a sum 4 composed of 2 coins. Previously we found a sum of 4 composed of 4 coins; having now found a better solution we update $S[4]$ to 2. The same thing is done for next sums - each time a better solution is found, the results are updated.

**Elementary**

To this point, very simple examples have been discussed. Now let's see how to find a way for passing from one state to another, for harder problems. For that we will introduce a new term called recurrent relation, which makes a connection between a lower and a greater state.

Let's see how it works:


As described above we must first find how to define a "state" which represents a sub-problem and thus we have to find a solution for it. Note that in most cases the states rely on lower states and are independent from greater states.

Let's define a state $i$ as being the longest non-decreasing sequence which has its last number $A[i]$ . This state carries only data about the length of this sequence. Note that for $i<j$ the state $i$ is independent from $j$, i.e. doesn't change when we calculate state $j$.

Let's see now how these states are connected to each other. Having found the solutions for all states lower than $i$, we may now look for state $i$. At first we initialize it with a solution of 1, which consists only of the $i$-th number itself. Now for each $j<i$ let's see if it's possible to pass from it to state $i$. This is possible only when $A[j] \leq A[i]$ , thus keeping (assuring) the sequence non-decreasing. So if $S[j]$ (the solution found for state $j$) + 1 (number $A[i]$ added to this sequence which ends with number $A[j]$ ) is better than a solution found for $i$ (ie. $S[j]+1>S[i]$ ), we make $S[i]=S[j]+1$. This way we
consecutively find the best solutions for each \( i \), until last state \( N \).

Let's see what happens for a randomly generated sequence: 5, 3, 4, 8, 6, 7:

<table>
<thead>
<tr>
<th>( i )</th>
<th>The length of the longest non-decreasing sequence of first ( i ) numbers</th>
<th>The last sequence ( i ) from which we &quot;arrived&quot; to this one</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 (first number itself)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2 (second number itself)</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Practice problem:**

Given an undirected graph \( G \) having \( N \) (\( 1 < N \leq 1000 \)) vertices and positive weights. Find the shortest path from vertex 1 to vertex \( N \), or state that such path doesn't exist.

Hint: At each step, among the vertices which weren't yet checked and for which a path from vertex 1 was found, take the one which has the shortest path, from vertex 1 to it, yet found.

Try to solve the following problems from TopCoder competitions:

- **ZigZag** - 2003 TCCC Semifinals 3
- **BadNeighbors** - 2004 TCCC Round 4
- **FlowerGarden** - 2004 TCCC Round 1

**Intermediate**

Let's see now how to tackle bi-dimensional DP problems.
**Problem:**
A table composed of $N \times M$ cells, each having a certain quantity of apples, is given. You start from the upper-left corner. At each step you can go down or right one cell. Find the maximum number of apples you can collect.

This problem is solved in the same way as other DP problems; there is almost no difference.

First of all we have to find a state. The first thing that must be observed is that there are at most 2 ways we can come to a cell - from the left (if it's not situated on the first column) and from the top (if it's not situated on the most upper row). Thus to find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell.

From above, a recurrent relation can be easily obtained:

$$S[i][j] = A[i][j] + \max(S[i-1][j], \text{if } i > 0 ; S[i][j-1], \text{if } j > 0)$$

(where $i$ represents the row and $j$ the column of the table, its left-upper corner having coordinates $\{0,0\}$; and $A[i][j]$ being the number of apples situated in cell $i,j$).

$S[i][j]$ must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.

**Pseudocode:**

```
For i = 0 to N - 1
    For j = 0 to M - 1
        S[i][j] = A[i][j] +
                   max(S[i][j-1], if j>0 ; S[i-1][j], if i>0 ; 0)

Output S[n-1][m-1]
```

Here are a few problems, from TopCoder Competitions, for practicing:

- **AvoidRoads** - 2003 TCO Semifinals 4
- **ChessMetric** - 2003 TCCC Round 4
Upper-Intermediate

This section will discuss about dealing DP problems which have an additional condition besides the values that must be calculated.

As a good example would serve the following problem:

Given an undirected graph $G$ having positive weights and $N$ vertices.

You start with having a sum of $M$ money. For passing through a vertex $i$, you must pay $S[i]$ money. If you don't have enough money - you can't pass through that vertex. Find the shortest path from vertex 1 to vertex $N$, respecting the above conditions; or state that such path doesn't exist. If there exist more than one path having the same length, then output the cheapest one. Restrictions: $1 < N <= 100$; $0 <= M <= 100$; for each $i$, $0 <= S[i] <= 100$. As we can see, this is the same as the classical Dijkstra problem (finding the shortest path between two vertices), with the exception that it has a condition. In the classical Dijkstra problem we would have used a uni-dimensional array $Min[i]$, which marks the length of the shortest path found to vertex $i$. However in this problem we should also keep information about the money we have. Thus it would be reasonable to extend the array to something like $Min[i][j]$, which represents the length of the shortest path found to vertex $i$, with $j$ money being left. In this way the problem is reduced to the original path-finding algorithm. At each step we find the unmarked state $(i,j)$ for which the shortest path was found. We mark it as visited (not to use it later), and for each of its neighbors we look if the shortest path to it may be improved. If so - then update it. We repeat this step until there will remain no unmarked state to which a path was found. The solution will be represented by $Min[N-1][j]$ having the least value (and the greatest $j$ possible among the states having the same value, i.e. the shortest paths to which has the same length).

Pseudocode:

Set states$(i,j)$ as unvisited for all $(i,j)$
Set $Min[i][j]$ to Infinity for all $(i,j)$

$Min[0][M]=0$

While(TRUE)

Among all unvisited states$(i,j)$ find the one for which $Min[i][j]$ is the smallest. Let this
state found be (k,l).

If there wasn't found any state (k,l) for which Min[k][l] is less than Infinity - exit While loop.

Mark state(k,l) as visited

For All Neighbors p of Vertex k.
  If (l-S[p]>=0 AND
      Min[p][l-S[p]]>Min[k][l]+Dist[k][p])
    Then Min[p][l-S[p]]=Min[k][l]+Dist[k][p]
    i.e.
    If for state(i,j) there are enough money left for going to vertex p (l-S[p] represents the money that will remain after passing to vertex p), and the shortest path found for state(p,l-S[p]) is bigger than [the shortest path found for state(k,l)] + [distance from vertex k to vertex p]), then
    set the shortest path for state(i,j) to be equal to this sum.
End For

End While

Find the smallest number among Min[N-1][j] (for all j, 0<=j<=M);
if there are more than one such states, then take the one with greater j. If there are no states(N-1,j) with value less than Infinity – then such a path doesn't exist.

Here are a few TC problems for practicing:

- Jewelry - 2003 TCO Online Round 4
- StripePainter - SRM 150 Div 1
- QuickSums - SRM 197 Div 2
- ShortPalindromes - SRM 165 Div 2

**Advanced**

The following problems will need some good observations in order to reduce them to a dynamic solution.
Problem StarAdventure - SRM 208 Div 1:

Given a matrix with $M$ rows and $N$ columns ($N \times M$). In each cell there's a number of apples.

You start from the upper-left corner of the matrix. You can go down or right one cell. You need to arrive to the bottom-right corner. Then you need to go back to the upper-left cell by going each step one cell left or up. Having arrived at this upper-left cell, you need to go again back to the bottom-right cell.

Find the maximum number of apples you can collect.

When you pass through a cell - you collect all the apples left there.

Restrictions: $1 < N, M \leq 50$ ; each cell contains between 0 and 1000 apples inclusive.

First of all we observe that this problem resembles to the classical one (described in Section 3 of this article), in which you need to go only once from the top-left cell to the bottom-right one, collecting the maximum possible number of apples. It would be better to try to reduce the problem to this one. Take a good look into the statement of the problem - what can be reduced or modified in a certain way to make it possible to solve using DP? First observation is that we can consider the second path (going from bottom-right cell to the top-left cell) as a path which goes from top-left to bottom-right cell. It makes no difference, because a path passed from bottom to top, may be passed from top to bottom just in reverse order. In this way we get three paths going from top to bottom. This somehow decreases the difficulty of the problem. We can consider these 3 paths as left, middle and right. When 2 paths intersect (like in the figure below)
we may consider them as in the following picture, without affecting the result:

![Diagram](image)

This way we'll get 3 paths, which we may consider as being one left, one middle and
the other - right. More than that, we may see that for getting an optimal results they
must not intersect (except in the leftmost upper corner and rightmost bottom corner).
So for each row \( y \) (except first and last), the \( x \) coordinates of the lines \( (x_1[y], x_2[y] \)
and respectively \( x_3[y] \)) will be: \( x_1[y] < x_2[y] < x_3[y] \). Having done that - the DP
solution now becomes much clearer. Let's consider the row \( y \). Now suppose that for
any configuration of \( x_1[y-1], x_2[y-1] \) and \( x_3[y-1] \) we have already found the paths
which collect the maximum number of apples. From them we can find the optimal
solution for row \( y \). We now have to find only the way for passing from one row to the
next one. Let \( \text{Max}[i][j][k] \) represent the maximum number of apples collected till row
\( y-1 \) inclusive, with three paths finishing at column \( i, j \), and respectively \( k \). For the next
row \( y \), add to each \( \text{Max}[i][j][k] \) (obtained previously) the number of apples situated in
cells \( (y,i), (y,j) \) and \( (y,k) \). Thus we move down at each step. After we made such a
move, we must consider that the paths may move in a row to the right. For keeping the
paths out of an intersection, we must first consider the move to the right of the left path,
after this of the middle path, and then of the right path. For a better understanding think
about the move to the right of the left path - take every possible pair of, \( k \) (where \( j<k \)),
and for each \( i \) (1 \( i<j \)) consider the move from position \( (i-1,j,k) \) to position \( (i,j,k) \).
Having done this for the left path, start processing the middle one, which is done
similarly; and then process the right path.

TC problems for practicing:

- **MiniPaint** - SRM 178 Div 1
Additional Note:
When have read the description of a problem and started to solve it, first look at its restrictions. If a polynomial-time algorithm should be developed, then it's possible that the solution may be of DP type. In this case try to see if there exist such states (sub-solutions) with the help of which the next states (sub-solutions) may be found. Having found that - think about how to pass from one state to another. If it seems to be a DP problem, but you can't define such states, then try to reduce the problem to another one (like in the example above, from Section 5).

Mentioned in this writeup:
TCCC '03 Semifinals 3 Div I Easy - ZigZag
TCCC '04 Round 4 Div I Easy - BadNeighbors
TCCC '04 Round 1 Div I Med - FlowerGarden
TCO '03 Semifinals 4 Div I Easy - AvoidRoads
TCCC '03 Round 4 Div I Easy - ChessMetric
TCO '03 Round 4 Div I Med - Jewelry
SRM 150 Div I Med - StripePainter
SRM 197 Div II Hard - QuickSums
SRM 165 Div II Hard - ShortPalindromes
SRM 208 Div I Hard - StarAdventure
SRM 178 Div I Hard - MiniPaint
参考文献


http://www.cplusplus.com/
http://www.cprogramming.com/