Notes of computers

PhilippeMENG

2021 年 2 月 16 日

目录

1 C 语言入门 **5**

1.1 程序的基本概念 5

1.2 自然语言和编程语言 9

1.3 程序的调试 12

1.4 第一个程序 14

1.4.1 C 语言标准 18

1.5 常量 21

1.6 变量 22

1.6.1 声明和定义 24

1.7 赋值 24

2 Lisp 语言入门 **27**

2.1 List Processing 27

2.1.1 Lisp Lists 27

2.1.2 Lisp Atoms 28

2.1.3 Whitespace in Lists 30

2.1.4 Run a Program 31

2.1.5 Generate an Error Message 32

2.1.6 Symbol Names and Function Definitions 34

2.1.7 The Lisp Interpreter 35

3 Emacs 入门 39

3.1 Understanding Files and Buffers 39

3.2 A Word About Modes 39

3.3 Starting Emacs 42

3.4 About the Emacs Display 43

Chapter 1

C 语言入门

1.1 程序的基本概念

程序（Program）告诉计算机应如何完成一个计算任务，这里的计算可以是数学运算，比如解方程，也可以是符号运算，比如查找和替换文档中的某个单词。从根本上说，计算机是由数字电路组成的运算机器，只能对数字做运算，程序之所以能做符号运算，是因为符号在计算机内部也是用数字表示的。此外，程序还可以处理声音和图像，声音和图像在计算机内部必然也是用数字表示的，这些数字经过专门的硬件设备转换成人可以听到、看到的声音和图像。

程序由一系列指令（Instruction）组成，指令是指示计算机做某种运算的命令，通常包括以下几类：

输入（Input）从键盘、文件或者其它设备获取数据。

输出（Output）把数据显示到屏幕，或者存入一个文件，或者发送到其它设备。

基本运算执行最基本的数学运算（加减乘除）和数据存取。

测试和分支测试某个条件，然后根据不同的测试结果执行不同的后续指令。

循环重复执行一系列操作。

对于程序来说，有上面这几类指令就足够了。你曾用过的任何一个程序，不管它有多么复杂，都是由这几类指令组成的。程序是那么的复杂，而编写程序可以用的指令却只有这么简单的几种，这中间巨大的落差就要由程序员去填了，所以编写程序理应是一件相当复杂的工作。编写程序可以说就是这样一个过程：把复杂的任务分解成子任务，把子任务再分解成更简单的任务，层层分解，直到最后简单得可以用以上指令来完成。

编程语言（Programming Language）分为低级语言（Low-level Language）和高级语言（Highlevel Language）。机器语言（Machine Language）和汇编语言（Assembly Language）属于低级语言，**直接用计算机指令编写程序**。而 C、C++、Java、Python 等属于高级语言，**用语句（Statement）编写程序，语句是计算机指令的抽象表示**。举个例子，同样一个语句用 C 语言、汇编语言和机器语言分别表示如下：

编程语言	表示形式
C语言	a=b+1;

汇编语言	mov 0x804a01c,%eax
	add $0x1,%eax
	mov %eax,0x804a018

机器语言	a1 1c a0 04 08 83 c0 01 a3 18 a0 04 08

计算机只能对数字做运算，符号、声音、图像在计算机内部都要用数字表示，指令也不例外，上表中的机器语言完全由**十六进制**数字组成。最早的程序员都是直接用机器语言编程，但是很麻烦，需要查大量的表格来确定每个数字表示什么意思，编写出来的程序很不直观，而且容易出错，于是有

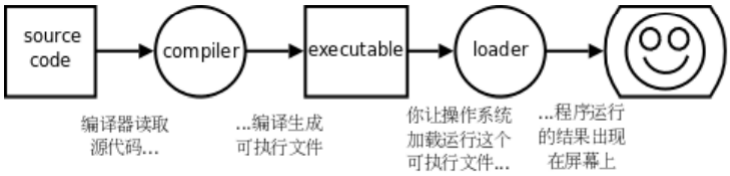
了汇编语言，把机器语言中一组一组的数字用**助记符（Mnemonic）**表示，直接用这些助记符写出汇编程序，然后让**汇编器（Assembler）**去查表把助记符替换成数字，也就把汇编语言翻译成了机器语言。从上面的例子可以看出，汇编语言和机器语言的指令是一一对应的，汇编语言有三条指令，机器语言也有三条指令，汇编器就是做一个简单的替换工作，例如在第一条指令中，把 `mov ? ,%eax` 这种格式的指令替换成机器码 `a1 ?`，? 表示一个地址，在汇编指令中是 `0x804a01c`，转换成机器码之后是 `1c a0 04 08`（这是指令中的十六进制数的**小端表示**）

从上面的例子还可以看出，**C 语言的语句和低级语言的指令之间不是简单的一一对应关系，一条 `a=b+1;` 语句要翻译成三条汇编或机器指令，这个过程称为编译（Compile），由编译器（Compiler）来完成，显然编译器的功能比汇编器要复杂得多**。用 C 语言编写的程序必须经过编译转成机器指令才能被计算机执行，编译需要花一些时间，这是用高级语言编程的一个缺点，然而更多的是优点。首先，用 C 语言编程更容易，写出来的代码更紧凑，可读性更强，出了错也更容易改正。其次，C 语言是可移植的（Portable）或者称为平台无关的（Platform Independent）。

平台这个词有很多种解释，可以指**计算机体系结构**（Architecture），也可以指**操作系统**（Operating System），也可以指**开发平台（编译器、链接器等）**。不同的计算机体系结构有不同的**指令集**（Instruction Set），可以识别的机器指令格式是不同的，**直接用某种体系结构的汇编或机器指令写出来的程序只能在这种体系结构的计算机上运行，然而各种体系结构的计算机都有各自的 C 编译器，可以把 C 程序编译成各种不同体系结构的机器指令，这意味着用 C 语言写的程序只需稍加修改甚至不用修改就可以在各种不同的计算机上编译运行**。各种高级语言都具有 C 语言的这些优点，所以绝大部分程序是用高级语言编写的，**只有和硬件关系密切的少数程序（例如驱动程序）才会用到低级语言**。还要注意一点，即使在相同的体系结构和操作系统下，用不同的 C 编译器（或者同一个 C 编译器的不同版本）编译同一个程序得到的结果也有可能不同，C 语言有些语法特性在 C 标准中并没有明确规定，各编译器有不同的实现，编译出来的指令的行为特性也会

不同，应该尽量避免使用不可移植的语法特性。

总结一下编译执行的过程，首先你用文本编辑器写一个 C 程序，然后保存成一个文件，例如 program.c（通常 C 程序的文件名后缀是.c），这称为**源代码（Source Code）或源文件**，然后运行编译器对它进行编译，**编译的过程并不执行程序，而是把源代码全部翻译成机器指令，再加上一些描述信息，生成一个新的文件**，例如 a.out，这称为**可执行文件**，可执行文件可以被操作系统加载运行，计算机执行该文件中由编译器生成的指令，如下图所示：



有些高级语言以**解释（Interpret** 的方式执行，**解释执行过程和 C 语言的编译执行过程很不一样**。例如编写一个 Shell 脚本 script.sh，内容如下：

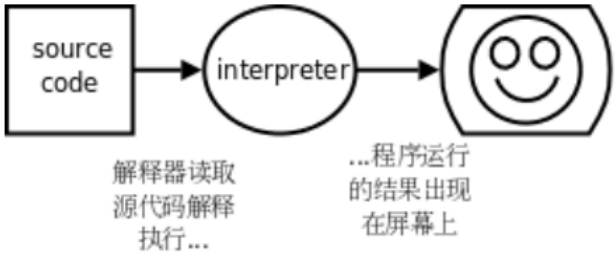
```
#! /bin/sh
VAR=1
VAR=$(($VAR+1))
echo $VAR
```

定义 Shell 变量 VAR 的初始值是 1，然后自增 1，然后打印 VAR 的值。用 Shell 程序/bin/sh 解释执行这个脚本，结果如下：

```
/bin/sh script.sh
2
```

这里的/bin/sh 称为**解释器（Interpreter），它把脚本中的每一行当作一条命令解释执行，而不需要先生成包含机器指令的可执行文件再执行**。如果把脚本中的这三行当作三条命令直接敲到 Shell 提示符下，也能得到同样的结果：

```
VAR=1
VAR=$(($VAR+1))
echo $VAR
2
```



编程语言仍在发展演化。以上介绍的机器语言称为第一代语言（1GL，1st Generation Programming Language），汇编语言称为第二代语言（2GL，2nd Generation Programming Language），C、C++、Java、Python 等可以称为第三代语言（3GL，3rd Generation Programming Language）。目前已经有了 4GL（4th Generation Programming Language）和 5GL（5th Generation Programming Language）的概念。

3GL 的编程语言虽然是用语句编程而不直接用指令编程，但语句也分为输入、输出、基本运算、测试分支和循环等几种，和指令有直接的对应关系。而 4GL 以后的编程语言更多是描述要做什么（Declarative）而不描述具体一步一步怎么做（Imperative），具体一步一步怎么做完全由编译器或解释器决定 SQL 语言（SQL，Structured Query Language，结构化查询语言）就是这样的例子。

1.2 自然语言和编程语言

自然语言（Natural Language）就是人类讲的语言，比如汉语、英语和法语。这类语言不是人为设计（虽然有人试图强加一些规则）而是自然进化

的。形式语言（Formal Language）是为了特定应用而人为设计的语言，例如数学家用的数字和运算符号。编程语言也是一种形式语言，是专门设计用来表达计算过程的形式语言。形式语言有严格的语法（Syntax）规则，例如，3+3=6 是一个语法正确的数学等式，而 3=+6$ 则不是。

语法规则是由符号（Token）和结构（Structure）的规则所组成的。

Token 的概念相当于自然语言中的单词和标点、数学式中的数和运算符，例如 3=+6$ 的问题之一在于 $ 不是一个合法的数也不是一个事先定义好的运算符。结构是指 Token 的排列方式，3=+6$ 还有一个结构上的错误，虽然加号和等号都是合法的运算符，但是不能在等号之后紧跟加号。关于 Token 的规则称为词法（Lexical）规则，而关于结构的规则称为语法（Grammar）规则。

当阅读一个自然语言的句子或者一种形式语言的语句时，你不仅要搞清楚每个词（Token）是什么意思，而且必须搞清楚整个句子的结构是什么样的（在自然语言中你只是没有意识到，但确实这样做了，尤其是在读外语时你肯定也意识到了）。这个分析句子结构的过程称为解析（Parse）。例如，当你听到 “The other shoe fell.” 这个句子时，你理解 the other shoe 是主语而 fell 是谓语动词，一旦解析完成，你就搞懂了句子的意思，如果知道 shoe 是什么东西，fall 意味着什么，这句话是在什么上下文（Context）中说的，你还能理解这个句子主要暗示的内容，这些都属于语义（Semantic）的范畴。

虽然形式语言和自然语言有很多共同之处，包括 Token、结构和语义，但是也有很多不一样的地方。

歧义性（Ambiguity）

自然语言充满歧义，人们通过上下文的线索和自己的常识来解决这个问题。形式语言的设计要求是清晰的、毫无歧义的，这意味着每个语句都必须有确切的含义而不管上下文如何。

冗余性（Redundancy）

为了消除歧义减少误解，自然语言引入了相当多的冗余。结果是自然语言经常说得啰里啰嗦，而形式语言则更加紧凑，极少有冗余。

与字面意思的一致性

自然语言充斥着成语和隐喻（Metaphor），我在某种场合下说“The other shoe fell”，可能并不是说谁的鞋掉了。而形式语言中字面（Literal）意思基本上就是真实意思，也会有一些例外，例如 C 语言转义序列，但即使有例外也会明确规定哪些字面意思不是真实意思，它们所表示的真实意思又是什么。

说自然语言长大的人（实际上没有人例外），往往有一个适应形式语言的困难过程。某种意义上，形式语言和自然语言之间的不同正像诗歌和说明文的区别，当然，前者之间的区别比后者更明显：

诗歌

词语的发音和意思一样重要，全诗作为一个整体创造出一种效果或者表达一种感情。歧义和非字面意思不仅是常见的而且是刻意使用的。

说明文

词语的字面意思显得更重要，并且结构能传达更多的信息。诗歌只能看一个整体，而说明文更适合逐字句分析，但仍然充满歧义。

程序

计算机程序是毫无歧义的，字面和本意高度一致，能够完全通过对 Token 和结构的分析加以理解。

现在给出一些关于阅读程序（包括其它形式语言）的建议。首先请记住形式语言远比自然语言紧凑，所以要多花点时间来读。其次，结构很重要，从上到下从左到右读往往不是一个好办法，而应该学会在大脑里解析：识别 Token，分解结构。最后，请记住细节的影响，诸如拼写错误和标点错误这些在自然语言中可以忽略的小毛病会把形式语言搞得面目全非。

很不幸，Syntax 和 Grammar 通常都翻译成“语法”，这让初学者非常混乱，Syntax 的含义其实包含了 Lexical 和 Grammar 的规则，还包含一部分语义的规则，例如在 C 程序中变量应先声明后使用。即使在英文的文献中 Syntax 和 Grammar 也常混用，在有些文献中 Syntax 的含义不包括 Lexical 规则，只要注意上下文就不会误解。另外，本书在翻译容易引起混淆的时候通常直接用英文名称，例如 Token 没有十分好的翻译，直接用英

文名称。

1.3 程序的调试

编程是一件复杂的工作，因为是人做的事情，所以难免经常出错。据说有这样一个典故：早期的计算机体积都很大，有一次一台计算机不能正常工作，工程师们找了半天原因最后发现是一只臭虫钻进计算机中造成的。从此以后，**程序中的错误被叫做臭虫（Bug），而找到这些 Bug 并加以纠正的过程就叫做调试（Debug）**。有时候调试是一件非常复杂的工作，要求程序员概念明确、逻辑清晰、性格沉稳，还需要一点运气。调试的技能我们在后续的学习中慢慢培养，但首先我们要区分清楚程序中的 Bug 分为哪几类。

编译时错误

编译器只能翻译语法正确的程序，否则将导致编译失败，无法生成可执行文件。对于自然语言来说，一点语法错误不是很严重的问题，因为我们仍然可以读懂句子。而编译器就没那么宽容了，只要有哪怕一个很小的语法错误，编译器就会输出一条错误提示信息然后罢工，你就得不到你想要的结果。虽然大部分情况下编译器给出的错误提示信息就是你出错的代码行，但也有个别时候编译器给出的错误提示信息帮助不大，甚至会误导你。在开始学习编程的前几个星期，你可能会花大量的时间来纠正语法错误。等到有了一些经验之后，还是会犯这样的错误，不过会少得多，而且你能更快地发现错误原因。等到经验更丰富之后你就会觉得，语法错误是最简单最低级的错误，编译器的错误提示也就那么几种，即使错误提示是有误导的也能够立刻找出真正的错误原因是什么。相比下面两种错误，语法错误解决起来要容易得多。

运行时错误

编译器检查不出这类错误，仍然可以生成可执行文件，但在运行时会出错而导致程序崩溃。对于我们接下来的几章将编写的简单程序来说，运行时错误很少见，到了后面的章节你会遇到越来越多的运行时错误。读者

在以后的学习中要时刻注意区分编译时和**运行时（Runtime）**这两个概念，不仅在调试时需要区分这两个概念，在学习 C 语言的很多语法时都需要区分这两个概念，有些事情在编译时做，有些事情则在运行时做。

逻辑错误和语义错误

第三类错误是逻辑错误和语义错误。如果程序里有逻辑错误，编译和运行都会很顺利，看上去也不产生任何错误信息，但是程序没有干它该干的事情，而是干了别的事情。当然不管怎么样，计算机只会按你写的程序去做，问题在于你写的程序不是你真正想要的，这意味着程序的意思（即语义）是错的。找到逻辑错误在哪需要十分清醒的头脑，要通过观察程序的输出回过头来判断它到底在做什么。

最重要的技巧之一就是调试。调试的过程可能会让你感到一些沮丧，但调试也是编程中最需要动脑的、最有挑战和乐趣的部分。从某种角度看调试就像侦探工作，根据掌握的线索来推断是什么原因和过程导致了你所看到的结果。调试也像是一门实验科学，每次想到哪里可能有错，就修改程序然后再试一次。如果假设是对的，就能得到预期的正确结果，就可以接着调试下一个 Bug，一步一步逼近正确的程序；如果假设错误，只好另外再找思路再做假设。“当你把不可能的全部剔除，剩下的——即使看起来再怎么不可能——就一定是事实。”（即使你没看过福尔摩斯也该看过柯南吧）。
也有一种观点认为，编程和调试是一回事，编程的过程就是逐步调试直到获得期望的结果为止。你应该总是从一个能正确运行的小规模程序开始，每做一步小的改动就立刻进行调试，这样的好处是总有一个正确的程序做参考：如果正确就继续编程，如果不正确，那么一定是刚才的小改动出了问题。例如，Linux 操作系统包含了成千上万行代码，但它也不是一开始就规划好了内存管理、设备管理、文件系统、网络等等大的模块，一开始它仅仅是 Linus Torvalds 用来琢磨 Intel 80386 芯片而写的小程序。据 Larry Greenfield 说，“Linus 的早期工程之一是编写一个交替打印 AAAA 和 BBBB 的程序，这玩意儿后来进化成了 Linux。”（引自 The Linux User’s Guide Beta1 版）在后面的章节中会给出更多关于调试和编程实践的建议。

1.4 第一个程序

通常一本教编程的书中第一个例子都是打印 “Hello, World.”，这个传统源自 [K&R]，用 C 语言写这个程序可以这样写：

Hello World

```
#include <stdio.h>
/* main: generate some simple output */
int main(void)
{
^^Iprintf("Hello, world.\n");
^^Ireturn 0;
}
```

将这个程序保存成 main.c，然后编译执行：

```
$ gcc main.c
$ ./a.out
Hello, world.
```

gcc 是 Linux 平台的 C 编译器，编译后在当前目录下生成可执行文件 a.out，直接在命令行输入这个可执行文件的路径就可以执行它。如果不想把文件名叫 a.out，可以用 gcc 的-o 参数自己指定文件名：

```
$ gcc main.c -o main
$ ./main
Hello, world.
```

虽然这只是一个很小的程序，但我们目前暂时还不具备相关的知识来完全理解这个程序，比如程序的第一行，还有程序主体的 `int main(void){...return 0;}` 结构，这些部分我们暂时不详细解释，现在只需要把它们看成是**每个程序按惯例必须要写的部分（Boilerplate）**。但要注意 **main 是一个特殊的**

名字，C 程序总是从 main 里面的第一条语句开始执行的，在这个程序中是指 printf 这条语句。

第 3 行的`/* ... */`结构是一个注释（Comment），其中可以写一些描述性的话，解释这段程序在做什么。注释只是写给程序员看的，编译器会忽略从`/*` 到 `*/`的所有字符，所以写注释没有语法规则，爱怎么写就怎么写，并且不管写多少都不会被编译进可执行文件中。

printf 语句的作用是把消息打印到屏幕。注意语句的末尾以; 号（Semicolon）结束，下一条语句 `return 0;` 也是如此。

C 语言用 {} 括号（Brace 或 Curly Brace）把语法结构分成组，在上面的程序中 printf 和 return 语句套在 main 的 {} 括号中，表示它们属于 main 的定义之中。我们看到这两句相比 main 那一行都缩进（Indent）了一些，在代码中可以用若干个空格（Blank）和 Tab 字符来缩进，缩进不是必须的，但这样使我们更容易看出这两行是属于 main 的定义之中的，要写出漂亮的程序必须有整齐的缩进。

正如前面所说，编译器对于语法错误是毫不留情的，如果你的程序有一点拼写错误，例如第一行写成了 stdoi.h，在编译时会得到错误提示：

```
$ gcc main.c
main.c:1:19: error: stdoi.h: No such file or directory
...
```

这个错误提示非常紧凑，初学者往往不容易看明白出了什么错误，即使知道这个错误提示说的是第 1 行有错误，很多初学者对照着书看好几遍也看不出自己这一行哪里有错误，因为他们对符号和拼写不敏感（尤其是英文较差的初学者），他们还不知道这些符号是什么意思又如何能记住正确的拼写？对于初学者来说，最想看到的错误提示其实是这样的：“在 main.c 程序第 1 行的第 19 列，您试图包含一个叫做 stdoi.h 的文件，可惜我没有找到这个文件，但我却找到了一个叫做 stdio.h 的文件，我猜这个才是您想要的，对吗？”可惜没有任何编译器会友善到这个程度，大多数时候你所得到的错误提示并不能直接指出谁是犯人，而只是一个线索，你需要根据这

个线索做一些侦探和推理。

有些时候编译器的提示信息不是 error 而是 warning,例如把上例中的 `printf("Hello, world.\n");` 改成 `printf(1);` 然后编译运行：

```
$ gcc main.c
main.c: In function ‘main’:
main.c:7: warning: passing argument 1 of ‘printf’ makes pointer from
integer without a cast
$ ./a.out
Segmentation fault
```

这个警告信息是说类型不匹配，但勉强还能配得上。警告信息不是致命错误，编译仍然可以继续，如果整个编译过程只有警告信息而没有错误信息，仍然可以生成可执行文件。但是，警告信息也是不容忽视的。出警告信息说明你的程序写得不够规范，可能有 Bug，虽然能编译生成可执行文件，但程序的运行结果往往是不正确的，例如上面的程序运行时出了一个段错误，这属于运行时错误。各种警告信息的严重程度不同，像上面这种警告几乎一定表明程序中有 Bug，而另外一些警告只表明程序写得不够规范，一般还是能正确运行的，有些不重要的警告信息 gcc 默认是不提示的，但这些警告信息也有可能表明程序中有 Bug。一个好的习惯是打开 gcc 的-Wall 选项，也就是让 gcc 提示所有的警告信息，不管是严重的还是不严重的，然后把这些问题从代码中全部消灭。比如把上例中的 `printf("Hello, world.\n");` 改成 `printf(0);` 然后编译运行：

```
$ gcc main.c
$ ./a.out
```

编译既不报错也不报警告，一切正常，但是运行程序什么也不打印。如果打开-Wall 选项编译就会报警告了：

```
$ gcc -Wall main.c
```

```
main.c: In function ‘main’:
main.c:7: warning: null argument where non-null required (argument
1)
```

如果 printf 中的 0 是你不小心写上去的（例如错误地使用了编辑器的查找替换功能），这个警告就能帮助你发现错误。虽然本书的命令行为了突出重点通常省略-Wall 选项，但是强烈建议你写每一个编译命令时都加上-Wall 选项。

注释可以跨行，也可以穿插在程序之中，看下面的例子。

带更多注释的 Hello World:

```
#include <stdio.h>
/*
* comment1
* main: generate some simple output
*/

int main(void)
{
printf(/* comment2 */"Hello, world.\n"); /*
comment3 */
**^^I^^I**return 0;
}
```

第一个注释跨了四行，头尾两行是 **注释的界定符 (Delimiter)** /和/，中间两行开头的 * 号 (Asterisk) 并没有特殊含义，只是为了看起来整齐，这不是语法规则而是大家都遵守的 C 代码风格 (Coding Style) 之一.

使用注释需要注意两点：

注释不能 **嵌套 (Nest)** 使用，就是说一个注释的文字中不能再出现`/*`和`*/`了，例如`/* text1/* text2 */ text3 */`是错误的，编译器只把`/* text1 /* text2 */`看成注释，后面的 `text3 */`无法解析，因而会报错。

有的 C 代码中有类似`// comment` 的注释，两个/斜线 (Slash) 表示**从这里直到该行末尾的所有字符都属于注释，这种注释不能跨行，也不能穿插在一行代码中间**。这是从 C++ 借鉴的语法，在 C99 中被标准化。

1.4.1 C 语言标准

C 语言的发展历史大致上分为三个阶段：Old Style C、C89 和 C99。Ken Thompson 和 Dennis Ritchie 最初发明 C 语言时有很多语法和现在最常用的写法并不一样，但为了 **向后兼容性 (Backward Compatibility)**，这些语法仍然在 C89 和 C99 中保留下来了，本书不详细讲 Old Style C，但在必要的地方会加以说明。C89 是最早的 C 语言规范，于 1989 年提出，1990 年首先由 ANSI(美国国家标准委员会，American National Standards Institute) 推出，后来被接纳为 ISO 国际标准 (ISO/IEC 9899:1990)，因而有时也称为 C90，最经典的 C 语言教材 [K&R] 就是基于这个版本的，C89 是目前最广泛采用的 C 语言标准，大多数编译器都完全支持 C89。C99 标准 (ISO/IEC 9899:1999) 是在 1999 年推出的，加入了许多新特性，但目前仍没有得到广泛支持，在 C99 推出之后相当长的一段时间里，连 gcc 也没有完全实现 C99 的所有特性。C99 标准详见 [C99]。本书讲 C 的语法以 C99 为准，但示例代码通常只使用 C89 语法，很少使用 C99 的新特性。

C 标准的目的是为了精确定义 C 语言，而不是为了教别人怎么编程，C 标准在表达上追求准确和无歧义，却十分不容易看懂，[Standard C] 和 [Standard C Library] 是对 C89 及其修订版本的阐释（可惜作者没有随 C99 更新这两本书），比 C 标准更容易看懂，另外，参考 [C99 Rationale] 也有助于加深对 C 标准的理解。

像`"Hello, world.\n"` 这种 **由双引号 (Double Quote) 引起来的一串字符称为字符串字面值 (String Literal)，或者简称字符串**。注意，程序的运行结果并没有双引号，printf 打印出来的只是里面的一串字符 `Hello, world.`，因此双引号是字符串字面值的界定符，夹在双引号中间的一串字符才是它的内容。注意，打印出来的结果也没有\n 这两个字符，这是因为

C 语言规定了一些 **转义序列 (Escape Sequence)**，这里的\n 并不表示它的字面意思，也就是说并不表示\和 n 这两个字符本身，而是合起来表示一个 **换行符 (Line Feed)** 。例如我们写三条打印语句：

```
printf("Hello, world.\n");
printf("Goodbye, ");
printf("cruel world!\n");
```

运行的结果是第一条语句单独打到第一行，后两条语句都打到第二行。为了节省篇幅突出重点,以后的例子通常省略 `#include` 和 `int main(void) { ... }` 这些 Boilerplate, 但需要加上这些构成一个完整的程序才能编译通过。C 标准规定的转义字符有以下几种：

转义序列	含义
'	单引号'（Single Quote 或 Apostrophe）
"	双引号"
\?	问号?（Question Mark）
\	反斜线\（Backslash）
\a	响铃（Alert 或 Bell）
\b	退格（Backspace）
\f	分页符（Form Feed）
\n	换行（Line Feed）
\r	回车（Carriage Return）
\t	水平制表符（Horizontal Tab）
\v	垂直制表符（Vertical Tab）

如果在字符串字面值中要表示单引号和问号，既可以使用转义序列\'和\?，也可以直接用字符' 和?，而要表示\或" 则必须使用转义序列，因为\字符表示转义而不表示它的字面含义，" 表示字符串的界定符而不表示它的字面含义。可见转义序列有两个作用：一是把普通字符转义成特殊字符，例如把字母 n 转义成换行符；二是把特殊字符转义成普通字符，例如\和" 是特殊字符，转义后取它的字面值。

C 语言规定了几个控制字符，不能用键盘直接输入，因此采用\加字母的转义序列表示。\a 是响铃字符，在字符终端下显示这个字符的效果是 PC 喇叭发出嘀的一声，在图形界面终端下的效果取决于终端的实现。在终端下显示 \b 和按下退格键的效果相同。\f 是分页符，主要用于控制打印机在打

印源代码时提前分页，这样可以避免一个函数跨两页打印。\n 和\r 分别表示 Line Feed 和 Carriage Return，这两个词来自老式的英文打字机，Line Feed 是跳到下一行（进纸，喂纸，有个喂的动作所以是 feed），Carriage Return 是回到本行开头（Carriage 是卷着纸的轴，随着打字慢慢左移，打完一行就一下子移回最右边），如果你看过欧美的老电影应该能想起来这是什么。用老式打字机打完一行之后需要这么两个动作，\r\n，所以现在 Windows 上的文本文件用\r\n 做行分隔符，许多应用层网络协议（如 HTTP）也用\r\n 做行分隔符，而 Linux 和各种 UNIX 上的文本文件只用\n 做行分隔符。在终端下显示\t 和按下 Tab 键的效果相同，用于在终端下定位表格的下一列，\v 用于在终端下定位表格的下一行。\v 比较少用，\t 比较常用，以后将“水平制表符”简称为“制表符”或 Tab。注意 "Goodbye, " 末尾的空格，字符串字面值中的空格也算一个字符，也会出现在输出结果中，而程序中别处的空格和 Tab 多一个少一个往往是无关紧要的，不会对编译的结果产生任何影响，例如不缩进不会影响程序的结果，main 后面多几个空格也没影响，但是 int 和 main 之间至少要有一个空格分隔开：

```
int main (void)
{
printf("Hello, world.\n");
return 0;
}
```

不仅空格和 Tab 是无关紧要的，换行也是如此，我甚至可以把整个程序写成一行，但是 include 必须单独占一行：

```
#include<stdio.h>
int main(void){printf("Hello, world.\n");return 0;}
```

这样也行，但肯定不是好的代码风格，去掉缩进已经很影响可读性了，写成现在这个样子可读性更差。如果编译器说第 2 行有错误，也很难判断是哪个语句有错误。所以，好的代码风格要求缩进整齐，每个语句一行，适当留空行。

1.5 常量

常量 (Constant) 是程序中最基本的元素，有字符 (Character) 常量、整数 (Integer) 常量、浮点数 (Floating Point) 常量和枚举常量。下面看一个例子：

```
printf("character: %c\ninteger: %d\nfloating point: %f\n", '}', 34, 3.14);
```

字符常量要用单引号括起来，例如上面的'}'，注意单引号只能括一个字符而不能像双引号那样括一串字符，字符常量也可以是一个转义序列，例如'\n'，这时虽然单引号括了两个字符，但实际上只表示一个字符。和字符串字面值中使用转义序列有一点区别，如果在字符常量中要表示双引号" 和问号?，既可以使用转义序列\" 和\?，也可以直接用字符" 和?，而要表示' 和\则必须使用转义序列。

计算机中整数和小数的内部表示方式不同，因而在 C 语言中是两种不同的类型（Type），例如上例的 34 和 3.14，小数在计算机术语中称为 **浮点数**。这个语句的输出结果和 Hello world 不太一样，字符串`"character: %c\ninteger: %d\nfloating point: %f\n"` 并不是按原样打印输出的，而是输出成这样：

```
character: }
integer: 34
floating point: 3.14
```

printf 中的第一个字符串称为 **格式化字符串（Format String）**，它规定了后面几个常量以何种格式插入到这个字符串中，在格式化字符串中%号（Percent Sign）后面加上字母 c、d、f 分别表示字符型、整型和浮点型的 **转换说明（Conversion Specification）**，转换说明只在格式化字符串中占个位置，并不出现在最终的打印结果中，这种用法通常叫做 **占位符（Placeholder）**。这也是一种字面意思与真实意思不同的情况，但是转换说明和转义序列又有区别：**转义序列是编译时处理的，而转换说明是在**

运行时调用 printf 函数处理的。源文件中的字符串字面值是`"character: %c\ninteger: %d\nfloating point: %f\n"`，
`\n` 占两个字符，而编译之后保存在可执行文件中的字符串是 `character: %c` 换行 `integer: %d` 换行 `floating point: %f` 换行，\n 已经被替换成一个换行符，而%c 不变，然后在运行时这个字符串被传给 printf，printf 再把其中的%c、%d、%f 解释成转换说明。有时候不同类型的数据很容易弄混，例如"5"、'5'、5，如果你注意了它们的界定符就会很清楚，第一个是字符串字面值，第二个是字符，第三个是整数。一定要严格区分它们之间的差别了。

需要规定一个转义序列\? 是因为 C 语言规定了一些 **三连符（Trigraph）**，在某些特殊的终端上缺少某些字符，需要用 Trigraph 输入，例如用??= 表示 # 字符。Trigraph 极不常用，介绍这个只是为了理解 C 语言规定转义序列的作用，即特殊字符转普通字符，普通字符转特殊字符，? 也是一种特殊字符。

1.6 变量

变量（Variable）是编程语言最重要的概念之一，变量是计算机存储器中的一块命名的空间，可以在里面存储一个值（Value），存储的值是可以随时变的，比如这次存个字符'a' 下次存个字符'b'，正因为变量的值可以随时变所以才叫变量。

常量有不同的类型，因此变量也有不同的类型，变量的类型也决定了它所占的存储空间的大小。例如以下四个语句定义了四个变量 fred、bob、jimmy 和 tom，它们的类型分别是字符型、整型、浮点型：

```
char fred;
int bob;
float jimmy;
double tom;
```

浮点型有三种，float 是单精度浮点型，double 是双精度浮点型，long double 是精度更高的浮点型。常量 3.14 应该看作 double 类型的常量，printf 的%f 也应该看作 double 型的转换说明。给变量起名不能太随意，上面四个变量的名字就不够好，我们猜不出这些变量是用来存什么的。而像下面这样起名就很好：

```
char firstletter;
char lastletter;
int hour, minute;
```

我们可以猜得到这些变量是用来存什么的，前两个变量的取值范围应该是'A' 'Z' 或'a' 'z'，变量 hour 的取值范围应该是 0 23，变量 minute 的取值范围应该是 0 59，所以应该给变量起有意义的名字。从这个例子中我们也看到两个相同类型的变量（hour 和 minute）可以一起声明。

给变量起名有一定的限制，C 语言规定必须以字母或下划线 _（Underscore）开头，后面可以跟若干个字母、数字、下划线，但不能有其它字符。例如这些是合法的变量名：Abc、_abc_、_123。但这些是不合法的变量名：3abc、ab$。

其实这个规则不仅适用于变量名，也适用于所有可以由程序员起名的语法元素，例如函数名、宏定义、结构体成员名等，在 C 语言中这些统称为标识符（Identifier）。

另外要注意，表示类型的 char、int、float、double 等虽然符合上述规则，但也不能用作标识符。在 C 语言中有些单词有特殊意义，不允许用作标识符，这些单词称为关键字（Keyword）或保留字 (Reserved Word)。通常用于编程的文本编辑器都会高亮显示（Highlight）这些关键字，所以只要小心一点通常不会误用作标识符。C99 规定的关键字有：

```
auto break case char const continue default do double
else enum extern float for goto if inline int long
register restrict return short signed sizeof static struct switch typedef
union unsigned void volatile while _Bool _Complex _Imaginary
```

还有一点要注意，一般来说应避免使用以下划线开头的标识符，以下划线开头的标识符只要不和 C 语言关键字冲突的都是合法的，但是往往被编译器用作一些功能扩展，C 标准库也定义了很多以下划线开头的标识符，所以除非你对编译器和 C 标准库特别清楚，一般应避免使用这种标识符，以免造成命名冲突。

请记住：理解一个概念不是把定义背下来就行了，一定要理解它的外延和内涵，也就是什么情况属于这个概念，什么情况不属于这个概念，什么情况虽然属于这个概念但一般推荐的做法（Best Practice）是要尽量避免这种情况，这才算是真正理解了。

1.6.1 声明和定义

C 语言中的声明（Declaration）有变量声明、函数声明和类型声明三种。如果一个变量或函数的声明要求编译器为它分配存储空间，那么也可以称为定义（Definition），因此定义是声明的一种。在接下来几章的示例代码中变量声明都是要分配存储空间的，因而都是定义。在下一章我们会看到函数的定义和声明也是这样区分的，分配存储空间的函数声明可以称为函数定义。声明一个类型是不分配存储空间的，但似乎叫“类型定义”听起来也不错，所以“类型定义”和“类型声明”表示相同的含义。声明和语句类似，也是以; 号结尾的，但是在语法上声明和语句是有区别的，语句只能出现在括号中，而声明既可以出现在中也可以出现在所有之外。

1.7 赋值

定义了变量之后，我们要把值存到它们所表示的存储空间里，可以用赋值（Assignment）语句实现：

```
char firstletter;
int hour, minute;
firstletter = 'a'; /* give firstletter the value 'a' */
```

```
hour = 11; /* assign the value 11 to hour */
minute = 59; /* set minute to 59 */
```

注意变量一定要先声明后使用，编译器必须先看到变量声明，才知道 firstletter、hour 和 minute 是变量名，各自代表一块存储空间。另外，变量声明中的类型表明这个变量代表多大的一块存储空间，这样编译器才知道如何读写这块存储空间。还要注意，这里的等号不表示数学里的相等关系，和 1+1=2 的等号是不同的，这里的等号表示赋值。在数学上不会有 i=i+1 这种等式成立，而在 C 语言中表示把变量 i 的存储空间中的值取出来，再加上 1，得到的结果再存回 i 的存储空间中。再比如，在数学上 a=7 和 7=a 是一样的，而在 C 语言中后者是不合法的。总结一下：定义一个变量，就是分配一块存储空间并给它命名；给一个变量赋值，就是把一个值保存到这块存储空间中。变量的定义和赋值也可以一步完成，这称为变量的初始化（Initialization），例如要达到上面代码的效果也可以这样写：

```
char firstletter = 'a';
int hour = 11, minute = 59;
```

在初始化语句中，等号右边的值叫做 Initializer，例如上面的'a'、11 和 59。注意，初始化是一种特殊的声明，而不是一种赋值语句。就目前来看，先定义一个变量再给它赋值和定义这个变量的同时给它初始化所达到的效果是一样的，C 语言的很多语法规则既适用于赋值也适用于初始化，但在以后的学习中你也会了解到它们之间的不同，请在学习过程中注意总结赋值和初始化的相同和不同之处。

如果在纸上“跑”一个程序（每个初学编程的人都要练这项基本功），可以用一个框表示变量的存储空间，在框的外边标上变量名，在框里记上它的值，如下图所示。

你可以用不同形状的框表示不同类型的变量，这样可以提醒你给变量赋的值必须符合它的类型。如果所赋的值和变量的类型不符会导致编译器报警告或报错（这是一种语义错误），例如：

```
int hour, minute;
hour = "Hello."; /* WRONG ! */
minute = "59"; /* WRONG !! */
```

注意第 3 个语句，把”59” 赋给 minute 看起来像是对的，但是类型不对，字符串不能赋给整型变量。既然可以为变量的存储空间赋值，就应该可以把值取出来用，现在我们取出这些变量的值用 printf 打印：

```
printf("Current time is %d:%d", hour, minute);
```

变量名用在等号左边表示赋值，而用在 printf 中表示把它的存储空间中的值取出来替换在那里。不同类型的变量所占的存储空间大小是不同的，数据表示方式也不同，变量的最小存储单位是字节（Byte），在 C 语言中 char 型变量占一个字节，其它类型的变量占多少字节在不同平台上有不同的规定.

Chapter 2

Lisp 语言入门

2.1 List Processing

To the untutored eye, Lisp is a strange programming language. In Lisp code there are parentheses everywhere. Some people even claim that the name stands for 'Lots of Isolated Silly Parentheses'. But the claim is unwarranted. Lisp stands for LISt Processing, and the programming language handles lists (and lists of lists) by putting them between parentheses. The parentheses mark the boundaries of the list. Sometimes a list is preceded by a single apostrophe or quotation mark, ' . Lists are the basis of Lisp.

2.1.1 Lisp Lists

In Lisp, a list looks like this: '(rose violet daisy buttercup). This list is preceded by a single apostrophe. It could just as well be written as follows, which looks more like the kind of list you are likely to be familiar with:

```
'(rose
  violet
  daisy
```

```
buttercup)
```

The elements of this list are the names of the four different flowers, separated from each other by whitespace and surrounded by parentheses,like flowers in a field with a stone wall around them.

Lists can also have numbers in them, as in this list: (+ 2 2). This list has a plus-sign, '+', followed by two '2's, each separated by whitespace.

In Lisp, both data and programs are represented the same way; that is, they are both lists of words, numbers, or other lists, separated by whitespace and surrounded by parentheses. (Since a program looks like data, one program may easily serve as data for another; this is a very powerful feature of Lisp.) (Incidentally, these two parenthetical remarks are not Lisp lists, because they contain ';' and '.' as punctuation marks.Here is another list, this time with a list inside of it:

```
'(this list has (a list inside of it))
```

The components of this list are the words 'this'. 'list'. 'has, and the list '(a list inside of it). The interior list is made up of the words 'a','list','inside','of','it'.

2.1.2 Lisp Atoms

In Lisp, what we have been calling words are called atoms. This term comes from the historical meaning of the word atom, which means 'indivisible'. As far as Lisp is concerned, the words we have been using in the lists cannot be divided into any smaller parts and still mean the same thing as part of a program; likewise with numbers and single character symbols like '+'. On the other hand, unlike an atom, a list can be split into parts.

In a list, atoms are separated from each other by whitespace. They can be right next to a parenthesis.Technically speaking, a list in Lisp consists of parentheses surrounding atoms separated by whitespace or surrounding other

lists or surrounding both atoms and other lists. A list can have just one atom in it or have nothing in it at all. A list with nothing in it looks like this: (), and is called the empty list. Unlike anything else, an empty list is considered both an atom and a list at the same time.The printed representation of both atoms and lists are called symbolic expressions or, more concisely, **s-expressions**. The word expression by itself can refer to either the printed representation, or to the atom or list as it is held internally in the computer. Often, people use the term expression indiscriminately. (Also, in many texts, the word form is used as a synonym for expression.)

Incidentally, the atoms that make up our universe were named such when they were thought to be indivisible; but it has been found that physical atoms are not indivisible. Parts can split off an atom or it can fission into two parts of roughly equal size. Physical atoms were named prematurely, before their truer nature was found. In Lisp, certain kinds of atom, such as an array, can be separated into parts; but the mechanism for doing this is different from the mechanism for splitting a list. As far as list operations are concerned, the atoms of a list are unsplittable.

As in English, the meanings of the component letters of a Lisp atom are different from the meaning the letters make as a word. For example, the word for the South American sloth, the 'ai', is completely different from the two words, 'a', and 'i'.

There are many kinds of atom in nature but only a few in Lisp: for example, numbers, such as $37, 511$, or 1729, and symbols, such as '+', 'foo', or 'forward-line'. The words we have listed in the examples above are all **symbols**. In everyday Lisp conversation, the word "atom" is not often used, because programmers usually try to be more specific about what kind of atom they are dealing with. Lisp programming is mostly about symbols (and sometimes numbers) within lists. (Incidentally, the preceding three word parenthetical remark is a proper list in Lisp, since it consists of atoms,

which in this case are symbols, separated by whitespace and enclosed by parentheses, without any non-Lisp punctuation.)

In addition, text between double quotation marks-even sentences or paragraphs-is an atom. Here is an example:

```
'(this list includes "text between quotation marks.")
```

In Lisp, **all of the quoted text including the punctuation mark and the blank spaces is a single atom.** This kind of atom is called a **string** (for 'string of characters') and is the sort of thing that is used for messages that a computer can print for a human to read. Strings are a different kind of atom than numbers or symbols and are used differently.

2.1.3 Whitespace in Lists

The amount of whitespace in a list does not matter. From the point of view of the Lisp language,

```
'(this list
   looks like this)
```

is exactly the same as this:

```
'(this list looks like this)
```

Both examples show what to Lisp is the same list, the list made up of the symbols 'this', 'list', 'looks', 'like', and 'this' in that order.

Extra whitespace and newlines are designed to make a list more readable by humans. When Lisp reads the expression, it gets rid of all the extra whitespace (but it needs to have at least one space between atoms in order to tell them apart.)

Odd as it seems, the examples we have seen cover almost all of what Lisp lists look like! Every other list in Lisp looks more or less like one of

these examples, except that the list may be longer and more complex. In brief, a list is between parentheses, a string is between quotation marks, a symbol looks like a word, and a number looks like a number. (For certain situations, square brackets, dots and a few other special characters may be used; however, we will go quite far without them.)

2.1.4 Run a Program

A list in Lisp-any list-is a program ready to run. If you run it (for which the Lisp jargon is **evaluate**), the computer will do one of three things: do nothing except return to you the list itself; send you an error message; or, treat the first symbol in the list as a command to do something. (Usually, of course, it is the last of these three things that you really want!)

The single apostrophe, ', that I put in front of some of the example lists in preceding sections is called a **quote**; when it precedes a list, it tells Lisp to do nothing with the list, other than take it as it is written. But if there is no quote preceding a list, the first item of the list is special: it is a command for the computer to obey. (In Lisp, these commands are called **functions**.) The list (+ 2 2) shown above did not have a quote in front of it, so Lisp understands that the + is an instruction to do something with the rest of the list: add the numbers that follow.

Here is how you can evaluate such a list: place your cursor immediately after the right hand parenthesis of the following list and then type $C{-}x\ C{-}e$:

```
(+ 2 2)
```

You will see the number 4 appear in the echo area. (In the jargon, what you have just done is "evaluate the list." **The echo area is the line at the bottom of the screen that displays or "echoes" text.**) Now try the same thing with a quoted list: place the cursor right after the following list and type $C - x\ C - e$:

```
'(this is a quoted list)
```

You will see (this is a quoted list) appear in the echo area.

In both cases, what you are doing is giving a command to the program inside of GNU Emacs called the **Lisp interpreter**-giving the interpreter a command to evaluate the expression. The name of the Lisp interpreter comes from the word for the task done by a human who comes up with the meaning of an expression-who "interprets" it.

You can also evaluate an atom that is not part of a list-one that is not surrounded by parentheses; again, the Lisp interpreter translates from the humanly readable expression to the language of the computer. But before discussing this, we will discuss what the Lisp interpreter does when you make an error.

2.1.5 Generate an Error Message

Partly so you won't worry if you do it accidentally, we will now give a command to the Lisp interpreter that generates an error message. This is a harmless activity; and indeed, we will often try to generate error messages intentionally. Once you understand the jargon, error messages can be informative. Instead of being called "error" messages, they should be called "help" messages. They are like signposts to a traveller in a strange country; deciphering them can be hard, but once understood, they can point the way.

The error message is generated by a built-in GNU Emacs debugger. We will 'enter the debugger'. You get out of the debugger by typing q.

What we will do is evaluate a list that is not quoted and does not have a meaningful command as its first element. Here is a list almost exactly the same as the one we just used, but without the single-quote in front of it. Position the cursor right after it and type $C-x\ C-e$:

```
(this is an unquoted list)
```

In GNU Emacs, a '*Backtrace*' window will open up and you will see the following in it:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function this)
  (this is an unquoted list)
  eval((this is an unquoted list))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Your cursor will be in this window (you may have to wait a few seconds before it becomes visible). To quit the debugger and make the debugger window go away, type:

q

Please type q right now, so you become confident that you can get out of the debugger. Then, type $C-x\ C-e$ again to re-enter it.

Based on what we already know, we can almost read this error message. You read the '*Backtrace*' buffer from the bottom up; it tells you what Emacs did. When you typed $C-x\ C-e$, you made an interactive call to the command eval-last-sexp. **eval is an abbreviation for 'evaluate' and sexp is an abbreviation for 'symbolic expression'.** The command means 'evaluate last symbolic expression', which is the expression just before your cursor.

Each line above tells you what the Lisp interpreter evaluated next. The most recent action is at the top. The buffer is called the '*Backtrace*' buffer because it enables you to track Emacs backwards.

At the top of the '*Backtrace*' buffer, you see the line:

```
Debugger entered--Lisp error: (void-function this)
```

The Lisp interpreter tried to evaluate the first atom of the list, the word 'this'. It is this action that generated the error message 'void-function this' The message contains the words 'void-function' and 'this'.

The word 'function' was mentioned once before. It is a very important word. For our purposes, we can define it by saying that a function is a set of instructions to the computer that tell the computer to do something.

Now we can begin to understand the error message: 'void-function this'. The function (that is, the word 'this') does not have a definition of any set of instructions for the computer to carry out.

The slightly odd word, 'void-function', is designed to cover the way Emacs Lisp is implemented, which is that when a symbol does not have a function definition attached to it, the place that should contain the instructions is 'void'.

On the other hand, since we were able to add 2 plus 2 successfully, by evaluating (+ 2 2), we can infer that the symbol + must have a set of instructions for the computer to obey and those instructions must be to add the numbers that follow the +.

2.1.6 Symbol Names and Function Definitions

We can articulate another characteristic of Lisp based on what we have discussed so far - an important characteristic: a symbol, like +, is not itself the set of instructions for the computer to carry out. Instead, the symbo is used, perhaps temporarily, as a way of locating the definition or set of instructions. What we see is the name through which the instructions car be found. Names of people work the same way. I can be referred to as 'MENG'; however, I am not the letters 'M', 'E', 'N','G' but am the consciousness consistently associated with a particular life-form. The name is not me, but

it can be used to refer to me.

In Lisp, one set of instructions can be attached to several names. For example, the computer instructions for adding numbers can be linked to the symbol plus as well as to the symbol + (and are in some dialects of Lisp) Among humans, I can be referred to as 'Philippe' as well as 'MENG' and by other words as well.

On the other hand, a symbol can have only one function definition attached to it at a time. Otherwise, the computer would be confused as to which definition to use. If this were the case among people, only one persor in the world could be named 'Philippe'. However, the function definition to which the name refers can be changed readily.

Since Emacs Lisp is large, it is customary to name symbols in a way that identifies the part of Emacs to which the function belongs. Thus, all the names for functions that deal with Texinfo start with 'texinfo-' and those for functions that deal with reading mail start with 'rmail-'.

2.1.7 The Lisp Interpreter

Based on what we have seen, we can now start to figure out what the Lisp interpreter does when we command it to evaluate a list. First, it looks to see whether there is a quote before the list; if there is, the interpreter just gives us the list. On the other hand, if there is no quote, the interpreter looks at the first element in the list and sees whether it has a function definition. If it does, the interpreter carries out the instructions in the function definition. Otherwise, the interpreter prints an error message.

This is how Lisp works. Simple. There are added complications which we will get to in a minute, but these are the fundamentals. Of course, to write Lisp programs, you need to know how to write function definitions and attach them to names, and how to do this without confusing either yourself

or the computer.

Now, for the first complication. In addition to lists, the Lisp interpreter can evaluate a symbol that is not quoted and does not have parentheses around it. The Lisp interpreter will attempt to determine the symbol's value as a variable.

The second complication occurs because some functions are unusual and do not work in the usual manner. Those that don't are called **special forms**. They are used for special jobs, like defining a function, and there are not many of them.

The third and final complication is this: if the function that the Lisp interpreter is looking at is not a special form, and if it is part of a list, the Lisp interpreter looks to see whether the list has a list inside of it. If there is an inner list, the Lisp interpreter first figures out what it should do with the inside list, and then it works on the outside list. If there is yet another list embedded inside the inner list, it works on that one first, and so on. It always works on the innermost list first. The interpreter works on the innermost list first, to evaluate the result of that list. The result may be used by the enclosing expression.

Otherwise, the interpreter works left to right, from one expression to the next.

Byte Compiling

One other aspect of interpreting: the Lisp interpreter is able to interpret two kinds of entity: humanly readable code, on which we will focus exclusively, and specially processed code, called **byte compiled code**, which is not humanly readable. Byte compiled code runs faster than humanly readable code.

You can transform humanly readable code into byte compiled code by running one of the compile commands such as byte-compile-file. Byte com-

piled code is usually stored in a file that ends with a '.elc' extension rather than a '.el' extension. You will see both kinds of file in the 'emacs/lisp' directory; the files to read are those with '.el' extensions.

As a practical matter, for most things you might do to customize or extend Emacs, you do not need to byte compile.

Chapter 3

Emacs 入门

3.1 Understanding Files and Buffers

You don't really edit files with Emacs. Instead, Emacs copies the contents of a file into a temporary buffer and you edit that. The file on disk doesn't change until you save the buffer. Like files, Emacs buffers have names. The name of a buffer is usually the same as the name of the file that you're editing. There are a few exceptions. Some buffers don't have associated files—for example, *scratch*(乱写) is just a temporary practice buffer, like a scratchpad; the help facility displays help messages in a buffer named *Help*, which also isn't connected to a file.

3.2 A Word About Modes

Emacs achieves some of its famed versatility by having various editing modes in which it behaves slightly differently. The word mode may sound technical, but what it really means is that Emacs becomes sensitive to the task at hand. When you're writing, you often want features like word wrap so that you don't have to press Enter at the end of every line. When you're

programming, the code must be formatted correctly depending on the language. For writing, there's text mode; for programming, there are modes for different languages, including C, Java, and Perl. Modes, then, allow Emacs to be the kind of editor you want for different tasks. Text mode and Java mode are major modes. A buffer can be in only one major mode at a time; to exit a major mode, you have to enter another one.

Mode	Function
Fundamental mode	The default mode
Text mode	For writing text
View mode	For viewing files but not editing
Shell mode	For running a shell within Emacs
Outline mode	For writing outlines
Indented text mode	For indenting text automatically
Paragraph indent text mode	For indenting the first line of each paragraph
Picture mode	For creating ASCII drawings using the keyboard
HTML mode	For writing HTML
SGML mode	For writing SGML and XML
LaTeX mode	For formatting files for TEX and LATEX
Compilation mode	For compiling programs
cc mode	For writing C,C++, and Java programs
Java mode	For writing Java programs
Perl mode and Cperl mode	For writing Perl programs
SQL mode	For interacting with databases using SQL
Emacs Lisp mode	For writing Emacs Lisp functions
Lisp mode	For writing Lisp programs
Lisp interaction mode	For writing and evaluating Lisp expressions

表 3.1: Major modes

Whenever you edit a file, Emacs attempts to put you into the correct major mode for what you're going to edit. If you edit a file that ends in *.c*, it puts you into cc mode. If you edit a file that ends in .el, it puts you in Lisp mode. Sometimes it looks at the contents of the file rather than just its name. If you edit a file formatted for TEX Emacs puts you in LaTeX mode.

If it cannot tell what mode you should be in, it puts you in fundamental mode, the most general of all. Because Emacs is extensible, addin modes are also available.In addition to major modes there are also minor modes. These define a particular aspect of Emacs's behavior and can be turned on and off within a major mode. For example, auto-fill mode means that Emacs should do word wrap; when you type a long line, it should automatically make an appropriate line break.

Mode	Function
Auto-fill mode	Enables word wrap
Overwrite mode	Replaces characters as you type instead of inserting them
Auto-save mode	Saves your file automatically every so often in a special auto-save file
Search mode	For searching
Flyspell mode	For flyspell spell-checker
Flyspell prog mode	For spell-checking programs with flyspell
Abbrev mode	Allows you to use word abbreviations
Paragraph indent text mode	For indenting the first line of each paragraph
Refill mode	A mode in which Emacs attempts to fill paragraphs as you edit them (a bit experimental)
Artist mode	For creating ASCII drawings using the mouse
Outline mode	For writing outlines
SGML name entity mode	For inserting special characters in HTML, SGML, and XML documents
ISO accents mode	For inserting accented characters in text files
Font lock mode	For highlighting text in colors and fonts to improve readability (separating, for example, comments from code visually)
Compilation mode	For compiling programs
Enriched mode	For saving text attributes
VC mode	For using various version control systems under Emacs
Info mode	A mode for reading Emacs's own documentation

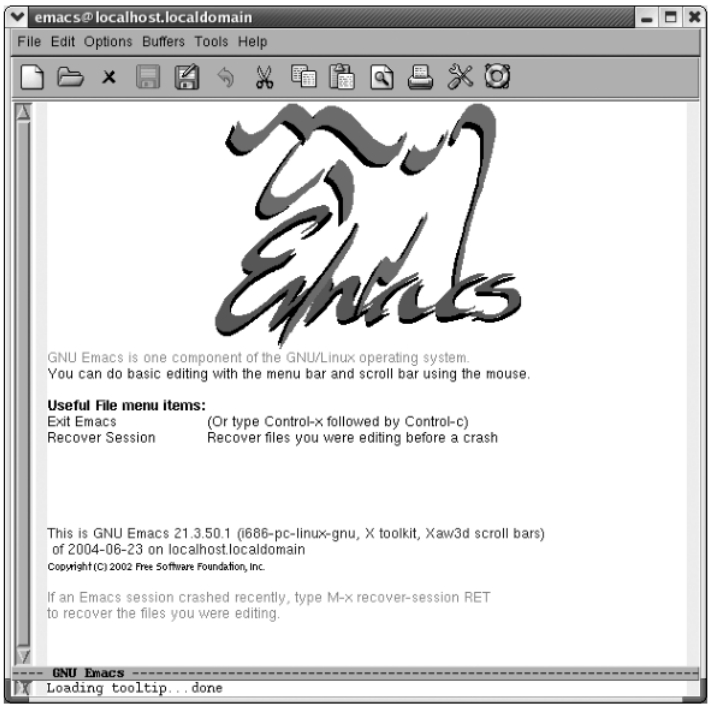
表 3.2: Minor modes

You may have noticed that several modes, including paragraph indent text mode, outline mode, and compilation mode, are both major and minor

modes. Each can be used alone—as a major mode—or with another major mode as a minor mode.

3.3 Starting Emacs

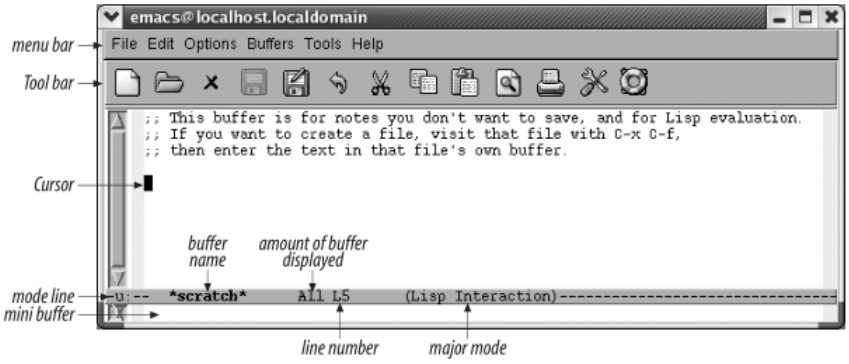
To start Emacs, simply click on the Emacs icon or type emacs on the command line and press Enter.



You' ll see a short message describing a few important menu items and the version of Emacs that you' re running. It may appear as a graphical splash screen (like the one shown here) or a text splash screen. This message disappears as soon as you type the first character. Emacs then puts you in an (almost) empty buffer called *scratch*, an ideal place for you to experiment.

3.4 About the Emacs Display

When you enter Emacs, you see a large workspace near the top of the window where you do your editing.



A cursor marks your position. The cursor is also called point, particularly among people who are more familiar with Emacs and in the online help system; therefore, it's useful to remember this term.

You don't have to do anything special before you start typing. As long as you type alphanumeric characters and punctuation, Emacs inserts them into your buffer. The cursor indicates where Emacs inserts the new characters; it moves as you type. Unlike many editors (particularly **vi**), Emacs does not have separate modes for inserting text and giving commands. If you get stuck for any reason, just press C-g.