C 语言笔记

The C Programming Language

theqiong.com

原书站点: http://cs.stanford.edu/people/eroberts/books/ArtAndScienceOfC/

谨献给...。

λ

History

C 语言是一种通用的、过程式的程序设计语言，广泛用于系统与应用软件的开发，其特点包括简洁的表达式、流行的控制流和数据结构、丰富的运算符集等。

C 语言最早是由 Dennis Ritchie 于 1969 年至 1973 年在贝尔实验室为 UNIX 系统设计的，并在 DEC PDP-11 计算机上实现。

之所以被称为“C”是因为 C 语言的很多特性是由一种更早的被称为 B 语言的编程语言中发展而来，或者说 C 预言是 B 语言的升级版。B 语言是 Ken Thompson 在 BCPL 语言的基础上开发的系统编程语言，BCPL 语言的起源可以追溯到一种最早的（也是影响最深远的）Algol 60 语言。

- Algol 60 语言衍生出来的语言包括 Pascal、Ada、Modula-2 及其他语言。
- C 语言的设计影响了许多后来的编程语言，例如 C++、Objective-C、Java、C# 等，它们都建立在 C 语言的语法和基本结构的基础上。
- C++ 语言包含了 C 语言的全部特性，并且在许多方面对 C 语言进行了扩展，增加了支持面向对象编程的特性。

C 语言不是一种“很高级”的语言，也不“庞大”，并且不专用于某一个特定的应用领域，其设计目标是提供一种能以简易的方式编译、处理低级存储器、产生少量的机器码以及不需要任何运行环境支持便能运行的程序，而且 C 语言也很适合搭配汇编语言来使用。

和其他任何编程语言一样，C 语言也有自己的优缺点。两者都源于这种语言预期的用途（编写操作系统和其他系统软件）和语言自身的基础理论体系：

- C 语言是一种低级语言。
  作为一种适合系统编程的语言，C 语言提供了对机器级概念（例如字节和地址）的访问，而这些都是其他编程语言试图隐藏的内容。此外，C 语言提供了与计算机内在指令紧密协调的操作，使得程序可以快速执行。
- C 语言是一种小型语言。
  C 语言提供了一套有限的特性集合，并在很大程度上依赖于标准库函数（这里，函数类似于其他编程语言中的过程或子程序）。
- C 语言是一种包容性语言。
  C 语言假设用户知道自己在做什么，因此它提供了更大的自由度，而且 C 语言没有提供详细的排错功能。

早期操作系统的核心大多由汇编语言组成，随着 C 语言的发展，C 语言已经可以用来编写操作系统的核心。1973 年，Unix 操作系统的核心正式用 C 语言改写，这是 C 语言第一次应用在操作系统的核心编写上，目前 C 语言编译器普遍存在于各种不同的操作系统中，例如 UNIX、MS-DOS、Microsoft Windows 及 Linux 等。

1980 年代，为了避免各开发厂商用的 C 语言语法产生差异，由美国国家标准局为 C 语言订定了一套完整的国际标准语法（称为 ANSI C）作为 C 语言的标准。1980 年代至今的有关程序开发工具，一般都支持符合 ANSI C 的语法。

C 语言在其特性使用上的限制非常少，在其他语言中认定为非法的操作在 C 语言中往往是允许的。例如，C 语言允许一个字符与一个整数值相加（或者是与一个浮点数相加）。这导致许多其他语言可以发现的错误，C 语言编译器却无法检测，比如一个额外的分号可能会导致无限循环，一个遗漏的 & 可能会引发程序崩溃。

现代编译器可以检查到常见的缺陷并且发出警告，但是没有一个编译器可以检测到全部缺陷。，因此很多错误无法被 C 语言编译器查出。

- 静态分析

与大多数 C 语言编译器相比，lint 可以检查 C 程序中潜在的错误，从而提供更广泛的错误分析，即使使用 gcc 的`-Wall`选项也仍然会遗漏一些 lint 能检测到的问题。

- 调试工具
  C 语言与汇编语言极为相似，因为直到程序运行时才能检测到大多数错误，需要调试工具来检测运行时错误等。
- 检查工具
  其他常用的工具包括越界检查工具（bounds-checker）和内存泄漏监测工具（leak-finder）。C 语言不要求检查数组下标，而越界检查工具增加了此项功能。内存泄漏监测工具可以帮助定位“内存泄露”，即那些动态分配后却从未被释放的内存块。
- 标准 C
  标准 C 增加了许多允许编译器检查错误的特性，这是经典 C 不具有的。

尽管 C 语言提供了许多低级处理的功能，但仍然保持着良好跨平台的特性，以标准规格写出的 C 语言程序可在许多计算机平台上（包括嵌入式系统、商业数据处理以及超级计算机等）进行编译。

- C 语言是一个有结构化程序设计、具有变量作用域（variable scope）以及递归功能的过程式语言。
- C 语言传递参数均是以值传递（pass by value），另外也可以传递指针（a pointer passed by value）。
- 不同的变量类型可以用结构体（struct）组合在一起。
- 只有 32 个保留字（reserved keywords），使变量、函数命名有更多弹性。
- 部分的变量类型可以转换，例如整型和字符型变量。
- 通过指针（pointer），C 语言可以容易地对存储器进行低级控制。
- 编译预处理（preprocessor）让 C 语言的编译更具有弹性。

K&R C

1978 年，丹尼斯·里奇（Dennis M. Ritchie）和布莱恩·柯林汉（Brain W. Kernighan）合作出版了《C 程序设计语言》的第一版。书中介绍的 C 语言标准也被 C 语言程序员称作“K&R C”，第二版的书中也包含了一些 ANSI C 的标准。K&R C 主要介绍了以下特色：

- 结构（`struct`）类型
- 长整数（`long int`）类型
- 无符号整数（`unsigned int`）类型
- 把运算符 =+ 和 =- 改为 += 和 -=。因为 =+ 和 =- 会使得编译器不知道用户要处理 `i = -10` 还是 `i =- 10`，使得处理上产生混淆。

即使在后来 ANSI C 标准被提出的许多年后，K&R C 仍然是许多编译器的最低标准要求，许多老旧的编译器仍然运行 K&R C 的标准。

ANSI C 和 ISO C

1983 年，美国国家标准协会为了创立 C 的标准组成了 X3J11 委员会，于 1989 年完成了 ANSI C[1]，并作为 ANSI X3.159-1989 “Programming Language C” 正式生效。这个版本的 C 语言经常被称作 “ANSI C” 或 “C89”。

1989 年，C 语言被 ANSI 标准化的一个目的是扩展 K&R C。在 K&R 出版后，一些新特性被非官方地加到 C 语言中。

- `void` 函数
- 函数返回 `struct` 或 `union` 类型
- `void *` 数据类型

[1] ANSI C 的目标是制定“一个无歧义性的且与具体机器无关的 C 语言定义”，而同时又要保持 C 语言原有的“精神”。

由 ANSI C 标准定义的 C 语言与 K& R C 之间最明显的变化是函数的声明与定义。使用 C 的软件开发者被鼓励遵循 ANSI C 文档的要求，因为它鼓励使用跨平台的代码。

ANSI C 被几乎所有广泛使用的编译器支持，而且多数 C 代码都是在 ANSI C 基础上编写的。任何仅仅使用标准 C 并且没有任何硬件依赖假设的代码实际上能保证在任何平台上用遵循 C 标准的编译器编译成功。如果没有这种预防措施，多数程序只能在一种特定的平台或特定的编译器上编译，例如，使用非标准库或图形用户界面库，或者有关编译器或平台特定的特性例如数据类型的确切大小和字节序。

ANSI C 规范了一些在 K&R C 中提及但没有具体描述的结构，特别是结构赋值和枚举。该标准还提供了一种新的函数声明形式，允许在使用过程中对函数的定义进行交叉检查。

在 ANSI 标准化自己的过程中，一些新的特性被加了进去，而且 ANSI 也规定了一套标准输入/输出、内存管理和字符串操作等扩展函数集的标准库。它精确地说明了在 C 语言原始定义中并不明晰的某些特性的行为，同时还明确了 C 语言中与具体机器相关的一些特性。

ANSI ISO（国际标准化组织）成立 ISO/IEC JTC1/SC22/WG14 工作组来规定国际标准的 C 语言。通过对 ANSI 标准的少量修改，最终通过了 ISO/IEC 9899:1990，这个版本有时候称为 C90，而且 C89 和 C90 通常指同一种语言。随后，ISO 标准被 ANSI 采纳。

传统 C 语言到 ANSI/ISO 标准 C 语言的改进包括：

- 增加了真正的标准库
- 新的预处理命令与特性
- 函数原型允许在函数声明中指定参数类型
- 一些新的关键字，包括 `const`、`volatile` 与 `signed`
- 宽字符、宽字符串与多字节字符
- 对约定规则、声明和类型检查的许多小改动与澄清

WG14 工作小组之后又于 1994 年，对 1985 年颁布的标准做了两处技术修订（缺陷修复）和一个补充（扩展）。下面是 1994 年做出的所有修改：

- 3 个新的标准库头文件 `iso646.h`、`wctype.h` 和 `wchar.h`
- 几个新的记号与预定义宏，用于对国际化提供更好的支持
- `printf/sprintf` 函数一系列新的格式代码
- 大量的函数和一些类型与常量，用于多字节字符和宽字节字符

为了降低 K&R C 和 ANSI C 标准之间的差异，`__STDC__ ("standard c")` 宏可以被用来将代码分割为 ANSI 和 K&R 部分。

```
#if __STDC__
extern int getopt(int, char * const *, const char *);
#else
extern int getopt();
#endif
```

上面最好使用 `#if __STDC__` 而不是 `#ifdef __STDC__`，因为一些实现可能会把 `__STDC__` 设置为 0 来表示不遵循 ANSI C。`__STDC__` 能处理任何没有被宏替换或者值为 0 的标示符。因而即使宏 `__STDC__` 没有定义来表示不遵循 ANSI C，`__STDC__` 仍然能像显示的那样工作。

在上面的例子，一个原型中使用了 ANSI 实现兼容的函数声明，而另一个使用了过时的非原形声明。它们在 C99 和 C90 中依旧是 ANSI 兼容的，但并不被鼓励使用。

C99

在 ANSI 的标准确立后，C 语言的规范在一段时间内没有大的变动，然而 C++ 在自己的标准化创建过程中继续发展壮大。«标准修正案一»在 1994 年为 C 语言创建了一个新标准，但是只修正了一些 C89 标准中的细节和增加更多更广的国际字符集支持。不过，这个标准引出了 1999 年 ISO 9899:1999 的发表，它通常被称为 C99。C99 被 ANSI 于 2000 年 3 月采用。

在 C99 中包括的特性有：

- 增加了对编译器的限制，比如源程序每行要求至少支持到 4095 字节，变量名函数名的要求支持到 63 字节(`extern` 要求支持到 31)。
- 增强了预处理功能。例如：
  - 宏支持取可变参数 `#define Macro(...) __VA_ARGS__`
  - 使用宏的时候，允许省略参数，被省略的参数会被扩展成空串。
  - 支持 // 开头的单行注释(这个特性实际上在 C89 的很多编译器上已经被支持了)
- 增加了新关键字 `restrict`,`inline`,`_Complex`,`_Imaginary`,`_Bool`
  - 支持 `long long`,`long double _Complex`,`float _Complex` 等类型
- 支持不定长的数组，即数组长度可以在运行时决定，比如利用变量作为数组长度。声明时使用 `int a[var]` 的形式。不过考虑到效率和实现，不定长数组不能用在全局，或 `struct` 与 `union` 里。
- 变量声明不必放在语句块的开头，for 语句提倡写成 `for(int i=0;i<100;++i)` 的形式，即 `i` 只在 for 语句块内部有效。
- 允许采用 `(type_name) {xx,xx,xx}` 类似于 C++ 的构造函数的形式构造匿名的结构体。
- 初始化结构的时候允许对特定的元素赋值，形式为：

```
struct test{
  int a[3], b;
}
foo[] = {
  [0].a = {1},
  [1].a = 2
};

struct test{
  int a, b, c, d;
}
foo = {.a = 1, .c = 3, 4, .b = 5} // 3,4 是对.c,.d 赋值的
```

- 格式化字符串中，利用`\u` 支持 unicode 的字符。
- 支持 16 进制的浮点数的描述。
- `printf/scanf` 的格式化串增加了对 `long long int` 类型的支持。
- 浮点数的内部数据描述支持了新标准，可以使用 `#pragma` 编译器指令指定。
- 除了已有的 `__line__`,`__file__` 以外，增加了 `__func__` 得到当前的函数名。
- 允许编译器化简非常数的表达式。
- 修改了`/`,`%` 处理负数时的定义，这样可以给出明确的结果，例如在 C89 中，

  `-22 / 7 = -3, -22 \% 7 = -1`

  也可以

`-22 / 7= -4, -22 \% 7 = 6`

而 C99 中明确为`-22 / 7 = -3, -22 % 7 = -1`,只有一种结果。

- 取消了函数返回类型默认为 `int` 的规定。
- 允许 `struct` 定义的最后一个数组不指定其长度,写做 `[](flexible array member)`。
- `const const int i` 将被当作 `const int i` 处理。
- 增加和修改了一些标准头文件,比如定义 `bool` 的 `<stdbool.h>`,定义一些标准长度的 `int` 的 `<inttypes.h>`,定义复数的 `<complex.h>`,定义宽字符的 `<wctype.h>`,类似于泛型的数学函数 `<tgmath.h>`,浮点数相关的 `<fenv.h>`。
- 在 `<stdarg.h>` 增加了 `va_copy`,其中 `va_copy` 能够复制 `va_list`,而 `va_copy(va2,va1)` 可以复制 `va1` 到 `va2`。
- 在 `<time.h>` 里增加了 `struct tmx`,对 `struct tm` 做了扩展。
- 输入输出对宽字符以及长整数等做了相应的支持。

C11

2011 年 12 月 8 日,ISO 正式发布了新的 C 语言的新标准 C11,之前被称为 C1X,官方名称为 ISO/IEC 9899:2011,这个标准是 C 程序设计语言的现行标准。

新的标准提高了对 C++ 的兼容性,并增加了一些新的特性。这些新特性包括:

- 对齐处理 (Alignment) 的标准化(包括 `_Alignas` 标志符,`alignof` 运算符,`aligned_alloc` 函数以及 `<stdalign.h>` 头文件)。
- `_Noreturn` 函数标记,类似于 gcc 的 `__attribute__((noreturn))`。
- `_Generic` 关键字。
- 多线程 (Multithreading) 支持,包括:
  - `_Thread_local` 存储类型标识符,`<threads.h>` 头文件里面包含了线程的创建和管理函数。
  - `_Atomic` 类型修饰符和 `<stdatomic.h>` 头文件。
- 增强的 Unicode 的支持。基于 C Unicode 技术报告 ISO/IEC TR 19769:2004,增强了对 Unicode 的支持。包括为 UTF-16/UTF-32 编码增加了 `char16_t` 和 `char32_t` 数据类型,提供了包含 unicode 字符串转换函数的头文件 `<uchar.h>`。
- 删除了 `gets()` 函数,使用一个新的更安全的函数 `gets_s()` 替代。
- 增加了边界检查函数接口,定义了新的安全的函数,例如 `fopen_s()`,`strcat_s()` 等。
- 增加了更多浮点处理宏。
- 匿名结构体/联合体支持。这个在 gcc 早已存在,C11 将其引入标准。
- 静态断言(Static assertions,`_Static_assert()`)在解释 `#if` 和 `#error` 之后被处理。
- 新的 `fopen()` 模式—`("…x")`,类似 POSIX 中的 `O_CREAT|O_EXCL`,在文件锁中比较常用。
- 新增 `quick_exit()` 函数作为第三种终止程序的方式。当 `exit()` 失败时可以做最少的清理工作。

Preface

Specification

虽然基本的程序设计并不需要具备高等数学和电子学的知识,但是现在的情况是很多人可能还不太会或者不习惯写独立函数。仔细想想,回顾一下以前看过的 C 语言教程,很多示例、功能代码都写在 main 函数中,输出的一系列信息字符串也是直接写在代码中,虽然这样比较简单,对于初学者来说,也比较容易理解,但这对从一开始就培养良好编程习惯是很不好的。

在多数大学,所学的第一个语言就是 C 语言。学第一个语言养成的风格和习惯,对未来的学习和工作都会产生重要影响,因此,从一开始就应该培养良好的编程风格和习惯。写教程的大牛们也应该注意这一点。

程序设计既是一门科学,也是一门艺术。形成良好的程序设计风格需要掌握很多知识,而不只是记住一组规则,必须通过实践并阅读其他程序来不断学习。

在程序设计中,最重要的是能否从陈述问题过渡到解决问题。要做到这一点,就必须以逻辑方式考虑问题,训练自己用计算机能够理解的方式表达自己的逻辑。最重要的是,不要被困难和挫折压倒,要坚持到底。若能坚持下来,就会发现解决问题是件多么令人兴奋的事情,它所带来的喜悦足以让人忘却学习过程中遇到的任何挫折。

所有的程序员,即使是最好的程序员,都会犯错误。在程序中找出这些错误是非常重要的,因此良好的可测试性和可维护性是代码的基本要求,应该一开始就培养这方面的意识。

任何程序,不管它看起来是多么的完美,总是会有人(原先的程序员或项目继承者)想对它进行一些修改,可能需要修改其中的一些逻辑错误或需要加入一些新功能。作为一个程序员,应该意识到所有的程序都可能在某一天改变,他有责任让修改程序的任务轻松一些。

要减轻维护所写的程序所需的工作,一个重要的途径就是使程序的可读性更强。

软件开发的一个基本事实是读一个程序的次数比写它的次数多得多。然而,一个程序最重要的读者不是机器而是人——那些将在程序生命周期中使用它的程序员。让程序变成编译器能接受的状态仅仅是程序设计工作的一部分。好的程序员会将大部分时间花在诸如给程序添加注释等一些编译器忽略的方面。当编译器遇到表示注释开始的/* 符号时,对其后的字符将不予理睬,直到遇到注释结束字符 */。

良好的风格和可读性对程序的维护来说是很重要的。给程序加上精辟的注释并使之对普通读者有所帮助会花费额外的时间,但这项投入会给以后修改程序节省更多的时间。

下面就什么是良好的程序设计风格,怎样才能做到和从格式上讲,判断一个程序写得好不好的标准又是什么提出了一些策略和指导方针,但要用编写的程序对其他读者来说是否易读进行检验。作为一个实验,可以阅读自己编写的一个程序,看一看自己第一次读它时能否读懂。

无论如何,遵循下面给出的一些格式上的规则有助于写出较好的程序。

1. 用注释告诉读者他们所需要知道的内容。
   要向读者解释那些很复杂的或只通过阅读程序很难理解的部分。如果希望有的读者能对程序进行修改,最好能简要介绍一下自己是怎样做的。另一方面,也不要过于详细地解释一些浅显易懂的内容,例如,如果一定要给这样的语句后添加注释:

   ```
   total += value; /* 给 total 加上 value*/
   ```

   需要这样的注释的人肯定不是能使用这个程序的人。最后也是最重要的,就是确保所加的注释能正确地反映程序当前的状态。当修改程序时,也要及时更新程序的注释。
2. 使用缩进来区别程序不同的控制级别。

恰当地使用缩进能突出程序中的函数体、循环和条件控制，它们对程序的可读性和结构的清晰性很重要。

3. 使用有意义的名字。
   例如，在支票结算的程序中，变量名 balance 清楚地告诉读者变量中包含的值是什么，假如只使用一个字母 b，可能会使程序更短更容易输入，但对读者来说，这个程序的可用性降低了。
4. 制定变量的命名规则，使读者能够从名字中了解其功能。
   变量名和数据类型名总是以小写字母开头的，例如 total、balance、entry 和 double 等，相比之下，函数名常常以大写字母开头，例如 GetInteger、GetReal 等。此外，若函数名是由几个单词组合而成时，就像 GetInteger 一样，每个单词的首字母都要大写，这样可以一目了然。
5. 在适当的情况下使用标准习语和约定。
   由于许多软件公司发布了有关程序设计风格和程序结构的内部标准，甚至有些还可能与这里的建议相悖，此时就要入乡随俗。如果使用的产品有内部标准，就应该遵守它，这样才能使其他程序员更容易理解。总的来说，只有在一定范围内对一些基本习惯达成共识后，程序设计才能顺利进行。
6. 避免不必要的复杂性。
   为了程序的可读性牺牲一些程序效率常常是值得的。我们的宗旨是让程序仍容易阅读。为了达到这个目的，最好重新审校自己的程序风格。

在进行程序设计时，最好早一些开始，完成后把它放在一边放几天，然后重新把它拿出来，看看对你来说它容易读懂吗？对别人来说，将来它容易维护吗？如果发现程序实际上没有意义或不易读懂，就应该花些时间来修改它。

再者，可以采用一些能适应变更的设计以便使将来对程序的修改容易一些。因为程序员知道哪些地方比其他地方更容易修改，所以程序员常常根据经验来判断程序的哪些地方应该尽可能设计得灵活些。

一般考虑到以后其他人也可能对程序做修改，所以希望可以做到只做一处修改，修改效果便能传递到程序的其他所有相关部分，这样才能使程序修改更加方便，并且不会因为修改程序的一部分但未修改其他相应部分而破坏程序。

Rules

程序员要知道程序设计语言中不同语句的作用和用法。有时，常常要逐句地检查程序，尤其是查找那些使程序不能正常运行的逻辑错误时，但这种详细的观察并不是检查程序的唯一方法，有时回过头从整体上检查程序会更有帮助。

认识现有模型的大体模式并利用它来构建新的程序是程序设计的基本策略，其中从程序的每个独立部分的层面上理解程序，这是归纳的方法。另外，还可以从更全局性的角度分析一个程序，主要考虑它作为一个整体是如何工作的，这种整体的角度使人可以从另一个角度分析程序，这对成功的程序设计来说至关重要。

归纳论（reductionism）是一种哲学，它认为只有理解一个事物的每个组成部分才能很好地理解该事物。

整体论（holism）正好与之相反，它认为整体并非每一部分的简单叠加。

在进行编程时，必须学会从这两种角度分析程序。若只注意大的方面，则不能理解解决问题需要的工具；若过分关注细节，则会只见树木，不见树林。

在进行开发时，最好的方法是交替地使用这两种视角，整体论有助于从整体上把握程序的作用，使程序员对程序设计过程的直觉更加敏锐，并能够从较高的层面研究程序，从而探究这个程序是干什么的；另一方面，在实际写程序时，则需要适当采用归纳法，以了解程序是怎样组合在一起的。

Programming

计算机依据程序运行，为了让计算机按人的意图处理事务，必须要预先设计好完成各种任务的程序，并且预先将它们存放在存储器中。

C 语言是为程序员设计的，并非初学者设计的，它的很多特性只有在理解了大的概念框架后才有意义，而初学者往往意识不到这一点。

从很多方面讲，C 语言都是一门复杂的语言，而控制复杂性又是程序员的重要职责。当面对一个很复杂的、不能立即解决的问题时，可将它分解成多个部分，然后单独思考每个部分的解决方案。

更进一步，当某个部分达到了一定的复杂度时，可以将它抽象出来，定义为一个简单的接口。这样，用户就不必仔细理解这个抽象的细节，从而简化了程序的概念结构。实际上，接口的设计标准，对于任何一个需要理解现代程序设计原理的人来说都是至关重要的。

C 语言的基本的复合结构包括数组、指针、文件和记录等。为了强调这些结构之间的联系，在引入数组后应尽快熟悉指针，从而便于揭示出封装在抽象数据类型中的内容。

C 语言库函数强调抽象的原则，可以隐藏复杂性。比如字符串操作尽管在概念上很简单，但在能灵活使用字符串之前，必须理解数组、指针和内存分配等机制。

C 语言中涵盖了一些重要的主题，如模块化开发和抽象数据类型等。现在，库和模块化开发已经成为了现代程序设计的基本概念。

另外，在 C 语言中，还应该尽量早的引入递归的概念。

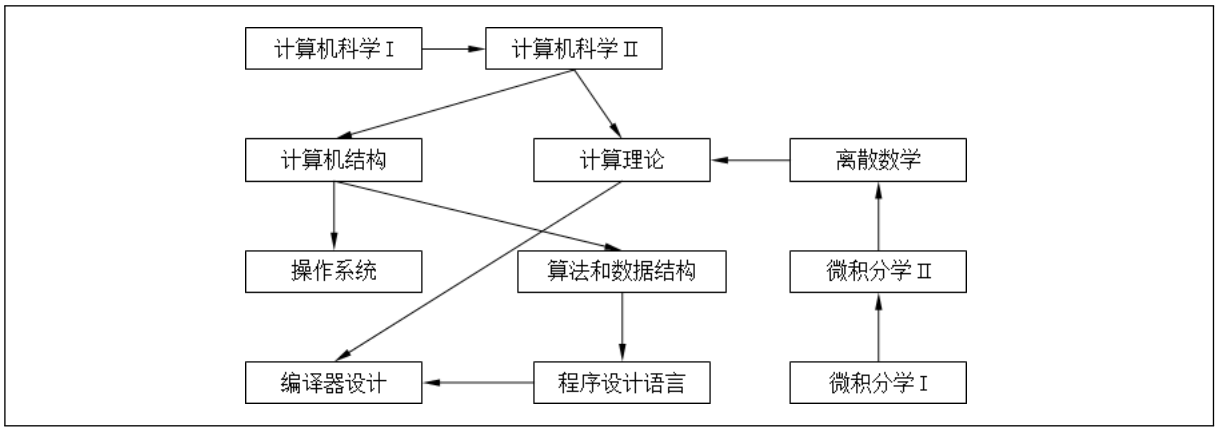
Algorithm

解决问题是一种重要的概括性的能力,它比学习程序设计语言的机制更重要。

在设计程序来解决问题时,随着问题越来越复杂,需要周密地考虑才能找到解决的策略,所以在写最终的程序前,需要考虑多个策略而不是一个策略。通过用多种解决方案解决一个问题,可以明白如何比较不同的策略,以及如何选取最佳的策略。

函数之所以对于程序设计来说很重要,其中的一个原因是函数提供了算法实现的基础。

算法本身是抽象的策略,通常用自然语言表达,而函数是以某种程序设计语言表示的算法的具体实现。当要将算法作为程序的一部分实现时,通常要写一个函数来执行算法,而该函数也可以调用其它函数处理它的一部分工作。



Eric S. Roberts

目　录

I Introduction 1

第 1 章 Overview . . . 3

第 2 章 Coding . . . 5

第 3 章 Document . . . 7

第 4 章 Library . . . 9
4.1 Introduction . . . 9
4.2 Standard Library . . . 10
4.3 GNU Library . . . 13

第 5 章 Preprocess . . . 17
5.1 Introduction . . . 17
5.2 Preprocessor . . . 18
5.3 Macros Definition . . . 25
5.3.1 Simple Macro . . . 27
5.3.2 Macro with Parameters . . . 28
5.4 Include File . . . 32
5.5 Conditional Compiling . . . 32
5.5.1 #if & #endif . . . 33
5.5.2 #ifdef & #ifndef . . . 34
5.5.3 #elif & #else . . . 34
5.6 Advanced Directives . . . 36
5.6.1 #error . . . 36
5.6.2 #line . . . 36
5.6.3 #pragma . . . 37
5.7 Predefined Macros . . . 38

第 6 章 Interprete . . . 41
6.1 Interpreter . . . 41
6.1.1 Overview . . . 41
6.1.2 Bytecode . . . 42
6.1.3 JIT . . . 42

第 7 章 Compiling . . . 45
7.1 Overview . . . 45
7.2 Stage . . . 46
7.3 Classification . . . 47
7.3.1 GCC . . . 50
7.3.2 G++ . . . 51

7.3.3 TCC . . . 53
7.3.4 Clang . . . 53
7.3.5 LLVM . . . 54

第 8 章 Linker . . . 57

第 9 章 Testing . . . 61
9.1 Test Processes . . . 61
9.2 Test Methods . . . 62
9.3 Test Types . . . 63
9.4 Test Periods . . . 63

第 10 章 Debugging . . . 65
10.1 Debugger . . . 66

第 11 章 Maintenance . . . 71

II Foundation 73

第 12 章 Introduction . . . 75
12.1 Hello, world. . . . 77
12.2 Comments . . . 77
12.3 Library . . . 78
12.4 Header file. . . . 79
12.5 Main . . . 81
12.6 Compiling . . . 84
12.7 Makefile . . . 86
12.8 Linking . . . 88
12.9 Debuging . . . 89

第 13 章 Lexical Structure . . . 93
13.1 White space. . . . 93
13.2 Comments . . . 93
13.3 Identifiers . . . 93
13.4 Constants . . . 94
13.5 Operators . . . 95
13.6 Punctuation tokes . . . 95

第 14 章 Data Types . . . 97
14.1 Integer Type . . . 97
14.2 Floating-point Type . . . 99
14.3 String Type . . . 100
14.4 Wide Characters . . . 101

第 15 章 Format I/O . . . 103
15.1 printf . . . 103
15.2 scanf . . . 104

第 16 章 Expressions . . . 107
16.1 Constants . . . 107
16.1.1 Integer Constants . . . 108
16.1.2 Float-point Constants . . . 108
16.1.3 String Constants . . . 109
16.2 Variables . . . 109
16.2.1 Variable Declaration . . . 110
16.2.2 Local Variables . . . 111
16.2.3 Global Variables . . . 111
16.3 Scope . . . 115
16.4 Assignment . . . 117
16.4.1 Integer . . . 118
16.4.2 Float-point . . . 119
16.5 Initialization . . . 120
16.6 Operators . . . 120
16.6.1 Arithmetic Operator . . . 120
16.6.2 Macro Operator . . . 121
16.6.3 Comma Operator . . . 123
16.7 Precedence . . . 123
16.8 Associativity . . . 124

第 17 章 Type Conversion . . . 127
17.1 Implicit Conversation . . . 127
17.1.1 Arithmetic Conversion . . . 128
17.1.2 Assignment Conversion . . . 130
17.2 Explicit Conversion . . . 130

第 18 章 Programming Idiom . . . 133
18.1 Increment/Decrement . . . 134

第 19 章 Control Statements . . . 135
19.1 Repeat-N-times Idiom . . . 135
19.2 Iteration . . . 136
19.2.1 Index Variables . . . 136
19.2.2 Initialization . . . 137
19.2.3 Read-until-sentinel Idiom . . . 138
19.3 Conditional Execution . . . 139
19.4 Debugging . . . 143
19.5 Formatted Output . . . 143
19.6 #define . . . 146

第 20 章 Statement Classification . . . 149
20.1 Simple Statements . . . 149
20.1.1 Embedded Assignment . . . 150
20.1.2 Multiple Assignment . . . 150
20.1.3 Block . . . 151
20.2 Control Statements . . . 152
20.3 Null Statement . . . 152
20.4 Boolean Data . . . 153
20.4.1 Relational Operators . . . 154

20.4.2 Logical Operators . . . 155
20.4.3 Short-circuit Evaluation . . . 157
20.4.4 Flags . . . 157
20.5 If Statements . . . 159
20.5.1 Single-line if statements . . . 161
20.5.2 Multiline if statements . . . 161
20.5.3 if/else statements . . . 162
20.5.4 Cascading if statements . . . 162
20.5.5 ?: Operator . . . 163
20.6 Switch Statements . . . 165
20.7 While Statements . . . 168
20.7.1 Infinite Loop . . . 170
20.7.2 Loop-and-a-half Problem . . . 171
20.8 For Statements . . . 173
20.8.1 Nested For Loop . . . 176
20.9 Do Statements . . . 178
20.10 Jump statements . . . 179
20.10.1 Break statement . . . 179
20.10.2 Continue Statements . . . 179
20.10.3 Goto Statements . . . 180

III Function **183**

第 21 章 Introduction . . . 185
21.1 Calling . . . 186
21.2 Definition . . . 188
21.3 Declaration . . . 190
21.3.1 Variable Declaration . . . 190
21.3.2 Function Declaration . . . 190
21.4 Argument . . . 190
21.4.1 Typecast . . . 191
21.4.2 Array . . . 192

第 22 章 Prototype . . . 193

第 23 章 Procedure . . . 195

第 24 章 Custom Function . . . 197
24.1 Return Statement . . . 197
24.2 Control Structure . . . 199
24.3 Return Value . . . 201
24.4 Exit Function . . . 202
24.5 Predicate Function . . . 203
24.6 String Comparation . . . 204

第 25 章 Function Call . . . 205
25.1 Parameter Passing . . . 206
25.2 Nested Functions . . . 209

第 26 章 Layout 215

第 27 章 Callback 217

第 28 章 Recursion 219
28.1 Introduction 219
28.2 Definition 220
28.3 Applications 221
28.3.1 Fixed-point combinator 221
28.3.2 Tail Recursion 222
28.3.3 Recursion Data 222
28.3.4 Quicksort 222

第 29 章 Top-down Design 227
29.1 Decomposition 227
29.2 PrintCalendar 228
29.3 PrintCalendarMonth 228

IV Compound Type 237

第 30 章 Overview 239

第 31 章 Array 241
31.1 Introduction 241
31.2 Definition 241
31.3 Declaration 241
31.4 Selection 243
31.4.1 Index Range 245
31.4.2 Character Index 246
31.4.3 Internal Representation 246
31.5 Memory Addressing 247
31.6 Memory Allocation 248
31.7 Reference Elements 249
31.8 Array Parameters 250
31.8.1 Elements Number 251
31.8.2 Passing Mechanism 253
31.8.3 Constant Array 254
31.8.4 Get/PrintIntegerArray 256
31.8.5 ReverseIntegerArray 257
31.8.6 SwapIntegerElements 258
31.8.7 Tabulation 261
31.9 Static Initialization 267
31.9.1 Array Subscript 267
31.9.2 Array Size 268
31.9.3 Array Replication 269

第 32 章 Matrix 271
32.1 Introduction 271
32.2 Parameter 272

32.3 Initialization . . . 273
32.4 Operation . . . 274

第 33 章 Stack . . . 275
33.1 Introduction . . . 275
33.2 Definition . . . 275
33.3 Software Stack . . . 276

第 34 章 Queue . . . 279
34.1 Introduction . . . 279
34.2 Implementation . . . 279

第 35 章 Records . . . 281
35.1 Introduction . . . 281
35.1.1 Overview . . . 282
35.1.2 History . . . 282
35.1.3 Operations . . . 283
35.1.4 Representation . . . 284
35.2 Concepts . . . 284
35.3 Definition . . . 285
35.4 Declaration . . . 286
35.5 Selection . . . 286
35.6 Initialization . . . 286
35.7 Applications . . . 287
35.7.1 Coordinate . . . 287
35.7.2 Data Record . . . 288

V Algorithm 291

第 36 章 Introduction . . . 293
36.1 History . . . 293
36.2 Characteristics . . . 294
36.3 Implementations . . . 294
36.4 Formal Algorithm . . . 295

第 37 章 Bubble Sort . . . 297

第 38 章 Prime . . . 299
38.1 Verifying strategy . . . 299
38.2 Demonstrating algorithm correctness . . . 300
38.3 Improving algorithmic efficiency . . . 300
38.4 Tradeoff alternative implementations . . . 303

第 39 章 Greatest Common Divisor . . . 305
39.1 brute-force algorithm . . . 305
39.2 Euclid algorithm . . . 306
39.3 Mathematical Methods . . . 308

第 40 章 Numerical algorithms . . . 309
40.1 Successive approximation . . . 309

40.2 Error handling . . . 311
40.3 Series expansion . . . 311
40.4 Taylor Series . . . 313
40.5 Numeric Types . . . 316
40.5.1 Integer Type . . . 316
40.5.2 Unsigned Integer Type . . . 317
40.5.3 Float Type . . . 318

第 41 章 Complexity . . . 319
41.1 Time Complexity . . . 319
41.2 Space Complexity . . . 319

VI Engineering 321

第 42 章 Introduction . . . 323
42.1 History . . . 323
42.2 Subdisciplines . . . 324

第 43 章 Overview . . . 325
43.1 Definition . . . 325
43.2 Comparation . . . 326
43.3 Methodology . . . 326

第 44 章 Development . . . 329
44.1 Source File . . . 329
44.2 Header File . . . 330
44.3 Including Files . . . 330
44.4 Shared Macros . . . 331
44.5 Shared Type Definitions . . . 331
44.6 Shared Function Prototypes . . . 332
44.7 Shared Variables Declarations . . . 332
44.8 Nested Includes . . . 334
44.9 Protect Header File . . . 334
44.10 Error Report . . . 335
44.11 Multi-files . . . 335
44.12 Building Program . . . 340
44.12.1 Makefile . . . 341
44.12.2 Linking . . . 342
44.13 Rebuilding . . . 342
44.14 Optional Macros . . . 344

第 45 章 Large Program . . . 345
45.1 Overview . . . 345
45.2 Module . . . 346
45.2.1 Module Relationship . . . 347
45.2.2 Module Types . . . 347
45.2.3 Information Hiding . . . 348
45.2.4 Abstract Type . . . 350
45.3 Differences . . . 352
45.3.1 Comments . . . 352

45.3.2 Symbol . . . 352
45.3.3 Void . . . 352
45.3.4 Arguments . . . 353
45.3.5 Reference . . . 353
45.3.6 Allocation . . . 353
45.4 Classes . . . 356
45.4.1 Definition . . . 357
45.4.2 Member . . . 358
45.5 Member Functions . . . 358
45.5.1 Declaration . . . 358
45.5.2 Definition . . . 359
45.6 Constructor . . . 360
45.7 Memory Allocation/Deallocation . . . 361
45.8 Deconstructor . . . 362
45.9 Overload . . . 363
45.9.1 Function Overloading . . . 363
45.9.2 Operator Overloading . . . 364
45.10 Input/Output . . . 365
45.11 Inheritance . . . 365
45.12 Virtual Functions . . . 367
45.13 Template . . . 368
45.14 Exception Handling . . . 369

VII Interface 371

第 46 章 Introduction . . . 373
46.1 Hardware Interface . . . 373
46.2 Software Interface . . . 373
46.3 API . . . 374

第 47 章 Reuse . . . 377
47.1 Library . . . 377
47.2 Patterns . . . 378
47.3 Frameworks . . . 378

第 48 章 Header file . . . 379
48.1 Interface . . . 379
48.1.1 InitGraphics . . . 384
48.1.2 MovePen,DrawLine . . . 384
48.1.3 DrawArc . . . 385
48.1.4 DrawBox . . . 386
48.1.5 DrawCenteredCircle . . . 388
48.1.6 Coordinates . . . 388
48.2 Contants . . . 389
48.3 Decomposition . . . 389
48.4 Bottom-up . . . 391

第 49 章 Development . . . 397
49.1 Overview . . . 397

49.2 Design . . . 398
49.2.1 Unified . . . 399
49.2.2 Simple . . . 399
49.2.3 Sufficient . . . 400
49.2.4 General . . . 400
49.2.5 Stable . . . 400

第 50 章 Pseudo-random . . . 403
50.1 Introduction . . . 403
50.1.1 Randomness . . . 403
50.1.2 Pseudorandomness . . . 404
50.2 Random Number . . . 406
50.3 Pseudo-random Number . . . 407
50.3.1 Generator . . . 408
50.3.2 Range . . . 409
50.3.3 Universalization . . . 413
50.3.4 Structure . . . 414
50.3.5 Implementation . . . 416
50.3.6 Usage . . . 417
50.3.7 Initialization . . . 419
50.3.8 Evaluating . . . 420
50.4 Pseudo-random Real Number . . . 421
50.5 Simulating Probabilistic Event . . . 422
50.6 Random-number Package . . . 423

VIII Module 429

第 51 章 Introduction . . . 431
51.1 Library . . . 433
51.2 Program . . . 433
51.2.1 Construction . . . 433
51.2.2 Compilation . . . 439
51.2.3 Interpretation . . . 440
51.3 Instruction . . . 441
51.3.1 opcode . . . 443
51.3.2 prefix . . . 443
51.3.3 modifier . . . 443
51.3.4 data . . . 444
51.4 Assembler . . . 446
51.5 Link . . . 449
51.6 Debug . . . 451
51.7 Disassembler . . . 453
51.8 Profiler . . . 454

第 52 章 Pig Latin . . . 459
52.1 Top-down design . . . 459
52.2 Using Pseudocode . . . 460
52.3 Implementing TranslateLine . . . 460
52.4 Space and Punctuation . . . 461

52.5 Refining Words . . . 462
52.6 Designing Token Scanner . . . 463
52.7 Specifying Interface . . . 466

第 53 章 Internal State . . . 469
53.1 Global Variable . . . 470
53.1.1 Initializing Global Variable . . . 470
53.2 Private Variable . . . 472
53.3 Private Function . . . 472

IX Pointer 477

第 54 章 Introduction . . . 479
54.1 Formal Description . . . 479
54.2 Address Value . . . 480
54.3 Memory Allocation . . . 482

第 55 章 Manipulation . . . 485
55.1 Declaration . . . 485
55.2 Operation . . . 486
55.3 Initialization . . . 486
55.3.1 Addressof . . . 487
55.3.2 Dereference . . . 487
55.4 Assignment . . . 488

第 56 章 Parameter . . . 489
56.1 Actual Argument . . . 489
56.2 Call By Reference . . . 490
56.3 Swaplnteger . . . 492
56.4 Returns . . . 493
56.4.1 Return Pointer . . . 493
56.4.2 Multiple Results . . . 494
56.5 Overusing . . . 495
56.5.1 Disable Pointer . . . 495
56.5.2 Protect Argument . . . 496

第 57 章 Array & Pointer . . . 497
57.1 Array Address . . . 497
57.2 Array Name . . . 498
57.3 Array Initialization . . . 500
57.4 Array Parameter . . . 501

第 58 章 Pointer Arithmetic . . . 503
58.1 Increase and Decrease Operators . . . 506
58.2 Pointer Ambiguity . . . 507
58.3 Pointer Comparation . . . 507
58.4 Incrementing and Decrementing Pointers . . . 508

第 59 章 Multidimensional Array . . . 511
59.1 Array Rows . . . 511

59.2 Array Columns . . . 512
59.3 Array Name . . . 512

第 60 章 Storage Allocation . . . 513
60.1 void * . . . 513
60.2 Dynamic Array . . . 514
60.3 Memory Allocation . . . 516
60.4 Memory Deallocation . . . 516
60.5 NULL pointer . . . 516

X String 519

第 61 章 Enumeration Type . . . 521
61.1 Overview . . . 521
61.2 Principle . . . 521
61.3 Implementation . . . 522
61.4 Definition . . . 522
61.4.1 macro . . . 522
61.4.2 typedef . . . 523
61.5 Operation . . . 526

第 62 章 Scalar Type . . . 527

第 63 章 Character Type . . . 529
63.1 Char type . . . 529
63.2 Character set . . . 530
63.3 Character Constant . . . 531
63.4 Escape Sequence . . . 532

第 64 章 Character Operation . . . 535
64.1 ctype.h . . . 536
64.2 Control statements involving characters . . . 537
64.3 Character input and output . . . 538
64.4 sizeof . . . 540

第 65 章 String Type . . . 541
65.1 Overview . . . 541
65.2 C/C++ String . . . 541

第 66 章 String Definition . . . 543
66.1 String Conceptes . . . 543
66.1.1 Char Array . . . 543
66.1.2 Array Pointer . . . 544
66.1.3 Abstract Type . . . 545
66.2 String Parameter . . . 545

第 67 章 String Abstractions . . . 547
67.1 String Literal . . . 547
67.1.1 Escape Sequence . . . 548
67.1.2 Extend String . . . 548

67.1.3 String Storage . . . 549
67.1.4 String Operation . . . 550
67.2 String Constant . . . 550
67.3 String Variable . . . 550
67.3.1 String Declaration . . . 552
67.3.2 String Initialization . . . 553
67.3.3 String Pointer . . . 554
67.4 String I/O . . . 555
67.4.1 String Output . . . 555
67.4.2 String Input . . . 556
67.4.3 Character I/O . . . 557
67.4.4 String Selection . . . 558

第 68 章 String Library . . . 559
68.1 strlib.h . . . 559
68.1.1 String Length . . . 560
68.1.2 Selecting Character . . . 561
68.1.3 String Concatenation . . . 561
68.1.4 Converting characters . . . 562
68.1.5 Extracting String . . . 563
68.1.6 Comparing Strings . . . 563
68.1.7 Searching String . . . 564
68.1.8 Case conversion . . . 566
68.1.9 Numeric conversion . . . 567
68.2 string.h . . . 568
68.2.1 strcpy . . . 571
68.2.2 strncpy . . . 574
68.2.3 strcat . . . 575
68.2.4 strncat . . . 576
68.2.5 strcmp . . . 577
68.2.6 strncmp . . . 578
68.2.7 strlen . . . 578
68.2.8 strchr . . . 581
68.2.9 strrchr . . . 582
68.2.10 strstr . . . 582
68.3 Library Efficiency . . . 583
68.3.1 String Tail . . . 583
68.3.2 String Copy . . . 585
68.3.3 String Invertation . . . 587
68.3.4 String Pass-through . . . 589

第 69 章 String Array . . . 591
69.1 Ragged Array . . . 591
69.2 Command-line Argument . . . 592

XI File 597

第 70 章 Introduction 599
70.1 Overview 599
70.2 EOF 600
70.3 Naming 600
70.4 Storage 601
70.5 Capacity 602
70.6 Format 603
70.7 Access 604

第 71 章 Text I/O 607
71.1 Comparision 607
71.2 Stream 608
71.2.1 FILE Pointer 608
71.2.2 Standard File 609
71.2.3 Redirection 609
71.3 Operation 610
71.3.1 Declaration FILE * 610
71.3.2 Open File 610
71.3.3 I/O Operation 612
71.3.4 Close File 612
71.4 Add File 613
71.5 Temporary File 614
71.6 File Buffer 615
71.6.1 fflush 615
71.6.2 setvbuf 616
71.6.3 setbuf 616
71.7 Remove File 616
71.8 Rename File 617

第 72 章 Character I/O 619
72.1 Update File 621
72.2 Redo Read 623

第 73 章 Line I/O 625

第 74 章 Format I/O 627
74.1 printf 627
74.1.1 Conversion Description 628
74.2 scanf 628
74.3 example 630

XII Error Handling 633

第 75 章 Introduction 635
75.1 Exception Handling 635
75.2 Maths Functions 635

第 76 章 Assert 637

第 77 章 Error 639
77.1 errno 639
77.2 perror 639
77.3 strerror 640

第 78 章 Signal 641
78.1 signal 641
78.2 raise 643

第 79 章 Jump 645
79.1 setjmp 645
79.2 longjmp 645

XIII Advanced 647

第 80 章 Declaration 649
80.1 Introduction 649
80.2 Storage Type 650
80.2.1 auto 650
80.2.2 static 650
80.2.3 extern 651
80.2.4 register 653
80.3 Type Qualifier 653
80.3.1 const 653
80.3.2 volatile 654
80.4 Declarator 654
80.4.1 Array 655
80.4.2 Pointer 655
80.4.3 Function 655
80.4.4 Typedefs 656
80.5 Initializer 656

第 81 章 Pointer 659
81.1 Memory Allocation 659
81.2 Dynamic Allocation 659
81.2.1 malloc 660
81.2.2 calloc 660
81.2.3 realloc 661
81.3 Free-up Storage 662
81.3.1 free 662
81.4 Advanced Pointer 663
81.4.1 NULL Pointer 663
81.4.2 Dangling Pointer 663
81.4.3 Dual Pointer 663
81.4.4 Function Pointer 663
81.5 Dynamic String 663
81.6 Struct Pointer 663
81.6.1 Definition 663
81.7 Dynamic Array 663

81.8 Listed List 664
81.8.1 Declare Node 664
81.8.2 Create Node 664
81.8.3 Select Node 664
81.8.4 Insert Node 664
81.8.5 Track Node 664
81.8.6 Delete Node 664
81.8.7 Sort Node 664

第 82 章 Blocks 665

第 83 章 Include guard 669

第 84 章 #pragma once 673

XIV Library 675

第 85 章 Overview 677
85.1 Header 677
85.2 Macros 678

第 86 章 Introduction 679
86.1 assert.h 679
86.2 ctype.h 679
86.3 errno.h 679
86.4 float.h 679
86.5 limits.h 684
86.6 locale.h 699
86.7 math.h 699
86.8 setjmp.h 706
86.9 signal.h 706
86.10 stdarg.h 706
86.11 stddef.h 706
86.12 stdio.h 707
86.13 stdlib.h 707
86.14 string.h 707
86.15 time.h 707

第 87 章 GNU C Library 709

Part I

Introduction

第 1 章

Overview

程序设计是给出解决特定问题程序的过程，是软件构造活动中的重要组成部分。

程序设计往往以某种程序设计语言为工具，给出这种语言下的程序，具体过程可以包括分析、设计、编码、测试、排错等不同阶段，专业的程序设计人员常被称为程序员。

某种意义上，程序设计的出现甚至早于电子计算机的出现。英国著名诗人拜伦的女儿爱达·勒芙蕾丝曾设计了巴贝奇分析机上计算伯努利数的一个程序，她还创建了循环和子程序的概念。

任何设计活动都是在各种约束条件和相互矛盾的需求之间寻求一种平衡，程序设计也不例外。在计算机技术发展的早期，由于机器资源比较昂贵，程序的时间和空间代价往往是设计关心的主要因素。随着硬件技术的飞速发展和软件规模的日益庞大，程序的结构、可维护性、复用性、可扩展性等因素日益重要。

如果所使用的翻译的机制是将所要翻译的程序代码作为一个整体翻译，并之后运行内部格式，那么这个翻译过程就被称为编译。因此，将人可阅读的程序文本（叫做源代码）作为输入的数据，然后输出可执行文件（object code）[1]。

如果程序代码是在运行时才即时翻译，那么这种翻译机制就被称作解译。经解译的程序运行速度往往比编译的程序慢，但往往更具灵活性，因为它们能够与执行环境互相作用。

另一方面，在计算机技术发展的早期，软件构造活动主要就是程序设计活动。但随着软件技术的发展，软件系统越来越复杂，逐渐分化出许多专用的软件系统，如操作系统、数据库系统、应用服务器，而且这些专用的软件系统愈来愈成为普遍的计算环境的一部分。这种情况下软件构造活动的内容越来越丰富，不再只是纯粹的程序设计，还包括数据库设计、用户界面设计、接口设计、通信协议设计和复杂的系统配置过程。

[1]编译器所输出的可执行文件可以是机器语言，由计算机的中央处理器直接运行，或者是某种模拟器的二进制代码。

第 2 章

Coding

解决问题,是一种兼具创造性、操作性的思维方式和智力活动。对问题的发现和澄清,经常是解决问题的第一步。

用计算机解决问题包括两个概念上不同的步骤。首先,应该构造出一个算法或在解决该问题的已有算法中挑选一个,这个过程称为算法设计(Algorithmic Design),第二步是用程序设计语言将该算法表达为程序,这个过程称为编码(Coding)。

在开发新算法时,都会先用自然语言说明这种算法,尽管这并不像人们想象的那么精确。但是,在通过编码向计算机表达算法时,使用自然语言是不恰当的,于是,要想让一种算法被计算机接受并理解,就得把它翻译成程序设计语言(Programming Language)。

程序设计语言是一种定义计算机程序的人造形式语言,由符号、专用字和一套规则构成。可以把程序设计语言理解为一种被标准化的交流技巧,用于构造计算机能够理解并完成特定任务的指令序列。

程序员使用程序设计语言能够准确地定义计算机所需要使用的数据,并精确地定义在不同情况下所应当采取的行动。

在 von Neumann 体系结构中,CPU 一次读取并执行一条指令,计算机能够直接执行的指令是内置在硬件中的指令,用程序设计语言编写的指令能够被翻译成计算机可以直接执行的指令。

早期程序员们使用机器语言来进行编程,直接操作机器代码,可读性极差。为了便于阅读,就将机器代码以英文字符串来表示,这就是汇编语言。

1953 年 12 月,IBM 公司工程师约翰·巴科斯(J. Backus)因深深体会编写程序很困难,而写了一份备忘录给董事长斯伯特·赫德(Cuthbert Hurd),建议为 IBM704 系统设计全新的电脑语言以提升开发效率。当时 IBM 公司的顾问冯·诺伊曼强烈反对,因为他认为不切实际而且根本不必要。但赫德批准了这项计划。1957 年,IBM 公司开发出第一套 FORTRAN(Formula Translation)语言,在 IBM704 电脑上运作。

高级语言的出现使得计算机程序设计语言不再过度地依赖某种特定的机器或环境。这是因为高级语言在不同的平台上会被编译成不同的机器语言,而不是直接被机器执行。FORTRAN 语言就是世界上第一个种高级语言,其主要目标就是实现平台独立。

通用程序设计语言的出现使程序设计不受特性各异的计算机的影响,而是使用通用的算法概念,这种算法概念可以运用于任何一种计算机系统。在内部,每个计算机系统都能理解一种低级语言,这种低级语言是由它的硬件类型所决定的。例如虽然 Apple Macintosh 计算机和 IBM PC 都能执行用 C 语言等编写出的程序,但它们使用的基本的机器语言是不同的。

有许多用于特殊用途的语言,只在特殊情况下使用。例如,PHP 专门用来显示网页,Perl 更适合文本处理,C 语言被广泛用于操作系统和编译器的开发(所谓的系统编程)。

第 3 章

Document

Doxygen 是一个 C++、C、Java、Objective-C、Python、IDL、Fortran、VHDL、PHP、C# 和 D 语言的文档生成器，可以在大多数操作系统上运行。

使用 Doxygen 可以将软件参考文档直接写在源代码中，而且可以交叉引用文档和源代码，使用户可以很容易地引用实际的源代码。例如，KDE 使用 Doxygen 作为其部分文档且 KDevelop 具有内置的支持。

与 JavaDoc 类似，Doxygen 提取文件从源文件的注解。

除了 Javadoc 的语法，Doxygen 还支持 Qt 使用的文档标记，并可以输出成 HTML、CHM、RTF、PDF、LaTeX、PostScript 或 man pages。

注释文档一般用两个星号标志。

```
/**
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param Description of method's or function's input parameter
 * @param ...
 * @return Description of the return value
 */
```

不过，也可以和 HeaderDoc 一样使用 `*!` 的标志。

```
/*!
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param Description of method's or function's input parameter
 * @param ...
 * @return Description of the return value
 */
```

以下说明如何使 C++ 的源文件产生文件，需要确保参数 `EXTRACT_ALL` 在 Doxyfile 设置为 YES。

```
/**
 * @file
 * @author John Doe <jdoe@example.com>
 * @version 1.0
 *
 * @section LICENSE
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
```

```
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details at
 * http://www.gnu.org/copyleft/gpl.html
 *
 * @section DESCRIPTION
 *
 * The time class represents a moment of time.
 */

class Time {

   public:

     /**
      * Constructor that sets the time to a given value.
      *
      * @param timemillis Number of milliseconds
      *      passed since Jan 1, 1970.
      */
     Time (int timemillis) {
        // the code
     }

     /**
      * Get the current time.
      *
      * @return A time object set to the current time.
      */
     static Time now () {
        // the code
     }
};
```

另一种方法是首选的一些参数的记录如下,这将产生同样的文件。

```
/**
 * Constructor that sets the time to a given value.
 *
 */
Time(int timemillis ///< Number of milliseconds passed since Jan 1, 1970.
)
{
   // the code
}
```

第 4 章

Library

4.1 Introduction

The C standard library is the standard library for the C programming language, as specified in the ANSI C standard. It was developed at the same time as the C POSIX library, which is a superset of it. Since ANSI C was adopted by the International Organization for Standardization, the C standard library is also called the ISO C library.

The original C language provided no built-in functions such as I/O operations, unlike traditional languages such as COBOL and Fortran. Over time, user communities of C shared ideas and implementations of what is now called C standard libraries. Many of these ideas were incorporated eventually into the definition of the standardized C language.

Both Unix and C were created at AT&T's Bell Laboratories in the late 1960s and early 1970s. During the 1970s the C language became increasingly popular. Many universities and organizations began creating their own variants of the language for their own projects. By the beginning of the 1980s compatibility problems between the various C implementations became apparent. In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called C89 standard in 1989. Part of the resulting standard was a set of software libraries called the ANSI C standard library.

POSIX (and SUS) specifies a number of routines that should be available over and above those in the C standard library proper; these are often implemented alongside the C standard library functionality, with varying degrees of closeness. For example, glibc implements functions such as fork within libc.so, but before NPTL was merged into glibc it constituted a separate library with its own linker flag argument. Often, this POSIX-specified functionality will be regarded as part of the library; the C library proper may be identified as the ANSI or ISO C library.

BSD libc is an implementation of C standard library used by BSD Operating Systems such as FreeBSD, NetBSD and OpenBSD. It first appeared in 4.4BSD, which was released in 1994. BSD libc has some extensions that are not defined in the original standard. Some of the extensions of BSD libc are:

- sys/tree.h - contains an implementation of Red-black tree and Splay tree.
- sys/queue.h -Implementations of Linked list, queues, tail queue, etc.
- fgetln() - defined in stdio.h. This can be used to read a file line by line.
- fts.h - contains some functions to traverse a file hierarchy.
- db.h - some functions to connect to the Berkeley DB.
- strlcat() and strlcpy() - secure alternates for strncat() and strncpy().
- err.h - contains some functions to print formatted error messages.
- vis.h - contains the vis() function. This function is used to display non-printable characters in a visual format.

Some languages include the functionality of the standard C library in their own libraries. The library may be adapted to better suit the language's structure, but the operation semantics are kept similar. The C++ language, for example, includes the functionality of the C standard library in the namespace std (e.g., std::printf, std::atoi, std::feof), in header files with similar names to the C ones (cstdio, cmath, cstdlib, etc.). Other languages that take similar approaches are D and the main implementation of Python known as CPython. In the latter, for example, the built-in file objects are defined as "implemented using C's stdio package", so that the available operations (open, read, write, etc.) are expected to have the same behavior as the corresponding C functions.

The C standard library is small compared to the standard libraries of some other languages. The C library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O. It does not include a standard set of "container types" like the C++ Standard Template Library, let alone the complete graphical user interface (GUI) toolkits, networking tools, and profusion of other functionality that Java and the .NET Framework provides as standard. The main advantage of the small standard library is that providing a working ISO C environment is much easier than it is with other languages, and consequently porting C to a new platform is comparatively easy.

4.2 Standard Library

The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services.

The application programming interface (API) of the C standard library is declared in a number of header files. Each header file contains one or more function declarations, data type definitions, and macros.

Unix-like systems typically have a C library in shared library form, but the header files (and compiler toolchain) may be absent from an installation so C development may not be possible. The C library is considered part of the operating system on Unix-like systems. The C functions, including the ISO C standard ones, are widely used by programs, and are regarded as if they were not only an implementation of something in the C language, but also de facto part of the operating system interface. Unix-like operating systems generally cannot function if the C library is erased.

On Microsoft Windows, the core system dynamic libraries (DLLs) provide an implementation of the C standard library for the Visual C++ compiler v6.0, the C standard library for newer versions of the Visual C++ compiler is provided by each compiler individually, as well as redistributable packages. Compiled applications written in C are either statically linked with a C library, or linked to a dynamic version of the library that is shipped with these applications, rather than relied upon to be present on the targeted systems. Functions in a compiler's C library are not regarded as interfaces to Windows.

Many other implementations exist, provided with both various operating systems and C compilers. Although there exist too many implementations to list, some popular implementations follow:

- BSD libc, implementations distributed under BSD operating systems.
- GNU C Library, used in GNU/Linux and GNU/HURD.
- Microsoft C Run-time Library, part of Microsoft Visual C++
- dietlibc, an alternative small implementation of the C standard library (MMU-less)
- uClibc, a C standard library for embedded Linux systems (MMU-less)
- Newlib, a C standard library for embedded systems (MMU-less)[5]
- klibc, primarily for booting Linux systems.
- EGLIBC, variant of glibc for embedded systems.
- musl, another lightweight C standard library implementation for Linux systems[6]
- Bionic, originally developed by Google for the Android embedded system operating system, derived from BSD libc.

After a long period of stability, three new header files (iso646.h, wchar.h, and wctype.h) were added with Normative Addendum 1 (NA1), an addition to the C Standard ratified in 1995. Six more header files (complex.h, fenv.h, inttypes.h, stdbool.h, stdint.h, and tgmath.h) were added with C99, a revision to the C Standard published in 1999, and five more files (stdalign.h, stdatomic.h, stdnoreturn.h, threads.h, and uchar.h) with C11 in 2011. In total, there are now 29 header files:

Table 4.1: C Standard Libraries Header files

Name	Description
<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers.
<ctype.h>	Defines set of functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>	For testing error codes reported by library functions.
<fenv.h>	Defines a set of functions for controlling floating-point environment.
<float.h>	Defines macro constants specifying the implementation-specific properties of the floating-point library.
<inttypes.h>	Defines exact width integer types.
<iso646.h>	Defines several macros that implement alternative ways to express several standard tokens. For programming in ISO 646 variant character sets.
<limits.h>	Defines macro constants specifying the implementation-specific properties of the integer types.
<locale.h>	Defines localization functions.
<math.h>	Defines common mathematical functions.
<setjmp.h>	Declares the macros setjmp and longjmp, which are used for non-local exits.
<signal.h>	Defines signal handling functions.
<stdalign.h>	For querying and specifying the alignment of objects.
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdatomic.h>	For atomic operations on data shared between threads.
<stdbool.h>	Defines a boolean data type.
<stddef.h>	Defines several useful types and macros.
<stdint.h>	Defines exact width integer types.
<stdio.h>	Defines core input and output functions.
<stdlib.h>	Defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions.
<stdnoreturn.h>	For specifying non-returning functions.
<string.h>	Defines string handling functions.
<tgmath.h>	Defines type-generic mathematical functions.
<threads.h>	Defines functions for managing multiple Threads as well as mutexes and condition variables.
<time.h>	Defines date and time handling functions.
<uchar.h>	Types and functions for manipulating Unicode characters.
<wchar.h>	Defines wide string handling functions.
<wctype.h>	Defines set of functions used to classify wide characters by their types or to convert between upper and lower case.

Three of the header files (complex.h, stdatomic.h, threads.h) are conditional features that implementations need not support.

Additionally, under Linux and FreeBSD, the mathematical functions (as declared in math.h) are bundled separately in the mathematical library `libm`. If any of them are used, the linker must be given the directive `-lm`.

The POSIX standard added several nonstandard C headers for Unix-specific functionality. Many have found their way to other architectures. Examples include unistd.h and signal.h. A number of other groups are using other nonstandard headers - most flavors of Linux have alloca.h and HP OpenVMS has the va_count() function.

Some compilers (for example, GCC) provide built-in versions of many of the functions in the C standard library; that is, the implementations of the functions are written into the compiled object file, and the program calls the built-in versions instead of the functions in the C library shared object file. This reduces function call overhead, especially if function calls are replaced with inline variants, and allows other forms of optimization (as the compiler knows the control-flow characteristics of the built-in variants), but may cause confusion when debugging (for example, the built-in versions cannot be replaced with instrumented variants).

However, the built-in functions must behave like ordinary functions in accordance with ISO C. The main implication is that the program must be able to create a pointer to these functions by taking their address, and invoke the function by means of that pointer. If two pointers to the same function are derived in two different translation unit in the program, these two pointers must compare equal; that is, the address comes by resolving the name of the function, which has external (program-wide) linkage.

On Unix-like systems, the authoritative documentation of the actually implemented API is provided in form of man pages. On most systems, man pages on standard library functions are in section 3; section 7 may contain some more generic pages on underlying concepts (e.g. `man 7 math_error` in Linux).

According to the C standard the macro `__STDC_HOSTED__` shall be defined to 1 if the implementation is hosted. A hosted implementation has all the headers specified by the C standard. An implementation can also be freestanding which means that these headers will not be present. If an implementation is freestanding, it shall define `__STDC_HOSTED__` to 0.

Some functions in the C standard library have been notorious for having buffer overflow vulnerabilities and generally encouraging buggy programming ever since their adoption. The most criticized items are:

- string-manipulation routines, including strcpy() and strcat(), for lack of bounds checking and possible buffer overflows if the bounds aren't checked manually;
- string routines in general, for side-effects, encouraging irresponsible buffer usage, not always guaranteeing valid null-terminated output, linear length calculation;
- printf() family routines, for spoiling the execution stack when the format string doesn't match the arguments given. This fundamental flaw created an entire class of attacks: format string attacks;
- gets() and scanf() family I/O routines, for lack of (either any or easy) input length checking.

Except the extreme case with gets(), all the security vulnerabilities can be avoided by introducing auxiliary code to perform memory management, bounds checking, input checking, etc. This is often done in form of wrappers that make standard library functions safer and easier to use. This dates back to as early as The Practice of Programming book by B. Kernighan and R. Pike where the authors commonly use wrappers that print error messages and quit the program if an error occurs.

The ISO C committee published Technical reports TR 24731-1 and is working on TR 24731-2 to propose adoption of some functions with bounds checking and automatic buffer allocation, correspondingly. The former has met severe criticism with some praise, the latter received mixed responses. Despite this, TR 24731-1 has been implemented into Microsoft's C standard library and its compiler issues warnings when using old 'insecure' functions.

The mktemp() and strerror() routines are criticized for being thread unsafe and otherwise vulnerable to race conditions.

The error handling of the functions in the C standard library is not consistent and sometimes confusing. This

can be fairly well summarized by the Linux manual page math_error which says: The current (version 2.8) situation under glibc is messy. Most (but not all) functions raise exceptions on errors. Some also set errno. A few functions set errno, but don't raise an exception. Very few functions do neither.

4.3 GNU Library

The GNU C Library[1], commonly known as *glibc*, is the GNU Project's implementation of the C standard library. Originally written by the Free Software Foundation (FSF) for the GNU operating system, the library's development had been overseen by a committee since 2001, with Ulrich Drepper as the lead contributor and maintainer. In March 2012, the steering committee voted to disband itself, in favor of a community-driven development process, with Ryan Arnold, Maxim Kuvyrkov, Joseph Myers, Carlos O'Donell, and Alexandre Oliva as non-decision making project stewards.

The GNU C Library released under the GNU Lesser General Public License, glibc was initially written mostly by Roland McGrath, working for the Free Software Foundation (FSF) in the 1980s, so glibc is free software.

In February 1988, FSF described glibc as having nearly completed the functionality required by ANSI C. By 1992, it had the ANSI C-1989 and POSIX.1-1990 functions implemented and work was under way on POSIX.2.

In the early 1990s, the developers of the Linux kernel have had a temporary fork of glibc—called "Linux libc", was maintained separately for years and released versions 2 through 5. When FSF released glibc 2.0 in January 1997, it had much more complete POSIX standards compliance, better internationalisation and multilingual function, IPv6 capability, 64-bit data access, facilities for multithreaded applications, future version compatibility, and the code was more portable. At this point, the Linux kernel developers discontinued their fork and returned to using FSF's glibc.

The last used version of Linux libc used the internal name (soname) libc.so.5. Following on from this, glibc 2.x on Linux uses the soname libc.so.6[1] (Alpha and IA64 architectures now use libc.so.6.1, instead). The soname is often abbreviated as libc6 (for example in the package name in Debian) following the normal conventions for libraries.

```
[root@theqiong ~]# whereis libc.so.6
libc.so: /usr/lib/libc.so.6 /usr/lib64/libc.so.6 /usr/lib64/libc.so
[root@theqiong ~]# file /usr/lib/libc.so.6
/usr/lib/libc.so.6: symbolic link to `libc-2.18.so'
[root@theqiong ~]# file /usr/lib64/libc.so.6
/usr/lib64/libc.so.6: symbolic link to `libc-2.18.so'
[root@theqiong ~]# whereis libc-2.18.so
libc-2.18: /usr/lib/libc-2.18.so /usr/lib64/libc-2.18.so
[root@theqiong ~]# file /usr/lib/libc-2.18.so
/usr/lib/libc-2.18.so: ELF 32-bit LSB shared object, Intel 80386,
version 1 (GNU/Linux), dynamically linked (uses shared libs),
BuildID[sha1]=c7e8cff8d76361681a89d78cb6b289e5f105cba4,
for GNU/Linux 2.6.32, not stripped
[root@theqiong ~]# file /usr/lib64/libc-2.18.so
/usr/lib64/libc-2.18.so: ELF 64-bit LSB shared object, x86-64,
version 1 (GNU/Linux), dynamically linked (uses shared libs),
BuildID[sha1]=efc76c94d401b3b7f0f8ecb893c829d82f10e4b2,
for GNU/Linux 2.6.32, not stripped
```

According to Richard Stallman, the changes that had been made in Linux libc could not be merged back into glibc because the authorship status of that code was unclear and the GNU project is quite strict about recording copyright and authors.

[1] For most modern operating systems, the version of glibc can be obtained by executing the lib file (for example, /lib/libc.so.6).

glibc provides the functionality required by the Single UNIX Specification, POSIX (1c, 1d, and 1j) and some of the functionality required by ISO C11, ISO C99, Berkeley Unix (BSD) interfaces, the System V Interface Definition (SVID) and the X/Open Portability Guide (XPG), Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems along with all X/Open UNIX extensions.

In addition, glibc also provides extensions that have been deemed useful or necessary while developing GNU.

Now, glibc is used in systems that run many different kernels and different hardware architectures. Its most common use is in systems using the Linux kernel on x86 hardware, however, officially supported hardware includes: x86, Motorola 680x0, DEC Alpha, PowerPC, ETRAX CRIS, s390, and SPARC. It officially supports the Hurd and Linux kernels. Additionally, there are heavily patched versions that run on the kernels of FreeBSD and NetBSD (from which Debian GNU/kFreeBSD and Debian GNU/NetBSD systems are built, respectively), as well as the kernel of OpenSolaris.[13] It is also used (in an edited form) and named libroot.so in BeOS and Haiku.

glibc has been criticized as being "bloated" and slower than other libraries in the past, e.g. by Linus Torvalds and embedded Linux programmers. For this reason, several alternative C standard libraries have been created which emphasize a smaller footprint. Alternative libcs are Bionic (based mostly on libc from BSD and used in Android), dietlibc, uClibc, Newlib, Klibc, musl, and EGLIBC (used in Debian and Ubuntu).

However, many small-device projects use GNU libc over the smaller alternatives because of its application support, standards compliance, and completeness. Examples include Openmoko[16] and Familiar Linux for iPaq handhelds (when using the GPE display software).

在计算机科学中,库是用于开发软件的子程序(函数)集合。

- 库不是独立的可执行程序;
- 库文件是预先编译链接好的文件;
- 库用于向其他程序提供服务。

库链接是指把一个或多个库包括到程序中,有两种链接形式:静态链接和动态链接。相应的,静态链接的库叫做静态库,动态链接的库叫做动态库。其中,可以动态链接的库包括 dynamic link library(Windows DLL)和在 UNIX 或 Linux 中的 Shared Library(GNU C Library)。

GNU C Library(glibc)是 GNU 发布的 libc 库(即 C 运行库)。本质上,glibc 是程序运行时使用到的一些 API 集合,它们一般是已预先编译好并以二进制代码形式存在 Linux 类系统中。

```
[root@theqiong ~]# file /usr/lib/libc-2.18.so
/usr/lib/libc-2.18.so: ELF 32-bit LSB shared object, Intel 80386,
version 1 (GNU/Linux), dynamically linked (uses shared libs),
BuildID[sha1]=c7e8cff8d76361681a89d78cb6b289e5f105cba4,
for GNU/Linux 2.6.32, not stripped
[root@theqiong ~]# file /usr/lib64/libc-2.18.so
/usr/lib64/libc-2.18.so: ELF 64-bit LSB shared object, x86-64,
version 1 (GNU/Linux), dynamically linked (uses shared libs),
BuildID[sha1]=efc76c94d401b3b7f0f8ecb893c829d82f10e4b2,
for GNU/Linux 2.6.32, not stripped
```

- 静态链接是由链接器在链接时将库的内容加入到可执行程序中的做法。链接器是一个独立程序,用于将一个或多个库或目标文件(先前由编译器或汇编器生成)链接到一块生成可执行程序。

  静态链接的最大缺点是生成的可执行文件太大,需要更多的系统资源,在装入内存时也会消耗更多的时间。
- 动态链接指的是在可执行文件装载时或运行时由操作系统的装载程序加载库。

  大多数情况下,同一时间多个应用可以使用一个库的同一份拷贝,操作系统不需要加载这个库的多个实例。

大多数操作系统将解析外部引用(比如库)作为加载过程的一部分。

- load-time loading

  使用动态链接的可执行文件包含一个叫做 import directory 的表,该表的每一项包含一个库的名字。根据表中记录的名字,装载程序在系统中搜索需要的库,然后将其加载到内存中预先不确定的位置,之后根据加载库后确定的库的地址更新可执行程序。可执行程序根据更新后的库信息调用库中的函数或引用库中的数据。这种类型的动态加载称为装载(load-time)时加载,被包括 Windows 和 Linux 的大多数系统采用。装载程序在加载应用软件时要完成的最复杂的工作之一就是加载时链接。
- run-time loading

  其他操作系统可能在运行时解析引用。在这些系统上,可执行程序调用操作系统 API,将库的名字、函数在库中的编号和函数参数一同传递。操作系统负责立即解析然后代表应用调用合适的函数。这种动态链接叫做运行时链接,因为每个调用都会有系统开销,运行时链接要慢得多,对应用的性能有负面影响。现代操作系统已经很少使用运行时链接。

动态链接的最大缺点是可执行程序依赖分别存储的库文件才能正确执行。如果库文件被删除、移动、重命名或者被替换为不兼容的版本,那么可执行程序就可能工作不正常,这就是常说的 DLL-hell。

第 5 章

Preprocess

5.1 Introduction

在计算机科学中，预处理器（Preprocessor）是处理程序中的输入数据，并产生能用来输入到其他程序的数据的程序。

预处理器通常会和编译器整合在一起，并且在编译前对源代码进行转换并输出给编译器，编译器检查程序是否有错误，并将其翻译为目标代码（机器指令）。

在 C 语言较早的时期，预处理器是一个单独的程序，并将它的输出提供给编译器，后来预处理器被集成到编译器中来提高编译速度。

预处理器认为所有未定义的常量的值为 0，但并不完全按照编译器的方式处理常量表达式，因此在预处理阶段非常有可能产生非法的输出，后来预处理器的许多功能被其它语言直接内建。

预处理器所作处理的数量和种类依赖于预处理器的类型，一些预处理器只能够执行相对简单的文本替换和宏展开，而另一些则有着完全成熟的编程语言的能力。

一个来自计算机编程的常见的例子是在进行下一步编译之前，对源代码执行处理。在一些计算机语言（例如 C 语言）中有一个叫做预处理的翻译阶段，其中词法预处理器是最低级的预处理器，因为它们只需要词法分析。也就是说，预处理器在语法分析处理之前，根据用户定义的规则进行简单的词法单元替换。

最常见的例子是 C 预处理器，采用以 '#' 为行首的预处理指令（preprocess directive），C 语言（C++）因为依赖预处理器而不同于其他的编程语言。

C 预处理器的指令和 C 语言的语法具有弱相关性，因此也可以用来处理其他文本文件。预处理器中的词法预处理器可以产生宏替换，包含其他文件的文本，并且条件性地编译或者包含文件，其他词法预处理器包括一般用途的 m4、跨平台构建系统（比如 autoconf）和宏处理器 GEMA 来操作上下文模式。

句法预处理器是由 Lisp 家族语言引进的，可以根据若干用户定义的规则转换语法树。对于某些程序语言（Lisp 和 OCaml 等），这些规则是使用同一种语言来写的 (compile-time reflection)。

某些编程语言依靠一个完全的外部语言来定义转换，例如 XSLT 处理器处理 XML 的方式，或与静态类型的对应语言 CDuse。

静态处理器常被用来自定编程语言的语法，并透过增加新的 primitives 或嵌入特定领域语言（Domain-Specific Programming Language）到一般用途的语言里来扩充。

关于自定义语法的一个好例子是在 OCaml 编程语言里两个不同语法的存在。程序可能平常地由“正常语法”或“校正过的语法”写成，并且按需求由两者之一进行程序优化。类似地，一些 OCaml 语言写成的程序借由新运算符的增加来自定化语言的语法。

对于从宏扩充语言最好的范例可在 LISP 语言家族里找到，LISP 语言本身就是简单的动态类型核心模块，Scheme 或 Common Lisp 的标准分配允许命令或面向对象的程序编辑，静态类型亦如此。几乎所有这些特性都由语法预处理执行，“宏扩充”编译阶段由 LISP 的编译器处理，这仍然可以视为预处理的一种形式，因为它在编译阶段前就进行了。

类似地，静态检查、类型安全正规表式或代码生成可能透过宏被加入到 OCaml 的语法和语义里，如同微线程（亦称为协程或 fibers）、单子或透明的 XML 操作。

任何“一般目的预处理器”(例如 M4)都可以当成模版引擎(template engine)使用。

现代 C 语言编程风格呼吁减少对预处理器的依赖, C++ 对语言的改变使得可以更进一步限制预处理器的使用。

- 宏替换被显示内联和模板替代。
- 包含变为编译期导入 (compile-time import)(这需要在目标代码中预先保存类型信息, 使用这个功能无法改进一个语言)。
- 条件编译被 if-then-else 和死代码消除替代。

5.2 Preprocessor

大多数预处理指令属于下面 3 种类型。

- 宏定义
  #define 指令定义一个宏, #undef 指令删除一个宏定义。
- 文件包含
  #include 指令导入一个指定的文件到源代码中。
- 条件编译
  #if、#ifdef、#ifndef、#elif、#else 和 #endif 指令可以根据编译器能够测试的条件来将代码包含到程序中或排除在程序外。

C 语言预处理器(cpp)实际上也可以认为它是 C 语言的宏预处理器[1], 可以把头文件、宏扩展、条件编译和线性控制信息等包含进入源代码。

现在, GNU 的汇编器已经具备了宏功能, 大多数高级编程语言也具备了条件编译和包含机制。在这些功能都无法工作时, GNU M4 可以作为一种一般目的预处理器来使用。

C 语言预处理器的行为是由指令控制的, 这些指令是由以 # 开头的命令, 可以对 C、C++ 和 Objective-C 代码进行预处理。

根据 C 语言标准, 预处理阶段可以分为四个或八个不同的转换阶段:

1. Trigraph replacement(三元组替换)
   The preprocessor replaces trigraph sequences with the characters they represent.
2. Line splicing(行拼接)
   Physical source lines that are continued with escaped newline sequences are spliced to form logical lines.
3. Tokenization(分词)
   The preprocessor breaks the result into preprocessing tokens and whitespace. It replaces comments with whitespace.
4. Macro expansion and directive handling(宏展开和指令处理)
   Preprocessing directive lines, including file inclusion and conditional compilation, are executed.

C 预处理器处理源代码文件, 替换其中的预处理指令, 并对程序注释进行相应的处理, 其输出将被直接交给编译器。

通过在编译器中打开特定的选项(在 UNIX 环境下通常是-P), 可以仅产生预处理器的输出。特别值得注意的是, C 预处理器将每一处注释(包括预处理注释)都替换为一个空格字符, 以及删除不必要的空白字符(包括在每一行开始用于缩进的空格符和制表符等)。

在许多 C 语言的实现中, C 预处理器是由 C 编译器在转换阶段调用的独立程序, 可以用于定义常量和包含接口文件等。

```
#include <stdio.h>

main()
{
```

[1] The preprocessor simultaneously expands macros and, in the 1999 version of the C standard, handles `_Pragma` operators.

```
  printf("Hello, world.\n");
}
$ gcc -E -P hello.c
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;
typedef long int __quad_t;
typedef unsigned long int __u_quad_t;
typedef unsigned long int __dev_t;
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned long int __ino64_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;
typedef long int __off_t;
typedef long int __off64_t;
typedef int __pid_t;
typedef struct { int __val[2]; } __fsid_t;
typedef long int __clock_t;
typedef unsigned long int __rlim_t;
typedef unsigned long int __rlim64_t;
typedef unsigned int __id_t;
typedef long int __time_t;
typedef unsigned int __useconds_t;
typedef long int __suseconds_t;
typedef int __daddr_t;
typedef int __key_t;
typedef int __clockid_t;
typedef void * __timer_t;
typedef long int __blksize_t;
typedef long int __blkcnt_t;
typedef long int __blkcnt64_t;
typedef unsigned long int __fsblkcnt_t;
typedef unsigned long int __fsblkcnt64_t;
typedef unsigned long int __fsfilcnt_t;
typedef unsigned long int __fsfilcnt64_t;
typedef long int __fsword_t;
typedef long int __ssize_t;
typedef long int __syscall_slong_t;
typedef unsigned long int __syscall_ulong_t;
typedef __off64_t __loff_t;
typedef __quad_t *__qaddr_t;
typedef char *__caddr_t;
typedef long int __intptr_t;
typedef unsigned int __socklen_t;
struct _IO_FILE;
```

```
typedef struct _IO_FILE FILE;


typedef struct _IO_FILE __FILE;
typedef struct
{
  int __count;
  union
  {
    unsigned int __wch;
    char __wchb[4];
  } __value;
} __mbstate_t;
typedef struct
{
  __off_t __pos;
  __mbstate_t __state;
} _G_fpos_t;
typedef struct
{
  __off64_t __pos;
  __mbstate_t __state;
} _G_fpos64_t;
typedef __builtin_va_list __gnuc_va_list;
struct _IO_jump_t; struct _IO_FILE;
typedef void _IO_lock_t;
struct _IO_marker {
  struct _IO_marker *_next;
  struct _IO_FILE *_sbuf;
  int _pos;
};
enum __codecvt_result
{
  __codecvt_ok,
  __codecvt_partial,
  __codecvt_error,
  __codecvt_noconv
};
struct _IO_FILE {
  int _flags;
  char* _IO_read_ptr;
  char* _IO_read_end;
  char* _IO_read_base;
  char* _IO_write_base;
  char* _IO_write_ptr;
  char* _IO_write_end;
  char* _IO_buf_base;
  char* _IO_buf_end;
  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int _fileno;
  int _flags2;
  __off_t _old_offset;
  unsigned short _cur_column;
  signed char _vtable_offset;
```

```
  char _shortbuf[1];
  _IO_lock_t *_lock;
  __off64_t _offset;
  void *__pad1;
  void *__pad2;
  void *__pad3;
  void *__pad4;
  size_t __pad5;
  int _mode;
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
};
typedef struct _IO_FILE _IO_FILE;
struct _IO_FILE_plus;
extern struct _IO_FILE_plus _IO_2_1_stdin_;
extern struct _IO_FILE_plus _IO_2_1_stdout_;
extern struct _IO_FILE_plus _IO_2_1_stderr_;
typedef __ssize_t __io_read_fn (void *__cookie, char *__buf, size_t __nbytes);
typedef __ssize_t __io_write_fn (void *__cookie, const char *__buf,
    size_t __n);
typedef int __io_seek_fn (void *__cookie, __off64_t *__pos, int __w);
typedef int __io_close_fn (void *__cookie);
extern int __underflow (_IO_FILE *);
extern int __uflow (_IO_FILE *);
extern int __overflow (_IO_FILE *, int);
extern int _IO_getc (_IO_FILE *__fp);
extern int _IO_putc (int __c, _IO_FILE *__fp);
extern int _IO_feof (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_ferror (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_peekc_locked (_IO_FILE *__fp);
extern void _IO_flockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
extern void _IO_funlockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_ftrylockfile (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_vfscanf (_IO_FILE * __restrict, const char * __restrict,
  __gnuc_va_list, int *__restrict);
extern int _IO_vfprintf (_IO_FILE *__restrict, const char *__restrict,
   __gnuc_va_list);
extern __ssize_t _IO_padn (_IO_FILE *, int, __ssize_t);
extern size_t _IO_sgetn (_IO_FILE *, void *, size_t);
extern __off64_t _IO_seekoff (_IO_FILE *, __off64_t, int, int);
extern __off64_t _IO_seekpos (_IO_FILE *, __off64_t, int);
extern void _IO_free_backup_area (_IO_FILE *) __attribute__ ((__nothrow__ , __leaf__));
typedef __gnuc_va_list va_list;
typedef __off_t off_t;
typedef __ssize_t ssize_t;

typedef _G_fpos_t fpos_t;

extern struct _IO_FILE *stdin;
extern struct _IO_FILE *stdout;
extern struct _IO_FILE *stderr;

extern int remove (const char *__filename) __attribute__ ((__nothrow__ , __leaf__));
extern int rename (const char *__old, const char *__new) __attribute__ ((__nothrow__ ,
    __leaf__));

extern int renameat (int __oldfd, const char *__old, int __newfd,
     const char *__new) __attribute__ ((__nothrow__ , __leaf__));
```

```
extern FILE *tmpfile (void) ;
extern char *tmpnam (char *__s) __attribute__ ((__nothrow__ , __leaf__)) ;

extern char *tmpnam_r (char *__s) __attribute__ ((__nothrow__ , __leaf__)) ;
extern char *tempnam (const char *__dir, const char *__pfx)
    __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__malloc__)) ;

extern int fclose (FILE *__stream);
extern int fflush (FILE *__stream);

extern int fflush_unlocked (FILE *__stream);

extern FILE *fopen (const char *__restrict __filename,
     const char *__restrict __modes) ;
extern FILE *freopen (const char *__restrict __filename,
      const char *__restrict __modes,
      FILE *__restrict __stream) ;

extern FILE *fdopen (int __fd, const char *__modes) __attribute__ ((__nothrow__ , __leaf__
    )) ;
extern FILE *fmemopen (void *__s, size_t __len, const char *__modes)
  __attribute__ ((__nothrow__ , __leaf__)) ;
extern FILE *open_memstream (char **__bufloc, size_t *__sizeloc) __attribute__ ((
    __nothrow__ , __leaf__)) ;

extern void setbuf (FILE *__restrict __stream, char *__restrict __buf) __attribute__ ((
    __nothrow__ , __leaf__));
extern int setvbuf (FILE *__restrict __stream, char *__restrict __buf,
     int __modes, size_t __n) __attribute__ ((__nothrow__ , __leaf__));

extern void setbuffer (FILE *__restrict __stream, char *__restrict __buf,
       size_t __size) __attribute__ ((__nothrow__ , __leaf__));
extern void setlinebuf (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern int fprintf (FILE *__restrict __stream,
     const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);
extern int sprintf (char *__restrict __s,
     const char *__restrict __format, ...) __attribute__ ((__nothrow__));
extern int vfprintf (FILE *__restrict __s, const char *__restrict __format,
      __gnuc_va_list __arg);
extern int vprintf (const char *__restrict __format, __gnuc_va_list __arg);
extern int vsprintf (char *__restrict __s, const char *__restrict __format,
      __gnuc_va_list __arg) __attribute__ ((__nothrow__));


extern int snprintf (char *__restrict __s, size_t __maxlen,
      const char *__restrict __format, ...)
    __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 4)));
extern int vsnprintf (char *__restrict __s, size_t __maxlen,
       const char *__restrict __format, __gnuc_va_list __arg)
    __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3, 0)));

extern int vdprintf (int __fd, const char *__restrict __fmt,
      __gnuc_va_list __arg)
    __attribute__ ((__format__ (__printf__, 2, 0)));
extern int dprintf (int __fd, const char *__restrict __fmt, ...)
    __attribute__ ((__format__ (__printf__, 2, 3)));
```

```
extern int fscanf (FILE *__restrict __stream,
    const char *__restrict __format, ...) ;
extern int scanf (const char *__restrict __format, ...) ;
extern int sscanf (const char *__restrict __s,
    const char *__restrict __format, ...) __attribute__ ((__nothrow__ , __leaf__));
extern int fscanf (FILE *__restrict __stream, const char *__restrict __format, ...)
    __asm__ ("" "__isoc99_fscanf") ;
extern int scanf (const char *__restrict __format, ...) __asm__ ("" "__isoc99_scanf") ;
extern int sscanf (const char *__restrict __s, const char *__restrict __format, ...)
    __asm__ ("" "__isoc99_sscanf") __attribute__ ((__nothrow__ , __leaf__));


extern int vfscanf (FILE *__restrict __s, const char *__restrict __format,
     __gnuc_va_list __arg)
    __attribute__ ((__format__ (__scanf__, 2, 0))) ;
extern int vscanf (const char *__restrict __format, __gnuc_va_list __arg)
    __attribute__ ((__format__ (__scanf__, 1, 0))) ;
extern int vsscanf (const char *__restrict __s,
     const char *__restrict __format, __gnuc_va_list __arg)
    __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__format__ (__scanf__, 2, 0)))
        ;
extern int vfscanf (FILE *__restrict __s, const char *__restrict __format, __gnuc_va_list
    __arg) __asm__ ("" "__isoc99_vfscanf")
    __attribute__ ((__format__ (__scanf__, 2, 0))) ;
extern int vscanf (const char *__restrict __format, __gnuc_va_list __arg) __asm__ ("" "
    __isoc99_vscanf")
    __attribute__ ((__format__ (__scanf__, 1, 0))) ;
extern int vsscanf (const char *__restrict __s, const char *__restrict __format,
    __gnuc_va_list __arg) __asm__ ("" "__isoc99_vsscanf") __attribute__ ((__nothrow__ ,
    __leaf__))
    __attribute__ ((__format__ (__scanf__, 2, 0)));


extern int fgetc (FILE *__stream);
extern int getc (FILE *__stream);
extern int getchar (void);

extern int getc_unlocked (FILE *__stream);
extern int getchar_unlocked (void);
extern int fgetc_unlocked (FILE *__stream);

extern int fputc (int __c, FILE *__stream);
extern int putc (int __c, FILE *__stream);
extern int putchar (int __c);

extern int fputc_unlocked (int __c, FILE *__stream);
extern int putc_unlocked (int __c, FILE *__stream);
extern int putchar_unlocked (int __c);
extern int getw (FILE *__stream);
extern int putw (int __w, FILE *__stream);

extern char *fgets (char *__restrict __s, int __n, FILE *__restrict __stream)
    ;
extern char *gets (char *__s) __attribute__ ((__deprecated__));

extern __ssize_t __getdelim (char **__restrict __lineptr,
       size_t *__restrict __n, int __delimiter,
```

```
       FILE *__restrict __stream) ;
extern __ssize_t getdelim (char **__restrict __lineptr,
      size_t *__restrict __n, int __delimiter,
      FILE *__restrict __stream) ;
extern __ssize_t getline (char **__restrict __lineptr,
     size_t *__restrict __n,
     FILE *__restrict __stream) ;

extern int fputs (const char *__restrict __s, FILE *__restrict __stream);
extern int puts (const char *__s);
extern int ungetc (int __c, FILE *__stream);
extern size_t fread (void *__restrict __ptr, size_t __size,
     size_t __n, FILE *__restrict __stream) ;
extern size_t fwrite (const void *__restrict __ptr, size_t __size,
      size_t __n, FILE *__restrict __s);

extern size_t fread_unlocked (void *__restrict __ptr, size_t __size,
       size_t __n, FILE *__restrict __stream) ;
extern size_t fwrite_unlocked (const void *__restrict __ptr, size_t __size,
       size_t __n, FILE *__restrict __stream);

extern int fseek (FILE *__stream, long int __off, int __whence);
extern long int ftell (FILE *__stream) ;
extern void rewind (FILE *__stream);

extern int fseeko (FILE *__stream, __off_t __off, int __whence);
extern __off_t ftello (FILE *__stream) ;

extern int fgetpos (FILE *__restrict __stream, fpos_t *__restrict __pos);
extern int fsetpos (FILE *__stream, const fpos_t *__pos);


extern void clearerr (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int feof (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int ferror (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;

extern void clearerr_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int feof_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int ferror_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;

extern void perror (const char *__s);

extern int sys_nerr;
extern const char *const sys_errlist[];
extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern FILE *popen (const char *__command, const char *__modes) ;
extern int pclose (FILE *__stream);
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

main()
{
  printf("Hello, world.\n");
}
```

- 预处理指令都以 # 开始，在 # 后是指令名，接着是指令所需要的其他信息。
- 在预处理指令的符号之间可以插入任意数量的空格或横向制表符。
- 指令总是在第一个换行符处结束。如果需要在下一行继续指令，则必须在当前行的末尾使用\字符。

  下面的预处理指令示例定义了一个宏来表示硬盘的容量。

```
#define DISK_CAPACITY (SIDES * \
                                TRACK_PER_SIDE * \
                                SECTORS_PER_TRACK * \
                                BYTES_PER_SECTOR)
```

- 预处理指令可以出现在源代码中的任何位置。
- 注释可以与预处理指令位于同一行。

5.3 Macros Definition

通常情况下，通过 #define 指令定义的宏都是某一类型的常量，C 预处理器会通过将宏的名字和它的定义存储在一起来响应 #define 指令。

在实际应用中，预处理器常量表达式中的操作数通常为常量、表示常量的宏或 defined 运算符的应用。当宏在后面的程序中被使用到时，C 预处理器会“扩展”宏并将其替换为它所定义的值。

在源代码中可以有两种类型的宏，*object-like* 和 *function-like*，其中 *object-like* 宏不使用参数，而 *function-like* 宏使用参数，因此声明一个标识符为宏的通用语法分别为：

```
// object-like macro
#define <identifier> <replacement token list>
// function-like macro, note parameters
#define <identifier>(<parameter list>) <replacement token list>
```

function-like 宏声明中，在标识符和第一个左括号之间不能有空格。如果出现了空格，宏定义将会被解释成 *object-like* 宏。

如果要取消宏，可以使用 `#undef`：

```
// delete the macro
#undef <identifier>
```

Whenever the identifier appears in the source code it is replaced with the replacement token list, which can be empty. For an identifier declared to be a function-like macro, it is only replaced when the following token is also a left parenthesis that begins the argument list of the macro invocation. The exact procedure followed for expansion of function-like macros with arguments is subtle.

Object-like macros were conventionally used as part of good programming practice to create symbolic names for constants, e.g.

```
#define PI 3.14159
```

An alternative in both C and C++, especially in situations in which a pointer to the number is required, is to apply the const qualifier to a global variable. This causes the value to be stored in memory, instead of being substituted by the preprocessor.

An example of a function-like macro is:

```
#define RADTODEG(x) ((x) * 57.29578)
```

This defines a radians-to-degrees conversion which can be inserted in the code where required, i.e., RADTODEG(34). This is expanded in-place, so that repeated multiplication by the constant is not shown throughout the code. The macro here is written as all uppercase to emphasize that it is a macro, not a compiled function.

The second x is enclosed in its own pair of parentheses to avoid the possibility of incorrect order of operations when it is an expression instead of a single value. For example, the expression `RADTODEG(r + 1)` expands correctly as `((r + 1) * 57.29578)` —without parentheses, `(r + 1 * 57.29578)` gives precedence to the multiplication.

Similarly, the outer pair of parentheses maintain correct order of operation. For example, `1 / RADTODEG(r)` expands to `1 / ((r) * 57.29578)` —without parentheses, `1 / (r) * 57.29578` gives precedence to the division.

C 预处理器已经定义了一些符号，包括 `__FILE__` 和 `__LINE__`，可以扩展到文件和行，例如：

```
// debugging macros so we can pin down message origin at a glance
#define WHERESTR "[file %s, line %d]: "
#define WHEREARG __FILE__, __LINE__
#define DEBUGPRINT2(...) fprintf(stderr, __VA_ARGS__)
#define DEBUGPRINT(_fmt, ...) DEBUGPRINT2(WHERESTR _fmt, WHEREARG, __VA_ARGS__)
//...

DEBUGPRINT("hey, x=%d\n", x);
```

在报错时，上述示例代码将打印 x 的值到 error 流的文件和行号中，而且 `WHERESTR` 参数是和字符串连在一起的。`__FILE__` 和 `__LINE__` 的值可以被 `#line` 指令操作，它指定了行号和文件名，例如：

```
#line 314 "pi.c"
puts("line=" #__LINE__ " file=" __FILE__);
```

在这里，这段示例代码生成了 `puts` 函数：

```
puts("line=314 file=pi.c");
```

Source code debuggers refer also to the source position defined with `__FILE__` and `__LINE__`. This allows source code debugging, when C is used as target language of a compiler, for a totally different language. The first C Standard specified that the macro `__STDC__` be defined to 1 if the implementation conforms to the ISO Standard and 0 otherwise, and the macro `__STDC_VERSION__` defined as a numeric literal specifying the version of the Standard supported by the implementation. Standard C++ compilers support the `__cplusplus` macro. Compilers running in non-standard mode must not set these macros, or must define others to signal the differences.

Other Standard macros include `__DATE__`, the current date, and `__TIME__`, the current time.

C99 added support for `__func__`, which contains the name of the function definition within which it is contained, but because the preprocessor is agnostic to the grammar of C, this must be done in the compiler itself using a variable local to the function.

Macros that can take a varying number of arguments (variadic macros) are not allowed in C89, but were introduced by a number of compilers and standardised in C99. Variadic macros are particularly useful when writing wrappers to functions taking a variable number of parameters, such as printf, for example when logging warnings and errors.

One little-known usage pattern of the C preprocessor is known as X-Macros. An X-Macro is a header file. Commonly these use the extension ".def" instead of the traditional ".h". This file contains a list of similar macro calls, which can be referred to as "component macros". The include file is then referenced repeatedly.

Many compilers define additional, non-standard, macros, although these are often poorly documented. A common reference for these macros is the Pre-defined C/C++ Compiler Macros project, which lists "various pre-defined compiler macros that can be used to identify standards, compilers, operating systems, hardware architectures, and even basic run-time libraries at compile-time".

操作符可以把两个标记拼接成一个，例如：

```
#define DECLARE_STRUCT_TYPE(name) typedef struct name##_s name##_t
// Outputs typedef struct g_object_s g_object_t;
DECLARE_STRUCT_TYPE(g_object);
```

5.3.1 Simple Macro

C 语言预编译器支持简单宏和带参数的宏。下面的简单宏中定义符号 *name* 等价于 *definition*，它们没有参数，不使用分号结尾。

```
#define name definition
```

在源文件剩下的部分中，预处理器寻找任何作为一个完整的记号出现的 *name*，并用 *definition* 中的字符替换 *name*。

简单的宏主要用来定义被 Kernighan 和 Ritchie 称为“明显常量（manifest constant）”的记号，这样宏可以替换的 C 语言记号就包括标识符、关键字、数、字符常量、字符串字面量、运算符和标点符号等。

由 C 预处理器执行的替换是文本替换，而且只会替换完整的记号而不会替换记号的片断，因此预处理器会忽略 C 语言记号中的宏名。

宏定义可由任意一组字符组成，程序员可以在常量中包含任意算术运算符，例如：

```
#define MaxStringSize 100
#define BufferSize (MaxStringSize+1)
```

这样，以后出现名字 BufferSize 的地方都用字符串

```
(100+1)
```

代替。定义两边的括号是至关重要的，这取决于 BufferSize 出现的位置，如果程序包含表达式

```
2 * BufferSize
```

那么，预处理器把它替换成

```
2 * (100 + 1)
```

但如果省略了括号，那么将先执行乘法运算，于是就会得出错误结果 201 而不是 202。

宏的定义中可以包含对另一个宏的调用，预处理器会一直处理到宏定义中不再包含对其他宏的调用为止。按照 C 语言标准，如果在扩展宏的过程中原先的宏名重复出现的话，宏名不会再次被替换。

#define 机制包含传递参数的功能，这种功能可以使被定义的符号有函数那样的行为，称为预处理器函数（或者宏指令或伪函数）。预处理器的这种功能常用于定义 ANSI 库中的某些函数，包括 stdio.h 中的 getc 和 ctype.h 中的谓词函数（包括 isalpha 和 isdigit 等），它比标准 C 函数的效率高很多，但也有很多局限性。

使用 #define 来为常量命名可以有许多优点。

- 程序更易于阅读
  在程序中通过宏定义常量可以避免包含大量的“魔法数”，因此对于源代码中的数字常量，只要不是 0 和 1，就应该定义成宏。
  使用宏来替换字符常量和字符串常量并不总是能提高程序的可读性，建议根据常量使用次数和是否会被修改等因素来决定。
- 程序更易于修改
  相比宏，“硬编码”的常量更难于修改，使用宏可以改变整个程序中出现的所有该常量的值。
- 可以帮助避免前后不一致或输入错误
  除了应用于定义常量外，宏还可以有其他应用。
- 可以对 C 语法做小的修改
  可以通过定义宏的方式给 C 语言符号添加别名，从而改变 C 语言的语法。例如，可以通过宏定义使用 Pascal 的 begin 和 end（而不是 C 语言的 {和}）。

```
#define BEGIN {
#define END }
```

使用宏还可以发明新的语言。例如，可以创建一个 LOOP“语句”来实现一个无限循环。

```
#define LOOP for(;;)
```

- 对类型重命名
  除了使用 typedef 来定义新类型外，还可以使用宏来实现此目的。

```
#define BOOL int
typedef int BOOL;
```

- 可控制条件编译
  宏定义中的替换列表可以为空，这样可以使用宏在程序中指明条件编译。例如，可以通过宏定义来表明需要将程序在“调试模式”下进行编译，并使用额外的语句输出调试信息。

```
#define DEBUG
```

5.3.2 Macro with Parameters

带参数的宏定义有如下格式：

```
#define identifier(x1, x2, ..., xn) replacement-list
```

在宏的名称和左括号之间必须没有空格，如果有空格，预处理器会认为是在定义一个简单的宏。宏的参数可以在替换列表中根据需要出现任意次。

当预处理器遇到一个带参数的宏，会将定义存储起来以便后续使用。在接下来的程序中，如果任何地方出现了 identifier(y1, y2, ..., yn) 格式的宏调用，预处理器会使用替换列表替代，并使用 y1 代替 x1，使用 y2 代替 x2，以此类推。

带参数的宏替换的过程类似于函数调用中实际参数与形式参数的按顺序传值。例如，假定定义了如下的宏：

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
#define IS_EVEN(n) ((n) % 2 == 0)
```

如果在后续的程序中出现了如下语句：

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

那么，在预处理阶段，C 预处理器会将上述行替换为下面的形式。

```
i = ((j+k) > (m-n) ? (j+k) : (m-n));
if(((i) % 2 == 0)) i++;
```

在实际开发中，带参数的宏可以用来定义简单的函数，上述 MAX 类似一个从两个值中返回较大的值的函数，而 IS_EVEN 则可以用于对参数进行判断，如果为偶数则返回 1，否则返回 0。

> 在定义宏时，如果宏的替换列表中有运算符，那么始终要把替换列表放在圆括号中。另外，如果宏有参数，那么参数的每一次出现都要添加圆括号。

下面的例子是一个更复杂的宏 TOUPPER(c)。

```
#define TOUPPER(c) ('a' <= (c) && (c) <= 'z' ? (c)-'a'+'A' : (c))
```

宏 TOUPPER(c) 检测一个字符 c 是否在 ‘a’ 与 ‘z’ 之间。如果在的话，这个宏会用 ‘c’ 减去 ‘a’ 再加上 ‘A’ 来计算出 c 所对应的大写字母。如果 c 不在这个范围，就保留原来的 c。

C 语言在 <ctype.h> 库中提供了大量的类似的宏，其中的 toupper 宏与这里的 TOUPPER 宏作用一致，但是更高效，可移植性也更好。

```
/* These are the same as `toupper' and `tolower' except that they do not
   check the argument for being in the range of a `char'. */
__exctype (_toupper);
__exctype (_tolower);
#endif /* Use SVID or use misc. */

/* This code is needed for the optimized mapping functions. */
#define __tobody(c, f, a, args) \
  (__extension__                                                \
   ({ int __res;                                                \
     if (sizeof (c) > 1)                                        \
      {                                                         \
        if (__builtin_constant_p (c))                           \
         {                                                      \
           int __c = (c);                                       \
           __res = __c < -128 || __c > 255 ? __c : (a)[__c]; \
         }                                                      \
        else                                                    \
          __res = f args;                                       \
      }                                                         \
     else                                                       \
       __res = (a)[(int) (c)];                                  \
     __res; }))

#if !defined __NO_CTYPE
# ifdef __isctype_f
__isctype_f (alnum)
__isctype_f (alpha)
__isctype_f (cntrl)
__isctype_f (digit)
__isctype_f (lower)
__isctype_f (graph)
__isctype_f (print)
__isctype_f (punct)
__isctype_f (space)
__isctype_f (upper)
__isctype_f (xdigit)
# ifdef __USE_ISOC99
__isctype_f (blank)
# endif
# elif defined __isctype
# define isalnum(c) __isctype((c), _ISalnum)
# define isalpha(c) __isctype((c), _ISalpha)
# define iscntrl(c) __isctype((c), _IScntrl)
# define isdigit(c) __isctype((c), _ISdigit)
# define islower(c) __isctype((c), _ISlower)
# define isgraph(c) __isctype((c), _ISgraph)
# define isprint(c) __isctype((c), _ISprint)
# define ispunct(c) __isctype((c), _ISpunct)
# define isspace(c) __isctype((c), _ISspace)
# define isupper(c) __isctype((c), _ISupper)
# define isxdigit(c) __isctype((c), _ISxdigit)
# ifdef __USE_ISOC99
#  define isblank(c) __isctype((c), _ISblank)
# endif
# endif

# ifdef __USE_EXTERN_INLINES
```

```
__extern_inline int
__NTH (tolower (int __c))
{
  return __c >= -128 && __c < 256 ? (*__ctype_tolower_loc ())[__c] : __c;
}

__extern_inline int
__NTH (toupper (int __c))
{
  return __c >= -128 && __c < 256 ? (*__ctype_toupper_loc ())[__c] : __c;
}
# endif

# if __GNUC__ >= 2 && defined __OPTIMIZE__ && !defined __cplusplus
# define tolower(c) __tobody (c, tolower, *__ctype_tolower_loc (), (c))
# define toupper(c) __tobody (c, toupper, *__ctype_toupper_loc (), (c))
# endif /* Optimizing gcc */

# if defined __USE_SVID || defined __USE_MISC || defined __USE_XOPEN
# define isascii(c) __isascii (c)
# define toascii(c) __toascii (c)

# define _tolower(c) ((int) (*__ctype_tolower_loc ())[(int) (c)])
# define _toupper(c) ((int) (*__ctype_toupper_loc ())[(int) (c)])
# endif

#endif /* Not __NO_CTYPE. */
  ... ...
```

带参数的宏可以包含空的参数列表。

```
#define getchar() getc(stdin)
```

空的参数列表不是一定确实需要，但可以使 getchar 更像一个函数，这也就是在 <stdio.h> 库中定义的 getchar 宏，在功能上 getchar 类似函数，但其实质是宏。

```
__BEGIN_NAMESPACE_STD
/* Read a character from STREAM.

   These functions are possible cancellation points and therefore not
   marked with __THROW. */
extern int fgetc (FILE *__stream);
extern int getc (FILE *__stream);

/* Read a character from stdin.

   This function is a possible cancellation point and therefore not
   marked with __THROW. */
extern int getchar (void);
__END_NAMESPACE_STD

/* The C standard explicitly says this is a macro, so we always do the
   optimization for it. */
#define getc(_fp) _IO_getc (_fp)

#if defined __USE_POSIX || defined __USE_MISC
/* These are defined in POSIX.1:1996.

   These functions are possible cancellation points and therefore not
   marked with __THROW. */
```

```
extern int getc_unlocked (FILE *__stream);
extern int getchar_unlocked (void);
#endif /* Use POSIX or MISC. */
```

使用带参数的宏替代实际的函数有两个优点。

- 提高程序执行速度。
  函数调用在执行时通常会有些额外开销——存储上下文信息、复制/传递参数的值等，而宏调用则没有这些运行开销。
- 宏的通用性更强。
  与函数的参数不同，宏的参数没有类型，因此预处理后的程序依然是合法的。
  宏可以接受任何类型的参数，例如使用 MAX 宏从两个数中选择较大的数时，数的类型可以是 int、long int、float 和 double 等。

带参数的宏不仅适用于模拟函数调用，还可以用作模板（boilerplate）来处理某些要重复输入的代码段。例如，为了输出整数 x，可以定义一个带参数的宏 PRINT_INT 来简化输出语句。

```
#define PRINT_INT(x) printf("%d\n", x)
```

这样，在实际程序开发中，下面的两条输出语句示例的意义是相同的。

```
PRINT_INT(i/j);
printf("%d\n", i/j);
```

在调试程序时会发现，宏（特别是带参数的宏）可能是程序中错误的根源，但是宏定义在预处理阶段就会被展开，这样当编译器检测到由多余符号所导致的错误时，只会将每一处使用这个宏的地方标记为错误，导致无法直接找到错误的根源。

- 使用宏的代码在编译后通常会变大，当嵌套调用宏时，宏替换会相互叠加从而使程序更为复杂。
- 宏参数没有类型检查，也不会进行类型转换。
  当调用函数时，编译器会检查每个参数来保证它们类型正确，或者将参数转换为正确的类型，或者由编译器来产生相应的错误信息。
- 无法用一个指针来指向一个宏。
  C 语言允许指针指向函数，但宏在预处理阶段就被删除，所以不存在类似的“指向宏的指针”。
- 宏可能会多次进行参数计算，而函数对它的参数只计算一次。如果参数有副作用，那么宏多次计算参数的值可能会产生意外的结果。
  考虑下面的例子，其中 MAX 的一个参数有副作用。

  ```
  n = MAX(i++, j);
  ```

  下面是预处理之后的结果。

  ```
  n = ((i++) > (j) ? (i++) : (j));
  ```

  如果 i 大于 j，那么 i 可能会被（错误地）计算了两次，同时 n 可能被赋予了错误的值。

宏调用和函数调用看起来是一样的，因此多次计算宏的参数而导致的错误可能非常难于发现。更糟糕的是，这类宏可能在大多数情况下正常工作，仅在特定参数有副作用时失效。为了自保护，最好避免使用带副作用的参数。

宏不允许被定义两次，除非新的宏的定义与旧的宏的定义是一样的。

可以编写名字与保留字或标准库中的函数名匹配的宏来实现某些功能。以库函数 sqrt 为例，如果参数为负数则返回一个由实现定义的值，而实际应用中可能希望返回 0。在不修改标准库函数的情况下，可以定义一个宏来使 sqrt 函数在参数为负数时返回 0。

```
#define sqrt(x) ((x)>0 ? sqrt(x) : 0)
```

预处理器会拦截 sqrt 的调用，并将它替换成宏扩展定义的条件表达式。在扫描宏的过程中，条件表达式内部的 sqrt 调用不会被替换，因此会被保留并由编译器处理。

5.4 Include File

C 预处理器通过 #include 指令来打开一个特定的文件，并将其内容作为要编译的源文件的一部分“包含”进来。

```
#include <stdio.h>

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

在上面的示例程序中，#include 指令知识 C 预处理器打开头文件 stdio.h，并用其内容替换 `#include <stdio.h>` 这一行。在 stdio.h 中定义了 C 语言标准输入/输出函数的原型，这里是 printf 函数。

C 语言中的 #include 规范以下面两种形式出现：

```
#include< 文件名>
#define " 文件名"
```

这两种形式在操作中的本质是相同的。在任意一种情况下，预处理器都会寻找具有这个名称的文件，并将 #include 行替换为那个文件的完整内容。这两种形式的唯一区别是预处理器在不同的位置寻找该文件。

- 如果文件名在尖括号中，预处理器在特别为系统文件（如标准 C 函数库）所指定的路径中进行搜索；
- 如果文件名出现在双引号中，预处理器会先在当前用户的文件系统中寻找该文件；如果未在用户域中找到该文件，预处理器才会继续检查系统路径。

被包含的文件就像原始的源文件一样要经过预处理，它也可能包含许多常量定义或其他 #include 行。对于被包含的文件的类型，预处理器没有限制，可以是接口，也可以是其他文件。

如果 C 源程序使用的是用户自定义的头文件，需要使用双引号来代替 <>，否则 C 预处理器将会去编译器指定的路径中去搜索头文件。使用双引号引入头文件时，C 预处理器只在当前源代码的目录中搜索头文件。

在 C 程序开发过程中，可以在 makefile 中指定头文件的路径，这样就可以在不同的操作系统平台上编译程序。

C 预处理器不仅可以引入其他的文件，比如通过 `#include "icon.xbm"` 来引入 XBM 图片文件，也可以通过类似 `#include guards` 或 `#pragma once` 来防止重复的文件引用。

5.5 Conditional Compiling

条件编译是根据预处理器所执行的测试结果来包含或排除程序的片断，C 语言预处理器可以识别大量用于支持条件编译的指令。

在接口定义的模板中就引入了条件编译指令 `#if`、`#ifdef`、`#ifndef`、`#else`、`#elif` 和 `#endif` 等，这样就可以指定源文件的某个部分应该仅在某种条件下被编译。

5.5.1 #if & #endif

在调试程序时，可能需要程序输出特定变量的值，使用 #if 和 #endif 指令可以保留调试函数，而且不影响编译过程。

```
#if VERBOSE >= 2
  printf("trace message");
#endif
```

同理，可以通过定义宏并指定其开关值来开启或关闭调试信息输出。

```
#define DEBUG 1

#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

在接下来的预处理阶段，#if 指令会测试 DEBUG 的值。如果 DEBUG 值非 0，那么预处理器将会把上述两个 printf 语句保留在程序中(#if 和 #endif 行会消失)。

如果将 DEBUG 的值改为 0 并重新编译程序，预处理器会将上述 4 行条件编译指令都删除，这样编译将无法看到这些 printf 语句，所以这些函数调用就不会在目标代码中出现，也就无法输出调试信息。实际程序开发中，可以将 #if 和 #endif 保留在最终的程序中，这样如果程序在运行时出错，可以通过开启 DEBUG 模式来继续产生诊断信息。

一般来说，#if 条件编译指令的格式如下：

```
#if const-expression
```

当预处理器读取到 #if 指令时，首先会计算常量表达式(const-expression)。

- 如果常量表达式的值为 0，那么 #if、#endif 及其中间的行将会在预处理阶段被删除。
- 如果常量表达式的值不为 0，那么在 #if 与 #endif 之间的行会被保留并继续被编译器处理。

#if 与 #endif 不会对最终程序产生影响，#endif 用于结束条件编译指令。

对于没有定义过的标识符，#if 指令会把它当作值为 0 的宏对待，因此如果省略 DEBUG 的定义，上述测试

```
#if DEBUG
```

会失败(但不会报错)，而测试

```
#if !DEBUG
```

会成功。

当然，条件编译指令也可以用于编写更高级的程序，这样就可以使程序能更方便的从一种计算机系统移植到另一种计算机系统。例如，大多数编译器都针对 Windows 操作系统隐式的定义了 _WIN32[2]，这允许在源代码编译时通过包含预处理命令来针对 Windows 操作系统进行编译。

defined 运算符仅用于预处理器，通常与 #if 指令结合使用。例如，下面的示例代码用于判断 DEBUG 宏是否定义过。

```
#if defined (DEBUG)
...
#endif
```

[2]只有少数的编译器使用 WIN32 来代替 _WIN32。如果不在源代码中指定，可以在编译时通过在命令行中添加-D_WIN32 来达到同样的目的。

当 defined 运算符应用于标识符时，如果该标识符是一个定义过的宏就返回 1，否则返回 0。只有当 DEBUG 宏事先被定义，#if 和 #endif 之间的代码才会被保留在程序中。另外，DEBUG 两侧的括号不是必要的，因此上述示例代码可以简写为：

```
#if defined DEBUG
...
#endif
```

defined 运算符仅能检测标识符 DEBUG 是否被定义为宏，这样就不需要给 DEBUG 赋值。

```
#define DEBUG
```

#ifdef 指令和 #ifndef 指令从 20 世纪 70 年代引入到 C 语言中，而 defined 运算符则是在 80 年代的 C 语言标准化过程中引入的。

相比 #ifdef 指令和 #ifndef 指令，defined 运算符增加了灵活性，可以使用 #if 指令和 defined 运算符来测试任意数量的宏，而不再是只能使用 #ifdef 指令和 #ifndef 指令对一个宏进行测试。例如，下面的指令检查是否 FOO 和 BAR 被定义了而 BAZ 没有被定义。

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

5.5.2 #ifdef & #ifndef

#ifdef 指令测试一个标识符是否已经定义为宏，而 #ifdef 指令的使用与 #if 类似。

```
#ifdef DEBUG
...
#endif
```

#ifndef 指令与 #ifdef 指令类似，只是测试的是标识符是否没有被定义为宏。

```
#ifndef DEBUG
```

等价于

```
#if !defined (DEBUG)
```

5.5.3 #elif & #else

#if 指令、#ifdef 指令和 #ifndef 指令可以和普通的条件判断语句一样嵌套使用。

当发生嵌套时，最好随着嵌套层次的增加而增加缩进，而且预处理器提供了 #elif 指令和 #else 指令等得到更多的便利，这样就可以与 #if 指令、#ifdef 指令、#ifndef 指令等替代使用来测试一系列条件。

```
#if expression1
...
#elif expression2
...
#else
...
#endif
```

上述示例等价于以 #if 和 #else[3] 来进一步嵌套的条件编译指令。

```
#if expression1
  ...
  #if expression2
  ...
  #else
  ...
#endif
```

[3] 在 #if 指令和 #endif 之间可以有多个 #elif 指令，但最多只能有一个 #else 指令。

条件编译对于调试是非常方便的，但它并不局限于此。

- 编写在不同操作系统和硬件平台之间可移植的程序。
  在下面的示例中会根据 WINDOWS、DOS 或 OS2 是否被定义为宏来将三组代码之一包含到程序中。

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#endif
```

这样，在程序的开头条件编译指令可以确定一个针对特定操作系统的宏。下面这段测试代码会判断宏 `__unix__` 是否被事先定义，如果定义了宏 `__unix__`，那么就引入 `unistd.h`，否则就使用宏 `_WIN32`，此时还要引入 `windows.h`。

```
/* __unix__ is usually defined by compilers targeting Unix systems */
#ifdef __unix__
# include <unistd.h>

/* _Win32 is usually defined by compilers targeting 32 or 64 bit Windows systems */
#elif defined _WIN32
# include <windows.h>

#endif
```

- 编写可以使用不同的编译器进行编译的程序。
  为了编译不同的 C 语言程序版本需要不同的编译器，这些编译器版本之间有一些差异。一些编译器版本会包含针对特定机器的扩展，而其他的则或不包含，或提供不同的扩展集。
  使用条件编译指令可以使程序适应于不同的编译器。在下面的示例代码中，__STDC__ 宏允许预处理器检测编译器是否支持标准 C。如果不支持标准 C，就可能需要使用经典 C 的函数声明来替代标准 C 的函数原型，因此对于每一个函数的声明，都需要使用下面的代码来测试。

```
#if __STDC__
Standard C function prototype
#else
K&R C function declaration
#endif
```

下面是一个更复杂的条件编译示例用于演示如何使用操作符，例如：

```
#if !(defined __LP64__ || defined __LLP64__) || defined _WIN32 && !defined _WIN64
      // we are compiling for a 32-bit system
#else
      // we are compiling for a 64-bit system
#endif
```

- 为宏提供默认定义。
  条件编译指令可以检测一个宏当前是否已被定义，如果没有被定义，则可以提供一个默认的定义。

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- 临时屏蔽包含注释的代码
  在标准 C 中不允许嵌套注释，无法直接“注释”掉包含注释的代码，此时可以使用 #if 指令解决这个问题。

```
#if 0
  comments
#endif
```

上述将代码以条件编译指令屏蔽的做法称为“条件屏蔽”。

C 语言标准要求预处理器检查 #if 指令和 #endif 指令之间所有的行，这些行必须由预处理记号组成，预处理记号类似于 C 语言记号(标识符、运算符和数等)。

除此之外，还可以使用条件编译实现保护头文件案以避免重复包含。

5.6 Advanced Directives

5.6.1 #error

使用 `#error` 指令可以从错误流中输出信息，例如：

```
#error "error message"
```

其中，“error message” 是一个 C 语言标记序列。如果预处理器遇到一个 #error 指令，它会输出一个出错消息。

对于不同的编译器，出错消息的具体形式也可能会不一样，但是 #error 指令指明的是严重的程序错误，大多数编译器会立即终止编译而不是查找其他错误。

实际开发中，#error 指令通常与条件编译指令一起用于检测正常编译过程中不应出现的情况。例如，为了确保程序在 int 类型范围在 100 000 以上的机器上才能编译，就需要在检测到 INT_MAX 宏不足 100 000 时调用 #error 指令。

```
#if INT_MAX < 100000
#error int type is too small.
#endif
```

#error 指令通常会出现在 #if ～ #elif ～ #else 序列中的 #else 部分。

```
#if defined(WINDOWS)
...
#elif defined(DOS)
...
#elif defined(OS2)
...
#else
#error No operating system specified.
#endif
```

5.6.2 #line

#line 指令可以用来改变给程序行编号的额方式，也可以用于使编译器认为它正在从一个有不同名字的文件中读取一个程序。

#line 指令有两种形式，在第一种形式中指定一个行号：

```
#line n
```

n 必须是介于 1 到 32 767 之间的整数，#line 指令可以使程序后面的行被编号为 n、n+1、n+2 等。

在 #line 的第二种形式中，需要同时指定行号和文件名。

```
#line n "file-name"
```

这样将使编译器认为后面的行来自文件，行号由 n 开始。

#line 指令的一种作用是改变 __LINE__（可能还有 __FILE__ 宏）。更重要的是，大多数编译器会使用来自 #line 指令的信息产生出错信息。

#line 指令主要用于以生成 C 代码作为输出的程序（例如 yacc）。

yacc（Yet Another Compiler Compiler）是 Unix/Linux 上一个采用 LALR(1) 语法分析方法来生成编译器的编译器（编译器代码生成器[4]）。IEEE POSIX P1003.2 标准定义了 Lex 和 Yacc 的功能和需求，其输入是巴科斯范式（BNF）表达的语法规则以及语法规约的处理代码，Yacc 输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。

yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），需要与词法解析器 Lex 一起使用，再把两部份产生出来的 C 程序一并编译。由于所产生的解析器需要词法分析器配合，因此 Yacc 经常和词法分析器的产生器 Lex 联合使用。

在使用 yacc 之前，需要准备一个包含 yacc 所需的信息以及 C 代码段的文件，这样 yacc 可以通过该文件生成一个 C 程序 y.tab.c，并合并了所提供的代码。接下来，按照正常方法编译 y.tab.c 时可以通过在 y.tab.c 中插入的 #line 指令使编译器认为代码来自原始文件，这样编译 y.tab.c 时产生的任何信息都会指向原始文件中的行，而不是 y.tab.c 中的行。

#line 指令可以使出错信息都指向原始文件而不是由 yacc 生成的文件，从而使得调试变得更容易。

5.6.3 #pragma

C99 introduced a few standard `#pragma` directives, taking the form `#pragma STDC ...`, which are used to control the floating-point implementation.

The `#pragma` directive is a compiler-specific directive, which compiler vendors may use for their own purposes. For instance, a `#pragma` is often used to allow suppression of specific error messages, manage heap and stack debugging and so on. A compiler with support for the OpenMP parallelization library can automatically parallelize a `for` loop with `#pragma omp parallel for`.

- Many implementations do not support trigraphs or do not replace them by default.
- Many implementations (including, e.g., the C compilers by GNU, Intel, Microsoft and IBM) provide a non-standard directive to print out a warning message in the output, but not stop the compilation process. A typical use is to warn about the usage of some old code, which is now deprecated and only included for compatibility reasons, e.g.:

```
(GNU, Intel and IBM)
#warning "Do not use ABC, which is deprecated. Use XYZ instead."
(Microsoft)
#pragma message("Do not use ABC, which is deprecated. Use XYZ instead.")
```

- Some Unix preprocessors traditionally provided "assertions", which have little similarity to assertions used in programming.
- GCC provides `#include_next` for chaining headers of the same name.
- Objective-C preprocessors have `#import`, which is like `#include` but only includes the file once.

Since the C preprocessor can be invoked independently to process files other than those containing to-be-compiled source code, it can also be used as a "general purpose preprocessor" (GPP) for other types of text processing. One particularly notable example is the now-deprecated imake system.

[4]一般认为 Berkeley Yacc 是目前最好的 yacc 变种。与 GNU bison 相比，避免了对特定编译器的依赖。

GPP does work acceptably with most assembly languages. GNU mentions assembly as one of the target languages among C, C++ and Objective-C in the documentation of its implementation of the preprocessor. This requires that the assembler syntax not conflict with GPP syntax, which means no lines starting with # and that double quotes, which cpp interprets as string literals and thus ignores, don't have syntactical meaning other than that.

However, since the C preprocessor does not have features of some other preprocessors, such as recursive macros, selective expansion according to quoting, string evaluation in conditionals, and Turing completeness, it is very limited in comparison to a more general macro processor such as m4.

#pragma 指令为要求编译器执行某些特殊操作提供了一种方法，#pragma 指令对非常大的程序或需要使用特定编译器的特殊功能的程序非常有用。

#pragma 指令有如下形式：

```
#pragma token
```

其中，token 是一般 C 语言的记号。

#pragma 指令通常只跟着一个记号，这个记号表示了一条编译器需要服从的命令。

一些编译器允许 #pragma 指令包含带参数的命令。

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

#pragma 指令中出现的命令集在不同的编译器上是不一样的。如果 #pragma 指令包含了无法识别的命令，编译器必须忽略这些 #pragma 指令，不允许产生出错信息。

5.7 Predefined Macros

在 C 语言中预定义了一些有用的宏，这些宏主要是提供当前编译的信息。

Table 5.1: 预定义宏

名称	类型	描述
__LINE__	整型常量	被编译的文件的行数
__FILE__	字符串字面量	被编译的文件的名字
__DATE__	字符串字面量	编译的日期(格式“Mmm dd yyyy”)
__TIME__	字符串字面量	编译的时间(“hh:mm:ss”)
__STDC__	整型常量	如果编译器接受标准 C，则值为 1

__DATE__ 宏和 __TIME__ 宏指明程序编译的时间信息。例如，下面的语句可以用来帮助区分同一个程序的不同版本。

```
printf("TheQiong(c) 2014\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

__LINE__ 宏和 __FILE__ 宏可以用来帮助除错。考虑下面检测被零除的除法错误发生位置的问题，以前当一个 C 程序因为被零除而导致中止时，通常没有信息指明哪条除法运算导致的错误，使用 __LINE__ 宏和 __FILE__ 宏可以帮助找到错误的根源。

```
#define CHECK_ZERO(divisor) \
  if(divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
    "of file %s ***", __LINE__, __FILE__)
```

接下来为了检测除零错误，通过在除法运算前被调用 CHECK_ZERO 宏，就可以用于显示出错的相关位置信息。

```
CHECK_ZERO(j);
```

```
k = i / j;
```

如果 j 是 0，就会显示出如下形式的信息。

```
*** Attempt to divide by zero on line 9 of file F00.c ***
```

在实际开发中，类似 CHECK_ZERO 的宏非常有用。
实际上，C 语言提供了用于检测错误的通用宏 <assert.h>。

第 6 章

Interprete

脚本语言(Scripting language)是为了缩短传统的“编写、编译、链接、运行”(edit-compile-link-run)过程而创建的计算机编程语言。

如果所使用的翻译的机制是将所要翻译的程序代码作为一个整体翻译,并之后运行内部格式,那么这个翻译过程就被成为编译。因此,一个编译器是一个将人可阅读的程序文本(叫做源代码)作为输入的数据,然后输出可执行文件(object code)。所输出的可执行文件可以是机器语言,由计算机的中央处理器直接运行,或者是某种模拟器的二进制代码。

如果程序代码是在运行时才即时翻译,那么这种翻译机制就被称作解译。经解译的程序运行速度往往比编译的程序慢,但往往更具灵活性,因为它们能够与执行环境互相作用。

在解释器中,变量的访问也是比较慢的,因为每次要访问变量的时候它都必须找出该变量实际存储的位置,而不像编译过的程序在编译的时候就决定好了变量的位置了。

早期的脚本语言经常被称为批处理语言或工作控制语言,而宏语言则可视为脚本语言的分支,两者也有实质上的相同之处。脚本语言程序通常是解释运行而非编译。

6.1 Interpreter

6.1.1 Overview

解释器(Interpreter)可以把高级编程语言代码一行一行直接转译执行。

解释器不会一次把整个程序转译出来。具体来说,它只像一位“中间人”,每次运行程序时都要先转成另一种语言再作运行,因此解释器的程序运行速度比较缓慢。

解释器每转译一行程序叙述就立刻运行,然后再转译下一行,再运行,如此不停地进行下去。

在许多方面,高级编程语言和脚本语言之间互相交叉,二者之间没有明确的界限。解释器的好处是它消除了编译整个程序的负担,但也会让运行时的效率打了折扣。相对地,编译器并不运行程序或原代码,而是一次将其翻译成另一种语言(如机器码)以供多次运行而无需再经编译。其制成品无需依赖编译器而运行,程序运行速度比较快。

几乎所有计算机系统的各个层次都有脚本语言(包括操作系统层、计算机游戏、网络应用程序、字处理文档和网络软件等),例如 AWK、BASIC、JavaScript、LISP、Shell、Perl、Prolog、Ruby、Python 和 PHP 等。

计算机科学历史上,第一个解释器是由 Steve Russell 基于 IBM 704 的机器代码写成的 LISP 解释器。

有时候,“interpreter”是指一些可以读取打孔卡的机器,这些机器可以读取卡片上的孔并以人们读得懂的格式打印出来。IBM 550 数字读卡机和 IBM 557 字母读卡机是主要的两个例子。

大多脚本语言共性包括良好的快速开发,高效率的执行,解释而非编译执行,和其它语言编写的程序组件之间通信功能很强大。

许多脚本语言用来执行一次性任务,尤其是系统管理方面。它可以把服务组件粘合起来,因此被广泛用于 GUI 创建或者命令行。操作系统通常提供一些默认的脚本语言,即通常所谓 shell 脚本语言。在大型项目中经常把脚本和其它低级编程语言一起使用,各自发挥优势解决特定问题。

脚本语言解释器运行程序的方法有:

1. 直接运行高级编程语言(如 Shell 内置的解释器)
2. 转换高级编程语言码到一些有效率的字节码(Bytecode),并运行这些字节码

3. 以解释器包含的编译器对高级语言编译,并指示处理器运行编译后的程序(例如 JIT)

Perl、Python、MATLAB 与 Ruby 等是属于第二种方法,而 UCSD Pascal 则是属于第三种方式。在转译的过程中,这组高级语言所写成的程序仍然维持在源代码的格式(或某种中继语言的格式),而程序本身所指涉的动作或行为则由解释器来表现。

使用解释器来运行代码会比直接运行编译过的机器码来得慢,是因为解释器每次都必须去分析并转译它所运行到的程序行,而编译过的程序就只是直接运行。但是,相对的这个直译的行为会比编译再运行来得快。这在程序开发的雏型化阶段和只是撰写试验性的代码时尤其来得重要,因为这个"编辑-直译-除错"的循环通常比"编辑-编译-运行-除错"的循环来得省时许多。

在使用解释器来达到较快的开发速度和使用编译器来达到较快的运行进度之间是有许多妥协的。有些系统(例如有一些 LISP)允许直译和编译的代码互相调用并共享变量。这意味着一旦一个子程序在解释器中被测试并除错过之后,它就可以被编译以获得较快的运行进度。许多解释器并不像其名称所说的那样运行原始代码,反而是把原始代码转换成更压缩的内部格式。举例来说,有些 BASIC 的解释器会把 keywords 取代成可以用来在 jump table 中找出相对应指令的单一 byte 符号。解释器也可以使用如同编译器一般的文字分析器(lexical analyzer)和语法分析器(parser)然后再转译产生出来的抽象语法树(abstract syntax tree)。

直译式程序相较于编译式程序有较佳的可携性,可以容易的在不同软硬件平台上运行。而编译式程序经过编译后的程序则只限定于运行在开发环境平台。

6.1.2 Bytecode

考量程序运行之前所需要分析的时间,存在了一个介于直译与编译之间的可能性。例如,用 Emacs Lisp 所撰写的源代码会被编译成一种高度压缩且优化的另一种 Lisp 源代码格式,这就是一种字节码(bytecode),而它并不是机器码(因此不会被绑死在特定的硬件上)。这个"编译过的"码之后会被字节码直译器(使用 C 写成的)转译。在这种情况下,这个"编译过的"码可以被说成是虚拟机(不是真的硬件,而是一种字节码解释器)的机器码。这个方式被用在 Open Firmware 系统所使用的 Forth 代码中:原始程序将会被编译成"F code"(一种字节码),然后被一个特定平台的虚拟机直译和运行。

6.1.3 JIT

即时编译(JIT)是指一种在运行时期把字节码编译成原生机器码的技术;这项技术是被用来改善虚拟机的性能的。大约在 1980 年代 Smalltalk 语言出现的时候 JIT 的技术就存在了,后来 JIT 技术的发展已经模糊了直译、字节码直译及编译的差异性,现在在.NET 和 Java 的平台上都有用到 JIT 的技术。

虽然大多数的语言可以既可被编译又可被解译,但大多数仅在一种情况下能够良好运行。在一些编程系统中,程序要经过几个阶段的编译。

一般而言,后阶段的编译往往更接近机器语言。这种常用的使用技巧最早在 1960 年代末用于 BCPL,编译程序先编译一个叫做"0 代码"的转换程序(representation),然后再使用虚拟器转换到可以运行于机器上的真实代码。这种成功的技巧之后又用于 Pascal、P-code 以及 Smalltalk 和二进制码。在很多时候,中间过渡的代码往往是解译,而不是编译的。

根据使用环境和运行过程,脚本语言可以分类如下:

- 工作控制语言和 shell

  此类脚本语言用于自动化工作控制—即启动和控制系统程序的行为。大多的脚本语言解释器也同时是命令行界面,如 Unix shell 和 MS-DOS COMMAND.COM,其他如 AppleScript 可以为系统增加脚本环境,但没有命令行界面。

- GUI 脚本

  GUI 出现带来一种专业的控制计算机的脚本语言。它在用户和图形界面,菜单,按钮等之间互动。它经常用来自动化重复性动作,或设置一个标准状态。理论上它可以用来控制运行于基于

GUI 的计算机上的所有应用程序，但实际上这些语言是否被支持还要看应用程序和操作系统本身。当通过键盘进行互动时，这些语言也被称为宏语言。

- 应用程序定制的脚本语言
  许多大型的应用程序都包括根据用户需求而定制的惯用脚本语言。同样地，许多电脑游戏系统使用一种自定义脚本语言来表现 NPC 和游戏环境的预编程动作。此类语言通常是为一个单独的应用程序所设计，虽然它们貌似一些通用语言，但它们有自定义的功能。
- Web 编程脚本语言
  Web 编程脚本语言提供 Web 页面的自定义功能，可以专业处理互联网通信，使用浏览器作为用户界面等。大多数现代 Web 编程语言都可以做一些通用编程。
- 文本处理语言
  处理基于文本的记录是脚本语言最早的用处之一，例如 awk 和 sed 最早是设计来帮助系统管理员处理调用 UNIX 基于文本的配置和 LOG 文件，Perl 最早是用来产生报告的。
- 通用动态语言
  通用动态语言包括 Groovy、Lua、Perl、PHP、Python、Ruby、Scheme 和 Tcl 等。
- 扩展/可嵌入语言
  少数的语言被设计通过嵌入应用程序来取代应用程序定制的脚本语言。开发者（如使用 C 等其它系统语言）放入使脚本语言可以控制应用程序的 hook，这样这些语言和应用程序定制的脚本语言是同种用途，但优点在于可以在应用程序之间传递一些功能。
  其中，JavaScript 直到现在仍然是网页浏览器内的主要编程语言，它的 ECMAScript 标准化保证了它成为流行的通用嵌入性语言。Tcl 作为一种扩展性语言而创建，但更多地被用作通用性语言，就如同 Python、Perl 和 Ruby 一样。

第 7 章

Compiling

7.1 Overview

在大部分计算机系统上，运行程序之前要先输出程序文本并将其保存在一个文件（file）中，文件是存储在计算机辅助存储设备里的信息集合的统称。每个文件都必须有一个文件名，通常用句点（.）将文件名分成两部分，如 myproc.c[1]。

包含程序文本的文件通常被称作源文件（source file），一个 C 语言程序一般由一个或多个源文件组成。输入文件或改变文件内容的过程称为编辑（editing）文件，而且各种计算机系统之间的文件编辑过程差异很大，因此不可能用一种适合于各种类型的硬件的方式来描述编辑过程。

虽然程序设计语言的形式多种多样，复杂度也不一，但都是由两个部分构成，即语法（syntax）和语义（semantics），其中语法说明了如何组织语言中的指令，规定有效指令的结构的正式规则，语义则是赋予程序设计语言中的指令含义的一套规则，说明了各条指令的含义。

一旦有了源文件，下一步就是使用编译器将源文件翻译成计算机可直接读懂的形式。这个过程也会因机器而异，但是在大多数情况下，编译器将源文件翻译成中间文件，这种中间文件称为目标文件（object file），其中包括适用于特定计算机系统的实际指令。

C 编译器把每一个源文件翻译成相应的目标文件，目标文件包含着一个特定的计算机系统上执行程序所需要的指令。将不同目标文件以及其他用于实现各种常用操作的称为库的目标文件链接起来就可以创建一个可执行文件（executable file）。

库（library）是预定义的目标文件，库中含有程序所要求的不同操作的机器指令，将所有独立的目标文件组合成一个可执行文件的过程称为链接（linking），可由下图表示：

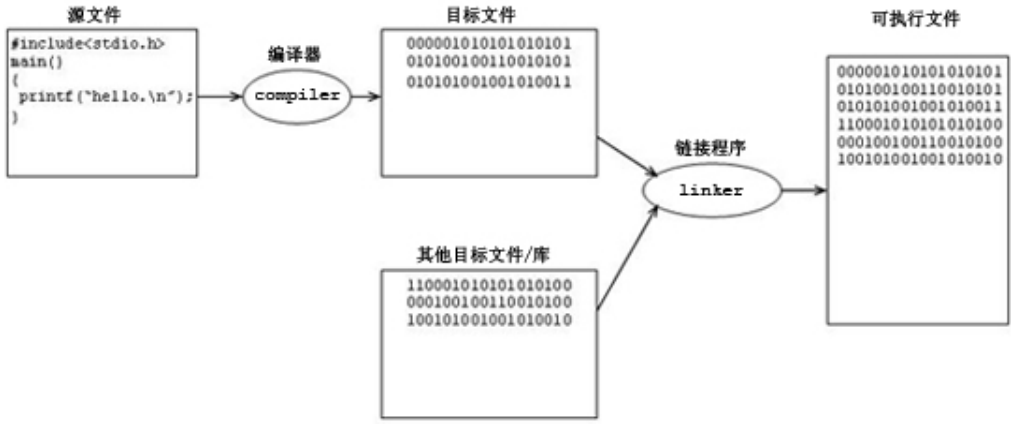


图 7.1: 编译过程示例

[1]建立文件时需要选择一个根名（root name），这是文件名中句点之前的那部分名称，它说明文件的内容，句点后的那部分名作为扩展名（extension），说明文件的类型。一些扩展名有预先指定的意义，例如扩展名.c 就表示文件中的程序文本是用 C 语言编写的。

7.2 Stage

编译一个源文件的过程包括下列四个阶段:

1. 预处理
   首先,预处理器(或编译器)开始介入并对源文件进行预处理——文本替换,具体来说就是用一组已有的操作变换整个源文件的文本,这些操作包括读入库接口和以符号常量的定义替代符号常量名。
   比如第一行的 `#include <stdio.h>` 直接替换成原来 stdio.h 的内容,然后保存。
2. 词法分析
   在词法分析阶段,编译器将经过预处理的源文件中的字符以形成一个个有意义的单元,这种单元叫做记号(token)。
3. 语法分析
   在语法分析阶段,编译器检验源文件中的记号是否符合 C 语言的语法(句法)规则。此外,语法分析阶段还负责解释程序的意义,即语义。
4. 生成代码
   编译器对经过语法分析的程序表示进行编译,生成汇编语言的指令。再由汇编器汇编上一步生成的汇编语言指令来生成二进制文件。不过这只是半成品,因为它还没有链接各种文件中的函数等,此时该文件为.o 文件。
   最后,链接器将彼此有关的目标文件进行关联,比如该程序调用了 printf 函数,则需要在链接之前单独编译了的提供 printf 函数定义的.o 文件。

在经过了种种这些复杂的处理后,这里最后才生成机器可以直接执行的二进制文件,当然实际情况往往比这还要复杂得多[?]。

在大多数现代计算机编程语言中,程序可以被分成如子程序的更小的组件,这些组件可以通过许多物理源文件分发,这些源文件被单独编译。当一个子程序在定义的位置以外的地方被使用时,就需要引入前置声明和函数原型的概念。例如,一个函数在一个源文件中有如下定义:

```
int add(int a, int b)
{
  return a + b;
}
```

在另一个源文件中引用的时候就可以声明成这样(包含函数原型):

```
int add(int, int);

int triple(int x)
{
  return add(x, add(x, x));
}
```

但是,这个简单的方法需要程序员为 add 在两个地方维护函数声明,一个是包含函数实现的的文件,以及使用该函数的文件。如果函数的定义改变了,程序员必须要更改散布在程序中的所有的原型。

头文件提供了解决办法。模块的头文件声明作为模块公共接口一部分的每一个函数、对象以及数据类型。例如,在下面的情况下,头文件仅包含 add 的声明。每一个引用了 add 的源文件使用 #include 来包含头文件:

```
/* File add.h */
#ifndef ADD_H
#define ADD_H

int add(int, int);

#endif /* ADD_H */
/* File triple.c */
#include "add.h"

int triple(int x)
{
  return add(x, add(x, x));
}
```

这样就减少了维护的负担：当定义改变的时候，只须更新声明的一个独立副本（在头文件中的那个）。在包含对应的定义的源文件中也可以包含头文件，这给了编译器一个检查声明和定义一致性的机会。

```
/* File add.c */
#include "add.h"

int add(int a, int b)
{
  return a + b;
}
```

通常，头文件被用来唯一指定接口，且多少提供一些文档来说明如何使用在该文件中声明的组件。在这个例子中，子程序的实现放在一个单独的源文件中，这个源文件被单独编译。（在 C 和 C++ 中有个例外，即内联函数。内联函数通常放在头文件中，因为大多数实现如果不知道其定义，在编译时便无法适当的展开内联函数。）

在某些计算机中，编译过程中的每一步都由计算机来完成，不需要人的参与，用户指示出想要运行的程序后，所有必要步骤就会自动执行。在另外一些计算机上，则需要用户单独执行编译和链接步骤。

无论在哪一种情况下，源文件都是用户都可以读懂的唯一文件，其他文件所包含的信息只有计算机才能读出。

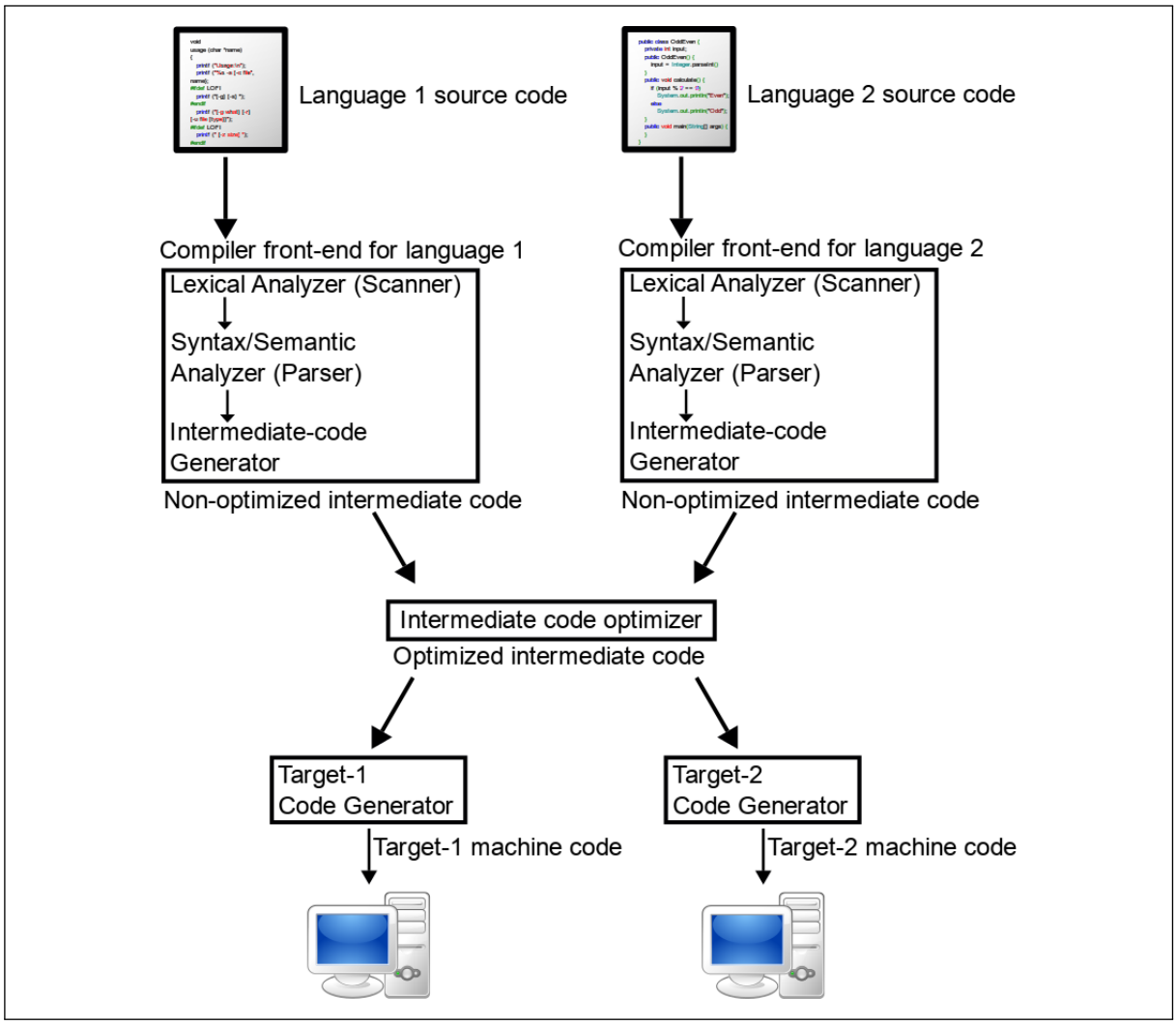
7.3 Classification

为了让用程序设计语言编写的程序能够在不同的计算机系统上运行，首先必须将源代码翻译成运行程序的计算机所特有的低级机器语言的程序。

在高级语言和机器语言之间执行这种翻译任务的程序叫做编译器（Compiler），编译器将原始程序（Source program）作为输入，翻译产生使用目标语言（Target language）的等价程序。

编译器可以将高级计算机语言所编写的源代码，翻译为计算机能解读、运行的低阶机器语言的程序——也就是可执行文件。源代码一般为高阶语言（High-level language），而目标语言则是汇编语言或目标机器的目标代码（Object code），有时也称作机器代码（Machine code）。

一个现代编译器的主要工作流程如下：



早期的计算机软件都是用汇编语言直接编写的，后来当人们发现为不同类型的 CPU 编写可重用软件的开销要明显高于编写编译器时，于是发明了高级编程语言。

由于早期的计算机的内存很少，当大家实现编译器时，遇到了许多技术难题。大约在 20 世纪 50 年代末期，与机器无关的编程语言被首次提出。随后，人们开发了几种实验性质的编译器。

- 第一个编译器是由葛丽丝·霍普于 1952 年为 A-0 系统编写的。
- 在 1957 年由 IBM 的约翰·巴科斯领导的 FORTRAN 则是第一个被实现出具备完整功能的编译器。
- 在 1960 年发布的 COBOL 成为一种较早的能在多种架构下被编译的语言。

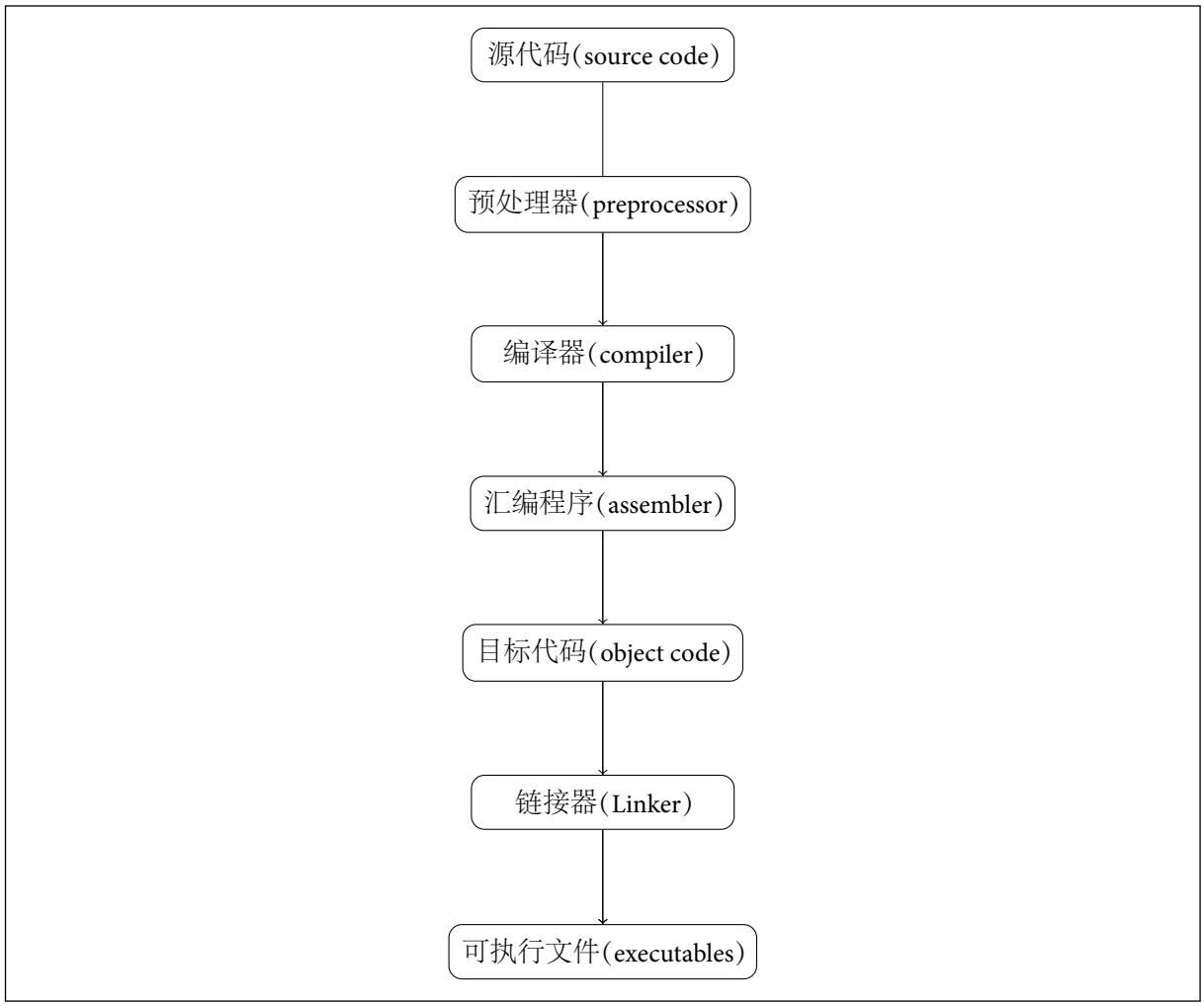
于是，高级语言在许多领域流行起来。随着新的编程语言支持的功能越来越多和计算机的架构越来越复杂，这使得编译器也越来越复杂。

早期的编译器是用汇编语言编写的。首个能编译自己源程序的编译器是在 1962 年由麻省理工学院的 Hart 和 Levin 制作的。从 20 世纪 70 年代起，实现能编译自己源程序的编译器变得越来越可行，不过还是用 Pascal 和 C 语言来实现编译器更加流行。制作某种语言的第一个编译器[2]，要么需要用其它语言来编写，要么就像 Hart 和 Levin 制作 Lisp 编译器那样，用解释器来运行编译器。

编译器的一种分类方式是按照生成代码所运行的系统平台划分，这个平台称为目标平台。

- 本地编译器输出的代码可以运行于与编译器所在相同类型的计算机和操作系统上。
- 交叉编译器的输出可以运行于不同的平台上。例如，嵌入式系统通常没有软件开发环境，因此为这类系统开发软件时，通常需要使用交叉编译器。

[2]编译器的构造与优化是计算机专业的大学课程，课程名称一般为编译原理。通常在课程中包含了如何实现一种教学用程序语言的编译器。一个著名的例子是 20 世纪 70 年代，尼克劳斯·维尔特用于讲解编译器的构造时使用的 PL/0 编译器。



对于编译器所输出到虚拟机上运行的代码，由于编译器及其输出的运行平台有可能相同也有可能不同，因此可以不区分这类编译器是本地编译器还是交叉编译器。

在 Linux 系统中，某个文件能不能被执行看它是否具有可执行的权限，不过真正运行的可执行文件其实是二进制文件。例如，`/usr/bin/passwd`，`/bin/touch` 等都是二进制文件，可以使用 `file` 命令来查看文件类型。

```
file /bin/touch
/bin/touch: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=a26ebafcffd4890bc4f6b7d09749f233744aa0f4, stripped
```

如果是二进制文件而且是可执行的，则会显示可执行文件类（ELF 64-bit LSB executable，或 ELF 32-bit LSB executable），同时还会说明是否使用共享库（uses shared libs），而如果是一般的文本文件，则无此返回结果。

Shell script 也可以在 Linux 中运行，不过它只是利用 shell（例如 bash）的功能，最终执行的除了 bash 自身提供的功能外，还是要调用系统中的其他二进制程序，而且 bash 本身也是二进制程序。

7.3.1 GCC

GCC(GNU Compiler Collection)是一套由 GNU 开发的编程语言编译器,以 GPL 及 LGPL 许可证发布。作为 GNU 计划的关键部分,亦是自由的类 Unix 及苹果计算机 Mac OS X 操作系统的标准编译器,GCC 常被认为是跨平台编译器的事实标准。

GCC 原名为 GNU C 语言编译器(GNU C Compiler),因为它原本只能处理 C 语言。GCC 很快扩展成可处理 C++、Fortran、Pascal、Objective-C、Java、Ada 以及 Go 与其他语言。

几乎全部的 GCC 都由 C 写成,除了 Ada 前端大部分以 Ada 写成。后来因为 LLVM、Clang 的崛起,令 GCC 更快将开发语言转换为 C++[3]。

GCC 通常是跨平台软件的编译器首选[4]。有别于一般局限于特定系统与运行环境的编译器,GCC 在所有平台上都使用同一个前端处理程序,产生一样的中介码,而且中介码在各个其他平台上使用 GCC 编译时,有很大的机会可得到正确无误的输出程序。

OpenMP 是一种跨语言的对称多处理机多线程并行程序的编程工具,也非常适合当今越来越流行的单 CPU 多核硬件环境。

从 gcc4.2 开始,OpenMP 成为其内嵌支持的并行编程规范,可以直接编译内嵌 OpenMP 语句的 C/C++/Fortran95 的源代码。gcc4.2 之前如果想在 C/C++/Fortran 中嵌入 OpenMP 语句的话,需要额外安装库和预处理器才能识别和正确处理这些语句。

- gcc 4.2.0 开始支持 OpenMP v2.5
- gcc 4.4.0 开始支持 OpenMP v2.5 及 v3.0

GCC 的外部接口类似于标准的 Unix 编译器,用户在命令行下键入 gcc 以及一些命令参数,以便决定每个输入文件使用的个别语言编译器,并为输出代码使用适合此硬件平台的汇编语言编译器,并且选择性地运行连接器以制造可运行的程序。

每个语言编译器都是独立程序,它们可以处理输入的源代码并输出汇编语言码。

GCC 中全部的语言编译器都拥有共通的中介架构:一个前端解析符合此语言的源代码,并产生一抽象语法树,以及一翻译此语法树成为 GCC 的暂存器转换语言的后端。编译器优化与静态代码解析技术(例如 FORTIFY_SOURCE,一个试图发现缓存溢出的编译器)在此阶段应用于代码上。最后,适用于此硬件架构的汇编语言代码以 Jack Davidson 与 Chris Fraser 发明的算法产出。

- 前端接口
  前端的功能在于产生一个可让后端处理之语法树,其语法解析器是递归语法解析器。
  直到最近,程序的语法树结构尚无法与欲产出的处理器架构脱钩,而语法树的规则有时在不同的语言前端也不一样,有些前端会提供它们特别的语法树规则。
  在 2005 年,两种与语言脱钩的新型态语法树纳入 GCC 中,称为 GENERIC 与 GIMPLE。语法解析变成产生与语言相关的暂时语法树,再将它们转成 GENERIC。之后再使用“gimplifier”技术降低 GENERIC 的复杂结构,成为一较简单的静态唯一形式(Static Single Assignment form,SSA)基础的 GIMPLE 形式。此形式是一个与语言和处理器架构脱钩的全局优化通用语言,适用于大多数的现代编程语言。
- 中介接口
  一般编译器作者会将语法树的优化放在前端,但其实此步骤并不看语言的种类而有不同,且不需要用到语法解析器,因此 GCC 作者们将此步骤归入通称为中介阶段的部分里。此类的优化包括消解死码、消解重复运算与全局数值重编码等。
- 后端接口
  GCC 后端的行为因不同的前处理器宏和特定架构的功能而不同,例如不同的字符尺寸、调用方式与大小尾序等。后端接口的前半部利用这些信息决定其 RTL 的生成形式,因此虽然 GCC 的 RTL 理论上不受处理器影响,但在此阶段其抽象指令已被转换成目标架构的格式。

[3]许多 C 的爱好者在对 C++ 一知半解的情况下主观认定 C++ 的性能一定会输给 C,但是 Taylor 给出了不同的意见,并表明 C ++ 不但性能不输给 C,而且能设计出更好,更容易维护的程序。

[4]当 GCC 需要移植到一个新平台上,通常使用此平台固有的语言来撰写其初始阶段。

GCC 的优化技巧依其发布版本而有很大不同，但都包含了标准的优化算法，例如循环优化、线程跳跃、共通程序子句消减、指令调度等等。而 RTL 的优化由于可用的情形较少，且缺乏较高级的信息，因此比较起近来增加的 GIMPLE 语法树形式，便显得比较不重要。

后端经由一重读取步骤后，利用描述目标处理器的指令集时所取得的信息，将抽象暂存器替换成处理器的真实暂存器。此阶段非常复杂，因为它必须关照所有 GCC 可移植平台的处理器指令集的规格与技术细节。

后端的最后步骤相当公式化，仅仅将前一阶段得到的汇编语言码借由简单的副函数转换其暂存器与存储器位置成相对应的机器码。

GDB 是 GNU 软件系统中的标准侦错器，其他特殊用途的除错工具还有 Valgrind—用以发现存储器泄漏(Memory leak)。

GNU 测量器(gprof)可以得知程序中某些函数花费多少时间以及其调用频率，此功能需要用户在编译时选定测量(profiling)选项。

GCC 内嵌汇编也称行内汇编，是把汇编语言代码块插在 C 语言语句之间。

GCC 目前包含了贝姆垃圾收集器(Boehm GC)，可应用于许多经由 C/C++ 开发的项目，同时也适用于其它执行环境的各类编程语言，包括 GNU 版 Java 编译器执行环境以及 Mono 的 Microsoft .NET 移植平台等。

Boehm GC 支持许多的操作系统平台，比如各种 Unix 操作系统，微软的操作系统以及麦金塔上的操作系统(Mac OS X)，还有更进一步的功能，例如渐进式收集(incremental collection)，平行收集(parallel collection)以及终结语意的变化(variety of finalizer semantics)。

垃圾收集器作用于未变性的(unmodified)C 程序时，只要简单的将 `malloc` 调用用 `GC_malloc` 取代，将 `realloc` 取代为 `GC_realloc` 调用，如此一来便不需要使用到 `free` 的函数。下列的代码示例演示了如何用 Boehm 取代传统的 `malloc` 以及 `free`。

```
#include "gc.h"
#include <assert.h>
#include <stdio.h>

int main()
{
  int i;

  GC_INIT();
  for(i = 0; i < 10000000; I)
  {
    int **p = (int **) GC_MALLOC(sizeof (int *));
    int *q = (int *) GC_MALLOC_ATOMIC(sizeof (int));

    assert(*p == 0);
    *p = (int *) GC_REALLOC(q, 2 * sizeof (int));
    if(i % 100000 == 0)
      printf("Heap size = %d\n", GC_get_heap_size());
  }

  return 0;
}
```

7.3.2 G++

GCC 和 G++ 都可以编译 C/C++ 代码，但需要说明的是[?]：

- 对于后缀为.c 的，GCC 把它当作是 C 程序，而 G++ 则把它当作是 C++ 程序；
- 对于后缀为.cpp 的，两者都会将其认为是 C++ 程序。

C++ 是 C 的超集，只是两者对语法的要求是有区别的，例如：

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  if(argv == 0) return;
  printString(argv);
  return;
}
int printString(char* string) {
  sprintf(string, "This is a test.\n");
}
```

如果按照 C 的语法规则，则可以正常编译。但如果把后缀改为.cpp，则会报告三个错误：

1. “printString 未定义”；
2. “cannot convert ‘char**’ to ‘char*”；
3. ”return-statement with no value“；

可见 C++ 的语法规则更加严谨，使用 C 语言开发时应该做到兼容 C++ 语言。

G++ 在编译阶段会调用 GCC，对于 C++ 代码，这两者是等价的。另外，由于 GCC 命令不能自动和 C++ 程序使用的库链接，所以通常用 G++ 来完成链接，但在编译阶段，G++ 会自动调用 GCC，二者等价。也可以说，编译可以用 GCC/G++，而链接可以用 `g++` 或者 `gcc -lstdc++`。

实际上，`__cplusplus` 宏只是标志着编译器是把代码按 C 还是 C++ 语法来解释，此时如果后缀为.c，并且采用 GCC 编译器，则该宏就是未定义的，否则，就是已定义。

无论是 GCC 还是 G++，用 `extern "c"` 时，都是以 C 的命名方式来为 symbol 命名，否则，都以 C++ 方式命名。试验如下：

```
me.h:
extern "C" void CppPrintf(void);

me.cpp:
#include <iostream>
#include "me.h"
using namespace std;
void CppPrintf(void)
{
  cout << "Hello\n";
}

test.cpp:
#include <stdlib.h>
#include <stdio.h>
#include "me.h"
int main(void)
{
  CppPrintf();
  return 0;
}
```

下面先给 me.h 加上 `extern "C"` 并进行测试：

```
[root@root G++]# g++ -S me.cpp
[root@root G++]# less me.s
.globl _Z9CppPrintfv //注意此函数的命名
      .type CppPrintf, @function
[root@root GCC]# gcc -S me.cpp
[root@root GCC]# less me.s
```

```
.globl _Z9CppPrintfv //注意此函数的命名
      .type CppPrintf, @function
```

结果是相同的。

去掉 me.h 中 `extern "C"`，再次进行测试：

```
[root@root GCC]# gcc -S me.cpp
[root@root GCC]# less me.s
.globl _Z9CppPrintfv //注意此函数的命名
      .type _Z9CppPrintfv, @function
[root@root G++]# g++ -S me.cpp
[root@root G++]# less me.s
.globl _Z9CppPrintfv //注意此函数的命名
      .type _Z9CppPrintfv, @function
```

结果仍然是相同的，可见 `extern "C"` 与采用 GCC/G++ 并无关系，以上的试验还间接的印证了前面的说法：在编译阶段，G++ 是调用 GCC 的。

7.3.3 TCC

Tiny C Compiler（缩写为 TCC）是 Fabrice Bellard 开发的用于低级计算机环境的 C 语言编译器。

TCC 是由 Obfuscated Tiny C Compiler (OTCC) 即小型混淆器用途发展而来。2001 年，Fabrice Bellard 为参加 International Obfuscated C Code Contest (IOCCC) 比赛而开发 TCC，并在其后将其扩展为小型混淆器程序，后来成为最终的 TCC。目前在 GNU 宽通用公共许可证 (LGPL) 协议规范下发布。

TCC 符合 ANSI C (C89/C90) 规范，亦符合新版的 ISO C99 标准规范，并支持 GNU C 扩展的内嵌汇编语言功能。

使用 TCC assembler 的语法要兼容于 GNU assembler，而且使用时的限制条件如下：

- 必须支持 C 或 C++ 的指令；
- 指针符号与 C 相同，因此无法使用 “.” 或 “$” 符号；
- 主要支持 32 位系统；
- 只能使用内联汇编代码。

TCC 包含一个可选的 memory(存储器) 和 boundschecker(程序检测器)，经过检测的代码可以随意地混合于标准代码内。另外，为方便编译，要将 TCC 源代码内的 libtcc.h 于置于 include 内。

7.3.4 Clang

Clang 是一个 C、C++、Objective-C 和 Objective-C++ 编程语言的编译器前端，采用底层虚拟机（LLVM）作为其后端，其目标是提供一个 GNU 编译器套装（GCC）的替代品。作者是克里斯·拉特纳，在苹果公司的赞助支持下进行开发，源代码授权是使用类 BSD 的伊利诺伊大学厄巴纳-香槟分校开源码许可。

Clang 项目包括 Clang 前端和 Clang 静态分析器等，早期 Clang 即支持 C、C++、Objective C。

Clang 项目在 2005 年由苹果计算机发起，是 LLVM 编译器工具集的前端（front-end），目的是输出代码对应的抽象语法树（Abstract Syntax Tree, AST），并将代码编译成 LLVM Bitcode，接着在后端（back-end）使用 LLVM 编译成机器语言。

在 Clang 语言中，使用 Stmt 来代表 statement。CLag 代码的单元（unit）皆为语句（statement），语法树的节点（node）类型就是 Stmt。另外 CLang 的表达式（Expression）也是语句的一种，Clang 使用 Expr 来代表 Expression，Expr 本身继承自 Stmt。节点之下有子节点列表（sub-node-list）。

Clang 本身性能优异，其生成的 AST 所耗用掉的内存仅仅是 GCC 的 20% 左右，测试证明 Clang 编译 Objective-C 代码时速度为 GCC 的 3 倍，还能针对用户发生的编译错误准确地给出建议。

7.3.5 LLVM

LLVM[2] 的命名最早源自于底层虚拟机(Low Level Virtual Machine)的缩写,它是一个编译器的基础建设,以 C++ 写成。

LLVM 是为了任意一种编程语言写成的程序,利用虚拟技术来创造出编译时期、链结时期、运行时期以及“闲置时期”的优化。它最早是以 C/C++ 为实现对象,目前它支持了包括 ActionScript、Ada、D 语言、Fortran、GLSL、Haskell、Java bytecode、Objective-C、Python、Ruby、Rust、Scala 以及 C#。

LLVM 项目起源于 2000 年伊利诺伊大学厄巴纳-香槟分校维克拉姆·艾夫(Vikram Adve)与克里斯·拉特纳(Chris Lattner)的研究发展而成,他们想要为所有静态及动态语言创造出动态的编译技术。LLVM 是以 BSD 授权来发展的开源码软件。在 2005 年,苹果计算机雇用了克里斯·拉特纳及他的团队,为苹果计算机开发应用程序系统,LLVM 为现今 Mac OS X 及 iOS 开发工具的一部分。

LLVM 最初被用来取代现有于 GCC stack 的代码产生器,许多 GCC 的前端已经可以与其运行,LLVM 目前支持 Ada、C 语言、C++、D 语言、Fortran 及 Objective-C 的编译,它使用许多的编译器,有些来自 4.0.1 及 4.2 的 GCC。

LLVM 吸引一些开发者来为许多语言开发新的编译器,其中一个最引发注意的就是 Clang,它是一个新的编译器,同时支持 C、Objective-C 以及 C++。Clang 主要得到苹果计算机的支持,Clang 的目的用以取代 GCC 系统的 C/Objective-C 编译器。

LLVM 提供了完整编译系统的中间层,它会将中间语言(IF, Intermediate form)从编译器取出与优化,优化后的 IF 接着被转换及链结到目标平台的汇编语言。LLVM 可以接受来自 GCC 工具链所编译的 IF,包含它底下现存的编译器。

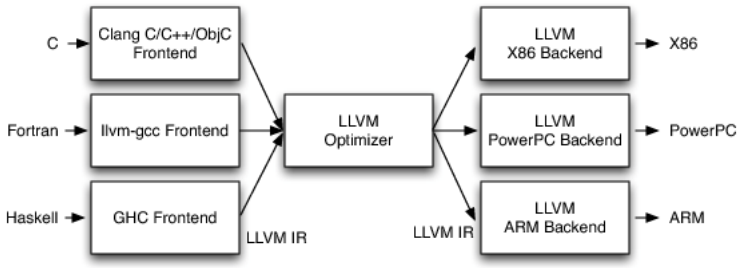


图 7.2: LLVM's Implementation of the Three-Phase Design

LLVM 也可以在编译时期、链结时期,甚至是运行时期产生可重新定位的代码(Relocatable Code)。

LLVM 支持与语言无关的指令集架构及类型系统。每个在静态单赋值形式(SSA)的指令集代表着,每个变量(被称为具有类型的暂存器)仅被赋值一次,这简化了变量间相依性的分析。LLVM 允许代码被静态的编译,包含在传统的 GCC 系统底下,或是类似 JAVA 等后期编译才将 IF 编译成机器码所使用的实时编译(JIT)技术。它的类型系统包含基本类型(整数或是浮点数)及五个复合类型(指针、数组、矢量、结构及函数),在 LLVM 具体语言的类型建制可以以结合基本类型来表示,举例来说,C++ 所使用的 class 可以被表示为结构、函数及函数指针的数组所组成。

LLVM JIT 编译器可以优化在运行时期时程序所不需要的静态分支,这在一些部分求值(Partial Evaluation)的案例是相当有效。当程序有许多选项,在特定环境之下多数是可被判断为不需要的。这个特色被使用在 Mac OS X Leopard(v10.5)底下 OpenGL 的管线,当硬件不支持某个功能时依然可以被成功地运作。OpenGL stack 底下的绘图程序被编译为 IF,接着在机器上运行时被编译,当系统拥有高级 GPU 时,这段程序会进行极少的修改并将传递指令给 GPU,当系统拥有低级的 GPU 时,LLVM 将会编译更多的程序,使这段 GPU 无法运行的指令,在本地端的中央处理器运行。LLVM 增进了使用 Intel GMA 芯片等低端机器的性能。一个类似的系统发展于 Gallium3D LLVMpipe,被合并进 GNOME,使其可运行在没有 GPU 的环境。

在现代操作系统中，Clang 较为容易与集成开发环境（IDE）集成，而且对于线程有更好的支持。GCC 下的 Objective-C 的开发已经停滞，而苹果计算机已经将其支持移至其他的维护分支。

LLVM 已经逐渐成长为众多编译工具及低级工具技术的统称，现今 LLVM 已经单纯成为一个品牌，适用于 LLVM 底下的所有项目，包含 LLVM 中介码（LLVM IR）、LLVM 除错工具、LLVM C++ 标准库等。

另外，Utrecht Haskell 编译器可以产生 LLVM 使用的代码，但它还在初期的开发阶段，不过在许多案例中显示出比起 C 代码产生器拥有更好的效率。Glasgow Haskell Compiler（GHC）拥有一个可以运作的 LLVM 后端，程序运行性能对比起原先的编译器可以达到 30% 的加速，它仅比一个由 GHC 所实现，并拥有多项优化技术的编译器还慢。

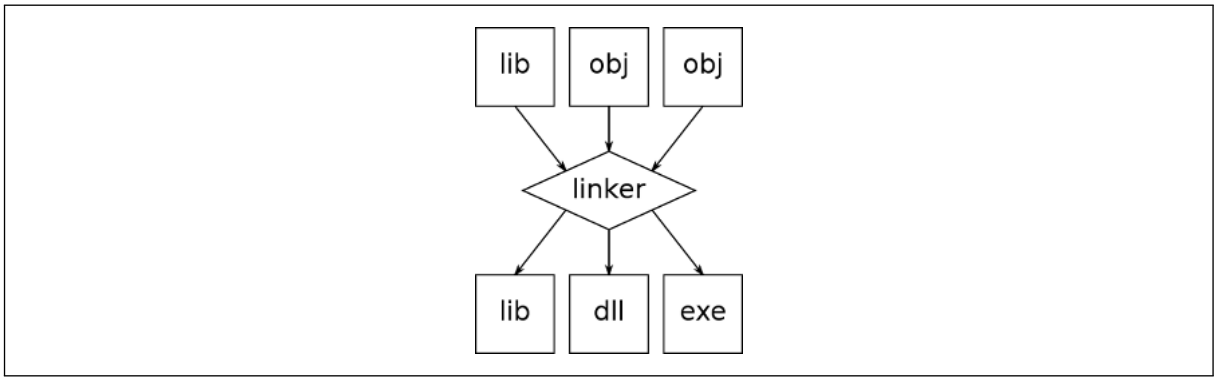
还有其他的组件在不同的开发阶段，包含（但不限于）Java bytecode、通用中间语言（CIL）、MacRuby（实现 Ruby 1.9）、Standard ML 及新的 graph coloring 暂存器配置等。

第 8 章

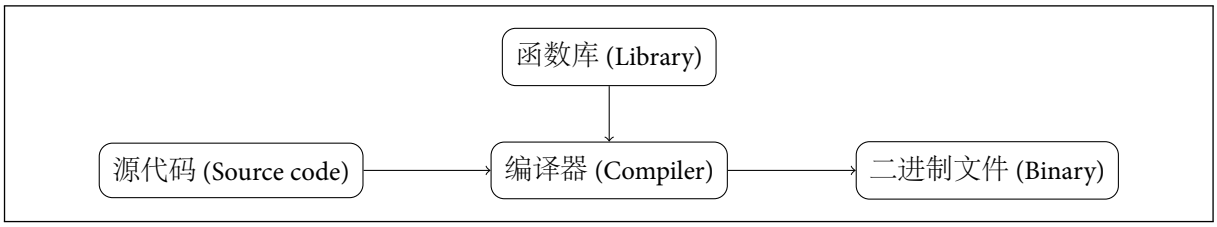
Linker

传统的编译器把源代码翻译成机器指令(即目标代码)后,但这还不是可以运行的。

链接器(Linker)可以将一个或多个由编译器或汇编器生成的目标文件和任何其他附加代码整合到一起,链接成为一个可执行文件。



这些附加代码中就包括库函数,这样在程序中引用、调用其他的外部子程序,或者是利用其他软件提供的“函数功能”,就可以在编译的过程中将该函数链接进来,这样编译器会将所有的程序代码与函数库做一个链接以生成正确的执行文件。大多数现代操作系统都提供静态链接和动态链接两种形式。



IBM 大型主机比如 OS/360 上的链接器是 linkage editor,在 Unix-like 系统上常用的链接器是 GNU ld,现在已经是 GNU Binary Utilities(binutils) 的一部分。

由 Google 的 Ian Lance Taylor 领导的团队开发的 gold 是 ELF 文件的链接器,后来也被加入 binutils,gold 最初是为了针对大型 C++ 程序的链接而开发的。

现在 binutils 是一整套的编程语言工具程序,可以用于创建和管理二进制程序、目标文件、库、配置文件和汇编代码等,并搭配 GCC、make 和 GDB 等程序进行开发。目前 binutils 包含下面这些程序:

Table 8.1: GNU Binutils

as	汇编器,用于把汇编语言代码汇编为指令码
ld	连接器,把目标代码文件链接为可执行文件
gprof	性能分析工具,用于显示程序简档信息
addr2line	把目标文件的虚拟地址转换为文件名、行号或符号

ar	对静态库进行创建、修改和取出的操作。
c++filt	过滤器，用于解码 C++ 的符号
dlltool	创建 Windows 动态链接库
gold	另一种连接器
nlmconv	把目标代码文件转换成 NetWare Loadable Module 文件格式
nm	显示目标文件内的符号
objcopy	复制和翻译目标文件，过程中可以修改
objdump	显示目标文件的相关信息，亦可反汇编
ranlib	产生存档文件或静态库内容的索引
readelf	按照 ELF 格式显示来自目标 ELF 文件的内容
size	列出目标文件或存档文件总体和段(section)的大小
strings	列出任何目标文件内的可打印字符串
strip	从目标文件中移除符号
windmc	产生 Windows 信息资源
windres	编译器，用于编译 Microsoft Windows 资源文件

在编译过程中，使用 GNU ld 可以将 object 文件链接成可执行文件(或库)，链接器脚本可以传递给 GNU ld 来检查链接过程。

ld.so 是 Unix 或类 Unix 系统上的动态链接器，常见的有两个变体：

- ld.so 针对 a.out 格式的二进制可执行文件
- ld-linux.so 针对 ELF 格式的二进制可执行文件

在 Linux 中对程序源代码进行编译的过程中，会生成所谓的目标文件(Object file)，目标文件是包括机器码和链接器可用信息的程序模块。目标文件以“*.o”的扩展名形式存在。

```
file test.o
ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

简单的讲，链接器的工作就是解析未定义的符号引用，将目标文件中的占位符替换为符号的地址。链接器还要完成程序中各目标文件的地址空间的组织，这可能涉及重定位工作。

当应用程序需要使用动态链接库里的函数时，由 ld.so 负责加载。搜索动态链接库的顺序依此是：

- 环境变量 LD_AOUT_LIBRARY_PATH(a.out 格式)、LD_LIBRARY_PATH(ELF 格式)
  在 Linux 中，LD_PRELOAD 指定的目录具有最高优先权。
- 缓存文件/etc/ld.so.cache。
  此为上述环境变量指定目录的二进制索引文件，更新缓存的命令是 ldconfig。
- 默认目录，先在/lib 中寻找，再到/usr/lib 中寻找。

由于可以通过修改上述环境变量，让具有特权的应用程序加载恶意动态链接库，从而导致攻击行为，因此对于 setuid/setgid 应用程序，动态链接器只在默认目录中寻找已被 setgid 的动态链接库。

根据编译器和操作系统的不同，编译和链接所需的命令也不尽相同，例如使用 gcc 进行编译时，系统自动进行链接操作，从而无需使用单独的链接命令。在编译和链接好程序后，编译器会把可执行程序放到文件 a.out 中。

gcc 编译器有很多选项，例如使用-o选项则允许为含有可执行程序的文件自定义名字。下面的示例会把 hello.c 文件生成的可执行文件命名为 hello。

```
gcc -o hello hello.c
```

另外，在使用 gcc 进行编译时，建议在编译时使用-Wall选项来更彻底地检查程序并警告可能发生的问题。

```
gcc –Wall –o hello hello.c
```

除了在命令行中对源代码进行编译外，也可以在“集成开发环境”中对源代码进行编辑、编译、链接、执行和调试等全部操作。

第 9 章

Testing

软件测试[1](software testing),描述一种用来促进鉴定软件的正确性、完整性、安全性和质量的过程。据此,您可能会想,软件测试永远不可能完整的确立任意计算机软件的正确性。然而,在可计算理论(计算机科学的一个支派)一个简单的数学证明推断出下列结果:不可能完全解决所谓"死机",指任意计算机程序是否会进入死循环,或者罢工并产生输出问题。换句话说,软件测试是一种实际输出与预期输出间的审核或者比较过程。

软件测试的经典定义是:在规定的条件下对程序进行操作,以发现程序错误,衡量软件质量,并对其是否能满足设计要求进行评估的过程。

软件测试有许多方法,但对复杂的产品运行有效测试不仅仅是研究过程,更是创造并严格遵守某些呆板步骤的大事。测试的其中一个定义:为了评估而质疑产品的过程;这里的"质疑"是测试员试着对产品做的事,而产品以测试者脚本行为反应作为回答。虽然大部分测试的智力过程不外乎回顾、检查,然而"测试"这个辞意味着产品动态分析——让产品流畅运行。程序质量可能,而且通常会随系统不同而有差异;不过某些公认特性是共通的:可靠性、稳定性、轻便性、易于维护、以及实用性。

代码覆盖率原本是种白箱测试活动。目标软件通过特殊选项或者函数馆编译并且/或者在特殊环境(程序里每个函数都被映射回源代码里函数起点)下运行。这个过程允许开发员与品管员查看系统中在正常情况下极少或从未被读写的部分(例如:异常处理之类)并且帮助测试员确认最重要的情况(函数点)都被测过了。

测试员可查看代码覆盖率测试结果来设计测试个案、相对应的输入或者设置组以增加重要函数的代码覆盖率。两种测试员常用的代码覆盖率形式:陈述式覆盖率(或称行覆盖率)以及路径覆盖率(或称边覆盖率)。行覆盖率回报到测试完成时,运行过哪些行,或者存储器大小。边覆盖率回报到测试完成时,哪些分支,或者程序决定点被运行过。正如覆盖率的"率"字所言,这两个都以百分比为单位。

通常代码覆盖率的工具与函数馆要求的性能、存储器、或者其他资源开销不为正常的软件营运接受。因此它们通常只存在实验室里。又,你可能会想到软件里的许多类无法一一通过这些代码覆盖率测试,虽然代码覆盖程度可通过分析但不是直接测试。

有些瑕疵也会受这些工具的影响。个别来说某些竞态条件(race condition)或者类似的对实时(real time)敏感度高的操作几乎不可能在代码覆盖率测试环境下侦知;相反的这类的瑕疵只会带来更多的测试码开销。

9.1 Test Processes

- Alpha
  Alpha 测试通常是阶段性的开发完成后所开始进行,一直持续到进入 Beta 测试阶段前的阶段。Alpha 测试是一种验证测试,在模拟的环境中以模拟的数据来运行。
  在这个阶段中,通常是在开发单位由开发人员与测试的测试人员,以模拟或实际操作性的方式进行验证测试。
- Beta

[1]软件测试一度被认为是编程能力偏低的员工的工作,直到今天,仍然有许多公司把优秀的人才放在编码上,也有更多公司让优秀的人才进行设计,可是很少公司让优秀的人才进行测试工作。实际的软件工程实践证明,让对软件思想有深刻理解的工程师进行软件测试,可以大幅度的提高软件质量。

在系统测试中通常先进行 Alpha 测试以验证信息系统符合用户以及设计需求所期望的功能。当 Alpha 阶段完成后，开发过程进入到 Beta 阶段，由公众参与的测试的阶段。Beta 测试可称为确认测试，在一个真实的环境中以实际的数据来运行测试，以确认性能，系统运行有效率，系统撤消与备份作业正常，通过测试让信息系统日后可以更趋完善。

- Closed Beta
  封闭测试（Closed Beta，常简作封测或 CB）是软件或服务等产品在开发完成后、将公开上市前的测试过程。相对于公开测试，封闭测试的主要用途是测试软件的功能和检查程序错误等等，因此通常只提供给少数人进行测试。有些公司会要求参与测试者签署保密协议，以避免测试的产品提前外流。MMORPG 的封测退出之后，游戏公司常会将角色数据删除，但也有少数不删的。
- Open Beta
  公开测试（Open Beta，常简作公测或 OB），一般常指软件或服务等产品在正式上市前开放给不特定人试用，虽然原意是希望试用者能够提报 bug，但并不是把试用者当做真正的验证人员。由于通常为免费性质，故常常能够吸引到大批的试用者参与，可视为另一种营销策略。另一方面也节省下测试人员的成本，和验证稳定度（对于多人使用的带宽及机器是否能负载，又称压力测试）的时间。
- Gamma
  Gamma 测试是一个很少被提及的非正式测试阶段，该测试阶段对应的是对“存在缺陷”产品的测试。考虑到任何产品都可以被称为“存在缺陷”的产品（测试只能发现产品中存在的问题，不能说明产品不存在问题），因此这个概念存在一定的不确定性。对 Alpha 和 Beta 测试常见的一个误解是“Beta 测试＝黑盒测试”。实际上，Alpha 和 Beta 测试对应在软件产品发布之前的 Alpha 和 Beta 阶段，而白盒、黑盒和灰盒测试技术是从技术和方法层面对测试的描述，不应该将这两部分概念混淆。

9.2 Test Methods

软件测试一般分为白箱测试和黑箱测试。

- black-box testing
  黑箱测试（black-box testing），也称黑盒测试，是软件测试方法，测试应用程序的功能，而不是其内部结构或运作。测试者不需具备应用程序的代码、内部结构和编程语言的专门知识。测试者只需知道什么是系统应该做的事，即当键入一个特定的输入，可得到一定的输出。测试案例是依应用系统应该做的功能，照规范、规格或要求等设计。测试者选择有效输入和无效输入来验证是否正确的输出。
  此测试方法可适合大部分的软件测试，例如单元测试（unit testing）、集成测试（integration testing）以及系统测试（system testing）。
- white-box testing
  白箱测试（white-box testing，又称透明盒测试 glass box testing、结构测试 structural testing 等）是一个测试软件的方法，测试应用程序的内部结构或运作，而不是测试应用程序的功能（即黑箱测试）。在白箱测试时，以编程语言的角度来设计测试案例。测试者输入数据验证数据流在程序中的流动路径，并确定适当的输出，类似测试电路中的节点。
  白箱测试可以应用于单元测试（unit testing）、集成测试（integration testing）和系统的软件测试流程，可测试在集成过程中每一单元之间的路径，或者主系统跟子系统中的测试。尽管这种测试的方法可以发现许多的错误或问题，它可能无法检测未使用部分的规范。

9.3 Test Types

- 功能测试
  按照测试软件的各个功能划分进行有条理的测试,在功能测试部分要保证测试项覆盖所有功能和各种功能条件组合。
- 系统测试
  对一个完整的软件以用户的角度来进行测试,系统测试和功能测试的区别是,系统测试利用的所有测试数据和测试的方法都要模拟成和用户的实际使用环境完全一样,测试的软件也是经过系统集成以后的完整软件系统,而不是在功能测试阶段利用的每个功能模块单独编译后生成的可执行程序。
- 极限值测试
  对软件在各种特殊条件,特殊环境下能否正常运行和软件的性能进行测试。
  - 特殊条件一般指的是软件规定的最大值,最小值,以及在超过最大、最小值条件下的测试。
  - 特殊环境一般指的是软件运行的机器处于 CPU 高负荷,或是网络高负荷状态下的测试,根据软件的不同,特殊环境也有所不同。
- 性能测试
  性能测试是对软件性能的评价。简单的说,软件性能衡量的是软件具有的响应及时度能力。因此,性能测试是采用测试手段对软件的响应及时性进行评价的一种方式。根据软件的不同类型,性能测试的侧重点也不同。

压力测试常常和性能测试相混淆。它们主要不同点是,压力测试要求进行超过规定性能指标的测试。例如一个网站设计容量是 100 个人同时点击,压力测试就要是采用 120 个同时点击的条件测试。

压力测试的通常判断准则包括:

- 系统能够恢复。
- 压力过程中不要有明显性能下降。

9.4 Test Periods

- 单元测试
  单元测试是对软件组成单元进行测试,其目的是检验软件基本组成单位的正确性,测试的对象是软件设计的最小单位:模块。
- 集成测试
  集成测试也称综合测试、组装测试、联合测试,将程序模块采用适当的集成策略组装起来,对系统的接口及集成后的功能进行正确性检测的测试工作。其主要目的是检查软件单位之间的接口是否正确,集成测试的对象是已经经过单元测试的模块。
- 系统测试
  系统测试主要包括功能测试、界面测试、可靠性测试、易用性测试、性能测试。功能测试主要针对包括功能可用性、功能实现程度(功能流程 & 业务流程、数据处理 & 业务数据处理)方面测试。
- 回归测试
  回归测试指在软件维护阶段,为了检测代码修改而引入的错误所进行的测试活动。回归测试是软件维护阶段的重要工作,有研究表明,回归测试带来的耗费占软件生命周期的 1/3 总费用以上。

与普通的测试不同,在回归测试过程开始的时候,测试者有一个完整的测试用例集可供使用,因此,如何根据代码的修改情况对已有测试用例集进行有效的复用是回归测试研究的重要方向,此外,回归测试的研究方向还涉及自动化工具,面向对象回归测试,测试用例优先级,回归测试用例补充生成等。

第 10 章

Debugging

程序错误(Bug)是指在软件运行中因为程序本身有错误而造成的功能不正常、死机、数据丢失、非正常中断等现象。

Bug 的管理可以划分为下面几个方面:

- 处理进度
  - New:代表新回报的 Bug
  - Resolved:代表 Bug 已处理完(见下方‘处理方式’)
  - Closed:处理完并已被验证
- 处理方式
  - Fixed:Bug 被解决
  - Later:必须到未来的版本才能解决。
  - Workaround:不能解决,但能用其他替代方法来避开问题的。
  - Duplicate:重复回报的 Bug
- Severity:Bug 造成的严重性
- Debugging(简称 Debug):指解决 Bug 的动作和过程(调试)。

编译器除了翻译之外,还执行另一项功能——调试(Debug)。和人类的自然语言一样,程序设计语言也有自己的词汇和句法规则,这些规则使计算机能够确定哪些语句表达正确而哪些表述有错。确定一个语句表述是否正确的规则称为语法规则(syntax rule)。程序设计语言有自己的语法,它决定如何将一个程序的元素组合在一起。

编译一个程序时,编译器首先检查程序的语法是否正确。若违反了语法规则,编译器将显示出错信息。由于违反语法规则而导致的错误称为语法错误(syntax error)。当从编译器得到一个语法错误的消息时,必须返回程序并改正错误。

然而语法错误还不是最令人沮丧的。最严重的程序设计错误就是程序语法正确,但显示不正确的结果,甚至根本就不显示结果。

通常,程序运行失败往往不是由于编写出的程序包含语法错误,而是由于合乎语法的程序却给出了不正确的答案或者根本没给出答案。检查程序时便会发现程序中有逻辑上的错误,这种错误称为逻辑错误(bug)。

Edsger Dijkstra 认为使用 bug 作为计算机错误的统称会让人们产生错觉,bug 有时会变成一种智力欺骗,隐藏了程序员自己制造错误的事实。

> 找出并改正逻辑错误的过程称为调试,调试是程序设计过程中重要的一环。

Edsger Dijkstra[1] 认为使用 bug 作为计算机错误的统称会让人们产生错觉,bug 有时会变成一种智力欺骗,隐藏了程序员自己制造错误的事实。

找出并改正逻辑错误的过程称为调试(debugging),调试是程序设计过程中重要的一环。

逻辑错误可能会非常难以察觉并且令人沮丧,有时程序员确信程序的算法是正确的,但随后发现

[1]Edsger Dijkstra 除了对程序设计语言的贡献之外,还以证明程序正确性(Program Correctness)的工作而闻名。程序正确性这个领域是把数学应用于计算机程序设计。研究者们致力于构造一种语言和证明方法,用于证明一个程序能无条件地按照它的规约执行,完全没有 bug。毫无疑问,无论应用程序是给客户开账单的系统,还是飞行控制系统,这一主张都极其有用。

它不能正确处理以前忽略了的一些情况;或者也许在程序的某个地方做了一个特殊的假定,但随后却忘记了;又或者犯了一个非常愚蠢的错误。

事实上,所有的程序员都会犯逻辑错误,特别地,我们还会制造一些逻辑错误。从许多方面来讲,发现自己犯了逻辑错误是通往程序员的道路上的一个重要环节,而优秀程序员的优秀之处并不在于他们能够避免逻辑错误,而是他们能努力将存在于完成的代码中的逻辑错误数量减到最少。

在程序设计中,最重要的是从陈述问题过渡到解决问题。要做到这一点,就必须以逻辑方式考虑问题,训练自己用计算机能够理解的方式表达自己的逻辑。

当设计完一种算法并将它翻译成合乎语法的程序后,必须知道工作还未完成,几乎可以肯定,程序中的某个地方肯定会存在逻辑错误。作为一个程序员,你的工作就是不断地找出并改正这种逻辑错误,应当总是怀疑自己编写的程序,并尽可能地彻底地进行测试。

10.1 Debugger

调试是发现和减少计算机程序或电子仪器设备中程序错误的一个过程,其基本步骤包括:

- 发现程序错误的存在
- 以隔离、消除的方式对错误进行定位
- 确定错误产生的原因
- 提出纠正错误的解决办法
- 对程序错误予以改正,重新测试

调试工具(Debugger)亦称侦错工具、除错工具、除错程序、调试器、除错器,可以用于调试其它程序的计算机程序及工具。能够让代码在指令组模拟器(ISS)中来检查运行状况以及选择性地运行,以便排错和除错。当开发的进度遇到瓶颈或找不出哪里有问题时,使用调试技术将是非常有用的。但是将程序运行在除错器下,这将比直接在运作的平台以及处理器上运行要来得慢。

当程序死机时,如果调试器是属于源代码阶段(source-level debugger)或符号阶段(symbolic debugger),除错器就可以显示出错误所在位置的源代码,并使其于集成开发环境里也能看见。

低级调试器(low-level debugger)或机器语言调试器(machine-language debugger)可以显示反汇编码(这里指的死机情况是指,当发生原因是因为程序员在设计上的疏失,使得程序无法继续正常运行的情况。例如程序尝试去调用某个对在该版本的 CPU 上而言是不合理的操作,或者是对保护或无法访问的存储器位置进行写入)。

典型的调试器通常能够在程序运行时拥有以下这些功能:

- 单步运行(single-stepping)
- 利用中断点(breakpoint)使程序遇到相应种类的事件(event)时停止(breaking)(一般用于使程序停止在想要检查的状态)
- 追踪某些变量的变化

大部份现代微处理器的设计中都至少会拥有这些特点,使得除错更加容易。有些调试器也有能力在想要除错的程序在运行状态时,并改变它的状态,而不仅仅只是用来观察而已。

软件要能够运行在不同的除错器下进行测试,是非常重要的。不过由于调试器出现将做对软件程序的内部时间的不可避免的变动,因此在多任务环境或分布式系统下,它也会更难去测试到运行时(runtime)的问题。

大部份的主流调试器都可以提供基于终端的命令提示接口(console-based command line)。调试器前端也可以集成在开发工具中作为调试引擎。

现在,GDB(GNU Debugger)已经是 GNU 操作系统上的标准调试器,而且它的使用并不局限于 GNU 操作系统。GDB 可以运行在许多 UNIX-like 操作系统上并支持 C、C++、Pascal 和 FORTRAN 等编程语言。从 GDB 7.0 开始,GDB 开始支持 Python 和逆向调试,即允许调试器后退,从而可以返回崩溃的程序返回前一步。

GDB 最初是在 1988 年由理查德·马修·斯托曼(Richard Stallman)所开发,它具有各种调试功能,能针对计算机程序的运行进行追踪与警告。开发人员使用 GDB 可以监视及修改程序的内部变量值,也可以监视与修改独立于主程序运作外以独立个体型态调用的函数。

GDB 能为多种不同处理器架构上运行的软件进行调试,其内部已具备了依据各种不同处理器的指令集所编译成的模拟推演程序(Simulator)。

GDB 的“远程”模式可以为嵌入式系统进行调试。这里,远程操作指的是 GDB 在一部机器内运行,而要进行排错的程序是在另一部机器上运行,接着要排错的机器上会再加装一个名为“Stub”的小程序,该程序能够与另一端的 GDB 程序沟通。沟通的路径可以是两部机器间的串口式接线(Serial Cable)或者是支持 TCP/IP 协议的网络。在 TCP/IP 网络及协议上再加载传输 GDB 专有的通信协议,如此便能进行远程调试。

调试是软件开发过程中一个必不可少的环节,在 Linux 内核开发的过程中也不可避免地会面对如何调试内核的问题。但是,Linux 系统的开发者出于保证内核代码正确性的考虑,不愿意在 Linux 内核源代码树中加入一个调试器。从 2.6.25 开始,Linux 主干内核开始内置了代码级调试器 kgdb。kgdb 提供了一种使用 gdb 调试 Linux 内核的机制,这样开发者就可以在内核代码中进行设置断点、检查变量值和单步跟踪程序运行等操作。

使用 KGDB 调试时需要两台机器,一台作为开发机(Development Machine),另一台作为目标机(Target Machine),两台机器之间通过串口或者以太网口相连。串口连接线是一根 RS-232 接口的电缆,在其内部两端的第 2 脚(TXD)与第 3 脚(RXD)交叉相连,第 7 脚(接地脚)直接相连。调试过程中,被调试的内核运行在目标机上,gdb 调试器运行在开发机上。在 FreeBSD 操作系统上还允许使用 FireWire 接线,并用直接内存访问(Direct Memory Access,DMA)的功能来协助远程调试。

在某些架构的处理器中,会以硬件方式提供一些调试功能的寄存器,以及可以设置观察点(Watchpoint)。使用观察点可以在指定的存储器地址被运行到或访问到时去触发观察点、触动一个中断点。

GDB 运用上最明显的限制是在“用户界面”的部分,默认只提供命令行接口(CLI),不过有一些前端程序可以进行弥补,包括 DDD、GDBtk/Insight、Xxgdb、Xcode debugger 等。另外一些集成开发环境(Code::Blocks, Dev-C++, Geany, KDevelop, Qt Creator, Eclipse 和 NetBeans 等)可以与 GDB 集成。

另外,有些排错工具也被设计成能与 GDB 搭配使用,例如存储器泄漏(memory leak)的检测程序。

在命令行中使用 GDB 的示例如下:

```
gdb program           debug "program" (from the shell)
run -v                run the loaded program with the parameters
bt                    backtrace (in case the program crashed)
info registers        dump all registers
disass $pc-32, $pc+32 disassemble
```

下面是一段 C 语言代码,其中可能会有某些错误。使用 GCC 编译时要使用-g 参数来包含相关的调试信息,然后就可以使用 GDB 进行调试。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t
foo_len (const char *s)
{
  return strlen (s);
}

int
main (int argc, char *argv[])
{
  const char *a = NULL;
  printf ("size of a = %d\n", foo_len (a));
  exit (0);
}
```

以下是用 GDB 进行除错的一段过程示例,欲进行侦错的程序已在堆栈追踪(Stack trace)区内:

```
[theqiong@localhost ~]$ gcc example.c -g -o example
[theqiong@localhost ~]$ ./example
Segmentation fault (core dumped)
[theqiong@localhost ~]$ gdb ./example

GNU gdb (GDB) Fedora (7.3.50.20110722-13.fc16)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+:GNU GPL version 3 or later<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/example...done.
(gdb) run
Starting program: /path/example

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400527 in foo_len (s=0x0) at example.c:8
8         return strlen (s);
(gdb) print s
$1 = 0x0
```

gdb 返回的信息显示问题出现在第 8 行，当调用 strlen 函数时的参数是 NULL。根据 strlen 函数的不同实现，gdb 的调试信息也可能是下面这样：

```
$ gdb ./example
GNU gdb (GDB) 7.3.1
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+:GNU GPL version 3 or later<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/gdb/example...done.

(gdb) run
Starting program: /tmp/gdb/example
Program received signal SIGSEGV, Segmentation fault.
0xb7ee94f3 in strlen () from /lib/i686/cmov/libc.so.6
(gdb) bt
#0  0xb7ee94f3 in strlen () from /lib/i686/cmov/libc.so.6
#1  0x08048435 in foo_len (s=0x0) at example.c:8
#2  0x0804845a in main (argc=<optimized out>, argv=<optimized out>)
at example.c:16
```

这个程序已处在运行阶段，之后通过 gdb 来找出这个程序中会导致运行错误的段落，然后将对应处的原代码用编辑器进行错误修订，更正完成后用编译器重新编译并再次运行。

这里为了修复错误，主函数中的变量 a 必须包含一个有效的字符串，下面是一个修复后的版本：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t
foo_len (const char *s)
{
  return strlen (s);
}

int
main (int argc, char *argv[])
{
  const char *a = "This is a test string";

  printf ("size of a = %d\n", foo_len (a));

  exit (0);
}
```

当再次以 gdb 进行调试时,会返回一个正确的结果。

```
GNU gdb (GDB) Fedora (7.3.50.20110722-13.fc16)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+:GNU GPL version 3 or later<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/example...done.
(gdb) run
Starting program: /path/example
size of a = 21
[Inferior 1 (process 14290) exited normally]
```

这里,gdb 将 printf 的输出结果打印到屏幕,并通知用户修改后的程序已经正常了。

第 11 章

Maintenance

软件开发的一个特殊方面是程序需要维护。

考虑一辆轿车或一座桥的情况，只有某些地方损坏时才需要维护，如一些金属部件生锈了，一个机械联动装置因过度使用而磨损了，或一些部件在事故中毁坏了，这时需要维护。

而这些情况都不会发生在软件上，代码本身不会生锈，一次次的重复使用同一个程序不会在任何方面降低其可用性，偶然的错误使用会有危险结果，但并不会破坏程序本身，即使破坏了也可以从备份中重建程序。

大部分程序必须周期性地更新以纠正逻辑错误，或应应用的要求修改，这个过程就称为软件维护。对程序开发的研究表明，大多数程序在发布后，用于维护的费用占到了总费用的 80% ～ 90%。

软件需要维护主要有两个原因。首先，即使经过大量测试，并在相关领域使用多年，源代码中仍然可能存在逻辑错误，其次，当出现一些不常见的情况或发生之前未预料到的情况时，之前隐藏的逻辑错误就会使程序运行失败。因此，调试是程序维护的一个基本部分。

但调试并非最重要的部分，更为重要的部分或许可以称为功能增强，尤其是从占据程序总体维护费用的百分比角度来说，编程是为了使用，用程序完成客户们需要完成的任务通常比其他方法更快更便宜。但同时，程序不可能完成顾客想做的每一件事，在某个程序使用了一段时间之后，客户会期望程序能做些别的事，或者有更大的容量，或者有一些更为简单但更有吸引力的特色等，由于软件具有极大的灵活性，供应商可选择响应这些要求。

不管人们是想要纠正逻辑错误还是要增强功能，在任何情况下都需要有人查看程序，找出需要做什么，做出必要的修改，确保这些修改能正确地工作，然后发布新的版本。

软件维护的过程一般来说，既费时又费力，花费昂贵而且容易出错，同时还会引入新的潜在的逻辑错误等。

使程序维护如此困难的一部分原因是大部分的程序员并不会花很长时间来编写他们的程序，对他们来说，程序能够工作就足够了[1]。使其他程序员能理解程序并维护它的编程学科，称为软件工程（Software Engineering）。

当在写程序时，应当想象一下两年以后，其他人来看这个程序时会有什么感受，程序思路清楚吗？程序自身能告诉读者你想要做什么吗？它容易修改吗？尤其是在一些你期望能做修改的地方？它看上去含糊不清且令人费解吗？设身处地地为将来的维护人员着想，有助于理解并编写出具备良好的（软件）工程风格的软件。

良好的软件工程并不像烹饪书一样有章可循，相反地，它是一门技巧，而且有一定的艺术性，在此，实践很重要，人们通过自己编写和阅读他人的程序而学习写一个良好的程序。

出色的程序设计能力需要训练[2]——训练在完成一项设计的匆忙过程中不走捷径，也不忘记未来维护人员的工作。良好的程序风格需要培养一种美感，懂得对一个程序来说，可读性和良好的表述意味着什么。

[1] 需要为程序员辩护的是，他们这样做通常是被迫的，因为公司的经费紧张，时间有限。但草率地将软件投放市场会给经济带来不良影响，因为最终公司要花更大的代价来进行额外的系统维护。

[2] 卡耐基梅隆大学的 Larry Flon（后来在 UCLA）在一篇关于程序设计语言的著名评论中说道："没有一种程序设计语言能够编写出完美无瑕的程序，这种语言过去没有，以后也不会有。"重要的不是语句结构，而是仔细和训练。

Bibliography

[1] Wikipedia. Gnu c library. URL http://en.wikipedia.org/wiki/GNU_C_Library.

[2] Wikipedia. Llvm, 2000. URL http://zh.wikipedia.org/wiki/LLVM.

Part II

Foundation

第 12 章

Introduction

在很多方面，学习任何程序设计语言都与学习使用外语与人沟通一样，我们需要掌握一定的词汇以知道其语义，需要学习语法以便连词成句，还必须了解一些惯用语以理解人们所说的实际意义。

每一种程序设计语言可以被看作是一套包含语法、词汇和含义的正式规范。这些规范通常包括：

- 数据和数据结构
- 指令及流程控制
- 引用机制和重用
- 设计哲学

另外，程序设计语言中的不成文规定还包括标识符（Identifier）命名规范等，比如关键字、保留字等。

现代计算机内部的数据都只以二元方式储存，即开-关模式（on-off）。现实世界中代表信息的各种数据，例如名字、银行账号、度量以及同样低端的二元数据，都经由程序设计语言处理，成为抽象的概念。

除了何时以及如何确定表达式和类型的联系，另外一个重要的问题就是语言到底定义了哪些类型，以及允许哪些类型作为表达式的值。比如，C 语言允许程序命名内存位置、内存区域以及编译时的常量，而且还允许表达式返回结构值（struct values）等。

功能性的语言一般允许变量直接使用运行时计算出的值，而不是指出该值可能储存的内存地址。

一个程序中专门处理数据的系统被称为程序语言的类型系统（type system），对类型系统的研究和设计被称为类型理论（type theory）。

编程语言可以被分为静态类型系统（statically typed systems）和动态类型系统（dynamically typed systems）。前者可被进一步分为包含声明类型（manifest type）的语言，即每一个变量和函数的类型都清楚地声明，或 type-inferred 语言（例如 MUMPS，ML）。

- 静态类型系统（statically typed systems），例如 C++ 和 Java 等。
- 动态类型系统（dynamically typed systems），例如 Lisp、JavaScript、Tcl 和 Prolog 等。

大多数语言还能够在内置的类型基础上组合出复杂的数据结构类型（使用数组、列表、堆栈、文件等等）。面向对象语言（Object Oriented Language）允许定义新的数据类型——即“对象”，以及运行于该对象的函数（functions）和方法（methods）。

常见的数据结构包括：

- 数组 (array)
- 堆栈 (stack)
- 队列 (queue)
- 链表 (linked list)
- 树 (tree)
- 图 (graph)
- 堆 (heap)
- 散列 (hash)

一旦数据被确定，机器必须被告知如何对这些数据进行处理。较简单的指令可以使用关键字或定义好的语法结构来完成。除此之外，还有其他指令可以用来控制处理的过程（例如分支、循环等）。

引用的中心思想是必须有一种间接设计储存空间的方法。最常见的方法是通过命名变量。根据

不同的语言,进一步的引用可以包括指向其他储存空间的指针。还有一种类似的方法就是命名一组指令。大多数程序设计语言使用宏调用、过程调用或函数调用。使用这些代替的名字能让程序更灵活,并更具重用性。

对那些从事计算机科学的人来说,懂得程序设计语言是十分重要的,因为在当今所有的计算都需要程序设计语言才能完成。例如,通过低级编程语言或高级编程语言,将实体间接达成传输链接或控制实体。高级编程语言可用于开发多功能的应用程序软件,如操作系统、工程计算软件等。

程序设计语言原本是被设计成专门使用在计算机上的,但它们也可以用来定义算法或者数据结构。程序设计是给出解决特定问题程序的过程,用程序实现设计好的算法的过程就是编码(Coding)。

程序设计语言使程序员能够比使用机器语言更准确地表达他们所想表达的目的,并可通过机械、计算机来完成人类需求的演算与功能。正是因为如此,程序员才会试图使程序代码更容易阅读。

软件开发过程往往以某种程序设计语言为工具,给出这种语言下的程序。具体而言,可以划分为分析、设计、编码、测试、调试等不同阶段。

程序设计的精髓在于解决问题,学习如何表达固然重要,但是如何找出解决问题的方法则是更大的挑战。

12.1 Hello, world.

为纪念 C 语言的设计者，第一个程序范例选自 C 语言的定义文档——Brian Kernighan 和 Dennis Ritchie 合著的 The C Programming Language，这个范例就是“Hello world”程序。

```
/*
*  File:hello.c
*  ------------
*  This program prints the message “Hello,world.” on the screen.
*  The program is taken from the classic C reference text “The
*  C Programming Language” by Brain Kernighan and Dennis Ritchie.
*/

#include <stdio.h>
#include "genlib.h"

main()
{
   printf("Hello,world.\n");
}
```

一个 C 语言源程序分为三个部分：程序注释、指令(库包含)和主程序。

```
/*
 * comments
 */

/*
 * instructions(preprocess instructions, declarations, etc)
 */

main()
{
  ... optional declarations ...
  ... body ...
}
```

即使是最简单的 C 语言程序也依赖这三个关键的语言特性：

1. 指令：在编译操作前修改程序的编辑命令；
2. 函数：被命名的可执行代码块，例如 main 函数；
3. 语句：程序运行时依次执行的命令。

12.2 Comments

在 C 语言中，注释(comment)是在“/*”与“*/”之间所有的文字。

注释是用自然语言描述程序的作用，可以占用连续的几行[1]，但标准 C 不允许一个注释嵌套在另一个注释中。

注释是写给人看的，而不是写给计算机的，它们向其他程序员传递该程序的有关信息，包括程序名、编写日期、作者、程序的用途以及其他内容。

[1]从 C99 开始，C 语言也支持以//开头的单行注释(这个特性实际上在 C89 的很多编译器上已经被支持了)。

- 每一个程序都应以一个专门的并从整体上描述程序操作过程的注释开始，这段注释称为程序注释（program comment），它包括程序文件的名称和一些描述程序操作的信息，以及程序的来源等信息。
- 注释还可以描述程序中特别复杂的部分，指出可能的使用者，对如何改变程序行为提出一些建议等。
- 当程序越来越复杂时，给出合适的注释是使程序易读的最好方法之一。

当 C 语言编译器将程序转换为可由机器执行的形式时，注释被完全忽略[2]。

12.3 Library

1940 年代当计算机刚刚问世的时候，程序员必须手动控制计算机。当时的计算机十分昂贵，可能唯一想到利用程序设计语言来解决问题的人是德国工程师康拉德·楚泽。

几十年后，计算机的价格大幅度下跌，而计算机程序也越来越复杂。也就是说，开发时间已经远比运行时间宝贵。于是，新的集成的、可视化的开发环境越来越流行。它们减少了所付出的时间、金钱（以及脑细胞），简化了程序开发流程，这当然也得益于可以重用的程序代码库。

库（library）是一种工具的集合，这些工具由其他程序员实现并用于执行特定的功能。例如，在程序范例 hello.c 中使用了两个库，分别是由 ANSI C 提供的标准输入输出库（stdio）和用户开发的通用库（genlib）。

C 语言本身并没有提供内置的库来处理通常的操作（比如输入/输出、内存管理、字符串操作等）。实际上，在编译和链接程序时，这些操作是通过 C 标准函数库来提供相关功能的。

C 标准函数库（C Standard library）是所有目前符合 ISO C 标准的头文件（head file）的集合，以及常用的标准函数库实现程序（例如 I/O 输入输出和字符串控制等）[3]。不像 COBOL、Fortran 和 PL/I 等编程语言，在 C 语言的工作任务里不会包含嵌入的关键字，所以几乎所有的 C 语言程序都是由标准函数库的函数来创建的。

不同的操作系统和编译器都根据 ISO C 标准提供了相应的标准库的实现，这样程序员使用库提供的工具就可以不用自己重新实现这些功能。

当程序需要包含库时，必须对每一个库使用一行 #include 以明确指定头文件，当需要编写一些更为复杂的程序时，会依赖一些重要的库。

要使用一个库，就必须在程序中给出足够的信息，以便编译器知道库里有哪些工具可用。在大多数情况下，这些信息以头文件（header file）的形式提供，头文件为编译器提供了对应库所定义的工具的描述。例如，stdio.h 是一个头文件的名称。

C 语言程序包含 stdio.h 的原因是 C 语言没有内置的“读”和“写”指令，因此进行输入/输出操作就需要标准输入输出库中的函数来实现，而 genlib.h 也是一个头文件的名称，它定义了通用库的内容。

这里，两个 #include 行使用的标点符号（双引号）是不同的。

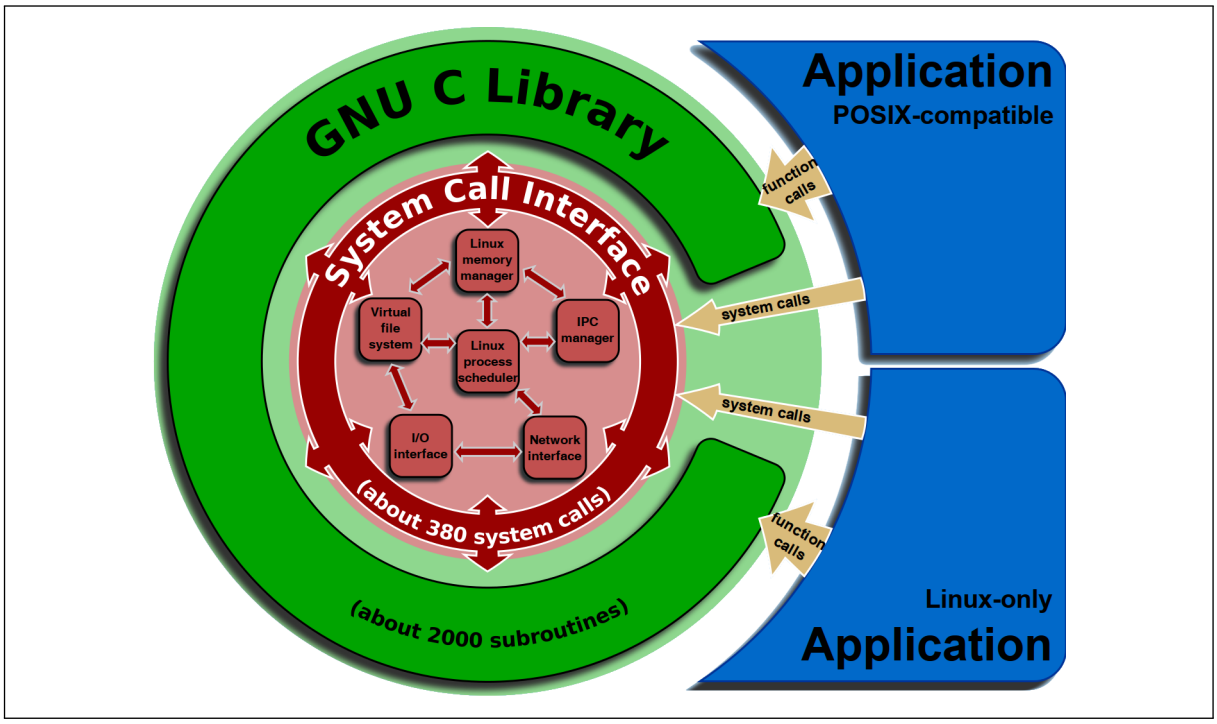
```
#include <stdio.h>
#include "genlib.h"
```

其中，stdio 库是使用 ANSI C 时总能获取的标准库，头文件用尖括号标记，个人编写的库以及其他的扩展库的头文件用双引号标明。

在编译 C 程序之前，预处理器会首先对 C 程序进行编辑，把预处理器执行的命令称为指令（instruction），例如 #include 指令说明在编译前把指定的头文件中的信息“包含”到程序中。

[2] 一些早期的编译器是简单的删除每条注释中的所有字符，后来标准 C 规定编译器必须用一个空格符替换每条注释。

[3] 一般情况下，标准库随编译器一起提供，它们一般是已预先编译好，以二进制代码形式存在于系统中。比如，Linux 操作系统随 gcc 编译器一起提供的 GNU C 库（GNU C Library，又称为 glibc）就是一种 C 函数库，它是程序运行时使用到的一些 API 集合。



12.4 Header file

每一个函数的名称与特性会被写成一个计算机文件，这个文件就称为头文件 (header file)，但是实际的函数实现是被分存到函数库文件里。

在程序设计中，特别是在 C 语言和 C++ 中，头文件或包含文件是一个文件，通常是源代码的形式，由编译器在处理另一个源文件的时候自动包含进来。一般来说，程序员通过编译器指令将头文件包含进其他源文件的开始(或头部)。

一个头文件一般包含类、子程序、变量和其他标识符的前置声明。需要在一个以上源文件中被声明的标识符可以被放在一个头文件中，并在需要的地方包含这个头文件。在 C 语言和 C++ 中，标准库函数习惯上在头文件中声明。

头文件的命名和领域是很常见的，但是函数库的组织架构也会因为不同的编译器而有所不同，标准函数库通常会随附在编译器上。因为 C 编译器常会提供一些额外的非 ANSI C 函数功能，所以某个随附在特定编译器上的标准函数库，对其他不同的编译器来说，是不兼容的。

大多 C 标准函数库在设计上做得相当不错。有些少部分的，会为了商业优势和利益，会把某些旧函数视同错误或提出警告。字符串输入函数 gets()(以及 scanf() 读取字符串输入的使用上)是很多缓存溢出的原因，而且大多的程序设计指南会建议避免使用它。另一个较为奇特的函数是 strtok()，它原本是作为早期的词法分析用途，但是它非常容易出错，而且很难使用。

1995 年，Normative Addendum 1 (NA1) 批准了三个头文件 (iso646.h, wchar.h, and wctype.h) 增加到 C 标准函数库中。C99 标准增加了六个头文件 (complex.h, fenv.h, inttypes.h, stdbool.h, stdint.h, and tgmath.h)。C11 标准中又新增了 5 个头文件 (stdalign.h, stdatomic.h, stdnoreturn.h, threads.h, and uchar.h)。至此，C 标准函数库共 29 个头文件。

名字	源自	描述
<assert.h>		包含断言宏，被用来在程序的调试版本中帮助检测逻辑错误以及其他类型的 bug。
<complex.h>	C99	一组操作复数的函数。

名字	源自	描述
<ctype.h>		定义了一组函数，用来根据类型来给字符分类，或者进行大小写转换，而不关心所使用的字符集（通常是 ASCII 或其扩展字符集，也有 EBCDIC）。
<errno.h>		用来测试由库函数报的错误代码。
<fenv.h>	C99	定义了一组用来控制浮点数环境的函数。
<float.h>		Defines macro constants specifying the implementation-specific properties of the float-point number library.
<inttypes.h>	C99	定义具有固定大小的整数类型
<iso646.h>	NA1	Defines several macros that are equivalent to some of the operators in C. For programming in ISO 646 variant character sets.
<limits.h>		Defines macro constants specifying the implementation-specific properties of the integer types.
<locale.h>		定义 C 语言本地化函数.
<math.h>		定义 C 语言数学函数。
<setjmp.h>		定义了宏 setjmp 和 longjmp，在非局部跳转的时候使用。
<signal.h>		定义 C 语言信号处理函数。
<stdalign.h>	C11	For querying and specifying the data structure alignment of objects.
<stdarg.h>		为了访问传递给函数的可变数目的参数
<stdatomic.h>	C11	用于在线程之间进行数据共享的原子操作
<stdbool.h>	C99	定义布尔数据类型
<stddef.h>		定义一些有用的类型和宏
<stdint.h>	C99	定义整数类型
<stdio.h>		定义核心输入/输出函数
<stdlib.h>		Defines numeric conversion functions, pseudo-random numbers generation functions, dynamicmemory allocation, process control functions
<stdnoreturn.h>	C11	用于指定非返回函数
<string.h>		定义 C 语言字符串处理函数
<tgmath.h>	C99	定义通用类型的数学函数
<threads.h>	C11	定义管理多线程、互斥锁以及条件变量的函数
<time.h>		定义日期和时间处理函数
<uchar.h>	C11	定义用于操作 Unicode 字符的类型和函数
<wchar.h>	NA1	定义宽字符处理函数
<wctype.h>	NA1	定义了一组根据类型对宽字符分类或对它们进行大小写转换的函数

来自 C 标准库的所有头文件，会以另一个名称包含在 C++ 标准中，一般是将原名称移去“.h”并在开头处加上“c”作为新的名称，例如“time.h”改成“ctime”。C++ 标准库的头文件与 C 标准库的头文件的唯一区别是，函数位于 `std::` 命名空间（虽然很少编译器真正如此）。

在访问声明在不同文件中的标识符问题上，头文件不是唯一的解决方法。他们也有缺点，当定义改变的时候可能仍然需要在两个地方来修改（头文件和源文件）。一些更新的语言（如 Java）省略掉了头文件，而使用命名规则（naming scheme），这就允许编译器来定位与接口和类实现相关的源文件。

12.5 Main

在 hello.c 程序中，程序的主体由如下内容组成：

```
main()
{
  printf("Hello,world.\n");
}
```

在 C 语言源程序中，函数（function）是一系列独立的程序步骤，类似于其他编程语言中的“过程”或“子程序”。

在 C 语言源程序中，将相应的程序步骤集合在一起，并赋予一个名字，就形成了一个函数。函数可以理解为构建程序的“块”，事实上 C 语言程序就是函数的集合。

函数所执行的步骤在大括号中列出[4]，这些步骤称为语句（statement）。大多数函数都由若干条语句组成，这些语句共同组成了函数的主体（body）。

尽管 C 语言程序可以包含多个函数，但是强制规定每一个完整的 C 语言程序必须有一个 main 函数，称为主控函数。

main 函数是程序执行的入口，在执行程序时系统会自动调用 main 函数，而且 main 函数程序终止时会向操作系统返回一个状态码，因此建议在 main 函数的末尾使用 return 语句结束，否则某些编译器可能会产生警告信息。

在运行 C 语言程序时，计算机从调用 main 函数开始，依次执行 main 函数主体中包含的语句。main 函数又调用其他函数，这些函数分别解决一部分问题。接下来，这些函数还可能调用其他可以进一步细化的问题的函数。例如在 Hello,world 示例程序中，main 函数调用了库函数 printf，printf 函数可以用于在屏幕上显示信息和数据值。

printf 函数是标准输入输入库中的工具，只要包含了库 stdio，就可以使用 printf 函数。

```
#include <stdio.h>
```

函数名称 printf 代表一组操作，当想要调用这些操作时，只要使用其函数名就可以一起引用这些函数。在程序设计中，通过使用某个函数名来执行指派的任务的行为称为调用（calling）函数。

在调用一个函数时，常常要提供一些额外的信息。例如，在 C 语言中，printf 是一个在屏幕上显示数据的函数，但应该显示什么数据呢？这些额外信息由一组出现在函数名后的括号内的参数说明，参数（argument）是一个函数的调用程序提供给函数的信息。

在 hello.c 程序中，printf 函数被赋予了一个参数，它是一个列于引号中的一串字符，称为字符串字面量（string literal）。字符串字面量是用双引号包围的一系列字符，而且当显示字符串字面量时，系统不会显示双引号，这也是程序设计中数据的一个例子。

```
"Hello, world.\n"
```

C 语言中有许多不同类型的数据，应该特别注意如何使用数据的问题。一般地，可以简单将数据（data）看作程序所操作的信息：所有要显示的信息，用户要求的输入，作为计算结果传递的数值或计算过程的中间结果。

这里字符串中最后一个字符为特殊字符，称作换行字符，用\n 表示。当函数 printf 到达句末的句点时，光标会停留于文本末尾的句点后。若想扩展程序使之显示更多的信息，也许会想在每个新信息开始时另起一行。换行字符（newline）使其成为可能，该字符在所有的计算机系统中都进行了定义，并且可以在一个字符串字面量中出现多次。

[4] C 语言使用 {和} 的方式非常类似于其他语言中的 begin 和 end 的用法，这也说明了 C 语言极度依赖缩写词和特殊符号。

当函数 printf 处理换行字符时，通过“\n”换行符可以终止当前行，并且把后续的输出转到下一行进行，因此屏幕上的光标会移至下一行的起始处继续显示，就像敲击了键盘上的 Return 键或 Enter 键一样。

在 C 语言中，程序必须包含换行字符以标记每一行的行尾，否则所有的输出结果将不分行地显示在一起。

```
/*
* File:add2.c
* ------------
* This program reads in two numbers, adds them together
* and prints their sum.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
   int n1,n2,total;

   printf("This program adds two numbers.\n");
   printf("1st number?");
   n1 = GetInteger();
   printf("2nd number?");
   n2 = GetInteger();
   total = n1 + n2;
   printf("The total is %d.\n",total);
}
```

在程序设计中，变量(variable)是一些在编写程序时值未知的数据的存放处，例如这里在编写两个数的加法程序时，程序员尚不知道用户要将哪两个数相加。当程序运行时，用户才会输入两个数。

为了在程序中引用这些目前尚未确定的数，可创建一个变量来保存这些需要记住的值，并给该变量命名。一旦要用到它包含的值时，可使用其变量名。

变量的名称要用心选择，以便将来阅读程序的程序员能够容易地分辨出每个变量的作用。

在 C 语言中，使用变量之前，必须先声明该变量。声明(declaring)一个变量就是告知 C 编译器引入了一个新的变量名，并指定了该变量可以保存的数据类型。

下面的语句声明了三个变量：n1，n2 和 total，并告知 C 编译器每个变量中保存一个整数。

```
int n1,n2,total;
```

C 语言程序的运行都是通过执行 main 函数中的每一条语句来实现的，因此在该程序执行第一条语句时，计算机仅在屏幕上显示如下消息：

```
This program adds two numbers.
```

并将光标移至下一行的起始处，该信息也说明了程序的作用。

程序运行开始后,会先显示该消息,然后余下的部分则可分为三个阶段,分别是输入阶段、计算阶段和输出阶段。

1. 输入阶段

在输入阶段,计算机必须要求用户输入两个加数并分别保存在变量 n1 和 n2 中。每个加数的输入过程包括两步。首先,程序应在屏幕上显示一个消息以使用户了解程序需要什么,这种类型的消息通常称为提示消息(prompt)。和向用户显示的其他消息一样,可使用 printf 来显示提示消息和其他消息。

大多数情况下,用于显示输出数据的 printf 函数调用需要用到换行字符,用于显示输入数据提示消息的 printf 函数则不需要使用换行字符。

为了读取数据,程序可以使用如下这个语句:

```
n1=GetInteger();
```

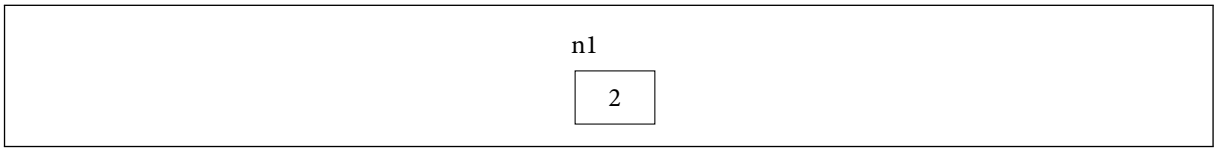
该语句是一个程序结构的实例,称为赋值语句。在 C 语言中,赋值语句(assignment statement)将等号右边的数值存储与等号左边的变量中。在本例中,赋值语句的右边是对函数 GetInteger 的调用,左边是变量 n1。

函数 GetInteger 是 simpio 库的一部分,用于从用户处读入整数。调用函数 GetInteger 时,程序等待用户用键盘输入一个完整的数值。当用户输入完毕并按下回车键后,该数值作为函数 GetInteger 的结果返回主程序。在程序设计的术语中,称 GetInteger 返回(return)用户键入的数值。

总的来说,这个赋值语句的作用就是调用 GetInteger 函数,让用户输入数值,并最终将 GetInteger 返回的数值存储于变量 n1 中。

在纸上跟踪程序的操作过程时,程序员通常使用一个简单的方框图形表示已将一个特定的值赋给一个变量,每个变量对应于图中的一个方框,每个方框有一个名字和一个值,方框的名字在函数运行过程中始终不变。当新的值存储到该变量中时,方框中的值随之改变。

为表明赋值语句已将数值 2 存储于变量 n1 中,就要画出一个方框,命名为 n1,并在框中写上 2 以表示该变量的值,如下图所示:



2. 计算阶段

程序的计算阶段由计算出两个数的和组成。在程序设计中,计算是通过写一个表达式来指定的,该表达式指定了必须的操作步骤,表达式的结果由赋值语句存储于一个变量中,以便程序后面的部分使用该结果。

和 C 语言中的任何赋值语句一样,计算机计算等号右边表达式的值,并将结果存储到表达式左边的变量中。

```
total =  n1 + n2;
```

这里的赋值语句的作用就是将变量 n1 和 n2 中存储的两个值相加,并将结果存储到变量 total 中。

3. 输出阶段

程序的输出阶段由执行显示计算结果的语句组成。可以用 printf 函数来完成结果的显示。

```
printf("The total is %d.\n",total);
```

这里% 及其后面的字母称为格式码(format code)。printf 函数的格式码的作用就是作为值的占位符。在输出过程中,值被插入到该位置,格式码中的字母用于指定输出格式。

另外, printf 函数可将任何数值作为输出的一部分显示出来。若想将一个整数作为输出的一部分, 就应该在 printf 函数调用的第一个参数中包含格式码%d, 而将要显示的实际值可以作为 printf 的另一个参数,按它们出现的次序排列。例如,若 n1 = 1 , n2 = 3 ,则下列语句对应的输出为:

```
    printf("%d + %d = %d.\n",n1,n2,n3);
```

结果就是:1 + 3 = 4。

12.6 Compiling

在 Linux 下面,如果要编译一个 C 语言源程序,可以使用 GCC 或 Clang。
要编译示例程序,我们只要在命令行下执行:

```
[theqiong@home ~] gcc hello.c
```

gcc 编译器将源文件编译为一个可执行文件—a.out

```
[theqiong@home ~] ll a.out
-rwxrwxr-x 1 theqiong theqiong 8502 Mar 7 02:12 a.out
[theqiong@home ~] ./a.out
Hello,world.
```

如果在 gcc 后面加上参数-o,并指定将源文件编译为某个特定名字的可执行文件,此时需要在命令行下执行如下命令:

```
[theqiong@home ~] gcc -o test hello.c
[theqiong@home ~] ll test
-rwxrwxr-x 1 theqiong theqiong 8503 Mar 7 02:24 test
[theqiong@home ~] ./test
Hello,world.
```

命令行中以 gcc 开始表示用 gcc 来编译源程序, -o 选项表示要求编译器输出的可执行文件名为 test,而 hello.c 是源程序文件。

gcc 编译器有许多选项,其中:

- -o 选项表示要求输出的可执行文件名。
- -c 选项表示只要求编译器输出目标代码,而不必要输出可执行文件。
- -g 选项表示要求编译器在编译的时候提供后续对程序进行调试的信息。

如果使用 Clang 编译器来编译 hello.c,可以在命令行中执行:

```
[theqiong@home ~] clang hello.c
hello.c:3:1: warning: type specifier missing, defaults to 'int'
[-Wimplicit-int]
main(){
^~~~
1 warning generated.
[theqiong@home ~] ll a.out
-rwxrwxr-x 1 theqiong theqiong 8401 Mar 7 02:33 a.out
[theqiong@home ~] clang -o testc hello.c
hello.c:3:1: warning: type specifier missing, defaults to 'int'
[-Wimplicit-int]
main(){
^~~~
1 warning generated.
```

```
[theqiong@home ~] ll testc
-rwxrwxr-x 1 theqiong theqiong 8401 Mar 7 02:39 testc
[theqiong@home ~] ./testc
Hello,world.
```

为了演示程序运行与用户输入的交互过程,下面将上述示例代码进行扩充如下:

```
/* main.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Hello,world.\n");
  printf("%s\n", argv[0]);
  printf("%s\n", argv[1]);
  printf("%s\n", argv[2]);
  return 0;
}
```

在这里, main 函数的形式为 `int main(int argc, char *argv[])`。在编译完成后运行程序,会发现参数 `argc` 代表的是用户输入的数据的数目,而数组 `argv[]` 的元素则是用户输入的数据。

```
[theqiong@home ~] gcc -o program source
[theqiong@home ~] ./program good bye
Hello,world.
./program
good
bye
```

其中:

- `argc` 在这里是 3;
- `argv[0]` 在这里是`./program`;
- `argv[1]` 在这里是`"good"`;
- `argv[2]` 在这里是`"bye"`。

这个示例程序还可以改写成给直观的形式如下:

```
#include <stdio.h>

int main(int argc,char *argv[])
{
  printf("you entered in reverse order:\n");

  while(argc--)
  {
    printf("%s\n",argv[argc]);
  }

  return 0;
}
```

可以以 for 循环改写成另一种形式,代码示例如下:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  printf("The following arguments were passed to main(): ");
  for(i=1; i<argc; i++) {
    printf("%s ", argv[i]);
  }
  printf("\n");
  return 0;
}
```

12.7 Makefile

假设现在有这样一个程序,源代码如下:

```
/* main.c */
#include "mytool1.h"
#include "mytool2.h"
int main(int argc,char **argv)
{
  mytool1_print("hello");
  mytool2_print("hello");
}
/* mytool1.h */
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif
/* mytool1.c */
#include "mytool1.h"
void mytool1_print(char *print_str)
{
  printf("This is mytool1 print %s\n",print_str);
}
/* mytool2.h */
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif
/* mytool2.c */
#include "mytool2.h"
void mytool2_print(char *print_str)
```

```
{
  printf("This is mytool2 print %s\n",print_str);
}
```

对于上述三段示例程序,可以这样来编译:

```
[theqiong@home ~] gcc -c main.c
[theqiong@home ~] gcc -c mytool1.c
[theqiong@home ~] gcc -c mytool2.c
[theqiong@home ~] gcc -o main main.o mytool1.o mytool2.o
```

这样的话我们也可以产生 main 程序,而且也不是很麻烦。但是考虑一下如果后面修改了其中的一个文件 (比如说 mytool1.c),可能就要重新输入上面的命令来进行编译。当程序源文件规模不断增加时,这样的方式就不再适合开发工作。

为此,我们可以使用 make 工具来完成这类工作,只要执行以下 make 命令就可以完成程序的编译,但在执行 make 之前,要先编写一个非常重要的文件—Makefile。对于上面的示例程序来说,可能的 Makefile 文件如下:

```
# make main
main:main.o mytool1.o mytool2.o
gcc -o main main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
gcc -c main.c
mytool1.o:mytool1.c mytool1.h
gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
gcc -c mytool2.c
```

这样,如果后续修改了源程序当中的某些文件时,只要执行 make 命令,编译器都只会去编译和被修改的文件有关的文件,其它的文件保持不动。

在 Makefile 中,以 # 开始的行都是注释行,Makefile 中最重要的是描述文件的依赖关系的说明,其一般的格式是:

```
target: components
TAB rule
```

其中,第一行表示的是依赖关系,第二行是规则,其中 TAB 表示那里是一个 TAB 键。比如上面的 Makefile 示例文件的第二行中

```
main:main.o mytool1.o mytool2.o
```

表示目标 (target)main 的依赖对象 (components) 是 main.o,mytool1.o 和 mytool2.o。

当倚赖的对象在目标修改后经过修改的话,就要去执行 Makefile 规则所指定的命令,这里是:

```
gcc -o main main.o mytool1.o mytool2.o
```

Makefile 有三个非常有用的变量,分别是 $@,$^ 和 $<,它们代表的意义分别是:

- `$@` —— 目标文件;
- `$^` —— 所有的依赖文件;
- `$<` —— 第一个依赖文件。

如果使用上面三个变量,那么可以简化 Makefile 示例文件为下面的形式:

```
# Simplified Makefile
main:main.o mytool1.o mytool2.o
gcc -o $@ $^
main.o:main.c mytool1.h mytool2.h
gcc -c $<
mytool1.o:mytool1.c mytool1.h
gcc -c $<
mytool2.o:mytool2.c mytool2.h
gcc -c $<
```

如果还想对 Makefile 继续简化，这里可以使用 Makefile 的缺省规则。

```
..c.o:
gcc -c $<
```

这个规则表示所有的.o 文件都是依赖与相应的.c 文件的，例如 mytool.o 依赖于 mytool.c，这样 Makefile 就可以继续简化成：

```
# Finally Simpilified Makefile
main:main.o mytool1.o mytool2.o
gcc -o $@ $^
..c.o:
gcc -c $<
```

12.8 Linking

当我们用 `gcc -o temp temp.c` 编译下面的 temp.c 示例程序时会出现错误。

```
/* temp.c */
#include <math.h>;
int main(int argc,char **argv)
{
  double value;
  printf("Value:%f\n",value);
}
```

具体错误提示可能是：

```
/tmp/cc33Kydu.o: In function `main':
/tmp/cc33Kydu.o(.text+0xe): undefined reference to `log'
collect2: ld returned 1 exit status
```

出现这个错误是因为编译器找不到 log 的具体实现，虽然程序中包括了正确的头文件，但是在编译的时候还是要连接确定的库。在 Linux 下，为了使用数学函数，我们必须和数学库连接，为此要加入`-lm` 选项，即

```
gcc -o temp temp.c -lm
```

这样才能够正确的编译。

对于一些常用的函数的实现，gcc 编译器会自动去连接一些常用库，这样我们就没有必要自己去指定了。有时在编译程序的时候还要指定库的路径，此时要用到编译器的`-L` 选项指定路径，比如说有

一个库在 /home/theqiong/mylib 下，这样当编译的时候就还要加上`-L/home/theqiong/mylib`。对于一些标准库来说，我们没有必要指出路径，只要它们在起缺省库的路径下就可以了。系统的缺省库的路径分别是/lib、/usr/lib、/usr/local/lib，在这三个路径下面的库，可以不指定路径。

12.9 Debuging

我们编写的程序不太可能一次性就会成功，在程序当中会出现许许多多想不到的错误（比如有未终止的注释等），这个时候就要对程序进行调试了。

最常用的调试软件是 GDB，如果需要在图形界面下调试程序，那么可以选择 xxgdb 或 ddd，不过要在编译的时候加入`-g` 选项。

GDB 具备各种侦错功能，能针对计算机程序的运行进行追踪与警告，使用 GDB 可以监视及修改程序的内部变量值，以及监视与修改独立于主程序运行外，以独立个体型态调用（调用使用）的函数。例如，使用 GDB 逐行的运行程序，就可以发现是否有些行被跳过了。

GDB 的“远程”模式多半运行在嵌入式系统排错，比如 GDB 在一部机器内运行，而要进行侦错的程序是在另一部机器上运行，接着欲排错的机器上会再加装一个名为“Stub”的小程序来与另一端的 GDB 程序沟通。沟通的路径可以是两部机器间的串接式接线（Serial Cable），也可以是支持 TCP/IP 协议传输的各种网络，在 TCP/IP 网络及协议上再加载传输 GDB 专有的排错操作用通信协议就能进行远程排错了。

不仅 GDB 有远程模式，KGDB 也同样具有远程模式，KGDB 主要是为运行中的 Linux 核心进行排错，而 GDB 则是主要是用在源代码的层次。运用 KGDB，负责核心程序的程序员可以将核心以近似于应用程序的除错方式来排错，包括为核心代码设置中断点（breakpoint）、让核心程序以步阶方式逐行运行以及观看变量值等。

在某些架构的处理器中，会以硬件方式提供一些排错功能的寄存器，以及可以设置观察点（Watchpoint），观察点的功用是当程序员指定的存储器地址被运行到或访问到时，观察点即会去触发、触动一个中断点。对此 KGDB 可以安装在一部传统机器上，并通过远程模式使用另一部接受排错机器上的硬件排错功效，同样的两部机器可用各种方式进行沟通，如串接式接线、以太网等，尤其在 FreeBSD 操作系统上还允许使用 FireWire 接线，并用直接内存访问（Direct Memory Access，DMA）的功能来协助远程排错。

GDB 运用上最明显的限制是在“用户界面”的部分，默认只有命令行接口（CLI）可用，而不具备较能亲合上手、直觉操作的图形用户界面（GUI），不过 GDB 也已经有几个前端程序为其补充，例如 DDD、GDBtk/Insight 以及 Emacs 中的“GUD 模式”等，有了这些补充后，GDB 在功效使用的便利性上就能够与“集成发展环境中的侦错功效使用”相接近。

另外，有些排错工具（软件）也被设计成能与 GDB 搭配使用，例如存储器泄漏（memory leak）的侦测程序。

在命令行中使用 GDB 的示例如下：

```
gdb prog.out      debug prog.out
(gdb) run         run
```

以下是用 GDB 进行除错的一段过程示例，要进行排错的程序已在堆栈追踪（Stack trace）区内：

```
/* quiet mode*/
$ gdb -q

(gdb) run
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xc11000
This program will demonstrate gdb
```

```
Program received signal SIGSEGV, Segmentation fault.
0x08048428 in function_2 (x=24) at crash.c:22
22      return *y;
(gdb) edit
(gdb) shell gcc crash.c -o crash -gstabs+
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: cannot close "shared object read from target memory": File in wrong format
`/home/sam/programming/crash' has changed; re-reading symbols.
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xa3e000
This program will demonstrate gdb
24
Program exited normally.
(gdb) quit
```

这个程序已处在运行阶段,之后找出这个程序中会导致运行错误的段落,然后将对应处的原代码用编辑器进行错误修订,更正完成后用 GNU 编译器(GCC)重新编译并再次运行。

检查下面的 C 程序代码:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t
foo_len (const char *s)
{
  return strlen (s);
}

int
main (int argc, char *argv[])
{
  const char *a = NULL;

  printf ("size of a = %d\n", foo_len (a));

  exit (0);
}
```

在使用 GCC 进行编译时, 加上-g 标志来输出适当的排错信息, 假设 C 语言示例代码文件名为 example.c, 可以使用下面的语句进行编译:

```
[theqiong@home ~] gcc example.c -g -o example
```

然后在运行示例时会出现的错误如下:

```
[theqiong@home ~] $ ./example
Segmentation fault
```

此时可以使用 GDB 进行排错。

```
[theqiong@home ~] $ gdb ./example
GNU gdb (GDB) Fedora (7.3.50.20110722-13.fc16)
Copyright (C) 2011 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/example...done.
(gdb) run
Starting program: /path/example

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400527 in foo_len (s=0x0) at example.c:8
8         return strlen (s);
(gdb) print s
$1 = 0x0
```

第 13 章

Lexical Structure

13.1 White space

除了在字符串和字符常量中出现以外，空白字符(最常见的是空格、制表符和换行符)用于分隔源文件中的其他记号，否则就被忽略。

空格可以帮助程序员理解代码的结构，习惯上来说，对于程序的每一行的缩排都是为了使控制结构更为清晰。同样的，空行用来分隔程序的不同部分。

13.2 Comments

在 C 语言中，注释(comment)是在“/*”与“*/”之间所有的文字。

注释是用自然语言描述程序的作用，可以占用连续的几行[1]，但一个注释不能嵌套在另一个注释中。

注释是写给人看的，而不是写给计算机的，它们向其他程序员传递该程序的有关信息。当 C 语言编译器将程序转换为可由机器执行的形式时，注释被完全忽略。

每一个程序都应以一个专门的并从整体上描述程序操作过程的注释开始，这段注释称为程序注释(program comment)，它包括程序文件的名称和一些描述程序操作的信息，以及程序的来源等信息。

注释还可以描述程序中特别复杂的部分，指出可能的使用者，对如何改变程序行为提出一些建议等。

当程序越来越复杂时，给出合适的注释是使程序易读的最好方法之一。

13.3 Identifiers

C 程序源代码中的宏、变量、函数、类型等实体的名字被统称为标识符(identifier)。在 C 语言中，构造标识符的规则为：

- 标识符必须以字母或下划线开头；
- 标识符中所有其他字符必须为字母、数字或下划线，不能有空格或其他特殊字符；
- 标识符不能使用下列关键字(keyword)[2]：

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

- 关键字只能用小写字母；
- 标准库的标识符是受限的，而且使用下划线的标识符也是受限的；

[1]从 C99 开始，C 语言也支持以//开头的单行注释(这个特性实际上在 C89 的很多编译器上已经被支持了)

[2]某些编译器把标识符(例如 asm、far 和 near)视为附加关键字。

- 标识符是大小写敏感的;
- 宏一般使用大写字母和下划线命名;
- 标准库函数的名字只能包含小写字母。

标准 C 对标识符的长度没有限制,但 C 语言编译器在判断两个标识符是否相同时可能不会考虑第 31 个字符以后的字符。某些 C 语言实现还可能会对模块间共用的标识符附加一些额外的限制。

> 更复杂的情况时,标准 C 对外链接的标识符有特殊的规定,而大多数函数名都属于这类标识符。因为链接器必须能识别这些名字,而且一些早期的链接器只能处理短名字,所以标准 C 声称只有前 6 个字符才是有意义的。
> 此外,还不区分字母的大小写,所以产生的后果是 ABCDEFG 和 abcdefg 可能会被作为相同的名字处理。
> 大多数编译器和链接器都比标准 C 所要求的要宽松,所以实际使用中这些规则都不会产生问题。

13.4 Constants

源代码可以包含下列通用类型的常量:整型、浮点型、字符型和字符串。每一种类型常量的构成规则如下:

- 整型
  整型常量一般写成一个数字字符串,用以表示一个十进制数。
  - 如果一个整型常量以 0 开头,则编译器会把这个数当作八进制整数;
    例如,常量 040 会被当作一个八进制数,代表的是十进制数 32。
  - 如果一个整型常量以 0x 开头,则编译器会把它当作十六进制整数。
    例如,常量 0xFF 对应的十进制数为 255。

  可以在数字串末尾加字母 L,明确表示该数字是一个长整型常量,因此常量 0L 等于 0,但这个值是长整型的。同样,如果用字母 U 作为后缀,这个常量会被当作是无符号常量。
- 浮点型
  C 语言中的浮点型常量有小数点,而且浮点型常量也可以用科学记数法写成 x.xxxxE±yy 的形式,表示 x.xxxx 乘以 10 的 yy 次方。
- 字符型
  在 C 语言中可以用单引号把字符括起来表示字符型常量,通过编码模式(通常是 ASCII)将每一个字符映射成一个数值。
  除了标准字符外,C 语言还支持下列表示特殊字符的转义序列(escape sequence)[3]:
  - `'\a'` 震铃字符(终端的嘟嘟声)
  - `'\b'` 退格
  - `'\f'` 换页(开始一个新的页面)
  - `'\n'` 换行
  - `'\r'` 回车(返回至本行首位)
  - `'\t'` (横向)制表符
  - `'\v'` 垂直制表
  - `'\\'` 字符\本身
  - `'\' '` 字符'(仅当一个字符时才需要反斜杠)
  - `'\" '` 字符"(仅在字符串中才需要反斜杠)
  - `'\ddd'` ASCII 代码为八进制数 ddd 的字符

[3]转义序列使字符串包含一些特殊字符而又不会使编译器引发问题,这些字符包括非打印的(控制)字符和对编译器有特殊含义的字符(例如")。

- '\xdd' ASCII 代码为十六进制 dd 的字符
- '\0' 空字符(用 0 作为它的字符代码)

- 字符串
  可以用双引号把字符括起来表示字符串常量。和对字符一样，C 语言对字符串也支持同样的转义序列。
  如果在一个程序中连续出现两个或更多的字符串常量，编译器会把它们连接到一起，这样可以把一个长字符串分写到几行上。

13.5 Operators

在 C 语言中，算术运算和逻辑运算用表达式表示。

在表达式中有多个运算符且没有括号进行分隔时，运算的先后顺序由运算符的优先级和结合性规则来决定先用哪个运算符。

- 当两个运算符争用一个项时，首先根据优先级确定运算顺序；
- 如果两个运算符优先级相同，根据运算符的结合性决定先用左边的运算符还是先用右边的运算符。

Table 13.1: C 语言运算符的优先级表

运算符	结合性
()、[]、->、.	左
一元运算符:-、++、--、!、&、*、~、(类型)、sizeof	右
*、/、%	左
+、-	左
<<、>>	左
<、<=、>、>=	左
==、!=	左
&	左
^	左
|	左
&&	左
||	左
?:	右
=、OP=	右
,	左

其中，运算符～、&、!、^、<< 和 >> 可以用来操作一个整数中的单独的位，这种操作需要程序员严格控制内部表示。

- C 语言没有指数运算符，如果要进行非正整数次幂运算，可以调用 pow 函数。
- C 语言规定% 不能用于浮点数，如果要进行浮点数的模运算，可以调用 fmod 函数。

13.6 Punctuation tokes

标点符号可以用来组成运算符和其他 C 语言句法需要的标志。

第 14 章

Data Types

为了使程序能在各种情况下使用，程序必须能够存储多种不同类型的数据。

当程序变得更加复杂时，就会用到以不同构造方法得到的信息集合，这些不同种类的信息统称为数据(data)。

C 语言中的每一个值都与一个数据类型相关联，数据类型是它的值域和一组操作的集合。

可以将 C 语言的数据类型分为两类，分别是：

- 不再包含内部组成部分的原子(atomic)类型；
- 包含其他类型值的复合(composite)类型。

C 语言支持两种根本不同的数值数据类型：整型和浮点型，其中整型的值全都是数，而浮点型的值则可能还有小数部分。另外，整型数据类型又分为两大类：有符号的和无符号的(unsigned)。

整数通常以 16 位或 32 位方式存储。在有符号数中，如果数为正数或零，那么最左边的位(符号位)为 0，如果是负数，则符号为 1。

在 C 语言中，最大的 16 位整数的二进制表示形式是 0111111111111111，对应的十进制值是 32 767(即 $2^{15}-1$)，而最大的 32 位整数是 01111111111111111111111111111111，对应的十进制值是 2 147 483 647(即 $2^{31}-1$)。

把不带符号位(把最左边的位看成是数值部分)的整数称为无符号数，那么最大的 16 位无符号整数是 65 535(即 $2^{16}-1$)，而最大的 32 位无符号整数是 4 294 967 295(即 $2^{32}-1$)。

14.1 Integer Type

数值类型决定了变量所能存储的最大值和最小值，同时也决定了是否允许在小数点后出现数字。

int 型变量可以存储整数，只是整数的取值范围是受计算机系统限制的，在某些计算机系统中不能超过 32 767。

实际上，C 语言的整型有不同的尺寸，int 类型是计算机给出的整数的“正常”尺寸(通常为 16 位或 32 位)。

- long int

  16 位整型数的上限值是 32 767，这会对许多应用产生限制，所以 C 语言还提供了长整型(long int)。

- short int

  某些时候，为了节省空间，我们会指示编译器存储比正常尺寸小的数，称为短整型(short int)。

默认情况下，C 语言中的整型变量都是有符号的，也就是说最左边的位保留为符号位。为了通知编译器变量没有符号位，需要把它声明成 unsigned 类型，无符号数主要用于系统编程和低级的、与机器相关的应用。

为了构造的整型数正好满足需要，可以指明变量是 long 型还是 short 型，signed 型还是 unsigned 型，或者把说明符组合起来(例如 long unsigned int)。

实际上，C 语言中只有下列 6 种组合可以产生不同的类型。

```
short int
unsigned short int
int
unsigned int
```

```
long int
unsigned long int
```

其他组合都是上述 6 种类型的同义词。例如，除非额外说明，否则所有整数都是有符号的，因此 long signed int 和 long int 是一样的类型。

另外，说明符的顺序没有要求，所以 unsigned short int 和 short unsigned int 是一样的。

C 语言允许通过省略单词 int 来缩写整型的名字。例如，unsigned short int 可以缩写为 unsigned short，而 long int 可以缩写为 long。

6 种整型的每一种所表示的取值范围都会根据机器的不同而不同，但是有两条所有编译器都必须遵守的原则。

- 首先，C 标准要求 short int、int 和 long int 中的每一种类型都要覆盖一个确定的最小取值范围。
- 其次，C 标准要求 int 类型不能比 short int 类型短，而 long int 类型不能比 int 类型短。

short int 类型的取值范围有可能和 int 类型的范围是一样的，而 int 类型的取值范围也可以和 long int 的一样。

下面说明了通常情况下在 16 位计算机上整型的取值范围，其中 short int 就和 int 有相同的取值范围。

Table 14.1: 16 位机的整型数据类型

类型	最小值	最大值
short int	-32 768	32 767
unsigned short int	0	65 535
int	-32 768	32 767
unsigned int	0	65 535
long int	-2 147 483 648	2 147 483 647
unsigned long int	0	4 294 967 295

下面说明了通常情况下在 32 位计算机上整型的取值范围，其中 int 就和 long int 有相同的取值范围。在标准库 <limits.h> 中可以找到定义了每种整型最大值和最小值的宏。

Table 14.2: 32 位机的整型数据类型

类型	最小值	最大值
short int	-32 768	32 767
unsigned short int	0	65 535
int	-2 147 483 648	2 147 483 647
unsigned int	0	4 294 967 295
long int	-2 147 483 648	2 147 483 647
unsigned long int	0	4 294 967 295

注意，short 和 long 整型在 16 位机和 32 位机上都有着各自相同的取值范围，这也引出里第一条可移植性技巧。

> 为了最大限度保证可移植性，对不超过 32767 的整数采用 int(或 short int)类型，而对其他的整数采用 long int 类型。

需要提醒的是，应该避免不分差别地使用长整型，因为在长整型上地操作需要地时间可能会多过在较小地整数上的。

14.2 Floating-point Type

整型数并不适用于所有应用,如果需要带小数点的数,或者极大数或极小数,此时就需要使用浮点(小数点是“浮动”的)格式存储的数。

在大多数程序设计语言中,包含小数部分的数值称为浮点数(floating-point number)。

- 浮点数所能表示的往往只是实际数值的一个近似值,因此使用浮点数时实际上是近似地表示数学中的实数。
- 浮点数类型变量可以存储比整型变量更大的数,而且还可以存储带小数位的数据。
- 浮点数类型也有一些缺陷,即它们需要的存储空间要大于整型类型变量,而且进行算术运算时浮点数类型变量要比整型变量慢。

使用浮点数时,这些数的值通常分成两部分存储:小数 (fraction) 部分 (或称为尾数 (mantissa) 部分)和指数(exponent)部分,这其中的小数点是“浮动的”,例如 12.0 这个数是以 1.5×2^3 的形式存储的,其中 1.5 是尾数部分,而 3 是指数部分。

C 语言提供 3 种浮点型数,它们对应不同的浮点格式。

- float:单精度浮点数
- double:双精度浮点数
- long double:扩展双精度浮点数

其中,最常用的浮点数类型是 double 型,它是双精度浮点数的缩写,它的精度是浮点型 float 的两倍。C 语言开发早期普遍采用 float 类型,现在则多用 double 来表示所有的浮点数。要在程序中存储浮点型数据,就必须尽量声明为 double 类型。

具体采用哪种类型依赖于程序对精度(和数量级)的要求。

- 当精度要求不严格时,float 型是很适合的类型。
- double 提供更高的精度,足够适用于绝大多数程序。
- long double 支持极高精度的要求,很少会用到。

C 语言标准没有说明 float、double 和 long double 类型提供的具体精度,原因在于不同的计算平台可以用不同方法存储浮点数。

大多数现代计算机都遵循 IEEE 754 标准的规范,该标准提供了两种主要的浮点数格式:单精度(32 位)和双精度(64 位)。数值以科学记数法形式存储,每一个数都是由 3 部分组成,分别是符号、指数和小数。

- 为指数部分保留的位数说明了数可能大(或小)的程度。
- 小数部分的位数说明了精度。

在单精度格式种,指数长度为 8 位,而小数部分占了 23 位,因此单精度数可以表示的最大值大约是 3.40×10^{38},其中精度是 6 个十进制数字。

Table 14.3: 浮点型的特征(IEEE 标准)

类型	最小正值	最大值	精度
float	1.17×10^{-38}	3.40×10^{38}	6 个数字
double	2.22×10^{-308}	1.79×10^{308}	15 个数字

IEEE 标准也描述另外两种格式——单扩展精度和双扩展精度,但没有说明这些格式中的位数,只是要求单扩展精度类型至少为 43 位,而双扩展精度类型至少要为 79 位。

long double 类型的长度随着机器的不同而变化,其最常通用的尺寸是 80 位和 128 位。在不遵循 IEEE 标准的计算机上,float 可以有和 double 相同的数值集合,或者 double 可以有和 long double 相同的数值集合,因此上述 IEEE 浮点数标准可能无效。

可以在 <float.h> 中找到定义浮点型特征的宏。

要编写一个使用浮点型数值的完整的程序，必须先读取并显示 double 型数值，在这里也可以使用扩展库 simpio 完成浮点数的读取。要读取浮点数，需要调用 GetReal 函数，不同的仅仅在于它返回的是一个 double 型的数值。在屏幕上显示浮点数也可以使用 printf 函数，但这时要使用另一个格式码，浮点数有多种格式代码，简单的是用%g，它代表一般的浮点数格式。

```
/*
* File:add2f.c
* ------------
* This program reads in two floating-point numbers, adds them together,
* and prints their sum.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  double n1,n2,total;

  printf( "This program adds two floating-point numbers.\n" );
  printf( "1st number?" );
  n1 = GetReal();
  printf( "2nd number?" );
  n2 = GetReal();
  total = n1 + n2;
  printf( "The total is %g\n" ,total);
}
```

14.3 String Type

文本数据的最基本的元素是字符，但只有字符组合起来，形成一个连续的单元时才会更有用，而且现代计算机处理的大多是文本而非数字。

在程序设计中，一连串字符称为字符串(string)，它使得在屏幕上显示消息成为可能，而且重要的是，字符串也是数据，因此可以用操作和存储数值数据的方法来处理字符串，更进一步考虑，字符串会是一个复杂的数据类型。

要使用字符串数据就必须为其数据类型命名，而尽管 C 语言设计者在相关库中提供了一些对字符串的操作，但他们并没有明确地定义 string 类型。不过在扩展库 genlib 中可以自己定义这种类型。库中定义的数据类型将成为数据类型表的一部分，可以作为内置类型来使用。因此，尽管实际上字符串类型是通过 genlib 定义的，但在设计程序时，还是应将其看作构成 C 语言所必需的一部分。

simpio 库中包含一个 GetLine 函数，可读取一整行数据并将其作为 string 返回。声明 string 类型变量的方法和声明其他简单数据类型的方法一样，也可以用 printf 函数将其显示在屏幕上，就像处理数值数据一样，唯一的区别在于用格式码%s 代替表示数值数据的%d 或%g。

无论何时使用数据，无论使用的是什么类型的数据，C 编译器都应该了解其数据类型。从整体上讲，一个数据类型(data type)可以由两个性质定义：值的集合，即值域(domain)以及操作的集合。

对数据类型而言，值域就是作为该类型的元素的值的集合，例如，整型数据的值域就是机器硬件所能构造的所有整数，字符型数据的值域就是键盘上出现的或可以显示在终端屏幕上的符号的集合。

```
/*
* File:greeting.c
* ----------------
* This program prints a more personal greeting than did
* the original "Hello,world." program by reading in the
* name of the user.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  string user;

  printf("What is your name?");
  user = GetLine();
  printf("Hello,%s.\n",user);
}
```

操作集由操作该类型数据的值的工具构成,也就是它们可以进行的运算的集合。在进行操作时,操作必须与值域的元素相对应,从而值域和操作这两个部分定义了数据类型。

14.4 Wide Characters

第 15 章

Format I/O

scanf 函数是与 printf 函数相对应的，其中的字母 f 含义相同，都是表示“格式化”，用于格式化的读和写。

scanf 函数和 printf 函数都需要使用格式串（format string）来说明输入或输出数据的样式，其中 scanf 函数需要知道将获得的输入数据的格式，而 printf 函数需要直到输出数据的显示格式。

```
printf("format-string", expression);
```

15.1 printf

printf 函数用于显示格式串（format string）的内容，并且在字符串指定位置插入可能的值。

- 调用 printf 函数时必须提供格式串，接着是用来在打印时插入字符串中的任意值。
- 显示的值可以是常量、变量或者是表达式等。
- 单独调用一次 printf 函数时可以打印的值的个数没有限制。

例如，使用 printf 显示变量的值时，可以使用如下的语句：

```
printf("%d", variable);
```

格式串包含普通字符和转换说明（conversion specification），其中转换说明以字符% 开头。

- 转换说明是用来表示打印过程中填充了值的占位符；
- 跟随在字符% 后的信息指定了把数值从内部（二进制）转换成打印（字符）的形式的方法。
- 格式串中的普通字符被简单复制到输出行中。

这里，占位符用来指明在打印过程中变量 variable 的值的显示位置和转换说明。例如，转换说明%d 指定 printf 函数把 int 型数值从二进制形式转换成十进制数字组成的字符串，如果要打印浮点型变量的值，则需要用%f 来代替%d。

默认情况下，%f 会显示出小数点后的 6 位数字，若需要%f 显示小数点后 n 位数字，可以把.n 放置在% 和 f 之间，例如：

```
printf("%.nf", variable);
```

C 语言没有限制 printf 函数可以显示的变量的数量，而且还可以显示任意表达式的值，从而可以简化程序。

```
volume = length * width * height;
printf("%d\n",volume);
printf("%d\n", length * width * height);
```

> printf 能显示表达式的能力说明了 C 语言的一个通用原则：在任何需要数值的地方，都可以使用具有相同类型的表达式。

- C 语言编译器不会检测格式串中转换说明的数量是否和输出项的数量相匹配。
- C 语言编译器也不会检测转换说明是否适合要显示的项的数据类型。如果使用了不正确的转换说明，将只会简单地产生无意义的输出。

格式码可以有%m.pX 和%-m.pX 两种形式，其中 m 和 p 都是可选的整型常量。

- 如果省略 p，那么分隔 m 和 p 的小数点也会被忽略；
- m（minimum field width）指定了要显示的最小字符数量；
- p（precision）依赖于转换说明符 X（conversion specifier）的选择；
- X 表明在显示数值前需要对其进行的转换。

当转义序列出现在 printf 函数的格式串中时，它们表示在显示中执行的操作。例如：

- 输出\a 会产生报警声；
- 输出\b 会使光标从当前位置回退一个位置；
- 输出\n 会使光标跳转到下一行的起始位置；
- 输出\t 会把光标移动到下一个制表符停止的位置；
- 输出\" 可以标记字符串的开始与结束。

15.2 scanf

scanf 函数根据特定的格式读取输入，而且 scanf 函数的格式串也可以包含普通字符和转换说明两部分。例如，为了读入一个 int 型数值，可以使用如下的 scanf 函数调用：

```
scanf("%d", &i); /* reads an integer; stores into i */
```

scanf 函数中转换说明的用法和 printf 函数在本质上是一样的。这里，字符串%d 说明 scanf 函数读入的是一个整数，而假设 i 是一个 int 型变量，用于存储 scanf 读入的值。这里，& 通常用于 scanf 函数但不是必需的。

同样地，读入一个 float 型数值时，需要类似的 scanf 调用语句：

```
scanf("%f", &x); /* reads a float value; stores into x */
```

其中，字符串%f 告知 scanf 函数去读入一个浮点数，而假设 x 是一个浮点型变量。浮点数可以含有小数点，但不是必须含有。

C 编译器无法检查出可能的匹配不当，导致 printf 和 scanf 函数都有一些不易察觉的陷阱，因此在使用 scanf 函数时必须检查转换说明的数量是否与输入变量的数量相匹配，并且检查每个转换是否符合相对应的变量。

调用 scanf 函数是读数据的一种有效但不理想的办法，用户输入了未预期的数据可能会导致程序将无法运行。一些 C 编译器可以检查出这种错误，但通常不是所有的时候都可以。

通常把 & 符号放在 scanf 函数调用的每个变量之前，如果在调用 scanf 函数时忘记使用 & 将无法把输入的值存储到变量中，此时变量将只能继续保留其原来的值。在 C 语言中，如果没有给变量赋初值，则变量原来的值可能是没有意义的随机值，丢失符号 & 的变量没有被赋值可能会得到类似“Possible use of ‘i’ before definition”的警告。

scanf 函数本质上是一种“模式匹配”函数，也就是试图把输入的字符组与转换说明匹配成组。调用 scanf 函数时，它会从左边开始处理字符串中的信息。对于格式串中的每一个转换说明，scanf 函数努力从输入的数据中定位适当类型的项，并且跳过必要的空格。接下来，scanf 函数读入数据项并在遇到不可能属于此项的字符时停止。

- 如果读入数据项成功，那么 scanf 函数会继续处理格式串的剩余部分。
- 如果任何项都不能成功读入，那么 scanf 函数将不再查看格式串的剩余部分（或余下的输入数据）而立即返回。

在寻找输入数据的起始位置时，scanf 函数会忽略空白（white-space）字符（空格符、横向和纵向制表符、换页符和换行符），从而可以把数字放在单独一行或者分散在几行内输入。

scanf 函数“忽略”[1]了最后的换行符，没有真正的读取它，在下一次调用 scanf 函数时才会读取这个换行符。

在识别整数和浮点数方面，scanf 函数的原则如下：

- 在要求读入整数时，scanf 函数首先找到一个数字、正号或负号，接着继续读取数字直到读到一个非数字时才停止。
- 在要求读入浮点数时，scanf 函数会寻找一个正号或负号(可选的)，随后是一串数字(可能含有小数点)，再后面是一个指数(可选的)。
- 指数由字母 e(或者字母 E)、可选的符号和一个或多个数字组成，而且转换说明%e、%f 和%g 可以互换并遵循相同的原则来识别浮点数。

在 printf 函数中使用转换说明%i 和%d 时，二者没有区别，但在 scanf 函数中%d 只能与十进制数形式的整数匹配，而%i 则可以匹配八进制、十进制或十六进制形式的整数，因此在 scanf 函数中建议使用%d 来规避%i 陷阱。

- 如果输入的数有前缀 0，那么%i 将导致 scanf 函数以八进制数来处理输入值；
- 如果输入的数有前缀 0x 或 0X，那么%i 将导致 scanf 函数以十六进制数来处理输入值；

在实际应用时，需要测试 scanf 函数是否成功读入了要求的数据(若不成功，还可以试图恢复[2])，或者避免使用 scanf 函数，而是采用字符格式读取所有数据后再把它们转换成数值形式。

- 当 scanf 函数遇到的字符不是当前项的内容时，会把该字符“放回原处”。
- 当 scanf 函数扫描下一个输入项或在下一次调用 scanf 函数时才会再次读入该字符。
- scanf 函数会把格式串中每个转换说明与一个输入项进行匹配，但仍然不会读入换行符，因此换行符将留给下一次 scanf 函数调用。

在处理格式串中的普通字符时，scanf 函数采取的动作依赖于这个字符是否为空白字符。

- 空白字符

  当在格式串中遇到一个或多个连续的空白字符时，scanf 函数从输入中重复读取空白字符直到遇到一个非空白字符(把该字符“放回原处”)为止。格式串中空白字符的数量无关紧要，每一个字符都可以和输入数据中任意数量的空白字符相匹配。

  或者说，在格式串中所有的空白字符都是等价的，一个空格字符或者空白转义序列都可以匹配任意数量的空格、换行符或其他空白字符。
- 其他字符

  当在格式串中遇到一个非空白字符时，scanf 函数将把它与下一个输入字符进行比较。如果二者匹配，那么 scanf 函数会放弃输入字符并继续处理格式串。如果二者不匹配，scanf 函数会把不匹配的字符放回输入中并异常退出，而不会进一步处理格式串或者从输入中读取字符。

尽管 scanf 函数与 printf 函数有相似的格式串和转换说明，但二者之间也有着明显的差异。

1. 在寻找数据项时 scanf 函数通常会跳过空白字符，因而除了转换说明外，格式串通常不需要包含字符。
2. 格式串的不正确使用将导致 scanf 函数行为异常。
3. scanf 函数格式串中的换行符等价于空格，它们都会导致 scanf 函数提前进入到下一个非空白的字符。例如，格式串%d\n 将导致 scanf 函数在读入一个整数后调转到下一个非空白字符处，这可能导致交互式程序一直“挂起”直到用户输入一个非空白字符为止。

[1]用户在使用键盘输入时，程序并没有读取输入，而是把用户的输入放入隐藏的缓冲区中交由 scanf 函数来读取。为了后续读取，scanf 函数把字符放回缓冲区中是非常容易的。

[2]在调用 scanf 函数失败时，可以终止或者尝试恢复程序，可能的恢复方法包括丢掉有问题的输入和要求用户重新输入。

第 16 章

Expressions

在希望程序执行计算时，应写出一个表达式以指定必要的操作，其形式类似于数学表达式。

C 语言强调使用表达式(expression)来将独立的项用运算符连接起来，其中每一个项(term)表示一个单独的数据值，运算符(operator)是一个表示运算的字符或一个字符序列。

在表达式中，项必须是如下选择之一：

1. 常量
   程序文本中出现的任何明确的数值都称为常量(constant)。
2. 变量
   变量是数据的存放处，其内容在程序执行期间可以改变。
3. 函数调用
   值经常通过调用其它函数来产生，这些函数可以是库函数，这些函数返回数据集给最初的表达式。
4. 括号中的表达式
   可以用括号来控制项的分组并影响操作的优先顺序，如同在数学表达式中使用括号一样。对于编译器来说，在计算开始前，括号中的表达式成为一个必须当作单元来处理的项。

在程序运行时，执行表达式中每一个特定操作的过程称为求值(evaluation)。当对表达式进行求值时，每一个运算符都对两边项所表示的数据值进行计算。当所有运算符求值完毕后，剩下的一个数据就是运算结果。例如，给出表达式 n1 + n2，求值过程包括读取变量 n1 和 n2 的值并将二者相加，求值结果就是两者的和。

```
n1+n2
```

C 语言支持表达式语句 (expression statement)，即一种允许把任何表达式都当作语句来使用的特性。换句话说，不论任何类型或计算结果，任何表达式都可以通过添加分号的方式转换成语句。

某些编译器可能会检查出无意义的表达式语句，并显示类似“Code has no effect.”的警告。

16.1 Constants

C 语言的表达式就是表示如何计算值的公式，最简单的表达式是常量和变量，其中常量(constant)是在程序运行过程保持不变的量。

常量是在程序中以文本形式显示的数据，而不是读、写或计算出来的。在 C 语言程序中，常量又可以作为表达式的组成部分，因而给出每一个基本数据类型的常量值就变得很重要。

C 语言允许使用十进制[1]、八进制和十六进制形式书写整型常量。

- 十进制常量包含数字 0 ～ 9，但是一定不能以零开头。

```
15  255  32767
```

- 八进制常量只包含数字 0 ～ 7，而且必须要以零开头。

[1]八进制、十六进制只是书写数的另一种方式，它们不会对数实际存储的方式产生影响。实际上，整数都是以二进制形式存储的，不考虑实际书写的方式，因此在任何时候都可以从一种书写方式切换到另一种书写方式或者几种书写方式混用。例如，10 + 015 + 0x22 的值为十进制数 55。八进制和十六进制更适合应用在低级程序的编写上。

```
017 0377 07777777
```

- 十六进制常量只包含数字 0 ～ 9 和字符 a ～ f，并且总是以 0x 开头。

```
0xf 0xff 0x9fff
```

十六进制常量中的字母既可以是大写也可以是小写。

```
0xff 0xFf 0xfF 0xFF 0Xff 0XfF 0XFf 0XFF
```

一般情况下，可以采用宏定义（macro definition）的特性定义常量，语法如下：

```
#define constant_name value;
```

此外，还可以使用宏来定义表达式：

```
#define constant_name expression;
```

在 C 语言程序编译过程中，预处理器会把每一个宏用其表示的值或表达式替换。
另外，在使用宏定义时要注意 C 语言的隐性约定，比如：

```
#define frac1 (5 / 9)
#define frac2 (5.0 / 9.0)
```

如果两个整数相除，则 C 语言会对结果采用取整操作，因此 (5 / 9) 的结果为 0，导致宏定义无意义。

16.1.1 Integer Constants

要将一个整型常量作为程序的一部分或作为输入数据，只要写出组成该常量的每一个数字即可。若该数是负数，那么在该数的前面加负号。

注意，不能使用逗号分隔长数据，因此 100 万要写成 1000000，而不能写成 1,000,000。

16.1.2 Float-point Constants

C 语言中的浮点型常量必须带有小数点或指数。若程序中出现 2.0，则该数在系统内部作为浮点数处理；若该数写成 2，则作为整型数处理。

浮点常量的指数指明以 10 为底的幂，而且需要在指数数值前放置字母 E（或 e），可选项 + 或-可以出现在字母 E（或 e）的后面。

用科学记数法的方法表示浮点值时，需要先将浮点数写成标准表示形式，后跟一个字母 E 以及一个整数指数，该整数指数可以有正号或负号。例如，光速（m/s）近似于 2.9979×10^8，那么在 C 语言中可以写成 2.9979E+8，其中 E 代表乘以 10 的指数次幂。

默认情况下，浮点常量都以双精度数的形式存储。换句话说，当 C 语言编译器会以 double 型变量的格式将在程序中读取到的浮点常量存储在内存中。

当程序中出现整型常量时，如果它属于 int 类型的取值范围，那么编译器会把此常量作为普通整数来处理，否则会作为长整型数来处理。

通常情况下，上述规则不会引发任何问题，因为在需要时 double 类型的值可以自动转换为 float 类型值。

在某些极个别情况下，可能会需要强制编译器以 float 或 long double 格式存储浮点常量。

- 为了强制编译器把常量作为单精度浮点数，只需在常量的末尾加上字母 F（或 f）

```
122.00F 5.0f
```

- 为了强制编译器把常量作为长整型数来处理，只需在其后面加上一个字母 L（或 l）。

```
15L 0377L 0x77ffL
```

- 为了指明无符号常量，可以在常量后面加上字母 U（或 u）

```
15U 0377U 0x77ffU
```

- 为了表示常量是无符号的长整型变量，可以组合使用字母 L 和 U，而且字母 L 和 U 的顺序和大小写都不影响。

```
0xffffffffUL
```

由于历史的原因，C 语言更倾向于使用 double 类型来存储浮点常量。在 K& R C 中认为在大型数组中使用 float 类型的主要原因是节省存储空间，或者有时是为了节省时间，因为在一些机器上双精度计算花销格外大，所以要求所有浮点计算都采用双精度的格式，但标准 C 没有这样的要求。

16.1.3 String Constants

在 C 语言中，将组成字符串的字符括在双引号里来表示字符串常量。字符串常量中可以包括字母、空格、标点符号及特殊的换行符等。

双引号并不是字符串的组成部分，仅用于标记字符串的开始和结尾。

16.2 Variables

大多数 C 程序在产生输出之前往往需要执行一系列的计算，因此需要在程序执行过程中有一种临时存储数据的办法。

> 变量是用来存储程序执行过程中可能会发生改变的数据的。
> 常量是程序执行过程中不会发生改变的数据，用户可以对其进行命名。

C 程序中的大多数数据都存储在变量中，可以将变量理解为(variable)是一个值的存放处。

变量有三个重要属性：名称、值和类型。为了理解三个属性之间的关系，可以将变量看作一个外面贴有标签的盒子。

- 变量的名字写在标签上，以区分不同的盒子。若有几个盒子(即变量)，可以通过名称来指定其中之一。
- 变量的值对应于盒子内装的东西，盒子标签上的名称从不改变，但可以经常从盒子中取出值或放入新值。
- 变量类型表明什么类型的数据可以存储在盒子中。例如，若指定一个盒子存放 int 型的值，则不能将 string 类型的值装入那个盒子。

C 语言中变量名称的构成遵循以下规则：

1. 名称必须以字母或下划线(_)字符开头。在 C 语言中，变量名中出现的大写和小写字母被视为不同的字符。
2. 名称中的其他字符必须是字母、数字或下划线，不得使用空格和其他特殊符号。
3. 名称不可以是以下关键字(keyword)之一，它们在 C 语言中有特殊用途。

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

4. 变量名可以为任意长度，但 C 编译器只认为前 31 个字符有意义[2]，因此，若两个变量名的前 31 个字符相同，则一些编译器将不能区分 31 位以后的字符的不同之处。为了防止这种混淆，最好避免使用多于 31 个字符的变量名。

[2]对于由各自独立的程序文件所共享的变量名，其有意义的字符更少。为安全起见，最好是确保文件共享的变量名的前 6 个字符不同。

5. 变量名应使阅读者易于明白其中存储的值是什么。人名、助词等虽然符合其他几条规则，但对于提高程序的可读性并无帮助。

16.2.1 Variable Declaration

在 C 程序中使用每一个变量之前必须先明确地指定每个变量的名称和类型，这个过程称为变量的声明（declaration）。

```
type name;
```

变量声明本身包括变量类型名，后面是一系列作为该类型实例的变量名，其中：

- type 指明了变量的数据类型；
- name 是一个指定变量名字的标识符；
- 同一类型的变量可以在一条语句中声明，变量名之间以逗号隔开。

变量声明同样也指定了变量的生命周期和使用范围。

- 在函数体内声明的变量称为相对于函数的局部。
- 声明在函数体外的变量称为外部变量（external variable）。

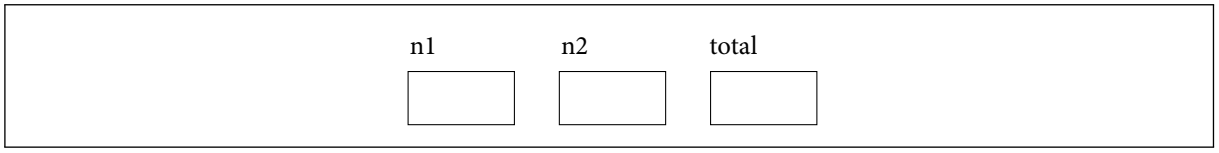
变量通常在函数的开始部分声明，而且在任何函数中声明变量都是合法的。通过变量声明创建了变量的名字和类型，并在它的名字和它可以保存的值的类型之间建立了联系。

例如，以下语句：

```
int n1,n2,total;
```

声明了三个变量，这些变量的名称是 n1、n2 和 total，它们都是 int 类型。

这里使用盒子比喻法可以得到该声明语句的效果就是创建了以下三个盒子，它们的名字分别是 n1、n2 和 total。

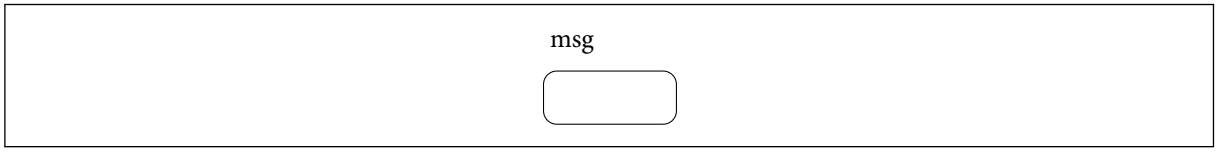


每个变量的初始值都是未定义的，所以在程序开始时，不应对三个盒子中的值作出任何假设。它们可能是随机的值，只有向它们输入值之后才能知道里面的值是什么。

若要声明不同类型的变量，可以在函数开头使用其他的声明语句。例如，若要声明 string 类型的变量 msg，可使用以下语句：

```
string msg;
```

用盒子的比喻来说，该变量声明创建了一个新的名为 msg 的盒子，如下所示：



这里使用了另一种形状的盒子以强调变量 msg 的类型有别于 n1、n2 和 total，其中变量 n1、n2 和 total 为 int 型，只能存放整型数据；变量 msg 为 string 型，只能存放字符串数据。

变量的类型会影响变量的存储方式以及允许对变量采取的操作，试图将错误的数据类型存放在这些变量中，就相当于将方的物体放入圆洞中，编译器会认为这是一种错误。

对于数组和指针变量,在声明句法中要指出指定的名字符合哪个类型。

16.2.2 Local Variables

大多数变量在组成函数体的语句块(block)的开始处声明,这样声明的变量就是局部变量(local variable)。

- 局部变量在它出现的函数体内有效,其他函数不能直接使用这些变量。
- 局部变量的生命周期是该函数被激活的时期。

默认情况下,局部变量具有下列性质。

- 自动执行期限
  变量的存储期限(storage duration)(或存储长度)是在变量存储有效期内程序执行的部分。
  调用闭合函数时"自动"分配局部变量的存储单元,函数返回时销毁分配,所以称局部变量具有自动的存储期限。
  在闭合函数返回时,局部变量并不保留值,因此当再次调用函数时,无法保证变量始终保留原有的值。
- 程序块作用域
  变量的作用域是可以引用变量的程序块,局部变量拥有程序块作用域,从变量声明的点开始一直到闭合函数体的末尾。
  局部变量的作用域不能延伸到其所属函数之外,所以其他函数可以使用同名变量。

当函数被调用时,在分配给该函数的栈帧内为它的每一个局部变量分配空间。当该函数返回时,所有局部变量都消失。

在局部变量声明中增加 static 关键字可以使变量从自动存储期限变为静态存储期限,静态局部变量始终有程序块作用域。

静态局部变量对其他函数是不可见的,这样可以隐藏来自其他函数的数据,但是它会为将来同一个函数的调用保留这些数据。

当函数是递归函数时,每次调用它时都会产生其自动变量的新副本,但是静态变量不会发生这样的情况,这样递归结构中所有的函数调用都共享同一个静态变量。

形式参数拥有和局部变量相同的性质,即自动存储期限和程序块作用域。事实上,形式参数和局部变量唯一真正的区别是,在每次函数调用时对形式参数自动进行初始化,并通过赋值将实际参数的值传递给形式参数。

16.2.3 Global Variables

如果一个变量声明出现在任何函数定义外,这样的变量声明是全局变量(global variable)。

- 全局变量在声明这个变量的模块里定义位置之后的所有部分都有效;
- 全局变量的生命周期是整个程序的执行周期。
- 全局变量能够保存那些在当前函数返回后仍然需要的值。

传递参数是给函数传送信息的一种方法,函数还可以通过外部变量进行交互,这些外部变量就是声明在任何函数体外的全局变量。

只有在必要时才会使用全局变量,建议所有全局变量都用关键字 static 标记,以保证它们局限于所在的模块。

全局变量的性质不同于局部变量。

- 静态存储期限
  全局变量拥有静态存储期限,存储在全局变量中的值将永久保留下来。
- 文件作用域

全局变量的文件作用域从声明的点开始直到闭合文件的结尾，这样跟随在全局变量声明后的所有函数都可以访问它。

静态变量拥有永久的存储单元，所以函数返回时变量的值不会丢失，这样在整个程序执行期间都会保留变量的值。

在多个函数必须共享一个变量时或者少数几个函数共享大量变量时，全局变量是很有用的。然而，在大多数情况下通过形式参数进行函数交互比通过共享变量的方法更好。

- 在程序修改期间，如果改变全局变量，那么将需要检查同一模块中的每个函数以确认该变化对函数的影响程序。
- 如果全局变量被赋予了错误的值，那么它可能很难确定有错误值的函数。
- 依赖外部变量的函数不是“孤立”的，因此很难在其他程序中复用依赖于外部变量的函数，这导致在其他程序中复用函数时将不得不带上任何该函数需要的外部变量。

一个普遍的弊端是，过于依赖全局变量的弊端将不得不在不同的函数中为不同的目的而使用同样的全局变量。

使用全局变量时，要确保它们都拥有有意义的名称，而局部变量就不需要这样。下面的示例演示了把应该是局部变量的变量变为全局变量时可能导致的错误。

```
int i;
void print_row(void)
{
  for(i = 1; i <= 10; i++)
    printf("*");
}

void print_matrix(void)
{
  for(i = 1; i <= 10; i++)
    print_row();
    printf("\n");
}
```

这个示例程序的错误是 print_matrix 函数无法显示 10 行 *，实际上只能显示出 1 行 *。第一次调用 print_row 函数后返回时，i 的值将为 11，接下来 print_matrix 函数中的 for 语句对变量 i 进行自增并把它与 10 进行比较，这导致循环终止且 print_matrix 函数返回。

在下面的随机数示例程序中，用户尝试用尽可能少的次数猜出程序产生的 0 ～ 100 的随机数。

这个随机数示例程序可以分解为下面的子任务。

- 初始化随机数生成器
- 选择随机数
- 用户交互并判断

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

int secret_number;

void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

main()
```

```
{
  char command;

  printf("Guess the secret number between 1 and %d.\n\n",
       MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');
  return 0;
}

/*************************************************************
 * initialize_number_generator: Initializes the random *
 *                      number generator using *
 *                      the time of day.    *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}
/*************************************************************
 * choose_new_secret_number: Randomly selects a number *
 *                    between 1 and MAX_NUMBER and *
 *                    stores it in secret_number. *
 *************************************************************/
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}

/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells *
 *           the user whether each guess is too low, *
 *           too high, or correct. When the guess is *
 *           correct, prints the total number of *
 *           guesses and returns.          *
 *************************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
```

```
  }
}
```

对于随机数的生成，示例程序与 time、srand 和 rand 这三个函数相关，这里将限制 rand 函数的返回值落在 1 ～ MAX_NUMBER 范围内。

当前版本的示例程序依赖于全局变量，把变量 secret_number 外部化以便 choose_new_secret_number 函数和 read_guesses 函数都可以访问它。如果对 choose_new_secret_number 函数和 read_guesses 函数进行改进，应该把变量 secret_number 移入 main 函数中，并使 choose_new_secret_number 函数返回新值，以及重写 read_guesses 函数以便变量 secret_number 可以作为参数传递给它。

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

main()
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
       MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');
  return 0;
}

/*************************************************************
 * initialize_number_generator: Initializes the random *
 *                       number generator using *
 *                       the time of day.    *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/*************************************************************
 * new_secret_number: Returns a randomly chosen number *
 *               between 1 and MAX_NUMBER.  *
 *************************************************************/

int new_secret_number(void)
{
```

```
  return rand() % MAX_NUMBER + 1;
}

/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells *
 *           the user whether each guess is too low, *
 *           too high, or correct. When the guess is *
 *           correct, prints the total number of *
 *           guesses and returns.          *
 *************************************************************/
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

16.3 Scope

C 语言允许包含声明的复合语句，称为程序块(block)。

```
{
  declarations; /* optional */
  ...

  statements;
  ...
}
```

局部变量的作用域是从声明处开始的，下面是一个程序块的示例。

```
if(i > j){
  int temp;

  temp = i; /* swaps values of i and j */
  i = j;
  j = temp;
}
```

默认情况下，声明在程序块中的变量的存储期限是自动的：进入程序块时为存储变量分配单元，而在退出程序块时销毁分配。

局部变量具有程序块作用域，也就是说，不能在程序块外部引用。实际开发程序时，如果在需要临时使用的变量，可以在函数体内的程序块中声明和使用临时变量，这样做有两个好处。

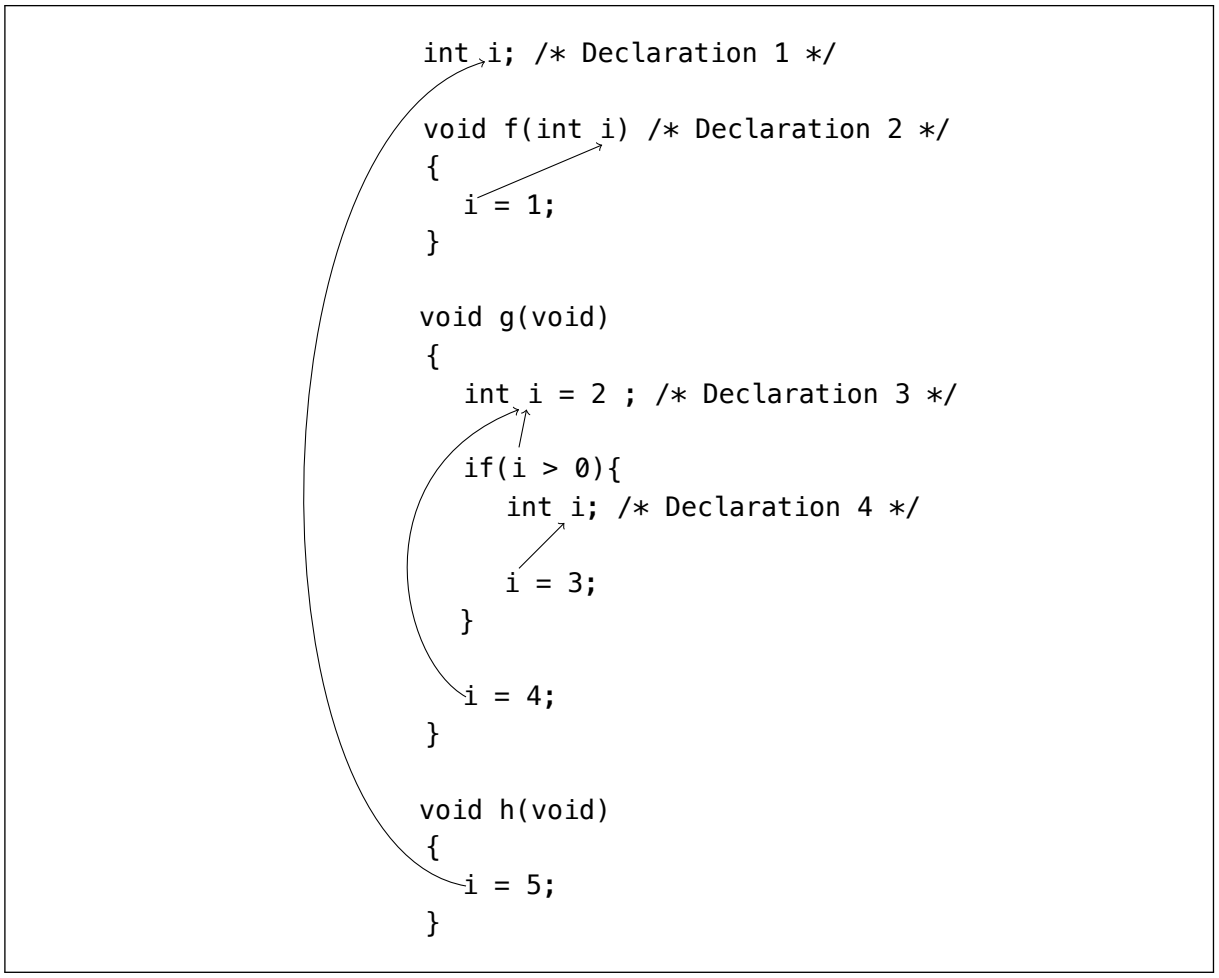
- 避免函数体起始位置的变量声明与临时变量混淆；
- 减少名字冲突。

在上述的示例中，在程序块中声明的临时变量严格局限于程序块，这里临时变量 temp 可以用来交换 i 和 j 的值，而且 temp 可以根据不同的目的用于同一函数中的其他地方。

C 语言的作用域规则可以使相同的标识符具有不同的含义，并使得程序员（和编译器）明确程序中给定点的相关意义。

下面是最重要的作用域规则：

- 当程序块内的声明命名一个标识符时，该标识符在该程序块内是可见的。
- 新的声明会临时“屏蔽”旧的声明，同时标识符会获得新的意义。
- 在程序块的末尾，标识符重新获得原来的意义。



1. 在声明 1 中，i 是具有静态存储期限和文件作用域的全局变量；
2. 在声明 2 中，i 是具有程序块作用域的形式参数；
3. 在声明 3 中，i 是具有程序块作用域的局部变量；
4. 在声明 4 中，i 是具有程序块作用域的临时变量。

声明 2 屏蔽了声明 1，所以赋值语句 i=1; 引用的是声明 2 中的形式参数，无法引用声明 1 中的全局变量。

声明 3 屏蔽了声明 1 和声明 2，所以判断语句 if(i>0) 引用的是声明 3 中的局部变量。

声明 4 屏蔽了声明 3，所以赋值语句 i=3; 引用的是声明 4 中的局部变量。

- 赋值语句 i=4; 引用的是声明 3 中的局部变量，声明 4 程序块执行结束变量 i 的值自动销毁，无法被其他语句引用。
- 赋值语句 i=5; 引用的是声明 1 中的变量。

C 语言宏是由预处理器处理的，因此宏不遵从通常的范围规则。

宏定义的作用域通常到这个宏所在文件的末尾，因此定义在函数内的宏并不是仅在函数体内有效，而是在定义宏之后的文件内都有效。

为了控制宏的作用域，可以使用 #undef 指令来“取消定义”，#undef 指令有如下形式：

```
#undef macro-name
```

这样可以删除一个宏的现有定义，以便于重新给出新的宏定义。如果当前宏名未被定义，则 #undef 指令没有任何意义。

16.4 Assignment

变量的值是通过赋值语句[3]获得的，C 语言中的赋值语句形式如下：

```
变量 = 表达式;
```

- 可以出现在赋值运算符左侧的表达式称为左值；
- 可以出现在赋值运算符右侧的常量、表达式和表达式称为右值。

通常情况下，赋值运算符的右侧可以是一个含有常量、变量和运算符的表达式，它也可以以嵌套赋值或多重赋值的形式出现。

简单语句由表达式跟上分号构成，当需要引入一种新语句时，一般使用如下所示的语法来简要介绍一种特殊语句类型以便参考。

语法：赋值语句

```
variable = expression;
```

其中：

`variable` 是想要设置的变量，`expression` 指定了特定的值。

在其他编程语言中，赋值是语句，而在 C 语言中，赋值是运算符。换句话说，赋值操作产生结果，因此上述的赋值表达式 `variable=expression` 的值就是赋值运算后 `variable` 的值。

大多数 C 语言运算符不会改变操作数的值，但是也有一些会改变，它们不仅仅会计算出值，还会产生副作用（side effect），例如简单赋值运算符（=）就改变了运算符左边的操作数。

要写一个赋值语句，必须以一个变量名（称为左值[4]）开始，然后依次是等号、表达式和分号，其中等号的左边可以是任何变量名，等号右边可以是任何表达式。上述范例的剩余部分（等号和分号）是固定的。

另外，利用变量原有值计算出新值并重新赋值给这个变量在 C 语言程序中是非常普遍的，因此 C 语言提供了复合赋值（compound assignment）运算符来更新已经存储在变量中的值。

当在函数定义中声明变量时，可画出一个新的盒子来存储变量的值，并用变量名来标记这个盒子。例如：一个函数开头如下声明：

```
int n1,n2;
string msg;
```

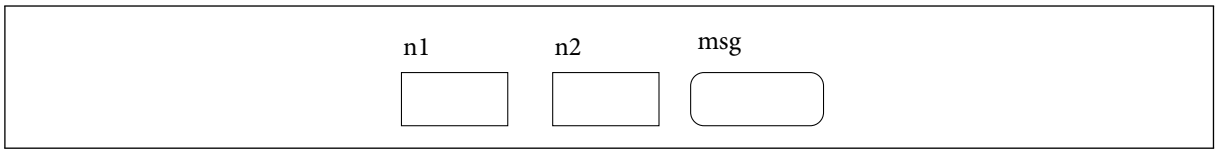
可以为函数中的每一个变量画一个盒子表示这些变量，如下所示：

这里变量周围的外框表示这些变量是在同一个函数中定义的。在这里，盒子最初为空，表明尚未对变量赋值，可以使用赋值语句对变量赋值。

每个变量每次只能存储一个相应类型的值，否则，C 语言编译器会将此语句标记为错误，从而可以看出变量最重要的一个性质——每个变量每次只能存放一个值，一旦对变量进行赋值，则变量将保存该值，直到该变量被赋予新的变量值为止。

[3] 赋值语句种使用的等号“=”在 C 语言中是一个二元运算符。

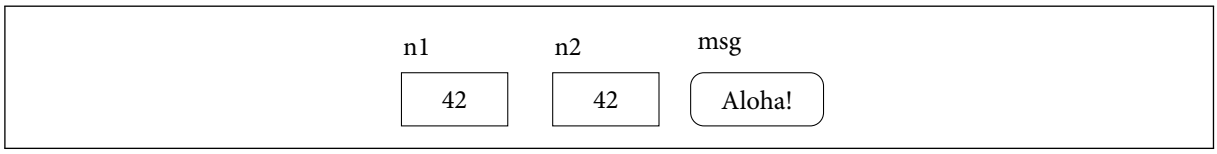
[4] 左值表示存储在计算机内存中的对象，而不是常量或计算结果。



当给一个变量赋予新值时，原来的值将丢失。若将一个变量赋予另一个变量，则原来的变量的值不会消失，因此赋值语句：

```
n1 = 42;
n2 = n1;
msg = "welcome";
msg = "Aloha!"
```

仅改变变量 n2 的值，n1 的值不变。当为变量赋予新值时会覆盖变量中原来的值，即原先的值会消失，被新赋予的值取代。



16.4.1 Integer

如果整型变量发生溢出，有可能的原因是程序赋给变量的值太大以至于无法存储在 int 类型中。溢出取决于操作数是有符号型还是无符号型。

- 当溢出发生在有符号数的操作上时，依据 C 语言标准，结果是“未定义的”。
  此时，我们无法准确地说出结果是什么，这依赖于机器的行为，有可能程序异常中断（对除以零的典型反应）。
- 当溢出发生在无符号数的操作上时，结果是定义了的。
  此时，我们可以获得正确结果对 2^n 进行取模运算的结果，这里的 n 是用于存储结果使用的位数。例如，如果用 1 加上无符号的 16 位数 65535，那么结果肯定是 0。

在发生溢出的情况下，我们的第一个想法是改变变量的类型，从 int 型变为 long int 型。但是，仅仅这样做是不够的，我们必须检查根据数据类型的改变对程序的其他部分的影响，尤其是需要检查变量是否用在 printf 函数或 scanf 函数的调用中。如果用到了，那么将需要改变调用中的格式串，因为%d 的转换只针对 int 型数值。

下面是读和写无符号、短和长的整数需要的转换说明符。

- 当读或写无符号整数时，使用字母 u、o 或 x 代替转换说明中的 d。
  - 如果使用了%u 说明符，那么读（或写）的数是十进制形式的。
  - 如果使用了%o 说明符，则指明读（或写）的数是八进制形式的。
  - 如果使用了%x 说明符，则指明读（或写）的数是十六进制形式的。

```
unsigned int u;
scanf("%u", &u); /*reads u in base 10*/
printf("%u", u); /*writes u in base 10*/
scanf("%o", &u); /*reads u in base 8*/
printf("%o", u); /*writes u in base 8*/
scanf("%x", &u); /*reads u in base 16*/
printf("%x", u); /*writes u in base 16*/
```

对于有符号的普通整数，只要它的值不是负值，都可以继续使用%o 和%x 来以八进制或十六进制输出，这些转换说明导致 printf 函数把有符号整数看成是无符号的。

- 如果符号位为 0，那么 printf 函数将假设符号位是数的绝对值部分。
- 如果符号位为 1，那么 printf 函数将显示一个超出预期的大数。

没有直接的方法输出负数的八进制或十六进制形式，只是这种情况非常少，因此可以首先判定是否为负数，并且自行在负数前显示一个符号。

```
if(i < 0)
  printf("-%x", -i);
else
  printf("%x", i);
```

- 当读或写短整型数时，在 d、o、u 或 x 前面加上字母 h。

```
short int s;
scanf("%hd", &s);
printf("%hd", s);
```

- 当读或写长整型数时，在 d、o、u 或 x 前面加上字母 l。

```
long int l;
scanf("%ld", &l);
printf("%ld", l);
```

```
/* Sums a series of numbers (using long int variables) */
#include <stdio.h>

main()
{
  long int n, sum = 0;
  printf("This program sum a series of integers.\n");
  printf("Enter integer (0 to terminate): ");

  scanf("%ld", &n);
  while(n != 0){
    sum += n;
    scanf("%ld", &n);
  }
  printf("The sum is: %ld\n", sum);

  return 0;
}
```

16.4.2 Float-point

转换说明%e、%f 和%g 用于读和写单精度浮点数，而 double 和 long double 类型值则要求略微不同的转换。

- 当读取 double 类型的数值时，在 e、f 或 g 前放置字母 l。

```
double d;
scanf("%lf", &d); /*reads double type variable*/
```

只能在 scanf 函数格式串中使用 l，不能在 printf 函数格式串中使用。

在 printf 函数格式串中，转换 e、f 和 g 可以用来输出 float 型和 double 型值。

- 当读或写 long double 类型的值时，在 e、f 或 g 前放置字母 L。

```
long double ld;
scanf("%Lf", &ld);
printf("%Lf", ld);
```

针对使用%lf 读取 double 型的值，而用%f 进行显示的问题，首先要注意到 scanf 函数和 printf 函数都是不同寻常的函数，它们都没有将函数的参数限制为固定数量，而且都有可变长度的参数列表。

当调用带有可变长度参数列表的函数时，编译器会安排 float 参数自动转换成为 double 类型，其结果是 printf 函数无法区分 float 型和 double 型的参数，因此在 printf 函数调用中可以使用%f 来表示 float 和 double 型的参数。

另一方面，scanf 函数是通过指针指向变量，%f 告诉 scanf 函数在所传地址位置上存储一个 float 型值，而%lf 告诉 scanf 函数在该地址上存储一个 double 型值。这里 float 和 double 的区别是十分重要的，二者可能具有不同的位模式，因此错误的转换说明会导致 scanf 函数存储错误的字节数量。

16.5 Initialization

在程序开始执行时，某些变量会自动设置为 0，但大多数情况下不会，这就导致用户往往无法预计变量的初始值。为了避免这种情况，可以使用赋值的办法来对变量进行初始化，或者在变量声明中加入初始值。

可以在执行开始时将全局变量初始化为包含特定的值，称为静态初始化（static initialization）。

下面是初始化一个原子变量的句法。

```
type name = initializer;
```

对于包含多个组成部分的复合类型的变量来说，可以将一组初始值包含在一对花括号内来初始化整个集合。如果该复合数据类型包含多层数据结构，可以使用更多的花括号来指定内部结构。

16.6 Operators

16.6.1 Arithmetic Operator

在一个表达式中，实际的运算是由连接各项的运算符表示的。最简单的运算符是数学表达式中的运算符，其中采用的是算术中的标准运算符，适用于所有数字型数据的算术运算符为：

- + 加法
- – 减法
- * 乘法
- / 除法

每一个运算符左右两边各连接一个小的表达式，形成一个新的表达式，这些次级表达式（或称子表达式，subexpression）称为该运算符的操作数（operand）。操作数常常是独立的项，但也可以为更复杂的表达式。

与传统的数学表达式一样，运算符“-”有两种形式。当“-”在两个操作数之间时，表示相减；当其左边没有操作数时，表示负号，因此-x 表示 x 的相反数，此时运算符“-”称为一元运算符（unary operator），因为它只用于一个操作数，其余的运算符（包括表示相减的“-”）称为二元运算符（binary operator），它们适用于一对操作数。

在 C 语言中，int 型数据和 double 型数据可随意结合。若使用一个二元运算符，其两边均为 int 型数，则结果为 int 型数。若其中一边或两边均为 double 型，则结果总为 double 型。因此，若变量 n 为 int 型数，则表达式 n+1 值为 int 型数，而表达式 n+1.5 的值总为 double 型。这种习惯作法可以确保计算结果尽可能精确。

首先讨论除法运算符两边的数均为整型数的情况。按 C 语言的规则，此运算的结果必须为整型数，但当不能整除时，将会把余数丢弃，若想计算出数学上正确的结果，应当至少有一个操作数为浮点数，例如：

```
/*
* File:inchtocm.c
* ------------------
* This program reads in a length given in inches and converts it
* to its metric equivalent in centimeter.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  double inch, cm;

  printf("This program converts inches to centimeters.\n");
  printf("Length in inches?");
  inch = GetReal();
  cm = inch * 2.54;
  printf("%g in = %g cm\n.", inch, cm);
}
```

```
9.0 / 4
9 / 4.0
9.0 / 4.0
```

上述每个表达式都可以得到浮点型数值 2.25，但 9/4 得到 2。也就是说，运算符/可能产生意外的结果，当两个操作数均为 int 型数时会将余数丢弃。

另外，在 C 语言中使用% 运算符计算余数，% 运算符要求左右两边的操作数均为 int 型数[5]，它返回的值是第一个操作数除以第二个操作数所得的余数。

```
9 % 4 = 1        19 % 4 = 3
0 % 4 = 0        20 % 4 = 0
1 % 4 = 0        20001 % 4 = 1
4 % 4 = 0
```

在程序设计过程中，% 运算符常用于检查一个数可否被另一个数整除，若要决定一个整数 n 能否被 3 整除，只要看表达式 n%3 的结果是否为 0 即可。

当% 两边的两个操作数中有一个带负号或两个均带负号时，C 语言的行为依赖于具体的实现，不同的 C 语言工具处理这种情况的行为是不同的。如果完全依赖于一种行为，将会使程序难于移植。为确保程序在所有机器上都能以相同的方式运行，应避免使用带负号的% 操作数。

16.6.2 Macro Operator

C 语言宏定义可以包含两个运算符：# 和 ##，编译器不会识别这两种运算符，它们只在预处理阶段运行。

[5]除了% 运算符外，其他算术运算符允许操作数可以是整数，也可以是浮点数，也可以是二者的混合

运算符仅允许出现在带参数的宏的替换列表中，并将一个宏的参数转换为字符串字面量，因此 # 操作也可以理解为“stringization(字符串化)”。如果要被“字符串化”的参数包含 “或\字符，# 运算符会将 “转换为\“，\转换为\\，考虑下面的宏：

```
#define STRINGIZE(x) #x
```

这种情况下，预处理器会将 STRINGIZE(“foo”) 替换为 “\“foo\” ”。

假设在调试过程中使用 PRINT_INT 宏来输出一个整型变量或表达式的值，使用 # 运算符可以使 PRINT_INT 为每个输出的值添加标签。

```
#define PRINT_INT(x) printf(#x " = %d\n", x)
```

这样，x 之前的 # 运算符通知预处理器根据 PRINT_INT 的参数创建一个字符串字面量，因此调用

```
PRINT_INT(i/j);
```

可以输出

```
printf("i/j" "= %d\n", i/j);
```

C 语言中相邻的字符串会被合并，因此上边的语句实际上等价于：

```
printf("i/j = %d\n", i/j);
```

运算符可以将两个记号(标识符等)“粘合”成一个记号。如果其中的一个操作数是宏参数，“粘合”会在当形式参数被相应的实际参数替换后发生。

考虑下面的宏：

```
#define MK_ID(n) i##n
```

当 MK_ID 被调用时，预处理器首先使用自变量(假设是 1)替换参数 n。接着，预处理器将 i 和自变量“粘合”成为一个记号(i1)。

下面的声明使用 MK_ID 创建了 3 个标识符。

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

该声明在预处理阶段会转换成下面的形式。

```
int i1, i2, i3;
```

在开发获取两个数中较大值的宏 MAX 时，当 MAX 的参数有副作用时会无法正常工作，一种解决方案是用 MAX 宏来实现一个 max 函数。实际上，进行比较操作时需要考虑的数据类型可能不统一，因此往往一个 max 函数是不够的。

在实现 MAX 宏时发现，除了实际参数的类型和返回值的类型外，这些 max 函数都是一样的，因此为每种情况都定义一个函数是不明智的。

可以定义带参数的宏来展开为 max 函数的定义，这样宏使用唯一的参数类型来表示形式参数和返回值的类型。另外，为了避免在同一文件中出现两个同名函数的问题，可以使用 ## 运算符为每个版本的 max 函数构造不同的名字。

```
#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{                                       \
  return x > y ? x : y;            \
}                                       \
```

这样在宏的定义中就可以将 type 和 _max 函数连接成为新的函数名。假设需要一个针对 float 值的 max 函数，就可以使用 GENERIC_MAX 宏来定义函数如下：

```
GENERIC_MAX(float);
```

经过预处理后，上述语句将被转换为：

```
float float_max(float x, float y)
{
  return x > y ? x : y;
}
```

在替换列表中依赖 ## 的宏通常不能嵌套调用，否则它们无法按照“正常”的方式扩展。

在下面的宏定义示例中，CONCAT(a,b) 会输出 ab，但 CONCAT(a,CONCAT(b,c)) 会给出一个怪异的结果。

```
#define CONCAT(x,y) x##y
```

C 语言标准指出，位于 ## 运算符之前和之后的宏参数在替换时不被扩展，因此 CONCAT(a,CONCAT(a,b)) 会被扩展成 aCONCAT(b,c)，而实际上没有 aCONCAT 宏，因此不会进一步扩展。

下面是针对上述问题的解决方案，通过定义第二个宏 CONCAT2 并将原来的表达式改为 CONCAT2(a,CONCAT2(b,c)) 来得到希望的结果 abc。

```
#define CONCAT2(x,y) CONCAT(x,y)
```

CONCAT2 的扩展列表不包含 ##，这样在扩展外部的 CONCAT2 宏调用时，预处理器将会同时扩展 CONCAT2(b,c)。

运算符也有同样的问题。如果 #x 出现在替换列表中，其中 x 是一个宏参数，其对应的实际参数也不会被扩展。

假设 N 是一个代表 10 的宏，且 STR(x) 包含替换列表 #x，STR(N) 扩展的结果为 “N” 而不是 “10”，解决的方法与 CONCAT 宏类似——定义第二个宏来调用 STR。

16.6.3 Comma Operator

在创建较长的宏时，逗号运算符会十分有用。特别是可以使用逗号运算符来使替换列表包含一系列表达式。

例如，下面的宏读入一个字符串，并把字符串输出到屏幕上。

```
#define ECHO(s) (gets(s), puts(s))
```

gets 函数和 puts 函数的调用都是表达式，因此使用逗号运算符连接它们是合法的，这样可以把 ECHO 宏当在一个函数来使用。

除了使用逗号运算符，还可以把 gets 函数和 puts 函数的调用放在大括号中形成复合语句。

```
#define ECHO(s) { gets(s); puts(s); }
```

如果一个宏需要包含一系列的语句，而不仅仅是一系列的表达式，那么就需要将定义放在语句结构中，并设置相应的条件。例如，上述宏定义中 gets 函数和 puts 函数的调用语句可以放在 do 循环中，并将条件设置为假。

```
#define ECHO(s) \
          do{       \
               gets(s); \
               puts(s); \
          }while (0);
```

这样，当调用 ECHO(str) 时，实际执行的语句如下：

```
do{ gets(str); puts(str); }while(0);
```

16.7 Precedence

在表达式中有多个运算符且没有括号进行分隔时，运算的先后顺序由运算符的优先级和结合性规则来决定先用哪个运算符。

在 C 语言中,优先级顺序是由符合标准数学用法的一系列排序规则决定的,它们被称为优先级法则(rules of precedence)。对算术表达式来说,常见的规则为:

1. C 编译器首先执行任何一元的负号运算符;
2. C 编译器接下来执行乘法运算符(*,/,%),若两个乘法运算符作用于同一个操作数,则优先执行最左边的运算符。
3. 执行加法运算符(+ 和-),同样,若两个加法运算符作用于同一个操作数,则优先执行最左边的运算符。

优先级法则仅用于两个运算符竞争同一个操作数时。在相同的优先级的情况下,优先级法则并未明显地指定先执行哪一个。如果子表达式是完全独立的,C 编译器可以按任何一种顺序去执行。

16.8 Associativity

结合性(associativity)仅适用于同一优先级的两个运算符竞争同一个操作数时。

作为通用规则,C 语言还允许用括号括在独立的子表达式外来指定运算顺序以得到想要的结果。

```
/*
* File:ave2f.c
* --------------
* This program reads in two floating-point numbers and
* computes their average.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

main()
{
  double n1, n2, average;

  printf("This program averages two floating-point numbers.\n");
  printf("1st number?");
  n1 = GetReal();
  printf("2nd number?");
  n2 = GetReal();
  average = ( n1 + n2 ) / 2;
  printf("The average is %g.\n",average);
}
```

下面考虑计算机对 C 程序中如下表达式的处理:

```
8 * ( 7 - 6 + 5 ) % ( 4 + 3 / 2 ) - 1
```

第一步,先对括号内的子表达式求值,可以从第一个括号 (7-6+5) 开始,得到如下:

```
8 * 6 % ( 4 + 3 / 2 ) - 1
```

实际上,C 编译器对带括号的子表达式的求值顺序是随意的,这取决于对计算机来说哪种顺序更方便,但最终结果是一样的。在编程时应注意避免导致不同求值顺序的情况。

第二步,先做括号中的除法,再计算括号中子表达式的值,得到如下:

```
8 * 6 % 5 - 1
```

此时,C 语言优先级法则指定应先做乘法和取余操作。

```
8 * 6 = 48
48 % 5 = 3
3 - 1 = 2
```

运算符的优先级和结合性的规则允许将 C 语言表达式划分成若干子表达式,而且如果表达式是完全符号化的,那么这些规则还可以确定添加圆括号的唯一方式。与之相矛盾的是,这些规则并不总是允许我们来确定表达式的值,这些表达式的值可能依靠子表达式的求值顺序来确定。

除了含有逻辑运算符以及逗号运算符的子表达式外,C 语言没有定义子表达式的求值顺序,因此如果表达式的值依赖于子表达式的计算顺序时,不同的编译器可能产生不同的结果。

第 17 章

Type Conversion

在执行算术运算时，计算机比 C 语言的限制更多。

一方面，为了让计算机执行算术运算，通常要求操作数有相同的大小（即位的数量相同），并且要求存储的方式也相同。

计算机可能可以直接将两个 16 位整数相加，但是不能直接将 16 位整数和 32 位整数相加，也不能直接将 32 位整数和 32 位浮点数相加。

另一方面，C 语言允许在表达式中混合使用基本数据类型。

在一个单独的表达式中可以组合整数、浮点数或字符等，C 编译器可能需要生成一些指令将某些操作数转换成相同类型，从而使得硬件可以对表达式进行计算。例如，如果对 16 位 int 型数和 32 位 long int 型数进行加法操作，那么编译器将会把 16 位 int 型值转换为 32 位值。

int 型值和 float 型值的存储方式不同，当 int 型数据和 float 型数据进行运算时，那么编译器会把 int 型值转换为 float 型来计算。

17.1 Implicit Conversation

在将不同数据类型的值结合在一起时，如果 C 语言编译器可以自动处理这些类型转换，则称为隐式转换（implicit conversion）。

当表达式中有不同类型的数值或执行赋值语句时，数值类型间就会进行自动类型转换（automatic type conversion）。

自动类型转换的过程隐含在计算过程中，它将一种类型的值转换为另一种兼容的类型。例如，当用数学运算符对一个整型数和一个浮点数运算时，整型数在计算执行之前自动转换为对应的 double 型数，因此对于`1+2.3`，则在执行加法运算前，计算机内部先将整型数 1 转换为浮点数 1.0。当/运算符作用于两个整型数时，结果是用第一个变量除以第二个变量并舍去余数所得到的整型数，要获得余数，可用% 运算符。

在 C 语言中，当对变量赋值时也会进行自动类型转换。若声明变量 total 为 double 类型，并将赋值语句写为：

```
total = 0;
```

那么作为赋值操作的一部分，要先将整型数 0 转换为 double 类型，与某些程序设计语言（和一些程序员）坚持将语句写成：

```
total = 0.0;
```

其作用是相同的，但 0 和 0.0 在数学上的意义是不同的，因此应使用逻辑上和具体应用最适合的形式。0 表示数值精确为 0，因为整型数是确切值；而在数学或统计学中，0.0 通常代表一个接近于 0 的数，它的值只精确到小数后一位。

将一个浮点型数据赋给一个整型变量时也会激发这种自动转换。在转换过程中会舍掉小数部分，舍去小数部分的操作（此处和整型除法中都会出现）称作截尾（truncation）。

```
/*
* File:cmtofeet.c
* ------------------
```

```
 * This program reads in a length given in centimeters and converts it to its
 * English equivalent in feet and inches.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  double totalInches,cm,inch;
  int feet;

  printf("This program converts centimeters to feet and inches.\n");
  printf("Length in centimeters?");
  cm = GetReal();
  totalInches = cm / 2.54;
  feet = totalInches / 12;
  inch = totalInches –feet * 12;
  printf("%g cm = %d ft % in.\n", cm, feet, inch);
}
```

C 语言自身提供里大量不同的基本数据类型(6 种整型和 3 种浮点型以及字符型等), 导致编译器执行隐式转换的规则比较复杂。

在发生下列情况时会进行隐式转换:

1. 当算术表达式或逻辑表达式中操作数的类型不相同时, C 语言执行算术转换;
2. 当赋值运算符右侧表达式的类型和左侧变量的类型不匹配时;
3. 当函数调用中使用的参数类型与其对应的参数的类型不匹配时;
4. 当 return 语句中表达式的类型和函数返回值的类型不匹配时。

17.1.1 Arithmetic Conversion

常用算术转换多用于二元运算符(包括算术运算符、关系运算符和判等运算符等)的操作数。

当整型与浮点型变量进行算术运算时, 类型转换会以更安全的方式进行, 即整数始终可以转换为浮点型(精度会有少量损失)。相反, 如果把浮点型转换为整型, 将有小数部分的损失, 而且如果原始数大于最大可能的整数或者小于最小的整数, 那么将会得到一个完全没有意义的结果。

常用算术转换的策略是把操作数转换成可以安全的适用于两个数值的“最狭小的”数据类型。粗略的说, 如果某种类型要求的存储字节比另一种类型少, 那么这种类型就比另一种类型更狭小。

为了统一操作数的类型, 通常可以将相对较狭小类型的操作数转换成另一个操作数的类型来实现, 也就是所谓的提升(promotion)。

最常用的提升是整型提升(integral promotion), 它把字符或短整型转换成整型数(或者某些情况下是 unsigned int 类型)。

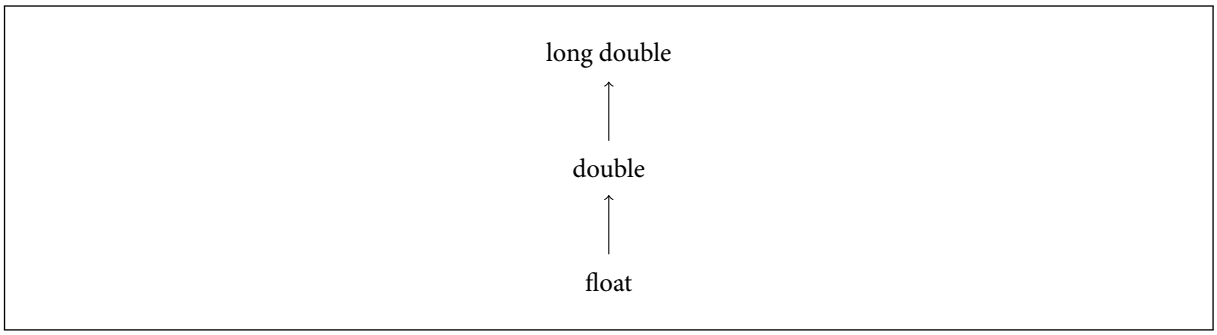
在 int 型整数不够大到可以包含所有可能的原始类型值的情况下, 整型提升会产生 unsigned int 型。因为字符通常是 8 位的长度, 而且至少可以保证 int 型未 16 位的长度, 所以整型提升几乎可以总会把字符转换成 int 型。但是, 如果短整型和普通整型的长度相同, 那么整型提升必将会把无符号短整数转化为 unsigned int 型, 因为最大的无符号短整型(在 16 位机上是 65535)要大于最大的 int 型数(即 32767)。

执行常用算术转换的规则可以划分成两种情况。

- 任一操作数的类型是浮点型的情况。

  按照下图将类型较狭小的操作数进行提升。

  也就是说, 如果一个 ie 操作数的类型为 long double, 那么把另一个操作数的类型转换成



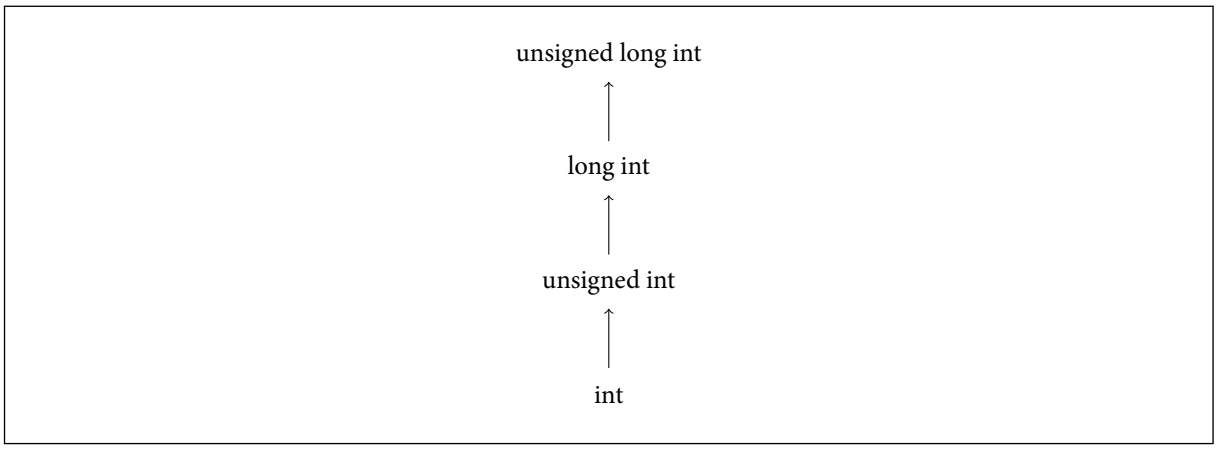
long double 类型。否则，如果一个操作数的类型为 double 类型，那么把另一个操作数也转换为 double 类型。

如果一个操作数的类型是 float 类型，那么把另一个操作数转换成 flaot 类型。

上述这些规则涵盖了混合整数和浮点数类型的情况。例如，如果一个操作数的类型是 long int 类型，并且另一个操作数的类型是 double 类型，那么把 long int 类型的操作数转换成 double 类型。

- 两个操作数的类型都不是浮点型的情况。

  首先对两个操作数进行整型提升（保证没有一个操作数是字符型或短整型），然后按照下图对操作数的类型进行提升。



有一种特殊情况，只有在 long int 类型和 unsigned int 类型长度相同（比如 32 位）时才会发生。在这种情况下，如果一个操作数的类型是 long int，而另一个的类型是 unsigned int，那么两个操作数都会转换成 unsigned long int 类型。

当把有符号操作数和无符号操作数整合时，会通过把符号位看成数的位的方法来把有符号操作数“转换”成无符号的值，这样的规则可能会导致某些隐藏的编程错误。

假设 int 型变量 i 的值位-10，而且 unsigned int 型变量 u 的值为 10。如果用 < 运算符比较变量 i 和 u，我们期望的结果是 1（真），但是实际上在比较操作前，变量 i 会被转换为 unsignede int 类型，原因是负数不能被表示成无符号整数，所以转换后的数值将不再为-10，而是一个大的正数（将变量 i 中的位看作是无符号数），导致 i<u 比较操作的结果为 0。

由于上述陷阱的存在，因此最好尽量避免使用无符号整数，特别是不要把它和有符号整数混合使用。

下面的例子显示了常用算术转换的实际执行情况。

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
```

```
float f;
double d;
long double ld;

i = i + c; /* c is converted to int */
i = i + s; /* s is converted to int */
u = u + i; /* i is converted to unsigned int */
l = l + u; /* u is converted to long int */
ul = ul + l; /* l is converted to unsigned long int */
f = f + ul; /* ul is converted to float */
d = d + f; /* f is converted to double */
ld = ld + d; /* d is converted to long double */
```

17.1.2 Assignment Conversion

算术转换不适用于赋值运算，C 语言会遵循另一条简单的规则如下：

把赋值运算符右边的表达式转换成左边变量的类型。如果变量的类型至少和表达式类型一样“宽”，那么这种转换将没有任何障碍。

```
char c;
int i;
float f;
double d;

i = c; /* c is converted to int */
f = i; /* i is converted to float */
d = f; /* f is converted to double */
```

这条规则在其他情况下是有损失的，如果把浮点数赋给整型变量会丢掉小数部分，发生“截尾”现象。

如果取值超过变量类型范围，那么当把值赋给一个较狭小类型的变量时将会得到无意义的结果，可能会导致编译或 (sp)lint 发出警告。

17.2 Explicit Conversion

如果需要更大程度的控制类型转换，C 语言允许使用强制运算符来明确地指出类型转换，而且必须在执行运算之前完成类型转换，这称为显式转换（Explicit Conversion）。

例如，已经声明了两个整型变量 num 和 den，要计算出两者的商（包括小数部分）并将其赋值给一个 double 类型的变量 quotient，此时语句不能写成：

```
quotient = num / den;
```

num 和 den 均为整型数，当对两个整型数进行除法运算时，结果会舍去小数部分。为避免出现这种问题，必须在进行除法运算之前将至少一个值转换为 double 型。

在 C 语言中，可使用强制类型转换（type cast）来明确指出类型转换。

强制类型转换是一个一元运算符，由括在括号内的目的数据类型跟上要转换的数值组成，这样可使用强制类型转换运算符来指定数据类型间的显式转换，例如：

```
double quotient;
int num,den;
quotient = num / den;
quotient = num / ( double ) den;
```

```
quotient = ( double ) num / den;
quotient = ( (double) num )/den;
```

第二个和第三个表达式的计算结果与第四个表达式相同，而且第四个表达式的运算顺序和第三个表达式相同，第一个表达式计算时，C 语言将先用一个整数去除一个整数，舍去小数部分后再将所得的整数转换为 double 类型，从而得不到与后面的表达式相同的结果。

下面的示例显示了使用强制类型转换表达式计算 float 型值小数部分。

```
float f, frac_part;
frac_part = f - (int) f;
```

C 语言的常用算术转换要求在进行减法运算前把 (int)f 转换成 int 类型，f 的小数部分在进行强制类型转换时被忽略掉了。

强制类型转换可以用来控制编译器并强制它进行所要求的转换。

在下面的示例中，触发的结果是一个整数，在把结果存储在 quotient 变量中之前，将会首先把结果转换成 float 类型。

```
float quotient;
int divided, divisor;

quotient = divided / divisor;
```

为了得到更精确的结果，可能需要在除法执行之前把 divided 和 divisor 的类型转换成 float 类型，下面使用强制类型转换改进上述示例程序。

```
float quotient;
int divided, divisor;

quotient = (float) divided / divisor;
```

这里，变量 divisor 不需要进行强制类型转换，因为把变量 divided 强制转换成 float 类型会强制编译器把 divisor 也转换成 float 类型。

C 语言把 (类型名) 视为一元运算符，优先级高于二元运算符，因此编译器会把下面的表达式

```
(float) divided / divisor
```

解释为

```
((float) divided) / divisor
```

下面的语句可以实现同样的效果。

```
quotient = divided / (float) divisor;
```

或者

```
quotient = (float) divided / (float) divisor;
```

强制类型转换可以用来避免溢出。在下面的示例中，表达式 j*j 的值是 1 000 000，并且变量 i 的值是 long int 型的，应该可以很容易地存储 j*j 的结果值，但最终 i 的结果却是错误的。

```
#include <stdio.h>

main()
{
  long int i;
  int j = 1000;

  i = j * j;
  printf("%ld\n", j*j);
  printf("%ld", i);
}
```

当两个 int 型值相乘时，结果也应该是 int 类型的，但是 j*j 的结果太大以至于在某些机器上无法表示成 int 类型，这时就会把结果转换成一个无意义的值，所以 j*j 和 i 的值都是无意义的，这种情况叫做整型溢出。

- 如果值是整型并且变量是无符号类型，那么发生溢出时会舍弃超出的位数；
- 如果变量是有符号类型，那么结果将由实现定义。
- 如果浮点数赋值给整型变量，会产生未定义的行为。

为了防止溢出产生无意义的结果，可以使用强制类型转换来避免这种问题的发生。

```
long int i;
int j = 1000;

i = (long int) j * j;
```

强制类型转换运算符的优先级高于 *，所以第一个变量 j 会被转换为 long int 类型，同时也强制提升第二个 j 的数据类型。

下面的示例说明了类型转换和运算顺序的关系。

```
long int i;
int j = 1000;

i = (long int) (j * j);
```

上述语句中，乘法运算将早于类型转换执行，因此溢出在强制类型转换之前就已经发生了。

第 18 章

Programming Idiom

多个独立的语句可以组合为复合语句，通常称为程序块。程序中常见的操作可以表示为程序设计习语，它提供了能在各种程序设计问题中使用的简单的语句模式。

C 语言中的程序设计习语（programming idiom）指的是一条或一组 C 语言。虽然可以就特定情况作出相应的修改，但它的大体结构是固定的。要编写高效的程序，必须学会在问题求解时灵活地使用这些程序设计习语，从而在头脑中产生一个关于解决方案的某方面的大体想法时，将能很自然地把那些想法翻译成恰当的习语，构造出想要的程序。

示例：读入一个整型数的习语

```
printf("prompt string");
variable = GetInteger();
```

这两行程序是一个程序设计范例（paradigm），即展示特定语句或习语语法结构的 C 代码块。范例中 prompt string 和 variable 都能用任何字符串和变量代替。这里，通过替换就可以用同一个基本习语请求输入其他数值。

有些 C 语言的习语，如复合赋值习语的存在主要是为一些常用操作提供方面的简化形式。对于下面的赋值语句：

```
balance = balance + deposit;
```

要了解这个赋值语句，不能将赋值中的“=”看作数学上的等于表达式，因为在数学上，方程

```
x = x + y
```

只有当 y = 0 时可解。除此之外，x 不可能等于 x+y，而赋值语句是一个主动的操作，它将右边表达式的值存入左边的变量中。因此，赋值语句

```
balance = balance + deposit;
```

不是断言 balance 等于 balance + deposit，它是一个命令，使得 balance 的值改变为它之前的值与 deposit 的值之和。

在 C 语言中，对一个变量执行一些操作并将结果重新存入变量的语句在程序设计时使用十分频繁，因此 C 语言的设计者引入了它的一个习惯的简记方式。对于任意的二元运算符 op，以下形式的语句

```
variable = variable op expression
```

都可以写成

```
variable op= expression
```

在赋值中，运算符与“=”结合的运算符称为复合赋值运算符（shorthand assignment operator），这种简化形式适用于 C 语言中的所有的二元运算符，例如 +=、-=、*=、/=。

```
v += e; /* 只求一次v的值 */
v = v + e; /* 求两次v的值 */
```

上面的示例表明，任何副作用都会导致两次求 v 的值。

18.1 Increment/Decrement

除简化赋值运算符外，C 语言还为另外两种常见的程序设计操作（给一个变量加 1 或减 1）提供了更加简化的形式。

- 将一个变量加 1 称为自增（incrementing）该变量；
- 将一个变量减 1 称为自减（decrementing）该变量。

为了用最简便的形式表示这样的操作，C 语言引入了 ++ 和−−运算符[1]。例如，

```
x++
```

等价于

```
x += 1;
```

而这又是

```
x = x + 1;
```

的缩略形式。

++ 和 −− 运算符只会在 C 程序中出现，而且它们从某种意义上说是 C 语言的标志性特征，没有其他任何符号能比它们更清楚地表示一个给定的程序是使用 C 语言而非其他语言编写的。由于这两个操作符在 C 语言中使用相当普遍，所以 C 语言的后继语言被称为 C++，因为这个名字意味着“C 的后继者”。

实际上自增和自减运算符的使用是很复杂的，原因在于它们既可以作为前缀（prefix）运算符使用也可以作为后缀（postfix）运算符使用，因而程序的正确性可能和选取适合的运算符形式紧密相关。

另外，和赋值运算符一样，自增和自减运算符也有副作用，它们会改变操作数的值。

C 语言标准没有规定自增和自减运算符生效的精确时间，在“顺序点”的概念中提到“在前一个顺序点和下一个顺序点之间应该对存储的操作数的值进行更新”。在 C 语言中有各种不同类型的顺序点，语句就是其中一种。在语句末尾，必须已经执行完所有语句中的自增和自减操作，只有在满足这些条件的情况下才可以执行下一条语句。

```
i = 1;
printf("i is %d\n", --i); /* prints "i is 0" */
printf("i is %d\n", i); /* prints "i is 0" */

i = 1;
printf("i is %d\n", i--); /* prints "i is 1" */
printf("i is %d\n", i); /* prints "i is 0" */
```

- 后缀 ++ 和后缀−−比一元的正号、负号优先级高，而且都是左结合的。
- 前缀 ++ 和前缀−−与一元的正号、负号优先级相同，而且都是右结合的。

在 C 语言的其他运算符（逻辑运算符与逗号运算符）中也强调顺序点，而且在函数调用中也有类似规定：

- 实际参数在函数调用执行之前就必须全部计算出来；
- 若实际参数是含有 ++ 或−−运算符的表达式，则必须在调用发生前进行自增或自减操作。

C 语言中规定自增和自减运算符可以用于所有数值类型，因此也可以处理 float 类型变量，但实际上极少针对 float 变量来使用它们。

[1] C 语言从 Ken Thompson 早期的 B 语言中继承了 ++ 和−−。对于现代编译器而言，使用 ++ 和−−不会使编译后的程序变得更短小或更快。

第 19 章

Control Statements

程序设计习语和范例只不过是构建程序的砖瓦，当遇到一个问题时，程序员的工作就是将这些砖瓦组织成一个连贯的能解决问题的程序。

计算机的一个最大优势就是能快速处理大量数据，使用计算机解决实际问题时处理的也不会是如何将两个数相加，而是如何解决更大规模的问题。

现在考虑一个 10 个数的求和问题。一种方案是依次读入 10 个数，然后将它们加起来，但这不是明智和高效的。另一种方案是在读入数据的同时将它们加起来，这样不会单独保存每一个数据，只存储当前的和，当读入第 10 个数时，也就算出了 10 个数的和。这个方案不用保存每个单独的数据，也就回答了如何在不声明 10 个变量的条件下计算 10 个数值的和的问题。此时只需要用两个变量：一个用于保存每个读入的数据，另一个用于保存当前的和。每当读入一个新的数值时，只需要将它加入到保存之前所有数值和的变量中就可以了，之后，又可以用该变量保存下一个数值，并按同样的方法处理，具体的解决方案步骤如下：

- 声明两个变量，一个当前值，一个当前和。
  在程序的开始添加如下的声明：

  ```
  int value,total;
  ```

- 当输入每个数值时，必须执行下面的步骤，这是“读入一个整型数习语”的典型应用：
  1. 请求用户输入一个整型数，将它存储在变量 value 中；
  2. 将 value 加入到保存当前和的变量 total 中。

  ```
  printf("?");
  value = GetInteger();
  ```

- 把 value 的值加到 total 中，这也是复合赋值习语的一个实例：

  ```
  total += value;
  ```

每个 C 语言程序中的语句都有一定的直接作用：读入数据、计算结果，然后在屏幕上输出数据。进一步说就是这些程序的所有语句都是按顺序执行的，即从 main 函数的第一条语句开始执行到最后一条语句结束。在解决复杂问题时，仅仅有严格的顺序执行是不够的。

下面要做的是寻找出让程序重复执行这些语句 10 次的方法。要指定重复，可以使用控制语句（control statement）。

19.1 Repeat-N-times Idiom

要将一个操作重复执行特定次数，C 语言的标准方法是使用 for 语句，它就是控制语句的一个例子。可以使用重复 N 次习语（repeat-N-times idiom）来揭示 for 语句。

```
for(i = 0;i < N;i ++){
  要重复的语句
}
```

在重复 N 次习语中，数值 N 代表要重复的次数，这里使用的变量 i 称为下标变量(index variable)，实际上它可以是任何整型变量。

在 C 语言中，控制语句由两个不同的部分构成：

1. 控制行：控制语句的第一行被称为控制行(control line)，它以标识语句类型的关键字开始，常包含定义整个控制操作的其他信息。在 for 语句习语中，它的控制行为：

```
for(i = 0;i < N;i ++)
```

for 语句的控制行用来控制大括号中的语句将被执行的次数。

2. 主体：大括号中的语句构成了该控制语句的主体(body)，在 for 语句中，这些语句将按控制行指定的次数重复执行。主体中的每一条语句独占一行，并且一般都比控制行多缩进 4 个空格，从而可以使控制语句的控制范围一目了然。

控制行和主体在概念上是相互独立的，写出一个控制语句的控制行后，就可以在相应的主体中放入任何需要的语句，因此 for 语句能用来重复执行任何语句。

19.2 Iteration

在程序设计中，重复执行一个操作的过程称为迭代(iteration)，迭代也是许多问题解决方案的基本组成部分，特别是那些涉及大量数据的问题。一般说来，解决这类问题的程序需要对每个数据执行同样的操作。

一般常使用循环(loop)来统称在类似 for 语句的控制语句的执行过程中被重复执行的程序段。“循环”这个词来源于早期的计算机，由于那时的程序是通过穿孔纸带输入计算机的。为了重复一系列的操作，程序员将纸带的首尾相连，使纸带成为一个圈，纸带上的指令传给纸带读入机后，纸带绕一圈回来，指令又被输入，从而便达到了根据需要重复执行的目的。

C 语言提供了三种循环语句：while 语句、do 语句和 for 语句，其中最简单的迭代结构就是 while 语句，它重复执行一条简单语句或程序块，直到条件表达式变为 FALSE 为止。

- while 语句用于判定控制表达式在循环体执行之前的循环；
- do 语句用于判定控制表达式在循环体执行之后的循环；
- for 语句根据自增或自减计数变量的初始值和范围来执行循环操作。

当一个 for 循环运行时，计算机顺序执行主体中的每一条指令，当最后一条语句执行完毕时，程序返回到循环开始处并检查是否重复了指定的次数。如果是，则程序从整个循环退出，接着执行 for 语句结尾大括号后的语句；如果还需要重复，计算机将重新从循环体的第一条语句起顺序执行每一条语句。循环里所有语句的一次完全执行称为一个周期(cycle)。

19.2.1 Index Variables

在 for 语句的控制行中：

```
for(i = 0; i < N; i ++)
```

其中，变量 i 被称为下标变量(index variable)，用于跟踪循环的周期数。

下标变量必须也像其他变量一样在函数的开始部分声明变量名。在 for 循环中，变量 i 用于跟踪已经执行了多少个周期。在第一个周期中，i 的值为 0。在下一个周期中，i 的值为 1。在接下来去的每一个周期中，i 的值都将加 1，直到最后一个周期，i 的值为 N-1，其中 N 是 for 控制行中指明的循环次数的上限，因此，在整个循环的执行过程中，变量 i 的值从 0 一直到 N-1，因此 for 循环又称为计数循环(counting loop)。

也可以修改 for 循环习语,让它从其他值开始计数,新的习语为:

```
for(i = first;i <= last;i ++){
  要重复的语句;
}
```

当使用这个习语时,下标变量 i 的值从值 first 开始计数,直到达到值 last 为止。这里的新词习语“<=”操作代替了原来的“<”。

修改后的 for 循环习语最大的好处在于它允许从 1 开始计数,这比从 0 开始符合真实习惯。

C 语言的传统 for 循环习语为:

```
for(i = 0;i < N;i ++)
```

而下面的代码也能正常工作:

```
for(i = 1;i <= N;i++)
```

注意:上面第二个 for 控制行最好在需要 i 的值时才使用。

```
/*
* File:count10.c
* ------------------
* This program counts from 1 to 10,display each number on the screen.
*/

#include <stdio.h>
#include "genlib.h"

main()
{
  int i;

  for(i = 1;i <= 10;i ++){
    printf("%d\n", i);
  }
}
```

19.2.2 Initialization

在循环的第一个周期开始之前,为使程序正确地工作,应该使 total 的值为 0 或其他确定的值。

```
for(i = 0;i < N;i ++){
  printf(''?'');
  value = GetInteger();
  total += value;
}
```

任何变量,在给它赋以确定的值之前,它的值是未定义的。但是,算法设计要求变量的初始值是确定的。在这里,为了使程序正确地工作,在 for 循环的第一个周期开始之前,total 的值必须是 0,因此需要在程序的开始部分添加下面的语句:

```
total = 0;
```

这样才能使后面的语句执行后得到正确的结果,而上述语句就是为了保证 total 有正确的初始值,而在循环开始前,明确地将它赋值为 0。

用赋值语句来保证一个变量有正确的初始值的过程叫做初始化(initialization),未初始化变量是常见的编程错误。

```
/*
* File:add10.c
* ---------------
* This program adds a list of ten numbers, printing the total at the
* end. Instead of read the numbers into separate variables, this
* program reads in each number and adds it to a running total.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

main()
{
  int i, value, total;

  printf(“This program adds a list of ten numbers.\n”);
  total = 0;
  for(i = 0;i < 10;i ++){
    printf(“?”);
    value = GetInteger();
    total += value;
  }
  printf(“The total is %d.\n”,total);
}
```

19.2.3 Read-until-sentinel Idiom

仅仅适用于两三个数据值的解决策略随着问题规模的扩大往往是不可取的,当需要将累加 10 个数的程序修改为能解决累加问题的更通用的程序时,首先考虑只需要对程序进行很小修改的方法。比如,可以在程序的开始部分请求用于输入数据个数,并将其存放在某个变量中,以此来替换 for 语句控制行中使用的常量。假设 n 被声明为一个整型变量,程序的开始几行可以修改为:

```
printf("This program adds a list of numbers.\n");
printf("How many numbers in the list?");
n = GetInteger();
total = 0;
for(i = 0;i < n;i ++){
    ...
}
```

这种解决方案的不足在于,设想要累加一列数据,则需要人工确定共有多少个数,所以应该继续改进。

从用户的观点出发,最好的办法就是定义一个特殊的输入数据,用户可以通过输入该数据来标识输入序列结束,这个用来结束循环的特殊的值称为标志(sentinel)。

选择一个合适的标志值取决于输入数据的本质,选出的标志值不应是一个合法的数据值,即它不能是用户要作为正常数据输入的值,例如,要累加一列整数,则 0 就是一个合适的标志。尽管在一些数据中可能出现 0,但用户可以忽略它,因为它并不影响最后的总和。

这里要注意的是,在编写程序统计考试平均分时的情况会有所不同,在这里,加入 0 成绩 0 会改变最终结果,在这种情况下,0 也是一个合法的数据值。要允许用户输入 0 作为实际分数,就必须选取另外的不会表示实际分数的值作为标志。在大多数考试中,不可能出现负分,因此在这种情况下,选取-1 作为标志是合适的。

要将 add10.c 扩展为新的 addlist.c 程序,唯一的变动是要修改循环结构,重复次数需要预先确定的 for 语句在这里已经不太合适,这时需要一个新的习语,它能一直读入数据,直到发现特定的输入标志为止,这样的习语就叫做读入-直到-标志(read-until-sentinel)习语,它有如下形式:

```
while(TRUE){
  提示用户并读入一个值;
  if(value == sentinel) break;
  程序的剩余部分;
}
```

在学习和实际的程序设计工作中,抛开所有疑惑直接使用并不能完全理解的习语的确有些困难,但是,实际上程序设计高手的一大本领就是能不用理解细节而直接使用库函数或现成代码段。随着程序的日益复杂,使用那些只能从大体上理解的工具的能力是一种很重要的技能。

大多数人都会根据更特定的日常行为来思考程序的用途,而很少人会以抽象的形式思考这个程序的实现。在日常行为中,把一列数相加是很常见的,例如,许多人每个月都会花一定的时间来结算自己的支票簿,这个行为就和加减一列数类似。要解决这种实用的一些问题,只需要将 addlist.c 重新包装,就可以形成结算支票的程序。

要如何改动原来的程序才能成为支票结算程序,这在很大程度上依赖于对它的要求。最初可以只做如下修改:

1. 修改程序开始部分的注释,以便将来读者能明白程序的作用。
2. 适当更改变量名使之更适合要解决的问题。
3. 为用户提供更明确的提示。
4. 使程序改用浮点型数据类型以便用户能输入元角分。
5. 允许用户输入一个初始余额值。
6. 使程序能在循环的每个周期后显示当前的余额值,以便用户能在每笔交易后跟踪帐户信息。

包括了以上这些修改的程序便是下面的程序 balance1.c

为了使这个程序在不修改基本结构的前提下正确地工作,用户必须用输入负数的方法来表示支票。在当前和中加入负数相当于减去相应的支票面额。这个约定很合理也很直观,所以只要程序给出必要的说明,用户就能很好地遵守这个约定。

19.3 Conditional Execution

为了满足实际应用的需要,需要继续引入额外的特性。例如,程序中可能需要检验用户何时被银行退过支票,而要完成这样的功能,首先要编写程序使其能做出判断。

在编写程序时,经常会遇到这样的情况:有时需要在只有满足某个条件的情况下,才执行一些

```
/*
* File:addlist.c
* ---------------
* This program adds a list of numbers. The end of the input is indicated
* by entering 0 as a sentinel value.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  int value, total;

  printf("This program adds a list of numbers.\n");
  printf("Signal end of list with a 0.\n");
  total = 0;
  while(TRUE){
    printf("?");
    value = GetInteger();
    if(value == 0) break;
    total += value;
  }
  printf("The total is %d.\n",total);
}
```

语句, 或者根据一些测试的结果在两个可选的过程中选择一个。程序中的这类操作称为条件执行(conditional execution)。

C 语言中表达条件执行的最简单的方法就是使用 if 语句, 可以以如下两种形式使用 if 语句:

```
if(conditional-test){
  ... statements executed if the test is true ...
}
```

或者

```
if(conditional-test){
  ... statements executed if the test is true ...
}else{
  ... statements executed if the test is false ...
}
```

- 当解决方案只有在特定环境下才执行一系列语句时, 可以使用 if 语句的第一种形式, 在这种情况下, 若环境不满足, 则那些语句就被跳过。
- 当解决方案需要有两种不同的过程, 在满足一定条件的情况下执行其中一个, 不满足条件的情况下执行另一个时, 可以使用 if 语句的第二种形式。

```
/*
* File:balance1.c(初始版本)
* ------------------------------
* This file contains the first version of a program to balance a
* checkbook. The user enters checks and deposits through the
* month(checks are entered as negative numbers).The end of
* the input is indicated by entering 0 as a sentinel value.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

main()
{
  double entry, balance;

  printf(“This program helps you balance your checkbook.\n”);
  printf(“Enter each check and deposit during the month.\n”);
  printf(“To indicate a check, use a minus sign.\n”);
  printf(“Signal the end of the month with a 0 value.\n”);
  printf(“Enter the initial balance:”);
  balance = GetReal();
  while(TRUE){
    printf(“Enter check (-) or deposit:”);
    entry = GetReal();
    if(entry == 0) break;
    balance += entry;
    printf(“Current balance = %g\n”,balance);
  }
  printf(“Final balance = %g\n”,balance);
}
```

范例中的 conditional-test(条件测试)部分是一个能提出问题的特殊类型的表达式,而这其中常会用到 C 语言的关系运算符(relational operator)组成的条件测试。

下面列出了 C 语言中定义的六个关系运算符,同时在后面的括号中给出了与它等价的数学符号:

- == 等于(=)
- != 不等于(≠)
- > 大于(>)
- < 小于(<)
- >= 大于等于(≥)
- <= 小于等于(≤)

这些运算符用来比较它们两端的数值,例如,要测试变量 x 是否大于等于 0,可以利用如下的条件测试:

```
x >= 0; /* x 相当于 balance */
```

通过条件测试,就可以判断一张支票是否被退回,从而实现对结算支票程序的修改。若用户输入的支票面额超出了当前的额度时,程序可以做两件事:

1. 显示一条信息告诉用户支票被退回;
2. 从帐户中扣去银行收取的退回支票的罚金,假设为 10 美元。

要想完成对程序的这个扩展,需要使用一个条件测试来检验支票是否超出用户当前的余额,一个方法是在每次操作后检查余额是否为负。

```
/*
* File:balance2.c(一个有逻辑错误的版本)
* ----------------------------------------------------
* This file contains a buggy second attempt at a program to balance a checkbook.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

main()
{
  double entry, balance;

  printf(“This program helps you balance your checkbook.\n”);
  printf(“Enter each check and deposit during the month.\n”);
  printf(“To indicate a check, use a minus sign.\n”);
  printf(“Signal the end of the month with a 0 value.\n”);
  printf(“Enter the initial balance:”);
  balance = GetReal();
  while(TRUE){
    printf(“Enter check (-) or deposit:”);
    entry = GetReal();
    if(entry == 0) break;
    balance += entry;
    if(balance < 0){
      printf(“This check bounces,$10 fee deducted.\n”);
      balance -= 10;
    }
    printf(“Current balance = %g\n”,balance);
  }
  printf(“Final balance = %g.\n”,balance);
}
```

这个程序与以前版本的唯一不同之处就是下面这条加在 while 循环末尾的 if 语句:

```
if(balance < 0){
  printf(“This check bounces,$10 fee deducted.\n”);
  balance -= 10;
}
```

这条语句的意思是:如果余额小于 0,程序将显示一条信息,并在余额中扣除一定的罚金。

19.4 Debugging

实际上,即使程序员对程序做了一些看似很小的修改后,都应该对修改后的程序做彻底的测试。不对代码进行测试是一个非常严重的失误,然而,更严重的失误是没有意识到所有的代码(不管代码有多简单)都需要测试,而且没有意识到只适用于两三个数据值的解决策略,随着问题规模的扩大,往往是不可靠的。

在上个程序中,如果帐户中的余额为 0 或已小于支票金额时,若继续消费,将会发生透支,此时程序会正确判断,并输出信息,然后扣除罚金 $10,至此,程序运转正常。

而如果继续测试,假设此时帐户已透支 $20,若用户再存入 $10,帐户余额变为-10$,但是程序中此时却开始出现逻辑错误,它会检测帐户余额,发现是负数,则仍会输出已透支信息,并继续扣除 $10,从而从这个错误状况可以找出逻辑错误所在。

看起来似乎对程序无关紧要的修改可能会引入严重的逻辑错误,对程序应该一直保持怀疑的态度,并对它进行尽可能彻底的测试。

如果要透支,必须满足两个条件:第一,用户必须是刚写了一张支票,第二,这张支票的填写致使余额为负。要纠正逻辑错误,必须在测试中包括这两个条件,特别是程序必须先确定已经写了一张支票,然后再去检查它是否透支,同时测试两个条件可以使用 && 运算符。

```
if(entry < 0 && balance < 0){
  printf("This check bounces,$10 fee deducted.\n");
}
```

而修复 bug 后,要确信程序能正确工作,应该对它进行更彻底的测试。具体说,就是要看它在以前错误的例子上能否正确运行。

要避免产生 bug 是不可能的,许多程序经过多年测试可能都未发现明显问题,但是当某个之前没有测试的到的条件满足时,该程序有可能突然崩溃。程序员所能做的就是尽可能地对程序进行彻底的测试,使遗留逻辑错误的可能性最小。

19.5 Formatted Output

printf 函数是 C 语言最与众不同的特性之一,它提供了一个功能强大且十分方便的信息显示机制,可以用来显示整数、实数和字符串等,而这些只是它的基本功能。

printf 库函数可以用来输出浮点数的%g 总是以最短的形式输出。从数学上来说,30.1 和 30.10 是等价的,尽管对一些应用来说,这种表示方法不太合适,但 printf 语句还是选择了前者。

调用 printf 函数的典型形式如下:

```
printf("control string", expression1, expression2, ...);
```

作为参数传递的表达式的个数取决于需要显示的数据的个数。调用 printf 函数时,也可以没有这些表达式,在这种情况下,就是下面的简单调用:

```
printf("control string");
```

如果 printf 语句在处理控制字符串时遇到了% 符号,它将以特殊的方法处理。printf 将% 和紧随其后的字母作为将在此位置显示的数值的占位符,显示的数值由 printf 函数表中第一个未使用的表达式提供,控制字符串中的第一个百分号与控制字符串之后的第一个表达式匹配,第二个百分号与第二个表达式匹配,以此类推,直到所有参数和百分号搭配完毕为止。

```
/*
* File:balance3.c(正确版本)
* ------------------------------
* This file contains a corrected version of a problem to balance
* a checkbox, including a working bounced-checked feature.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

main()
{
  double entry,balance;

  printf(“This program helps you balance your checkbook.\n”);
  printf(“Enter each check and deposit during the month.\n”);
  printf(“To indicate a check,use a minus sign.\n”);
  printf(“Signal the end of the month with a 0 value.\n”);
  printf(“Enter the initial balance:”);
  balance += GetReal();
  while(TRUE){
    printf(“Enter check (-) or deposit:”);
    entry = GetReal();
    if(entry == 0) break;
    balance += entry;
    if(balance < 0 && entry < 0){
      printf(“This check bounces.$10 fee deducted.\n”);
      balance -= 10;
    }
    printf(“Current balance = %g\n”,balance);
  }
  printf(“Final balance = %g\n”,balance);
}
```

程序员有责任保证控制字符串中的百分号的数目与控制字符串之外传给 printf 的表达式数目精确地匹配,但是 C 编译器无法检查这条规则是否被遵守。假设有一个程序调用 printf 函数时,它的占位符与值的个数不符,那么继续运行它会产生不可预测的结果。

printf 能以多种格式码输出数值(printf 最后的字母 f 表示 formatted(格式))。为了能准确地判断出一个值该如何显示,控制字符串中的每个百分号后面都有一个指出输出格式的关键字母。

百分号与关键字母的组合称为格式码(format code),每个格式码要求相应的表达式有相应的特定的数据类型。注意,C 编译器不能检测它们的类型是否匹配,因此使用时需要谨慎,确保对参数使用的是适当的格式码。

使用 printf 函数时,要确保参数的个数与控制串中百分号开头的置换符个数相匹配,此外,还要确保每个参数的类型与对应的格式码是一致的。

下面是 printf 函数中最常见的格式码:

1. %d — “以 10 为基数”十进制整数。
   在%d 格式下数值将以标准十进制计数法的数字字符串形式显示。如果是负数,数值前会加上负号。
2. %f — “定点十进制”形式的浮点数,没有指数。
   在%f 格式下,数值作为数字字符串显示,并在适当位置显示小数点。
3. %e — 指数(科学记数法)形式的浮点数。
   在%e 格式下,数值将用标准程序设计语言的科学计数法的形式显示,例如 d.ddddde±xx 对应于科学中的表示法 d.ddddd×10xx,如果用格式码%E 代表%e,那么除了用大写 E 来表示指数外,输出的结果是一样的。
4. %g — 指数形式或定点十进制形式的浮点数,形式的选择根据数的大小决定。
   在这种格式下,数值用%f 和%e 格式中较短的一个显示。如果用格式码%G 代替%g,则任何以科学计数法输出的数值都会用大写的 E。在不能事先确定输出值有多大的情况下,%g 可能是最好的输出格式。
5. %s — 字符串型。
   在%s 格式下,相应的表达式也必须是字符串,它将字符逐个显示在终端屏幕上,在该字符串出现的百分号没有什么特殊效果。
6. %% — 百分号。
   %% 并不真正地指明一个格式,它只是说明% 本身也是输出的一部分,因此在格式串中遇到两个连续的% 时将输出一个字符%。

```
printf("%f %%", variable);
```

与转换说明符 f 不同,g 的转换将不显示尾随零,而且如果要显示的数值没有小数点后的数字,那么 g 不会显示小数点。在无法预知显示数的大小或数值变化范围很大的情况下,说明符 g 对于数的显示是特别有用的。

- 在用于显示大小适中的数时,说明符 g 采用定点十进制数形式;
- 在显示非常大或非常小的数时,说明符 g 会转换成指数形式以便可以需要较少的字符。

制表和按格式组织报表仍是现代实际程序设计中的一个重要方面。在处理表显示的时候,它最大的特点之一就是数据信息是按列垂直对齐的,从而可以使其他使用者可以很快看出各个值的含义。

- 字段宽度(field width)
- 对齐方式(alignment)
- 数据精度(precision)

为了生成之前所示的栏目清楚、可读性强的表,必须能控制输出格式的几个属性。首先,要产生垂直列,就必须使该列的所有数据条目占用同样大小的空间。分配给一列中每个条目的字符空间称为字段宽度(field width)。选择合理的字段宽度,可以使列中最大的数据项能在列中显示出来。

从软件工程策略的角度考虑,尽管多余的空间越少越好,但还是应该为将来可能的扩展留出一定的空间。

第二个需要考虑的格式属性是对齐方式(alignment),当数字以表形式显示时,标准的方法是让同一列数字的最后一列对齐,这样更便于查看。这种对齐方式称为右对齐(right alignment),因为所有的数据条目都是右侧对齐的。另外同一列名称的第一位字母是左边对齐的,这种对齐方式叫做左对齐(left alignment),它是非数值数据最常用的对齐方式。

最后也是最重要的就是控制显示数据值的数据精度(precision)。计算机可以使计算结果达到想要的精度,但是实际上,只要让数据足够精确即可,不必追求过高的精度。

但是,很多使用表的这种数据的人都认为,所有其中的德数据都是准确的,毕竟它们是计算机计算出来的。而事实上,输出数据永远不糊比输入的数据精确性高,显示过多的数字精度只能给出精确的假象,对提高统计的精确性并无帮助。为了避免这类假象,确保不要输出超过已知正确范围的多余位数。

printf 函数通过在格式码中加入额外的格式信息来提供域宽、对齐方式和输出数据精度的手段。出现在百分号和关键字母之间的看似浮点数的附加信息实际是由下面所介绍的部分组成的，每一个部分都是可选的。

1. 负号：它表示域中数据使用左对齐方式，若没有出现负号，则数据将默认右对齐。
2. 数字字段宽度：它指定用于输出字段的最小字符数，假如要显示的值所占用的空间比字段宽度规定的数值小的时候，将在该域中填充（padded）额外的空格直到满足一定大小。如果字段宽度前没有负号，将在数据左边加额外的空格以便右对齐。如果出现了负号，那么额外的空格将加在数据右边。

   注意，字段宽度指定的是最小的宽度，如果数值太大，不能在指定大小的字段中显示，那么只是扩大字段宽度使之能容纳整个数值，尽管这样会打破列的对齐状态。如果没有指定字段宽度，那么数据值将使用自己需要的确切字符位数来显示，不会在两边加空格。
3. 小数点及精度参数：这个格式信息的解释依赖于格式码，对%e 和%g 来说，精度参数指定了小数点后的位数；对%g 来说，精度参数说明了最大的有效位数。对%s 来说，精度参数说明字符串中最多能显示的字符个数，它避免了过长的字符串对列宽产生的不利影响，如果没有精度参数，那么 printf 显示整个数据。

在使用格式码时，把所有的这些格式说明放在一起，就可以写出格式化输出表格所需的 printf 语句。为了提高程序的可读性，可以使用如下的语句格式。

```
printf("%-14.14s %6d %6d %4.1f%%\n", state, totalArea, forestArea, pecent);
```

19.6 #define

C 语言中进行集中编辑修改的最好的工具就是 #define 结构。在最简单的情况下，#define 有如下的典型形式：

```
#define symbol value
```

在这个范例中，symbol 是一个名字，它遵循变量的命名规则，value 是 C 语言中的一个常量，在 #define 出现过以后，无论 symbol 在程序哪个部分出现，它都将被指定的常量所替代。

例如，在下面的程序进行支票结算时，在程序的开始部分写下这样的语句：

```
#define BouncedCheckFee 10.00
```

就可以利用这个定义来重写程序中的 if 语句：

```
if(entry < 0 && balance < 0){
  printf(“This check bounces,$%.2f fee deducted.\n”,BouncedCheckFee);
  balance -= BouncedCheckFee;
}
```

从此以后，如果要修改支票罚金，接手该程序的程序员只需要修改程序开头的 #define 语句即可。

包括 printf 格式说明修改和 BouncedCheckFee 常量定义的支票结算程序的最终版本如下，而且程序中还包含了帮助新程序员理解如何修改 BouncedCheckFee 的注释。

使用 #define 机制来设置很可能会修改的值的方法是软件工程的一个重要组成部分。

```
/*
* File:balance4.c
* ---------------------
* This file contains the final version of a program to balance a
* checkbook.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”

/*
* Constant:BouncedCheckFee
* ----------------------------------
* To change the charge assessed for bounced checks, change the
* definition of this constant. The constant must be a floating-point
* value(i.e., must contain a decimal point.)
*/

#define BouncedCheckFee 10.00

/* Main program */
main()
{
  double entry, balance;

  printf(“This program helps you balance your checkbook.\n”);
  printf(“Enter each check and deposit during the month.\n”);
  printf(“To indicate a check,use a minus sign.\n”);
  printf(“Signed the end of the month with a 0 valule.\n”);
  printf(“Enter the initial balance:”);
  balance = GetRea();
  whild(TRUE){
    printf(“Enter check (-) or deposit:”);
    entry = GetReal();
    if(entry == 0) break;
    balance += entry;
    if(entry < 0 && balance < 0){
      printf(“This check bounces,$%.2f fee deducted.\n”, BouncedCheckFee);
    }
    printf(“Current balance = %.2f\n”,balance);
  }
  printf(“Final balance = %.2f\n”,balance);
}
```

第 20 章

Statement Classification

如果在没有经过仔细推敲就对程序做了一些修改,或对程序没有进行彻底的测试,很容易在程序中引入一些不易察觉的逻辑错误,这样程序将可能得不到想要的结果。从某种程度上说,逻辑错误在程序设计过程中是难以避免的,但可以通过良好的程序设计规范来降低问题的严重性并节约大量时间。

每一个 C 语言程序都是通过依次执行 main 函数内的语句而运行的。与大多数程序设计语言一样,C 语言的语句分成两大类:简单语句(simple statement)和控制语句(control statement)。

根据语句执行过程中的顺序所产生的影响方式,C 语言的控制语句可以分为以下 3 类:

- 选择语句(Selection Statements)
  if 语句和 switch 语句允许程序在一组可选项中选择一条特定的执行路径。
- 循环语句(Iteration Statements)
  while 语句、do 语句和 for 语句支持重复(循环)操作。
- 跳转语句(Jump Statements)
  break 语句、continue 语句和 goto 语句引起无条件地跳转到程序中某个位置(return 语句也属于此类)。

简单语句执行一些基本的操作,因此又可以划分成由几条语句组合成一条语句的复合语句和不执行任何操作的空语句。

控制语句影响着其他语句执行的次序,从而可以让程序解决一些更复杂的问题。例如,for 语句能根据指定的次数重复执行多条语句,while 语句允许重复执行多条语句直到满足特定的条件为止。

控制语句依靠条件测试来确定执行不同的语句,条件测试表达式可能会具有特殊的“布尔”类型或“逻辑”类型。布尔类型只有两个值,即真和假。

C 语言没有布尔类型,而是规定条件测试表达式的值是 0(假)或 1(真)。

20.1 Simple Statements

在 C 语言中,不论其功能如何,所有的语句都遵守以下规则:

> 一个简单语句由一个表达式及紧跟其后的分号构成。

这就使这些简单语句都有一个统一的结构,也使 C 编译器能辨别一个程序中的合法语句。由上可知,一个简单语句的范例如下:

```
表达式;
```

在表达式后加上分号就使它成为合法的语句形式。注意,尽管任何表达式跟一个分号的组合都是 C 语言的一个合法语句,但并不是这样组合肯定会产生有效的语句。要使一个语句有效,它必须有实际的明确的效果,例如以下语句

```
n1 + n2;
```

由表达式 n1 + n2 紧跟一个分号构成，它是一个合法语句，但它不是一个有效语句。因为它对计算结果没有进行任何操作，该语句将 n1 和 n2 的值相加，然后将结果丢弃（某些编译器也会对这类无效语句给出一个警告）。

C 语言中典型的简单语句是赋值语句（包括复合形式的赋值和自增/自减操作）和能执行一些有用操作的函数调用语句，如 printf。

在 C 语言中，赋值所用的等号是一个二元运算符，运算符“=”带两个操作数，左右两边各一个。

目前可以认为赋值运算符左边的操作数一定是一个变量，在复杂的应用条件下也可能会放松限制。当赋值运算符被执行时，首先计算右边表达式的值，随后将结果存储在赋值运算符左边的变量中。

由于用于赋值的等号是一个运算符，所以事实上，

```
total = 0
```

的确是一个表达式，而

```
total = 0;
```

就是一个简单语句。

20.1.1 Embedded Assignment

如果一个赋值语句是一个表达式，那么表达式本身应该有值；而且，如果一个赋值表达式产生一个值，那么也一定能将这个赋值表达式嵌入到一些更复杂的表达式中去。

当赋值表达式作为更大的表达式的一部分时，该赋值子表达式的值即为它用来赋值的值。例如，如果表达式 x = 6 作为另一个运算符的操作数，那么该赋值表达式的值就是赋给变量 x 的值，因此，表达式 (x = 6) + (y = 7) 等价于分别将 x 和 y 的值设为 6 和 7，并且整个表达式的值为 13。

```
讨论 (x = 6) + (y = 7)
```

由于“=”的优先级比“+”低，所以这里要必须加上括号。将赋值语句作为更大的表达式的一部分，称为赋值嵌套（embedded assignment）。

在较大的表达式中应该限制使用赋值嵌套，因为它会使变量的值发生的改变很容易被忽略。

20.1.2 Multiple Assignment

在使用赋值嵌套的特殊环境中，最适合解决的问题就是当将同一个值赋给多个变量的情况。C 语言对赋值的定义中允许用以下一条语句代替单独的几条赋值语句：

```
n1 = n2 = n3 = 0;
```

它将三个变量的值均设为 0。该语句只所以能达到预期效果是因为 C 语言的赋值操作是从右到左进行的，该条语句等价于：

```
n1 = (n2 = (n3 = 0));
```

表达式 n3 = 0 先被计算，它将 n3 设为 0，并将 0 作为赋值表达式的值传出。随后这个值又赋给 n2，结果再赋给 n1，这种语句叫做多重赋值（multiple assignment）。

当编写多重赋值时，要保证所有的变量都是同类型的，以避免在自动类型转换时出现与预期不相符的结果的可能性。例如，假设变量 d 声明为 double 类型，变量 i 声明为 int 类型，则以下这条语句

```
d = i = 1.5;
```

这条语句在计算时，先将 1.5 截去小数部分赋给 i，因此得到值 1，而此赋值嵌套表达式的值就是要赋给 d 的值，也就是将整数 1 赋给 d，而不是将浮点数 1.5 赋给 d，该值赋给 d 时，再进行第二次类型转换，所以最终赋给 d 的值是 1.0。

在 C 语言的优先级中，赋值运算符（包括复合赋值运算符，如 +=、*=）的优先级比算术运算符低。如果两个赋值语句竞争一个操作数，赋值是从右到左进行的，这条规则与其它运算符正好相反，其他运算符都是从左到右进行的，优先级相同的运算符的计算方向称为结合性（associativity）。

算术运算符等都是从左到右计算的，因此称为左结合的，而赋值运算符等是从右到左计算的，称为右结合的。

20.1.3 Block

可以用简单语句指定计算机的动作，然而对大多数程序来说，解决策略都需要由若干条顺序步骤协同工作。

将完成某项功能的动作翻译成实际的程序步骤需要用多个独立的语句，它们都是主程序体的一部分。为了说明某一连串的语句是一个连贯的单元，可以把这些语句组装成一个程序块（block），用大括号括起来强制编译器将其作为单独的一条语句来处理。

```
{
  optional declarations
  statement1
  statement2
  statement3
  . . .
  statementn
}
```

在 C 语言程序中，术语块（block）可以用来表示函数体，或块语句（包含声明的复合语句等）。例如，简单程序本身就可以理解为一个程序块，其中的语句常常根据上下文的关系有不同程序的缩进，编译器会忽略这些缩进，但这种直观的效果对阅读程序确实很有帮助。

经验表明，每个新层次开始时用三到四个空格缩进会使程序的结构非常清晰。适当的缩进对良好的程序设计来说是很重要的，必须努力使自己的程序有一致的缩进风格。

在程序块中的语句之前允许有变量声明，一般的变量声明只用于定义函数体的程序块内。

另外，程序块中唯一容易使人混淆的问题是分号扮演的角色。在 C 语言中，分号是简单语句语法的一部分，它的作用主要是语句终结符而非分隔符，这对于曾经使用过 PASCAL 语言的人会造成一定的障碍，因为 PASCAL 语言的语法规则略有不同。具体来说，主要是以下几点：

1. 在 C 语言程序块的最后一个简单语句后面总会有分号，而在 PASCAL 中常常不会出现分号，尽管大多数编译器允许将分号作为可选项。
2. 在 C 语言程序块的结束大括号后不会有分号。在 PASCAL 的关键字 END 后，根据上下文可以有分号，也可以没有分号。
3. 在 C 语言中使用分号的习惯有助于程序的维护，并且当习惯以后不会造成任何问题。

为了强调，就编译器而言，它们是语句，因此程序块有时也被称为复合语句（compound statement）。尽管复合语句的作用类似于单条语句，但是复合语句不用以分号结尾。

例如，C 编译器在遇到控制结构中的程序块时，会将整个程序块看作是控制结构中的一个简单语句，因此不管何时在习语或程序范例中看到“statement”（语句）符号，都可以用简单语句或程序块替代它。

20.2 Control Statements

如果没有任何转向指示，C 程序的语句将按照出现的次序顺序地执行一次，然而现实问题的解决策略往往要求能反复执行一系列步骤或从多个动作中选择一个。能够影响其他语句的执行方式的语句叫做控制语句。

C 语言中的控制语句分成两大类：

1. 条件：在解决问题时，常常需要根据一些条件测试的结果，在两个或多个独立的程序执行路径中做出选择。比如有的程序会要求当某个值为负时执行一系列操作，否则执行另一些操作。需要做出决定的那些控制语句称为条件语句。在 C 语言中，有两种条件语句的形式，即 if 语句和 switch 语句。
2. 迭代：有时要处理的问题涉及很多数据项，则程序通常需要将某一操作重复地执行若干次，或在某个条件成立的情况下反复执行某个操作。在程序设计语言中，这样的反复叫做迭代。在 C 语言中，用作迭代的基本语句包括 while 语句和 for 语句。

C 语言中所有的控制语句由两部分组成，分别是控制行和主体，其中控制行说明重复或条件的本质，主体由那些受控制行影响的语句组成。在条件控制语句中，主体可能分成几个独立的部分，在某种情况下执行这一组语句，在另一种情况下执行其他的语句。

控制语句的主体由其他一些语句组成，不论是条件还是迭代，控制语句本身的作用是应用于它主体内的语句。主体内的语句可以是任何类型的，它们可以是简单语句，可以是复合语句，甚至其本身就是一个包含了其他语句的控制语句。当一个控制语句出现在其他控制语句内时，叫做嵌套(nested)。

在一个控制语句中嵌入另一个控制语句的能力是现代程序设计语言的一个重要特性。

20.3 Null Statement

一个分号本身组成了一个没有任何作用的语句，这种语句形式叫做空语句(null statement)。

空语句常用在一个控制结构体中，一般在这种情况下的重要工作都在头语句中完成了。

空语句可以用来构造空循环体的循环部分。例如，在下面的循环中，通过将条件移入控制行表达式中可以变成空循环体。

```
/* Origin loop */
for (d = 2; d < n; d ++)
  if(n % d == 0) break;
/* Null statement loop */
for (d = 2; d < n && d % n != 0; d ++)
  ; /* empty */
```

习惯上，空语句单独放置在一行中，否则可能会混淆 for 语句后面的语句是否是其循环体。

- 虚空的 continue 语句

```
for(d=2; d<n && n%d != 0; d ++)
  continue;
```

- 空的复合语句

```
for(d=2; d<n && n%d != 0; d ++)
  {}
```

空语句会引发一类缺陷(比如引起 if、while 或 for 语句提前结束等)，一般编译器是无法检测出这类错误的。

- 在 if 语句中的圆括号后面接分号时，无论控制表达式的值是什么，if 语句都只执行同样的动作。

```
if (d == 0) ;
  printf("Error: division by zero\n");
```

printf 函数调用不在 if 语句内，无论 d 的值是否等于 0，都会执行此函数调用。

- 在 while 语句中的圆括号后面接分号会产生无限循环。

```
i = 10;
while (i > 0) ;
{
  printf("T minus %d and continuing\n", i);
  --i;
}
```

另一种可能是循环终止,但是在循环终止后只执行一次循环体语句。

```
i = 11;
while (--i > 0) ;
{
  printf("T minus %d and continuing\n", i);
}
```

这个例子只显示:`T minus 0 and continuing`

- 在 for 语句中的圆括号后面接分号会导致只执行一次循环体语句。

```
for(i = 10; i > 0; i --) ;
  printf("T minus %d and continuing\n", i);
```

同样,这个例子也只显示:`T minus 0 and continuing`

由此可见,把普通循环转换成带空循环体的循环不会带来很大的好处,虽然简洁,但不会提高效率。

只有在少数情况下,带空循环体的循环会比其他更清楚。例如,带空循环体的循环更便于读字符数据。

当空语句应用在语句标号时,假设需要在复合语句的末尾放置标号,标号不能独立存在,它必须有语句跟在后面,这时可以在标号后放置空语句来解决这个问题。

```
{
  ...
  goto end_of_stmt;
  ...
  end_of_stmt: ;
}
```

在源代码的预处理指令部分,可以使用 # 来单独占一行形成空指令。这种情况下,空指令没有任何意义,它只是用来标记条件编译模块之间的间隔。

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

20.4 Boolean Data

在解决问题的过程中,常常需要让程序根据对某个特定条件的测试来决定程序后面的行为。例如,if 语句及其他控制结构中常会遇到只能在 TRUE 或 FALSE 中取值的表达式,像这种合法值只能为 TRUE 或 FALSE 的数据称为布尔型数据(Boolean data)。数学家 George Boole 开创了使用布尔型数据的代数方法。

大多数现代程序设计语言都定义了一个特殊的布尔类型,它的值域仅有两个值,不过 C 语言中没有定义这样的类型[1],这不利于理解逻辑判断的本质。

[1]在 C 语言的 ISO 9899:1999 标准中,增加和修改了定义 `bool` 的 `<stdbool.h>`。

可以通过在扩展库中定义 bool 类型来使用布尔型数据，这样就可以定义两个新常量—TRUE 和 FALSE 来弥补 C 语言中的不足，从而就可以在程序中声明布尔型变量，并像操作其他数据一样操作它。

C 语言中定义了一些使用布尔型数据的运算符，这些运算符可以分成两大类：关系运算符和逻辑运算符。

20.4.1 Relational Operators

C 语言中定义了六个关系运算符并与数学中的比较运算符呼应，关系运算符用来比较两个值并产生 0(假)或 1(真)作为结果。这些运算符分成两个优先级，其中比较两个数值大小关系的运算符有：

- `>` 大于
- `<` 小于
- `>=` 大于等于
- `<=` 小于等于

C 语言允许用关系运算符比较整数和浮点数，以及允许的混合类型的操作数。

C 语言关系运算符是左结合的，而且优先级比算术运算符(`+` 或`-`)低，例如表达式 `i+j<k-1` 意味着 `(i+j)<(k-1)`。

表达式 `i<j<k` 在 C 语言中是合法的，但是结果并不符合数学意义。

$$i < j < k \Longleftrightarrow (i < j) < k$$

表达式首先检测 i 是否小于 j，然后用比较后产生的结果 1 或 0 来和 k 进行比较，表达式不测试 j 是否位于 i 和 k 之间，因此正确的表达式应该是：`i<j && j<k`

判断相等关系的运算符也是左结合的，而且优先级比关系运算符还低。

- `==` 等于
- `!=` 不等于

注意，一个等号是赋值运算符，两个等号是关系运算符，用于判断相等关系。由于 C 编译器不能总把这两个运算符误用的错误捕捉到，从而一个等号常常将表达式变成嵌套赋值，而且这在 C 语言中按照语法是完全合法的，虽然其语义已经发生了转变。

例如，要判断变量 x 的值是否等于零，以下的条件表达式：

```
if(x = 0)
  ...
```

结果却会引起混淆，这条语句不会检查 x 的值是否为 0，而是将 x 赋值为 0，然后 C 语言把它理解为测试结果为 FALSE[2]，而正确地判断 x 的值是否为 0 的写法如下：

```
if(x == 0)
  ...
```

可以利用关系运算符和判等运算符返回整数值的事实解决问题，但同时也可能会使程序难以阅读。

关系运算符只能用于比较原子数据(atomic data)值——不能再分解成更小的部分的数据，例如，整型、浮点型、布尔型以及字符型等都可以组成原子数据。但字符串型(string)数据就不是原子数据，它可以再分为独立的字符。

[2]如果注意到应该正常出现运算符 == 的地方出现的是运算符 =，那么一些编译器会产生诸如“Possibly incorrect assignment”的警告。

20.4.2 Logical Operators

除了关系运算符可以比较任何类型的原子值并产生布尔型的结果外，C 语言还定义了三个逻辑运算符（Logical Operator），它们的操作数为布尔型的值，而且产生的结果也是 0 或 1。把逻辑表达式组合起来可以产生另一个布尔值。

- ! 逻辑非（如果后面的操作数为 FALSE，则值为 TRUE）；
- && 逻辑与（如果两个操作数均为 TRUE，则值为 TRUE）；
- || 逻辑或（如果其中一个操作数为 TRUE，则值为 TRUE）。

上面这三个运算符是按优先级递减的顺序排列的，它们可以简称为非、与、或，但是简称并不一定能准确地表达逻辑意义。为了避免这种不准确，用更形式化、更数学化的方法来看待这些运算符才会更有意义。

数理逻辑中用真值表（truth table）来定义逻辑运算符，它能更清楚地展现出布尔表达式的值如何随着操作数的值的变化而变化。下面给定布尔值 p 和 q，相应的真值表可以表述如下：

Table 20.1: && 运算符的真值表

p	q	p && q
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FLASE	FALSE
TRUE	TRUE	TRUE

Table 20.2: || 运算符的真值表

p	q	p || q
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FLASE	TRUE
TRUE	TRUE	TRUE

从而可以明确 || 的数学意义：至少有一个成立。

Table 20.3: ! 运算符的真值表

p	!p
FALSE	TRUE
TRUE	FALSE

使用逻辑运算符可以构建复杂的逻辑表达式，如果想确定如何计算一个很复杂的逻辑表达式，可以先将其分解成这种最基本的操作，然后再为每一个基本表达式建立真值表，可以快速得到结果。

逻辑运算符! 的优先级和一元正号、负号的优先级相同，而 && 和 || 的优先级低于关系运算符和判等运算符。另外，运算符! 是右结合的，而 && 和 || 都是左结合的。

逻辑运算符会将任何非零值操作数作为真值来处理，同时将任何零值操作数作为假值。

- 如果表达式的值为 0，那么! 表达式的结果为 1；
- 如果表达式 1 和表达式 2 的值都是非零值，那么表达式 1&& 表达式 2 的结果为 1；

- 如果表达式 1 和表达式 2 的值中任意一个是(或两个都是)非非零值,那么表达式 1|| 表达式 2 的结果为 1;
- 在所有其他情况中,逻辑运算符产生的结果都为 0。

在使用逻辑运算符时,容易引起混淆的是! 或!= 或 && 或 || 一起出现的情况。因为当提到某种情况不为真时(即需要用到! 或!= 的情况),日常用语和数学逻辑表达有时是相悖的,因此更要注意避免错误。

例如在程序中要表达这样一个意思:x 不等于 2 或 3,通过这个条件测试语言表述,可能会写出如下语句:

```
if(x != 2 || x != 3)
  ...
```

如果用数学观点来观察这个条件测试,会发现只要 x 不等于 2 或只要 x 不等于 3,if 测试中的表达式均为 TRUE。无论 x 取什么值,其中一个表达式必定为真,因为如果 x 为 2,则它必不可能为 3,反之亦然,因此上面写出的 if 测试将永远为 TRUE。在这里,正确地写法应该是:

```
if(x != 2 && x != 3)
  ...
```

常见错误:当 && 和 || 运算符与涉及! 和!= 运算符的关系测试一起使用时,必须非常小心。自然语言在逻辑上有一些模糊,而程序设计要求非常精确。

要避免这个问题,需要加强对自然语言的理解,使它能够准确地表达各种条件。当只要不满足 x 为 2 或 x 为 3,if 语句的条件测试就通过,于是可以直接将这句话翻译成 C 语句如下:

```
if(!(x == 2 || x == 3))
  ...
```

只是这条语句并不直观,而由于真正想测试的是下列条件是否都满足:

1. x 不等于 2
2. x 不等于 3

如果以这种形式来考虑这个问题,就可以把这个测试写为:

```
if(x != 2 && x != 3)
  ...
```

而这个简化是以下面一个普遍成立的数理逻辑关系的特例。

对任何逻辑表达式 p 和 q:

```
!(p || q) 等价于 (!p && !q)
```

而与它对应的转换关系是:

```
!(p && q) 等价于 (!p || !q)
```

这两条规则称为德·摩根定律(De Morgan Law)。过于依赖自然语言逻辑而忽略这些规则的运用是程序设计中出现 bug 的重要根源。

另一个常见的错误是合并几个关系测试时忘记正确地使用逻辑连接。在数学中常可以看到如下表达式:

```
0 < x < 10
```

虽然它在数学中有意义,按在 C 语言中,却是无意义的。为了测试 x 既大于 0 又小于 10,需要用如下这样的语句来表达:

```
0 < x && x < 10
```

常见错误:测试一个数是否在某一特定范围内,只用关系运算符把它们组合起来(就像数学中常用的方法那样)是不够的,条件的两个部分必须明确地用 && 连接起来,如:

```
(0 < x) && (x < 10)
```

20.4.3 Short-circuit Evaluation

C 语言对 || 和 && 运算符的解释与其他很多程序设计语言对 || 和 && 的解释不太一样。例如，在 PASCAL 中，这些运算符（在 PASCAL 中记作 AND 和 OR）需要计算出条件运算符两边的表达式的结果，尽管有时候结果在计算到一半时就能确定，而 C 语言的设计者们采用了一个不同的、更方便的方法。

当 C 语言中计算如下形式的表达式时，子表达式总是从左往右各自被计算，一旦能确定结果就中止计算，称为计算“短路”。

```
exp1 && exp2
```

或

```
exp1 || exp2
```

例如，若 && 表达式中 exp1 为 FALSE，则不需要计算 exp2，因为结果能确定为 FALSE。同样，在 || 表达式中，如果第一个操作数值为 TRUE，就不需要再计算第二个操作数。形如以上这样只要结果确定就停止计算的方法，叫做简化求值（short-circuit evaluation）。

简化求值最大的好处在于第一个条件能控制第二个条件的执行。在很多情况下，复合条件的第二部分只有在第一部分满足某个条件时才有意义。比如，要表达以下两个条件，第一，整型变量 x 的值非零；第二，x 能整除 y。由于表达式 y%x 只有在 x 不为 0 时才计算，所以用 C 语言可以表达这个条件测试为：

```
(x != 0) && (y % x = 0)
```

但是相应的表达式在 PASCAL 中却得不到预期的结果，因为无论何时它都要求计算出 && 的两个操作数。如果 x 为 0，尽管看起来对 x 有非零测试，PASCAL 程序还是会因为除零错误而中止。形如该例中 (x != 0) 这样的为了防止随后部分出现计算错误的条件测试叫做监护条件（guard）。

20.4.4 Flags

bool 型变量在程序设计中又称作标志（flag），例如当按如下方式声明一个布尔型变量时：

```
bool done;
```

变量 done 便成了一个标志，可以用它来记录程序中是否完成了一定的操作，可以像对其他任何变量一样给标志赋值，例如：

```
done = FALSE;
```

或

```
bool = TRUE;
```

也可以将任何值为布尔型的表达式赋值给一个布尔型变量。比如，某程序中要求只要变量 itemRemaining 的值为 0，立即结束一些特定的操作，那么可以用如下的方法给 done 赋予合适的值：

```
done = (itemRemaining == 0);
```

以上语句的意思是：“计算 itemRemaining == 0 的值，其结果可为 TRUE 或 FALSE，再将结果保存到变量 done 中”。

这里的括号并不是必须的，但常用它来强调，这是将一个条件测试的结果赋给一个变量。

尽管语句 `done = (itemRemaining == 0)` 足以将正确的布尔型值保存到变量 done 中，而初学者往往更倾向于用如下比较长的 if 语句来达到相同的效果：

```
if(itemRemaining == 0){
      done = TRUE;
}else{
      done = FALSE;
}
```

虽然这些语句能达到预期效果，但它们效率低且不够简明。这里，第二个版本用了五行代码完成了只需一行代码就能完成的功能，并使得程序长度无谓地增加。

在使用布尔型数据时，一定要清楚布尔值可以像其他任何类型的值一样直接用来赋值。用标志作为条件测试的一部分时也有类似的问题，例如要测试 done 是否为 TRUE 时，可以直接用

```
if(done)
```

而不是

```
if(done == TRUE) /* 这里==TRUE是多余的 */
  ...
```

尽管第二个表达式有同样结果，但它的等于测试是多余的，变量 done 的值本身就肯定是 if 语句要求的 TRUE 或 FALSE 之一，不需要测试 done 的值是否为 TRUE，而且这个额外的测试并不能提供任何新的有用信息。

常见错误：在使用布尔型的数据时，要小心避免冗余。主要要注意将一个布尔值与常量 TRUE 做比较，以及用一个 if 语句产生一个布尔型的结果值时，其实这个值已经可以作为条件表达式使用的情况。

下面以计算闰年为例，说明逻辑运算符在程序设计中的应用。

> 天文学家发现，地球绕太阳公转一周的时间比 365 天稍多一些。由于每公转一周约比 365 天多四分之一天，所以每四年就要在日历上加上一天，使那一年称为闰年。这样的调整使日历与天文年同步，但还是有一点偏差。为了使每一年的开始不会慢慢地与季节偏离，实际用来计算闰年的规则还要复杂一些。闰年每四年出现一次，除了以 00 结尾的年，以 00 结尾的年只有能被 400 整除时才是闰年，因此 1900 年虽能被 4 整除，但不是闰年，而 2000 年则是闰年。
> 要求编写程序解决：输入年份并确定它是否为闰年。

若为闰年，必满足下列两个条件之一：

1. 该年份能被 4 整除，但不能被 100 整除；
2. 该年份能被 400 整除。

假设年份的变量为 y，以下布尔表达式将给出正确结果：

```
((y % 4 == 0) && (y % 100 == 0)) || (y % 400 == 0)
```

若考虑 C 语言的优先级规则，这个表达式中的括号实际上都不需要，使用括号只是为了使较长的布尔表达式更容易阅读。

计算这个布尔表达式的值，并将它存储在一个叫 `isLeapYear` 的标志中，就可以在程序的其他部分通过测试标志变量得知闰年的条件是否成立。

```
/*
* File:leapyear.c
* --------------------
* This program will read the year and determine whether a year is leap year.
*/
#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
main()
{
  int year;
  bool isLeapYear;

  printf("Program to determine whether a year is a leap year.\n");
  printf("What year?");
```

```
  year = GetInteger();
  isLeapYear = ((year % 4 == 0) && (year % 100 == 0) ) || (year % 400 == 0);
  if(isLeapYear){
    printf( "%d is a leap year.\n" ,year);
  }else{
    printf( "%d is not a leap year.\n" ,year);
  }
}
```

20.5 If Statements

C 语言中表示条件执行的最简单的办法是使用 if 语句。if 语句有以下两种形式：

```
if(condition) statement
if(condition) statement1 else statement2
```

if 语句允许程序通过测试表达式的值来决定从两种选项中选择哪一种，其中条件（condition）部分是一个布尔表达式，语句（statement）部分可以是简单语句，也可以是程序块。

当解决方案需要在满足特定条件的情况下执行一系列语句时，可以用 if 语句的第一种形式。如果条件不满足，构成 if 语句主体的那些语句将被跳过。例如在 balance.c 中：

```
if(entry < 0 && balance < 0){
  printf( "This check bounces,$10 fee deducted.\n" );
  balance -= 10;
}
```

支票可能被退回也可能没有退回，而程序只在支票被退回的情况下才能执行这些动作。

当解决方案中必须根据测试的结果在两组动作中选择其一时，可以使用 if 语句的第二种形式。例如，在如下的程序 oddeven.c 中，该程序读入一个数，将它按奇偶分类，决定结果的条件表达式为：

```
n % 2 == 0
```

这个表达式首先计算 n 除以 2 的余数，并检查它是否为 0，这样就可以判断 n 的奇偶。如果是偶数，那么立即执行 if 的下一条语句，该语句会报告这个数是偶数，如果余数不为 0，则 n 为奇数，else 的下一行语句将报告这个数为奇数。

执行 if 语句时，先计算条件表达式的值，如果条件表达式的值为 TRUE（非零），所执行的程序块是 if 语句的 then 子句，条件为 FALSE（零）时执行的程序块是 else 子句。

通常，if 语句中的表达式能判定变量是否落在某个数值范围内，例如为了判定 $0 \le i < n$ 是否成立，最好写成：

```
if(0 <= i && i < n) ...
```

如果要判定相反的情况（即 i 在允许范围外），最好写成

```
if(i < 0 || i > n) ...
```

当 if 语句的 else 子句是可有可无时，有可能会造成歧义，这种情况叫做悬空 else 问题（dangling-else problem）。假设写了几个逐层嵌套的 if 语句，其中有些 if 语句有 else 子句而有些没有 else 子句，那么就很难判断其中的 else 子句是属于哪个 if 语句的。当遇到这个问题时，C 编译器采取一个简单的规则，即每个 else 子句属于离它最近的且还未和其他 else 匹配的 if 语句，即和它之前最近的一个 else 子句的 if 语句配对。

例如，在下面的示例中就产生了悬空 else 问题。

```
/*
* File:oddeven.c
* ------------------
* This program is used to classify a number as even or odd.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  int n;

  printf("Program to classify a number as even or odd.\n");
  printf("n = ?");
  n = GetInteger();
  if(n % 2 == 0){
    printf("That number is even.\n");
  }else{
    printf("That number is odd.\n");
  }
}
```

```
if(y != 0)
  if(x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0.\n");
```

根据 C 语言编译器采取的对悬空 else 问题的规则，正确的缩进格式如下：

```
if(y != 0)
  if(x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0.\n");
```

尽管这条规则对编译器来说是很简单的，但对人来说要快速识别 else 子句属于哪个 if 语句仍然是比较困难的。

通过采用比 C 语言基本要求更严格的、更规则的程序设计风格，可以避免悬空 else 造成的歧义。下面的规则概括了如何在 if 语句中使用程序块来消除这个问题：

> if/else 分块规则：对任何需要多行或需要 else 子句的 if 语句来说，都应该用大括号将在 if 语句控制下的各个独立的程序块括起来。

```
if(y != 0)
  if(x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0.\n");
```

这里，为了使 else 子句属于外层的 if 语句，可以把内层的 if 语句用大括号括起来如下：

```
if(y != 0){
  if(x != 0)
    result = x / y;
} else {
  printf("Error: y is equal to 0.\n");
}
```

使用 if/else 分块规则后，则 if 语句必可以以如下四种形式之一出现：

1. 单行 if 语句：条件极短，例如 break。
2. 多行 if 语句：语句被划分为一个程序块。
3. if-else 语句：将受控于 if 语句的语句划分为两个程序块，哪怕每块只有一条语句。
4. 级联 if 语句：表达一系列的条件测试。

20.5.1 Single-line if statements

当 if 语句没有 else 子句，而主体部分只有一条简单语句，而且短得可以与 if 本身放在一行时，可以采用简单的单行 if 语句格式，如以下所示。

```
语法：单行if语句
   if(condition) statement;
其中：
   condition是将要测试的布尔值。
   statement是当condition为TRUE时将要执行的一条简单语句。
```

在这种情况下，如果使用大括号并将 if 语句从一行扩展为三行，将使程序无谓地变长且不易阅读。这种类型的例子如下：

```
if(value == sentinel) break;
```

20.5.2 Multiline if statements

当 if 语句的主体有多条语句或有一条语句使之不能与 if 放在一行时，这些语句将被划分为一个程序块，如以下所示：

```
语法：多行if语句
   if(condition){
     statements;
   }
其中：
   condition是将要测试的布尔值。
   statement是当condition为TRUE时将要执行的一个语句块。
```

在这种情况下，如果条件为 TRUE，将执行这些语句；如果条件为 FALSE，程序将不做任何动作，继续执行 if 语句的下一条语句。

C 语言对可以出现在 if 语句内部的语句的类型没有限制。事实上，在 if 语句内部嵌套其他 if 语句是非常普遍的。

```
if(i > j)
  if(i > k)
    max = i;
```

```
  else
    max =k;
else
  if(j > k)
    max = j;
  else
    max = k;
```

if 语句可以嵌套任意层数,并以大括号进行分块达到手动指定编译顺序的目的。

```
if(i > j){
  if(i > k)
    max = i;
  else
    max =k;
} else{
  if(j > k)
    max = j;
  else
    max = k;
}
```

20.5.3 if/else statements

为避免悬空 else 问题,带有 else 子句的 if 语句主体将用大括号划分为两个程序块。

从技术上讲,只有当条件语句的主体包含不止一条语句时,才有必要在程序块前后加大括号,然而从系统地使用括号的角度上讲,加上括号会使程序产生混淆的情况大大减少,更有利于程序的维护。

if/else 语句的程序设计范例如下面所示:

```
语法: if/else语句
   if(condition){
     statementsT
   }else{
     statementsF
   }
其中:
   condition是将要测试的布尔值。
   statementsT是当condition为TRUE时将要执行的一个程序块。
   statementsF是当condition为FALSE时将要执行的一个程序块。
```

20.5.4 Cascading if statements

在设计程序时,常会遇到需要判定一系列条件的情况,一旦其中某一种条件为真就立刻停止,级联 if 语句常常是编写这些系列判定的最好办法。下面的语法展示了当实际有两种以上情况时,if 语句的一个十分重要而特殊的用法。

这种方式的特征在于 if 语句的 else 部分是由对另一个可选条件的测试构成的,这样的语句叫做级联 if 语句(cascading if statement)。

级联 if 语句并不是新的语句类型,它仅仅是普通的 if 语句,只是把另一条 if 语句作为 else 子句的替换。下面的示例说明了使用级联 if 语句来判定 n 是小于 0,等于 0,还是大于 0 时的输出。

```
if(n < 0)
```

语法：级联if语句

```
if(condition1){
  statements1
}else if(conditon2){
  statements2
}else if(condition3){
  statements3
}else{
  statementsnone
}
```

其中：

每个conditioni是一个布尔表达式。

每个statementsi是当条件conditioni为TRUE时将要执行的一个程序块。

statementsnone是当所有条件都为FALSE时将要执行的一个程序块。

```
  printf("n is less than 0.\n");
else
  if(n == 0)
    printf("n is equal to 0.\n");
  else
    printf("n is greater than 0.\n");
```

对于级联 if 语句，通常可以不进行缩进，而是把每个 else 都与最初的 if 对齐：

```
if(n < 0)
  printf("n is less than 0.\n");
else if(n == 0)
  printf("n is equal to 0.\n");
else
  printf("n is greater than 0.\n");
```

级联 if 语句中可以有任意个 else if 子句。在很多情况下，使用 switch 语句处理选择多个独立情况的问题会更加高效。

在下面的程序 signtest.c 中，就是利用级联 if 语句判断一个数是正数、零还是负数。

在程序中，没有必要显式地检查 n < 0 的情况，因为倘若程序能够到达最后一个 else 子句时，说明之前的检查已经排除了正数和零的情况，不可能还有其他请，它只能是负数，而在使用 switch 语句时，则用另一种方式，即 default 子句来处理这样的情况。

在很多情况下，使用 switch 语句处理选择多个独立情况的问题会更加高效。

20.5.5 ?: Operator

C 程序设计语言中提供了另一种更加简练的用来表达条件执行的机制——**?:** 运算符。在某些情况下，**?:** 机制会非常有用。

?: 运算符又被称为问号冒号，但实际上，这两个运算符并不紧挨着出现。和 C 中的其他运算符不同，**?:** 在运用时分成两部分且带有三个操作数，它的一般形式为：

```
(condition) ? expression1 : expression2
```

条件运算符是 C 语言中唯一一个需要三个操作数的运算符，因此又称为三元运算符（ternary operator）。其中，加在条件上的括号从技术上讲是不需要的，但是可用括号来突出测试条件的边界。

当 C 编译器遇到**?:** 运算符时，首先计算条件值，如果条件值为 TRUE，则计算 expression1 的值，并将它作为整个表达式的值；如果条件结果为 FALSE，则整个表达式的值为 expression2 的值。

可以把?: 运算符看作以下 if 语句的缩略形式：

```
/*
* File:signtest.c
* ------------------
* The program can classify a name by ite sign.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  int n;

  printf( "Program to classify a number by its sign.\n" );
  printf( "n = ?" );
  n = GetInteger();
  if(n > 0){
    printf( "That number is positive.\n" );
  }else if(n == 0){
    printf( "That number is zero.\n" )
  }else{
    printf( "That number is negative.\n" );
  }
}
```

```
if(condition){
  value = expression1;
}else{
  value = expression2;
}
```

其中，存储在变量 value 中的值即为整个**?:** 表达式的值。例如，可以用?: 运算符将 x 和 y 中值较大的一个赋给 max，原程序代码如下：

```
if(x > y)
  return x;
else
  return y;

return (x > y ? x : y);
```

使用?: 运算符的一种最常见的情况是调用 printf 进行输出时，输出结果可能因为某个条件而略有不同。比如编写一个程序统计某物品的数量，并在统计结束后将物品的数量存储在变量 nItems 中，在将结果报告给用户时，最直接的方法就是用如下的 printf 语句进行输出。

```
printf("%d items found.\n",nItems);
```

但是考虑到英语中单复数的情况，当 nItems 的值为 1 时，应为 nItem，虽然也可以通过在 if 语句包含两个 printf 语句来更正这个英语语法错误。

```
if(nItems > 1){
  printf("%d items found.\n",nItems);
}else{
```

```
    printf("%d item found.\n",nItems);
}
```

但却发现用了五行语句，而其实这可以用?: 运算符来解决。

```
printf("%d item%s found.\n", nItems, (nItems > 1) ? "s" : " ");
```

输出时，如果 nItems 的值大于 1，则在字符串 item 后加上一个字符串 s，反之就加上一个空字符串。

条件表达式也普通用于某些类型的宏定义中，例如可以用?: 运算符以自然语言方式输出布尔变量的值。许多程序需要变量能存储假值或真值，但是 C 语言缺少适当的布尔类型，C++ 认识到这个问题后增加了内建的布尔类型。

由于 bool 类型不是标准 C 语言的一部分，因此没有内建的输出布尔型值的机制，不过暂时可以采用模拟布尔型变量的办法来解决这个问题。这种模拟的办法是先声明 int 型变量，然后将其赋值为 0 或 1，示例代码如下：

```
int flag;
flag = 0;
...
flag = 1;
```

上述办法虽然可行，但因为没有明确地表示 flag 的赋值只能是布尔值，也没有明确指出 0 和 1 就是表示假和真，因此对于程序的可读性没有帮助。为了使程序便于理解，一个好的办法是用类似 TRUE 和 FALSE 来定义宏。

```
#define TRUE 1;
#define FALSE 0;
```

这样，对 flag 的赋值就有了更加自然的形式如下：

```
flag = FALSE;
...
flag = TRUE;
```

- 为了判定 flag 是否为真，可以写成：`if(flag == TRUE) ...` 或者只写成 `if(flag) ...`
- 为了判定 flag 是否为假，可以写成：`if(flag == FALSE) ...` 或者只写成 `if(!flag) ...`

下面使用 `printf` 和?: 来显示布尔型变量 errorFlag 的值：

```
printf("errorFlag = %s\n",(errorFlag) ? "TRUE" : "FALSE");
```

为了在 C 语言中更好的模拟布尔类型，可以定义用作类型的宏如下：

```
#define BOOL int;
```

这样，声明布尔型变量时可以用 BOOL 代替 int。

```
BOOL flag;
```

虽然在编译时，C 语言编译器始终把 flag 看成是 int 变量，但在编写程序时 flag 可以表示布尔条件。

在 C 语言中，可能会过度使用?: 运算符，如果一个程序中重要的判断结果隐藏在这种嵌套的?: 运算符中，后面的代码很可能就会遗漏这个判断。另一方面，如果可以使用?: 代替复杂的 if 语句来处理一些小的细节，那么?: 将会大大简化程序的结构。

> 当 int 型变量和 float 型的值混合在一个条件表达式中时，?: 表达式的类型为 float 型。

20.6 Switch Statements

当一个程序逻辑上要求根据特定条件做出真假判断并执行相应的动作时，if 语句是理想的解决方案。然而，还有一些程序需要更复杂的判断结构，它们会与两个以上的可选项进行比较，这些可选项

被划分为一系列互斥的情况。比如，在第一种情况下，程序应该执行 x 步骤；在另一种情况下，程序应该执行 y 步骤；在第三种情况下，程序应该执行 z 步骤，等等。

在这种类型的应用中，使用级联 if 语句可以达到需求，但最适合处理这种情况的是 switch 语句[3]，它的语法如下：

```
Syntax: switch statement
switch(e){
  case c1:
    statements1
    break;
  case c2:
    statements2
    break;
  . . . more cases clauses . . .
  default:
    statements_def
    break;
}
```

其中：

- `e` 是决定执行什么语句的控制表达式；
- 每个 `ci` 是一个常量；
- 每个 `statementsi` 是当 `ci` 等于 `e` 时执行的语句序列；
- `statements_def` 是当任何 `ci` 都无法与 `e` 匹配时执行的语句序列。

switch 语句的起始行是：

```
switch(e)
```

其中 e 是一个表达式，称为控制表达式(control expression)。C 语言把字符当成整数来处理，因此可以在 switch 语句中对字符进行判定，但不能用浮点数和字符串。

switch 语句的主体分成许多独立的由关键词 case 或 default 引入的语句组。

- 每个 case 关键词和紧随其后直到下一个 case 或 default 之前的所有语句合称为 case 子句；
- default 关键词及其相应语句合称为 default 子句。

例如，在语法框中的以下语句构成了第一个 case 子句。

```
case c1:
  statements1
  break;
```

当程序执行到 switch 语句时，首先计算表达式 e 的值，然后依次与 c1、c2 等进行匹配。每一个 case 子句所用的值(这里的 c1、c2 等)必须是某种标量类型的常量(例如整型常量，或者是行为类似于整型值的任何值，如字符型)，不能包含变量或函数引用。

如果其中一个常量与控制表达式的值匹配成功，则执行相应 case 子句中的语句。当程序到达子句结尾的 break 语句时，该子句指定的操作已经结束，程序将从整个 switch 结构后的语句继续执行。如果没有一个 case 常量与控制表达式的值匹配，那么将执行 default 子句的语句。

- C 语言不允许有重复的 case 子句，但对 case 子句的顺序没有限制，特别是 default 子句不一定要放置在最后。
- 在 switch 语句中的 break 子句，并非 C 语言的语法所必须的，这里建议将 break 语句作为 switch 语法的一部分。

[3]switch 语句比级联 if 语句更易阅读，而且 switch 语句往往比 if 语句执行速度快。

- 在 C 语言的定义中，子句中如果没有 break 语句，程序将在执行完所选 case 子句之后执行紧跟其后的 case 子句等。

虽然这种设计在某些情况下是有用的，但它解决的问题远比上它所带来的麻烦，没有在 switch 语句的每个子句后加 break 语句是程序出错的常见原因。

一般情况下，所有的 case 子句都将以 break 语句结尾（有时也会用 return），执行 break 语句会导致程序“跳”出 switch 语句，从而可以将控制转移到整个 switch 语句后面的语句。

switch 语句中需要 break 语句的原因在于 switch 语句实际上是一种“基于计算的跳转”。当计算控制表达式的数值后，控制跳到与 switch 表达式的值相匹配的 case 子句处。情况标号只是说明 switch 内部位置的标记，在执行完 case 子句中的最后一条语句后，程序控制“向下跳转”到下一种情况的第一种语句上，而忽略下一个 case 子句的情况标号。

如果没有 break（或其他的跳准语句）语句，程序控制将会从一种情况继续到下一种情况。switch 语句只有一种例外的情况，允许多个指定不同常量的 case 关键词连续出现在同一组语句之前，比如 switch 语句可能会出现以下形式：

```
case 1:
case 2:
  statements
  break;
```

它意味着，当控制表达式取值为 1 或 2 时将执行同样的语句。C 编译器将这种结构看作是两个 case 子句，只不过第一个子句是空的。由于空子句没有 break 语句，当程序选择这条路径时，将继续执行第二个子句。从概念的角度看，最好还是不要将这种结构看作是一个代表两种可能的 case 子句。

在 switch 语句中，default 子句是可选的。假如没有 default 子句，当任何 case 子句都未匹配成功时，程序会从 switch 语句退出，不执行任何动作，转而继续执行 switch 结构之后的语句。

为了避免程序忽略一些意外情况，在每个 switch 语句中都使用 default 子句是一种良好的程序设计习惯，除非确定已经在 case 语句中列举了所有的可能情况。

由于 switch 语句通常可能会很长，如果 case 子句本身较短，那么程序比较容易阅读。如果有足够的空间将 case 关键字、子句的语句和 break 语句放在同一行会更好。

下面的程序 cardrank.c 就是这种 switch 语句风格的一个例子，它是 switch 语句在一个玩牌程序中的应用。在这个游戏中，每一组牌用数字 1-13 代表，但当要显示这些牌时就会遇到问题，2-10 可以直接显示数字，但 1、11、12 和 13 不能直接显示数字，而应该用 Ace、Jack、Queen 和 King 来表示，因此程序 cardrank.c 用 switch 语句为每张牌显示适当的符号。

```
/*
* File:cardrank.c
* --------------------
* This program will prints the rank of the card(1-13) and not change its name.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  int n;

  printf("What is the rank of the card(1-13)?");
  n = GetInteger();
  switch(n){
    case 1:  printf("Ace\n"); break;
    case 11: printf("Jack\n"); break;
    case 12: printf("Queen\n"); break;
```

```
    case 13: printf("King\n"); break;
    default: printf("%d\n", n); break;
  }
}
```

switch 语句只能在用整型或类似整型的常量标识的 case 子句中选择，这大大限制了它的使用。在实际编程中，常会用字符串常量或者变量作为 case 子句的指示符。由于 switch 语句不能在这些情况下使用，所以只能用级联 if 语句替代，然而在条件允许的情况下，使用 switch 语句会使程序更具可读性，更高效。

虽然 switch 语句中最后一个 case 子句不需要 break 语句，但常见的额做法还是放一个 break 语句来提醒若增加 case 子句不要出现“丢失 break”的问题。

20.7 While Statements

循环是重复执行某些其他语句(循环体)的一种语句。每个循环都有一个控制表达式，每次执行循环体(循环重复一次)时都要对控制表达式进行计算。如果表达式为真，也就是值不为零，那么继续执行循环。

在 C 语言中，最简单的迭代结构就是 while 语句，它重复执行一条简单语句或程序块，直到条件表达式变为 FALSE 为止。

while 语句的范例如下面所示：

```
Syntax:while statement
  while(condition expression){
    statements
  }
其中:
  condition expression是用来决定循环是否继续开始下一个周期的条件测试;
  statements是将要重复执行的语句。
```

与 if 语句一样，当循环体只由一条简单语句构成时，C 编译器允许去掉加在循环体两边的大括号。为了增强程序的可读性，在所有的 while 循环体两边都加大括号，这样就可以将循环体看作一条语句(只是没有分号结尾)。

整条语句(包括 while 控制行本身和循环体)一起构成了一个 while 循环，while 循环的运行情况如下：

- 程序执行 while 语句时，先计算出条件表达式的值，看它是 TRUE 还是 FALSE。
- 如果是 FALSE(假)，循环终止(terminate)，并接着执行在整个 while 循环之后的语句。
- 如果是 TRUE(真)，将执行整个循环体，而后再回到 while 语句的第一行，再次对条件进行检查。

对循环中所有语句的一次执行，称作一个周期(cycle)。

很多程序会使用到“倒数计数”的功能，下面的示例实现了一个简单的“倒数计数”功能。

```
i = 10;
while(i > 0){
  printf("T minus %d and continuing\n", i);
  i--;
}
```

这里通过“倒数计数”的示例可以引发对 while 语句的一些讨论如下：

- 在 while 循环终止时，控制表达式的值为假，否则将继续执行循环。
- 控制表达式是在循环体执行之前进行判定的，因此可能根本不执行 while 循环体。如果第一次进入“递减计数”循环时的变量 i 的值是负数或零时，那么将不会执行循环。

- while 语句可以有多种写法。例如在 printf 函数调用的内部进行变量 i 的自减操作的简明示例代码如下：

```
while(i > 0) printf("T minus %d and continuing\n", i--);
```

在考察 while 循环的操作时，有两个很重要的原则：

1. 条件测试是在每个循环周期之前进行，包括第一个周期，如果一开始测试结果便为 FALSE，则根本不会执行循环体。
2. 对条件的测试只在一个循环周期开始时进行。如果恰好条件值在循环体的某处变为 FALSE。程序在整个周期完成之前都不会注意它。在下一个周期开始前再次对条件进行计算，如果为 FALSE，则整个循环结束。

一种 while 循环的使用方法是专为标志检测设计的，这只代表一种特殊情况。为了说明 while 语句的普遍用法，需要找出一个问题，使它的条件测试很自然地出现在循环开始的地方。

问题：计算一个数字中各位数的和。

算法：首先需要定义一个变量存储总和，将它初始化为 0。通过循环累加各位数字，最后输出结果。

程序的大体结构如下：

```
main()
{
  int n, dsum;

  printf( "This program sums the digits in an integer.\n" );
  printf( "Enter a positive integer:" );
  n = GetInteger();
  dsum = 0;

  /* 将整型数中的各位上的数字加入dsum */
  printf( "The sum of the digits is %d\n" , dsum);
}
```

这里，语句：

```
将整型数中的各位上的数字加入 dsum
```

清楚地说明了一个循环结构，因为整型数中的每一位都要重复一定的操作。如果能很容易地得到整型数的位数，便可以采用 for 循环来累加各位数，但是找出一个整型数的位数的难度与直接把它的各位数相加一样，这个程序最好的解决办法是不停的将整型数的每一位上的数字加到总和中，直到已经加入最后一个为止。让循环一直执行，直到一个条件发生的情况，常用 while 语句来编码。

这个问题的本质在于如何将一个整型数的各位数字拆分开来，关键是要想到用/和% 这两个算术运算符可以达到目的。因为任何一个整数的最后一位恰好是它除以 10 所得到的余数，写成表达式为 n%10，而 n/10 得到的是整数 n 除了最后一位的其他部分。

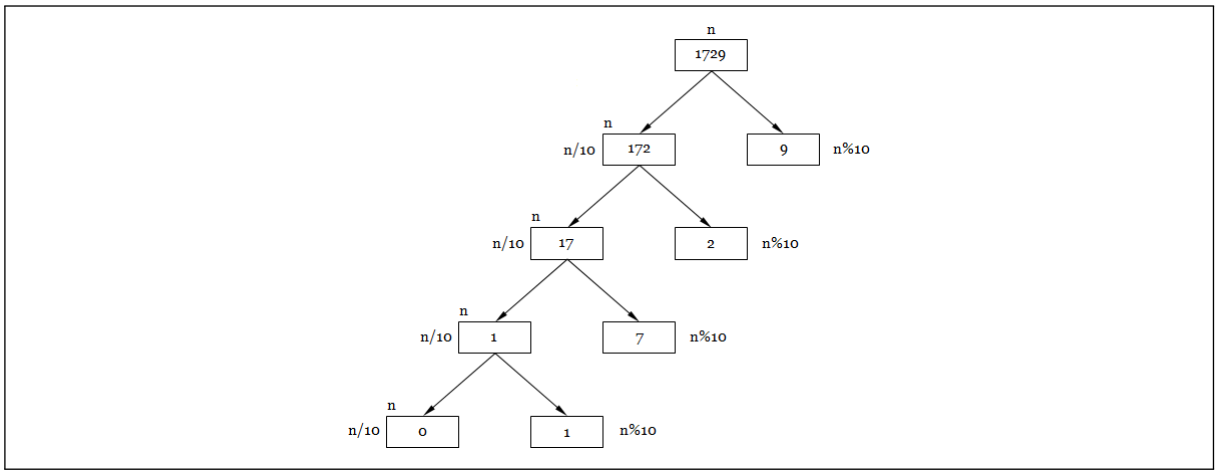
下面以 1729 为例，以/和% 按照二叉树的形式分解得到：

因此为了求这个整数的各位数字的和，需要做的是在循环的每一个周期中将 n%10 的值加到变量 dsum 中，再将整数 n 除以 10，下一个周期把原数的十位上的数字加到和中。这样反复执行，直到整数的每一位都处理完毕为止。

问题在于如何确定何时结束，事实上，每个周期中都将 n 除以 10，最终将使 n 变成 0，此时已经处理完该整数的所有数位，也就可以退出循环了。换句话说，只要 n 的值大于 0，就应继续循环，所以，该程序中的 while 语句应该如下所示：

```
while(n > 0){
  dsum += n%10;
  n /= 10;
}
```

从而得到完整的程序如下：




```
/*
* File:digitsum.c
* ------------------
* This program sums the digits in an integer.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
  int n, integer;

  printf( "This program sums the digits in an integer.\n" );
  printf( "Enter an positive integer:" );
  n = GetInteger();
  dsum = 0;
  while(n > 0){
    dsum += n%10;
    n /= 10;
  }
  printf( "The sum of the digits is %d\n" , dsum);
}
```

在 while 循环中，有时为了缩短循环会简化比较表达式，例如：

```
/* Origin while statement */
while(i > 0) printf("T minus %d and continuing\n", i --);
/* Simplified while statement */
while(i) printf("T minus %d and continuing\n", i --);
```

显而易见，新写法更加简洁，但它也缺点。首先，虽然新循环可以清楚地显示出在 i 达到 0 值时循环终止，但不能清楚地说明是向上计数还是向下计数，而在原来的写法中根据控制表达式 i>0 可以推导出向上还是向下的信息。其次，如果循环开始执行时 i 为负值，那么新循环的行为会不同于原始版本，原始版本循环会立刻停止，而新循环则不会。

20.7.1 Infinite Loop

当在程序中使用循环时，一定要确保循环的控制条件最终会变为 FALSE，以便能推出循环。

如果 while 循环的控制行中的条件总是为 TRUE，那么计算机将一个周期接着一个周期不停地执

行下去，这种情况叫做无限循环（infinite loop）。例如将 digitsum.c 的 while 循环的控制条件中的运算符“>”写成“>=”，如下所示：

```
while(n >= 0){
  dsum += n%10; /*循环将不停地执行下去*/
  n /= 10;
}
```

在上述的逻辑错误中，当 n 减到 0 时，循环不会结束，相反，计算机将会一遍又一遍地执行循环体，因为每次 n 都等于 0。

常见错误：仔细斟酌在 while 循环中使用的条件表达式，确保循环最终会结束。永远不会结束的循环称为无限循环。

事实上，可能会故意用非零常量作为控制表达式来构造无限循环。例如在 Nginx 中，所有实际上的业务处理逻辑都在 worker 进程。worker 进程中有一个函数执行无限循环，不断处理收到的来自客户端的请求并进行处理，这种过程直到整个 nginx 服务被停止。Worker 中这个函数执行内容如下：

1. 操作系统提供的机制（例如 epoll、kqueue 等）产生相关的事件。
2. 接收和处理这些事件，如果接受到的是数据，则产生更高层的 request 对象。
3. 处理 request 的 header 和 body。
4. 生成响应，并发送回客户端。
5. 完成 request 的处理。
6. 重新初始化定时器及其他事件。

无限循环的一般形式如下：

```
while (1) ...
```

传统上的无穷循环还包括 for(;;)，原因在于早期的编译器经常强制程序在每次执行 while 循环时测试条件 1，不过对现代编译器来说，在性能上这两种无限循环没有差别。

要结束一个无限循环，可以在循环体内包含跳出循环控制的语句（break、goto、return）或者调用了导致程序终止的函数（包括从键盘上输入特殊的命令以中断程序执行并强制退出），这个特殊的命令因机器的不同而不同。

20.7.2 Loop-and-a-half Problem

适合使用 while 循环的情形是在要重复进行的操作之前有一些条件测试，即在循环体的任何语句执行之前要有条件测试。如果要解决的问题符合这个结构，那么 while 循环是一个很好的工具。

但是，许多问题并不能简单地套入标准的 while 循环来加以解决，有些问题不能在操作开始时作一个常规的测试，决定循环是否该结束的测试逻辑会自然而然地出现在循环中间。

考虑一个具体例子，读入数据直到读到标志值的问题。如果用自然语言描述，基于标志的循环的结构由如下步骤组成：

1. 读入一个值；
2. 如果读入的值与标志值相等，则退出循环；
3. 执行在读入那个特定值的情况下需要执行的语句。

实际上，在循环的最开始没有能决定循环是否该结束的测试时，循环的结束条件要等到读入值等于标志值时才出现。为了检查这个条件，程序必须先读入某个值。如果程序还没有读到值，那么终止条件没有任何意义。在程序能做任何有意义的测试之前，程序必须已经执行了循环中读入数据的部分。当一个循环中有一些操作必须在条件测试之前执行时，它就是所谓的半途退出问题（loop-and-a-half problem）。

C 语言解决半途退出问题的一个方法是使用 break 语句。break 语句除了能在 switch 语句中使用之外，它还有使最内层的循环立即终止的作用。

通过 break 语句就能用类似下面这种自然的结构来编写标志问题中的循环。

```
while(TRUE){
  提示用户输入并读入数据
  if(value == sentinel) break;
  根据数据作出处理
}
```

对于第一行的 while(TRUE) 的意义，按照 while 循环的定义，循环将一直执行直到括号里的条件值变为 FALSE 为止，而 TRUE 是一个常量，它不会变成 FALSE，因此就 while 语句本身而言，循环永远不会结束，程序退出循环的唯一方法就是执行其中的 break 语句。

不用 while(TRUE) 控制行或 break 语句，也能编写这样的循环，但是要这样做的话，需要改变循环内的一些语句的顺序，请求输入数据的语句将在两个地方出现，一次在循环开始之前，一次在循环体内。

当需要“提示用户并读入第一个数据”时，采用这种结构的基于标志的循环的范例如下：

```
while(value != sentinel){
  根据数据作出处理
  提示用户并读入新数据
}
```

下面的程序展示了如何用以上范例而不用 break 语句实现 addlist.c 程序。

```
/*
* File:addlist1.c
* ---------------
* This program adds a list of numbers. The end of the input is indicated by
* entering 0 as a sentinel value.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
* About sentinel
* ------------------
* A new method of setting sentinel instead of using break notice.
*/

/* Main Program */

main()
{
  int value, total;

  printf("This program adds a list of numbers.\n");
  printf("Signal end of list with a 0.\n");
  total = 0;
  printf("?");
  value = GetInteger();
  while(value != 0){
    total += value;
    printf("?");
    value = GetInteger();
  }
  printf("The total is %d\n", total);
}
```

但是这个方法有两个不足。首先，循环中操作的顺序与人们所期望的顺序不同，用自然语言来解

释这个算法时，第一步要得到一个数，第二步是把它加到总数中，而 addlist1.c 程序中的 while 循环颠倒了这两个语句在循环中的顺序，使程序难以读懂。其次，范例中要求读入数据的语句出现两次，重复的代码会带来一系列维护上的问题，因为对其中一部分的修改不会改变另外一些副本。

在 C 语言中，当执行 break 语句时，控制结构立即跳出最近的 switch、while、for 或 do 语句，而且很多经验也表明，合理使用 break 语句来解决半途退出问题比不用 break 语句更能正确地编写程序[4]。

只是在使用 break 来解决半途退出问题的过程中往往会造成 break 语句的滥用，防止滥用 break 语句的一个办法就是彻底禁用。因此，尽量地仅在解决 while 循环的半途退出问题时使用 break 语句，而在其他容易引起程序结构模糊的情况下不使用 break 语句。

20.8 For Statements

for 语句在将一个操作重复执行特定次数的情况下使用，下面给出了 for 语句的一般形式：

```
语法：for语句
  for(init; test; step){
    statements
}
其中：
    init（初始表达式）是初始化循环时计算的表达式；
    test（测试表达式）是一个条件测试，决定循环是否继续，就像while语句一样；
    step（步长）是为下一个循环周期做准备的表达式；
    statements是要被重复执行的语句。
```

for 循环的操作是由 for 控制行中的三个表达式 init、test 和 step 决定的。

- init 表达式指出 for 循环应该如何初始化，通常用于设置下标变量的初始值。例如，语句 `for(i = 0; ...)` 表示循环开始时将下标变量 i 初始化为 0，而如果循环是 `for(i=-7; ...)` 表示循环开始时将下标变量 i 初始化为-7，以此类推。
  下标变量在每个循环周期中都会被更新。
- test 表达式是和 while 语句中的控制条件要求一样的条件测试，只要测试表达式的值为 TRUE，循环就会继续。因此，下面的语句：

  ```
  /* 从0向上递增到n-1 */
  for(i = 0; i < n; i ++) ...
  ```

  说明该循环开始时 i 的值为 0，当 i 小于 n 时继续下一个循环，它最终共有 n 个周期，其中 i 的值依次为 0、1、2、...、n-1，而循环

  ```
  /* 从1向上递增到n */
  for(i = 1; i <= n; i ++) ...
  ```

  说明该循环开始时 i 的值为 1，当 i 小于等于 n 时继续下一个循环，这个循环最终也将执行 n 个周期，其中 i 的值为 1、2、3、...、n。
  另外，for 控制行还可以以递减的方式执行，如下面的示例所示：

  ```
  /* 从n-1向下递减到0 */
  for(i = n-1; i >= 0; i --) ...
  /* 从n向下递减到1 */
  for(i = n; i > 0; i --) ...
  ```

- step 表达式表示下标变量的值在各个周期中如何变化，最常见的 step 规格说明的形式就是用 ++ 运算符自增下标变量的值，但这并不是唯一形式，例如可以用自减运算符--或用 `+=2` 而不是 `++` 表示每次加 2。

[4]证实这个发现的最著名的研究是 Elliot Soloway、Jeffrey Bonar 和 Kate Ehrlich 合著的 *Cognitive Strategies and Looping Constructus: an Empirical Study*（发表于 *Communications of the ACM*，November 1983）

for 语句适合应用在使用“计数”变量的循环中，然而它也可以用于其他类型的循环中。下面的例子中用反向计数将下标变量从 10 递减到 0，也说明任何变量都可以作为下标变量。

在这个例子中，下标变量叫做 t 很可能是因为它表示火箭点火升空前倒数的 10 秒的变量。

```
/*
* File:liftoff.c
* --------------
* Simulates a count down for a rocket launch.
*/

#include <stdio.h>
#include  “genlib.h”

/*
* Constant:startingCount
* ----------------------------
* Change this constant to use a different starting value for the countdown.
*/

#define startingCount 10

/* Main Program */

main()
{
  int t;

  for(t = startingCount; t >= 0; t --){
    printf( “%2d\n” ,t);
  }
  printf( “Liftoff!\n” );
}
```

下标变量和其他变量一样，在任何情况下都要在程序的开始部分对它进行声明。

for 语句和 while 语句关系紧密，而且某些情况下，for 语句与 while 语句是可以相互转换的。例如，for 语句的一般形式可以转换为：

```
init;
while(test){
   statements;
   step;
}
```

表达式 init、test 和 step 中的每一个都是可选的，但它们中间的分号是必需的。

- init 代表初始化步骤，并且只执行一次；
- test 用来控制循环的终止；
- step 是在每次循环的最后被执行的一个操作。

如果 init 不出现，则将不执行任何初始化操作，如果 test 不出现，则将假设条件测试为 TRUE。如果 step 不出现，则在循环的周期内将不执行任何操作，这样控制行就成了：

```
for(; ;);
```

它等效于：

```
while(TRUE)
```

虽然大多数 for 循环可以利用标准模式转换成 while 循环，但当 for 循环体中含有 continue 语句时，while 语句将不再有效。下面的两个循环并不等价，原因在于当 i 等于 0 时，原始循环并没有对 n 进行自增操作，但是新循环却做了。

```
/* origin while loop */
n = 0;
sum = 0;
while(n < 10){
  scanf("%d", &i);
  if(i == 0) continue;
  sum += i;
  n ++;
}
/* for loop */
sum = 0;
for(n = 0; n < 10; n ++){
  scanf("%d", &i);
  if(i == 0) continue;
  sum += i;
}
```

for 语句的灵活性在于其可能不需要所有的表达式，因此 C 语言允许忽略 for 控制行中的任何一个或全部的表达式，只保留控制行中的分号。

- 如果忽略了 for 控制行的 init 表达式，可以由一条单独的赋值语句初始化；
- 如果忽略了 for 控制行的 step 表达式，循环体就有责任确认 test 表达式的值最终会变为假；
- 如果同时忽略了 for 控制行中的 int 和 step 表达式，循环的结果和 while 语句没有任何却别；
- 如果忽略了 for 控制行中的 test 表达式，那么它默认为真，for 语句进入无限循环。例如，可以用下列的 for 语句建立无限循环：

```
for( ; ; ) ...
```

C 语言的 for 语句比其他相似编程语言的 for 语句功能要强大，但也会带来更多潜在的问题，原因在于 C 语言对控制循环行为的控制行表达式没有任何限制。虽然这些表达式通常对同一个变量进行初始化、判定和更新，但没有要求它们之间以任何方式进行关联。

如果 for 语句中有两个以上的初始表达式，或者希望在每次循环时一次对几个变量进行自增操作，可以使用逗号表达式(comma expression)作为 for 语句控制行中的 inti 或 step 表达式。

逗号表达式的计算要经过两步来实现。

1. 第一步，计算第一个表达式并且扔掉计算出的值。
2. 第二步，计算第二个表达式并把这个值作为整个逗号表达式的值。
3. 计算第一个表达式始终会有副作用，如果没有，那么第一个表达式就没有了存在的意义。

C 语言提供逗号运算符是为了在单独一个表达式的情况下使用两个或多个子表达式。换句话说，逗号运算符允许将两个表达式“粘贴”在一起构成单独的一个表达式。不会经常需要把多个表达式粘贴在一起，某些宏的类型可以从逗号表达式中受益。

逗号表达式和复合语句的相似之处在于，复合语句允许用户把一组语句当作唯一的一条语句来使用。

for 语句是唯一除上述之外还可以发现逗号运算符的地方，例如下面语句：

```
sum = 0;
for(i = 1; i <= N;i ++){
  sum += i;
}
```

等同于：

```
for(sum = 0, i = 1; i <= N; i ++){
  sum += i;
}
```

表达式 `sum = 0, i = 1` 首先把 0 赋值给 sum，然后把 1 赋值给 i，这样 for 语句就可以初始化两个变量。

C 语言中 for 语句的灵活性在链表中会获得极大的收益，但是 for 语句也很容易被误用，应该清楚地表达 for 语句的循环控制行表达式。

20.8.1 Nested For Loop

当程序变得越来越复杂时，常常需要将一个 for 循环嵌入到另一个 for 循环中。在这种情况下，内层的 for 循环在外层循环的每一次迭代中都将执行它的所有的周期。每个 for 循环都要有一个自己的下标变量以避免下标变量间的相互干扰。下面的 timestab.c 是一个嵌套 for 循环的例子：

```
/*
* File:timestab.c
* ---------------
* Generates a multiplication table where each axis runs from
* LowerLimit to UpperLimit.
*/

#include <stdio.h>
#include  "genlib.c"

/*
* Constants
* ------------
* LowerLimit --   Starting value for the table
* UpperLimit --   Final value for the table
*/

#define LowerLimit 1
#define UpperLimit 10

/* Main Program */

main()
{
  int i, j;

  for(i = LowerLimit; i <= UpperLimit; i ++){
    for(j = LowerLimit; j <= UpperLimit; j++){
      printf( "%4d" ,i*j);
    }
    printf( "\n" );
  }
}
```

程序 timestab.c 显示了 10×10 的乘法表，外层 for 循环用 i 作为下标变量，控制表的行的变化。在每一行中，内层 for 循环控制该行中列的变化，并显示每一个条目的值 i*j(即行号乘以列号)。注意，用来使光标移到下一行的 `printf("\n")` 调用是出现在外层循环中的，因为它只有每一行结束时才应被执行一次，而不是在显示一行中每一个值后就执行。

事实上，以下的 for 语句和 while 语句是等效的。

```
for(init;test;step){
  ...
}
```

```
init;
while(test){
  statements;
  step;
}
```

Table 20.4: `for` 和 `while` 的关系

尽管 for 语句可以很轻易地用 while 改写，但在适当的情况下使用 for 是大有好处的。在 for 语句中，所有需要了解的关于将要被执行的循环周期的信息都包含在语句的控制行中，例如，当在某个程序中看到如下语句时：

```
for(i = 0; i < 10; i ++){
  ... body ...
}
```

可以很清楚地知道循环体中的语句将被执行 10 次，对应的 i 值从 0 ～ 9。等效的 while 循环是：

```
i = 0;
while(i < 10){
  ... body ...
  i ++;
}
```

其中循环底部的自增操作在循环体很大时很可能会遗漏。

由于 for 循环中的 init、test 和 step 部分可以是任何表达式，所以没有任何明显的理由要求 for 循环的下标值必须是整数，在 for 循环中下标从 0 ～ 10，每次自增 2 也是完全可以的。代码如下：

```
for(i = 0; i <= 10; i += 2)
  ...
```

这意味着也应该可以将 x 声明为 double 型，再将它从 1.0 起每次自增 0.1 直到 2.0，代入如下：

```
for(x = 1.0; x <= 2.0; x += 0.1) /* 这个测试很可能会失败 */
  ...
```

在有些机器上，这条语句可以得到预期结果，而在另一些机器上，可能会得不到最后一个值。例如下面这个 for 循环：

```
for(x = 1.0; x <= 2.0; x += 0.1){
  printf( "%.1f\n" ,x); /* 这个循环可能会不显示2.0 */
}
```

当它在某些计算机上运行时，会产生以下输出：

```
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
```

注意，从循环控制看来，本应显示的值 2.0 丢弃了。

问题出在浮点数并不是百分之百精确的，比如浮点型值 0.1 很接近于数学上的分数 $\frac{1}{10}$，但它们并不是精确地相等。每当下标 x 加 0.1 时，误差便被累计，到最后与 2.0 比较来决定是否结束循环时，x 的值可能就成了 2.00000001 或类似的不是小于或等于 2.0 的数，这样就导致 for 循环的条件不满足，则它在少运行一个周期的情况下就退出了。解决这个问题的最好办法就是限制只能用整型数作为 for 循环的变量，因为整型是精确的，不会造成上述问题。

假如一定要从 1.0 累加 0.1 到 2.0，可以采用从 10 累加到 20，再将它除以 10：

```
for(i = 10;i <= 20; i ++){
  x = i/10;
  printf( "%.1f\n" ,x);
}
```

这个 for 循环正确地产生 1.0、1.1、1.2、...、2.0 这 11 个数。

除了 for 循环外，在其他一些情况下同样需要警惕比较浮点型数据会引发的问题。考虑到浮点型数据在特定计算机内存储精度限制，一些看起来相等的数字可能并不完全相等。

运算符	结合性	优先级
一元 - ++ -- ! (类型转换)	从右到左	从高到低
* / %	从左到右	
+ -	从左到右	
< <= > >=	从左到右	
== !=	从左到右	
&&	从左到右	
||	从左到右	
?:	从右到左	
= op=	从右到左	

20.9 Do Statements

do、continue、goto 语句的实质也是控制语句，这些语句也可以通过与其他控制语句的配合来实现，所以它们并不是 C 语言程序设计所必须的。更重要的是，这些控制形式很容易因为滥用而造成程序结构的复杂化。

do 语句类似于 while，只是测试是在每次执行完循环体之后才进行而非开头进行。do 语句的一般形式为：

```
do{
  statements;
}while(test expression);
```

do 语句和 while 语句往往没有什么区别，只是 do 语句至少要执行一次 doyuju 的循环体，而 while 语句在控制表达式初始值为 0 时会完全跳过不执行循环体。

无论需要与否，所有的 do 语句都使用大括号，否则没有大括号的 do 语句很容易被误认为 while 语句。

do 语句在某些场合很有用，只是使用 do 语句的程序更容易出错，原因在于 do 语句循环体至少运行一次，即使条件表达式一开始就是 FALSE 也是这样。

在计算整数中数字的位数的示例程序中，使用 do 语句非常方便。

```
/*
 * Program: numdigit.c
 * ==============
 * Calculates the number of digits in an integer
 */

#include <stdio.h>

main()
{
  int digits = 0, n;

  printf("Enter a nonnegative integer: ");
  scanf("%d", &n);

  do{
```

```
    n /= 10;
    digits++;
  }while(n > 0);

  printf("The number has %d digit(s).\n", digits);

  return 0;
}
```

使用 while 语句来实现时，如果 n 的初始值为 0，那么将根本不执行上述循环，而且程序的执行结果为:`The number has 0 digit(s).`

20.10 Jump statements

为了退出循环，可以在循环体之前(使用 while 语句和 for 语句)或之后(使用 do 语句)设置退出点，有时候也需要使用 break 语句等在循环中设置退出点或者对循环设置多个退出点。

- break 语句用来跳出循环并且把程序控制传递到循环后的下一条语句；
- continue 语句用来跳过部分循环重复的剩余部分，但是不跳出循环。
- goto 语句允许程序可以跳到函数内的任何语句上。

20.10.1 Break statement

break 语句可以用于把程序控制从 switch 选择语句中转移出来，也可以用于跳出 while、do 或 for 循环语句。

对于在循环体的中间而不是在循环的开始或结束处设置退出点的情况，break 语句是特别有用的。例如，break 语句可以使循环在遇到特殊输入值时终止。

break 语句可以把程序控制从最内层封闭的 while、do、for 或 switch 语句中转移出来，但当这些语句出现嵌套时，break 语句只能跳出一层嵌套。

```
while(...){
  switch(...){
    ...
    break;
    ...
  }
}
```

这里，break 语句可以把程序控制从 switch 语句中转移出来，但是却不能跳出循环。

20.10.2 Continue Statements

continue 使程序立即开始执行离循环最近的下一个循环周期。continue 语句和 break 类似，但它无法使程序控制跳出循环。

- break 语句把程序控制正好转移到循环体末尾之后，因此使用 break 语句可以使程序控制跳出循环；
- continue 语句把程序控制正好转移到循环体结束之前的一点，因此使用 continue 语句还是会把程序控制留在循环内。
- break 语句可以用于 switch 语句和循环(while、do 和 for)，而 continue 语句只能用于循环。

continue 语句的形式为：

```
continue;
```

在进行数字求和的示例程序中，读入数字 0 后执行 continue 语句将使程序控制跳过循环体的剩余部分到达末尾，但是程序控制还继续留在循环中。

```
n = 0;
sum = 0;
while(n < 10){
  scanf("%d", &i);
  if(i == 0) continue;
  sum += i;
  n ++;
  /* contine jumps here. */
}
```

如果不使用 continue 语句，上述的示例可以写成：

```
n = 0;
sum = 0;
while(n < 10){
  scanf("%d", &i);
  if(i != 0){
    sum += i;
    n ++;
  }
}
```

总的来说，continue 语句只是在某些特殊情况下会非常有用，例如编写的循环要读入一些输入的数据，或者如果它们都很复杂，那么可以考虑 continue 语句。

```
for( ; ; ){
  读入数据;
  if(数据的第一条测试失败)
    continue;
  if(数据的第二条测试失败)
    continue;
  ...
  if(数据的最后一条测试失败)
    continue;
  处理数据;
}
```

尽管方便，但 continue 也使程序的可读性降低，尤其是用于复杂控制循环结构时。

20.10.3 Goto Statements

break 语句和 continue 语句都是跳转语句，它们把程序控制从程序中的一个位置转移到另一个位置。然而，这两者都是受限制的：

- break 语句的目标是在闭合的循环结束之后的那一点；
- continue 语句的目标是在循环结束之前的那一点。

C 语言可用的最低级的控制结构是 goto 语句，它可以使程序控制跳转到函数中任何有标号的语句处，而 break 语句和 continue 语句只能往前跳。

goto 语句的一般形式为：

```
goto label;
```

标识符 *label* 指定当前函数中的某一条语句，该语句用标识符 *label* 及一个冒号作为标记，如下所示：

```
label: statements
```

语句可以有多个标号，而且执行 goto 语句可以把控制转移到标号后的语句上，但这些语句必须和 goto 语句在同一个函数中。

如果 C 语言没有 break 语句，下面示范了使用 goto 语句提前退出循环的方法：

```
for(d = 2; d < n; d ++)
  if(n % d == 0) goto done;
done:
if(d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime.\n", n);
```

goto 语句是早期编程语言的主要内容，用于使控制直接转到被 *label* 标记的语句，但 goto 的不规则的使用会破坏程序的结构，使它变得极其难懂和难以维护，建议不要使用 goto 语句[5]。

> goto 语句不是天生的魔鬼，只是通常它有更好的替代方式。使用过多的 goto 语句才可能使程序迅速退化成“垃圾代码”。
> goto 语句使程序变得难以修改，原因在于 goto 可能会使某段代码用于多种不同的目的，例如 goto 语句既可以通过前面语句的“失败”，也可以通过通过多条 goto 语句的一条到达前面放置了标号的语句上。

break、continue 和 return 语句本质上都是受限制的 goto 语句，它们和 exit 函数一起足够处理其他编程语言中大多数需要 goto 语句的情况。

break 语句可以把程序控制从最内层封闭的 while、do、for 或 switch 语句中转移出来，但当这些语句出现嵌套时，break 语句只能跳出一层嵌套，这里使用 goto 语句不仅可以跳出嵌套，还可以跳出循环。

```
while(...){
  switch(...){
  ...
    goto loop_done; /* break won't work here. */
  ...
  }
}
loop_done: ...
```

goto 语句对于嵌套循环的退出也是很有用的。例如，在基于菜单的交互式程序中，可以向用户显示可供选择的命令列表，并且在用户选定了某条命令后执行相应的操作。在选定的操作结束后，提示用户输入下一条命令并执行或退出。

这类命令的核心是循环，在循环内将会有输入命令、读取命令、执行命令和退出等内容，可以用下列的示例来说明。

```
for( ; ; ){
  提示用户输入命令;
  读取命令;
  执行命令;
}
```

具体来说，执行命令时将会使用 switch 语句（或者级联 if 语句），因此可以进一步扩展如下：

```
for( ; ; ){
  提示用户输入命令;
  读取命令;
  switch(命令){
    case 命令1: 执行操作1; break;
    case 命令2: 执行操作2; break;
    ...
    case 命令n: 执行操作n; break;
```

[5] 1968 年，Dijkstra 在结构化程序设计方法的研究中提出了“GOTO 是有害的”，希望通过程序的静态结构的良好性保证程序的动态运行的正确性。

```
        default: 提示错误信息; break;
      }
    }
```

当用户执行完程序后需要从 switch 语句以及外围的循环中退出时，可以使用 return 语句。

return 语句可以使程序终止并且返回操作系统，return 语句后不能紧跟 break 语句，否则该 break 语句将永远不会执行，而且许多编译器还将显示警告错误。

Part III

Function

第 21 章

Introduction

术语“函数”[1]来源于数学中根据一个或多个给定参数进行数值计算的规则。在计算机科学中，函数(或子程序)[2]是一个大型程序中的某部份代码，由一个或多个语句块组成。

- 子程序(subroutine)是一个概括性的术语，任何高级程序所调用的程序都被称为子程序。子程序经常被使用在汇编语言层级上，其主体(body)是一个代码区块，当它被调用时就会进入运行。
- 在面向过程的语言中，函数(function)是对具有相关性语句的归类和对某过程的抽象。函数本身可以看作是结构(Struct)和类(Class)的前身，利用函数名称可以接收回传值(或返回值)。

```
c = max(a,b);
```

- 过程(procedure)是一种子程序，它能够接受不同的参数来执行某些特别的动作。

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

- 在面向对象程序设计语言中，类型或对象中的函数被称为方法(method)。

具体来说，C 语言函数仅仅是一组组合在一起的语句，这组语句有一个名字，可以用这个名字表示这一组完整的操作，可以将程序变得更短、更简单。如果没有函数，随着程序的规模和复杂性的增加，简单的程序就会变得不可管理。

为了理解函数如何从概念上减小程序的复杂性，需要从两个方面来理解这个概念。

- 从归约的角度看，需要了解函数如何工作，最终可以预测函数的行为；
- 从整体上看一个函数时，可以理解函数为什么如此重要以及如何有效地使用函数。

C 语言对“函数”的使用很宽松，它指定函数完成某项特定任务，而且相较于其他代码，具备相对的独立性(这有可能造成一些副作用，特别是在返回值是 void 类型时)。在其他编程语言中，函数和过程是被分开的，C 语言认为所有程序都是函数，这两者被认为是相同的。

一个函数表示一组用于完成某一操作的程序设计步骤。从这个角度来看，函数类似于一个完整的程序，而且所有的 C 语言程序的入口都被命名为 main 函数。

函数和程序在概念上的根本的不同在于谁来使用它。作为一个用户，当他启动一个应用时，其实就在运行一个完成某些指定动作的程序，因此程序是被外部用户调用并服务于外部用户的。而函数则提供了让程序调用事先定义好的一组操作的机制，函数的操作完全是程序内部的事。

例如，对于显示“Hello,world.”的程序而言，有如下不同的场景：

- 作为“Hello,world.”程序的用户，可以不知道程序调用了 printf 函数作为它操作的一部分，只需要看到“Hello,world.”出现在屏幕上；
- 作为 C 语言程序员，知道只要调用 printf 函数就可以将消息显示在屏幕上。
- 最困难的工作是由系统程序员来完成的，他要编写代码来实现 printf 函数所需要的所有功能，只要实现了 printf 函数，其他程序员就能通过调用来使用 printf 函数，而不需要再编写实现这个功能的语句了，甚至不需要知道实现这个功能的语句是什么。

[1]在数学中，一个函数是描述每个输入值对应唯一输出值的这种对应关系，符号为 f(x)。

[2]根据不同的应用场景，子程序可以翻译为 subroutine、procedure、function、routine、method 或 subprogram。

21.1 Calling

C 语言中的函数不同于数学中的“函数”。首先，函数(function)是一连串组合在一起的语句，并被给定一个名字。

其次，C 语言中的函数不一定需要参数，也不一定要计算数值(在某些编程语言中，“函数”返回一个值，而“过程”不返回值，C 语言没有这样的区别)。

函数是 C 语言程序的构建块，每个函数本质上是一个自带声明和语句的小程序。执行与函数相关的一组语句的行为称为调用(calling)这个函数，因此利用函数可以把程序划分成小块。

在 C 语言中，调用函数时是写一个函数名，后面跟一组括在一个圆括号中的表达式，这些表达式称为实际参数(argument)[3]。

实际参数是调用程序传给函数的信息。如果函数不需要从它的调用程序那里获取信息，就不需要实际参数，但是函数调用时必须要跟一对空的圆括号。

- 如果在调用函数时丢失圆括号，所得的结果仍然是合法的表达式语句(虽然没有意义)，但是语句没有执行结果。一些编译器会针对这种情况报出类似“Code has no effect.”的警告。
- 编译器会把不跟圆括号的函数名看作是指向函数的指针。因为指向函数的指针有合法的应用，所以编译器不能自动假定函数名不带圆括号是错误的。

一旦被调用，函数就从实际参数中获取数据，完成相应的工作，然后返回调用它的程序点。注意，调用程序所做的工作以及如何精确地返回到调用点是函数调用机制所定义的特性之一，返回到调用程序的操作称为从函数返回(returning)。作为返回操作的一部分，函数也能将结果返回给调用程序。例如，在下面的语句中

```
n1 = GetInteger();
```

GetInteger 函数完成了从用户处读取一个整型数的任务后，它把这个整数作为 GetInteger() 调用的值返回给调用程序，这个操作被称为返回一个值(returning a value)。

从某种意义上讲，参数提供了函数的输入，返回值是它的输出，输出返回给调用程序。尽管在概念上是非常类似的，但严格区分输入操作和在函数作用域中使用实际参数是很重要的。

函数提供了从用户获取输入的机制，例如，当需要一个输入值时，终端用户必须用键盘或其他设备输入这些值，而函数的实际参数则提供了一种使函数从它的调用程序那里获取输入值的方法，它是从程序的另一部分而不是从用户那里获取。以实际参数形式传递的数据必须由用户在程序前面的某一地方输入，并可能作为程序的某一部分进行计算。

另外，也必须仔细区分输出操作的使用和返回一个结果的技术，其中当进行输出操作时，输出结果将显示在终端设备上，而当被调用的函数返回一个结果时，该信息回到了调用程序，调用程序可以以任何方式使用该结果。当需要用实际参数和返回值的时候，有的人可能会倾向于在函数内使用输入/输出操作。

函数可以复用，一个函数最初可能只是某个程序的一部分，但可以将其用于其他程序中。程序员在调用函数时可以忽略其内部细节，只需将注意力集中于函数的整体作用，因此实际上函数是减小程序复杂性的关键工具。

为了理解如何在 C 语言中使用函数，必须知道：

- 函数调用是一个简单的表达式，它可以出现在任何表达式可以出现的地方；
- 函数的实际参数也可以是一个表达式，它本身也可包含函数调用或任何合法表达式中的操作。

现在，假设我们需要经常计算两个 float 型数值的平均值，而 C 语言没有“求平均值”的函数，用户需要自己定义它。例如，下面的函数 average 是一个自定义的“求平均值”函数。

```
float average(float a, float b)
```

[3]要仔细区分程序中的输入/输出和函数中的实际参数和返回值的概念。其中，

- 输出/输出程序和它的用户通信；
- 实际参数和返回值允许函数和它的调用程序之间通信。

```
{
  return (a + b) / 2;
}
```

- 返回类型(return type)代表调用函数返回的数据类型;
- 函数的形式参数(parameter)代表在调用函数时传递的参数,并且每个形式参数都必须有类型;
- 函数的形式参数本质上是变量,其初始值在调用函数时才提供;
- 函数体中的 return 语句可以是函数“返回”到调用它的地方;
- 函数调用可以出现在任何需要使用其返回值的地方;
- 调用(calling)函数时需要提供函数名及其实际参数列表(argument list);
- 实际参数用来给函数提供信息,并初始化形式参数。

注意,函数调用中的实际参数不能是任意的表达式,而必须是“赋值表达式”,且这类表达式不能用逗号作为运算符,除非逗号是在圆括号中。换句话说,在函数调用 f(a,b) 中,逗号是标点符号,而在 f((a,b)) 中,逗号是运算符。

下面的函数调用语句将计算并输出 x 和 y 的平均值。

```
printf("Average: %g\n", average(x, y));
```

这里,调用 average(x,y) 的效果就是把变量 x、y 的值复制给形式参数 a、b,然后执行 average 函数的函数体。

实际参数不一定是变量,任何正确类型的表达式都可以。

- 最里层的被调用函数 average 执行其函数体中的语句,并返回 x 和 y 的平均值;
- 外层的 printf 函数用于显示函数 average 的返回值,它同时也是 printf 函数的实际参数;
- 如果没有把 average 函数的返回值保存在任何地方,它将会在最外层的函数返回后被丢弃;
- 如果需要继续使用被调用函数的返回值,可以将其赋值给某个变量。

```
    avg = average(x, y);
```

实际上,并不是每个被调用的函数都返回一个值。例如,进行输出操作的函数可能不需要返回任何值。

```
void print_count(int n);
{
  printf("T minus %d and continuing.\n", n);
}
```

为了指定不返回值的函数,需要声明这类函数的返回类型是 void(在 C 语言中,void 用作占位符),而且不返回值的函数调用必须是语句,而不能是表达式。

```
/*
 * Prints a countdown.
 */
#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and continuing.\n", n);
}

main()
{
  int i;

  for(i = 10; i > 0; -- i)
    print_count(i);
  return 0;
}
```

变量 i 的值在最开始时是 10，当调用 print_count 函数时会把 i 的值复制给 n。

在第一个循环周期，print_count 函数会输出：

```
T minus 10 and continuing.
```

在第一个循环周期结束后，函数 print_count 会返回调用它的地方，这里是 for 语句的循环体。每次调用 print_count 函数时，变量 i 的值都不同，所以 print_count 函数输出的信息也不同。

C 语言中的过程函数仅用于执行操作，没有形式参数，也不提供显式返回值。

```
void print_pun(void)
{
  printf("To be, or not to be, that is the question.\n");
}
```

为了调用不带实际参数的函数，必须在函数名后跟一对空的圆括号。

```
/*
 * Prints a pun.
 */
#include <stdio.h>

void print_pun(void)
{
  printf("To be, or not to be, that is the question.\n");
}

main()
{
  print_pun();
  return 0;
}
```

这里，调用 print_pun 函数的是 main 函数，因此当最里层的 printf 函数返回时，print_pun 函数也就返回了 main 函数。

综上所述，对于 void 型和非 void 型的函数调用，总结如下：

- void 型的函数调用是语句，调用后边始终跟着分号；
- 非 void 型的函数调用是表达式，可以把调用产生的值存储在变量中，还可以进行测试、输出或其他操作。
- 如果需要，可以丢弃非 void 型函数的返回值。

有些情况下，丢掉函数的返回值是有意义的。例如，printf 函数会返回显示的字符的个数。

```
num_chars = printf("To be or not, this is a question.\n");
```

实际应用中，可能不需要显示字符的数量，因此通常会丢掉 printf 函数的返回值。为了清楚地表示故意丢弃非 void 型函数的返回值，C 语言允许在函数调用前加上 (void)。

```
(void) printf("To be or not, this is a question.\n");
```

这样，就可以把 printf 函数的返回值强制类型转换成 void 类型，也就是把返回值丢掉了。在 C 语言标准库中，有大量函数的值并例行公事的丢掉了。

21.2 Definition

函数定义有如下的句法形式：

```
result-type name(argument-specifiers)
{
   ... optional declaration ...
```

```
    ... body ...
}
```

其中，

- result-type 是函数的返回值类型；
- name 是函数名；
- argument-specifiers 是一个形式参数列表，形式参数之间以逗号隔开；
- optional declarations 部分包含函数中使用的局部变量的声明；
- body 由实现函数所需的语句组成。

在编译由 Kernighan 和 Ritchie 编写的著名的“hello, world”程序时会产生如下的警告信息：

```
$ vim hello.c
#include <stdio.h>

main()
{
  printf("hello, world.\n");
}
$ gcc -Wall hello.c
hello.c:3:1: warning: return type defaults to ‘int’ [-Wreturn-type]
 main()

hello.c: In function ‘main’:
hello.c:6:1: warning: control reaches end of non-void function [-Wreturn-type]
}
```

- 如果忽略返回类型，C 编译器会假定函数返回值的类型是 int 类型；
- 函数无法返回数组，但是没有全体关于返回类型的限制；
- 指定返回类型为 void[4]类型说明函数没有返回值。
- 如果返回类型冗长（例如 unsigned long int），可以把返回类型单独放在一行。

针对编译时警告信息，对源程序进行如下的修改：

```
$ vim hello.c
#include <stdio.h>

int main(void)
{
  printf("hello, world.\n");
  return 0;
}
$ gcc -Wall hello.c
$ gcc -Wall -o hello hello.c
```

在函数定义中，形式参数之间以逗号分隔，并且需要指明每个形式参数的类型。即使有些形式参数具有相同的数据类型，也必须对每个形式参数分别进行类型说明。如果函数没有形式参数，也需要以 void 说明。

函数体可以包含声明和语句，而且在函数体中声明的变量专属于该函数，称为局部变量。其他函数不能对局部变量进行检查或修改，也因此在一个函数中用作参数名或变量名的标识符可以在其他函数中复用。

一些编程语言允许过程和函数互相嵌套，但是 C 语言不允许函数定义嵌套，即一个函数的定义不能出现在另一个函数体中，这个限制可以使编译器简单化。

另外，函数体可以为空。在程序开发过程中留下空函数体是有意义的（如果没有时间完成函数，通过函数体留空可以为函数预留空间以备将来完善）。

[4] K& R C 缺少 void 类型，如果函数没有返回值，程序员经常会忽略掉返回类型，但不建议这样做，否则 C 编译器可能会假定返回值的类型是 int 型。

21.3 Declaration

21.3.1 Variable Declaration

在使用 C 语言编程时，使用变量之前必须对其进行声明，也就是要指定变量的名字和类型。对于具有相同类型的变量，可以把变量声明合并。

例如，在 main 函数中包含变量声明时，就必须把声明放置在语句之前。

```
main()
{
   ... variable declaration ...
   ... body ...
}
```

21.3.2 Function Declaration

K& R C 中并没有引入函数声明，如果没有明确提供函数原型，编译器会假设它具有某些特性。在 C 语言被标准化之后，所有的函数在使用前也必须声明。

如果函数未定义或在调用函数之后才定义，编译器没有任何关于被调用函数的信息，此时编译器不知道被调用函数的返回类型、形参类型和形参列表，不过编译器没有产生错误信息，而是对被调用函数进行假设。

- 假设被调用函数返回 int 类型的值(函数返回值的默认类型为 int 型)；
- 假设给被调用函数传递了正确数量的实际参数；
- 假设在提升后实际参数具备正确的类型。

当编译器对被调用函数的信息无法确认后，程序无法工作。为了避免定义前调用引发的问题，一种把法是将每个函数的定义都在调用函数之前进行，另一种办法是在调用前声明每个函数。

```
result-type name(parameter-list);
```

在调用函数之前先声明函数的好处在于可以说明函数间的调用关系，而且函数声明应该位于函数体外。

函数的声明必须与函数的定义一致，但函数声明不需要说明函数形式参数的名字，只需要显示它们的类型。不过，通常建议保留形式参数的名字来注释每个形式参数的目的，并提示程序员在调用函数时有关实际参数出现时必须依据的次序(实际参数的值是按序复制到形式参数的)。

为了与 K& R C 的函数声明相区别，标准 C 将函数声明称为函数原型(function prototype)。函数原型为如何调用函数提供了一个完整的描述：形参列表、参数类型和返回值类型。

21.4 Argument

形式参数(parameter)出现在函数定义中，它们以假名字来表示表示函数调用时提供的值。

实际参数(argument)出现在函数调用中，不一定是变量，任何正确类型的表达式都可以。

在 C 语言中，实际参数是通过值传递的：调用函数时，计算出每个实际参数的值并且把它赋值给相应的形式参数。在函数执行过程中，对形式参数的改变不会影响实际参数的值，从效果上来说，每个形式参数的行为好像是把变量初始化成与之匹配的实际参数的值。

实际参数按值传递既有利也有弊，既然形式参数的修改不会影响到相应的实际参数，那么可以把形式参数作为函数内的变量来使用，从而可以减少真正需要的变量的数量。

下面是一个用来计算数 x 的 n 次幂的函数。

```
int power(int x, int n)
{
  int i, result = 1;
```

```
  for(i = 1; i <= n; i ++)
    result = result * x;
  return result;
}
```

因为 n 只是原始指数的副本，可以在函数体内修改它，所以就不需要使用变量 i 了。

```
int power(int x, int n)
{
  int result = 1;

  while(n-- > 0)
    result = result * x;
  return result;
}
```

实际应用中，C 语言关于实际参数按值传递的要求使它很难编写确定的函数类型。

21.4.1 Typecast

C 语言允许在实际参数的类型与形式参数的类型不匹配的情况下进行函数调用。管理如何转换实际参数的规则与编译器是否在调用前遇到函数(或者函数的完整定义)的原型有关。

- 编译器在调用前遇到原型，每个实际参数的值被隐式的转换成相应形式参数的类型。例如，如果把 int 类型的实际参数传递给期望得到 float 型数据的函数，则会自动把实际参数转换成 float 类型。
- 编译器在调用前没有遇到原型，编译器执行默认的实际参数提升。
  1. 把 float 型的实际参数转换成 double 类型；
  2. 执行整数的提升(把 char 型和 short 型的实际参数转换成 int 型)。

默认的实际参数提升可能无法产生期望的结果。

```
main()
{
  int i;

  printf("Enter number to be squared: ");
  scanf("%d", &i);
  printf("The answer is %g\n", square(i));

  return 0;
}

double square(double x)
{
  return x * x;
}
```

在调用 square 函数时，编译器没有遇到 square 函数原型，也没有遇到 square 函数定义。在这种情况下，编译器执行默认的实际参数提升，实际参数的类型维持 int 型，而 square 函数期望的是 double 类型的实际参数，所以 square 函数将产生无效的结果。

通过在调用前声明 square 函数或者把变量强制转换为正确的类型，可以解决这个问题。

```
printf("The answer is %g\n", square((double) i));
```

默认的实际参数提升不会总获得期望的效果这一事实使得始终在调用函数前声明函数变得更加必要。

21.4.2 Array

数组经常被用作实际参数。当形式参数是一维数组时，可以(而且是通常情况下)不说明数组的长度。

```
int f(int a[])
{
  ...
}
```

实际参数可以是任何一维数组，且数组元素拥有正确的类型。

C 语言没有为函数提供任何简便的方法来确定传递给它的数组的长度[5]，如果函数需要，必须把长度作为额外的实际参数提供。

下面的函数说明了一维数组型实际参数的用法。当给出具有 int 型值的数组 a 时，sum_array 函数返回数组 a 中元素的和，也就需要知道数组 a 的长度，因此需要把长度作为第二个实际参数。

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for(i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

调用 sum_array 函数时，第一个参数是数组的名字，第二个参数是数组的长度，因此 sum_array 函数的原型如下：

```
int sum_array(int a[], int n);
```

通常情况下，可以忽略形式参数的名字，因此 sum_array 的函数原型可以简化为：

```
int sum_array([], int);
```

在把数组名传递给函数时，不要在数组名的后边放置方括号。

函数无法检测通过传递是否获得了正确的数组长度，因此可以告诉函数数组比实际小的多。实际数组中剩余的空间将被浪费，而且函数也不知道剩余空间的存在。如果告诉函数数组型实际参数比实际的大，函数将超出数组的末尾。

当形式参数是多维数组时，只能忽略第一维的长度，因此也就无法传递具有任何维数的多维数组，这个问题可以使用指针数组的方式来解决。

- 在把数组传递给函数时，是把指向数组第一个元素的指针传给了函数。
- C 语言按照行主序存储数组，即首先存储第 0 行的元素，然后存储第 1 行的元素，以此类推。
- 为了存储数组，编译器必须知道数组中每一行的大小，而行的大小由列数决定，因此数组中第一维的参数可以忽略，其他维数则必须说明。

[5]虽然可以用 sizeof 运算符计算出数组变量的长度，但是它无法给出关于数组型形式参数的信息，因此无法用 `sizeof(a) / sizeof(a[0])` 来计算形式参数的长度。

第 22 章

Prototype

C 语言中的函数声明称为函数原型(function prototype),它具有下列的格式:

```
result-type name(argument_specifiers);
```

result-type (返回类型) 字段指出了函数的结果的类型, name (名字) 字段指出函数的名字。argument_specifiers(参数描述符)字段指出传递给这个函数的参数名字和类型。参数描述符字段是一个表,每个参数类型字段之间用逗号分开,而且参数规格说明中每个类型名后面还可以跟一个变量名来提供一些额外的信息。

```
语法：函数原型
result-type name(argument_specifiers);
其中：
result-type是函数返回值的类型；
name是函数名；
argument_specifiers是一组由逗号分开的参数类型规格说明的表，每个参数规格说明有一个类型，后面的参数名是可选的。
```

函数声明类似于变量声明,函数原型与函数定义的形式完全相同,只是原型的后面不是整个函数体而是一个分号。

- 变量声明告诉编译器变量的名字和它包含的值的类型;
- 函数的声明更加详细,它包含了:
    1. 函数的名字;
    2. 每个参数的类型,大多数情况下还包括了参数的名字;
    3. 函数返回值的类型。
- 函数的声明中可以不写形式参数变量名。

例如,math 库中 sqrt 函数的原型如下:

```
SYNOPSIS

  #include <math.h>

  double sqrt(double x);
  float sqrtf(float x);
  long double sqrtl(long double x);

DESCRIPTION

  The sqrt() function returns the nonnegative square root of x.

RETURN VALUE

  On success, these functions return the square root of x.
  If x is a NaN, a NaN is returned.
  If x is +0 (-0), +0 (-0) is returned.
  If x is positive infinity, positive infinity is returned.
  If x is less than -0, a domain error occurs, and a NaN is returned.
```

该原型说明函数 sqrt 有一个参数,它的类型是 double,同时返回一个 double 类型的值。

注意,原型只指定了调用程序和函数之间的传递的值的类型,从原型看不出定义这个函数的真正语句,甚至看不出函数要干什么。然而 C 编译器不需要这个信息,它必须知道的只是函数的返回值类型及参数的类型。

例如,对于 sqrt 函数,C 编译器只需要知道 sqrt 取一个 double 类型的值,并返回一个 double 类型的值。其实,函数的确切地作用是通过函数名和相关的文档告诉程序员的。

```
$ man sqrt
NAME
  sqrt, sqrtf, sqrtl - square root function

SYNOPSIS
    #include <math.h>

    double sqrt(double x);
    float sqrtf(float x);
    long double sqrtl(long double x);

    Link with -lm.

  Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    sqrtf(), sqrtl():
      _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600 || _ISOC99_SOURCE ||
          _POSIX_C_SOURCE >= 200112L;
    or cc -std=c99
...
```

为程序员提供有用信息的另一个途径是为每一个参数提供一个描述性的名字,该名字可以标识特定参数的作用(还可以包含相应的参数规格说明)。参数的名字为使用该函数的程序员提供了重要的信息,但对程序没有任何实质性的影响,例如,在 math.h 中 sin 被声明为:

```
double sin(double);
```

它仅指出了参数的类型,需要使用这个函数的程序员可能更希望看到这个原型被改写为:

```
double sin(double angleInRadians);
```

以这种形式编写的原型提供了一些有用的新信息:sin 函数有一个 double 类型的参数,该参数是以弧度表示的一个角度。

在自己编写的函数中,应该为参数指定名字,并在相关的介绍函数操作的注释中注明这些名字。

如果函数没有关键字,C 语言使用关键字 void 作为参数规格说明,例如,simpio 库中的函数 GetInteger 没有从它的调用程序那里获取参数,返回的是一个 int 类型的值,于是这个函数的原型可以写为:

```
int GetInteger(void);
```

第 23 章

Procedure

在程序设计中，调用一个程序，或者是为了得到它的返回值，或者是需要产生某种作用，其中：

- 需要返回值的函数可以用 return 语句来指定所“返回”给调用点的值；
- 不需要返回值而仅是起到某种作用的函数被称为过程(procedure)。

在不同的程序设计语言中，可能会对函数和过程有不同的定义，其中 PASCAL 和 FORTRAN 就提供了完全不同的机制定义函数和过程，在这些语言中，函数和过程是两个不同的概念实体。

在 C 语言中，这两个概念是合在一起的，大多数有关 C 语言的著作中都用术语函数表示函数和过程，只是要认识到某些函数时不返回结果的，因此，在强调某个函数没有返回值时，就会使用过程这个概念。

C 语言中可以通过在函数原型中返回类型的地方用关键词 void 来表示过程。例如下面的原型：

```
void GivenInstructions(void);
```

声明一个没有参数和返回值的函数，对任何函数来讲，原型出现在程序的开始部分；同样地，GivenInstructions 的实现出现在程序后面的部分中，这个过程的一般实现如下：

```
void GivenInstructions(void)
{
  printf( “This program performs some important calculation\n” );
  printf( “for the user. This function, or one very much like it,\n” );
  printf( “can be used to give the user whatever instructions are\n” );
  printf( “required to use the program, such as the format for \n” );
  printf( “input data and the like.\n” );
}
```

和大多数过程一样，GivenInstructions 没有包括 return 语句，而是直接执行到最后，C 语言理解这种情况为过程的执行完成了，然而过程也可以用 return 语句强制立即从过程返回到调用点。当 return 出现在过程中时，其后没有表达式，仅仅是如下格式：

```
return;
```

当主程序要显示这些提示时，就会调用该过程，其格式与调用函数一致：

```
main()
{
  GivenInstructions();
  程序的其他部分;
}
```

通过将程序中的提示部分组成一个单独的过程，可以改进程序的结构，增强其可读性。而对程序的其他部分感兴趣的人可以跳过这一行代码，转而注重程序其他部分的执行细节；而对程序中的提示感兴趣的人可以直接找到 GivenInstructions 过程的实现，并集中阅读程序的这一部分。

第 24 章

Custom Function

在系统程序员看来，库函数只对它们的正确性感兴趣，不会说明函数的全部内容。作为程序员，通常只要使用库函数，而不必知道任何内容细节。但有时候，也有可能会使用不属于任何库的函数，于是便需要定义自己的函数。

例如，写一个程序将大多数国家使用的摄氏温度转换为美国使用的华氏温度，这里需要定义一个转换函数，可以用在程序的其他地方。将这个温度从一个计量标准转换为另一种计量标准只要使用下列公式即可：

$$F = \frac{9}{5}C + 32$$

在 C 语言程序中，增加一个新的函数由两个步骤组成：

1. 需要指定这个函数原型，它通常位于整个程序的头部，在 #include 行之后；
2. 在程序的稍后部分，需要提供该函数的实现，即指定函数包含的每一个步骤。

原型是很短的，仅指出函数的实际参数和返回类型，实现则较长，它要提供实现的细节。在编写一个函数时，通常最好从编写它的原型开始。在命名函数时，要使得读程序的人容易确定函数要做什么。

```
double CelsiusToFahrenheit(double c);
```

函数的实现是由它的原型开始，去掉结尾的分号，然后加入函数体。函数体（function body）是一个程序块，由一组括在一对花括号中的语句组成，程序块中的语句前面可以有变量声明。

24.1 Return Statement

通常情况下，在执行完最后一条语句后函数将自动返回。但是，执行任何形式的 return 语句都会导致当前的函数立即返回到它的调用函数，并把表达式的值作为函数的值传递给调用函数。

- 函数中可以有多条 return 语句；
- 在给定的函数调用中，只能执行一条 return 语句。

如果非 void 函数要永远执行到函数体的末尾，那么返回的值是未定义[1]的，因此非 void 的函数必须使用 return 语句来说明函数的返回值。

使用 return 语句可以有两种形式来指出函数要返回的值。其中，对于过程来说，可以把 return 语句写成：

```
return;
```

如果函数需要返回一个结果，则函数体中至少要包含一个 return 语句，这种情况下的 return 语句范例如下所示：

```
语法：return语句
  return(expression);
其中：
expression是要返回的值；
```

在大多数情况下，return 语句包括一个由圆括号[2]括起来的 expression，它指出了结果值。

[1] 如果编译器检查出非 void 函数有“离开”函数体末尾的可能性，那么将产生诸如“Function should return a value”的信息。

[2] 习惯上要把代表要返回的值的表达式放在括号里，但这种写法不是必须的。如果没有返回值，也仍然需要 return 语句。

- 圆括号是可选的，但 C 语言通常加上圆括号来增加可读性，但它也可以用于没有结果值的函数。
- expression 可以是常量、变量和表达式等。
- 函数无法返回数组，但是没有关于返回类型的限制。

下面的示例说明了在 return 语句的表达式如何使用条件运算符。

```
return i > j ? i : j;
```

如果 return 语句中表达式的类型和函数的返回类型不匹配，那么系统将会表达式的类型隐式的转换成返回类型。

如上所述，C 语言中的 return 语句包含两个概念：“我已经完成了”和“这就是答案”。在另一些程序设计语言中，如 PASCAL 和 FORTRAN，指出函数执行完成和指定它的结果是两个单独的操作。

这里，使用 return 语句可以完成实现一个 CelsiusToFahrenheit 函数的所有需求：

```
double CelsiusToFahrenheit(double c)
{
  return ((9.0 / 5.0) * c + 32);
}
```

这个函数计算相应表达式的值，并将该值作为函数的结果值返回。

函数 CelsiusToFahrenheit 本身不能组成一个完整的程序，每个完整的 C 语言程序都有一个名字为 main 的函数作为程序的入口，当程序开始执行的时候就调用 main 函数。

为了测试 CelsiusToFahrenheit 函数，要写一个 main 版本，它使用 CelsiusToFahrenheit 函数生成一个温度转换表。

```
/*
* File:c2ftable.c
* ------------------
* This program illustrates the use of function by generating a table of Celsius
* to Fahrenheit conversions.
*/

#include <stdio.h>
#include  “genlib.h”

/*
* Constants
* ------------
* LowerLimit –Starting value for temperature table
* UpperLimit –Final value for temperature table
* StepSize –Step size between table entries
*/

#define LowerLimit 0
#define UpperLimit 100
#define StepSize 5

/* Function prototype */

double CelsiusToFahrenheit(double c);

/* Main Program */

main()
{
  int c;

  printf( “Celsius to Fahrenheit table.\n” );
```

```
  printf( "C       F\n" );
  for(c = LowerLimit; c <= UpperLimit; c += StepSize){
    printf( "%3d %3g\n" ,c,CelsiusToFahrenheit(c));
  }
}

/*
* Function:CelsiusToFahrenheit
* Usage:f = CelsiusToFahrenheit(c);
* ----------------------------------
* This function returns the Fahrenheit equivalent of the Celsius temperature c.
*/

double CelsiusToFahrenheit(double c)
{
  return (9.0 / 5.0 * c + 32;
}
```

主程序由一个用来生成表的 for 循环组成，温度转换表中的每一行都是由下面的语句生成。

```
printf( "%3d %3g\n" , c, CelsiusToFahrenheit(c));
```

这个语句调用 CelsiusToFahrenheit 函数计算对应于摄氏温度 c 的华氏温度，结果值作为实际参数传递给 printf，printf 继续该值。

注意，在 for 循环中，下标变量 c 声明为 int 型，尽管 CelsiusToFahrenheit 被定义为取一个 double 型的参数。而由于整型数是精确的，因此可以保证程序运行正确的次数。当用参数 c 调用 CelsiusToFahrenheit 时，通过自动类型转换将参数值转换为 double 类型。

在正式的程序中，注释往往较多，而且每个函数也应该有它自己的描述性的注释，这样程序的读者可以将每个函数作为一个单元来理解。

为函数写的最有用的注释之一是给出一个函数是如何被使用的实例。一般地，函数的注释中都应包含一个提供这种实例的“Usage”行。

24.2 Control Structure

在大多数情况下，编写一个函数需要做出许多测试或编写一个循环，虽然这样的细节问题增加了实现一个函数的复杂性，但不会改变函数的基本形式。例如，库函数 abs 计算它的整数实际参数的绝对值，它的原型为：

```
int abs(int n);
```

abs 标准库函数在 ANSI C stdlib 库中定义，其函数原型如下：

```
SYNOPSIS
  #include <stdlib.h>

  int abs(int j);
  long int labs(long int j);
  long long int llabs(long long int j);

  #include <inttypes.h>

    intmax_t imaxabs(intmax_t j);

  Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

  llabs():
    _XOPEN_SOURCE >= 600 || _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L;
```

```
       or cc -std=c99

    DESCRIPTION
      The abs() function computes the absolute value of the integer argument j. The labs(),
          llabs() and imaxabs() functions compute the absolute value of the argument j of the
          appropriate integer type for the function.

    RETURN VALUE
         Returns the absolute value of the integer argument, of the appropriate integer type
             for the function.

    ATTRIBUTES
       Multithreading (see pthreads(7))
          The abs(), labs(), llabs(), and imaxabs() functions are thread-safe.
```

当用户自己实现 abs 函数时，绝对值的定义指出：

- 如果参数是负数，函数将返回它的相反数，即一个正数；
- 如果参数是正数或 0，函数将返回参数值本身。

因此可以写出如下的 abs 函数的实现：

```
int abs(int n)
{
  if(n < 0){
    return (-n);
  }else{
    return (n);
  }
}
```

从上述的函数的实现中可以看到，return 语句可以出现在函数体的任何地方。同样，可以定义一个函数 MinF 返回两个浮点实际参数中较小的一个：

```
double MinF(double x, double y)
{
  if(x < y){
    return (x);
  }else{
    return (y);
  }
}
```

当要定义一个函数 Factorial，它取一个整型数 n 作为参数，并返回 1 ～ n 的乘积，即返回这个参数的阶乘，前几个数的阶乘如下所示：

```
Factorial(0) = 1         （由定义所得）
Factorial(1) = 1        = 1
Factorial(2) = 2        = 1*2
Factorial(3) = 6        = 1*2*3
Factorial(4) = 24       = 1*2*3*4
Factorial(5) = 120      = 1*2*3*4*5
Factorial(6) = 720      = 1*2*3*4*5*6
Factorial(7) = 5040     = 1*2*3*4*5*6*7
```

Factorial 函数取一个整型数，返回一个整型数，它的原型如下所示：

```
int Factorial(int n);
```

比较得出，计算阶乘的任务在许多方面类似于求一组数的和。在实际求一组数的和的程序 addlist.c 中，声明了一个变量 total 来记录运行过程中的和值，程序开始时，total 被初始化为 0，每当输入

一个新值时，它就被加到 total 中去，因此 total 反映了至今为止输入的数的总和。而在实现阶乘的程序中，情况非常类似，只是必须保存乘积的记录而不是和的记录。为了实现该功能，必须：

1. 定义一个变量 product；
2. 把它初始化为 1；
3. 将 1 ～ n 之间的整数逐个与它相乘；
4. 将 product 的最终值作为函数的结果返回。

为了得到第三步中所需的每一个整数，需要一个 for 循环，它从 1 开始一直增加直到 n 为止。for 循环需要一个下标变量，通常用 i 表示，因此 Factorial 需要声明两个变量：

```
int product, i;
```

其中，变量 product 保存运行过程中的乘积值，i 是下标变量。下面是 Factorial 的实现：

```
int Factorial(int n)
{
  int product, i;

  product = 1;
  for(i = 1; i <= n; i ++){
  product *= i;
  }
  return (product);
}
```

24.3 Return Value

在 C 语言中，函数返回的值可以是数字值，也可以是非数字值。

默认情况下，main 函数的返回类型是 int 型，使用下面的形式可以使返回类型成为显式的。

```
int main()
{
  ...
  return 0;
}
```

main 函数的返回值是状态码，在某些操作系统中程序终止时可以检测到状态码，确保每个 C 程序都返回状态码是很好的实践。上述示例中，main 函数以 `return 0;` 结尾是为了保证编译顺利通过，否则有可能编译器会报出“Function should return a value”的警告信息。

- 如果程序正常终止，main 函数应该返回 0；
- 如果程序异常中止，main 函数应该返回非 0 值。实际上，非 0 返回值也可以用于其他目的。

通过测试 main 函数的返回值来判断程序是否正常终止，这主要依赖于使用的操作系统。许多操作系统允许在“批处理文件”或“外壳文件（Shell）”内测试 main 的返回值，这类文件包含可以运行几个程序的命令。例如，

```
if errorlevel 1 ...
```

- 在 DOS 批处理文件中，上述这行命令测试最后程序是否以大于或等于 1 的状态码终止。
- 在 UNIX 系统中，每种 Shell 都有自己测试状态码的方法，在 bash 中用变量 $? 包含最后程序运行的状态，而在 csh 中类似的变量是 $status。

例如，要编写一个对日期进行操作的程序，那么有一个将用数值表示的月份（1 ～ 12）转换为用相应的文字表示月份名（从 January 到 December）的 string 类型的函数是非常有用的。虽然计算机内部处理数值型的值较容易（例如，比较两个月份，看哪一个更早一些），但用文字输出月份名会使可读性更强。为实现该功能，可以定义一个函数 MonthName，其具体实现如下：

```
string MonthName(int month)
{
  switch(month){
  case 1:    return ( “January” );
  case 2:    return ( “February” );
  case 3:    return ( “March” );
  case 4:    return ( “April” );
  case 5:    return ( “May” );
  case 6:    return ( “June” );
  case 7:    return ( “July” );
  case 8:    return ( “August” );
  case 9:    return ( “September” );
  case 10:   return ( “October” );
  case 11:   return ( “November” );
  case 12:   return ( “December” );
  default:   return ( “Illegal month” );
  }
}
```

为使用该函数,在程序的其他部分应该调用 MonthName,然后用 printf 显示这个结果。

例如,若整型变量 month、day 和 year 的值分别为 7、20 和 1969(阿波罗 II 登上月球的日子),语句

```
printf( “%s %d, %d\n” , MonthName(month), day, year);
```

会产生一个输出如下:

```
July 20, 1969
```

注意到在 MonthName 函数的 switch 语句中,每个 case 子句中的 return 语句自动地退出整个函数,因此不需要再写一个 break 语句。

- 在 switch 语句中,如果在设计程序时让每个 case 子句都以 break 或 return 语句结尾,可以避免调试中的许多困难。
- 在 main 函数中,`exit (0)` 和 `return 0` 是完全等价的,二者都用于终止程序执行,并且向操作系统返回 0 值。
- 在程序终止时没有执行 return 语句时,仍然会有某个值返回到操作系统,但是无法保证这个值是什么。

24.4 Exit Function

在 main 函数中执行 return 语句是终止程序的一种方法,另一种方法是调用 stdlib.h 提供的 exit 函数。

exit 函数具有在任何函数中终止程序执行的能力,传递给 exit 函数的实际参数和 main 函数的返回值具有相同的含义,两者都说明程序终止时的状态。

- 为了说明正常终止,传递 0。

```
exit(0); /* normal termination */
```

- 因为 0 是模糊的位,C 语言允许用传递 EXIT_SUCCESS 来代替(效果是相同的)。

```
exit(EXIT_SUCCESS); /* normal termination */
```

- 传递 EXIT_FAILURE 说明异常中止。

```
exit(EXIT_FAILURE); /* abnormal termination */
```

EXIT_SUCCESS 和 EXIT_FAILURE 都是定义在 stdlib.h 中的宏,EXIT_SUCCESS 和 EXIT_FAILURE 的值都是由实现定义的,典型的值分别是 0 和 1。

作为终止程序的方法,return 语句和 exit 函数关系紧密。事实上,语句

```
return expression;
```

在 main 函数中等价于：

```
exit (expression);
```

任何函数(包括 main 函数)都可以调用 exit 函数，exit 函数可以便于模式匹配程序很容易地定位程序中全部的退出点。

24.5 Predicate Function

虽然 C 语言的函数可以返回任何类型的值，但有一种结果类型值值得特别关注，这就是在扩展库 genlib 中定义的 bool 类型。

返回 bool 类型值的函数被称为谓词函数(predicate function)，bool 类型仅有两个值 TRUE 和 FALSE，因此一个谓词函数，不管它有多少参数，也不管它内部处理有多复杂，最终总是返回这两个值中的某一个。

调用谓词函数类似于问一个 yes/no 的问题，并得到答案。考虑下面的函数，给定一个整数 n，回答“n 是偶数吗?”这个问题：

```
bool IsEven(int n)
{
  return (n % 2 == 0);
}
```

- 当参数被 2 除没有余数时，它就是偶数；
- 如果 n 是偶数，表达式

  ```
  n % 2 == 0
  ```

  为 TRUE，这个值被作为函数 IsEven 的结果返回。
- 如果 n 是奇数，函数返回 FALSE。

由于 IsEven 返回一个布尔值，可以直接把它放在条件出现的地方。例如，下面的主程序用 IsEven 列出所有 1 ～ 10 之间的偶数。

```
main()
{
  int i;

  for(i = 1; i <= 10; i ++){
    if(IsEven(i)) printf( “%2d\n” ,i);
  }
}
```

在开始使用谓词函数时，常见的错误就是判断条件的冗余。例如：

```
if(IsEven(i) == TRUE) . . . /* == TRUE是冗余的 */
```

作为谓词函数的另一个实例，可以编写一个程序来测试给定的年份是否为闰年，程序如下：

```
bool IsLeapYear(int year)
{
  return (((year % 4 == 0) && (year % 100 == 0)) || (year % 400 == 0));
}
```

在确定 year 是否为闰年的程序中，把这个函数引入程序就不需要在测试时明确地包含这个完整的计算，在程序的其他地方可以用如下语句调用函数：

```
if(IsLeapYear(year)) . . .
```

24.6 String Comparation

在引入字符串函数库后，就可以对字符串数据进行各种操作，比如利用 StringEqual 函数可以区分两个字符串是否包含完全相同的字符，其函数原型为：

```
bool StringEqual(string s1,string s2);
```

它指出 StringEqual 函数以两个字符串作为实际参数，并返回一个布尔值。如果两个字符串 s1 和 s2 中的字符完全相同，这个值为 TRUE，如果两个字符串中的字符有一点点不同，StringEqual 函数就返回 FALSE。例如，下面的例子：

```
StringEqual( “abc” , “abc” );  返回TRUE
StringEqual( “abc” , “abcd” ); 返回FALSE
StringEqual( “abc” , “ABC” );  返回FALSE
```

在需要问用户一个问题并根据回答采取某些行动时，可以使用 StringEqual 函数。假设编写了一个与用户做游戏的程序，可以给用户提供一个重玩得机会，通过在屏幕上提供如下的适当提示并向用户询问，然后调用 GetLine 获取响应：

```
printf( “would you like to play again?” );
answer = GetLine();
```

StringEqual 函数使得程序可以针对响应完成某些操作，例如如下的主程序：

```
main()
{
 string answer;

 while(TRUE){
  PlayOneGame();
  printf( “would you like to play again?” );
  answer = GetLine();
  if(StringEqual(answer, “no” )) break;
 }
}
```

如果用户给出单词 no，程序从 while 循环中退出，如果用户给出其他响应，将再玩一次。

第 25 章

Function Call

从整体角度考虑函数有助于理解函数是如何使用的和作为程序设计资源，函数为我们提供了什么。为了逐步确保所写的函数会完成它们应该做的工作，还需要逐步理解函数内部的实现过程和函数调用过程机制。

下面的程序 fact.c 包括 Factorial 函数和一个显示阶乘表的主程序。可以把 fact.c 程序看成两部分，主程序只是从 LowerLimit 到 UpperLimit 计数，在每一次循环中，对下标变量 i 的值调用 Factorial 函数，并显示函数结果。一般用名字 i 表示用于计算 for 循环的循环次数的下标变量，而且 i 是一个没有特别意义的下标变量，用来跟踪 for 循环的进展。

完整的 fact.c 程序如下：

```
/*
* File:fact.c
* ------------
* This program includes the Factorial function and a test program
* that prints the factorials of the numbers between the limits
* LowerLimit and UpperLimit, inclusive.
*/

#include <stdio.h>
#include "genlib.h"

/*
* Constants
* ------------
* LowerLimit -- Starting value of factorial table
* UpperLimit -- Final value for factorial table
*/

#define LowerLimit 0
#define UpperLimit 10

/* Function prototype */

int Factorial(int n);

/* Main Program */

main()
{
  int i;

  for(i = LowerLimit; i <= UpperLimit; i ++){
    printf("%d! = %5d\n", i, Factorial(i));
  }
}

/*
```

```
* Function:Factorial
* Usage:f = Factorial(n);
* ----------------------
* Returns the factorial of the arguments n, where factorial is defined as the
* product of all integers from 1 up to 10.
*/

int Factorial(int n)
{
  int product, i;
  product = 1;
  for(i = 1; i <= n; i ++){
    product *= i;
  }
  return (product);
}
```

当把程序作为整体来看待时，每次只看一个函数时，该函数是有意义的，而且每个函数定义只使它自己有意义，而随着程序规模的增长，已没有办法将它们作为一个整体来理解，要理解大的程序，只有把它分块，每一分块足以使得它自己有意义。

在阅读程序的每个片段时，第一个容易混淆的问题是可能有几个名称相同的变量，但要注意的是，这几个变量有不同的值。第二个也同样会混淆的问题是在程序的其他部分，其中几个不同的名字的变量用来指向同一个值。通过下面的讲述，这两个问题将可以得到解决。

25.1 Parameter Passing

为了理解程序的两个部分是如何协同工作的，先要理解 C 语言本身如何处理这些混乱问题的，而只有理解了这种处理方式，才可以单独考虑每个函数，而只有了解了函数调用中的实际参数值和函数中用来保存这个值得变量在语义上的区别，才可以弄清楚以下两个问题：

- C 语言是如何使同一个名字有不同的值；
- C 语言是如何使得同一个概念值用不同的名字表示。

当主程序调用 Factorial（它本身是作为函数 printf 的实际参数）时，实际参数是表达式 i，当这个语句被执行时，其效果相当于检查 i 的当前值，并把它传给函数 Factorial，它的原型为：

```
int Factorial(int n);
```

原型中声明了一个整型变量 n——该变量作为实际参数的占位符。在函数头部定义的作为占位符的变量被称为形式参数（formal parameter）。

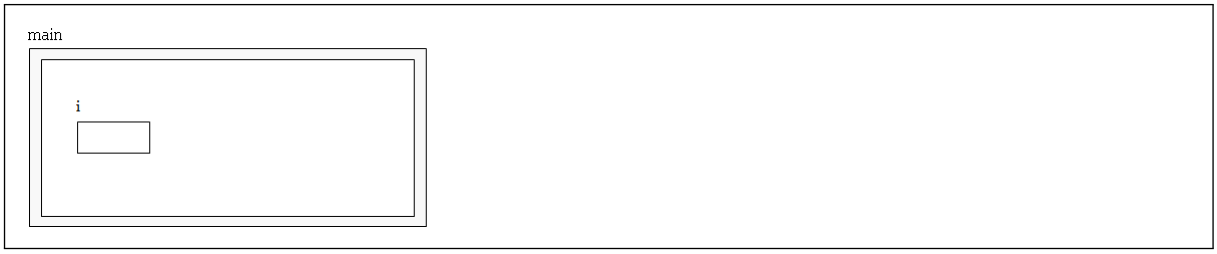
当函数被调用时，将执行下列步骤：

1. 计算每个实际参数的值作为调用程序操作的一部分，因为实际参数是表达式，所以这个计算可以包括运算符和其他函数。在新的函数真正被调用前，要计算出这些值。
2. 每个实际参数值被复制到对应的形式参数变量中。如果有多个实际参数，必须按次序将其复制到形式参数。第一个实际参数复制到第一个形式参数，以此类推。如果有必要的话，在实际参数值和形式参数之间要执行自动类型转换，就如同在赋值语句中一样。例如，如果 int 类型的值要传给形式参数声明为 double 类型的函数中，在把值复制到形式参数变量中之前，必须把该整型数转换为等价的双精度浮点数。
3. 执行函数体中的语句，直到遇见 return 语句为止。
4. 计算 return 中的表达式，如果需要的话，将表达式的值转换为函数指定的返回类型。
5. 在函数调用的地方用返回值替代，继续执行调用程序。

每次调用一个函数，都会产生一组变量。当变量以方框表示时，所有的方框都被包括在表示函数 main 的大方框中，而如果要对一个有多个函数的大程序进行跟踪的话，那么每次在一个函数调用另一个函数时，都要画一组新的变量方框。

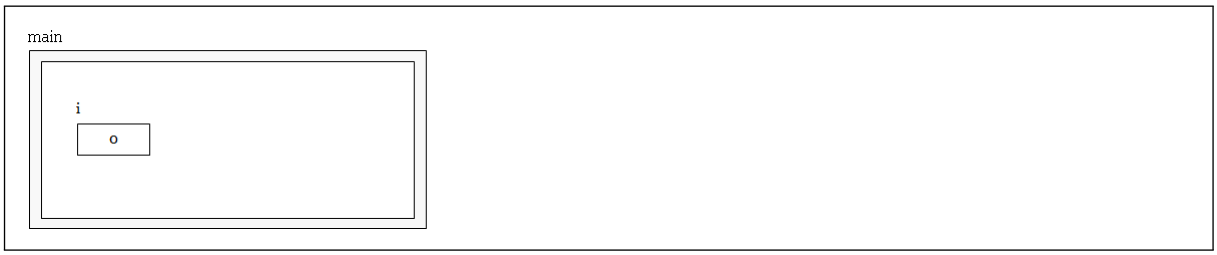
在函数中声明的每一个变量（包括形式参数）都必须有一个方框，这些变量仅在声明它们的函数中有意义，在函数外不能使用，因此被称为局部变量（local variable）。

在计算机内部，在函数内声明的的所有变量（包括参数）都保存在一个栈帧中。例如，在 fact.c 中的 main 函数运行时，首先需要为 main 函数中的变量创建空间，函数 main 只声明了一个变量（循环变量 i），因此 main 函数的变量可表示为下图的形式：

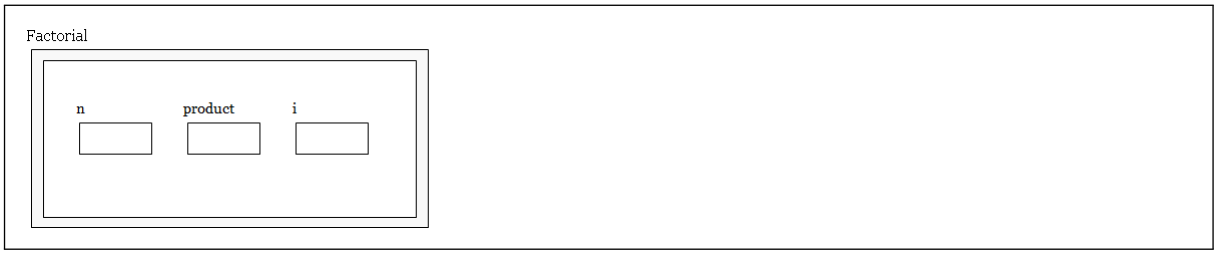


变量框外的双线将所有与特定函数调用相关的变量围起来，这个变量集就称为该函数的帧（frame）或栈帧（stack frame）。

在 main 函数执行时，for 循环的第一个周期中 i 的值为 0，可以把这个值放在帧中对应的变量框里以表示这个条件，然后 main 函数调用 printf，作为计算传递给函数 printf 的实际参数的一部分。



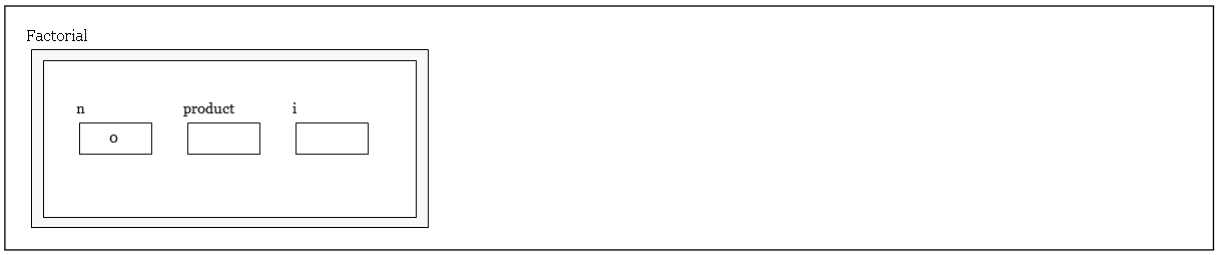
计算表达式 Factorial(i) 的值时，为了在帧图中表示计算机的动作，可以从查看当前帧中大的变量 i 的值开始，在当前帧中可以发现这个值为 0，然后为 Factorial 建立一个新帧，其中值 0 是第一个（也是仅有的一个）实际参数。Factorial 函数有三个变量：形式参数 n，局部变量 product 和 i，因此 Factorial 帧中有三个变量单元，如图所示：



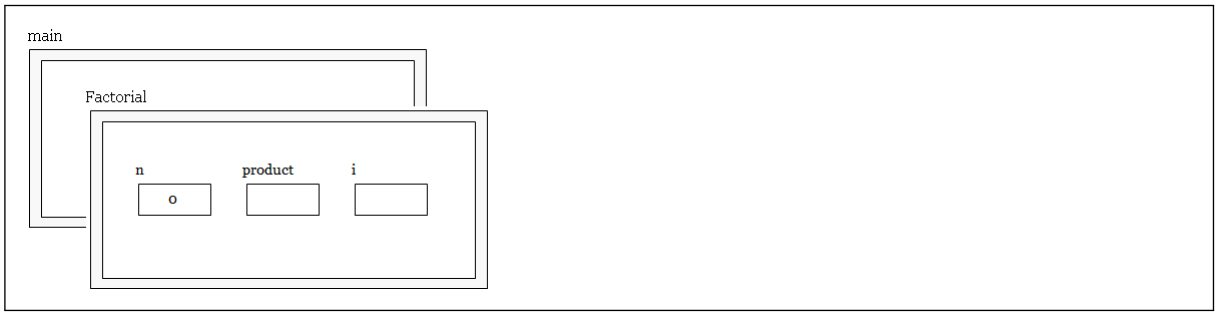
当主调函数将实际参数传给被调函数时，形式参数将用实际参数值（即 0）进行初始化，则 Factorial 帧中的内容如下所示：

当为 Factorial 创建一个新帧时，main 帧没有完全去掉，该帧暂时被放在一边，直到 Factorial 的操作结束为止。

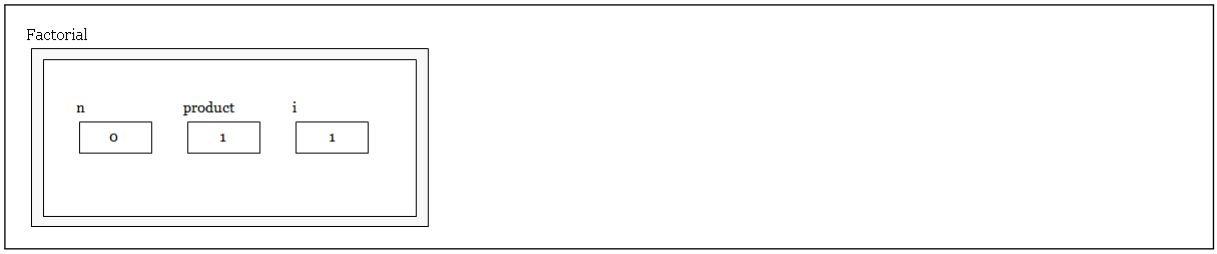
为了在用方框图表示的概念模型上表示这种情况，最好将每个新帧图画在一张索引卡上，然后将新帧放在表示原来的帧的卡上，即覆盖在它上面。例如，当调用函数 Factorial 时，表示 Factorial 帧的索引卡放在 main 帧上面，而整个帧集合形成了一个栈，最近的帧放在最上面，这就是栈帧这个术语的由来。



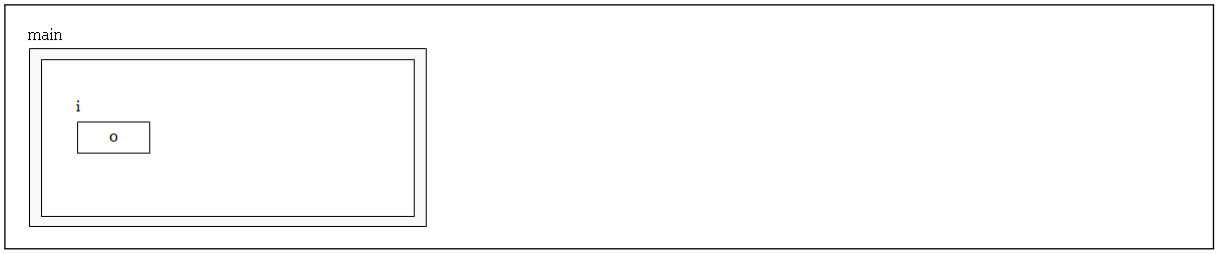
一旦 Factorial 被激活，main 帧仍然存在，只是现在看不到它的任何内容，这里要特别注意，名字 i 不再引用 main 函数中声明的变量，而只引用 Factorial 中的变量名 i。



下一步是执行 Factorial 的函数体，方法是在当前的帧上运行函数的每一步。变量 product 被初始化为 1，程序到达 for 循环。在 for 循环中，变量 i 被初始化为 1。但由于该变量值已经大于 n，所以 for 循环体没有被执行，因此，当程序到达 return 语句时，帧的内容变为：



当 Factorial 返回时，product 的值作为函数的结果值传回调用程序。从函数返回也意味着扔掉它的帧，使 main 中的变量重新显现出来，如下图所示：



然后，结果值 1 被传给 printf 函数，printf 函数经历同样的过程，然而操作的细节用户是不可见的，因为用户不知道 printf 在内部是如何工作的。最终，printf 函数将结果值 1 显示在屏幕上，然后返回。之后 main 函数继续执行 for 循环的下一周期。

当函数返回时，从调用点开始继续执行，这个调用点又称为返回地址，并将这个返回地址保存在栈帧中。

25.2 Nested Functions

函数的主要作用在于一旦一个函数被定义,不仅主程序可以使用它,而且它还可以作为实现其他函数的工具,使得这些函数可以用作为更复杂的工具,这些函数再被其他函数调用,以此类推,从而创建出具有任意复杂度的层次结构。

考虑这样的问题:从 n 个不同物体的集合中选出 k 个物体,有多少种不同的方法?

由这个问题引出的函数在数学上被称为组合函数(combination function),数学上通常记为:

$$\binom{n}{k}$$

或以函数符号表示为 $C(n,k)$。

可以推导出,组合函数可以用阶乘来定义:

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

为了使问题更加具体,假设桌子上的物体是五个硬币,分别是 1 美分、5 美分、1 角、25 美分和 5 角。现在要讨论的是从桌上去两个硬币有几种取法。如果将所有组合列出来,会发现有 10 中不同的组合。下面用数学计算验证:

$$C(5,2) = \frac{5!}{k! \times (5-2)!} = 10$$

如果用 C 语言实现这个函数,最好用一个比较长的名字,最好是用与数学中相同的函数名。由于函数调用通常出现在离函数定义很远的程序段中,而且在大的程序中很难找到某一函数的定义,因此通常选择的函数名要能传递有关函数的足够的信息。另外,局部变量仅用在一个简单的函数体内,因此可以很容易地看出它的作用。

给定 Factorial,直接把数学中的定义转换过来就可以实现组合函数[1]。

```
int Combinations(int n, int k)
{
  return (Factorial(n) / (Factorial(k) * Factorial(n-k)));
}
```

然后,可以写一个简单的主程序测试 Combinations 函数:

```
main()
{
  int n, k;

  printf( "Enter number of objects in the set (n)?" );
  n = GetInteger();
  printf( "Enter number to be chosen (k)?" );
  k = GetInteger();
  printf( "C(%d, %d) = %d\n" , n, k, Combinations(n, k));
}
```

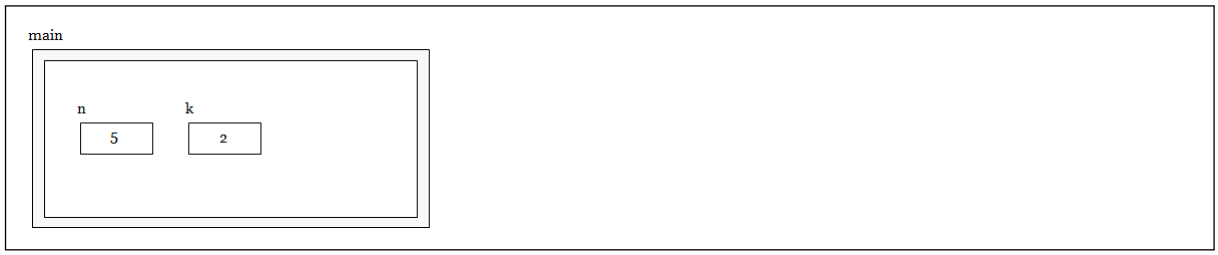
完整的 combine.c 程序如下:

```
/*
* File:combine.c
* --------------------
* This program tests a function to compute the mathematical combination
* function Combinations(n, k), which gives the number of ways to choose
* a subset of k objects from a set of n distinct objects.
```

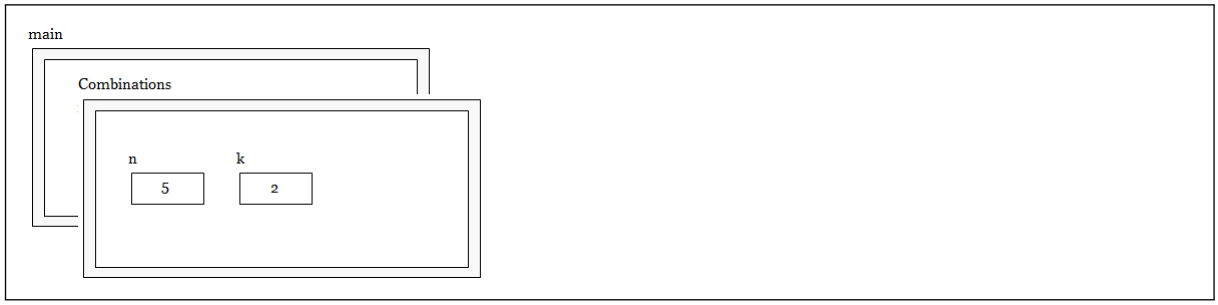
[1]关于阶乘的数学定义不能导出一个特别有效的 Combinations 函数的实现,实际上可以选择一个不同的实现策略。但是这里该定义的可读性更强,并且提供了一个机会来说明一个函数调用另外一个函数的机制。

```
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototype */

int Combinations(int n, int k);
int Factorial(int n);

/* Main Program */

main()
{
  int n, k;

  printf("Enter number of objects in the set (n)?");
  n = GetInteger();
  printf("Enter number to be chosen (k)?");
  k = GetInteger();
  printf("C(%d, %d) = %d\n", n, k, Combinations(n, k));
}

/*
* Function:Combinations
* Usage:ways = Combinations(n, k);
* ----------------------------------------
* Implements the Combinations function, which returns the number of
* distinct ways of choosing k objects from a set of n objects. In mathematics,
* this function is often written as C(n, k), but a function called C is not very
* self-descriptive, particularly in a language which has precisely the same name.
*/

int Combinations(int n, int k)
{
  return (Factorial(n) / (Factorial(k) * Factorial(n-k)));
}

/*
* Function:Factorial
* Usage:f = Factorial(n);
* -----------------------
* Returns the factorial of the argument n, where factorial is defined as the product
* of all integer from 1 up to n.
*/

int Factorial(int n)
{
  int product, i;

  product = 1;
  for(i = 1;i <= n; i++){
    product *= i;
  }
  return (product);
}
```

在程序 combine.c 中，为 main 函数建立一个帧，它声明了两个变量：n 和 k，当用户输入两个数后，程序到达了 printf 语句，这时帧中的变量值如下图所示：



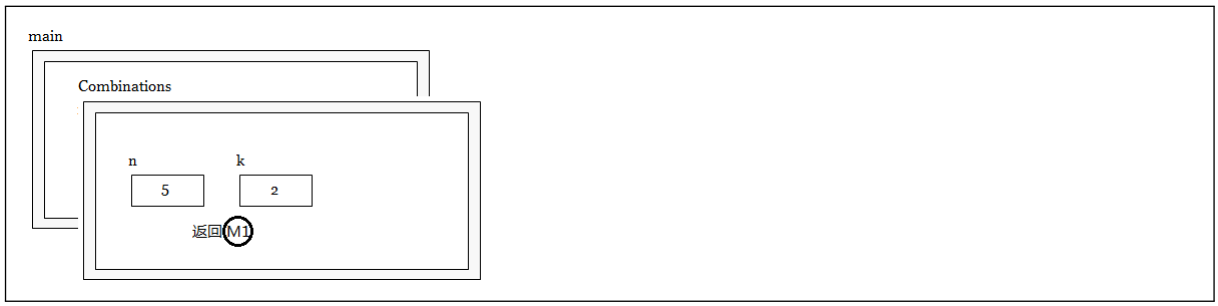
为了执行 printf 语句，计算机必须先算出调用 Combinations 函数得到的值，这将创建一个覆盖在原有帧上的一个新帧，而且每个新帧必须精确地记录在它做出调用前程序进行了哪些操作。



在 main 函数中出现了调用 Combinations 时，用 M1 标记如下：

```
main()
{
  int n, k;

  printf( "Enter number of objects in the set (n)?" );
  n = GetInteger();
  printf( "Enter number to be chosen (k)?" );
  k = GetInteger();
  printf( "C(%d, %d) = %d\n" , n, k, Combinations(n, k));
}
```

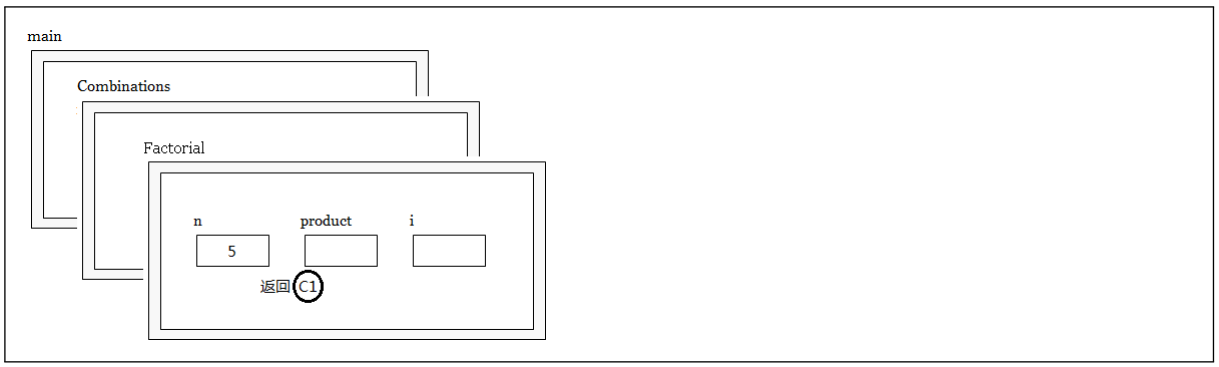
当计算机执行一个新的函数调用时，它记录了一旦调用完成后调用程序应该从哪里继续执行。执行的继续点被称为返回地址（return address），在这些图中用一个圆形标记指出。为了记住程序执行的位置，应该在帧图上记录调用点，如：



一旦创建一个新的帧，程序开始执行 Combinations 函数的函数体，它将用标记记录每个 Factorial 函数的调用：

```
int Combinations(int n, int k)
{
  return (Factorial(n) / (Factorial(k) * Factorial(n-k)));
}
```

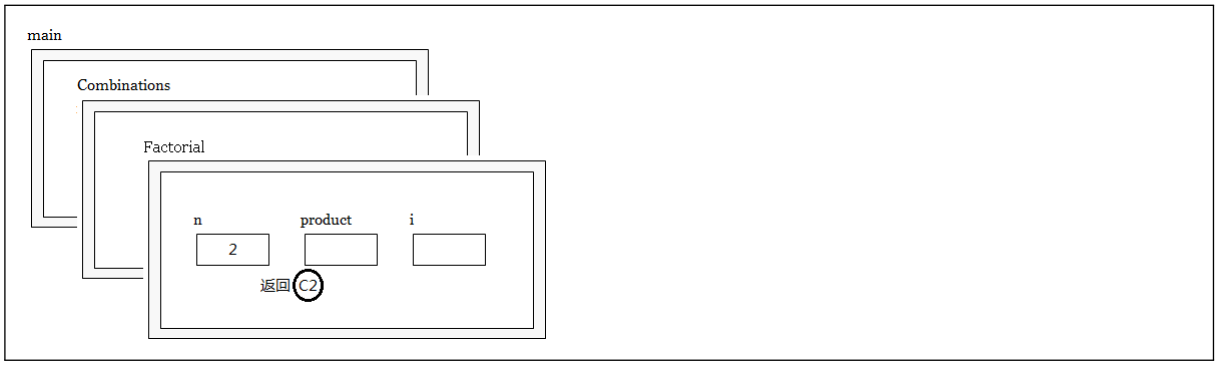
为了执行这个语句，计算机必须做出三个调用函数 Factorial 的标记。按照 ANSI C 制定的标准，编译器可以以任何次序做出这些调用。在这种情况下，假设选择按从左到右的顺序计算，第一个调用请求计算机计算 Factorial(n)，这将创建如下的新帧：



Factorial 函数执行后，返回 120 到标记 C1 指出的调用点。Factorial 函数的帧就消失了，回到 Combinations 的帧中，继续下一步计算，可以用返回并将调用结果填入调用的位置的方法来说明当前的状态，如下所示：

```
return (120/(Factorial(k) * Factorial(n-k)));
```

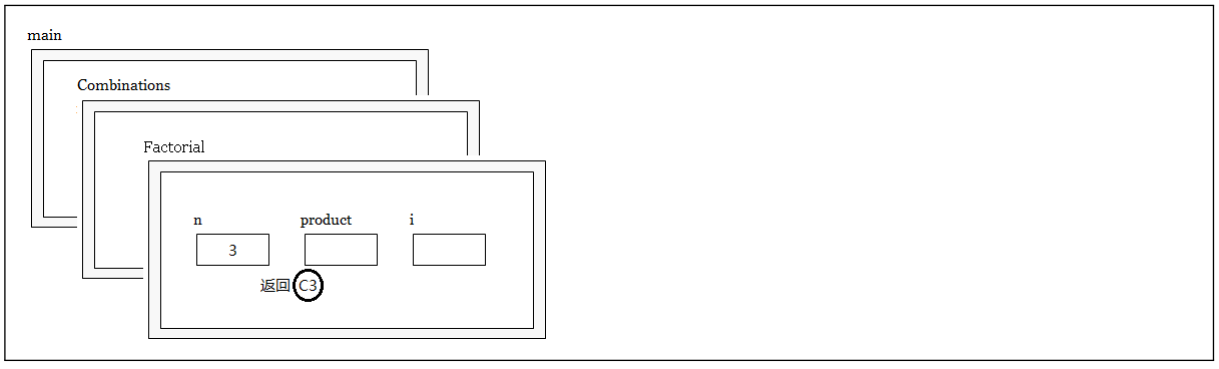
这里 120 外面的方框指出这个值不是程序的一个部分，而是一些以前计算的结果。从这一点开始，将继续计算第二个 Factorial 调用，其中实际参数为 k，因为 k 有值 2，这个调用将产生如下所示的帧：



Factorial 函数再一次执行它的操作，但没有进一步调用。该函数将 2! = 2 返回到调用点 C2，于是表达式当前的状态为：

```
return (120/(2 * Factorial(n-k)));
```

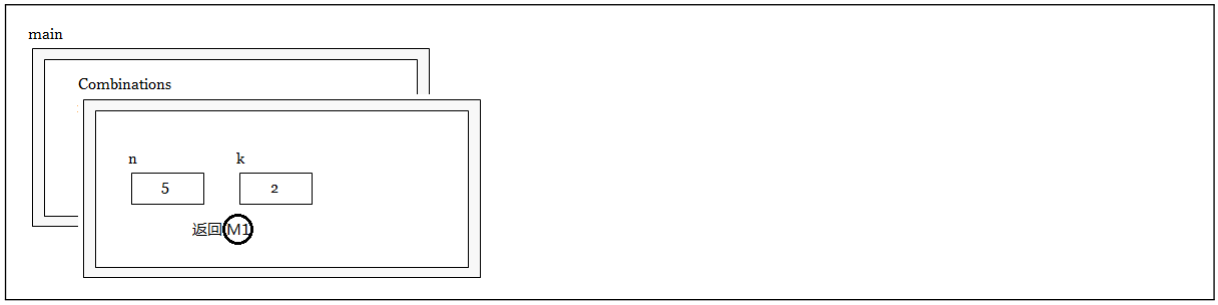
继续执行下一次 Factorial 函数的调用，它从计算表达式 n-k 开始，对于 n = 5，k = 2，于是又创建一个新帧：



从这个时刻开始，函数 Factorial 计算 (5 – 2)! 的值，并将结果值返回到调用程序的 C3 位置，这时表达式的状态为：

```
return (120/(2 * 6));
```

到现在，Combinations 函数获得了计算结果所需的所有值，求得的结果值为 10。为了找到这个结果的用途，需要咨询 Combinations 帧，它又重新出现在栈顶上，如下图所示：



这个帧指出，程序应该获取这个返回值，并用它替换 main 函数中的 M1 处的函数调用，此时已经计算出 Combinations(n, k) 的值为 10，于是可以用它置换 printf 语句中的 Combinations 的调用，从而得到如下状态：

```
printf( "C(%d, %d) = %d\n" , n, k, 10);
```

执行完这条语句后，printf 可产生如下输出：

```
C(5, 2) = 10
```

尽管利用这种深入剖析内部细节的方法可以理解 C 语言中的函数调用机制，但不要把它当成一种习惯。相反，更应该学会用非形式化的方法考虑函数，尝试从直观上理解函数是如何工作的。

当程序调用一个函数时，被调函数执行它自己的操作，然后程序从调用点继续执行，如果函数返回一个结果，调用程序在以后的计算中可以自由应用此结果。作为一个程序员，应该找到这个点，在此可以舒服地考虑这个过程，而不需要为细节问题而烦恼，计算机会处理这些细节。

第 26 章

Layout

C 语言源程序在经过编译预处理之后，会进入词法分析阶段，编译器将经过预处理的源文件中的字符进行分隔以形成一个个有意义的单元，这种单元叫做记号(token)，由此 C 语言源程序可以看作是一串记号。

记号可以理解为在不改变意义的基础上无法再进行分隔的字符组，运算符、标识符、关键字以及字符串字面量等都是记号。

在大多数情况下，C 语言允许在记号之间插入任意数量的间隔，这些间隔可以是空格符、知表符和换行符等。

- 实际上可以将 main 函数压缩在一行中。
- C 语言源程序不能写在一行中，因为每条预处理指令都要求独立成行。

虽然 C 程序中记号之间的空格数量没有严格要求，但不允许两个记号合并后产生新的记号的情况，也不允许在记号内添加空格。

针对 C 程序布局的规则如下：

- 语句可以划分在任意多行内；
- 记号间的空格应便于肉眼识别记号；
- 缩进有助于识别程序嵌套；
- 空行可以把程序划分成逻辑单元。

第 27 章

Callback

在计算机程序设计中，回调函数（Callback[1]）是指通过函数参数传递到其它代码的，某一块可执行代码的引用。这一设计允许了底层代码调用在高层定义的子程序。

回调的用途十分广泛。例如，假设有一个函数，其功能为读取配置文件并由文件内容设置对应的选项。若这些选项由散列值所标记，则让这个函数接受一个回调会使得程序设计更加灵活：函数的调用者可以使用所希望的散列算法，该算法由一个将选项名转变为散列值的回调函数实现；因此，回调允许函数调用者在运行时调整原始函数的行为。

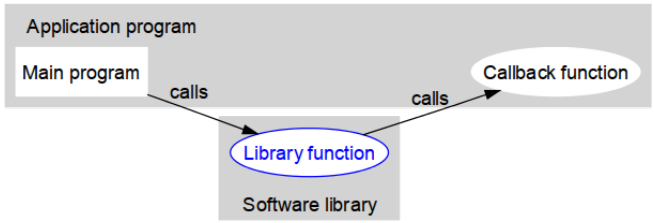


图 27.1: 回调通常与原始调用者处于相同的抽象层

回调的另一种用途在于处理信号或者类似物。例如一个 POSIX 程序可能在收到 SIGTERM 信号时不愿立即终止；为了保证一切运行良好，该程序可以将清理函数注册为 SIGTERM 信号对应的回调。

回调亦可以用于控制一个函数是否作为：Xlib 允许自定义的谓词用于决定程序是否希望处理特定的事件。

下列 C 语言代码描述了利用回调处理 POSIX 风格的信号（在本示例中为 SIGUSR1）的过程。值得注意的是，在处理信号的过程中，调用 printf(3) 是不安全的。

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig(int signum)
{
  printf("Received signal number %d!\n", signum);
}

int main(int argc, char *argv[])
{
  signal(SIGUSR1, sig);

  pause();

  return 0;
}
```

系统调用 pause(3) 会导致这个例子不做任何有意义的事，但这样做可以给你充分的时间来给这个

[1]Callback 即 call then back，或简称回调，被主函数调用运算后会返回主函数。

进程发送信号。(在类 Unix 系统上,可以调用 `kill -USR1 <pid>`,其中 <pid> 代表该程序的进程号。运行之后,该程序应当会有反应。)

注意,如果利用循环代替 pause() 会导致 CPU 占用率攀升到 100%。

回调的形式因程序设计语言的不同而不同。

- C, C++ and Pascal 允许将函数指针作为参数传递给其它函数。其它语言,例如 JavaScript, Python, Perl[1][2] 和 PHP,允许简单的将函数名作为参数传递。
- Objective-C 中允许利用 @selector 关键字传递 SEL 类型的函数名。在实现中,SEL 类型被定义为函数名字符串。
- 在类似于 C# 与 VB.NET 的运用.NET Framework 的语言中,提供了一种型别安全的引用封装,所谓的' 委托',用来定义包含类型的函数指针,可以用于实现回调。
- .NET 语言中用到的事件与事件处理函数提供了用于回调的通用语法。
- 函数式编程语言通常支持第一级函数,可以作为回调传递给其它函数,也可以作为数据类型存储或是返回给其它函数。
- 某些语言,比如 Algol 68, Perl, 新版本的.NET 语言以及多数函数式编程语言中,允许使用匿名的代码块(lambda 表达式),用以代替在别处定义的独立的回调函数。
- 在 Apple 或是 LLVM 的 C 语言扩展中,包含称为块的语言特性,可以作为函数的参数传递,作为回调的一种实现。
- 在缺少函数类型的参数的面向对象的程序语言中,例如 Java,回调可以用传递抽象类或接口来模拟。回调的接收者会调用抽象类或接口的方法,这些方法由调用者提供实现。这样的对象通常是一些回调函数的集合,同时可能包含它所需要的数据。这种方法在实现某些设计模式时比较有用,例如访问者模式,观察者模式与策略模式。
- C++ 允许对象提供其自己的函数调用操作的实现,即重载 operator()。标准模板库和函数指针一样接受这类对象(称为函数对象)作为各种算法的参数。

第 28 章

Recursion

28.1 Introduction

在数学与计算机科学中，递归(Recursion)是指在函数的定义中调用函数自身。递归一词还较常用于描述以自相似方法重复事物的过程。例如，当两面镜子相互之间近似平行时，镜中嵌套的图像是以无限递归的形式出现的。也可以理解为自我复制的过程。

关于递归的语言例子，可以简述如下：

> 从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……’”

递归经常被用于解决计算机科学的问题。在一些编程语言(如 Scheme 中)，递归是进行循环的一种方法。

下面是一个使用 Java 语言来实现的使用递归求 n 的阶乘的示例。

```
import java.util.Scanner;
  public class Factorial{
    public static void main(String[] args){
      Scanner keyboard = new Scanner(System.in);
      int n = keyboard.nextInt();
      System.out.println(factorial(n));
    }
    public static long factorial(int n){
      if (n==1){
        return 1;
      } else {
        return n * factorial(n-1);
      }
  }
}
```

如果用 C 语言来实现，可以根据公式 $n! = n \times (n-1)$ 来递归地计算出 $n!$ 的结果。

```
int fact(int n)
{
 if(n <= 1)
   return 1;
 else
   return n * fact(n-1);
}
```

一些编程语言(例如 Lisp)极度地依赖递归，而另一些语言不允许使用递归，C 语言介于中间。

为了观察递归的工作情况，下面来跟踪 fact 函数中语句的运行。

```
i = fact(3);
/*=========================*/
```

```
fact(3)发现3不是小于等于1的，所以fact(3)调用
  fact(2)，接着fact(2)发现2不是小于等于1的，所以fact(2)调用
    fact(1)，fact(1)发现1是小于等于1的，所以fact(1)返回1，从而导致
  fact(2)返回2x1=2，从而导致
fact(3)返回3x2=6
```

在 fact 函数最终传递 1 之前，未完成的 fact 函数的调用先“堆积”成几层。接下来，在最终传递 1 的那一点上，fact 函数的先前的调用开始逐个的“解开”，直到 fact(3) 的原始调用最终返回结果 6 为止。

把条件表达式放入 return 语句可以精简 fact 函数。

```
int fact(int n)
{
  return n <= 1 ? 1 : n * fact(n-1);
}
```

调用函数时，被调用的函数首先要仔细的测试“终止条件”，这样可以避免无限递归。

C 语言允许间接递归，即函数调用的同时又导致调用自身。只要主调函数和被调函数最终都可以终止，间接递归是合法的。

28.2 Definition

在数学和计算机科学中，递归指由一种(或多种)简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况。

例如，下列为某人祖先的递归定义：

- 某人的双亲是他的祖先(基本情况)。
- 某人祖先的双亲同样是某人的祖先(递归步骤)。

斐波那契数列是典型的递归案例：

- $F_0 = 0$(初始值)
- $F_1 = 1$(初始值)
- 对所有大于 1 的整数 n：$F_n = F_{n-1} + F_{n-2}$(递归定义)

尽管有许多数学函数均可以递归表示，但在实际应用中，递归定义的高开销往往会让人望而却步。例如：

- $0! = 1$(初始值)
- 对所有大于 0 的整数 n：$n! = n \times (n-1)!$(递归定义)

一种便于理解的心理模型，是认为递归定义对对象的定义是按照“先前定义的”同类对象来定义的。例如：你怎样才能移动 100 个箱子？答案：你首先移动一个箱子，并记下它移动到的位置，然后再去解决较小的问题：你怎样才能移动 99 个箱子？最终，你的问题将变为怎样移动一个箱子，而这是你已经知道该怎么做的。

如此的定义在数学中十分常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数。

以下是另一个可能更有利于理解递归过程的解释：

1. 我们已经完成了吗？如果完成了，返回结果。如果没有这样的终止条件，递归将会永远地继续下去。
2. 如果没有，则简化问题，解决较容易的问题，并将结果组装成原始问题的解决办法。然后返回该解决办法。

数学中常见的以递归形式定义的案例参见函数、集合以及分形等。

这样就有一种更有趣的描述：“为了理解递归，则必须首先理解递归。”或者更准确地，按照安德鲁·普洛特金的解释：“如果你已经知道了什么是递归，只需记住答案。否则，找一个比你更接近侯世达的人，然后让他/她来告诉你什么是递归。”

28.3 Applications

递归在计算机科学中是指一种通过重复将问题分解为同类的子问题而解决问题的方法。递归式方法可以被用于解决很多的计算机科学问题，因此它是计算机科学中十分重要的一个概念。

绝大多数编程语言支持函数的自调用，在这些语言中函数可以通过调用自身来进行递归。

计算机科学家尼克劳斯·维尔特如此描述递归：

> 递归的强大之处在于它允许用户用有限的语句描述无限的对象。因此，在计算机科学中，递归可以被用来描述无限步的运算，尽管描述运算的程序是有限的。
>
> ——尼克劳斯·维尔特

计算理论可以证明递归的作用可以完全取代循环，因此在很多函数编程语言（如 Scheme）中习惯用递归来实现循环。

在支持自调用的编程语言中，递归可以通过简单的函数调用来完成，如计算阶乘的程序在数学上可以定义为：

$$\text{fact(n)} = \begin{cases} 1, & \text{if n} = 0 \\ \text{n} \times \text{fact(n-1)}, & \text{if n} > 0 \end{cases}$$

这一程序在 Scheme 语言中可以写作：

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

28.3.1 Fixed-point combinator

即使一个编程语言不支持自调用，如果在这语言中函数是第一类对象（即可以在运行期创建并作为变量处理），递归可以通过不动点组合子（英语：Fixed-point combinator）来产生。以下 Scheme 程序没有用到自调用，但是利用了一个叫做 Z 算子（英语：Z combinator）的不动点组合子，因此同样能达到递归的目的。

[h!t]

```
(define Z
  (lambda (f)
    ((lambda (recur) (f (lambda arg (apply (recur recur) arg))))
     (lambda (recur) (f (lambda arg (apply (recur recur) arg)))))))

(define fact
  (Z (lambda (f)
       (lambda (n)
         (if (<= n 0)
             1
             (* n (f (- n 1))))))))
```

这一程序思路是，既然在这里函数不能调用其自身，我们可以用 Z 组合子应用 (application) 这个函数后得到的函数再应用需计算的参数。

28.3.2 Tail Recursion

尾部递归是指递归函数在调用自身后直接传回其值,而不对其再加运算。尾部递归与循环是等价的,而且在一些语言(如 Scheme 中)可以被优化为循环指令。因此,在这些语言中尾部递归不会占用调用堆栈空间。以下 Scheme 程序同样计算一个数字的阶乘,但是使用尾部递归:

[h!t]

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

28.3.3 Recursion Data

数据类型可以通过递归来进行定义,比如一个简单的递归定义为自然数的定义:“一个自然数或等于 0,或等于另一个自然数加上 1”。Haskell 中可以定义链表为:

[h!t]

```
data ListOfStrings = EmptyList | Cons String ListOfStrings
```

这一定义相当于声明“一个链表或是空串行,或是一个链表之前加上一个字符串”。可以看出所有链表都可以通过这一递归定义来达到。

28.3.4 Quicksort

没有函数会对递归情况进行多次,因为每个函数调用它自身只有一次。递归对要求函数调用自身两次或多次的复杂算法非常有帮助。

实际上,递归经常作为分治法(divide-and-conquer)技术的结果自然的出现。

分治法在算法设计中用于把一个大问题划分为多个较小的问题,然后采用相同的算法分别解决这些小问题。这其中的经典示例就是快速排序算法(quicksort)。

快速排序算法的操作如下(为了简化,假设要排序的数组的下标从 1 到 n):

1. 选择数组元素 e(作为“分割元素”),然后重新排列数组使得元素从 1 一直到 i-1 都是小于或等于元素 e 的,元素 i 包含 e,而元素从 i+1 一直到 e 都是大于或等于 e 的。
2. 通过递归地采用快速排序方法,对从 1 到 i-1 地元素进行排序。
3. 通过递归地采用快速排序方法,对从 i+1 到 n 地元素进行排序。

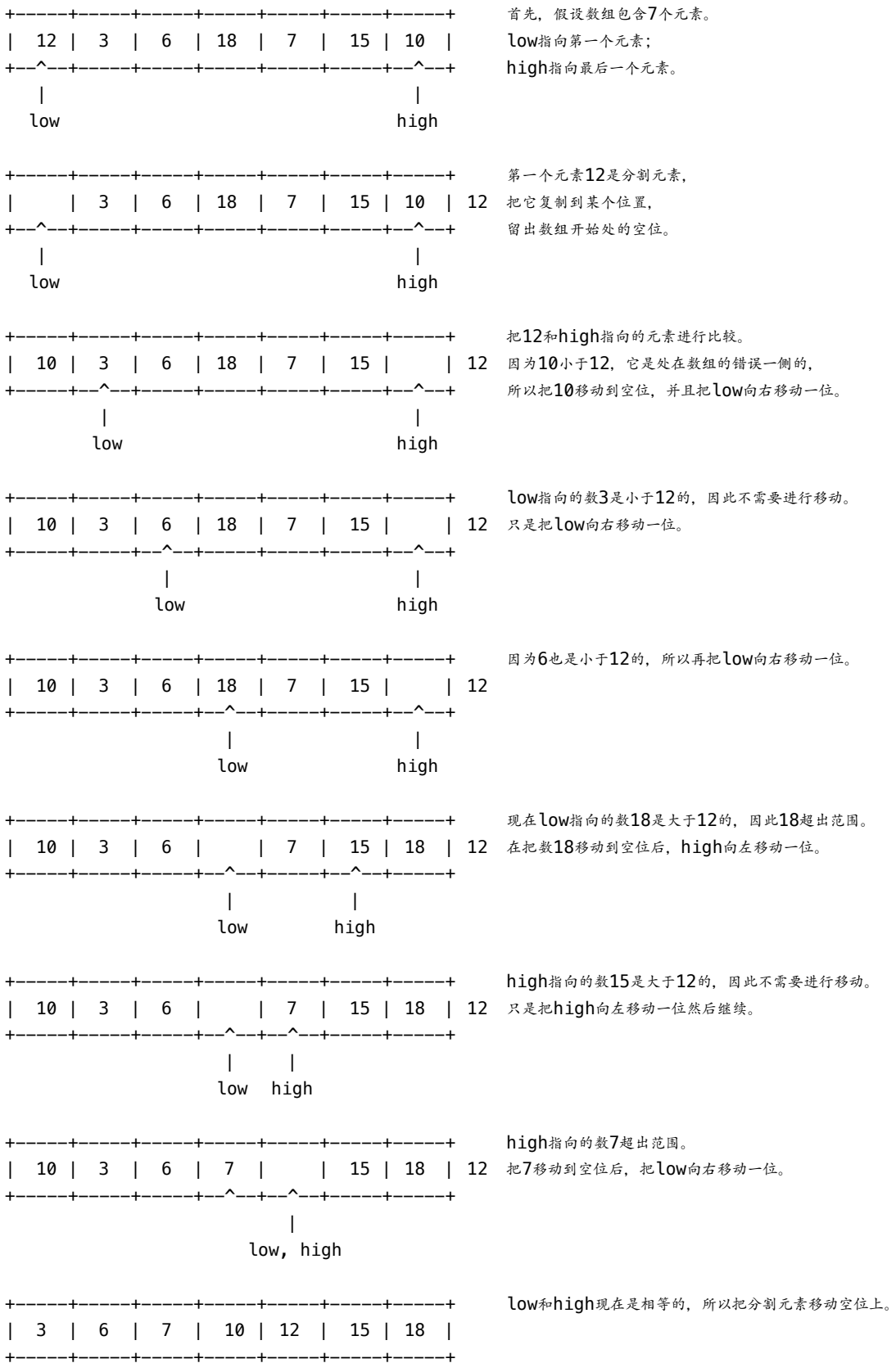
执行完第 1 步后,元素 e 处在正确的位置上。因为 e 的左侧的元素全部都是小于或等于 e 的,所以一旦第 2 步对这些元素进行排序,那些这些小于或等于 e 的元素也将会处在正确的位置上。类似的理由也可以应用于 e 右侧的元素。

显然,快速排序中的第 1 步是很关键的,有许多种方法可以用来分割数组,有些地方比其他的方法好。下面将采用的方法是很容易理解的,但是它不是特别有效,第一步将概括地描述分割算法。

快速算法依赖于两个名为 low 和 high 地标记,这两个标记用来跟踪数组内地位置。

开始,low 指向数组中地第一个元素,而 high 指向末尾元素。首先把第一个元素(分割元素)复制给其他地方的一个临时存储单元,从而从数组中留出一个“空位”。接下来,从右向左移动 high,直到 high 指向小于分割元素的数时停止。然后把这个数复制给 low 指向的空位,这将产生一个新的空位

(high 指向的)。现在从左向右移动 low, 寻找大于分割元素的数。在找到时, 把这个找到的数复制给 high 指向的空位。重复执行此过程, 交替操作 low 和 high 直到两者在数组中间的某处相遇时停止。此时, 两个标记都将指向空位, 只要把分割元素复制给空位就可以了。下面的示意图演示了这个过程。



此时就实现了快速排序的目标:分割元素左侧的所有元素都小于或等于 12, 而其右侧的所有元素

都大于或等于 12。完成分割数组之后，就可以使用快速排序算法对数组的前 4 个元素(10、3、6 和 7)和后 2 个元素(15 和 18)进行递归快速排序。

在把快速排序算法翻译成 C 代码时，首先开发一个递归函数 quicksort，该函数采用快速排序算法对数组元素进行排序，然后开发一个函数 split 用于分割数组。

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

main()
{
  int a[N], i;

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}

void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }
```

```
    a[high] = part_element;
    return high;
  }
```

虽然此版本的快速排序算法可行，但是它不是最好的，还有许多办法可以用来改进这个程序的性能。

- 改进分割算法。
  为了改进分割算法，可以不再选择数组中的第一个元素作为分割元素，较好的方法是取第一个元素、中间元素和最后一个元素的中间值。分割过程本身也可以加速。特别是，在两个 while 循环中避免测试 low < high 是可能的。
- 采用不同的方法进行小数组排序。
  不再递归地使用快速排序算法用一个元素全部下至数组尾，针对小数组地方法是采用较为简单地方法。
- 改进快速排序为非递归。
  虽然快速排序算法本质上是递归算法，并且递归格式的快速排序是最容易理解的，但是实际上若去掉递归会更有效率。

第 29 章

Top-down Design

利用过程和函数,可以将一个大的程序设计问题分解为较小的部分,从而使每一部分都比较容易理解。将一个问题划分为可管理的片段的过程被称为分解(decomposition),这个过程表示了程序设计的最基本的策略。

然而,找出正确的的分解是一个困难的任务,该任务需要相当多的实践,经过合理的分析,很好地选择了每一个片段后,这每个片段作为一个单元在概念上是完整的,并且程序作为一个整体很容易理解。如果选择得不好,将会妨碍分解的完成。

当遇到一个写程序的任务时,最好的策略一般是从编写主程序开始。从主程序的观点出发,将问题作为一个整体考虑,然后试着去找出整个任务的主要片段。一旦确定了程序的主要片段,可以沿着这条主线将整个问题继续分解为独立的组件。一般这些组件中的某一些本身还很复杂,所以通常将它们分解成更小的片段,一直持续到每个程序片段都足够简单,可以独立解决为止,这个过程被称为自顶向下[1]的设计(top-down design),或逐步求精(stepwise refinement)。

从程序的概要描述开始,通过把概要中的每个片段再分解成更简单的步骤来不断精化每一个片段,这样可以通过编写一组函数来解决一个复杂问题,其中的每一个函数都足够简单。考虑如下的问题:

开发一个显示日历的程序,日历按完整的一年来运行,按下面的格式显示每一个月:

```
                February      1992
Su    Mo    Tu    We    Th    Fr    Sa
                                    1
2     3     4     5     6     7     8
9     10    11    12    13    14    15
16    17    18    19    20    21    22
23    24    25    26    27    28    29
```

该实现必须有足够的智能,例如,知道 1992 年 2 月 1 日是星期六,知道闰年的 2 月份有 29 天。此外,程序要以正确的排列形式将数据显示在屏幕上,等等。

29.1 Decomposition

从主程序开始考虑,首先要做的分析是程序要做什么?一般说来,日历程序从用户那里读入年份,然后展示这一年的日历。这里,给用户提供一些提示也是一个好主意,特别是如果程序在用户必须了解的特殊情况或限制下结束时更是如此。

逐步求精的原则指出,一旦完成了某个程序的概要描述,就应该在此结束,并把它写下来,调用过程或函数表示那些还要继续细化的程序。

如果采用这种方法,主程序可以写成:

```
main()
{
  int year;
```

[1] 1969 年, Wirth 提出采用“自顶向下逐步求精、分而治之”的原则进行大型程序的设计,其基本思想是从欲求解的原问题出发,运用科学抽象的方法,把它分解成若干相对独立的小问题,依次细化,直至各个小问题获得解决为止。

```
 GivenInstructions();
 Year = GetYearFromUser();
 PrintCalendar(year);
}
```

第一个语句调用 GivenInstructions 过程，用于给用户提供一些指示，第二个语句调用函数 GetYearFromUser，该函数可以用来处理读入的年份的过程。GetYearFromUser 的实现可以简单地表示为显示一个提示符，并调用 GetInteger 函数，但也可以包括一些其他的操作，如检查用户输入的年份是否合法。主程序的最后一行调用函数 PrintCalendar，给它传递所需的年份作为参数，最后一个函数的实现将涉及解决整个问题的绝大部分工作。

在上述 main 实现所示的详细级别上，程序已经具有完整的意义，这些工作是成功分解的关键，只要继续向下分解，就会发现新的过程和函数来解决这些任务中的新的有用的片段，然后，根据这些片段实现解决方案层次结构中的每一层。

29.2 PrintCalendar

从 main 程序的调用判断，PrintCalendar 的原型是：

```
void PrintCalendar(int year);
```

在抽象级别考虑 PrintCalendar 函数，考虑它要完成的功能，它通过打印 12 个单独的月份完成日历的显示，利用这些想法足以编写一个适合这个分解层次的该函数的实现。PrintCalendar 的函数体是一个简单的循环，它调用另外一个函数显示某一个月的月历，然后打印一个换行符使得该月与下一个月之间有一个空行。

```
void PrintCalendar(int year)
{
 int month;

 for(month = 1; month <= 12; month ++){
  PrintCalenderMonth(month, year);
  printf("\n");
 }
}
```

其中，PrintCalendarMonth 函数有两个参数，月份和年份。因为这两个信息它都需要，其中参数 month 让函数知道要显示哪一个月，因为特定月份的月历随着年份的不同而不同，因此需要参数 year。

29.3 PrintCalendarMonth

实现 PrintCalendarMonth 的困难在于如何用下面的格式显示一个月的日历：

```
              Februay     1992
Su    Mo    Tu    We    Th    Fr    Sa
                                    1
2     3     4     5     6     7     8
9     10    11    12    13    14    15
16    17    18    19    20    21    22
23    24    25    26    27    28    29
```

为了把取得的数字值 month 转换为普通的名字，可以使用 switch 语句来完成该功能，然后为了显示月历的头两行，可以用如下的语句实现：

```
printf(" %s       %d\n", MonthName(month), year);
printf("Su Mo Tu We Th Fr Sa\n");
```

月历的其余部分由 1 到这个月的天数之间的整数组成，使用 for 循环可以显示，关键在于如何以规定的格式显示，一方面，该月份必须从一个星期中的某一天开始，另一个方面，在每个周六之后，输出必须换行。

为了解决格式问题，必须记录周中的日子及月中的日子。对于如何显示每周中的日子，一种方法是对它进行简单的编号，给定年份的布局，最常用的编号方式是定义星期天为 0，星期一为 1，以此类推，直到星球六，它被编号为 6，可以选择把这些值定义为常量：

```
#define Sunday    0
#define Monday    1
#define Tuesday   2
#define Wednesday 3
#define Thursday        4
#define Friday    5
#define Saturday        6
```

这样，在程序中可以用这些日子的名字来表示，从 0 开始给每周中的日子编号有一些好处。

可以用取模运算实现安兴起的循环操作，如果变量 weekday 保存对应于该周中当前日子的整数，表达式 (weekday + k) % 7 指出 k 天后的那一天是星期几。例如，如果今天是星期五（weekday 的值是 5），今天后的第 10 是星期一，因为表达式 (5+ 10) % 7 = 1。特别是，可以用这个公式写下列语句，它表示移到本周中的下一天：

```
weekday = (weekday + 1) % 7;
```

但类似的表达式：

```
weekday ++; /* ++运算符使值超过6 */
```

用来这里就不太合适。通过除以 7 取余可以保证结果值总是在 0 6 之间。当采用取余操作将计算的结果限制在一个较小的周期性的范围内时，使用的是数学上称为取模运算（modular arithmetic）的过程。

如果保存了星期几的记录，那么在 PrintCalendarMonth 函数中通过一个主循环可以完成星期几与日期的对应：

```
for(day = 1; day <= nDays; day ++){
  printf("%2d",day);
  if(weekday == Saturday) printf("\n");
  weekday = (weekday + 1) % 7;
}
```

该循环显示了每一个数，记录了它是星期几，在每个星期六之后换行，日历的最后一行必须以换行结束，因此循环后还必须紧跟以下的语句：

```
if(weekday != Sunday) printf( "\n" );
```

这样就保证即使这个星期不是在星期日结束，最后一行后面也有换行符。至此，还剩下三个任务：

1. 计算出该月的天数；
2. 确定该月的第一天是星期几；
3. 缩排月历的第一行，使得第一天出现在正确的位置。

逐步求精的策略建议不要在分解的这个层次上解决这三个问题，而且事实上，可以将这三个操作转化为函数调用，应用这个策略，可以写一个完整的 PrintCalendarMonth 的实现：

```
void PrintCalendarMonth(int month, int year)
{
  int weekday, nDays, day;

  printf(" %s       %d\n", MonthName(month), year);
  printf("Su Mo Tu We Th Fr Sa\n");
  nDays = MonthDays(month, year);
  weekday = FirstDayOfMonth(month, year);
```

```
  IndentFirstLine(weekday);
  for(day = 1; day <= nDays; day ++){
   printf( "%2d" ,day);
   if(weekday == Saturday) printf("\n");
   weekday = (weekday + 1) % 7;
  }
  if(weekday != Sunday) printf("\n");
}
```

用逐步求精的方法实现 PrintCalendarMonth 意味着将其中会用到的三个函数（MonthDays、FirstDayOfMonth 和 IndentFirstLine）的实现放到以后再写。

其中最容易实现的函数是这段函数中的最后一个：IndentFirstLine，这个函数从 FirstDayOfMonth 中获取一个星期中的第几天，保证月历的第一行前有足够的空格，使第一天出现在正确的位置。如果这个月从星期天开始，月历将从第一行的起始位置开始，如果从星期一开始，程序将打印出表示一天所需的空格来表示缺少的星期天。因为每个月历上的项目占用了三个空格，所以 IndentFirstLine 的实现可以表示为一个循环，该循环为每一个缺少的日子打印三个空格，从一个星期的开头开始计算，下面的程序完成了这个任务。

```
void IndentFirstLine(int weekday)
{
  int i;

  for(i = 0; i < weekday; i++){
   printf("  ");
  }
}
```

在余下的两个函数中，编写 MonthDays 相对比较简单。通过下面这条规律可以顺利写出程序：

```
一三五七八十腊
三十一天永不差
四六九冬三十天
惟有二月二十八（九）
```

上述规律列出了一些独立的情况，建议用 switch 语句，这个函数的完整的实现如下：

```
int MonthDays(int month, int year)
{
  switch (month){
   case 2:
    if(IsLeapYear(year)) return (29);
    return (28);
   case4: case 6: case 9: case 11:
    return (30);
   default:
    return (31);
  }
}
```

由于直接计算某一天是星期几比较困难，一个简单可行的方法是取出历史上的任意一天，从那天开始向前计算。计算机的处理速度是相当快的，使用者不会发现有延迟，例如，1900 年 1 月 1 日是星期一，从那天开始，按是否为闰年，每年加 365 天或 366 天。

对于当年的要处理的月份之前的月，加上这个月的天数。用取模运算完成这些计算，并取除以 7 之后的余数，该程序可以计算任何 1900 年以后的某一天是星期几，其实现如下：

```
int FirstDayOfMonth(int month, int year)
{
  int weekday, i;
```

```
  weekday = Monday;
  for(i = 1900; i < year; i++){
    weekday = (weekday + 365) % 7;
    if(IsLeapYear(i)) weekday = (weekday + 1) % 7;
  }
  for(i = 1; i < month; i++){
    weekday = (weekday + MonthDays(i, year)) % 7;
  }
  return (weekday);
}
```

在结束了前面的编码后，完成该程序只需要填充一些细节，例如要实现函数 GivenInstructions 和 GetYearFromUser。

由于 FirstDayOfMonth 的定义要求处理的年份不能早于 1900 年，因此，最好能在这些函数中做一些限制，强迫满足这个要求。例如下面的 GetYearFromUser 的实现检查输入的年份是否满足这个条件，如果不满足，则给用户一个重新输入的机会。

```
int GetYearFromUser(void)
{
  int year;

  while(TRUE){
    printf( "Which year?" );
    year = GetInteger();
    if(year >= 1900) return(year);
    printf( "The year must be at least 1900.\n" );
  }
}
```

剩下的工作还需要从前面的例子中拷贝 IsLeapYear 和 MonthName 函数的实现，确保包括所有的函数原型、写注释、最终编译和测试程序。

下面是 calendar.c 的完整代码：

```
/*
* File:calendar.c
* ---------------
* This program is used to generate a calendar for a year entered by the year.
*/

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
* Constants:
* ----------
* Days of the week are represented by the integers 0~6.
* Months of the year are identified by the integers 1~12.
* Because the numeric representation for months is in common use, no special
* constants are defined.
*/

#define Sunday    0
#define Monday    1
#define Tuesday   2
#define Wednesday 3
#define Thursday  4
#define Friday    5
```

```
#define Saturday        6

/*    Function prototypes */

void GivenInstructions(void);
int GetYearFromUser(void);
void PrintCalendar(int year);
void PrintCalendarMonth(int month, int year);
void IndentFirstLine(int weekday);
int MonthDays(int month, int year);
int FirstDayOfMonth(int month, int year);
string MonthName(int month);
bool IsLeapYear(int year);

/*    Main Program */

main()
{
  int year;

  GivenInstructions();
  Year = GetYearFromUser();
  PrintCalendar(year);
}

/*
* Function:GivenInstructions
* Usage:GivenInstructions
* ----------------------------------
* This procedure prints out instructions to the user.
*/

void GivenInstructions(void)
{
  printf( "This program displays a calendar for a full year.\n" );
  printf( "The year must not be before 1900.\n" );
}

/*
* Function:GetYearFromUser
* Usage:year = GetYearFromUser();
* ----------------------------------------
* This function reads in a year from the user and returns that value.
* If the user enters a year before 1900,the function gives the user another chance.
*/

int GetYearFromUser(void)
{
  int year;

  while(TRUE){
    printf( "Which year?" );
    year = GetInteger();
    if(year >= 1900) return(year);
    printf( "The year must be at least 1900.\n" );
  }
}
```

```
/*
* Function:PrintCalendar
* Usage:PrintCalendar(year);
* ----------------------------------
* This procedure prints a calendar for an entire year.
*/

void PrintCalendar(int year)
{
  int month;

  for(month = 1; month <= 12; month ++){
    PrintCalenderMonth(month, year);
    printf( "\n" );
  }
}

/*
* Function:PrintCalendarMonth
* Usage:PrintCalendarMonth(month, year);
* ------------------------------------------------
* This procedure prints a calendar for the given month and year.
*/

void PrintCalendarMonth(int month, int year)
{
  int weekday, nDays, day;

  printf( "  %s       %d\n" , MonthName(month), year);
  printf( "Su Mo Tu We Th Fr Sa\n" );
  nDays = MonthDays(month, year);
  weekday = FirstDayOfMonth(month, year);
  IndentFirstLine(weekday);
  for(day = 1; day <= nDays; day ++){
    printf( "%2d" ,day);
    if(weekday == Saturday) printf( "\n" );
    weekday = (weekday + 1) % 7;
  }
  if(weekday != Sunday) printf( "\n" );
}

/*
* Function:IndentFirstLine
* Usage:IndentFirstLine(weekday);
* ---------------------------------------
* This procedure indents the first line of the calendar by printing enough
* blank spaces to get to the position on the line corresponding to weekday.
*/

void IndentFirstLine(int weekday)
{
  int i;

  for(i = 0; i < weekday; i++){
    printf( "  " );
  }
}
```

```
/*
* Function:MonthDays
* Usage:nDays = MonthDays(month, year);
* ------------------------------------------------
* MonthDays returns the number of days in the indicated month and year.
* The year is required to handle leap years.
*/

int MonthDays(int month, int year)
{
  switch (month){
    case 2:
      if(IsLeapYear(year)) return (29);
      return (28);
    case4: case 6: case 9: case 11:
      return (30);
    default:
      return (31);
  }
}

/*
* Function:FirstDayOfMonth
* Usage:weekday = FirstDayOfMonth(month, year);
* ------------------------------------------------------------
* This function returns the day of the week on which the indicated month begins.
* This program simply counts forward from January1, 1900, which was a Monday.
*/

int FirstDayOfMonth(int month, int year)
{
  int weekday, i;
  weekday = Monday;
  for(i = 1900; i < year; i++){
    weekday = (weekday + 365) % 7;
    if(IsLeapYear(i)) weekday = (weekday + 1) % 7;
  }
  for(i = 1; i < month; i++){
    weekday = (weekday + MonthDays(i, year)) % 7;
  }
  return (weekday);
}


/*
* Function:MonthName
* Usage:name = MonthName(month);
* ------------------------------------------
* MonthName converts a numeric month in the range 1~12 into
* the string name for that month.
*/

string MonthName(int month)
{
  switch(month){
    case 1: return ( “January” );
    case 2: return ( “February” );
    case 3: return ( “March” );
```

```
    case 4: return ( "April" );
    case 5: return ( "May" );
    case 6: return ( "June" );
    case 7: return ( "July" );
    case 8: return ( "August" );
    case 9: return ( "September" );
    case 10: return ( "October" );
    case 11: return ( "November" );
    case 12: return ( "December" );
    default: return ( "Illegal month" );
  }
}

/*
* Function:IsLeapYear
* Usage:if(IsLeapYear) . . .
* --------------------------------
* This function returns TRUE if year is a leap year.
*/

bool IsLeapYear(int year)
{
  return (((year % 4 == 0) && (year % 100 == 0)) || (year % 400 == 0));
}
```

Part IV

Compound Type

第 30 章

Overview

和控制结构一样，数据结构也是程序设计的中心。通过将数据组装成较大的结构，能极大地提高程序设计的能力，并能写出更有用和更激动人心的程序。

使用复杂的算法对简单数据进行操作时，随着算法的具体化，运行这些算法的程序变得庞大和复杂。

为了降低算法的复杂性，程序员使用逐步求精的策略把一个大程序分成更小的、更容易管理的函数。当这些函数变得很复杂时，它们又会被分成更小、更简单的函数。

完整的过程式解决方案采取层次结构的形式，高一级的函数会调用低一级的函数，低一级的函数会调用更低一级的函数，直到函数不需要调用别的函数就可以直接执行为止。

另外，数据类型也会形成一个层次结构，可以用简单数据类型来定义复杂的数据类型。只要程序需要，这些复杂的数据类型又可以用来定义更复杂的数据类型。

通过简单数据类型来定义新的数据类型的主要好处，是可以把一些独立的数值联系起来，从而形成有机的集合。将一组数据作为一个整体来考虑，可以极大地减少程序概念上的复杂性。

程序中的控制语句和函数调用定义了算法控制结构，数据类型定义的层次构成了数据结构（data structure），这两个概念（控制结构和数据结构）共同构成了现代程序设计的基础。

在 1976 年，程序设计语言 Pascal 的创始者 Niklaus Wirth 在一篇程序设计方面的文章的标题中把这种原理以方程的形式写出：

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

第 31 章

Array

31.1 Introduction

C 语言支持聚合(aggregate)变量,这类变量可以存储数值的集合。

C 语言提供了两种聚合类型:数组(array)和结构(struct),其中数组用来存储具有同一数据类型并且按顺序排列的一组数据。

数组在各种计算机编程语言中的表示式略有不同,但是几乎每一种编程语言都有这种结构和观念。现在,数组已经不只是一种编程专用的术语,而是计算机运作中非常重要的技术和概念。

普通数组采用一个整数作为下标,而在多维数组采用一系列有序的整数来标注,这种整数列表之中整数的个数始终相同,并被称为数组的“维度”。

简单数组强烈依赖计算机硬件的内存,因而不适用于现代的程序设计。如果需要使用可变大小、硬件无关性的数据类型,Java 等程序设计语言均提供了更高级的数据结构——ArrayList、Vector 等动态数组。

当前计算机能够显示、打印中文字符都是数组的应用,字符串也是基于数组的一种重要数据结构。数据库也是数组概念的一种扩充和延伸。

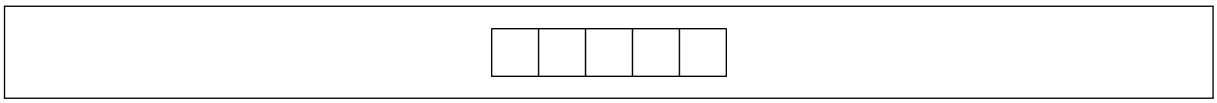
31.2 Definition

C 语言提供了几种根据已存在的数据类型定义新的数据类型的方法,其中数组是程序设计中最普通的复合结构。

数组(array)是一些独立数值的集合,它具有两个显著的特征:

1. 数组是有序的。
   必须能把数组中的每个分量按顺序排列,
2. 数组是同质的。
   数组中的每个数值必须有同样的类型,因此可以定义一个整型数组或是一个浮点型数组,但是数组中允许有两种数据类型。

直观上,我们可以把数组看成一系列的方框,数组中的每个值占一个方框,这样每一个值称为一个元素(element)。例如,下图代表一个有五个元素的数组。



31.3 Declaration

在设计之初,数组是在形式上依赖内存分配而成的,所以必须在使用前预先请求空间,这使得数组有以下特性:

1. 请求空间以后大小固定,不能再改变(数据溢出问题);

2. 在内存中有空间连续性的表现，中间不会存在其他程序需要调用的数据；
3. 在 C 语言中，程序不会对数组的操作做下界判断，也就有潜在的越界操作的风险（比如会把数据写在运行中程序需要调用的核心部份的内存上）。

和 C 语言中其他的变量一样，数组在使用前必须声明。下面的语法框中给出数组声明的一般形式。

语法：数组声明

```
elementtype arrayname[size];
```

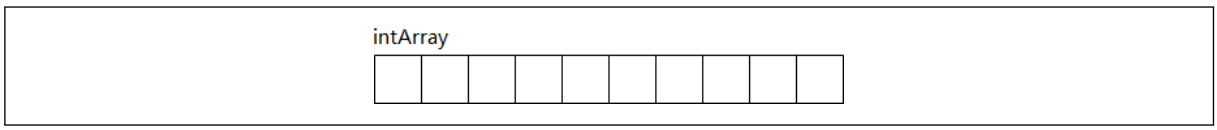
其中，elementtype 是数组中每个元素的类型，arrayname 是声明为一个数组的变量名，size 是数组的元素数目。

通过下面的语句

```
int intArray[10];
```

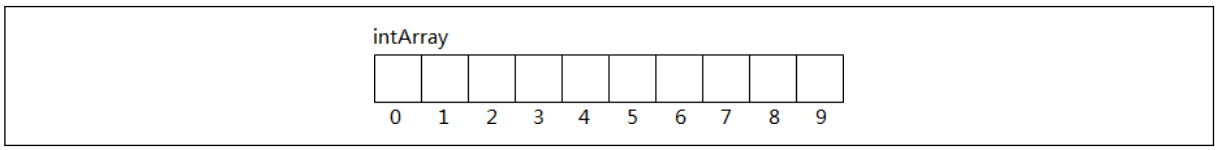
声明了一个名为 intArray，且拥有 10 个元素的数组，每个元素的类型都是 int 型的。

在描述上述数组定义时，可以画出 10 个方框，且命名为 intArray：



为了存取特定的数组元素，可以将数组中的每一个元素都通过对应的下标（index）确定。

在 C 语言中，数组的下标始终从 0[1]开始，一直到比数组大小小 1 的数为止，因此长度为 n 的数组元素的索引是从 0 到 n-1。例如，有 10 个元素的数组的下标就是 0、1、2、3、4、5、6、7、8 和 9。



当声明一个数组变量时，数组的元素可以是任何类型，而且必须以常量或常量表达式来指定数组的大小。

数组的长度可以用任何（整数）常量表达式指明。对于数组长度可变的应用来说，常用的方法是在数组中分配足够的空间，使之能保存最大的数据量，而实际使用的只是其中的一部分。

- 元素的最大个数称为数组的分配长度。
- 实际使用的元素个数称为有效长度。

在程序中经常需要调整数组的的长度，因此应该使用宏（一个符号化的常量）而不是一个具体的数值来指定数组的大小。例如，定义数组不能用下列方法：

```
int intArray[10];
```

而应该将大小定义成宏：

```
#define NElements 10
```

然后把数组声明改写为：

```
int intArray[NElements];
```

数组选择表达式是左值，所以数组元素可以和普通变量一样使用。

```
intArray[0] = 1;
printf("%d\n", intArray[9]);
++intArray[i];
```

[1]数组下标从 0 开始可以使编译器简化，也可以提升数组下标运算的速度。

数组选择表达式可能会产生副作用。追踪下面的示例代码后可以发现，在把变量 i 设置为 0 后，while 语句判断变量 i 是否小于 N。如果是，那么将数值 0 赋值给 a[0]，随后 i 自增，然后重复循环。但是，如果使用 a[++i] 就会出现问题，这样在第一次循环操作期间将会把 0 赋值给 a[1]。

```
i = 0;
while(i < N)
  a[i++] = 0;
```

同样地，下面这个循环在把数组 b 复制给数组 a 时也可能会无法工作。

```
i = 0;
while(i < N)
  a[i] = b[i++];
```

在每次对 a[i] 赋值之前，必须确定好对应于 a[i] 和 b[i++] 的内存位置。一种可能是将会首先确定好对应 a[i] 的内存位置以便把 b[i] 复制给 a[i]，另一种情况则可能会首先确定好对应 b[i++] 的内存位置，然后变量 i 自增后才把 b[i] 的值复制给 a[i]。

为了避免数组下标产生的副作用，可以将下标中的自增操作符移走。

```
for(i=0; i<N; i++)
  a[i] = b[i];
```

与其他任何变量一样，数组命名遵循变量命名的一般规则，从而可以用数组的名字向程序的阅读者说明存储的是何种类型的数据。

例如，如果想定义一个数组，存储运动会上裁判团给出的分数（如体操等）。每个裁判的评分是从 0 ～ 10，10 是最高分。因为分数可能会是小数（如 9.9），所以数组元素的类型必须设置成 double 型（即标准的浮点类型），使用语句：

```
double scores[NJudges];
```

声明了一个数组，名字为 scores，每个元素的类型是 double，用以存放每个裁判给出的分数。如果有五个裁判，则常量 NJudges 被定义为：

```
#define NJudges 5
```

这样，数组 scores 的声明引入了一个有 5 个元素的数组。

当声明一个数组时，每个元素的数值并没有被初始化。和简单变量一样，不能依赖数组中任何元素的内容，除非明确地对其进行了初始化。如果要进行初始化，需要一种可能对每个元素赋初值的机制。

31.4 Selection

在 C 语言中，每个数组有两个基本特性：

- 元素类型（element type）：存储在数组元素中的数值的类型。
- 数组大小（array size）：数组所包含的元素个数。

当在程序中通过声明来建立一个新的数组时，必须指明数组的元素类型和数组大小，从而在数组内可以根据元素所处的位置对其进行单独选择。

要指定数组中的一个元素，既需要指定数组的名字，又要指定对应于元素在数组中的下标，在数组中指定特定元素的过程叫选择（selection）。

在 C 语言中，选择数组元素通过在数组的名字后面加上方括号，然后在方括号中加上下标来表示，因此结果是一个选择表达式（selection expression）。

```
array-name[index]
```

在程序中，选择表达式的作用和普通变量的作用相似，可以把选择表达式用在其他表达式中。

特别地，可以对选择表达式进行赋值。当第一个裁判（裁判 #0，因为在 C 语言中，数组元素是从 0 开始的）给选手 9.2 分时，我们就可以把分数以赋值的形式放在数组中：

```
scores[0] = 9.2;
```

在使用数组时，必须要分清数组元素的下标和数组元素值的区别。例如，数组第一个方框的下标是 0，而值是 9.2。

在内存中，数组元素通常是连续存储的，数组的第一个元素的地址称为基地址，选择特定元素所需的调整量称为该元素的偏移量，因此可以改变数组元素的内容，但绝对不可以改变数组元素的下标值。

数组可以和 for 循环结合使用，下面的示例是关于长度为 N 的数组的典型[2]操作。

```
/* clear array a */
for(i=0; i<N; i++)
  a[i] = 0;

/* reads data into array a */
for(i=0; i<N; i++)
  scanf("%d", &a[i]);

/* sums the elements of array a */
for(i=0; i<N; i++)
  sum += a[i];
```

数组元素的下标值不一定是常量，而可以是任何值为整数或标量类型的表达式。在大多数情况下，选择表达式是 for 循环的循环变量，这样就可以很容易地依次对每个数组元素进行操作。例如，可以用下列语句对 scores 数组中的元素赋初值 0。

```
#define NJudges 5
. . .
double scores[NJudges];
for(i = 0; i < NJudges; i ++){
  scores[i] = 0;
}
```

下面给出一个简单的数组操作的实例 gymjudge.c。该程序要求用户输入每个裁判员给出的分数，然后显示平均分数。

```
/*
 * File: gymjudge.c
 * ----------------
 * This program averages a set of five gymnastic scores.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * NJudges -- Number of judges
 */

#define NJudges 5

/* Main program */

main()
{
```

[2]数组选择表达式和普通变量一样，因此在调用 scanf 函数读取数组元素时，必须使用取地址符号 &。

```
   double gymnasticScores[NJudges];
   double total, average;
   int i;

   printf("Please enter a score for each judge.\n");
   for (i = 0; i < NJudges; i++) {
      printf("Score for judge #%d: ", i);
      gymnasticScores[i] = GetReal();
   }
   total = 0;
   for (i = 0; i < NJudges; i++) {
      total += gymnasticScores[i];
   }
   average = total / NJudges;
   printf("The average score is %.2f\n", average);
}
```

31.4.1 Index Range

在 C 语言中，每个数组的第一个元素的下标值是 0。然而，在许多情况下使用这种设计方式会引起用户的误解。

对于一个非专业的 gymjudge.c 程序的用户来说，他们可能对 judge#0 的存在感到不可理解。在现实生活中，我们习惯从 1 开始给元素编码，因此程序要求用户输入编号为 1 ～ 5 的裁判给出的分数会更自然一些。

解决这个问题一般有两种标准的方法：

1. 在内部使用的下标值从 0 ～ 4，但当从用户处得到数据或是在屏幕上显示数据时，下标值加 1。如果使用这种方法，在程序 gymjudge.c 中唯一需要改动的就是为每个输入值显示提示信息的 printf 语句。

   ```
   printf("Score for judge #%d: ", i + 1);
   ```

2. 声明一个有额外元素的数组，数组元素的下标值从 0 ～ 5，然后完全忽略 0 元素。

   ```
   main()
   {
     double gymnasticScores[NJudges + 1];
     double total, average;
     int i;

     printf( "Please enter a score of each judge.\n" );
     for(i = 1; i <= NJudges; i ++){
       printf( "Score for judge #%d:" , i);
       gymnasticScores[i] = GetReal();
     }

     total = 0;
     for(i = 1; i <= NJudges; i ++){
       total += gymnasticScores[i];
     }
     average = total / NJudges;
     printf( "The average score is %.2f\n" ,average);
   }
   ```

注意，在数组 scores 的声明中，使用 NJudges + 1 来说明 scores 数组的大小。声明中的数组大小是可以用表达式来表示的，但是表达式中的各个项必须是常量。

第一种方法的优点是数组内部的下标值还是从 0 开始，这样就可以方便地使用现有的基于此机

制的程序。缺点是程序需要两种不同的下标表示方法:一个是用户的外部集合,而另一个为程序员的内部集合。

在实际程序开发中,如果希望数组的下标从 1 到 10 而不是从 0 到 9,可以声明一个有 11 个元素的数组,这样数组的下标将会从 0 到 10,从而可以通过忽略掉下标为 0 的元素。

尽管用户只看见一种一致的、熟悉的下标模式,但是考虑两种下标表示方法会增加程序处理的复杂性,因此第二种方法的优点是内部的下标表示方法与你和用户进行交流的方法是一致的。

31.4.2 Character Index

在 C 语言中,使用字符作为数组下标也是可以的。

C 语言把字符作为整数来处理,在使用字符作为下标前,首先可能需要对字符进行“标量化”。

假设希望数组 letter_count 可以对字母表中每个字母进行跟踪计数,这个数组将需要 26 个元素,所以采用下列方式对其进行声明。

```
int letter_count[26];
```

不能直接使用字母作为数组 letter_count 的下标,字母的整数值不是落在 0 到 25 的区间内的。

为了把小写字母落在合适的范围内,可以简单采用减去 'a'(或 'A')的方法。

例如,如果 ch 含有小写字母,那么为了对相应的 ch 字母进行计数,所以 ch 的清零操作可以写成:

```
letter_count[ch - 'a'] = 0;
```

31.4.3 Internal Representation

为了能更好地应用数组,首先应该深刻了解 C 语言如何在内存中表示数据。只有了解了数组在计算机中是如何表示的,才能明白 C 语言如何处理数组的。

在 C 语言中,把数组作为参数进行传递和数组在内存中的表示方法有很大关系,虽然这不太容易让人理解,但是只有熟悉了数据的内部表示法之后,才能了解数组参数是如何工作的,及如何有效地使用它们。

为了弄清数据在计算机中是如何存储的,先要仔细地了解一下在一个典型的机器中内存系统是如何工作的。

从本质上讲,计算机中所有的数值都是按照被称为比特的基本单元的形式存储的。1 个比特(bit)精确记录一个数值,它可能取两种状态中的一个。如果把机器中的电路当成一个小的开关,那么可以把这两种状态标识为断开和闭合。单词 bit 实际上是 binary digit 的缩写,因此把这两种状态用 0 和 1 表示就很顺理成章了,0 和 1 这两个数字是计算机运算所基于的二进制系统所用的两个数字。

因为 1 个比特包含的信息量极为有限,所以比特本身并没有为存储数据提供一个很方便的机制。为了方便地存储整型和字符型等常见类型的信息,将若干个独立的比特组成一个较大的单元,作为完整的存储单元。这种最小的组合单位被称为字节(byte),它足以容纳一个需要 8 个比特的字符。

注意,虽然几乎在所有的现代计算机中,一个字节由 8 个比特组成,但 ANSI C 并没有要求一个字节一定要那么长,仅有的相关规则是 char 类型的对象被定义为占据一个字节。因此,一个字节的长度必须足以容纳一个字符代码。

在大多数计算机中,字节也可以组成更大一些的称为字(word)的结构,一个字的大小通常要求能够容纳一个整型数据。有些计算机用两个字节组成字,有些计算机用四个字节组成一个字,也有很少的计算机的规定十分特殊。

计算机可用的内存容量变化很大,通常用千字节(kilobyte,KB)、兆字节(megabyte,MB)和吉字节(gigabyte,GB)来衡量。

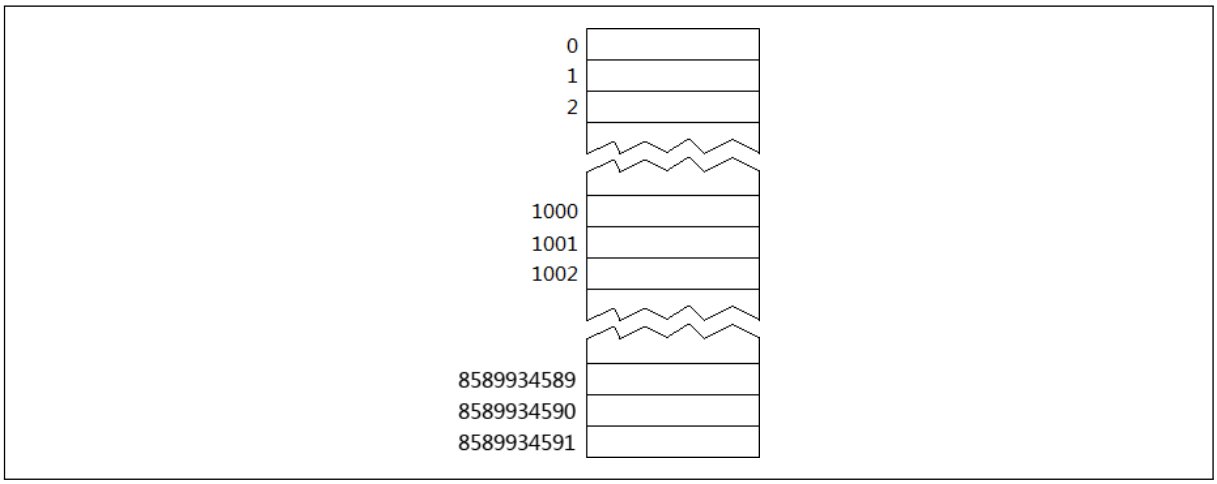
在大多数领域中,前缀 kilo 和 mega 分别代表 1 000 和 1 000 000。然而,对计算机而言,十进制系统与机器的内部结构是不匹配的,因此只是用这些前缀表示接近于原来的数值的 2 的某个整数幂。因此,在程序设计中,这些前缀真正表示的意思应该是:

$$\text{kilo(K)} = 2^{10} = 1024$$
$$\text{mega(M)} = 2^{20} = 1048576$$
$$\text{giga(G)} = 2^{30} = 1073741824$$

31.5 Memory Addressing

在存储系统中，每个字节都会有一个数字的地址(address)。

一般来说，第一个字节在计算机中的地址为 0，第二个字节的地址为 1，依此类推，直到机器中所含有的字节数为止。例如，可以用下图表示 8GB 的计算机的内存。



内存中的每一个字节可以容纳一个字符的信息。如果声明一个字符变量 ch，编译器会在当前的函数帧中为这个变量保留一个字节的空间。

假设这个字节的地址为 1000，当程序执行下述的语句时，字符‘A’的内部表示就会被存储在地址为 1000 的单元中。由于‘A’的 ASCII 码是 65，因此内存的分配结果如图(a)所示。

```
ch = 'A';
```

每个变量都都会有一个地址以指明其存储在内存中的某个地方，只是在进行程序设计时并不能预先知道存储某一变量的确切地址。

在示意图(a)中，变量 ch 存储在地址 1000 中，但这完全是强制性的。尽管如此，画出存储图并标出内存地址的起始点还是很有用的，有助于理解在程序运行时计算机内存中的变化。

当程序调用函数时，函数中的变量被分配到内存中的某个位置，但是不能提前知道这些变量的地址。在程序运行时才会给变量分配地址，在变量的声明周期中地址不再改变，而它的内容可以改变。

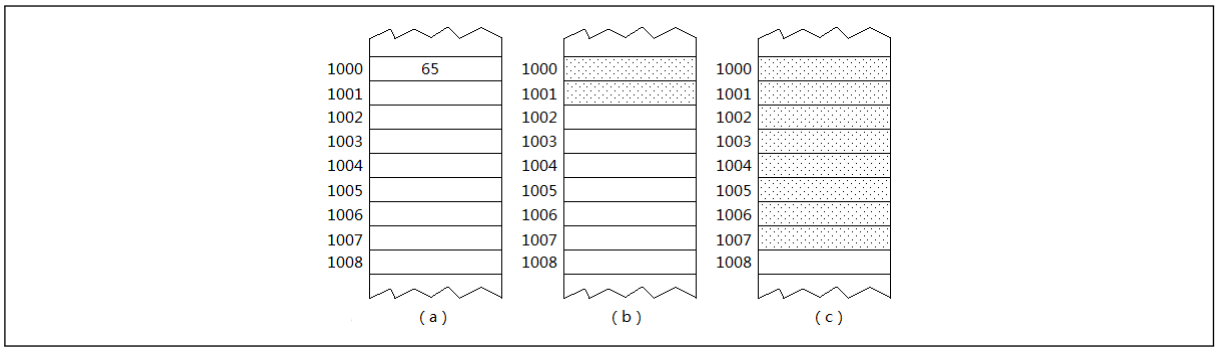
根据所存储的数据类型，变量需要不同大小的内存空间，同时根据计算机的不同，相同的变量类型需要的存储空间也有所不同。

比一个字符大的数值放在内存中连续的字节单元中。例如，如果在一台计算机中一个整型变量占两个字节，那么这个整型变量就需要两个字节的连续的内存单元，因此它存放在图(b)所示的阴影部分中。需要多个字节的数值用第一个字节的地址标识，所以图(b)阴影部分所存放的整型数值是存储在地址为 1000 的字。

作为第二个例子，double 类型的数值一般需要八个字节，所以存储在地址 1000 中的 double 类型变量将会占用从 1000 1007 的存储空间，如图(c)阴影部分所示。

实际上，只有字符型数据在所有的计算机中被定义成一个字节的长度，其余类型的数据在计算机中存储所需要的字节数，对于不同的计算机是不同的。

例如，整型变量在计算机机中占用 2 个字节的内存，但在一些大型机占用 4 个字节的内存。同样，一个 double 型变量一般需要 8 个字节的内存空间，但有些计算机也有例外。



在某个计算机上编译程序的过程中，可以用运算符 sizeof 来确定存储某一个变量所需要的字节数。一般来说，C 语言编译器的设计者会选择对计算机最有效的数据存储长度。

31.6 Memory Allocation

当在程序中声明一个变量时，编译器会给变量预留内存空间，这种预留内存空间的过程就叫分配(allocation)。

- 全局变量在程序开始执行时分配，直到程序执行完之后才会释放内存空间。
- 局部变量只有当函数调用时才会分配内存空间。

局部变量本身在分配给这个函数的存储空间中分配，函数的存储空间叫做函数的帧(frame)。

只要函数在运行，局部变量在帧中的地址始终是不变的。当函数返回时，帧以及它的所有变量都被丢弃，以便让别的函数使用。

- 如果声明了一个简单变量，编译器就会为这个变量预留其需要的内存空间，也就是 sizeof 的返回值。
- 如果声明了一个数组变量，编译器就会为数组元素预留一段连续的内存空间保存组成数组的所有元素的值。

根据数组元素的类型，数组的每个元素可能需要 1 个字节的内存空间，也可能需要几个字节的内存空间。例如，变量声明：

```
char charArray[20];
```

将会预留 20 个字节的内存空间，因为每一个字符仅占一个字节。但是，有些数据类型的每个元素要占有 1 个以上的字节。

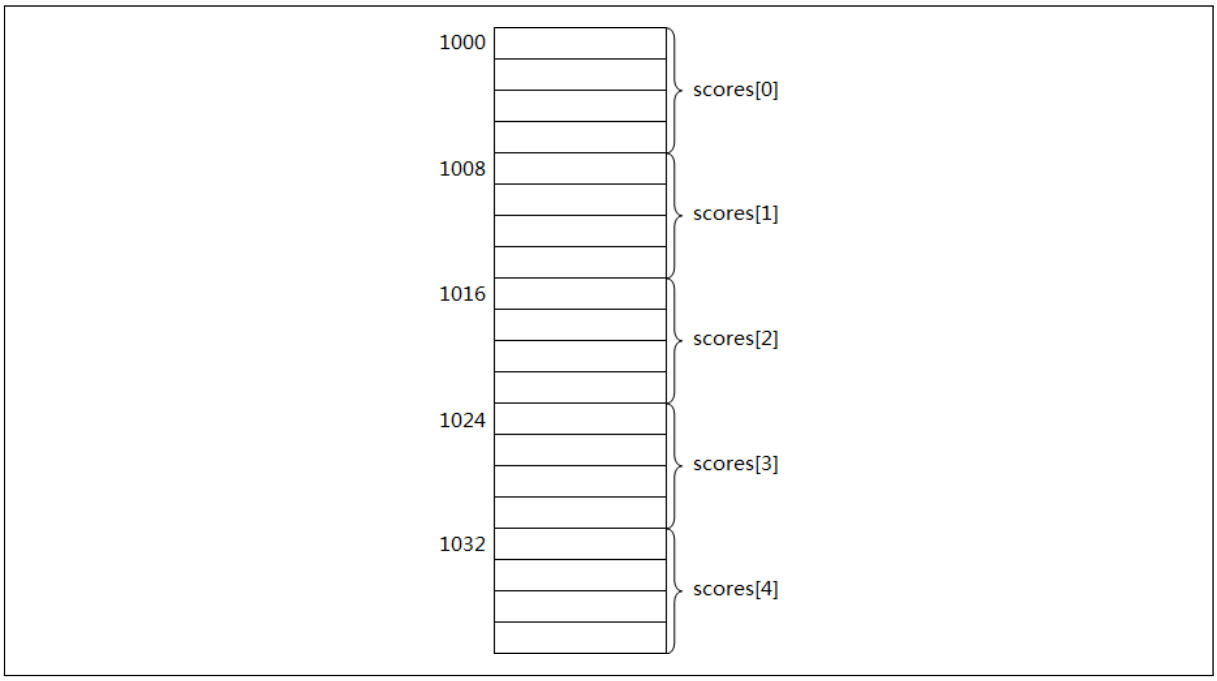
例如，假设上述 NJudges 被设为 5，数组元素都是 double 型的，每个数组元素都需要八个字节的内存空间，因此一共就需要 8 字节/元素 × 5 个元素 = 40 个字节。

```
#define NJudges 5
...
main()
{
  double scores[NJudges];
  ...
}
```

为了存放整个 scores 数组，编译器要在当前帧中分配 40 个字节。如果帧在内存中的开始地址是 1000，数组元素的存放情况将如下图所示。

在为 score 数组分配的内存空间中，0 号元素将存放在 1000 ～ 1007，1 号元素将存放在 1008 ～ 1016，依此类推。

当 C 语言编译器遇到一个选择表达式 scores[i]，它将通过把下标值和每个元素的大小相乘再加上数组的首地址来计算出数组元素的地址。在上面示意图中，元素的大小是 8，数组的首地址是 1000。因



此，如果 i 的大小为 2，那么 scores[i] 地址就为

$$2 \times 8 + 1000$$

这个地址是 1016，正好与图中 scores[2] 的地址相匹配。在这种情况下，数组元素的首地址（就是 1000）也叫做数组的基地址（base address），而找到正确元素所需的调整量（就是 2×8，即 16）叫做偏移量（offset）。

31.7 Reference Elements

C 语言不要求检查下标的范围，如果引用了一个下标值超出数组范围的元素，得到的结果将没有意义。

具体来说，当引用超出数组范围的元素时，C 语言会通过下面的方法进行处理。

如果 C 编译器能够检测到超出了数组范围，可以给出一个错误信息。然而在很多情况下，程序会照常运行，没有错误的反馈信息。在大多数系统中，程序通过把偏移量加上数组的基地址计算出选择表达式的值，而偏移量本身又是通过将元素的序号乘以元素的大小而得出的。

超出数组范围的元素的地址内容都与数组无关，因此计算机不能检测出错误，选择超出范围的数组元素也是不允许的，此时程序或者不能运行，或者得出的结果毫无意义，从而无法解决具体的问题。

下面的例子说明引用超出数组范围的元素时的错误。

```
int a[10], i;
for(i=1; i<=10; i++)
  a[i] = 0;
```

对于某些编译器来说，这个表面上正确的 for 语句会产生一个无限循环。当变量 i 的值变为 10 时，程序会将数值 0 存储在 a[10] 中。但是，实际上元素 a[10] 并不存在，所以在元素 a[9] 后数值立刻进入内存。如果内存中变量 i 放置在 a[9] 的后面，这种情况下变量 i 将会重新设置为 0，从而导致再次开始循环。

当程序有一些无法说明的错误行为时，首先应该怀疑是否数组选择超出了范围。一个调试程序的技巧就是在引用数组元素前，先输出元素的下标值。例如，如果当我们在计算 scores[i] 前加上语句：

```
printf( "i = %d\n" , i);
```

那么当程序执行到这一行时，就可以知道是否会有上述错误出现。如果输出结果是 5 或-23，或者是其他的超出合理范围的数值，就可以知道从哪里入手来查找问题。当发现并解决了问题之后，就可以把这一行输出语句去掉了。

在某些情况下，最安全的方法是加上一些条件测试语句，以检测数组元素的下标是否超出范围。例如，在计算 scores[i] 之前，可以加上下面的语句：

```
if(i < 0 || i > NJudges){
  Error( "Index i (value %d) is out of bounds."  i);
}
```

通过上述语句，程序将会在 i 超出数组范围的情况下返回错误信息反馈，以便我们发现一些无法检测到的逻辑错误。

> 常见错误：只要在程序中使用数组，就要确保用于从数组中选择元素的下标值不能超过数组边界。在大多数计算机中，引用数组范围之外的元素不会被系统作为错误检测出来，但会造成不可预料的后果。

另一方面，为下标操作加上上述的条件测试语句，会增加程序的长度和复杂性。

一般来说，当我们认为数组元素的下标值很有可能超出范围时，才应该加上一些测试语句。例如，检测下面的 for 循环语句中的 i 值是不必要的：

```
for(i = 0; i < NJudges; i ++) . . .
```

这里，for 循环语句本身就限制数值 i 不会超出范围。

然而，当使用一个作为参数传递的下标值时，就要特别小心。在这种情况下，我们不太容易控制此下标值的大小，因为它来自程序的另外的部分。

在一个大的程序中，也许是别的程序员编写的那一部分程序，所以就不知道是否那个程序员是否会和我们自己一样小心处理下标值。在这种情况下，测试我们接收到的参数值将会大大减少调试时间。

31.8 Array Parameters

编写大程序的关键点就是要把它分解成许多的小的函数，每一个函数都需要小到足以作为一个单元来理解，函数之间通过传递参数来交换信息。

数组作为参数传递给函数和传递简单的变量是不一样的。如果一个大的程序里面含有数组，那么在分解这个大的程序时就需要函数把整个数组作为参数进行传递。

为了写出一些应用数组的更复杂的实例，应该学会怎样把数组作为参数从一个函数传递给另外一个函数。

问题：编写程序实现下面的功能(程序命名为 reverse)。

1. 读入一串整型数据，直到输入 0 作为输入结束标号为止；
2. 把这些数据逆序排列；
3. 输出经过重新排列的数据。

在一般情况下，首先在读入数时将其存储在一个数组中，然后通过数组反向开始循环输出数组元素，在这个过程中不会对数组中的元素进行实际的移动。

```
/* Reverse a series of numbers */
#include <stdio.h>

#define N 10

main()
```

```
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for(i=0; i<N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order: ");
  for(i=N-1; i>=0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

为了说明数组作为参数的传递过程,重要的是把程序分解成三个函数,分别对应于程序的三个阶段——读入输入数据并将其保存到数组中,对数组元素逆序排列,然后输出结果。

用上述思路分解程序的话,主函数就会有如下的结构:

```
/* 这个程序只是逆序表程序的一个大概的轮廓, 最终的版本有不同的参数结构 */
main()
{
  int list[NElements];

  GetIntegerArray(list);
  ReverseIntegerArray(list);
  PrintIntegerArray(list);
}
```

然而,这种通用的设计方法会产生两个问题:

1. 在编写的时候,数组元素的个数是用常量 NElements 确定的,但在问题说明中,要求元素的个数在用户输入结束标志 0 之后才能确定。
2. 函数 GetIntegerArray 和 ReverseIntegerArray 只有在它们改变了参数数组中的数值时才有用。

到现在为止,还未引入改变函数的参数的机制。如果数组和简单变量不一样的话,那么整个程序的设计就必须改变了。

31.8.1 Elements Number

从问题说明中可以看出,程序 reverse.c 并不能事先确定数组元素的个数,而是让用户输入任意个值,用一个预先设定好的输入结束标志值表示数组结束。这种设计方法对用户来说十分方便,但却给程序员的编码带来了问题。

因为数组变量 list 是在主程序中定义的,所以编译器会在程序开始时立刻给它分配内存空间。但此时,用户并没有输入数据,也就不能精确知道数组应该多大。尽管如此,编译器也要求我们确定数组的大小,而且数组声明中指定的大小必须是常量,这样它的值在编译时才能确定。因此,这个问题就是找到一种方法可以使数组容纳不同数目的元素,尽管实际上数组的大小已经被编译器要求确定。

解决这个问题的一般方法是声明一个比所需要的数组更大的数组,只使用它的一部分。因此,不是根据实际的元素个数去定义数组的大小,原因在于无法预先知道实际的元素个数,只好预先设定一个常量,指出最大的元素个数,用这个常量来声明数组的大小。

例如,在 reverse.c 的主程序中,可以声明变量 list 为:

```
int list[MaxElements];
```

可以选择一个比预计在实际应用中可能遇到的个数大的数作为 MaxElements 的值,而且因为还要考虑到增长的需要,一般应该选用一个比较大的数作为 MaxElements。

例如，如果编写一个要用到大约 100 个元素的 reverse.c，那就应该选用一个比 100 更大的数作为 MaxElements，这样在该程序的任何应用中，实际的元素个数都会比这个数小，而且很可能程序最后只使用了 75 个数组元素，其他的都没有用到。

对于这样一个简单的程序而言，不必担心会占用所有内存，也不必担心会浪费内存。如果内存的利用率很重要，就必须为数组明确地分配内存。

当为数组分配 MaxElements 个元素时会遇到一个问题，那就是使用该数组的函数应该知道实际有多少数组元素被使用。

为了提供这个信息，程序应该专门设置一个变量来记录在某个时刻正在使用的元素个数。例如，可以在主函数中定义一个变量 n 来表示数组中有效元素的个数，当然它一般会小于 MaxElements。

- 在声明中指定的数组的大小被称为数组的分配长度（allocated size）——这里就是常量 MaxElements；
- 实际使用的元素的个数——这里就是变量 n，叫做数组的有效长度（effective size）。

如果函数使用一个已经存在的数组，它需要知道数组的有效长度。

例如，函数 PrintIntegerArray 需要以数组的有效长度作为参数，以便知道有多少个数组元素要输出，因此在主程序中调用 PrintIntegerArray 时，参数表中要包含数组名 list，同时也要包含 n。

```
PrintIntegerArray(list, n);
```

因为调用 PrintIntegerArray 需要传递两个参数，所以函数的原型必须声明两个相匹配的参数：保存整型数据的数组和表示数组有效长度的一个整数。

在 C 语言中，除了数组的大小是可选以外，数组参数的说明就如同数组的声明一样。

要注意的是，虽然把函数 PrintIntegerArray 原型写成如下形式也是合法的。

```
void PrintIntegerArray(int array[MaxElements], int n);
```

但是，一般是把参数说明中的上限参数 MaxElements 去掉，而写成后一种形式：

```
void PrintIntegerArray(int array[], int n);
```

这种说明风格更能表明函数 PrintIntegerArray 是可以使用不同分配长度的数组，并通过传递参数 n 来确定实际要输出的元素数目。

与其他函数一样，在函数原型中声明的形式参数名和实际参数名字可以是不一样的。形式参数和实际参数的对应关系基于它们在参数表中的位置：第一个形式参数对应第一个实际参数，依此类推。

在这个例子中，数组名在主程序中是 list，而在子函数中是 array，选用不同的名字使得可以很容易地区分在主程序中和其他函数中的数组。

ReverseIntegerArray 函数也需要访问数组数据和有效长度，所以它和 PrintIntegerArray 有同样的参数。

ReverseIntegerArray 的原型为如下形式：

```
void ReverseIntegerArray(int array[], int n);
```

函数 GetIntegerArray 却有着不同的结构。主程序是不可能传递数组的有效长度给 GetIntegerArray 函数的，在函数 GetIntegerArray 被调用的时候，主程序是不知道数组的有效长度的。当用户输入数据时，函数 GetIntegerArray 通过计数确定数组的有效长度，直到用户输入了结束标志值为止。此时，GetIntegerArray 必须告诉主程序输入了多少个数值，因此它是返回一个整型数值的函数。

尽管 GetIntegerArray 函数不需要将数组的有效长度作为参数，但它需要知道数组的分配长度。如果用户输人的数值超过数组的容量，函数 GetIntegerArray 能报告有错误产生就很重要了。此外，如果 GetIntegerArray 函数能将输入结束标志值作为参数，那就更通用了，

可以调用同样的函数去读入有不同结束标志值的数据串。因此，GetIntegerArray 函数原型可以写成：

```
int GetIntegerArray(int array[], int max, int sentinel);
```

31.8.2 Passing Mechanism

在函数中把数组作为参数时，必须了解数组参数的传递机制。当传递一个简单变量给调用函数时，根据按值传递机制，函数会收到被调用参数的一个拷贝。

函数是用拷贝进行工作而不是用原来的数值工作，因此函数不会改变调用参数的数值。然而，当一个有数组参数的函数被调用时，形式参数和实际参数之间的关系就会改变。

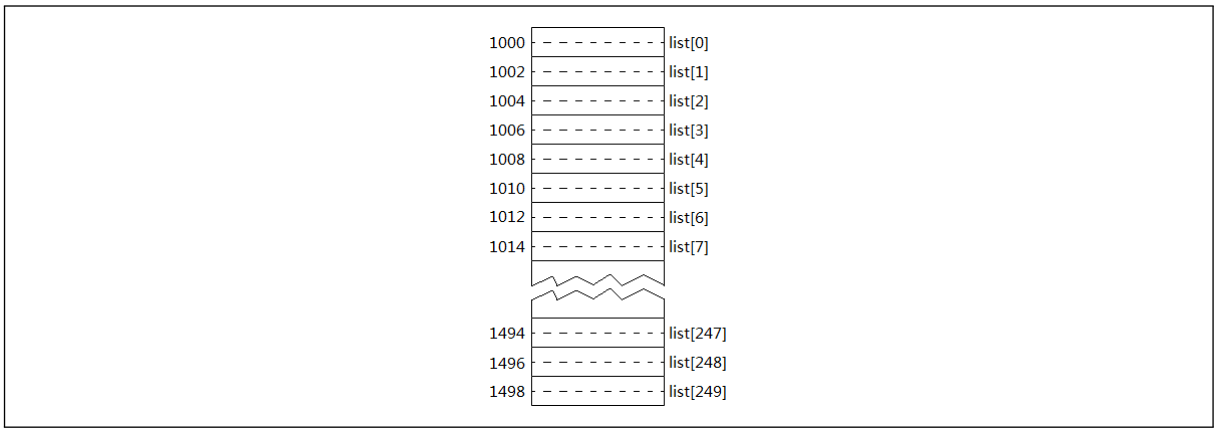
为了理解数组如何从一个函数传送到另外一个函数，需要考虑数组在内存中的表示方式。

假设 MaxElements 被定义成 250，那么下面的数组声明

```
int list[MaxElements];
```

要求编译器在 main 的帧中为有 250 个整型数据的数组分配空间。虽然不可能确切知道在计算机中使用什么地址，这里可以假设分配给数组 list 的内存起始地址是 1000，连续分配 250 个整型大小的字。

如果现在使用的计算机中，整型数据占两个字节，那么数组 list 的数据将会存储在地址 1000 ～ 1499 中，如下图所示。



当编写一个有数组参数的函数时，通常希望函数直接利用对应于数组的内存中的数值，C 语言很好的做到了这一点。

当数组作为参数传送给函数时，只有数组的基地址记录在局部帧中。如果选择在函数范围内定义的局部数组中的一个元素，那么把偏移量加上基地址就会生成在调用数组范围之内的地址。

最终结果是，在函数头部定义的形式数组参数成为函数调用中实际数组参数的同义词，而不是它的拷贝。

为理解数组操作的细节，可以思考主程序在调用 ReverseIntegerArray 函数时会有什么情况发生。

```
void ReverseIntegerArray(int array[], int n);
```

如果数组含有五个数值 1、2、3、4 和 5，那么对于函数 ReverseIntegerArray，帧的初始配置如下图所示：

如果 ReverseIntegerArray 函数做这项工作，它会把前 n 个数组元素逆序排列，而且基地址为 1000，结果如下图所示：

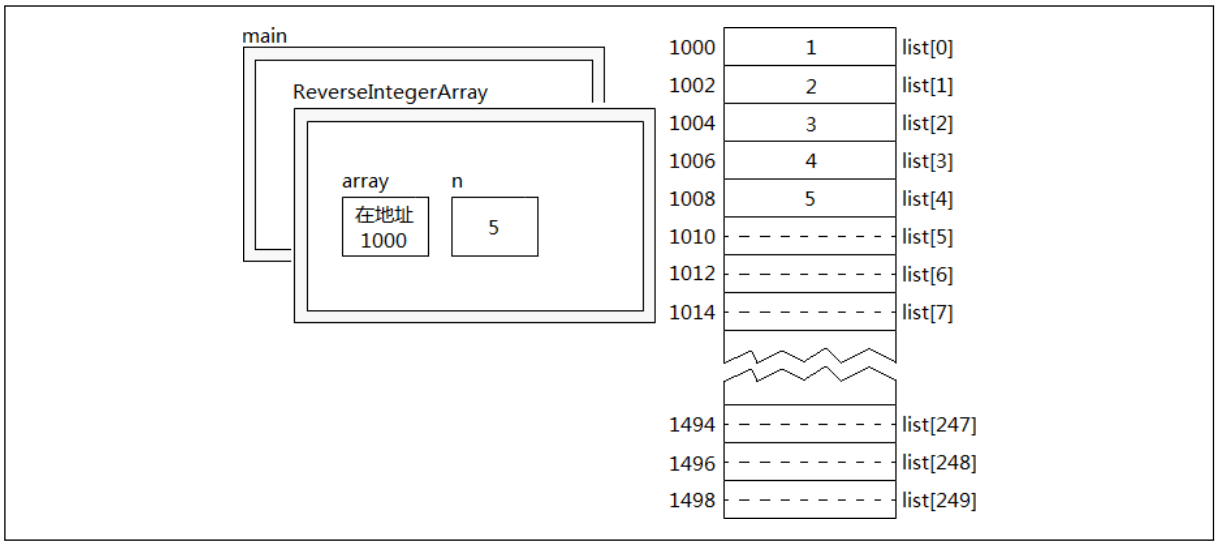
当函数 ReverseIntegerArray 返回时，它的帧会消失，但是对内存中起始地址为 1000 的前五个元素的空间的改变却是永久的，因为这些地址不是这个帧的一部分。

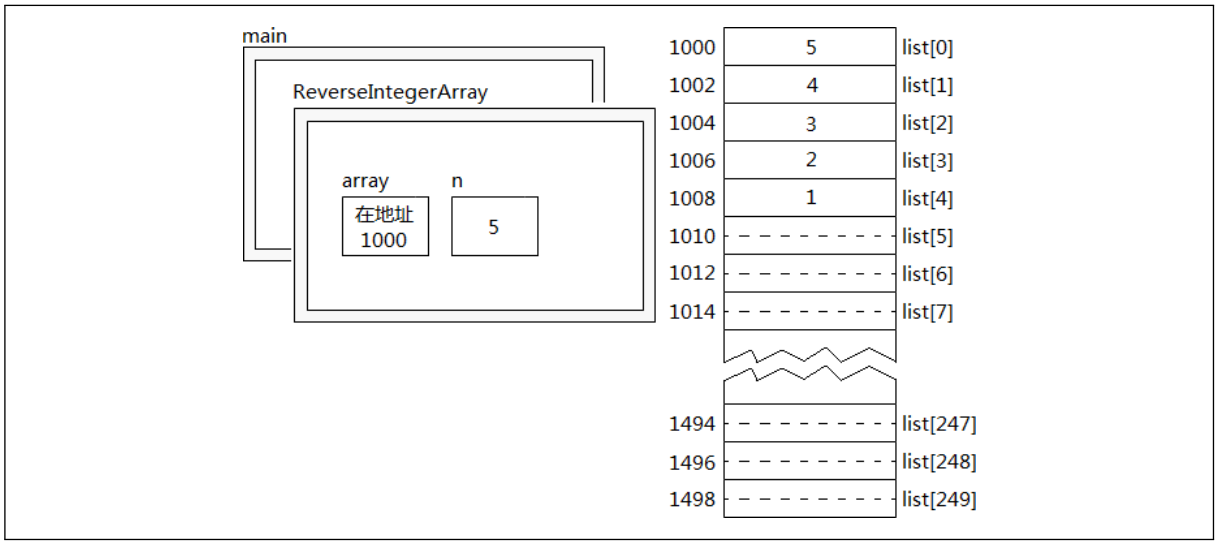
虽然 C 语言处理数组参数的方式有利于函数方便地使用数组元素，但必须要记住在 C 语言中，处理简单变量的方式是完全不同的。

如果数组参数的处理和其他的参数一样，那么调用函数

```
ReverseIntegerArray(list n);
```

就会在帧中为 ReverseIntegerArray 函数生成一个新的数组，并且复制数组 list 中 250 个元素到相应的局部变量 array 中。这种方法使得 ReverseIntegerArray 函数无法影响 list 的值，从而使函数变得无用。





即使像 PrintIntegerArray 函数那样无需改变数组参数的值，C 语言中对于数组参数的处理仍旧提高了程序的效率，因为它不需要复制整个数组，在数组很大的情况下可以节省很多的时间。

31.8.3 Constant Array

当函数接受一个数组参数但不改变它的值时，ANSI C 允许在参数声明前加上关键词 const 来把数组变为“常量”，如下所示：

```
const int months = {31,28,31,30,31,30,31,31,30,31,30,31};
void PrintIntegerArray(const int array[], int n);
```

关键词 const 让编译器知道，数组中的信息在程序执行过程中对任何客户都是不变的，该程序（函数）不能改变任何数组元素的值。

下面的示例程序说明了二维数组和常量数组的使用。首先派发一副标准纸牌[3]，然后要求玩家指明手里应该握有几张牌。

这个程序涉及到的问题包括：

- 如何从一副牌中随机抽取纸牌？
- 如何避免两次抽到同一张牌？

为了随机抽取牌，可以使用 C 语言的库函数。

[3]标准纸牌的花色有梅花、方块、红桃或黑桃，而且纸牌的等级有 2、3、4、5、6、7、8、9、10、J、Q、K 或 A。

1. <time.h> 提供的 time 函数返回当前的时间，并且这个时间被编码成单独的数。
2. <stdlib.h> 提供的 srand 函数初始化 C 语言的随机数生成器。通过把 time 函数的返回值传递给 srand 函数可以避免程序在每次运行时都派发相同的纸牌。
3. <stdlib.h> 提供的 rand 函数在每次调用时都会产生一个随机数。通过执行取余运算% 可以使得 rand 函数的返回值落在 0 ～ 3(表示牌的花色)的范围内，或者落在 0 ～ 12(表示纸牌的等级)的范围内。

为了避免两次都拿到同一张纸牌，需要跟踪已经选择好的纸牌。这里，程序将使用二维数组 in_hand 来表示纸牌总数，其中数组有 4 行(每行表示一种纸牌的花色)和 13 列(每一列表示纸牌的一种等级)。

在程序开始时，所有数组元素都将为 0(假)。每次随机抽取一张纸牌时，将检查数组 in_hand 的元素与这张纸牌是否相对应，如果对应则为真，否则为假。

- 如果判定结果为真，那么就需要抽取其他的纸牌。
- 如果判定结果为假，则将数值 1 存储到与这张纸牌相对应的数组元素中，以提醒这张纸牌已经抽取过了。

如果证实纸牌是“新”的，也就是说还没有抽取过这张纸牌，就需要把纸牌的等级和花色数值翻译成字符并显示出来。

为了把纸牌的等级和花色翻译成字符格式，程序将设置两个字符数组，一个用于纸牌的等级，另一个用于纸牌的花色。这两个字符数组在程序执行期间不会发生改变，所以可以把它们声明为 const。

```
/* Deals a random hand of cards */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13
#define TRUE 1
#define FALSE 0

typedef int Bool;

main()
{
  Bool in_hand[NUM_SUITS][NUM_RANKS] = {0};
  int num_cards, rank, suit;
  const char rank_code[] = {'2', '3', '4', '5', '6', '7', '8', '9', '10', 't', 'j', 'q', '
      k', 'a'};
  const char suit_code[] = {'c', 'd', 'h', 's'};

  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand: ");
  while(num_cards > 0){
    suit = rand() % NUM_SUITS; /* picks a random suit */
    rank = rand() % NUM_RANKS; /* picks a random rank */
    if(!in_hand[suit][rank]){
      in_hand[suit][rank] = TRUE;
      num_cards --;
      printf(" %c%c", rand_code[rank], suit_code[suit]);
    }
  }
```

```
  printf("\n");

  return 0;
}
```

const 的使用不限于数组，但是实际上 C 编译器不一定都能实现 const。在某些系统上，const 关键词被忽略；而在另一些系统上，使用 const 关键词的程序不能成功编译，因此在实践中是否使用 const，要视编译器的情况而定。

在 C 语言中，数组参数的行为不同于其他的参数，所以有必要按照下面的规则考虑它们的区别：

> 数组参数规则：如果调用一个以数组作为形式参数的函数，那么用作实际参数的数组的存储空间被形式参数所共享。改变形式参数中某一元素的值也会改变实际参数数组中的对应元素。

当把一个数组作为参数传递给函数时，对应的形式参数被初始化以保存调用数组的基地址。在函数内部，所有涉及到对形式参数进行的操作都转化为对该数组进行操作，因此数组和函数的参数实质上是等同的，而其他的参数类型只是被调参数的拷贝。

当用数组作为形式参数时，可以在声明中忽略数组的大小。在这种情况下，需要定义另一个参数来传递数组的有效长度。

31.8.4 Get/PrintIntegerArray

为了完成程序 reverse.c，需要实现被主程序调用的每一个函数。虽然可以以任何顺序写这些函数，但一般是先写最简单的。

在这个例子中，最简单的函数当然是 PrintIntegerArray，它的主要任务是遍历并分行输出数组元素。

遍历过程只需要一个 for 循环语句，开始为 0，直到遍历完所有的有效元素为止，for 循环语句的上限是数组的有效长度 n。

函数 PrintIntegerArray 的实现如下：

```
static void PrintIntegerArray(int array[], int n)
{
  int i;
  for(i = 0; i < n; i ++){
    printf(%d\n, array[i]);
  }
}
```

函数 GetIntegerArray 的实现有一些复杂，主要是要进行检测以确保用户输入的数据没有超过数组的最大容量。因为 C 语言没有提供避免输入数据超过数组的限制的措施，所以这种检测就至关重要。

如果没有检测且用户输入过多的数据，这些数据就会覆盖其他的重要的信息，使程序莫名其妙地不能运行。下面的 GetIntegerArray 函数的实现就注意检测是否输入过多数据：

```
static int GetIntegerArray(int array[], int max, int sentinel)
{
  int n, value;
  n = 0;
  while(TRUE){
    printf( "?" );
    value = GetInteger();
    if(value == sentinel) break;
    if(n == max) Error( "Too many input items for array" );
    array[n] = value;
    n ++;
```

```
  }
  return (n);
}
```

实现了 GetIntegerArray 和 PrintIntegerArray 函数之后，在编写 ReverseIntegerArray 之前就要对它们进行测试。例如，可以用下面的主程序对两个函数进行编译：

```
main()
{
  int list[MaxElements], n;
  n = GetIntegerArray(list, MaxElements, sentinel);
  PrintIntegerArray(list, n);
}
```

从用户的观点来看，这个程序并没有什么特殊的地方，程序只是读入一组整型数据，然后按照同样的顺序输出它们。但从一个程序员的角度来看，这个程序却有很大意义。

如果此程序能正常工作，就可以知道 GetIntegerArray 和 PrintIntegerArray 两个函数也在正常工作，从而在编写 ReverseIntegerArray 的时候就不必有后顾之忧。

在不同阶段测试程序是一项很重要的程序设计技巧，每个程序员应该学会使用此技巧。

31.8.5 ReverseIntegerArray

ReverseIntegerArray 函数的基本算法如下：

> 要逆序排列数组，需要交换第一个元素和最后一个元素，第二个元素和倒数第二个元素，依此类推，直到所有的元素都被交换为止。因为数组元素下标的起始值是 0，n 个元素的数组中的最后一个元素的下标为 n-1，倒数第二个元素的下标值是 n-2，依此类推。

事实上，给出任何一个整数下标值 i，倒数第 i 个元素的下标值就是

```
n - i - 1
```

为了对 array 的元素进行逆序排列，就要交换 array[i] 和 array[n-i-1] 中的数值，i 的数值是从数组的起始位置 0 直到数组的中间位置 n / 2。

当到达中间位置时，数组的后半部分的元素也达到了要求，for 循环语句每一次循环都要重新排列两个数组元素，前半部分一个，后半部分一个。

在伪代码中，函数 ReverseIntegerArray 的实现如下：

```
static void ReverseIntegerArray(int array[], int n)
{
  int i;
  for(i = 0; i < n / 2; i ++){
    Swap the values in array[i] and array[n -i - 1]
  }
}
```

交换数组的两个数值的操作是非常有用的，在其他程序中也可以使用，所以应该把它定义成一个独立的函数，并用函数调用来代替函数 ReverseIntegerArray 中的伪代码。

我们可以试着以如下形式这样进行程序设计，但是要注意交换两个元素的函数调用写成如下形式是错误的。

```
Swap(array[i], array[n - i - 1]); /* 这个设计是不能工作的 */
```

在这个函数调用中，函数 Swap 的参数是独立的数组元素。数组元素就如同简单变量一样，被拷贝到相应的形式参数中。函数 Swap 可以很容易地交换这些值的局部拷贝，但是不能对调用的实际参数进行永久赋值。

为了避免这个问题，可以把整个数组传送给执行交换操作的函数，同时将两个用来指出需要交换的位置的下标传送过去。例如，调用

```
SwapIntegerElements(array, i, n - i - 1);
```

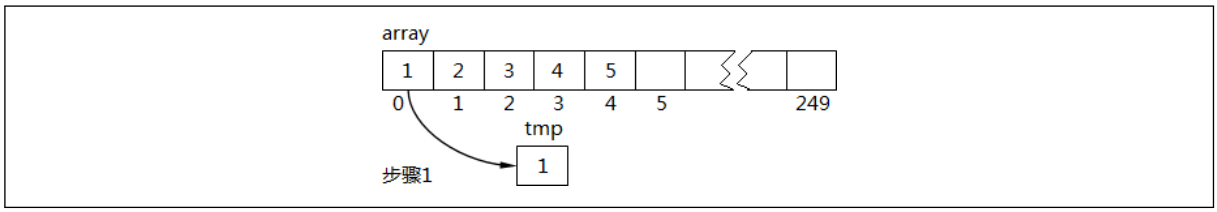
交换下标值为 i 和 n - i - 1 的数组元素，应该用上述语句替换 ReverseIntegerArray 函数中的伪代码。

31.8.6 SwapIntegerElements

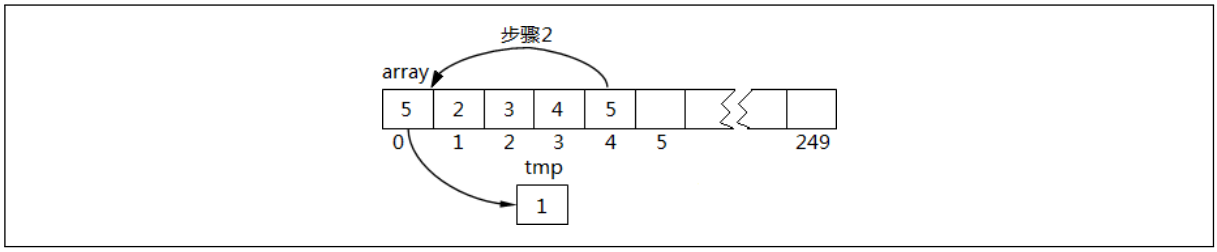
实现函数 SwapIntegerElements 要比它刚出现的时候复杂一些，不能把一个元素值赋值给另外一个元素，因为目标元素的初始值会丢失。解决此问题的最简单的办法就是用一个局部变量暂时把此元素值存放起来。如果保持一个元素值不丢失，就可以很轻松地对它进行赋值操作了，然后再把暂时存放在局部变量的数值拷贝过来。

假设想交换下标值为 0 和 4 的数组元素，可以分成三步实现这种操作：

1. 把 array[0] 的值暂时存放到一个临时变量中，如下图所示：

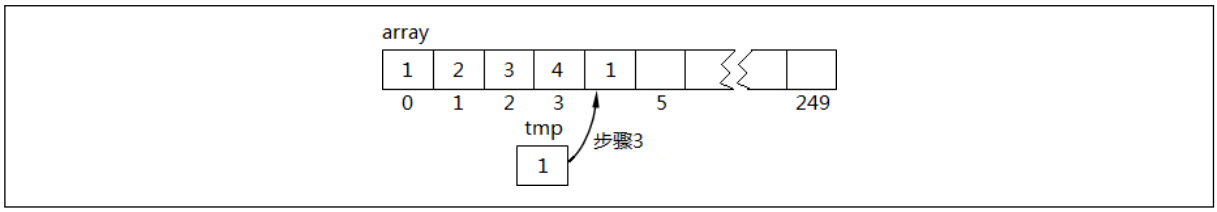


2. 把 array[4] 中的数值复制到 array[0] 中，如下图所示：



因为元素 array[0] 原来的数值已经存放在 tmp 中，所以没有信息丢失。

3. 把 tmp 中的值赋给 array[4]，如下图所示：



上述这三步是实现函数 SwapIntegerElements 的基本步骤。

```
static void SwapIntegerElements(int array[], int p1, int p2)
{
  int tmp;
  tmp = array[p1];
  array[p1] = tmp[p2];
  tmp[p2] = tmp;
}
```

这个函数补全了 reverse .c 程序，完整的程序实现如下所示。

```
/*
 * File: reverse.c
```

```
 * ---------------
 * This program reads in an array of integers, reverses the
 * elements of the array, and then display the elements in
 * their reversed order.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * MaxElements -- Maximum number of elements
 * Sentinel -- Value used to terminate input
 */

#define MaxElements 250
#define Sentinel 0

/* Private function prototypes */

static int GetIntegerArray(int array[], int max, int sentinel);
static void PrintIntegerArray(int array[], int n);
static void ReverseIntegerArray(int array[], int n);
static void SwapIntegerElements(int array[], int p1, int p2);
static void GiveInstructions(void);

/* Main program */

main()
{
   int list[MaxElements], n;

   GiveInstructions();
   n = GetIntegerArray(list, MaxElements, Sentinel);
   ReverseIntegerArray(list, n);
   PrintIntegerArray(list, n);
}

/*
 * Function: GetIntegerArray
 * Usage: n = GetIntegerArray(array, max, sentinel);
 * -------------------------------------------------
 * This function reads elements into an integer array by
 * reading values, one per line, from the keyboard. The end
 * of the input data is indicated by the parameter sentinel.
 * The caller is responsible for declaring the array and
 * passing it as a parameter, along with its allocated
 * size. The value returned is the number of elements
 * actually entered and therefore gives the effective size
 * of the array, which is typically less than the allocated
 * size given by max. If the user types in more than max
 * elements, GetIntegerArray generates an error.
 */

static int GetIntegerArray(int array[], int max, int sentinel)
{
```

```
   int n, value;

   n = 0;
   while (TRUE) {
      printf(" ? ");
      value = GetInteger();
      if (value == sentinel) break;
      if (n == max) Error("Too many input items for array");
      array[n] = value;
      n++;
   }
   return (n);
}

/*
 * Function: PrintIntegerArray
 * Usage: PrintIntegerArray(array, n);
 * -----------------------------------
 * This function displays the first n values in array,
 * one per line, on the console.
 */

static void PrintIntegerArray(int array[], int n)
{
   int i;

   for (i = 0; i < n; i++) {
      printf("%d\n", array[i]);
   }
}

/*
 * Function: ReverseIntegerArray
 * Usage: ReverseIntegerArray(array, n);
 * -------------------------------------
 * This function reverses the elements of array, which has n as
 * its effective size. The procedure operates by going through
 * the first half of the array and swapping each element with
 * its counterpart at the end of the array.
 */

static void ReverseIntegerArray(int array[], int n)
{
   int i;

   for (i = 0; i < n / 2; i++) {
      SwapIntegerElements(array, i, n - i - 1);
   }
}

/*
 * Function: SwapIntegerElements
 * Usage: SwapIntegerElements(array, p1, p2);
 * ------------------------------------------
 * This function swaps the elements in array at index
 * positions p1 and p2.
 */
```

```
static void SwapIntegerElements(int array[], int p1, int p2)
{
   int tmp;

   tmp = array[p1];
   array[p1] = array[p2];
   array[p2] = tmp;
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * --------------------------
 * This function gives instructions for the array reversal program.
 */

static void GiveInstructions(void)
{
   printf("Enter numbers, one per line, ending with the\n");
   printf("sentinel value %d. The program will then\n", Sentinel);
   printf("display those values in reverse order.\n");
}
```

31.8.7 Tabulation

程序的数据结构反映了在实际应用领域中数据的组织形式。如果要编写的程序是为了解决含有一组数据的问题，就应该在程序中使用数组来表示这组数据。例如，在 gymjudge.c 程序中，此问题包含一组分数，每个分数对应五个裁判。因为每个分数在应用的概念领域形成一个表，所以在程序中可以使用数组来表示数据。每个数组元素对应于表中的一个数据。因此，scores[0] 对应裁判 #0 的打分，scores[1] 对应于裁判 #1 的打分，依此类推。

一般来说，只要应用中包含的数据能表示成以下列表的形式，就可以选择数组来表示这组数据。

$$a_0, a_1, a_2, a_3, a_4, \cdots, a_{n-1}$$

程序员一般会把数组元素的下标称作为下角标（subscript），反映了数组是用来保存数学中带有下角标的数据这个事实。

然而，在一些数组的重要应用中，实际领域中的数据和程序中的数据的关系有不同的形式。对某些应用来说，不是在数组的连续元素中存放数值，而是用数据生成数组下标值。这些下标值被用来选择数组中的元素，记录数据的某些统计特性。

要理解这种方法的工作原理以及它和数组的一般用法的不同之处，可以通过看一个实际的例子来验证。

为了描述投资在不同利率上的收益，可以使用表格来显示在不同利率下的回报。

```
/* Prints a table of compound interest */
#include <stdio.h>

#define NUM_RATES (sizeof(value)/sizeof(value[0]))
#define INITIAL_BALANCE 100.00

main()
{
  int i, low_rate, num_years, year;
  float value[5];

  printf("Enter interest rate: ");
```

```
    scanf("%d", &low_rate);
    printf("Enter number of years: ");
    scanf("%d", &num_years);

    printf("\n Years");
    for(i=0; i<NUM_RATES; i++){
      printf("%6d%", low_rate+i);
      value[i] = INITIAL_BALANCE;
    }
    printf("\n");

    for(year=1; year <= num_years; year++){
      printf("%3d ", year);
      for(i=0; i<NUM_RATES; i++){
        value[i] += (low_rate+i)/100.0 * value[i];
        printf("%7.2f", value[i]);
      }
      printf("\n");
    }

    return 0;
}
```

这里,使用 NUM_RATES 来控制两个 for 循环。如果改变数组 value 的大小,循环将会自动调整。

```
Enter interest rate: 6
Enter number of years: 5

Years     6      7      8      9      10
  1    106.00 107.00 108.00 109.00 110.00
  2    112.36 114.49 116.64 118.81 121.00
  3    119.10 122.50 125.97 129.50 133.10
  4    126.25 131.08 136.05 141.16 146.41
  5    133.82 140.26 146.93 153.86 161.05
```

问题:编写一个程序读入用户输入的文本行,并记录 26 个英文字母出现的次数。当用户输入一个空行表示输入结束的时候,程序应该输出一个表,说明在输入数据中各个字母出现的次数。

为了生成上述的字母出现频率表,程序必须逐个字符地搜索输入文本中的每一行。一旦一个字母出现,程序就会更新一个记录此字母出现次数的数值。这个程序值得注意的地方就是要设计一种数据结构,可以记录 26 个字母出现的次数。

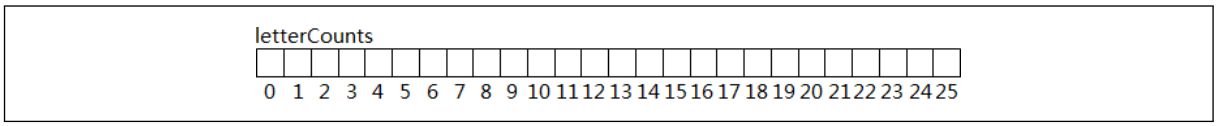
如果不使用数组,就需要定义 26 个不同的变量 nA、nB、nC 依次到 nZ 来解决这个问题,然后用一个 switch 语句来检查 26 种情况:

```
switch(toupper(ch)){
  case 'A': nA ++; break;
  case 'B': nB ++; break;
  case 'C': nC ++; break;
  . . .
  case 'Z': nZ ++; break;
}
```

实际上,上述这种方法是一个很长的且重复性的程序,更好的方法就是把 26 个变量放入一个数组,然后使用字符代码选择数组中合适的元素。每个元素里存放一个整型数值,表示对应于数组中该下标的字母当前出现的次数。

如果调用数组 letterCounts,可以把它声明为:

```
int letterCounts[NLetters];
```



其中 NLetters 被定义为常数 26，这个声明给一个有 26 个元素的整型数组分配内存空间，如下图所示：

在输入数据中每出现一个字母时，就要增加 letterCounts 中相应元素的值。

寻找需要自增的元素的过程就是用字符运算把一个字母转换成在范围 0 ～ 25 中的整数。这个转换过程由函数 LetterIndex 完成，该函数的实现如下：

```
int LetterIndex(char ch)
{
  if(isalpha(ch)){
    return (toupper(ch) – 'A' );
  }else{
    return (–1);
  }
}
```

1. 如果 ch 是一个大写或小写字母，函数 LetterIndex 将会返回一个在 0 ～ 25 之间的数值。
2. 如果 ch 不是一个字母，函数 LetterIndex 将会返回-1。

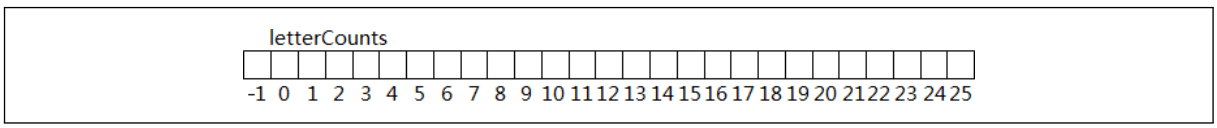
为了记录每个字母出现的次数，需要做到两点：

1. 通过调用函数 LetterIndex 把字母转化为下标值。
2. 如果下标值不是-1，就要自增数组 letterCounts 中相应下标值的元素的值。

使用 C 语言实现上述功能得到函数 RecordLetter 的实现如下所示：

```
void RecordLetter(char ch, int letterCounts[])
{
  int index;
  index = LetterIndex(ch);
  if(index != –1) letterCounts[index] ++;
}
```

对于函数 RecordLetter 来说，检测函数 LetterIndex 返回的值是否是-1 很重要。如果不作这个检测，程序就有可能增加 letterCount[-1] 中的数值。即使此元素事实上并不存在，大部分的编译器还是会检查正好位于数组开始前的那个整型字，如下图中所示：



在数组 letterCounts 之前的内存中可能存放程序需要的其他数据，所以改变这个位置的数据就有可能导致程序运行结果不正确，也有可能使程序无法运行。

在记录每个字母出现的次数之前，应该先确保数组中元素的初始值在程序运行之前为 0。程序的此项任务由函数 ClearIntegerArray 来完成，这个函数在别的程序中也很有用。

函数 ClearIntegerArray 的代码为：

```
static void ClearIntegerArray(int array[], int n)
{
  int i;
  for(i = 0; i < n; i ++){
    array[i] = 0;
  }
}
```

在程序的最后，函数 DisplayLetterCounts 生成字母的出现频率表。

```
void DisplayLetterCounts(const int letterCounts[])
{
  char ch;
  int num;
  for(ch =  'A' ; ch <=  'Z' ; ch ++){
    num = LetterCounts[LetterIndex(ch)];
    if(num != 0) printf( "%c %4d\n" , ch, num);
  }
}
```

注意,for 循环语句的循环范围是从‘A’～‘Z’而不是从 0 ～ 25 是很重要的,这里使用对应于现实问题的下标值,会有利于我们理解 for 循环语句。

同时还应该注意到,在这个例子中也无需检测函数 LetterIndex 的返回值是否是-1,因为函数 DisplayLetterCounts 的结构已经确保不会有超出范围的情况发生。

在下面程序的最终实现中还包括逐字扫描输入数据的代码。

```
/*
 * File: countlet.c
 * ----------------
 * This program counts the occurrences of individual letters
 * that appear in text read in from the user. This program
 * might be useful as a tool in solving cryptograms.
 */

#include <stdio.h>
#include <ctype.h>
#include "simpio.h"
#include "strlib.h"
#include "genlib.h"

/*
 * Constants
 * ---------
 * MaxLines -- Maximum number of input lines
 * NLetters -- Number of letters
 */

#define MaxLines 100
#define NLetters 26

/* Private function declarations */

static void CountLetters(int letterCounts[]);
static void CountLettersInString(string str, int letterCounts[]);
static void RecordLetter(char ch, int letterCounts[]);
static void DisplayLetterCounts(const int letterCounts[]);
static int LetterIndex(char ch);
void ClearIntegerArray(int array[], int n);

/* Main program */

main()
{
   int letterCounts[NLetters];

   printf("This program counts letter frequencies.\n");
   printf("Enter a blank line to signal end of input.\n");
   ClearIntegerArray(letterCounts, NLetters);
```

```
   CountLetters(letterCounts);
   DisplayLetterCounts(letterCounts);
}

/*
 * Function: CountLetters
 * Usage: CountLetters(letterCounts);
 * ----------------------------------
 * This function updates the values in the letterCounts array
 * by scanning through a series of strings read in from the
 * user. A blank line is used to signal the end of the input
 * text.
 */

static void CountLetters(int letterCounts[])
{
   string line;

   while (TRUE) {
      line = GetLine();
      if (StringLength(line) == 0) break;
      CountLettersInString(line, letterCounts);
   }
}

/*
 * Function: CountLettersInString
 * Usage: CountLettersInString(str, letterCounts);
 * -----------------------------------------------
 * This function updates the values in the letterCounts array for
 * each character in the string str.
 */

static void CountLettersInString(string str, int letterCounts[])
{
   int i;

   for (i = 0; i < StringLength(str); i++) {
      RecordLetter(IthChar(str, i), letterCounts);
   }
}

/*
 * Function: RecordLetter
 * Usage: RecordLetter(ch, letterCounts);
 * --------------------------------------
 * This function records the fact that the character ch has
 * been seen by incrementing the appropriate element in the
 * letterCounts array. Non-letters are ignored.
 */

void RecordLetter(char ch, int letterCounts[])
{
   int index;

   index = LetterIndex(ch);
   if (index != -1) letterCounts[index]++;
}
```

```
/*
 * Function: DisplayLetterCounts
 * Usage: DisplayLetterCounts(letterCounts);
 * -----------------------------------------
 * This function displays the letter frequency table, leaving
 * out any letters that did not occur.
 */

void DisplayLetterCounts(const int letterCounts[])
{
   char ch;
   int num;

   for (ch = 'A'; ch <= 'Z'; ch++) {
      num = letterCounts[LetterIndex(ch)];
      if (num != 0) printf("%c %4d\n", ch, num);
   }
}

/*
 * Function: LetterIndex
 * Usage: index = LetterIndex(ch);
 * -------------------------------
 * This function converts a character into the appropriate index
 * for use with the letterCounts array. In this implementation,
 * LetterIndex converts characters in either case to an integer
 * in the range 0 to 25. If ch is not a valid letter, LetterIndex
 * returns -1. Clients should check for a -1 return value unless
 * they are able to guarantee that the argument is a letter.
 */

int LetterIndex(char ch)
{
   if (isalpha(ch)) {
      return (toupper(ch) - 'A');
   } else {
      return (-1);
   }
}

/*
 * Function: ClearIntegerArray
 * Usage: ClearIntegerArray(array, n);
 * -----------------------------------
 * This function sets the first n elements in the array to 0.
 */

void ClearIntegerArray(int array[], int n)
{
   int i;

   for (i = 0; i < n; i++) {
      array[i] = 0;
   }
}
```

31.9 Static Initialization

数组变量和其他变量一样，都可以被声明为局部变量或全局变量，而且可以在声明时对其进行初始化。

- 为了避免使用全局变量的缺陷，除非有一些特殊要求，否则数组变量都被声明为局部变量。
- 在某一程序模块中使用的全局数组应该被声明为静态变量，以避免别的程序模块使用它们。
- 如果一个数组被声明为静态全局变量，它的每个元素在程序运行之前可以被初始化。

对于数组变量，指定初始值的等号后面是对应于每个元素的一组初始值，可以用一对花括号把初始值括起来对数组进行初始化。

最通用的静态初始化格式就是在声明中定义一个数组，并通过一个常量表达式列表来对其进行赋初值。

```
static int digits[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

如果初始化式比数组短，那么数组中剩余的元素赋初值为 0。

```
/* initial value of digits is {0, 1, 2, 3, 4, 5, 0, 0, 0, 0} */
static int digits[10] = {0, 1, 2, 3, 4, 5};
```

利用上述特性，可以将全部数组元素初始化为 0。

```
/* initial value of digits is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
static int digits[10] = {0};
```

注意，初始化式为空是非法的，初始化式长过要初始化的数组也是非法的。

另外，编译器可以通过初始化式的长度来确定数组的大小，因此在显式地初始化一个数组时，可以忽略掉数组的长度。

```
static int digits[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

在对数组进行初始化后，数组元素的数量就是固定的了。

31.9.1 Array Subscript

数组的下标可以是任意的标量类型，包括整型、字符型或者枚举型。

标量类型可以用在任何整型数据出现的地方，因而任何标量类型值都可以作为数组的下标值，这样不仅增加了标量类型的功能，也简化了它们的应用。

例如，如果想用 FALSE 和 TRUE 两个名字来输出布尔类型的数据，可以声明数组 booleanNames 如下：

```
typedef enum{FALSE, TRUE} bool;
static string booleanNames[] = {"FALSE", "TRUE"};
...
printf("flag = %s\n", booleanNames[flag]);
```

这里，常量 FALSE 的内部值为 0，TRUE 的内部值为 1，这样就可以使用这些数值作为数组 booleanNames 的下标值来生成对应于布尔型值的字符串。这种情况也同样适用于连续枚举类型。

下面的示例程序需要检查数中是否有重复的数字，这里布尔型值的数组可以用来跟踪数中出现的数字。

```
/* Checks numbers for repeated digits */
#include <stdio.h>

#define TRUE 1
#define FALSE 0

typedef int Bool;
```

```
main()
{
  Bool digit_seen[10] = {0};
  int digit;
  long int n;

  printf("Enter a number: ");
  scanf("%ld", &n);

  while(n>0){
     digit = n%10;
     if(digit_seen[digit]) break;
     digit_seen[digit] = TRUE;
     n /= 10;
  }

  if(n>0)
    printf("Repeated digit.\n");
  else
    printf("No repeated digit.\n");

  return 0;
}
```

开始时,数组 digit_seen 中每个元素的值都为 0。当给出数 n 时,程序逐个检查每个数字,并将数字存储在变量 digit 中,然后用数字作为数组 digit_seen 的下标。

- 如果 digit_seen[digit] 为真,那么表示 digit 至少在 n 中出现了两次。
- 如果 digit_seen[digit] 为假,那么表示 digit 之前未出现过,程序会把 digit_seen[digit] 设置为真并继续执行。

31.9.2 Array Size

当对一个数组进行初始化时,可以在声明中不标明数组的大小,编译器会计算出初始值的个数,然后为数组预留对应元素的空间。

```
static int digits[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

另外,不写出数组范围还有一些好处。对于一些需要在程序的生命周期内改变初始值个数的数组来说,让编译器从初始值的个数计算出数组的大小对程序的可维护性很有用处,这样可以不必在程序执行的过程中维护元素的个数。

例如,假设编写一个含有一个数组的程序,这个数组列出美国城市中人口超过 1 000 000 的城市名。从 1990 年的人口普查数据来看,可以把数组声明为:

```
static string bigCities[] = {
  "New York",
  "Los Angeles",
  "Chicago",
  "Houston",
  "Philadelphia",
  "San Diego",
  "Detroit",
  "Dallas",
};
```

后来以 2000 年的人口普查数据为依据时，应该在此数组中加入 Phoenix 和 San Antonio。要加入这两个城市，只需要把它们的名字直接放到上述初始化表的后面即可，编译器会自动增加数组的大小来容纳新的数据。

在数组 bigCities 中的最后一个初始值的后面有一个逗号，当然也可以不加逗号。

> 作为一个好的程序设计习惯，一般都会在数组中最后一个初始值后面加上逗号，这样就可以无需改变初始化列表中的已存在的城市名而直接加入新的城市名。

在编译程序时，编译器通过计算初始值的个数可以得到数组元素的个数，这里引入了一个问题就是，怎样使程序可以得到数组的元素的个数。

在 C 语言中，通过 sizeof 运算符可以确定数组的元素个数。

```
sizeof(a) / sizeof(a[0])
sizeof a / sizeof a[0]
```

用自然语言描述就是，这个表达式表示用数组中的第一个元素的大小去除整个数组的大小。因为数组中的每个元素的大小都是一样的，不管元素是什么类型，上述表达式的结果就是数组中元素的个数。因此，可以用下列表达式初始化一个变量 bigCities 保存数组中的城市数：

```
static int nBigCities = sizeof bigCities / sizeof bigCities[0];
```

从而，可以将数组 a 的清零操作改写，而且即使当数组长度发生变化时也不需要改变循环。

```
for(i=0; i < sizeof(a) / sizeof(a[0]); i++)
  a[i] = 0;
```

利用宏来表示数组的长度可以获得同样的好处，但是 sizeof 技术更好一些。

```
#define SIZE (sizeof(a) / sizeof (a[0]))
for(i=0; i < SIZE; i++)
  a[i] = 0;
```

31.9.3 Array Replication

C 语言不允许通过赋值运算符在两个数组间进行复制操作。

把一个数组复制给另一个数组，最简单的实现方法是利用循环对数组元素逐个进行复制。

```
for(i=0; i<N; i++)
  a[i] = b[i];
```

另一种可行的方法是使用来自 <string.h> 的函数 memcpy 来把内容从一个地方按字节复制到另一个地方。

为了把数组 b 复制给数组 a，使用函数 memcpy 的格式如下：

```
memcpy(a, b, sizeof(a));
```

特别是针对大型数组，memcpy 的速度比普通循环更快。

第 32 章

Matrix

32.1 Introduction

数学上的“矩阵(matrix)”与在计算机编程语言中的数组(array)的表示方法和意义上略有不同。

数学上的矩阵看起来像这样:

$$a = \begin{bmatrix} 3 & 6 & 2 \\ 0 & 1 & -4 \\ 2 & -1 & 0 \end{bmatrix}$$

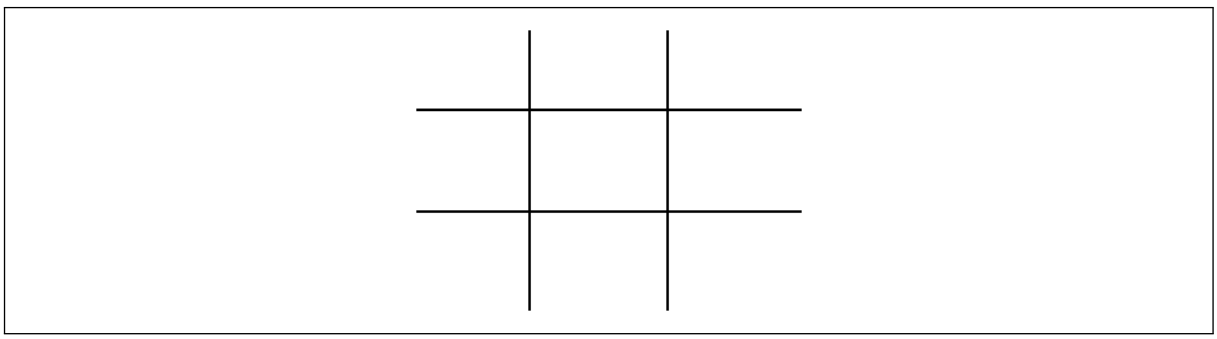
而计算机上的数组看起来像这样,例如 C 语言中的数组:

```
int a[3][3] = {
          {3, 6, 2},
          {0, 1, -4},
          {2, -1, 0}
};
```

在 C 语言中,数组元素可以是任何类型的,其中数组的元素又是数组的数组就叫做多维数组(multidimensional array)。

- 二维数组常用于表示分割成行和列的矩形的数据结构,因此这种二维结构的类型也叫做矩阵(matrix)。
- 三维或是多维的数组在 C 语言中虽然是合法的,但是很少出现。

下面以在程序中表述 tic-tac-toe 游戏作为讨论二维数组的例子,这种游戏是在一个分成三行三列的面板上进行的,如下图所示。



参加游戏的玩家轮流在这些空的方块中填入 X 和 O,尽量使相同的符号在水平方向、垂直方向或者对角线上排成一行。

为了描述这种游戏的面板,最好的选择就是使用一个三行三列的二维数组。虽然也可以使用枚举类型来表示在方块中可能出现的三种情况——空格,X 和 O,但是用 char 型数据作为元素类型并用字符 ‘ ’、‘X’ 和 ‘O’ 表示方块的三种情况可能会更简单一些。

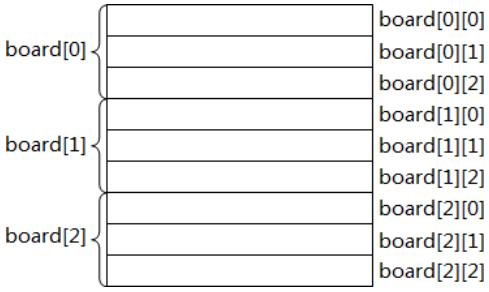
tic-tac-toe 游戏面板的声明可以写为:

```
char board[3][3];
```

接下来，就可以通过给出两个下标值来确定面板方块中的字符，一个下标确定行号，一个下标确定列号。在这种表示方法中，行号和列号的范围是 0 ～ 2，面板中各个方块有如下名字：

虽然以表格形式显示二维数组，但实际上 C 语言在在计算机内部是按照行主序存储数组的。

首先，把数组变量 board 表示为一个有三个元素的数组，每个元素又是一个有三个字符的数组，因此分配给 board 的内存空间有 9 个字节，排列次序如下图所示。



在数组 board 的二维图表中，第一个下标值是行号。这种规定是强制性的，因为矩阵的二维几何完全是概念性的。在内存中，这些数值组成一个一维列表。

如果想让第一个下标值表示列号，第二个下标值表示行号，唯一需要改变的就是基于概念几何的函数，例如显示面板当前状态的函数。

然而，在内部安排上，当数组元素存放在内存中时，第一个下标值变化的很慢，因此在 board[0] 中的元素在内存中都先于 board[1] 中的元素。

32.2 Parameter

多维数组在函数间的传递和一维数组的传递是一样的。在函数头部声明参数很像变量的初始声明，并且包含下标信息。例如，下面的函数显示数组 board 的当前状态：

```
static void DisplayBoard(char board[3][3])
{
  int row, column;
  for(row = 0; row < 3; row ++){
    if(row != 0) printf( "------+-----------+--------" );
    for(column = 0; column < 3; column ++){
      if(column != 0) printf( "|" );
        printf( "%c" , board[row][column]);
    }
    printf( "|" );
  }
}
```

在 DisplayBoard 中，许多代码都是用来格式化输出，以使面板能以更易读的形式出现。

```
 X |  O  |  X
---+-----+-----
   |  X  |  O
---+-----+-----
 X |     |  O
```

在函数接受一个多维数组作为参数时，C 语言要求指定每一个下标的大小，第一个下标值例外（它是可选的）。

```
static void DisplayBoard(char board[][3]);
```

程序可以以同样的方式工作，因为对 C 语言来讲，知道数组的基地址和第二个下标的大小就可以确定数组 board 中每一个元素的地址，而不管行数是多少。然而，略去第一个下标值使声明很不对称，因此，一般情况下，在声明多维数组参数时，每一个下标的范围都应该写上。

32.3 Initialization

和初始化一维数组一样，通过嵌套一维初始化式的方法可以产生二维数组的初始化式。

```
int m[3][9] = {
  {1, 2, 3, 4, 5, 6, 7, 8, 9},
  {1, 3, 5, 7, 9, 11, 13, 15, 17},
  {2, 4, 6, 8, 10, 12, 14, 16, 18}
};
```

为了强调多维数组的整体结构，用来初始化内部数组的数值都被放到一组花括号中，这样每一个内部初始化式都是一个数组的初始化式，可以继续采用类似的方法来构造高维数组。

```
static double identityMatrix[3][3] = {
  {1.0, 0.0, 0.0},
  {0.0, 1.0, 0.0},
  {0.0, 0.0, 1.0},
};
```

上述声明指出了一个 3×3 的浮点数矩阵，这个特殊的矩阵就是数学中经常出现的单位矩阵（identity matrix）。

> 在数学中，单位矩阵在主对角线上的值为 1，而其他地方的值为 0，其中主对角线上行、列的索引值是完全相同的。

C 语言为多维数组提供了多种方法来缩写初始化式。

- 如果初始化式没有大到足以填满整个多维数组，可以把数组中剩余的元素赋值为 0。

```
static int m[5][9] = {
  {1,1,1,1,1,1,1,1,1},
  {1,1,1,1,1,1,0,1,1}
};
```

- 如果内层的初始化式没有大到足以填满数组的一行，可以把该行剩余的元素赋值为 0。

```
static int m[5][9] = {
  {1,1,1,1,1,1,1,1,1},
  {1,1,1,1,1}
};
```

C 语言允许忽略掉内层的大括号来让编译器在填满一行后自动开始填充下一行。

```
static int m[5][9] = {
  1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,0
};
```

实际上，在多维数组中忽略掉内层大括号可能是危险的操作，额外的元素（或丢失的元素）会影响到剩下的初始化式，因此某些编译器会产生类似“Initialization is only partially bracketed.”的警告。

32.4 Operation

在 C 语言中，把多维数组看作数组的数组，第一个下标值表示在最外面的数组中选择一个元素，而第二个下标值表示在相应的数组中再选择元素，以此类推。

选择表达式 m[i][j] 可以用于在 i 行 j 列中存取数组 m 的元素，其中 m[i] 指明了数组 m 的第 i 行，而 m[i][j] 则选择了第 i 行中的第 j 个元素。如果把 m[i][j] 替换成 m[i,j]，C 语言会把逗号看成是逗号运算符，所以 m[i,j] 等同于 m[j]。

和参数的情况一样，静态初始化的多维数组的声明必须指定所有下标值的范围，第一个下标值除外，因为它可以通过计算初始值个数确定。但如果作为传送参数，最好在声明多维数组时明确指定每个下标值的范围。

在对多维数组进行操作时，嵌套的 for 循环是处理多维数组的理想选择。例如，在单位矩阵的初始化问题中，使用嵌套的 for 循环每执行一步可以遍历每行的索引，或者遍历每列的索引，从而可以方便地访问数组中的每一个元素。

```
#define N 10

float ident[N][N];
int row, col;

for(row=0; row<N; row++)
  for(col=0; col<N; col++)
    if(row==col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

三维或是多维数组的数组在 C 语言中虽然是合法的，但是很少出现。和其他编程语言中的多维数组相比，C 语言中的多维数组扮演的角色相对较弱，这主要是因为 C 语言为存储多维数据提供了更加灵活的方法——指针数组。

第 33 章

Stack

33.1 Introduction

栈(stack)只允许在一端进行操作的数据结构,并按照后进先出(LIFO, Last In First Out)的原理运作。

栈和数组都可以存储具有相同数据类型的多个数据项,只是栈中数据项的操作是受限制的,而且大多数 CPU 都有用作堆栈指针的寄存器。

栈作为一种特殊的串行形式的数据结构,其特殊之处在于只能允许在链结串行或阵列的顶端(称为堆栈顶端指标)进行压入(push)数据和弹出(pop)数据的运算。

只能往栈中压入数据项(把数据项加上 1 结束作为“栈顶 (top)”),或者从栈中弹出数据项(从同样的末尾移走数据项),因此又把栈称为 LIFO(后进先出)数据结构,并禁止测试或修改不在栈顶的数据项。

另外堆栈也可以用一维阵列或连结串行的形式来完成,另外一个相对的堆栈操作方式称为队列(Queue)。

33.2 Definition

以下是堆栈的抽象定义。

```
init: -> Stack
push: N x Stack -> Stack
top: Stack -> (N U ERROR)
pop: Stack -> Stack
isempty: Stack -> Boolean
```

此处的 N 代表某个元素(如自然数),而 U 表示集合求交。

对应的语义如下:

```
top(init()) = ERROR
top(push(i,s)) = i
pop(init()) = init()
pop(push(i, s)) = s
isempty(init()) = true
isempty(push(i, s)) = false
```

这里的例程是以数组实现来实现栈。

```
#include <stdio.h>
#include <stdlib.h>

/*堆栈数据结构*/
struct Stack
{
  int Array[10];//数组空间
  int Top;//堆栈顶部指针
};
```

```
/*检查堆栈是否为空*/
bool stack_empty(Stack *Stack1)
{
  if(Stack1->Top==0)
  {
    return true;
  }
  else
  {
    return false;
  }
}
/*压入数据*/
void push(Stack *Stack1,int x)
{
  Stack1->Top=Stack1->Top+1;
  Stack1->Array[Stack1->Top]=x;
}
/*弹出数据*/
int pop(Stack *Stack1)
{
  if(stack_empty(Stack1))
  {
    printf("underflow");
  }
  else
  {
    Stack1->Top=Stack1->Top-1;
    return Stack1->Array[Stack1->Top+1];
  }

}
int main()
{
  struct Stack *Stack1=(struct Stack *)malloc(sizeof(struct Stack));//声明数据结构空间
  Stack1->Top=0;//初始化
  push(Stack1,3);//压入3
  push(Stack1,4);//压入4
  push(Stack1,1);//压入1
  push(Stack1,10);//压入10
  printf("%d ",pop(Stack1));//弹出10
  printf("%d ",pop(Stack1));//弹出1
  printf("%d ",pop(Stack1));//弹出4
  system("pause");
}
```

33.3 Software Stack

栈可以用链表和数组两种方式实现，一般为一个堆栈预先分配一个大小固定且较合适的空间，因此在 Stack 结构下包含一个数组。如果空间实在紧张，也可用链表实现，且去掉表头。

通过把数据项存储在数组(contents)中来实现栈时，使用整型变量(top)来标记栈顶的位置。

- 往栈中压入数据时，可以把数据项简单存储在 contents 中标记为 top 的位置上，然后自增 top。当栈为空时，top 值为 0。
- 从栈中弹出数据时，自减 top 值，并把 top 作为 contents 的索引取回弹出的数据项。

基于上述描述，可以在下面的示例中实现栈，其中所有函数都需要访问变量 top，其中 2 个函数还都需要访问 contents 数组，因此需要把 contents 和 top 设置为外部变量。

```
#define STACK_SIZE 100
#define TRUE 1
#define FALSE 0

typedef int Bool;

int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

Bool is_empty(void)
{
  return top == 0;
}

Bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if(is_full() )
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if(is_empty() )
    stack_underflow();
  else
    return contents[--top];
}
```

第 34 章

Queue

34.1 Introduction

34.2 Implementation

第 35 章

Records

35.1 Introduction

In computer science, a storage record[?] is:

- A group of related data, words, or fields treated as a meaningful unit; for instance, a Name, Address, and Telephone Number can be a "Personal Record".
- A self-contained collection of information about a single object; a record is made up of a number of distinct items, called fields.
- In IBM mainframes, a record is a basic unit of device-to-program data transfers. Mainframe files, properly called data sets, are traditionally structured collections of records, as opposed to modern byte stream access files. Records may have a fixed length or variable length.

In Unix-like systems, a number of programs (for example, awk, join, and sort) are designed to process data consisting of records (called lines) each separated by newlines, where each record may contain a number of fields separated by spaces, commas, or some other character.

In the context of a relational database, a row[?] —also called a record or tuple —represents a single, implicitly structured data item in a table. In simple terms, a database table can be thought of as consisting of rows and columns or fields.[1] Each row in a table represents a set of related data, and every row in the table has the same structure.

For example, in a table that represents companies, each row would represent a single company. Columns might represent things like company name, company street address, whether the company is publicly held, its VAT number, etc.. In a table that represents the association of employees with departments, each row would associate one employee with one department.

In a less formal usage, e.g. for a database which is not formally relational, a record is equivalent to a row as described above, but is not usually referred to as a row.

The implicit structure of a row, and the meaning of the data values in a row, requires that the row be understood as providing a succession of data values, one in each column of the table. The row is then interpreted as a relvar composed of a set of tuples, with each tuple consisting of the two items: the name of the relevant column and the value this row provides for that column.

Each column expects a data value of a particular type. For example, one column might require a unique identifier, another might require text representing a person's name, another might require an integer representing hourly pay in cents.

-	Column 1	Column 2
Row (Record) 1	Row 1, Column (Field)1	Row 1, Column 2
Row 2	Row 2, Column 1	Row 2, Column 2
Row 3	Row 3, Column 1	Row 3, Column 2

35.1.1 Overview

In computer science, a record[?] (also called struct or compound data) is a basic data structure (a tuple may or may not be considered a record, and vice versa, depending on conventions and the language at hand). A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.

For example, a date could be stored as a record containing a numeric year field, a month field represented as a string, and a numeric day-of-month field. As another example, a Personnel record might contain a name, a salary, and a rank. As yet another example, a Circle record might contain a center and a radius. In this instance, the center itself might be represented as a Point record containing x and y coordinates.

Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

A record type is a data type that describes such values and variables. Most modern computer languages allow the programmer to define new record types. The definition includes specifying the data type of each field and an identifier (name or label) by which it can be accessed. In type theory, product types (with no field names) are generally preferred due to their simplicity, but proper record types are studied in languages such as System F-sub. Since type-theoretical records may contain first-class function-typed fields in addition to data, they can express many features of object-oriented programming.

Records can exist in any storage medium, including main memory and mass storage devices such as magnetic tapes or hard disks. Records are a fundamental component of most data structures, especially linked data structures. Many computer files are organized as arrays of logical records, often grouped into larger physical records or blocks for efficiency.

The parameters of a function or procedure can often be viewed as the fields of a record variable; and the arguments passed to that function can be viewed as a record value that gets assigned to that variable at the time of the call. Also, in the call stack that is often used to implement procedure calls, each entry is an activation record or call frame, containing the procedure parameters and local variables, the return address, and other internal fields.

An object in object-oriented language is essentially a record that contains procedures specialized to handle that record; and object types are an elaboration of record types. Indeed, in most object-oriented languages, records are just special cases of objects, and are known as plain old data structures (PODSs), to contrast with objects that use OO features.

A record can be viewed as the computer analog of a mathematical tuple. In the same vein, a record type can be viewed as the computer language analog of the Cartesian product of two or more mathematical sets, or the implementation of an abstract product type in a specific language.

35.1.2 History

The concept of record can be traced to various types of tables and ledgers used in accounting since remote times. The modern notion of records in computer science, with fields of well-defined type and size, was already implicit in 19th century mechanical calculators, such as Babbage's Analytical Engine.

Records were well established in the first half of the 20th century, when most data processing was done using punched cards. Typically each record of a data file would be recorded in one punched card, with specific columns assigned to specific fields.

Most machine language implementations and early assembly languages did not have special syntax for records, but the concept was available (and extensively used) through the use of index registers, indirect addressing, and self-modifying code. Some early computers, such as the IBM 1620, had hardware support for delimiting records and fields, and special instructions for copying such records.

The concept of records and fields was central in some early file sorting and tabulating utilities, such as IBM's Report Program Generator (RPG).

COBOL was the first widespread programming language to support record types,[citation needed] and its record definition facilities were quite sophisticated at the time. The language allows for the definition of nested records with alphanumeric, integer, and fractional fields of arbitrary size and precision, as well as fields that automatically format any value assigned to them (e.g., insertion of currency signs, decimal points, and digit group separators). Each file is associated with a record variable where data is read into or written from. COBOL also provides a MOVE CORRESPONDING statement that assigns corresponding fields of two records according to their names.

The early languages developed for numeric computing, such as FORTRAN (up to FORTRAN IV) and Algol 60, did not have support for record types; but latter versions of those languages, such as Fortran 77 and Algol 68 did add them. The original Lisp programming language too was lacking records (except for the built-in cons cell), but its S-expressions provided an adequate surrogate. The Pascal programming language was one of the first languages to fully integrate record types with other basic types into a logically consistent type system. IBM's PL/1 programming language provided for COBOL-style records. The C programming language initially provided the record concept as a kind of template (struct) that could be laid on top of a memory area, rather than a true record data type. The latter were provided eventually (by the typedef declaration), but the two concepts are still distinct in the language. Most languages designed after Pascal (such as Ada, Modula, and Java) also supported records.

35.1.3 Operations

A programming language that supports record types usually provides some or all of the following operations:

- Declaration of a new record type, including the position, type, and (possibly) name of each field;
- Declaration of variables and values as having a given record type;
- Construction of a record value from given field values and (sometimes) with given field names;
- Selection of a field of a record with an explicit name;
- Assignment of a record value to a record variable;
- Comparison of two records for equality;
- Computation of a standard hash value for the record.

The selection of a field from a record value yields a value.

Some languages may provide facilities that enumerate all fields of a record, or at least the fields that are references. This facility is needed to implement certain services such as debuggers, garbage collectors, and serialization. It requires some degree of type polymorphism.

In systems with record subtyping, operations on values of record type may also include:

- Adding a new field to a record, setting the value of the new field.
- Removing a field from a record.

In such settings, a specific record type implies that a specific set of fields are present, but values of that type may contain additional fields. A record with fields x, y, and z would thus belong to the type of records with fields x and y, as would a record with fields x, y, and r. The rationale is that passing an (x,y,z) record to a function that expects an (x,y) record as argument should work, since that function will find all the fields it requires within the record. Many ways of practically implementing records in programming languages would have trouble with allowing such variability, but the matter is a central characteristic of record types in more theoretical contexts.

Most languages allow assignment between records that have exactly the same record type (including same field types and names, in the same order). Depending on the language, however, two record data types defined separately may be regarded as distinct types even if they have exactly the same fields.

Some languages may also allow assignment between records whose fields have different names, matching each field value with the corresponding field variable by their positions within the record; so that, for example, a complex number with fields called real and imag can be assigned to a 2D point record variable with fields X and Y. In this alternative, the two operands are still required to have the same sequence of field types. Some languages may also require that corresponding types have the same size and encoding as well, so that the whole record can be assigned as an uninterpreted bit string. Other languages may be more flexible in this regard, and require only that each value field can be legally assigned to the corresponding variable field; so that, for example, a short integer field can be assigned to a long integer field, or vice-versa.

Other languages (such as COBOL) may match fields and values by their names, rather than positions.

These same possibilities apply to the comparison of two record values for equality. Some languages may also allow order comparisons ('<'and '>'), using the lexicographic order based on the comparison of individual fields. [citation needed]

PL/I allows both of the preceding types of assignment, and also allows structure expressions, such as a = a+1; where "a" is a record, or structure in PL/I terminology.

In Algol 68, if Pts was an array of records, each with integer fields X and Y, one could write Pts.Y to obtain an array of integers, consisting of the Y fields of all the elements of Pts. As a result, the statements Pts[3].Y := 7 and Pts.Y[3] := 7 would have the same effect.

In the Pascal programming language, the command with R do S would execute the command sequence S as if all the fields of record R had been declared as variables. So, instead of writing Pt.X := 5; Pt.Y := Pt.X + 3 one could write with Pt do begin X := 5; Y := X + 3 end.

35.1.4 Representation

The representation of records in memory varies depending on the programming languages. Usually the fields are stored in consecutive positions in memory, in the same order as they are declared in the record type. This may result in two or more fields stored into the same word of memory; indeed, this feature is often used in systems programming to access specific bits of a word. On the other hand, most compilers will add padding fields, mostly invisible to the programmer, in order to comply with alignment constraints imposed by the machine—say, that a floating point field must occupy a single word.

Some languages may implement a record as an array of addresses pointing to the fields (and, possibly, to their names and/or types). Objects in object-oriented languages are often implemented in rather complicated ways, especially in languages that allow multiple class inheritance.

The following show examples of record definitions:

```
struct date{
  int year;
  int month;
  int day;
};
```

35.2 Concepts

数组的引入说明了计算机程序设计的一个重要理念——利用复合的数据结构可以表示复杂的信息集合。

现代程序设计语言普遍能够将各自独立的数据组成一个彼此相关的整体。过程和函数可以将不同的操作统一在一个名称下。

另外，复合的数据结构（例如数组）可以在数据处理方面实现一致的统一。在程序中声明一个数

组,就可以将任意多的数据聚集起来构成一个概念上的整体,需要时可以从数组中选出元素并单独进行操作,也可以将它们作为一个整体进行操作。

当为现实世界中的一组数据建模时,数组是非常有用的工具,而这样的数据往往具有两条基本性质:首先,数据必须是有序的,因而可以通过索引来引用特定的元素。其次,数据必须是同质的,即所有的元素的基本类型必须相同。当试图对现实世界中的一组或一系列相似的事物进行建模的时候,数组往往是最理想的工具。另一方面,一组无序的、异质的数据在 C 语言中也可以看作一个整体,一般将这样的一组数据称为结构(structure),而在计算机科学中则被称为记录(record)。

在很多情况下,通过复合数据类型可以将小的程序段整合成为单一的、更高级的结构,有助于简化程序并提高程序的可读性。

例如,打印工资单时需要能够取得雇员的姓名、职位、社保号、薪金、个人所得税等信息,这些信息就组成了一个雇员的记录。

记录可以理解为表格中的行,每个记录都由若干个部分组成,这些组成部分提供了关于记录某一方面的信息。每个组成部分通常被称为字段(field),或者被称为成员(member)。

在 C 语言中,记录的每个部分被称为字段,而且每条记录都由单独的字段组成,但是字段表达了一个整体的含义,这种概念上的完整性代表了一个复合的数据结构。例如,对于一个雇员记录,可以讨论 Name 字段或 Salary 字段。每个字段都有相应的类型,不同的字段可以有不同的类型,这里 Name 字段和 Title 字段都是字符串,而 Salary 字段则是浮点类型。

真实世界中的信息往往由多个部分组成,而且各个部分又具有整体性,对于这样的信息应该用复合的数据结构处理。

- 如果各个部分是有序的、同质的,则应该使用数组。
- 如果各个部分是无序的,即使类型相同也需要使用记录。

在 C 语言中创建一个记录,需要执行如下两个步骤:

1. 定义一个新的结构类型。
   在声明结构变量之前,必须先定义一个新的结构类型。
   类型定义指明了结构类型是由哪些字段组成,字段的名称,以及字段中信息的类型,这样结构类型定义为所有具有新类型的对象定义了一个模式,但是并不会分配任何存储空间。
2. 声明新结构类型的变量。
   定义了新的结构类型之后,需要声明该类型的结构变量,这样才可以将数据值存入其中。

定义结构类型和声明结构类型的变量是两个完全不同的操作,结构类型定义只为声明其他变量提供了一个模板,本身并没有存储空间。

35.3 Definition

在结构类型的定义中,新类型的名称(new-type-name)在最后一行出现,之前为组成该结构的字段说明。

```
typedef struct{
  field-declarations
}new-type-name;
```

其中,field-declarations 为标准的变量声明,用来定义结构中的字段。

结构类型中的字段由一系列的字段声明定义,这些字段声明和函数中的变量声明类型。例如,下面定义了一个新的结构类型 employeeT 来表示雇员信息。

```
typedef struct{
  string name;
  string title;
  string ssnum;
  double salary;
  int withholding;
```

```
}employeeT;
```

上述的结构类型定义为具有新类型 employeeT 的所有对象提供了一个模板，每一个对象都具有 5 个字段，由 name 字段开始，直至 withholding 字段结束。

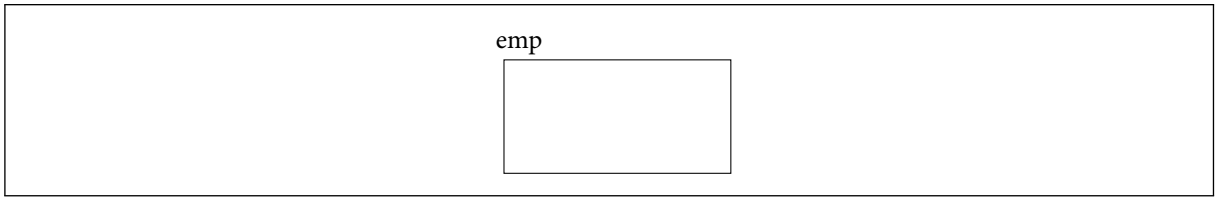
35.4 Declaration

声明新的结构类型变量的语法和声明普通变量的语法相同。

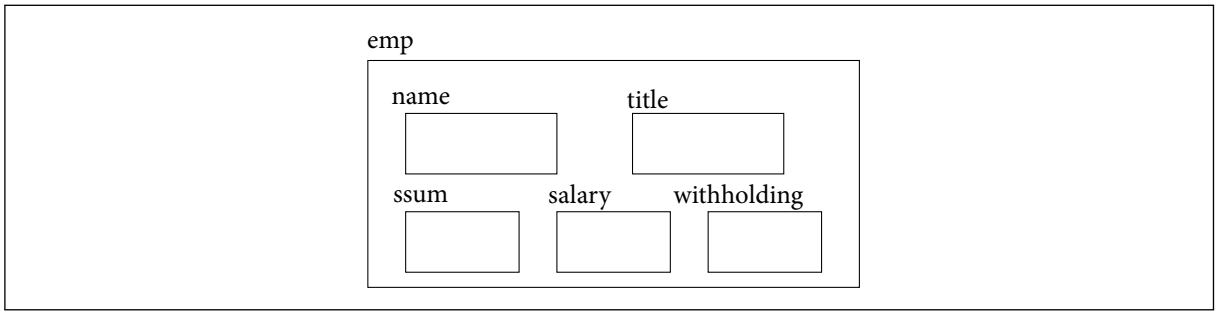
```
employeeT emp;
```

如果需要用方框图表示新结构变量，可以采取两种方式。

- 如果从整体的角度来看待结构变量（即在概念上从一个较高的层次来观察），结构变量可以看作是普通的变量。



- 如果从细节方面查看结构类型变量，结构变量内部还包含由普通变量表示的字段。



35.5 Selection

在声明了结构类型变量后，就可以使用变量名来指代该记录，进一步也可以打开记录结构来对单个字段进行操作。

如果需要引用记录中的某一个字段，需要在结构变量名后面附加符号“.”和字段。例如，如果要输出存储在结构变量 emp 中雇员的职位，需要使用下面的代码：

```
employeeT emp;
printf("%s", emp.title);
```

通过点运算符“.”选择字段被称为记录选择（record selection）。

35.6 Initialization

点运算符返回一个左值，表示可以对一个记录选择表达式进行赋值，因此可以通过赋值的方式对结构变量进行初始化。例如，可以执行下面的语句来初始化某个记录。

```
employeeT emp;
emp.name = "Haiseburg";
```

```
emp.title = "Project Manager";
emp.ssnum = "111-22-3333";
emp.salary = 23000.00;
emp.withholding = 1;
```

如果声明的变量为一个静态全局变量,可以对其内容进行静态初始化,而且初始化时需要按照结构定义中的顺序来进行。例如,下面的代码示例对静态全局记录变量 manager 进行声明和初始化。

```
static employeeT manager = {
  "LiLei", "Manager", "231-23-3243", 2500.00, 1
};
```

35.7 Applications

35.7.1 Coordinate

实际应用中的结构类型大都庞大而复杂,例如在使用图形库时,需要频繁的使用坐标。

在最简单的情况下,可以以 x 和 y 值作为参数来处理坐标。从概念上来讲,函数对于坐标的处理并不如对于点的处理来的容易,一个点由 x 和 y 的坐标确定,但是两个坐标又代表了同一个实体,这样就可以将单独的坐标结合为记录,每一个点就可以看作单独的实体。

为了用单独的实体描述一个点,第一步应该定义类型 pointT 为一组坐标值。

```
typedef struct{
  double x, y;
}pointT;
```

接下来可以编写处理 pointT 类型的函数和过程,例如下面的函数将两个坐标结合起来形成一个 pointT 的值。

```
pointT CreatePoint(double x, double y)
{
  pointT p;

  p.x = x;
  p.y = y;
  return (p);
}
```

在定义了 pointT 类型和处理 pointT 类型变量的函数后,可以用来声明变量,并对其进行初始化。

```
pointT origin;
origin = CreatePoint(0,0);
```

数组和记录之间的一个重大区别就是记录变量是一个左值,而数组变量不是左值。

只要两个记录变量为相同类型,就可以将一个记录赋值给另一个记录。例如,下面的赋值表示 rec1 所有字段的内容都是从 rec2 中相应的字段复制过来的。

```
point rec1, rec2;
rec2 = CreatePoint(1,1);
rec1 = rec2;
```

记录也可以作为函数的参数进行传递。例如,如果定义了函数 AddPoint 来进行坐标的运算,那么当把结构类型变量作为参数传递给 AddPoint 函数时,函数返回的坐标就是两个点的坐标之和。

```
pointT AddPoint(pointT p1, pointT p2)
{
  pointT p;
```

```
  p.x = p1.x + p2.x;
  p.y = p1.x + p2.y;
  return (p);
}
```

这里 p1 和 p2 都是 pointT 类型的值，当调用函数 AddPoint(p1, p2) 时，函数返回点的坐标就是 p1 和 p2 相应坐标的和。

当调用参数为记录的函数时，记录参数的值就复制到相应的实参。和 C 语言处理其他参数一致，当函数改变实参的值，或改变某个字段内容时，形参还保留着原来的值。

35.7.2 Data Record

现代程序设计语言允许利用一个类型来构造更复杂的新类型，引入结构类型之后，可以定义记录数组，或者包含数组的记录，而且这样的新类型同样可以用来创建更加复杂的类型。

例如，定义了结构类型 employeeT 之后，接下来可以用它来声明数组，而数组的元素的类型就是 employeeT。

下面的声明语句声明了具有 10 个整型值的数组 scores。

```
int scores[10];
```

如果声明的 employeeT 类型的数组，那么数组中的每个元素都是 employeeT 类型的。

```
employeeT staff[10];
```

和操作普通数组相同，可以选择 staff 数组的任意元素。例如，可以使用 staff[1] 来选择第 2 个元素代表的整个记录。

对于 staff[1] 代表的记录，可以进一步选取其中的字段。这里，可以使用 staff[1].name 来选取记录中的 name 字段。

```
staff[1].name
```

对于记录中的字段，可以进一步进行选取。例如，name 字段是一个字符串，那么可以继续对 name 中元素进行选取。

在处理复杂数据结构的程序中，常常可以看到一长串的选取操作，从记录中选取字段，再从数组中选取元素。例如，可以使用下面的表达式来取得 staff 数组中第一个记录的 name 字段的首字母。

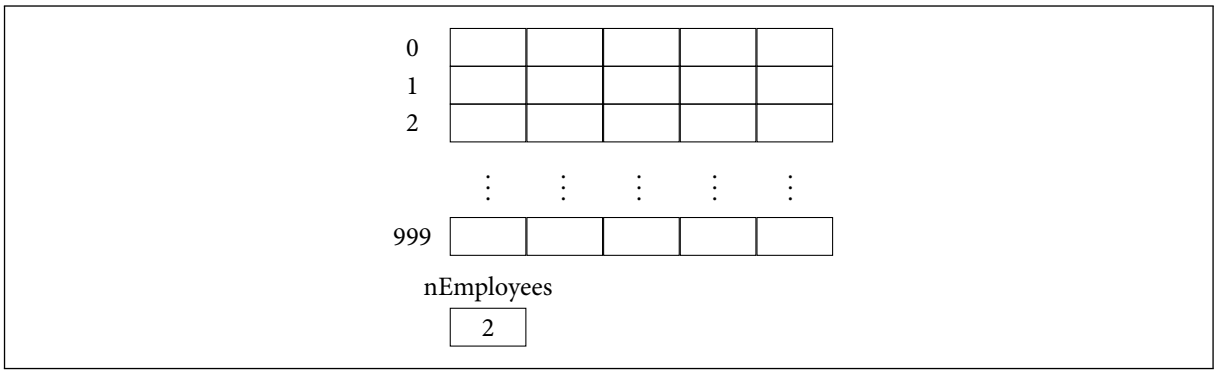
```
staff[0].name[0]
```

声明一个记录数组的方式与声明普通数组相同——即按照最大长度分配存储空间，再追踪由一个整型变量表示的实际长度。例如，在下面的示例中最多可以处理 1000 个记录，由存储在变量 nEmployees 中的值代表当前员工数量。

```
#define MaxEmployees 1000

employeeT staff[MaxEmployees];
int nEmployees;
```

C 语言允许通过过程或函数对记录数组进行操作。例如，下面的函数可以列出记录数组 staff 中所有的 name 字段和 title 字段。

```
static void ListEmployees(employeeT staff[], int nEmployees)
{
  int i;

  for(i = 0; i < nEmployees; i++){
    printf("%s (%s)\n", staff[1].name, staff[i].title);
  }
}
```



进一步,可以通过通过下面的函数计算 salary 值平均值。

```
static double AverageSalary(employeeT staff[], int nEmployees)
{
  double total;
  int i;

  total = 0;
  for(i = 0; i < nEmployees; i++){
    total += staff[i].salary;
  }
  return (total / nEmployees);
}
```

Part V

Algorithm

第 36 章

Introduction

在数学和计算机科学之中，算法（Algorithm）为一个计算的具体步骤，常用于计算、数据处理和自动推理。精确而言，算法是一个表示为有限长列表的有效方法，而且算法应包含清晰定义的指令用于计算函数。

大多数问题都是可以使用不同的算法来解决，选择一个最适合应用的算法是程序员任务中最重要的部分。

当在不同的算法之间选择时，需要考虑许多不同的因素，包括效率、可维护性和清晰性，因此选择一种特定的算法需要在这些指示之间进行权衡。

函数对于程序设计来说是很重要的，其中的一个原因是函数提供了算法实现的基础。

- 算法本身是抽象的策略，通常用自然语言表达。
- 函数是以某种程序设计语言表示的算法的具体实现。

当要将算法作为程序的一部分实现时，通常要写一个函数来执行该算法，而该函数也可以调用其他函数来处理它的一部分工作。随着问题越来越复杂，需要周密地思考才能找到解决的策略，并且往往还要考虑多个策略而不是一个策略。

算法中的指令描述的是一个计算，当其运行时能从一个初始状态和初始输入（可能为空）开始，经过一系列有限而清晰定义的状态最终产生输出并停止于一个终态。

一个状态到另一个状态的转移不一定是确定的。随机化算法在内的一些算法包含了一些随机输入。

36.1 History

形式化算法的概念部分源自尝试解决希尔伯特提出的判定问题，并在其后尝试定义有效计算性或者有效方法中成形。这些尝试包括库尔特·哥德尔、Jacques Herbrand 和斯蒂芬·科尔·克莱尼分别于 1930 年、1934 年和 1935 年提出的递归函数，阿隆佐·邱奇于 1936 年提出的 λ 演算，1936 年 Emil Leon Post 的 Formulation 1 和艾伦·图灵 1937 年提出的图灵机。

即使在当前，依然常有直觉想法难以定义为形式化算法的情况。

算法在中国古代文献中称为“术”，最早出现在《周髀算经》、《九章算术》。特别是《九章算术》，给出了四则运算、最大公约数、最小公倍数、开平方根、开立方根、求素数的埃拉托斯特尼筛法，线性方程组求解的算法等。三国代的刘徽给出求圆周率的算法——刘徽割圆术。

- 唐代：《一位算法》一卷，《算法》一卷；
- 宋代：《算法绪论》一卷、《算法秘诀》一卷。最著名的是杨辉的《杨辉算法》；
- 元代：《丁巨算法》；
- 明代：程大位《算法统宗》
- 清代：《开平算法》、《算法一得》、《算法全书》。

英文名称“Algorithm”来自于 9 世纪波斯数学家花拉子米（比阿勒·霍瓦里松，波斯语：□□□□□□□，拉丁转写：al-Khwarizmi），因为比阿勒·霍瓦里松在数学上提出了算法[1]这个概念。

[1]算法一词来源于 9 世纪阿拉伯数学家 Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî，他写了一篇名为 *Kitab al jabr w'al-muqabala* 的论文（其中第一次用到单词 algebra）。

“算法”原为“algorism”，即“al-Khwarizmi”的音转，意思是“花拉子米”的运算法则，在 18 世纪演变为“algorithm”。

欧几里得算法被人们认为是史上第一个算法。

第一次编写程序是 Ada Byron 于 1842 年为巴贝奇分析机编写求解解伯努利微分方程的程序，因此 Ada Byron 被大多数人认为是世界上第一位程序员。因为查尔斯·巴贝奇（Charles Babbage）未能完成他的巴贝奇分析机，这个算法未能在巴贝奇分析机上执行。

因为“well-defined procedure”缺少数学上精确的定义，19 世纪和 20 世纪早期的数学家、逻辑学家在定义算法上出现了困难。20 世纪的英国数学家图灵提出了著名的图灵论题，并提出一种假想的计算机的抽象模型，这个模型被称为图灵机（Turing Machine）。

图灵机的出现解决了算法定义的难题，图灵的思想对算法的发展起到了重要的作用。

36.2 Characteristics

算法的核心是创建问题抽象的模型和明确求解目标，之后可以根据具体的问题选择不同的模式和方法完成算法的设计。

以下是 Donald Knuth 在他的著作 The Art of Computer Programming 里对算法的特征归纳：

1. 输入：一个算法必须有零个或以上输入量。
2. 输出：一个算法应有一个或以上输出量，输出量是算法计算的结果。
3. 明确性：算法的描述必须无歧义，以保证算法的实际执行结果是精确地符合要求或期望，通常要求实际运行结果是确定的。
4. 有限性：依据图灵的定义，一个算法是能够被任何图灵完备[2]系统模拟的一串运算，而图灵机只有有限个状态、有限个输入符号和有限个转移函数（指令）。而一些定义更规定算法必须在有限个步骤内完成任务。
5. 有效性：又称可行性。算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现。

36.3 Implementations

算法的常用设计模式包括：

- 完全遍历法和不完全遍历法：在问题的解是有限离散解空间，且可以验证正确性和最优性时，最简单的算法就是把解空间的所有元素完全遍历一遍，逐个检测元素是否是我们要的解。这是最直接的算法，实现往往最简单。但是当解空间特别庞大时，这种算法很可能导致工程上无法承受的计算量。这时候可以利用不完全遍历方法——例如各种搜索法和规划法——来减少计算量。
- 分治法：把一个问题分区成互相独立的多个部分分别求解的思路。这种求解思路带来的好处之一是便于进行并行计算。
- 动态规划法：当问题的整体最优解就是由局部最优解组成的时候，经常采用的一种方法。
- 贪婪算法：常见的近似求解思路。当问题的整体最优解不是（或无法证明是）由局部最优解组成，且对解的最优性没有要求的时候，可以采用的一种方法。
- 线性规划法：在数学中，线性规划 (Linear Programming，简称 LP) 问题是目标函数和约束条件都是线性的最优化问题。
- 简并法：把一个问题通过逻辑或数学推理，简化成与之等价或者近似的、相对简单的模型，进而求解的方法。

算法不单单可以用计算机程序来实现，也可以在人工神经网络、电路或者机械设备上实现。

算法的常用实现方法包括：

- 递归方法与迭代方法

[2]在可计算性理论里，如果一系列操作数据的规则（如指令集、编程语言、细胞自动机）可以用来模拟单带图灵机，那么它是图灵完备的。这个词源于引入图灵机概念的数学家艾伦·图灵。

- 顺序计算、并行计算和分布式计算: 顺序计算就是把形式化算法用编程语言进行单线程串行化后执行。其余见词条。
- 确定性算法和非确定性算法:
- 精确求解和近似求解:

36.4 Formal Algorithm

算法是计算机处理信息的本质, 因为计算机程序本质上是一个算法来告诉计算机确切的步骤来执行一个指定的任务, 如计算职工的薪水或打印学生的成绩单。一般地, 当算法在处理信息时, 会从输入设备或数据的存储地址读取数据, 把结果写入输出设备或某个存储地址供以后再调用。

第 37 章

Bubble Sort

求最大值算法是算法的一个简单的例子。

现在有一串随机数列，要求找到这个数列中最大的数。如果将数列中的每一个数字看成是一颗豆子的大小，可以将下面的算法形象地称为“捡豆子”：

- 首先将第一颗豆子放入口袋中。
- 从第二颗豆子开始检查，如果正在检查的豆子比口袋中的还大，则将它捡起放入口袋中，同时丢掉原先口袋中的豆子。反之则继续下一颗豆子。直到最后一颗豆子。
- 最后口袋中的豆子就是所有的豆子中最大的一颗。

下面是一个用 ANSI C 代码表示的形式算法。

```
int max(int *array, int size)
{
  int mval = *array;
  int i;
  for (i = 1; i < size; i++)
    if (array[i] > mval)
      mval = array[i];
  return mval;
}
```

更进一步，为了找出数组中的最大元素和最小元素，可以定义一个 max_min 函数并向其传递两个指向变量的指针 max 和 min，这样函数返回两个值并分别存储在指定的变量中。

max_min 函数具有下列原型：

```
void max_min(int a[], int n, int *max, int *min);
```

这样，max_min 函数的调用可以具有下列的形式：

```
max_min(b, N, &big, &small);
```

其中，b 是整型数组，N 是数组 b 中的元素数量，big 和 small 都是普通变量，分别代表最大元素和最小元素。

当 max_min 函数找到数组 b 中的最大元素时，通过给 *max 赋值来把值存储到 big 中，同理可以通过给 *min 赋值来把最小元素的值存储到 small 中。

```
/* Finds the largest and smallest elements in an array */
#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

main()
{
  int b[N], i, big, small;
  printf("Enter %d numbers: ", N);
  for(i=0; i<N; i++)
    scanf("%d", &b[i]);
```

```
  max_min(b, N, &big, &small);

  printf("Largest number: %d\n", big);
  printf("Smallest number: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for(i=1; i<n; i++){
    if(a[i] > *max)
      *max = a[i];
    else if(a[i] < *min)
      *min = a[i];
  }
}
```

第 38 章

Prime

西方数学从古埃及开始发端，在古希腊时代进入了一个黄金时代，其中便形成了研究非负整数的特性的数论（number theory），其中的一个重要问题就是确定一个数是否为素数，如果一个正整数 n 只能被 1 和它本身整除，则这个正整数 n 就是素数（prime），而且按照该定义可知，整数 1 不是素数，因为它只被 1 整除。

素数在研究编码的密码学（cryptography）中起着重要的作用。在现代电子通信领域中，计算机经常用于执行编码和解码操作，许多可用的、最好的编码技术都是基于素数的。

设计一个用于确定整数 n 是否为素数的函数时，若直接从素数的定义出发，最直接的方法是统计约数的个数，检查是否有两个约数，而且由于 n 的约数必须小于等于 n，因此只要检查所有 1 到 n 之间的数，就会找出所有的约数，因此可以通过下列步骤来确定 n 是否为素数。

1. 检查 1 到 n 之间的每个数，看它是否能整除 n；
2. 每次遇到一个新约数，计数器加 1；
3. 在所有的数都被测试后，检查约数计数器的值是否为 2。

上述策略可以作为测试一个数是否为素数的函数 IsPrime 的实现基础。

```
bool IsPrime(int n)
{
  int divisors, i;

  divisors = 0;
  for(i = 1; i <= n; i ++){
    if(n % i == 0) divisors += 1;
  }
  return (divisors == 2);
}
```

该函数的原型指出，IsPrime 取一个整数 n，返回一个布尔值，是一个谓词函数，该实现用变量 divisors 保存到目前为止发现的约数的个数。

在程序开始处，divisors 被初始化为 0，当发现一个新的约数时，divisors 加 1。如果检查了 1 到 n 之间的所有数之后，约数个数正好等于 2，则 n 是素数，这个测试可以表示为布尔表达式

```
divisors == 2
```

IsPrime 函数将这个值作为它的结果值返回。

38.1 Verifying strategy

一个策略首先要能工作，然后才需要验证其是否有效。IsPrime 函数表示了确定一个数是否为素数的算法。为了验证 IsPrime 确实是一个算法，要从有关算法的以下要求开始

1. 定义是明确的，无二义性的；
2. 有效性，即它的步骤都是可执行的；
3. 有限性，即在执行有限步后算法会结束。

首先验证 IsPrime 是否满足定义明确，并无二义性的要求。对于这第一个要求，对用自然语言表示的算法，这个条件通常难以满足，所有的人类语言可能都有些模糊。当用自然语言表示一个算法时，

可能会省略某些步骤或掩盖某些重要的细节。然而当用一种程序设计语言表示一个算法时，该语言的定义会指定精确的解释。

虽然程序员在阅读程序时可能会误解程序的某些方面，但一个正确运行的编译器只会以一种确定的方法解释语句。在程序设计语言中，某一语句的意义称为语言的语义（semantic）。由于程序设计语言的语义比人类语言的语义更加严格，因此，当算法被表示为程序形式时，很容易满足第一个条件。

第二，IsPrime 函数中的步骤是否有效或者是否能执行？以程序形式表示的算法有助于满足这个要求，而且 C 语言设计语言的语义为程序中的每一种结构指定了它的意义，编译器会产生机器执行所必需的指令。

第三，IsPrime 会在有限步后结束吗？通过检查程序的结构可以看到，函数中仅有的长期运行的部分是一个 for 循环。从控制行可以看出，每次调用 IsPrime 函数，循环将执行 n 个周期，如果 n 非常大，函数会执行很长时间，但最终是会返回的。

通过上面的讨论，IsPrime 满足这三个条件，因此它确定是一个算法。

38.2 Demonstrating algorithm correctness

除了要确定一个策略的实现是一个算法外，还要确定这个算法的正确性。

在 IsPrime 算法中，对每一个可能的 n 值，它都能给出正确的答案吗？这里对 IsPrime 的实现可以概述证明其正确性的方法。

对于程序中的变量 divisors，在 for 循环的每个周期中，divisors 记录了到这点为止遇到的约数个数。在 for 循环开始前，还没有发现任何约数，因此 divisors 被初始化为 0。在循环的每一个周期中，当发现一个新的约数时，程序将给 divisors 加 1，因此，在第 i 个周期结束时，divisors 保存了 1 到 i 之间的约数个数。因为对每一个循环值 i，这个特性都为真，所以在第 n 个循环结束时也为真，于是变量 divisors 包含了 1 到 n 之间的约数个数，如果约数个数为 2，则 n 一定是素数。

在循环开始时为真，在每个循环周期结束前也为真的特性称为循环不变式（loop invariant）。对简单程序来说，通过说明维护了相应的循环不变式，可以很容易地说明程序的正确性。对于复杂的程序来说，证明其正确性是相当困难的。在这些情况下，很少能依赖一些形式化的方法来确定程序是否递交了正确的答案。

通常，有两种方法可以增加程序员对程序正确性的把握。

第一，按着程序代码一步一步执行，确定程序的行为和提出的要求一致，这个过程被称为桌面检查（desk-checking）。要学会用怀疑的眼光执行桌面检查并找到推理程序的技术，这需要实践和训练。

第二种方法称为测试（testing），即利用尽可能多的测试实例来运行程序，对每种情况都检查执行结果是否正确。例如，可以编写一个主程序，对 1 1000 之间的数调用 IsPrime，然后检查输出，确信所有的数据都被正确分类，这样对算法的正确性就会更有把握。

然而，完全严格的测试几乎是不可能的，因为要测试的情况是不计其数的。Edsger Dijkstra 曾说过：“测试能发现错误的存在，但永远不能发现它们不存在”。当编写一个较大的复杂的程序时，必须把这个原理牢记在心。

常见错误：当在编写一个程序时，尽量彻底地测试程序是很重要的。尽管这样，程序中还会有许多测试没有发现的错误，因为通常不可能将每一种可能出现的情况都测试一下。

38.3 Improving algorithmic efficiency

解决某一问题时，可以设计多种程序来完成，确定一个数是否为素数也可以用许多种方法，而使用的每一种方法的效率都是不一样的。

对于 IsPrime 函数实现的算法，当要检查的数字很大时，它就不太可行了。例如，如果对数字 1000000 调用 IsPrime，该函数需要检查 1 ～ 1000000 之间的 1 百万个数，确定它们是否为约数。

这里，测试所有的数是愚蠢的，因为 1000000 明显不是素数，除了 1 和它本身外，它还能被 2 整除，因此可以确定的是，一定存在更好的方法，不用检查所有的数就能得到答案。

有多种修改这个基本的算法的方法用于改善 IsPrime 函数实现的效率。例如，下面的三种修改方案都能明显地改善 IsPrime 执行的效率。

1. 从 1000000 的例子中可以看出，IsPrime 没有必要检查所有的约数，只要发现任何大于 1 小于 n 的约数，则说明 n 不是素数，于是就可以停下来报告 n 不是素数，因此需要改变程序结构，一旦发现了约数，函数立即返回。
2. 一旦函数已经检查了 n 是否能被 2 整除，就不需要再检查它是否能被其他偶数整除。如果 n 能被 2 整除，程序就停下来，报告 n 不是素数。如果 n 不能被 2 整除，那么它也不可能被 4 或 6 等其他偶数整除，因此，一旦确定 n 不可能被 2 整除，IsPrime 只需要检查奇数。
3. 该程序不需要检查直到 n 为止的所有约数，例如，它可以在一半的地方就停止。因为任何大于 n/2 的值不可能被 n 整除，再进一步思考会发现，该程序不需要试探任何大于 n 的平方根的约数。

为了理解其中的原因，假设 n 能被某一个整数 d1 整除，由整除的定义可知，n/d1 也是一个整数，称为 d2。由于 d1 × d2 = n，如果其中一个因子大于 n 的平方根，那么另一个一定小于 n 的平方根，因此如果 n 有任何约数的话，肯定有一个小于它的平方根，这个结果意味着程序中的 for 循环只要运行 $i \leq \sqrt{n}$ 次。

如果把上述三个策略组合起来，就可以写一个更有效的 IsPrime 函数的实现，技巧在于写程序时必须仔细，以保证函数能正确工作，写过后要对程序进行桌面检查和测试。

```
bool IsPrime(int n)
{
  int i;

  if(n % 2 == 0) return (FALSE);
  for(i = 3; i <= sqrt(n); i +=2){
    if(n % i == 0) return (FALSE);
  }
  return (TRUE);
}
```

在该实现中出现了一些很严重的问题，最明显的错误是：当参数值为 1 和 2 时，函数返回一个错误的答案。按照素数的数学定义，1 不是素数，该函数却错误地报告 1 是素数，因为 for 循环没有发现任何约数。

此外，在避免检查 2 以外的偶数约数时，函数使用了语句：

```
if((c % 2) == 0) return(FALSE);
```

该语句未考虑有一个偶数素数（即 2 本身）这个事实。这里检查了 n 是否为偶数，但没有检查它是否为 2，所以在这种情况下，IsPrime 也给出了错误的答案。

为了纠正这些错误，最简单的方法是单独检查 1 和 2，可以在程序的开始插入下列语句：

```
if(n <= 1) return(FALSE);
if(n == 2) return(TRUE);
```

这种在算法处理前检查一些特殊情况的语句在程序设计中是相当普遍的。虽然这些测试本身是相当简单的，但是困难的是要注意到这些特殊情况的存在。

常见问题：在设计一个通用算法时，考虑是否存在使通用算法失败的特例是很重要的。如果存在这种特例，必须确保在程序中明确地处理这些特例。

在纠正原先算法的实现中的错误的同时，可能会引入新的问题，甚至会起到相反的作用，降低效率。例如在下面的 for 循环的控制行中：

```
for(i = 3; i <= sqrt(n); i +=2)
```

尽管在计算机中能在很短的时间里计算平方根，但它还是比完成简单的算术运算（如乘和除等）花费的时间更长。在上面的语句中，IsPrime 在每个 for 循环周期中都要调用 sqrt 函数，尽管每一次循环返回的结果都是相同的，因为 n 的值在循环内部并未改变，sqrt(n) 的值也不会变。

为了避免重复地调用 sqrt 而且得到的结果是同一个，可以在循环的开始前先计算出 sqrt(n)，并把它存入一个变量。例如，可以引入一个 double 类型的变量 limit，用下面的两个语句替换前面的 for 循环控制行。

```
limit = sqrt(n);
for(i = 3; i <= limit; i += 2)
```

这个简单的改变明显地改善了 IsPrime 实现的效率。

常见问题：当在某一程序中使用循环时，检查一下是否在循环中存在某些放在循环开始前执行更好的运算。如果存在，则可以计算一次结果，把它存入一个变量中，然后在循环中使用这个变量，这样可以改善程序的效率。这个实现还有一个更难看出的问题：

发现这个逻辑错误是很难的，因为它可能在测试中不出现，可能用成千上万个输入值测试这个函数，每次得到的都是正确的答案，但仍然会有人可以用以前没有测试过的例子使函数失败，而且 IsPrime 的这个实现在不同的机器上可能会出现不同的结果。

为了理解这个问题，首先提出的是用浮点数判断严格的相等是很危险的。假设数 n 是某一个素数的平方，例如，假设 n 是 49，它是 7 的平方，而当对 49 调用 sqrt 函数时，它的返回的值在计算机和数学领域中，往往不可能完全相等。在严格的数学领域中，这个平方根是 7，但计算机不是按精确的方式匹配数学运算，浮点数仅仅是近似，sqrt(49) 返回的结果可能是 6.999999999999，尽管这个数比 7 小一点点，但这个差别足以影响测试的结果。

```
i <= limit;
```

上述程序中的这个测试需要在数学上完全精确，用浮点数不能保证这一点。如果 i 是 7，而 limit 是 6.999999999999，则循环的最后一个周期将不会执行，程序永远不糊检查 n 是否可以整除 7，因为 7 是 49 唯一的因子。不检查 7 是否为 49 的因子意味着 49 将被错误地分类为素数。另一方面，如果 sqrt(49) 返回的是 7.0 或 7.0000000000001，IsPrime 将会给出正确的答案，因此这个实现的正确性取决于硬件如何执行浮点数计算。

让实现的正确性依赖于运行它的计算机的特性是一个严重的错误，而解决这个问题的方法也很简单，可以修改这个程序使之与机器的精度无关。如果 n 的平方根小于某一界限，为了确保精确，函数总是多检查一个可能的约数，而多测试一个约数没什么害处，这是确保答案正确性所付出的很小的代价，因此要改正程序，所需要做的只是修改 limit 的赋值语句为：

```
limit = sqrt(n) + 1;
```

将 limit 声明为 int 类型也是一个很好的程序设计的习惯，这样可以确保 for 循环控制行中所用的所有值都是整数。

由上面的测试可以得出 IsPrime 的最后版本如下所示，它现在已经纠正了所有的错误。

```
bool IsPrime(int n)
{
  int i, limit;

  if(n <= 1) return(FALSE);
  if(n == 2) return(TRUE);
  if(n % 2 == 0) return(FALSE);

  limit = sqrt(n) + 1;
  for(i = 3; i <= limit; i +=2){
    if(n % i == 0) return(FALSE);
  }
  return(TRUE);
}
```

在改善程序的其他方面时，决不能牺牲正确性。

38.4 Tradeoff alternative implementations

比较 IsPrime 的最初实现方案和最终版本，相对的优势在于，最后的版本更有效，效率对任何将这个函数作为大应用的一部分的用户都有重大的意义。另一方面，必须认识到最初的版本也有它的优势，特别是，程序的第一个版本可读性更好，它更容易看出实现产生的结果是否服从素数的定义，而最后一个版本的复杂性反映在更难使它工作。

当为某一个问题选择一个算法时，最关心的是正确性，而使程序更加有效的是一个极好的目标，但不要为追求效率而使程序给出错误的答案。

在保证了算法的正确性以后，还有些因素对程序设计的成功也是很重要的，其中包括效率、清晰度和可维护性等。

对于实际的应用而言，效率是非常重要的，尤其是对于需要快速响应的应用，例如，一个花 5 分钟才能检测两架飞机是否在碰撞路线上的航空交通控制系统是无用的；而能在 1 秒钟内发出警告的系统可能会救人性命。

在多个程序员合作完成一个程序的环境中，清晰度应该是优先级最高的问题，每个程序员必须能读懂自己之外的人所编写的代码的含义，因此每个人都必须采用各种手段努力帮助别人，这样大家才能很容易地工作在一起。

对于一个希望使用很长时间的程序来说，它的可维护性是最基本的要求，而一个经过仔细设计并使之能适应变化的程序比没有考虑完善的程序更容易维护。

必须要明白，如果按照某些标准，一个算法可能比另一个算法要好，而从全局的观点来看，通常没有一个算法是最好的。例如，一个在运行时间上优势突出的算法可能难以理解，这会给其他人造成使用上的难度；相反，清晰度好的程序通常不是特别有效。用哪一个程序取决于应用的需求，这种考虑称为权衡（tradeoff），在各个竞争因素中找到平衡是很重要的。

第 39 章

Greatest Common Divisor

另一个问题可以更有效地说明算法的选择会影响效率，一个特别好的实例就是找出最大公约数的问题，这个问题也来自于经典数学。

给出两个数 x、y，其最大公约数（Greatest Common Divisor，缩写为 GCD）是能够同时把这两个数整除的最大数。编写函数，它有两个整型数的参数，返回的是它们的最大公约数，该函数的原型可以写为：

```
int GCD(int x, int y);
```

利用最基本的测试算法，这个问题可以有几个可用的解决方案。

39.1 brute-force algorithm

计算 GCD 的最简单的方法基于最通用的方法，称为 brute-force 方法（brute-force method），该方法测试每一种可能性。

开始，简单地“猜测”GCD(x, y) 是 x（它不能大于 x，但仍能被 x 整除），然后检查这个假设，用 x 和 y 除以这个值，看能否整除。如果能整除，答案即可以确定；如果不能整除，将这个假设值减 1，再继续测试。

使用 brute-force 策略的函数如下：

```
int GCD(int x, int y)
{
  int g;
  g = x;
  while(x % g != 0 || y % g != 0){
    g--;
  }
  return (g);
}
```

在设计算法时，以下几个问题必须要考虑：

1. GCD 的 brute-force 的实现是不是总能给出正确的答案；
2. 该算法是不是总能结束；
3. 该函数是否可能永远运行。

要确定程序是否总能给出正确的答案，需要考虑 while 循环中的条件：

```
x % g != 0 || y % g != 0
```

while 的条件指出在什么情况下循环将继续，为了找出什么样的条件会使循环终止，必须把 while 的条件取反，对一个包含 && 或 || 的条件取反可以用德·摩根定律来完成。

德·摩根定律指出在 while 循环的出口处，下列条件必须成立：

```
x % g == 0 && y % g == 0
```

从这个条件可以立即看出，g 的最终值一定是公约数。

要确定 g 事实就是最大公约数，必须考虑嵌在 while 循环体中的策略。策略中的关键因素是该程序反向测试所有的可能性，GCD 永远不可能大于 x（或 y，对于该问题而言），brute-force 算法从这个值

开始搜索，如果程序一旦跳出 while 循环，它肯定已经测试了 x 和 g 的当前值之间的每一个值，因此如果有一个较大的值能把 x 和 y 整除，那么程序在较早的 while 循环的迭代中就会发现这个数。

为了说明函数会终止，必须首先认识到即使没有找到最大公约数，g 的值最终会到达 1，此时 while 循环肯定会终止，因为不管 x 和 y 取什么值，1 总能把 x 和 y 整除。

39.2 Euclid algorithm

GCD 实现的 brute-force 算法在效率上同样不是很好，例如考虑一下对 1000005 和 1000000 调用该函数时，brute-force 算法在找到公约数 5 之前将运行 100 万次 while 循环的循环体。

现在需要考虑另一种算法，它要比 brute-force 算法所需的执行步骤少，但却保证能结束，并返回正确的答案，这是聪明性和易理解所付出的代价。

在古希腊的数学家欧几里德的著作 Elements（book 7，Proposition III）中包含了一个解决这个问题的算法[1]，使用自然语言可以把欧几里德算法描述如下：

1. 取 x 除以 y 的余数，称余数为 r；
2. 如果 r 是 0，过程完成，答案是 y；
3. 如果 r 非 0，设 x 等于原来 y 的值，y 等于 r，重复整个过程。

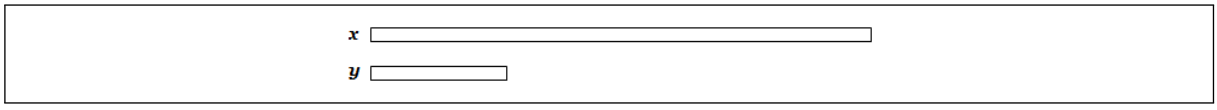
将该算法翻译为 GCD 函数的实现如下：

```
int GCD(int x, int y)
{
  int r;
  while(TRUE){
    r = x % y;
    if(r == 0) break;
    x = y;
    y = r;
  }
  return (y);
}
```

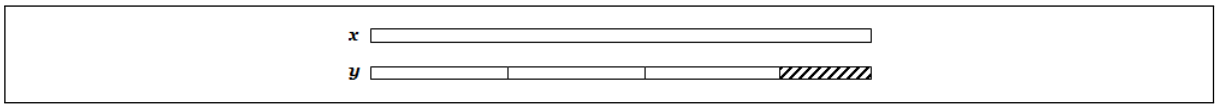
GCD 算法的欧几里德实现同样能找出两个数的最大公约数，它与 brute-force 的实现有两点不同，一方面，它能更快地计算出结果；另一方面很难证明它的正确性。

虽然很难给出欧几里德算法正确性的形式化证明，但可以通过古希腊人所用的数学思想模型给出这个证明的概述。

在古希腊数学中，几何学占据了中心位置，数字被认为是距离，例如当欧几里德开始找两个数（如 55 和 15）的最大公约数时，他将这个问题设想成找出一根最大的用于测量这两个距离的木棍，该木棍能用来划分两个距离中的每一个距离，因此，可以把这个问题想象为两根木棍，一根 55 单位长，另一根 15 单位长，如下所示：



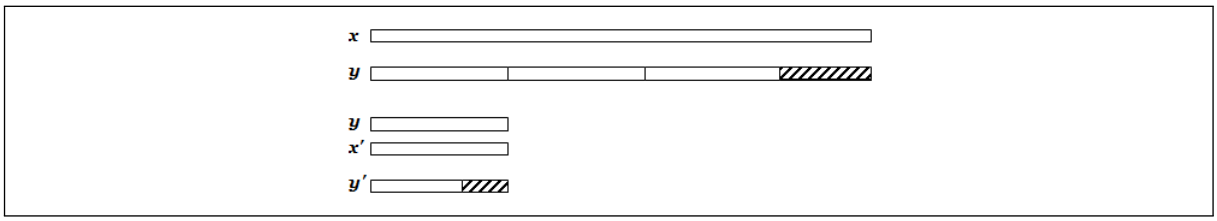
该问题就转化为要找到一个新的测量木棍，以该木棍为单位可以均分两根木棍 x 和 y，欧几里德算法从用较短的一个距离作为单位来划分长的木棍开始：



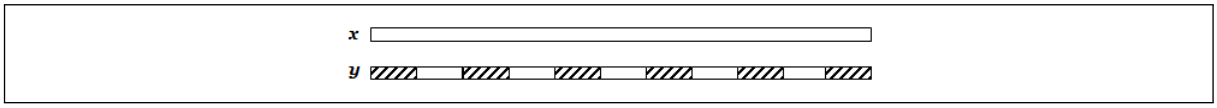
[1]欧几里得算法被人们认为是史上第一个算法。

除非较大的数能完全整除较小的数，否则总会有一个余数，如上图中的阴影部分所示。在这个例子中，以 15 为单位可以将 55 分为三份，但还余 10，这意味着阴影部分是 10 个单位长，欧几里德算法最基本的想法是原先两个距离的最大公约数也必须是较短的木棍的长度和途中阴影部分表示的距离的最大公约数。

有了这个观察结果，解决原先的问题可简化为涉及两个较小的数得更简单的问题，这里的两个新的数是 15 和 10，可以重新运用欧几里德算法找出它们的 GCD。可以从把两个新值 x 和 y 作为合适的测量长度着手，然后用较小的一个长度作为划分较长的长度的尺度。



这个过程再一次产生了一段剩余的区域，这次长度为 5。如果再重复一次这个过程，发现长度为 5 的阴影部分本身就是 x 和 y 的公约数，因此，根据欧几里德的思想，也是原来两个数 x 和 y 的公因子，于是在下面的图中表明这个新的值确实是原先两个数的公约数。



在 Element 这本书里，欧几里德提供了这个命题的完整的证明。

具体来说，求两个自然数变量 M 和 N 的最大公约数的算法如下：

1. 如果 M < N，则交换 M 和 N
2. M 被 N 除，得到余数 R
3. 判断 R = 0，正确则 N 即为“最大公约数”，否则下一步
4. 将 N 赋值给 M，将 R 赋值给 N，重做第一步。

下面是使用 ANSI C 实现的求最大公约数的算法步骤。

```
void swapi(int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int gcd(int m, int n)
{
  int r;
  do
  {
    if (m < n)
      swapi(&m, &n);
    r = m % n;
    m = n;
    n = r;
  } while (r);
  return m;
}
```

利用 if 函数以及递归则能做出更为精简的代码，更可省去交换的麻烦。

```
int gcd(int a,int b)
```

```
{
   if(a%b)
      return gcd(b,a%b);
   return b;
}
```

为了说明 brute-force 和 Euclid 算法效率的不同，考虑两个数 1000005 和 1000000，为了找出这两个数的 GCD，brute-force 算法需要执行一百万步，而欧几里德算法只需要计算两步。

这里，欧几里德算法开始时，x 是 1000005，y 是 1000000，在第一个循环周期中，r 被设为 5，由于 r 的值不是 0，程序设 x 为 100000，设 y 为 5，重新执行，在第二个循环周期，r 的新值为 0，由此程序从 while 循环中退出并报告答案为 5。

从而可以看出，算法的选择对于解决方案的效率有极大的影响。

39.3 Mathematical Methods

不全为 0 的两个整数的最大公约数的确存在，因为这两个整数的公约数集合是有限的，求这两个整数的最大公约数的一个方法是求出两个整数的所有正的公约数，然后去其中最大的。

定义：如果整数 a 和 b 的最大公约数是 1，就说它们是互素的。

定义：如果整数 a1、a2、...、an 是两两互素的，如果只要 1 ≤ i < j ≤ n，就有 gcd(ai, aj) = 1。

因而求两个整数的最大公约数的另一个方法是利用这两个整数的素因子分解，假定两个全不为 0 的整数 a 和 b 的素因子分解为：

$$a = p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n}, b = p_1^{b_1} p_2^{b_2} \cdots p_n^{b_n}$$

其中每个指数都是非负整数，而且出现在 a 和 b 分解中的所有素数都包含在两个分解之中，必要时以 0 为指数出现，于是 gcd(a, b) 由下面的公式给出：

$$gcd(a,b) = p_1^{min(a_1,b_1)} p_2^{min(a_2,b_2)} \cdots p_n^{min(a_n,b_n)}$$

其中 min(x, y) 代表两个数 x 和 y 的最小值，为证明这一计算 gcd(a, b) 的公式是有效的，必须证明等式右边的整数整除 a 和 b，而且没有比它大的数也能整除 a 和 b。

由于右边这个数中每个素数的指数都不超过 a 和 b 的分解中该素数的指数，所以它的确能整除 a 和 b，进一步说，没有更大的整数解整除 a 和 b，因为分解式中每个素数的指数都不能再增大，而且其他素数也不能加进来。

例：120 和 500 的素因子分解分别是

$$120 = 2^3 \cdot 3 \cdot 5$$

和

$$500 = 2^2 \cdot 5^3$$

所以它们的最大公约数是：

$$gcd(120,500) = 2^{min(3,2)} \cdot 3^{min(1,0)} \cdot 5^{min(1,3)} = 2^2 \cdot 3^0 \cdot 5^1 = 20$$

例：55 和 15 的素因子分解分别为 55 = 5·11 和 15 = 3·5，所以它们的最大公约数是：

$$gcd(55,15) = 3^{min(1,0)} \cdot 5^{min(1,1)} \cdot 11^{min(1,0)} = 3^0 \cdot 5^1 \cdot 11^0 = 5$$

第 40 章

Numerical algorithms

计算机科学中用来实现各种数学运算(包括函数等)的技术称为数值算法(numberical algorithm)。下面讨论的是实现计算平方根函数问题的两种不同的方法。

40.1 Successive approximation

解决所有数值问题的最通用的策略之一是连续逼近的技术。连续逼近(successive approximation)是找出近似答案的一般策略,它由下列步骤组成:

1. 先对答案进行猜测;
2. 一旦有了猜测值,就可以用猜测值产生一个更佳的值。例如,假设测试了你的答案,发现太大,可以把它变小一点,用这个小的值作为新的猜测值。相反,如果原先的猜测值太小,那么可以把它变大一点作为下一个检测值。
3. 如果可以保证,每一轮猜测与真实答案越来越接近,那么重复这个过程,最终会产生一个足以满足应用需要的猜测值。

连续逼近技术的困难部分是第三步——选择一个新的猜测值,使它满足第三步开始的条件。选择新的猜测值的策略必须考虑特定问题的一些知识。

在连续逼近的过程中所需要找到的是一系列的猜测值,每一个猜测值都比以前的猜测值更接近答案。随着序列的继续产生,可以使猜测值与答案达到任意接近的程度。如果随着计算的项越来越多,一系列的值会接近于一个极限,那么这个数列就称为收敛(converge)的。

用连续逼近方法解决平方根问题的策略是由艾萨克·牛顿在 17 世纪发明的,计算平方根的牛顿法最好用一个实例来说明。

假设要找出 16 的平方根,可以随意作一个猜测,但必须小于 16,例如可以从 8 开始,而 82= 64 > 16,于是可以证明 8 太大了,因而需要选择一个较小的数,但要确定的是,到底要小多少以及有没有使用这个问题的某些性质来选择下一个猜测值的方法?

牛顿的观点是正确的平方根一定介于当前的猜测值和原先的值除以猜测值的结果之间。在这个实例中,猜测值是 8,16 除以 8 是 2,也就是说,8 太大了,2 太小了,这两个值被认为是位于正确答案的两侧。

为了保证更精确的猜测,可以取两个值的平均值,结果将更接近于 16 的平方根,虽然仍然是一个近似值。在这个实例中,将 8 和 2 的平均值 5 作为新的猜测值。

此时,只要简单地重复这个过程即可,16 除以 5 是 3.2,平均 5 和 3.2 产生一个新的猜测值 4.1,它此时更接近于正确答案。如果照此继续这个过程,后两个猜测值是 4.001219512 和 4.00000018584。通过第四个猜测值,牛顿法已经产生了一个非常接近于正确答案的值,这个过程永远也不会产生一个完全正确的值,但可以继续应用这个技术直至得到一个满足所要求接近程度的近似值。

而如果永远不能得到一个完全正确的值,那如何知道什么时候该停下来呢?最直接的方法是继续这个过程直到答案“足够接近”。

“足够接近”的含义是计算结果和确切值之间的差足够小,以至于可以忽略这个差值。该策略还有另外一个重要的好处,可以直接使用下面提供的决定该过程是否完成所需的工具。在这个过程中,实现了一个函数 ApproximatelyEqual,如果两个浮点数在 Epsilon 指定的精度范围内相等,则返回 TRUE。

Epsilon 被定义为程序的一部分，如果把 ApproximatelyEqual 函数拷贝到平方根程序中，一旦猜测值的平方近似等于原先值，程序可以停止。

下面介绍谓词函数 ApproximatelyEqual(x, y)，如果两个浮点数 x 和 y 近似相等，则返回 TRUE。所谓的近似相等是两个值差的绝对值除以两个值中绝对值较小的值的结果小于一个常数 ε，用数学公式可以表示为：

$$\frac{|x-y|}{min(|x|,|y|)} < \varepsilon$$

在程序中，可以用 #define Epsilon 0.000001 定义常量 ε，用 #define 可以很容易地改变希望取到的精度，也可以用 math 库中的 fabs 函数取浮点数的绝对值。

ApproximatelyEqual 函数在避免判断浮点数相等时遇到的精度问题是很有用的。为了说明这个原理，用 ApproximatelyEqual 函数为 for 循环构建一个正确的测试条件：

```
for(x = 1.0; x <= 2.0; x += 0.1){
  printf( "%.1f\n" , x);
}
```

使得循环正确地显示值 1.0、1.1、1.2 等，包括 2.0，由于浮点数精度的限制，在某些机器上这个循环将不会包括值 2.0。

这个非形式化处理过程可以用如下更形式化的自然语言表示：

1. 为了计算数 x 的平方根，从选择一个任意的猜测值 g 开始，一种可能就是将 g 设为 x，尽管也可以选择任何其他的正数值。
2. 如果猜测值 g 足够接近于正确的平方根，算法结束。函数将 g 作为结果返回。
3. 如果 g 不够精确，用 g 和 x/g 的平均值作为新的猜测值。因为这两个值中的一个小于确切的平方根，另一个则大于确切的平方根，选择平均值会得到一个更接近于正确答案的值。
4. 把新的猜测值存入变量 g，从第二步开始重复这个过程。

接下来在实现这个算法时，函数被命名为 Sqrt(将 S 大写以区别于 math 库中定义的 sqrt 函数)。

```
double Sqrt(double x)
{
  double g;
  if(x == 0) return (0);
  if(x < 0) Error( "Sqrt called with negative argument %g" , x);
  g = x;
  while(!ApproximatelyEqual(x, g*g)){
    g = (g + x / g) / 2;
  }
  return (g);
}
```

Sqrt 的实现中的前面两个语句用来说明两个重要的情况，首先 0 的平方根是 0。

```
if(x == 0) return (0);
```

如果没有该语句，在第一个循环周期中计算机 x / g 时，牛顿法将会遇上除 0 而被终止。在数学上，除 0 是无意义的。计算机程序中如果除 0 的话，程序的行为是不可预测的，因而这里要先检查 x 是否为 0，并立即返回一个正确答案。

函数中的第二条语句用于处理给 Sqrt 函数传入一个负数的情况。

```
if(x < 0) Error( "Sqrt called with negative argument %g" , x);
```

在数学中，平方根函数除了对虚数之外，对负数无意义，但在设计程序时应对是否会错误地传入一个负值而进行应对，而且这种情况下，函数应该报告错误，使得它的调用程序知道出现了这种情况。

40.2 Error handling

程序中有很多种方法报告执行过程中出现的错误情况，在上面的例子中使用定义在扩展库 genlib 中的 Error 函数向用户报告错误。

与 printf 函数一样，Error 函数取一个控制字符串后跟另一个参数。当 Error 被调用时，会显示一个字符串“Error:”，后跟一个控制字符串，和 printf 函数的格式码的用法一致，在格式代码的位置将会用一个值替换。在行尾，Error 函数会自动包括一个换行符，将光标移到下一行的开始。

Error 与 printf 函数的主要区别在于，Error 不用返回到它的调用程序，而是一旦显示错误信息，整个程序就终止，如同已经执行完主程序中的所有语句一样，不再执行任何语句。

在程序中，响应一个错误的过程称为错误处理(error handling)，使一个程序输出一个错误信息，并停止执行并不是特别复杂的错误处理策略，尤其是在库函数中，最好能提供一些调用程序可以用来采取正确行为的机制。当错误情况出现时，程序作出某些反应是非常重要的。如果不检查这些情况，程序可能给出不正确的答案，或根本不可能产生答案。例如，如果 Sqrt 不检查负数参数，那么为了估计一个不存在的答案，程序将进入无限循环，使一个程序报告错误并终止是比使它永远运行更有用的响应。

40.3 Series expansion

牛顿法是计算平方根问题的一种可行的方法，另外还有一些方法可以更容易地应用在实际问题中，其中应用最广泛的方法之一是级数展开(series expansion)。在这种方法中，函数的值通过对级数项的求和来估计，如果每加入一个新的项都会使总和更接近于一个所期望的值，则级数是收敛的，可以用这个级数展开法估计结果。

为了说明级数展开的思想，先引入悖论的概念。

在公元前 5 世纪，埃里亚的哲学家 Zeno 发明了悖论。他建议该行为是不可行的，假设你想穿过一个房间，要做到这一点，首先必须走到一半的地方，从这个地方开始，再走过余下的一半，这意味着再走过原始距离的四分之一。从这里开始，必须再走总距离的八分之一到下一个中点，一直这样走下去，因为余下的距离总是一个能被一分为二的量，所以 Zeno 认为，这个过程将永远不会结束，也就是说，永远不可能到达目的地，这个论点称为 Zeno 悖论(Zeno's paradox)。

在 Zeno 悖论中，横穿房间这个过程中的每一步都是走到剩余距离的一半，如果把 Zero 问题以数学形式表示，那么已经走过的每段距离的总和可以表示为下式：

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\cdots+\frac{1}{2^n}+\cdots$$

Zeno 正确地观察到在这个级数中有无限个项，但 Zeno 在公元前 5 世纪时可能还无法认识到无限级数可以有一个有限的和，这是事实，虽然看上去有些矛盾。

在一个数学的级数中，被加起来形成总和的每一个值称为项，在函数中的项本身可以涉及一些数学运算(如上面的 Zeno 计算中每个项所用的分数标记)，所以级数中的项的定义与在程序设计语言中表达式的讨论中所用的项的定义有所不同，在表达式中，项是一个表示单独的数据值。

Zeno 级数的和是什么？从数学的观点来看，可以用 S 来标记级数的和：

$$S=\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\cdots+\frac{1}{2^n}+\cdots$$

将等式的两边都乘以 2，则结果的等式为：

$$2S=1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots+\frac{1}{2^{n-1}}+\frac{1}{2^n}+\cdots$$

此时，等式的右边第一项以后，该无限级数正好与原来的级数有完全一样的项，因此，这些项可以

用 S 来替换，于是产生了一个更简单的公式：

$$2S = S + 1$$

于是可以得到 S = 1。

从计算机程序设计的角度可以写一个程序通过把这些项加在一起来计算这个级数的近似值：

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots + \frac{1}{2^n} + \cdots$$

从 1 开始给这些项编号，则和式中的第 i 项可写为 $\frac{1}{2^i}$。

这种第 i 项 $\frac{1}{2^i}$ 涉及 i 次幂的数学级数称为幂级数（power series），幂级数的计算在计算数学中有非常重要的作用。

由于已经有了一个按照项的编号计算每一项的值的公式，所以可以很容易地编写一个程序，用这个公式计算每一项的值，然后将其加到一个总和中。

其实，大多数幂级数都有更简单的计算方法，与其从头计算每一项，倒不如从前一项计算后一项，从而节省时间。在 Zeno 级数中，第一项是 $\frac{1}{2}$，后一项总是前一项的一半，因此，如果保持当前项的总和，程序可以用下列结构找到级数的和。

```
sum = 0;
term = 0.5;
while(TRUE){ /*该程序片段包含无限循环。*/
  sum += term;
  term /= 2;
}
```

该程序在实现上的难度在于它永不停止，while 语句没有出口的条件，因而使程序进入无限循环，在实际编程时要避免该问题。

在编写一个实现级数展开的程序时，浮点数仅是实数的一个估计值的事实有着非常大的作用。随着级数的展开，term 的值越来越小，到某些点时，term 的值小到机器的有限精度已经不能表示 sum + term 的精确值，例如，假设 sum 的值为 2.3，而 term 的值为 0.0000000000000000001。当然，此时在数学上 sum + term 的值应为 2.3000000000000000001，但是机器可能无法表示 20 位精度的浮点数，因此机器输出的答案可能只是 2.3，term 的值在最终将变得毫无意义，该事实导出了停止 while 循环的有效策略，如果用

```
while(sum != sum + term)
```

取代无限循环中的 while 控制行，那么只要 term 加到 sum 上时产生一些可注意到的影响，则循环将继续。

计算 Zeno 级数的完整的主程序如下：

```
main()
{
  double sum, term;
  sum = 0;
  term = 0.5;
  while(sum != sum +term){
    sum += term;
    term /= 2;
  }
  printf( "The term of Zeno' s series is %g\n" , sum);
}
```

40.4 Taylor Series

许多数学函数都有一个相关的幂级数，从而可以使用求和技术产生近似结果，可以使用 Taylor 级数展开来估计平方根。

18 世纪的数学家布鲁克·泰勒发现，至少对一定范围内的参数值，连续可导的函数可以用以下公式近似表示：

$$f(x) \doteq f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + f'''(a)\frac{(x-a)^3}{3!} + \cdots + f^{(n)}(a)\frac{(x-a)^n}{n!}$$

在这个公式中，符号 a 表示一个常量，记号 f'、f'' 和 f''' 等分别表示函数 f 的一阶、二阶和三阶导数，以此类推，上面这个公式称为 Taylor 公式（Taylor's formula）。

考虑 f 是平方根函数的情况，在数学中，取 x 的平方根与求 x 的 $\frac{1}{2}$ 次幂是一样的，因此 f(x) 可以定义为 $f(x) = x^{\frac{1}{2}}$。

为找出 f'，对于用 x^c 求导的通用公式有 $f' = cx^{c-1}$，其中 c 为任意常数，在平方根函数中，c 是 $\frac{1}{2}$，因此一阶导数是 $f' = \frac{1}{2}x^{1\frac{1}{2}}$。

然后，运用同样的规则产生另外的一些导数，每一次都用前一次系数乘以前一次的指数，然后指数减 1，于是平方根函数的后面的高阶导数即为：

$$\begin{aligned} f''(x) &= -\frac{1}{4}x^{-\frac{3}{2}} \\ f'''(x) &= \frac{3}{8}x^{-\frac{5}{2}} \\ &\cdots \end{aligned}$$

在 Taylor 公式中，这些导数用常数 a 计算，因此选择一个容易计算这些导数的常数是很重要的。例如，如果设 a 等于 1，1 的任意次幂总是 1，上述公式就只剩下系数，如下所示：

$$\begin{aligned} f(1) &= 1 \\ f'(1) &= \frac{1}{2} \\ f''(1) &= -\frac{1}{4} \\ f'''(1) &= \frac{3}{8} \\ &\cdots \end{aligned}$$

将这些值代入 Taylor 公式中，则下面的幂级数给出了平方根函数的近似值：

$$\sqrt{2} \doteq 1 + \frac{1}{2}(x-1) - \frac{1}{4}\frac{(x-1)^2}{2!} + \frac{3}{8}\frac{(x-1)^3}{3!} - \frac{15}{16}\frac{(x-1)^4}{4!} + \cdots + f^{(n)}(1)\frac{(x-1)^n}{n!}$$

另外，只要再满足一个条件，Taylor 级数就可以在一定范围内通用，在此级数中计算的项越多，结果就越精确。

这样，通过对 Taylor 级数的分析，可以通过幂级数展开计算平方根的近似值。

使用计算机求解时，就需要通过一个函数实现上面的公式，而计算幂级数的和的最有效的策略是找出用前一项计算新的项的方法。为了将这个原则也用于估计平方根的 Taylor 级数展开，可以从给这些项编号开始。

$$
\begin{aligned}
t_0 &= 1 \\
t_1 &= \frac{1}{2}(x-1) \\
t_2 &= -\frac{1}{4}\frac{(x-1)^2}{2!} \\
t_3 &= \frac{3}{8}\frac{(x-1)^3}{3!} \\
t_4 &= -\frac{15}{16}\frac{(x-1)^4}{4!} \\
&\cdots \\
t_n &= f^{(n)}(1)\frac{(x-1)^n}{n!}
\end{aligned}
$$

为了分析后一项与其前一项有什么不同，可以认为每一项都有三个独立的部分：

$$\text{coeff}\frac{\text{xpower}}{\text{factorial}}$$

从而这个问题变为研究这几个部分分别是如何随项的变化而变化的。

考虑 xpower 部分，将当前的 xpower 乘以 (x – 1) 就得到了下一项的 xpower，把当前的 xpower 值保存在一个变量中，就能确保在每个循环周期中通过一次乘法更新这个当前值。

关于 factorial 部分，对第 i 项，这个部分的值总是 i!，阶乘的特性是任何数 i 的阶乘是 i 乘以比它小的那个数的阶乘，即

$$(i+1)! = (i+1)\cdot i\cdot(i\text{–}1)\cdot(i\text{–}2)\cdot\cdots\cdot 2\cdot 1 = (i+1)\cdot i!$$

因此，为了准备第 (i + 1) 个循环周期，所需要做的就是把当前的 factorial 乘以 (i + 1)。

而对于 coeff 部分，与其他的项一样，它也是由乘积产生的，一般来讲，为了得到第 (i + 1) 个循环周期的 coeff 值，只需要简单地把第 i 个周期的值乘以 $\frac{1}{2} - i$。

从而可以利用这些公式从每个部分的当前值中计算出下一项中各部分的值。如果变量 coeff、xpower 和 factorial 包含当前项的这些部分，那么为了计算下一项中这些部分的值，所需要的只是将下列语句作为函数的主循环：

```
coeff *= (0.5 –i);
xpower *= (x –1);
factorial *= (i + 1);
```

正如计算 Zeno 级数求和的程序一样，主循环必须执行到加入一个项的值不会改变总和的值为止，可以写一个循环测试这个条件，并初始化所有的变量，从而产生如下的函数 TSqrt。

```
double TSqrt(double x)
{
  double sum, factorial, coeff, term, xpower;
  int i;
  factorial = coeff = xpower = 1;
  sum = 0;
  term = 1;
  for(i = 0; sum != sum +term; i ++){
    sum += term;
    coeff *= (0.5 –i);
    xpower *= (x –1);
    factorial *= (i + 1);
    term = coeff * xpower / factorial; /*如果x≥2，则函数失败 */
  }
  return(sum);
}
```

由于上面给出的 TSqrt 函数并不能通用，因为作为 Taylor 级数展开时，平方根函数的公式仅当参数值位于一个有效范围内时才有效。在该范围内计算趋于收敛，也就能达到`sum = sum + term`的条件，因而该范围被称为收敛半径(radius of convergence)。

对平方根函数用 a = 1 计算，Taylor 级数公式希望 x 处于范围 0 < x < 2 之间。如果 x 在收敛半径外，则展开式中的项会越来越大，Taylor 级数离答案也就越来越远，这个限制减小了 TSqrt 函数的用途，虽然在这个范围内，它还是有效的。

当面临这种限制时，应该想一种方法把这个通用问题转化为某一特定问题，这个问题正好能满足当前这个解决方案的需要，因此需要找到一种根据落在范围内的平方根计算落在范围外的大数的平方根的方法。

为了实现这个问题的转换，参考方式是：

$$\sqrt{4x} = \sqrt{4}\sqrt{x} = 2\sqrt{x}$$

由此可知，平方根内的 4 可以转换为平方根外的 2。

这个结论提供了一种完成该解决方案的工具，如果有一个用 4 去除它，而用 TSqrt 函数计算余数的平方根，然后再用 2 乘以这个结果，如果除一次 4 还不能使它落在所期望的范围内，则在调用 TSqrt 前多除几次 4，只要最后把结果再乘以同样次数的 2 即可。

处理这个解决方案中新的工作的最简单的方法是将平方根函数分解为两部分，前面已经实现了在有限范围内计算平方根的函数 TSqrt，不改变这个函数，可以写一个 Sqrt 函数，该函数执行必要的除法，使得这个数落在收敛半径内，并在最后完成相应的乘法，以修正答案。与前面的要求一致，该函数也需要在适当的地方加入检查 x 为 0 或负数这些特殊情况的语句。

```
/*
* Function:Sqrt
* Usage:root = Sqrt(x);
* --------------------------
* Returns the square root of x, calculated using a Taylor series expansion,
* as described in the text. The Sqrt function is actually implemented as
* two functions. The job of the out Sqrt function is to divide the argument
* repeatedly by 4 until it is in the range 0 < x < 2, where the Taylor series
* converges. It then calls TSqrt to perform the actual Taylor series calculation.
* When finished, Sqrt adjusts the answer by multiplying the result by 2 for
* each time it needed to be divided by to bring it in range.
*/

double Sqrt(double x)
{
  double result, correction;
  if(x == 0) return 0;
  if(x < 0) Error( "Sqrt called with negative argument %g" , x);
  correction = 1;
  while(x >= 2){
    x /= 4;
    correction *= 2;
  }
  return (TSqrt(x) * correction);
}

/*
* Function:TSqrt
* Usage: root = TSqrt(x);
* --------------------------
* Returns the square root of x, calculated by expanding the Taylor series around
* a = 1, as described in the text. The function is effective only if x is in the range
* 0 < x < 2. Term i in the summation has the form.
```

```
*                xpower
*     coeff * ---------------
*                factorial
* where coeff comes from the derivative of the function, factorial is i!, and xpower
* is the ith power of (x -a). Each of these components is computed from its previous
* value.
*/

double TSqrt(double x)
{
  . . .
}
```

40.5 Numeric Types

在数值算法中，控制数值数据的范围和精度是很重要的。在某种程度上，这些特性取决于硬件，因为计算机内的所有数据都存储在它的内存系统中，不同类型的计算机系统内存划分单元的大小可以不同。

从程序员的观点来看，内部结构的不同，其影响主要体现在数值型的数据在不同的机器上可能受到不同的限制，这使得编写一个可移植的程序非常困难。

可移植（portable）程序是一个能在不同的计算机系统上以完全相同的方式成功运行的程序。为了满足编写可移植程序的需要，并保证数值计算能达到所需的精度，ANSI C 包括了一些不同的整型和浮点类型，虽然 int 和 double 已经可以满足大多数数值应用，但有时使用一些其他的数值类型可能使程序更加完美，这样可以确保编译器在不同的计算机系统上都能预留足够的空间去保存数据的某些部分。

40.5.1 Integer Type

在计算机内存中，int 类型的数值被存放在一个单独的容量有限的单元中，因此这一单元的大小有一个上限，它取决于机器和所使用的 C 编译器。在 PC 中，int 类型的最大值为 32767，对计算标准来讲，这是相当小的，因为它不能实现计算每年有多少秒等的运算。

为了解决这个问题，C 语言定义了几种整数类型，它们之间的区别在于保存的数值范围的不同，其中三种主要的类型是 int、long 和 short，每一种都可以在前面加上一个关键字 unsigned。

数据类型 int 表示特定机器上的标准整数类型，ANSI C 确保 int 类型的值域至少要为 32767，对编译器的设计没有其他任何限制。在某些计算机系统上，int 类型可以有相当大的范围，在某些系统中 int 类型可以达到 2147483647 的上限。

注意，作为整数类型上限的数值并不是随便取的，计算机内部使用二进制系统存储，每个这样的上限总是 2 的某一个整数次幂减 1。

ANSI C 程序的设计者可以选择更精确地定义整数类型的可用范围。例如，可以声明每台机器上 int 类型的上限为 2147483647，如果这样做的话，程序从一个系统移植到另一个系统就更加地简单，它们在这些机器上都会有相同的行为。

为每台机器创建这样一条规则的问题在于，如果这样做的话，会迫使许多机器牺牲它的效率来确保遵守这条规则，如果某一计算机能够有效地实现小的整数，但使用一些大的整数时，必须执行一些额外的操作，迫使这样的机器用大的整数是要付出代价的。为了避免这个代价，因此 C 语言允许编译器用该机器最方便的长度表示 int 类型的值。

因为 C 语言对 int 类型的值所做的唯一的保证是它们至少为 32767，仅当能确保变量或表达式的值不会超过这个限制时才能使用 int 类型。如果能预料到值会大于 32767，最好用数据类型 long 告诉

编译器需要较大的整数范围。ANSI C 指定 long 类型的变量必须能至少保存上限为 2147483647 的值，但有些编译器可能允许保存更大的值。

数据类型 short 使程序员可以使用存储空间比 int 类型更少的变量，只是在语言中保留 short 类型主要是由于一些历史原因，在现代程序设计中已经很少使用。

在一个表达式中有不同大小的整数类型时，结果的类型应是计算中所涉及的最长的那个类型，例如，如果将一个 int 类型和 long 类型的变量相加，结果将是 long 类型，这条规则确保结果类型有与操作数相符的数值范围。

当使用 short 或 long 类型的值时，必须用不同的 printf 的格式码，为了显示 long 类型的值，必须在转换规格说明中%d 的 d 前面加一个字母 l，表明这个值是 long 类型。类似的，为了打印 short 类型的值，必须在 d 的前面加上代表 half 的字母 h，因为在早期的 C 语言版本中，short 类型的数占用的存储空间是 int 型的一半。

40.5.2 Unsigned Integer Type

在 C 语言中，整型数据 int、long 和 short 都可以在前面加上关键字 unsigned。加上 unsigned 后表示一种新的数据类型，这种类型仅允许表示非负值。

无符号变量不需要表示负数，声明一个无符号类型的变量可以保存两倍的整数值，例如 int 类型的最大数值为 32767，那么 unsigned int 类型的最大数值将是 65535。C 语言允许将 unsigned int 类型缩写为 unsigned。

unsigned int 类型的值在 printf 调用中用%u 来显示，要显示 unsigned long 和 unsigned short 可以分别用%lu 和%hu。

```
#include <stdio.h>

main()
{
  unsigned char i;

  for(i=0; i<=255; i++)
    printf("%d\n", i);
}
```

在计算机中，ASCII 字符集是最常用的字符集，使用 7 位代码 char 类型的取值范围是 0000000 ～ 1111111，因此字符可以看成是 0 ～ 127 的整数。

C 编译器以字符对应的整数值来处理字符，通常情况下，有符号字符类型的取值范围是-128 ～ 127，无符号字符类型的取值范围是 0 ～ 255。

```
  .file        "test.c"
  .intel_syntax noprefix
  .section     .rodata
.LC0:
  .string  "%d\n"
  .text
  .globl   main
  .type    main, @function
main:
.LFB0:
  .cfi_startproc
  push    rbp
  .cfi_def_cfa_offset 16
  .cfi_offset  6, -16
  mov     rbp, rsp
  .cfi_def_cfa_register 6
  sub     rsp, 16
```

```
  mov    BYTE PTR [rbp-1], 0
.L2:
  movzx  eax, BYTE PTR [rbp-1]
  mov    esi, eax
  mov    edi, OFFSET FLAT:.LC0
  mov    eax, 0
  call   printf
  movzx  eax, BYTE PTR [rbp-1]
  add    eax, 1
  mov    BYTE PTR [rbp-1], al
  jmp    .L2
  .cfi_endproc
.LFE0:
  .size      main, .-main
  .ident     "GCC: (GNU) 4.8.2 20131212 (Red Hat 4.8.2-7)"
  .section   .note.GNU-stack,"",@progbits
```

在上述代码中,在 char 类型前增加 unsigned 关键字将字符类型转换为无符号字符类型。

在字符十六进制编码规则中,0xFF 进一位是 0x00,因此编码最大可以取 255,而进位又回到循环的起始状态 0,导致代码中的 for 循环条件永为真,代码进入无限循环。

在大多数情况下,不需要使用无符号类型,除非需要对机器中的内部表示保持非常严格的控制。

40.5.3 Float Type

和整数一样,浮点数也在一定长度的内存单元中表示,因此也有一定的大小限制,然而对浮点数来讲,内存的限制影响了它的精度(可用的有效位有 n 位)以及范围。由于精度是硬件的函数,因此浮点数的精度随着机器的不同而不同,例如,若在某台机器上,浮点数可以有 10 位有效位,在这台机器上,值 1.0 和 1.000000000001 是一样的,因为机器不能精确表示 10 位以后的数。

与整型数一样,C 语言提供了三种类型的浮点数,float 是精度最小的,但比 double 类型占用的内存要少。随着内存越来越便宜,容量越来越大,现在程序员开始习惯于将浮点数表示为更精确的 double 类型。在 C 语言中,所有涉及 float 类型或 double 类型值的计算都把它当作 double 类型来处理,数据类型仅影响数据的存储。

然而,对一些精度要求非常高的计算,double 类型的精度可能还不够,对于这些需要非常高的精度的应用,ANSI 标准也包括了一种称为 long double 的类型。

第 41 章

Complexity

41.1 Time Complexity

算法的时间复杂度是指算法需要消耗的时间资源。一般来说，计算机算法是问题规模 n 的函数 f(n)，算法的时间复杂度也因此记作

$$T(n) = O(f(n))$$

算法执行时间的增长率与 f(n) 的增长率正相关，称作渐近时间复杂度（Asymptotic Time Complexity），简称时间复杂度。

常见的时间复杂度有：常数阶 O(1)，对数阶 O(log2n)，线性阶 O(n)，线性对数阶 O(nlog2n)，平方阶 $O(n^2)$，立方阶 $O(n^3)$，...，k 次方阶 $O(n^k)$，指数阶 $O(2^n)$。

随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

41.2 Space Complexity

算法的空间复杂度是指算法需要消耗的空间资源。其计算和表示方法与时间复杂度类似，一般都用复杂度的渐近性来表示。同时间复杂度相比，空间复杂度的分析要简单得多。

Part VI

Engineering

第 42 章

Introduction

42.1 History

Software engineering is the study and application of engineering to the design, development, and maintenance of software.

The field is one of the youngest of engineering, having been started in the early 1940s and the term itself not used until 1968. As such, there is still much controversy within the field itself as to what the term means and if it is even the best term to describe the field. Other terms such as software development and information technology are widely used instead.

When the first digital computers appeared in the early 1940s, the instructions to make them operate were wired into the machine. Practitioners quickly realized that this design was not flexible and came up with the ”stored program architecture” or von Neumann architecture. Thus the division between ”hardware” and ”software” began with abstraction being used to deal with the complexity of computing.

Programming languages started to appear in the 1950s and this was also another major step in abstraction. Major languages such as Fortran, ALGOL, and COBOL were released in the late 1950s to deal with scientific, algorithmic, and business problems respectively. E.W. Dijkstra wrote his seminal paper, ”Go To Statement Considered Harmful”, in 1968 and David Parnas introduced the key concept of modularity and information hiding in 1972 to help programmers deal with the ever increasing complexity of software systems.

The term ”Software Engineering” was first used in 1968 as a title for the world's first conference on Software Engineering, sponsored and facilitated by NATO. The conference was attended by international experts on software who agreed on defining best practices for software grounded in the application of engineering. The result of the conference is a report that defines how software should be developed [i.e., software engineering foundations]. The original report is publicly available.

The discipline of Software Engineering was coined to address poor quality of software, get projects exceeding time and budget under control, and ensure that software is built systematically, rigorously, measurably, on time, on budget, and within specification. Engineering already addresses all these issues, hence the same principles used in engineering can be applied to software. The widespread lack of best practices for software at the time was perceived as a ”software crisis”.

Barry W. Boehm documented several key advances to the field in his 1981 book, 'Software Engineering Economics'. These include his Constructive Cost Model (COCOMO), which relates software development effort for a program, in man-years T, to source lines of code (SLOC).

$$T = k * (SLOC)^{1+x}$$

The book analyzes sixty-three software projects and concludes the cost of fixing errors escalates as we move the project toward field use. The book also asserts that the key driver of software cost is the capability of the software development team.

In 1984, the Software Engineering Institute (SEI) was established as a federally funded research and development center headquartered on the campus of Carnegie Mellon University in Pittsburgh, Pennsylvania, United States. Watts Humphrey founded the SEI Software Process Program, aimed at understanding and managing the software

engineering process. His 1989 book, Managing the Software Process,[13] asserts that the Software Development Process can and should be controlled, measured, and improved. The Process Maturity Levels introduced would become the Capability Maturity Model Integration for Development(CMMi-DEV), which has defined how the US Government evaluates the abilities of a software development team.

The modern generally accepted practice for Software Engineering has been cataloged as a Guide to the Software Engineering Body of Knowledge (SWEBOK) which has become an internationally accepted standard ISO/IEC TR 19759:2005.

42.2 Subdisciplines

Software engineering can be divided into ten subdisciplines. They are:

- Software requirements: The elicitation, analysis, specification, and validation of requirements for software.
- Software design: The process of defining the architecture, components, interfaces, and other characteristics of a system or component. It is also defined as the result of that process.
- Software construction: The detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.
- Software testing: The dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.
- Software maintenance: The totality of activities required to provide cost-effective support to software.
- Software configuration management: The identification of the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle.
- Software engineering management: The application of management activities—planning, coordinating, measuring, monitoring, controlling, and reporting—to ensure that the development and maintenance of software is systematic, disciplined, and quantified.
- Software engineering process: The definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself.
- Software engineering tools and methods: The computer-based tools that are intended to assist the software life cycle processes, see Computer Aided Software Engineering, and the methods which impose structure on the software engineering activity with the goal of making the activity systematic and ultimately more likely to be successful.
- Software quality: The degree to which a set of inherent characteristics fulfills requirements.

Typical formal definitions of software engineering are:

- "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software".[39]
- "an engineering discipline that is concerned with all aspects of software production"[40]
- "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines"

The term has been used less formally:

- as the informal contemporary term for the broad range of activities that were formerly called computer programming and systems analysis;
- as the broad term for all aspects of the practice of computer programming, as opposed to the theory of computer programming, which is called computer science;
- as the term embodying the advocacy of a specific approach to computer programming, one that urges that it be treated as an engineering discipline rather than an art or a craft, and advocates the codification of recommended practices.

第 43 章

Overview

43.1 Definition

软件工程是研究和应用如何以系统性的、规范化的、可定量的过程化方法去开发和维护软件，以及如何把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来的学科。它涉及到程序设计语言、数据库、软件开发工具、系统平台、标准、设计模式等方面。

软件应用于现代社会的多个方面，包括电子邮件、嵌入式系统、人机界面、办公软件包、操作系统、编译器、数据库、游戏等。同时，各个行业几乎都有计算机软件的应用（比如工业、农业、银行、航空、政府部门等）。这些应用促进了经济和社会的发展，提高人们的工作效率，同时提升了生活质量。

软件工程师是对应用软件创造软件的人们的统称，软件工程师按照所处的领域不同可以分为系统分析员、系统架构师、软件设计师、程序员、测试工程师、界面与交互设计师等等。人们也常常用程序员来泛指各种软件工程师。

在 1970 年代和 1980 年代的软件危机中，许多软件最后都得到了一个悲惨的结局，软件项目开发时间大大超出了规划的时间表。

在 1986 年，IBM 大型电脑之父佛瑞德·布鲁克斯发表了他的著名论文《没有银弹》，在这篇著名的论文中他断言："在 10 年内无法找到解决软件危机的银弹"。从软件危机被提出以来，人们一直在寻找解决它的方法，并且有一系列的方法被提出并且加以应用（比如结构化程序设计、面向对象的开发、CMM、UML 等）。

与此同时，软件开发人员也发现软件开发的难度越来越大。在软件工程界被大量引用的案例是 Therac-25 的意外：在 1985 年六月到 1987 年一月之间，六个已知的医疗事故来自于 Therac-25 错误地超过剂量，导致患者死亡或严重辐射灼伤。

鉴于软件开发时所遭遇困境，北大西洋公约组织（NATO）在 1968 年举办了首次软件工程学术会议，并于会中提出"软件工程"来界定软件开发所需相关知识，并建议"软件开发应该是类似工程的活动"。

软件工程自 1968 年正式提出至今，这段时间累积了大量的研究成果，广泛地进行大量的技术实践，借由学术界和产业界的共同努力，软件工程正逐渐发展成为一门专业学科。

- 创立与使用健全的工程原则，以便经济地获得可靠且高效率的软件。
- 应用系统化，遵从原则，可被计量的方法来发展、操作及维护软件；也就是把工程应用到软件上。
- 与开发、管理及更新软件产品有关的理论、方法及工具。
- 一种知识或学科，目标是生产质量良好、准时交货、符合预算，并满足用户所需的软件。
- 实际应用科学知识在设计、建构电脑程序，与相伴而来所产生的文件，以及后续的操作和维护上。
- 使用与系统化生产和维护软件产品有关之技术与管理的知识，使软件开发与修改可在有限的时间与费用下进行。
- 建造由工程师团队所开发之大型软件系统有关的知识学科。
- 对软件分析、设计、实施及维护的一种系统化方法。
- 系统化地应用工具和技术于开发以计算机为主的应用。
- 软件工程是关于设计和开发优质软件。

ACM 与 IEEE Computer Society 联合修定的 SWEBOK（Software Engineering Body of Knowledge）提到，软件工程领域中的核心知识包括：

- 软件需求（Software requirements）
- 软件设计（Software design）
- 软件建构（Software construction）
- 软件测试（Software test）
- 软件维护与更新（Software maintenance）
- 软件构型管理（Software Configuration Management, SCM）
- 软件工程管理（Software Engineering Management）
- 软件开发过程（Software Development Process）
- 软件工程工具与方法（Software Engineering Tools and methods）
- 软件质量（Software Quality）

软件的开发到底是一门科学还是一门工程，这是一个被争论了很久的问题。实际上，软件开发兼有两者的特点，但是这并不意味着它们可以被互相混淆。很多人认为软件工程基于计算机科学和信息科学就如传统意义上的工程学之于物理和化学一样。

佛瑞德·布鲁克斯在《人月神话：软件项目管理之道（The Mythical Man-Month）》提到，将没有银子弹（silver bullet）可解决，开发软件的困难是内生的，只能渐进式的改善。整体环境没有改变以前，唯一可能的解，是依靠人的素质，培养优秀的工程师。

另外，Peter McBreen 认为软件工程意味着更高程度的严谨性与经过验证的流程，并不适合现阶段各类型的软件开发。Peter McBreen 在著作《Software Craftsmanship: The New Imperative》提出了所谓“craftsmanship”的说法，认为现阶段软件开发成功的关键因素，是开发者的技能，而不是“manufacturing”软件的流程。

Capers Jones 曾对美国软件组织的绩效做过评估，所得到结论是：软件工程的专业分工不足，是造成质量低落、时程延误、预算超支的最关键因素。2003 年的 The Standish Group 年度报告指出，在他们调查的 13522 个项目中，有 66％的软件项目失败、82％超出时程、48％推出时缺乏必需的功能，总计约 550 亿美元浪费在不良的项目、预算或软件估算上。

43.2 Comparation

作为计算机程序设计工业化的体现，软件工程存在于各种应用中，并贯穿于软件开发的各个方面。而程序设计通常包含了程序设计和编码的反复迭代的过程，它是软件开发的一个阶段。

软件开发过程是随着开发技术的演化而随之改进的。从早期的瀑布式（Waterfall）的开发模型到后来出现的螺旋式的迭代（Spiral）开发，以致最近开始兴起的敏捷软件开发（Agile），它们展示出了在不同的时代软件产业对于开发过程的不同的认识，以及对于不同类型项目的理解方法。

软件工程力图对软件项目的各个方面作出指导，从软件的可行性分析直到软件完成以后的维护工作。软件工程认为软件开发与各种市场活动密切相关。比如软件的销售，用户培训，与之相关的软件和硬件安装等。软件工程的方法学认为一个独立的程序员不应当脱离团队而进行开发，同时程序的编写不能够脱离软件的需求，设计，以及客户的利益。

43.3 Methodology

软件工程的方法有很多方面的意义，包括项目管理、分析、设计、程序的编写、测试和质量控制等。

软件设计方法可以区别为重量级的方法和轻量级的方法，其中：

- 著名的重量级开发方法包括 ISO 9000、CMM 和统一软件开发过程（RUP）等，而且重量级的方法中产生大量的正式文档。

- 轻量级的开发过程没有对大量正式文档的要求。著名的轻量级开发方法包括极限编程(XP)和敏捷过程(Agile Processes)。

“敏捷开发”(Agile Development)被认为是软件工程的一个重要的发展。它强调软件开发应当是能够对未来可能出现的变化和不确定性作出全面反应的。

敏捷开发被认为是一种“轻量级”的方法，而且在轻量级方法中最负盛名的应该是“极限编程”(Extreme Programming)。

与轻量级方法相对应的是“重量级方法”的存在(包括 CMM/PSP/TSP 等)，重量级方法强调以开发过程为中心，而不是以人为中心。

面向方面的程序设计(Aspect Oriented Programming，简称 AOP)被认为是近年来软件工程的另外一个重要发展。这里的方面指的是完成一个功能的对象和函数的集合，与 AOP 相关的内容有泛型编程(Generic Programming)和模板等。

根据《新方法学》这篇文章的说法，重量级方法呈现的是一种“防御型”的姿态。在应用“重量级方法”的软件组织中，由于软件项目经理不参与或者很少参与程序设计，无法从细节上把握项目进度，因而会对项目产生“恐惧感”，不得不要求程序员不断撰写很多“软件开发文档”。而轻量级方法则呈现“进攻型”的姿态，这一点从 XP 方法特别强调的四个准则—“沟通、简单、反馈和勇气”上有所体现。目前有一些人认为，“重量级方法”适合于大型的软件团队(数十人以上)使用，而“轻量级方法”适合小型的软件团队(几人、十几人)使用。当然，关于重量级方法和轻量级方法的优劣存在很多争论，而各种方法也在不断进化中。

一些方法论者认为人们在开发中应当严格遵循并且实施这些方法。但是一些人并不具有实施这些方法的条件。实际上，采用何种方法开发软件取决于很多因素，同时受到环境的制约。

在实践中，注意区分软件开发过程和软件过程改进之间的重要区别是很重要的。诸如 ISO 15504、ISO 9000、CMM、CMMI 这样的名词阐述的是一些软件过程改进框架，它们提供了一系列的标准和策略来指导软件组织如何提升软件开发过程的质量、软件组织的能力，而不是给出具体的开发过程的定义。

第 44 章

Development

大多数程序都是由多个文件来构成的,其中典型的 C 语言程序一般都是由源文件(source file)以及头文件(header file)等组成的。

其中,源文件包含函数的定义和外部变量,而头文件包含可以在源文件之间共享的信息。

44.1 Source File

根据惯例,C 语言程序源文件的扩展名为.c,每个源文件包含程序的部分内容(主要是函数的定义和变量)。

C 语言源文件中必须有一个包含 main 函数并将其作为程序的起始点。

在下面的示例中计划编写一个计算机程序,用于计算按照逆波兰符号(Rerse Polish Notation,RPN)读入的整数表达式。

逆波兰符号是指运算符都跟在操作数的后面,如果输入表达式

```
30 5 - 7 *
```

计算器程序在计算上述该表达式的值时,首先要逐个读入操作数和运算符,因此可以使用栈来跟踪中间结果的方式计算逆波兰表达式。

- 如果程序读取数,就把数压入栈。
- 如果程序读取运算符,就从栈顶弹出两个数并进行相应的运算,然后把结果压入栈。

当程序执行到用户输入的末尾时,表达式的值将留在栈中。

计算机示例程序将按照下面的方式来计算上述表达式。

1. 把 30 压入栈。
2. 把 5 压入栈。
3. 从栈顶弹出两个数做减法运算(30-5=25),然后把结果压入栈。
4. 把 7 压入栈。
5. 从栈顶弹出两个数做乘法运算,然后把结果压入栈。

在把上述策略转换为程序时,main 函数将用循环来执行下列动作:

- 读取“符号”(数或运算符)。
- 如果符号是数,那么把它压入栈。
- 如果符号是运算符,那么从栈顶弹出两个操作数并进行运算,然后把结果压入栈。

对程序进行功能划分并分割成源文件时,可以把相关的函数和变量放入同一文件。

- 读入记号的函数可能和任何需要用到记号的函数一起属于某个源文件(例如 token.c)。
- 与栈操作相关的函数(例如 is_empty 函数、make_empty 函数、push 函数、pop 函数和 is_full 函数等)可能都属于另一个源文件(例如 stack.c),而且表示栈的变量也属于 stack.c 源文件。
- main 函数则属于与程序入口相关的源文件(例如 calc.c)。

把程序分割成多个源文件有许多显著的优点。

- 把相关的函数和变量集合在单独一个文件中可以明确程序的结构。
- 可以单独对每一个源文件进行编译。

如果程序规模很大而且需要频繁修改,这种方式可以极大地节约时间。

- 把函数集中在单独的源文件中，可以更容易在其他程序中重用这些函数。

> C 语言标准使用术语”源文件“来指明与程序相关的所有文件，包括.c 文件、.h 文件和其他文件等。

44.2 Header File

在把程序按功能划分为多个源文件后，接下来通过 #include 指令来解决随之产生的问题。

- 函数调用
  某个源文件中的函数如何调用定义其他文件中的函数？
- 变量访问
  某个源文件中的函数如何访问其他源文件中的外部变量？
- 共享宏和类型定义
  两个源文件如何共享同一个宏定义或类型定义？

#include 指令的引入使得在任意数量的源文件中共享信息成为可能，这些信息可以是函数原型、宏定义和类型定义等。

#include 指令指示预处理器打开指定的文件，并且把其中的内容插入到当前位置。如果是多个源文件需要访问相同的信息，那么就可以把该信息放入文件中，然后利用 #include 指令把需要的内容插入每个源文件中。

#include 指令包含的文件称为头文件(或者包含文件)。按照惯例，头文件的扩展名是.h。

在开发程序时，通常需要多个头文件来包含不同用途的函数组和类型定义等。如果使用单独一个大的头文件来包含所有的声明不可避免的会产生无用的信息，而且对其进行修改将导致在重新构建程序时需要对所有源文件重新进行编译。

44.3 Including Files

#include 指令有两种形式。

- #include < 文件名 >
  尖括号用于引用 C 语言自身库的头文件等。C 语言标准不要求在尖括号中的名字是文件，从而留下了开放的可能。
- #include “文件名”
  双引号用于其他所有头文件，也包括任何自定义的文件。

编译器定位头文件的方式是两种格式间的细微差异，下面是大多数编译器遵循的规则：

- #include < 文件名 > 会搜索系统头文件所在的目录(或多个目录)。
  例如，在 UNIX-like 系统中，通常把系统头文件保存在目录/usr/include 中。
- #include “文件名” 会搜索当前目录，然后搜索系统头文件所在的目录(或多个目录)。
  通常可以改变搜索头文件的位置，例如使用-I 路径等命令行选项来达到自定义引入扩展库的目的。

在 #include 指令中的文件名可以含有辅助定位文件的信息(例如目录的路径或驱动器号)。

```
#include "C:\test\stack.h" /* DOS path */
#include "/test/stack.h" /* UNIX path */
```

预处理器不会把 #include 指令的双引号中的内容作为字符串字面量，通常最好的做法是在 #include 指令中不包含路径或驱动器的信息，这样方便把程序移植到其他硬件平台上或者移植到其他操作系统中。

在上述的 #include 指令示例中指定的驱动器或路径信息不可能是一直有效的，因此需要改写成与驱动器无关的，而且指定的目录应该与当前目录相关。

```
#include "../stack.h"
#include "..\include\stack.h"
#include <sys/stat.h>
```

使用 #include 指令包含源文件也是合法的，但是容易出现问题。例如，假设在 foo.c 中定义的函数 f 在 bar.c 和 baz.c 中都会需要，就可能需要把 foo.c 包含到 bar.c 和 baz.z 中。

```
#include "foo.c"
```

在接下来的编译过程中不会报错，但是在链接阶段会出现问题，因为链接器会发现函数 f 的两个目标代码的副本。

44.4 Shared Macros

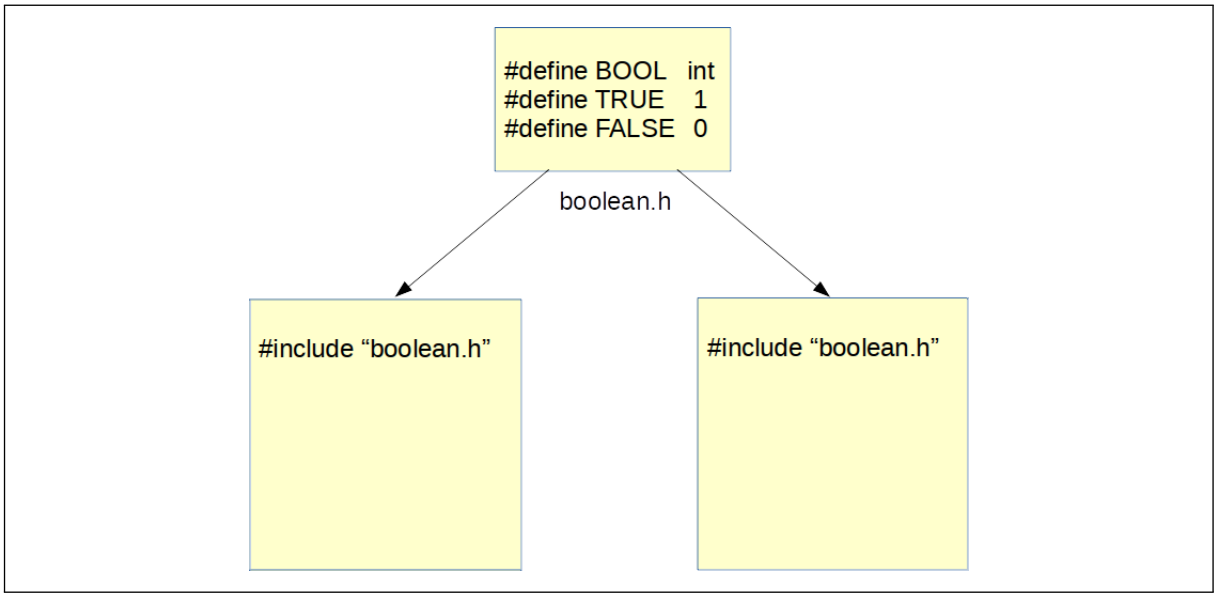
大多数大规模程序都包含用于某些（或者全部）源文件的共享的宏定义和类型定义，这些定义应该放在头文件中。

假设目前开发的程序使用自定义宏 BOOL、TRUE 和 FALSE，为了避免在每个需要的源文件中重复定义这些宏，可以把这些宏定义放在 boolean.h 中。

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

这样，任何需要这些宏的源文件只需通过 #include 指令引入 boolean.h 就可以使用自定义的宏。

```
#include "boolean.h"
```



44.5 Shared Type Definitions

类型定义在头文件中也很普遍。

例如，如果不使用 BOOL 宏，可以使用 typedef 指令来自己定义 BOOL 类型。

```
#define TRUE 1
#define FALSE 0
```

```
typedef int BOOL;
```

将宏定义和类型定义放入头文件中有很多好处。

- 节约了把定义复制给需要的源文件的时间。
- 方便了程序的修改。
  改变宏定义或类型定义只需要编辑单独的头文件，这样源文件中就自动会随之改变。
- 避免了源文件中包含相同的宏或定义的不同定义导致的冲突。

44.6 Shared Function Prototypes

在标准 C 中不允许调用未声明的函数，这样如果编译器没有函数运行可依赖，编译器将假定函数的返回值类型是 int 类型，而且还会假定形式参数的数量和函数调用时的实际参数数量是匹配的。

通过默认的实际参数提升，实际参数自身将自动转化为“标准格式”，这样的情况下编译器的假设也可能是错误的，从而导致程序无法执行，也没有任何作为原因的线索。

在标准 C 中，当调用定义在其他文件中的函数时，要始终确保编译器在调用之前得到函数的原型，为此可能需要在调用点所处的文件中对函数进行声明，不过这种方式很难保证函数的原型在所有文件中都相同。

为了保证函数原型与函数定义匹配，以及在函数发生改变后保证函数的一致性，可以把函数的原型放入头文件中并在调用函数的位置包含头文件。

假设函数 f 在源文件 foo.c 中定义，而且函数 f 可能被用于多个源文件，可以将函数 f 的原型放入头文件 foo.h 中。这样，通过将头文件 foo.h 也包含到 foo.c 中可以使编译器检查 foo.h 中函数 f 的原型与 foo.c 中函数 f 的定义是否匹配。

在含有函数 f 定义的源文件中应该始终包含声明函数 f 的头文件，否则将可能导致难以发现的错误。另外，仅用于源文件 foo.c 的函数不需要在头文件中进行声明，否则也会产生误解。

在逆波兰计算器的示例中，源文件 stack.c 中将包含与栈相关的函数：make_empty、is_empty、is_full、push 和 pop 等，这些函数的原型应该都放入头文件 stack.h 中。

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

这样，在 calc.c 中通过包含 stack.h 可以使编译器知道每个函数的返回类型，以及形式参数的数量和类型等。

在包含这些函数的定义的源文件 stack.c 中也包含 stack.h 以便于编译器检查 stack.h 中的函数原型是否和 stack.c 中的定义相匹配。

44.7 Shared Variables Declarations

在 C 语言中，变量可以在函数间共享，也可以在文件中共享。

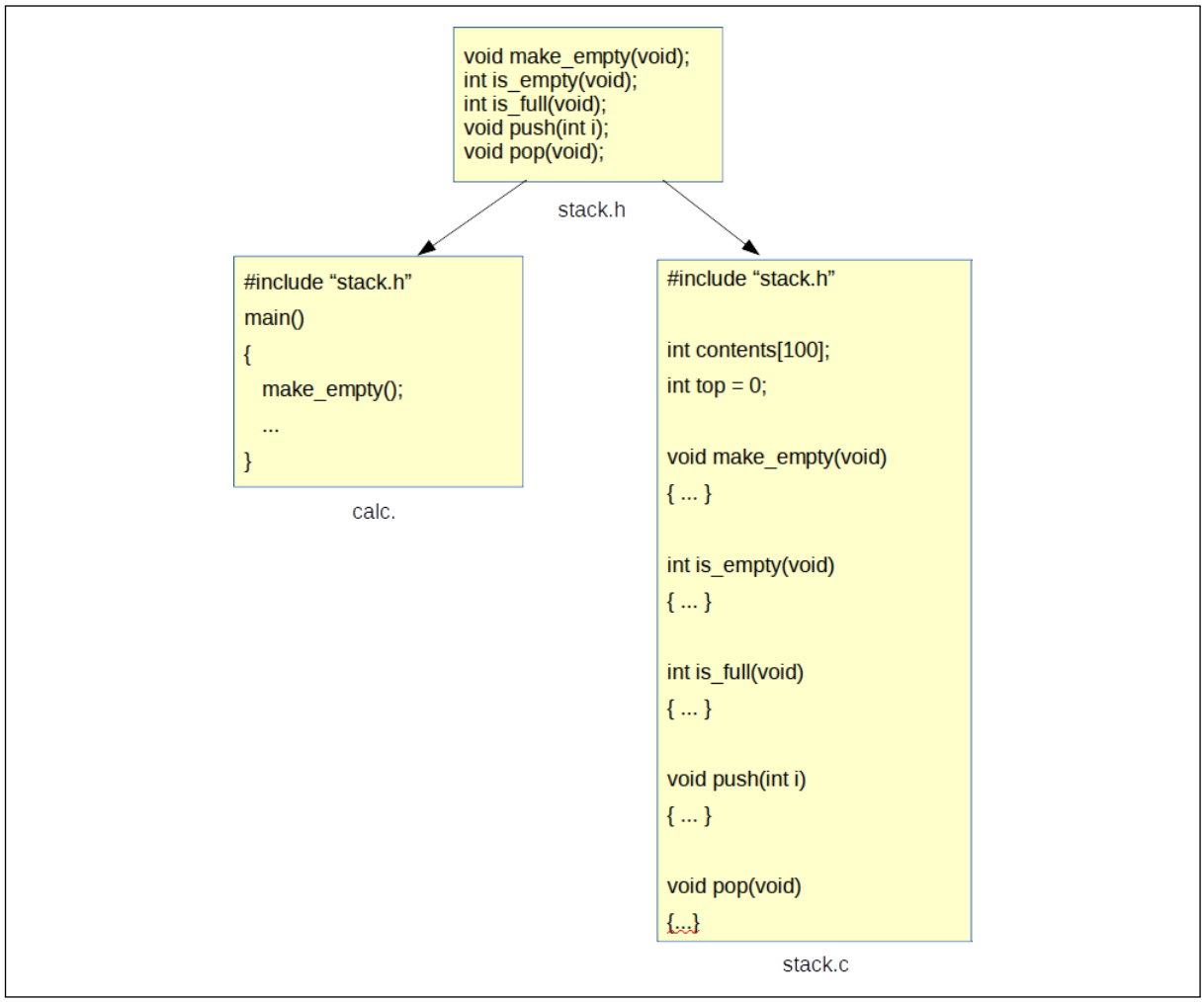
为了共享函数，可以把函数的定义放在源文件中，并在需要调用函数的其他文件中通过引入头文件来导入函数原型。C 语言中共享变量的方式与共享函数的方式非常类似。

- 对于不需要共享的变量，不需要区别变量的声明和定义。

  ```
  int i;
  ```

  这种情况下，变量的声明和定义在一起，编译器也会为变量预留空间。

  对于需要共享的变量，C 语言提供了 extern 关键字来说明变量的声明和定义是分开的。




```
extern int i;
```

通过 extern 关键字来告诉编译器变量是在程序中的其他位置定义的(大多数情况下指的是在不同的源文件中),因此不需要为变量分配空间。

extern 关键字可以用于所有类型的变量,如果在数组的声明中使用了 extern 关键字,说明编译器不用为数组分配空间,因此也就可以忽略数组的长度。

```
extern int a[];
```

为了在源文件中共享变量,首先需要把变量的定义放置在某个源文件中,而且可以对变量进行初始化,这样当编译该文件时,编译器会为该变量分配内存空间。

虽然数组和指针关系密切,但是在声明共享数组时要注意,指针不能被声明为共享指针,即类似下面的声明是错误的。

```
extern int *a;
```

在用于表达式时,数组会“衰退”成指针,例如将数组名作为函数调用中的实际参数的情况。但是,在变量声明中,数组和指针是两种不同的类型。

接下来,在其他的文件中通过使用 extern 关键字来声明变量后,就可以在这些文件中访问/或修改该变量,但是编译器在编译这些文件时不会为变量分配额外的内存空间。

当在文件中共享变量时,所有其他文件对变量的访问/或修改都是对同一个内存空间进行操作,因此会面临和共享函数时相似的问题:如何确保变量的所有声明和变量的定义一致。

当同一个变量的声明出现在不同文件中时,编译器无法检查变量声明是否和变量定义相匹配。例如,某一个文件中可以包含如下定义:

```
int i;
```

同时,在另一个文件可能包含不同的声明。

```
extern long int i;
```

这类错误可能导致程序的行为异常。

为了避免矛盾,通常把共享变量的声明放置在头文件中,需要访问特殊变量的源文件可以包含相应的头文件。此外,含有变量定义的源文件包含每一个含有变量声明的头文件,可以使编译器检查两者是否匹配。

值得注意的是,虽然在文件中共享变量是 C 语言的惯例,但是它有重大的缺点,因此在程序设计时应该考虑是否可以不需要共享变量。

44.8 Nested Includes

在头文件中可以包含 #include 指令,也就是头文件自身也可以包含头文件。

考虑下面的情况,在 stack.h 中的函数 is_empty 和 is_full 只能返回 0 或 1,那么声明它们的返回类型是 BOOL 类型是一个更好的设计。

```
#include "boolean.h"

BOOL is_empty(void);
BOOL is_full(void);
```

这里,通过在 stack.h 中包含 boolean.h 可以确保编译时 BOOL 类型的定义是有效的。

C 语言的早期版本根本不允许嵌套包含,而且在传统用法上也不推荐使用嵌套包含,不过嵌套包含在 C++ 语言中得到了普遍应用。

44.9 Protect Header File

如果在源文件中多次包含同一个头文件,那么可能会产生编译错误的结果,而且当头文件又包含其他头文件时问题会更普遍。

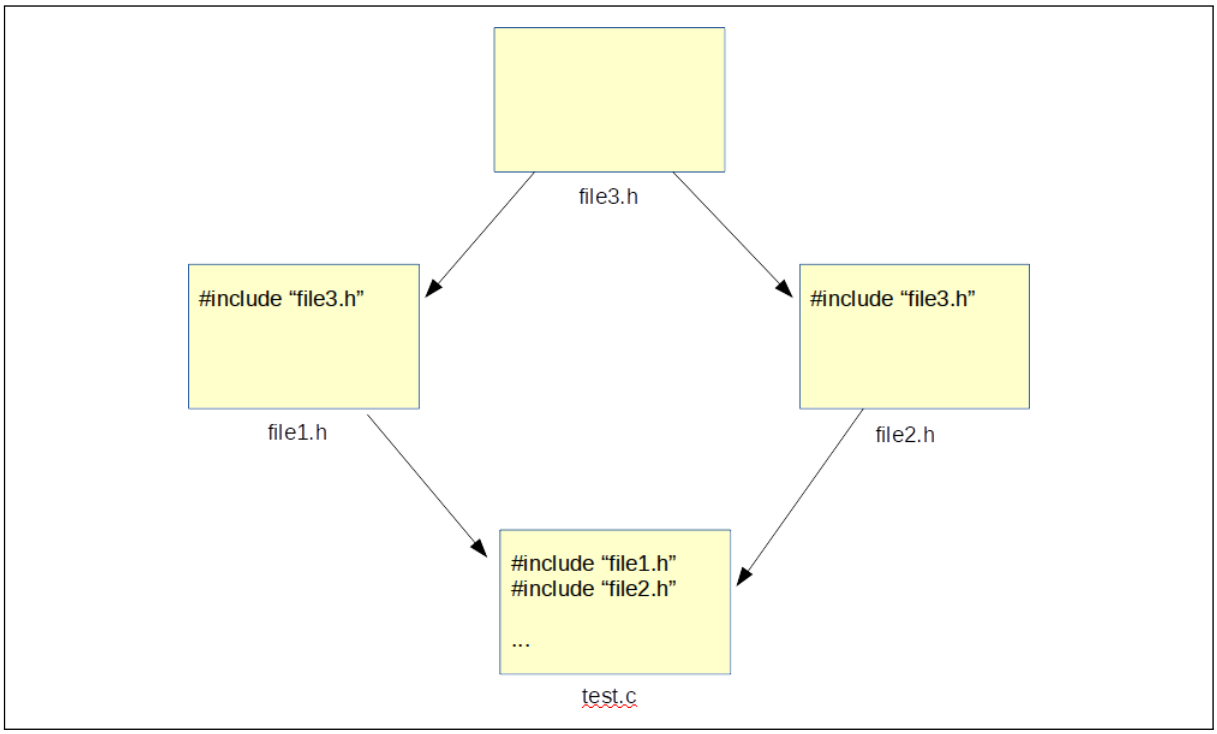
例如,在下面的示例中,file1.h 包含 file3.h,file2.h 包含 file3.h,而 test.c 同时包含 file1.h 和 file2.h,在编译 test.c 时就会编译两次 file3.h。

多次包含同一个头文件不会总是导致编译错误,如果文件只包含宏定义、函数原型和/或变量声明,那么不会影响编译结果。如果头文件中包含类型定义,则会产生编译错误。

另外,在源文件中包含了不是真正需要的头文件时,并不会对程序造成损害(可能发生的最坏的情况是在编译源文件时镜像增加),除非头文件的声明或定义与源文件产生冲突,这要归功于 C 语言对于头文件的保护机制。

保护头文件可以规避相同的类型定义带来的问题,而且节约编译时间。对此,C 语言引入了 #ifndef 和 #endif 两个指令来闭合文件内容。例如,为了保护 boolean.h 中的类型定义,可以使用 #ifndef 和 #endif 来改写 boolean.h。

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0

typedef int BOOL;

#endif
```



这样，在第一次包含 boolean.h 时，将不定义宏 BOOLEAN_H，所以预处理器允许保留在 #ifndef 和 #endif 之间的内容。

如果再次包含 boolean.h，那么预处理器将把 #ifndef 和 #endif 之间的内容删除，从而保护了头文件 boolean.h。

这里，给宏(BOOLEAN_H)取类似于头文件的名字是避免和其他的宏产生冲突。

44.10 Error Report

#error 指令可以用于检查头文件的包含条件等。例如，如果某一头文件包含只能在 DOS 环境下运行的函数原型，为了保证只能在 DOS 程序中包含它，在头文件中可以包含 #ifndef(或 #if)指令来检查宏。

```
#ifndef DOS
#error Function prototypes only support DOS.
#endif
```

这样，如果非 DOS 的程序试图包含该头文件时，编译过程将在 #error 指令处停止。

44.11 Multi-files

在程序设计阶段需要确定程序中的函数、外部变量以及如何把函数逻辑化的分布在相关的组中，接下来就需要根据头文件和源文件的规则来把程序划分为多个文件。

首先，把每组函数集合放入单独的源文件中(例如 foo.c)并创建与源文件同名的头文件(例如 foo.h)，这样就可以在头文件中放置函数的原型，在源文件中放置函数的定义等[1]。

接下来，每个需要调用函数的源文件将相应的头文件包含进来，而且包含函数定义的源文件应该包含对应函数原型的头文件(即包含自身的头文件)，这样可以使编译器检查头文件中的函数原型与函数定义是否一致。

[1]注意，在头文件中不需要也不应该声明只为内部使用而设计的函数原型。

一般情况下，main 函数所在的源文件的名字应该与程序的名字匹配。例如，如果希望程序名字为 bar，那么 main 函数应该位于 bar.c 中。另外，程序中其他源文件不调用的函数，都应该和 main 函数放置在同一个源文件中。

在下面的示例中，将实现用于文件格式化的示例程序 fmt，fmt 程序读入文本文件并产生格式化的输出(输出到屏幕上或者是通过重定向输出到特定文件中)。

```
fmt <quote
```

或

```
format <quote >newquote
```

通常情况下，除了额外的空格和删除的空行，以及做过填充[2]和调整的行，示例程序 fmt 的输出应该和输入一样。

首先，假设没有单词的长度超过 20 个字符(把与单词相邻的标点符号看成是单词的一部分)。如果程序遇到较长的单词，它需要忽略前 20 个字符后的所有字符，用一个单独的星号替换它们。例如，单词

```
abcdefghijklmnopqrstuvwxyz
```

将会格式化为

```
abcdefghijklmnopqrst*
```

示例程序不能一个一个地写单词，而应该把输入存储到一个“行缓冲区”中，直到有足够的空间填满一行，从而得到程序的核心。

```
for(; ;){
   读单词;
   if(不能读单词){
      不用调整地写行缓冲区的内容;
      终止程序;
   }

   if(单词不适合在行缓冲区中){
      调整写缓冲区中的内容;
      清除行缓冲区;
   }

   往行缓冲区中添加单词;
}
```

fmt 示例程序需要函数处理单词，并且需要函数处理行缓冲区，因此有必要将源代码划分为 3 个文件。

- 把所有和单词相关的函数放在一个文件(word.c)中。
- 把所有和行缓冲区相关的函数放在另一个文件(line.c)中。
- 把 main 函数放在主文件 fmt.c 中。

另外，还需要两个头文件 word.h 和 line.h 来包含相关的函数原型。

接下来，通过检查主循环可以发现只和单词相关的函数是 read_word。(如果 read_word 函数读入到输入文件的末尾，那么将通过读取“空”单词的方式来给主循环发出信号。)由此得到头文件 word.h。

```
/* word.h */

#ifndef WORD_H
#define WORD_H
```

[2]“填充”行意味着添加单词直到再多一个单词就会导致行溢出时才停止。“调整”意味着在单词间添加额外的空格以便于每行有精确的相同长度，必须进行调整才能使一行内单词间的空格相等(或者几乎是相等的)。对输出的最后一行将不进行调整。

```
/*********************************************************
 * read_word: Reads the next word from the input and *
 *         stores it in word. Makes word empty if no *
 *         word could be read because of end-of-file. *
 *         Truncates the word if its length exceeds *
 *         len.                                *
 *********************************************************/
void read_word(char *word, int len);

#endif
```

这里,宏 WORD_H 是用于保护头文件的。

继续分析主循环,可以将其划分为执行单词操作的函数,并且在 line.h 中包含对应的函数原型。

- flush_line:非格式化的写行缓冲区的内容。
- space_remaining:检查单词是否适合在缓冲区中。
- write_line:格式化的写行缓冲区的内容。
- clear_line:清空行缓冲区。
- add_word:向行缓冲区中添加单词。

```
/* line.h */

#ifndef LINE_H
#define LINE_H

/*********************************************************
 * clear_line: Clears the current line.     *
 *********************************************************/
void clear_line(void);

/*********************************************************
 * add_word: Adds word to the end of the current line. *
 *        If this is not the first word on the line, *
 *        puts one space before word.       *
 *********************************************************/
void add_word(const char *word);

/*********************************************************
 * space_remaining: Returns the number of characters left *
 *             in the current line.        *
 *********************************************************/
int space_remaining(void);

/*********************************************************
 * write_line: Writes the current line with *
 *         justification.                   *
 *********************************************************/
void write_line(void);

/*********************************************************
 * flush_line: Writes the current line without *
 *         justification. If the line is empty, does *
 *         nothing.                         *
 *********************************************************/
void flush_line(void);

#endif
```

在实现示例程序的功能时，首先包含相关的头文件，然后使用在头文件中声明的函数来实现主程序——这里是实现处理单词的主循环。

```
/* fmt.c */
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

main()
{
  char word[MAX_WORD_LEN+2];
  int word_len;


  clear_line();
  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

当在 main 函数中调用 read_word 函数时，main 函数告诉 read_word 函数截断任何超过 21 个字符的单词。当 read_word 函数返回后，main 函数检查 word 包含的字符串的长度是否超过了 20 个字符。如果超过了，那么读入的单词必须至少有 21 个字符长(在截断前)，然后 main 函数使用星号来替换第 21 个字符。

在编写头文件对应的源文件时，可以在其中包含额外需要的函数。例如，如果添加一个小的“内部”函数 read_char 就可以更容易地实现 read_word 函数。

在 word.c 中，read_char 函数的用途只是每次读一个字符，并且把遇到的换行符或制表符转换为空格。

```
/* word.c */

#include <stdio.h>
#include "word.h"

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';
```

```
  return ch;
}

void read_word(char *word, int len)
{
  int ch, pos = 0;
  while ((ch = read_char()) == ' ')
    ;

  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }

  word[pos] = '\0';
}
```

自定义的 read_char 和 getchar 函数相比有一定的不同，在 read_char 函数中把变量 ch 声明为 int 类型，而且 getchar 函数实际上返回的是 int 型值而不是 char 型值。当不能连续读入时（通常是因为读入到了输入文件的末尾），getchar 函数返回 EOF。

read_word 函数的实现由两个 while 循环组成。

- 首先，在第一个循环中会跳过空格，在遇到第一个非空字符时停止（EOF 不是空的，如果到达了输入文件的末尾，循环停止）。
- 然后，在接下来的第二个循环读字符直到遇到空格或 EOF 时停止，循环体把字符存储到 word 中直到达到 len 的限制时停止。

第二个循环中的 if 判断表达式说明，在单词的长度大于 len 时，循环将继续读入字符，但是不再存储这些字符。

read_word 函数继续执行，并以空字符结束单词来构造字符串。如果 read_word 函数在找到非空字符前遇到 EOF，pos 将在末尾置为 0，从而使得 word 成为空字符串。

在 line.c 中提供了头文件 line.h 中声明的函数的定义，并且使用变量来跟踪行缓冲区的状态。

- 变量 line 用于存储当前行的字符。
- 变量 line_len 用于存储当前行的字符数量。
- 变量 num_words 用于存储当前行的单词数量。

```
/* line.c */

#include <stdio.h>
#include <string.h>
#include "line.h"

#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}

void add_word(const char *word)
```

```
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}

void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}

void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

为了格式化的输出每一行内容，write_line 函数在 line 中一个一个地写字符，当需要添加额外的空格时就在每对单词之间停顿。

额外空格的数量存储在变量 spaces_to_insert 中，该变量的值由 extra_spaces / (num_words - 1) 得到，其中 extra_spaces 的初始值是最大行长度和当前行长度的差。

在输出每个单词之后 extra_spaces 和 num_words 都会发生变化，因此 spaces_to_insert 也将变化。例如，如果 extra_spaces 的初始值为 10，num_words 的初始值为 5，那么将有 2 个额外的空格跟着第 1 个单词，有 2 个额外的空格跟着第 2 个单词，有 3 个额外的空格跟着第 3 个单词，以及有 3 个额外的空格跟着第 4 个单词。

44.12 Building Program

构建大规模程序和构建单源文件程序具有相同的基本步骤。

1. 编译

构建程序时必须对程序中的每个源文件单独进行编译，当包含头文件的源文件被编译时会自动编译头文件，因此不需要单独编译头文件。

编译后产生的目标文件（object file）包含来自每个源文件的目标代码，扩展名为.o（UNIX-like 系统）或.obj（DOS 系统）。

2. 链接

链接器把目标代码和库函数的目标代码链接到一起并生成可执行程序。

另外，链接器还需要负责解决编译器遗留的外部引用问题，外部引用问题一般发生在某个文件中的函数调用其他文件中定义的函数，或者访问其他文件中定义的变量的情况中。

大多数编译器支持一次性完成程序构建过程。例如，对于 UNIX-like 系统中的 GCC 编译器，可以使用下面的命令来构建 fmt 示例程序。

```
$ gcc -o fmt fmt.c line.c word.c
```

首先，GCC 编译器把 3 个源文件编译成目标代码，并且分别命名为 fmt.o、line.o 和 word.o。

然后，GCC 编译器会自动把目标文件发送给链接器，连接器把这些文件结合成一个单独的文件（编译器选项-o 用来指明可执行文件的名字）。

44.12.1 Makefile

UNIX 系统中引入了 makefile 的概念来提高构建大型软件的效率。

makefile 文件包含了构建程序所需的必要信息，其中不仅列出了作为程序部分的文件，而且还说明了文件之间的依赖关系。例如，如果文件 foo.c 中包含头文件 bar.h，那么就认为 foo.c“依赖于”bar.h，bar.h 的变化将导致重新编译 foo.c。

下面是针对 fmt 示例程序的 makefile。

```
fmt: fmt.o line.o word.o
  gcc -o fmt fmt.o line.o word.o
fmt.o: fmt.c line.h word.h
  gcc -c fmt.c
word.o: word.c word.h
  gcc -c word.c
line.o: line.c line.h
  gcc -c line.c
```

在 makefile 中，每一组的第一行给出了目标文件及其依赖的文件，这样当其依赖的某个文件发生变化时就需要重新编译目标文件。

每一组的第二行是用来执行的编译命令。

在其他系统中的 makefile 命令与 UNIX-like 系统是类似的。例如，如果使用 bcc 编译器，makefile 文件只需要很小程度的修改（不再编译选项-o 是因为第一个目标文件的名字就可以确定可执行文件的名字）。

```
fmt.exe: fmt.obj line.obj word.obj
  bcc fmt.obj line.obj word.obj
fmt.obj: fmt.c line.h word.h
  bcc -c fmt.c
word.obj: word.c word.h
  bcc -c word.c
line.obj: line.c line.h
  bcc -c line.c
```

make 工具可以根据 makefile 来构建（或重新构建）程序，make 工具会检查与说明的每个文件相关的时间和日期来确定文件是否过期，然后在构建程序时它会自动唤醒编译器和链接器等。

一般情况下，makefile 中的内容是冗余的，可以结合 bash 等工具进行精简。

另外，集成开发环境的出现引入了“工程文件”的概念用于替代 makefile。

44.12.2 Linking

在 C 语言中，编译和链接是完全独立的，头文件的存在是为了给编译器而不是给链接器提供信息。如果需要调用 foo.c 中的函数 f，那么需要确保对 foo.c 进行了编译，而且还要确保通过头文件通知链接器必须搜索 foo.c 的目标文件。

在任何情况下，大多数链接器都只会链接程序实际需要的函数。

在链接期间，可能会报告一些在编译期间无法发现的错误。例如，如果程序中缺少函数定义或变量定义，链接器就无法解决外部引用，从而会出现类似“Undefined symbol”或“Unresolved external reference”等信息。

下面是链接期间经常出现的错误。

- 拼写错误
  如果变量名或函数名拼写错误，那么链接器将作为丢失来进行报告。
- 文件丢失
  如果链接器无法找到某个文件中的函数，那么可能会报告文件丢失。
- 库丢失
  如果链接器无法找到库函数就会报告函数库丢失。例如，在 UNIX-like 系统中的链接阶段无法找到数学函数时，需要使用命令行选项-lm 才能解决。

44.13 Rebuilding

在开发期间，一般极少需要编译全部文件。大多数情况下，仅仅是测试程序并进行新特性的开发和修改，然后再重新构建程序。

为了节约时间，在重新构建的过程中，只是对那些可能受到最后一次变化影响的文件重新进行编译。

- 如果仅影响到某个源文件，那么在重新构建过程中只有该文件会被重新编译，然后链接器将重新链接并生成程序。
  例如，如果对上述的 word.c 中函数 read_char 进行精简，那么重新构建程序时将会重新编译 word.c 文件，但是其他源文件不会被重新编译。

```
int read_char(void)
{
  int ch = getchar();
  return (ch == '\n' || ch == '\t' ? ' ' : ch);
}
```

- 如果仅影响到某个头文件，那么将会重新编译包含该头文件的源文件。
  包含某个被修改了的头文件的源文件可能会潜在的受到影响，因此保守的做法是将包含该头文件的源文件都重新编译。
  例如，如果对 read_word 函数的原型和实现进行修改，那么重新构建程序时就需要对包含头文件的相关源文件进行重新编译。

```
/* word.h */

#ifndef WORD_H
#define WORD_H

/*************************************************************
 * read_word: Reads the next word from the input and *
 *         stores it in word. Makes word empty if no *
 *         word could be read because of end-of-file. *
 *         Truncates the word if its length exceeds *
```

```
 *        len.                                  *
 ************************************************************/

/************************************************************
 * modify: Returns the number of characters stored. *
 ************************************************************/
int read_word(char *word, int len);

#endif
```

在对头文件进行修改后,需要修改包含其对应的函数定义的源文件。

```
/* word.c */

#include <stdio.h>
#include "word.h"

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';

  return ch;
}

/* void -> int */
int read_word(char *word, int len)
{
  int ch, pos = 0;
  while ((ch = read_char()) == ' ')
    ;

  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }

  word[pos] = '\0';

  /* void -> int */
  return pos;
}
```

头文件和源文件的修改也影响到了主程序的实现,因此需要对主程序的源文件进行修改。

```
/* fmt.c */
/* Formats a file of text */

#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

main()
{
  char word[MAX_WORD_LEN+2];
  int word_len;
```

```
    clear_line();
    for (;;) {
      word_len = read_word(word, MAX_WORD_LEN + 1);
      if (word_len == 0) {
        flush_line();
        return 0;
      }
      if (word_len > MAX_WORD_LEN)
        word[MAX_WORD_LEN] = '*';
      if (word_len + 1 > space_remaining()) {
        write_line();
        clear_line();
      }
      add_word(word);
    }
  }
```

在 UNIX-like 系统中重新构建 fmt 示例程序时，只需要重新编译受影响的源文件，没有受到影响的源文件不需要重新编译，然后对目标文件重新进行链接。

```
$ gcc -o fmt fmt.c word.c line.o
```

makefile 可以用来自动进行重新构建，make 工具通过检查每个文件的最后修改日期和时间来确定从程序的最后一次构建后哪些文件发生了变化，然后把发生变化的文件和直接或间接依赖于它们的全部文件一起重新编译。

44.14 Optional Macros

在编译程序时，C 语言编译器提供了一些指定宏的值的做法，从而使得可以不需要编辑任何文件就可以对宏的值进行修改，这对于利用 make 自动构建程序有很大的帮助。

大多数 UNIX 编译器（或某些非 UNIX 编译器）支持选项-D，可以用于在命令行中指定宏的值。

```
$ gcc -DDEBUG=1 foo.c
```

在上述的示例中，编译时设置在 foo.c 中定义的宏 DEBUG 的值为 1，这类似于下面的宏定义：

```
#define DEBUG 1
```

这样，如果选项-D 定义的宏在没有指定的值时，使用-D 选项可以设置其为 1。

某些编译器也支持-U 选项，可以用来取消宏，使用效果和 #undef 相同。

```
# gcc -UDEBUG foo.c
```

第 45 章

Large Program

45.1 Overview

以前的计算机无法运行大型的程序，不过现在更快的 CPU 和更大的主存已经允许我们编写一些以前完全不可行的程序。图形界面的流行同样大大增加了程序的平均长度，例如现在大多数功能完整的程序至少都有 100 000 行代码。

C 语言并不是专门用来编写大规模程序的，起码在 C 语言在设计之初不是为了大型程序设计的。但是，后来许多大规模程序的确是使用 C 语言编写的，这导致程序设计和开发都很复杂，而且需要更多的耐心和细心。

在 1970 年代和 1980 年代的软件危机中，许多软件最后都得到了一个悲惨的结局，软件项目开发时间大大超出了规划的时间表。人们开始意识到编写一个大型程序和编写一个小型程序有很大的不同。

大型程序一般需要多人协同开发，更加需要注意编写风格。另外，大型程序需要有仔细的文档，在后续还需要对程序进行多次修改，因此需要对维护进行细致的规划。

在 1986 年，IBM 大型电脑之父佛瑞德·布鲁克斯发表了他的著名论文《没有银弹》，在这篇著名的论文中他断言："在 10 年内无法找到解决软件危机的银弹"。从软件危机被提出以来，人们一直在寻找解决它的方法，并且有一系列的方法被提出并且加以应用（比如结构化程序设计、面向对象的开发、CMM、UML 等）。

相对于小型程序，编写一个大型程序需要更仔细的设计和更详细的计划，从而使得程序更易读，也更易于维护。在程序设计领域为了解决如何在软硬件环境逐渐复杂的情况下使软件得到良好的维护的问题，引入了面向对象程序设计。

面向对象程序设计在某种程度上通过强调可重用性解决了大型软件开发中的很多问题。通过面向对象设计，我们将真实世界的对象映射到抽象的对象，随之引入了"类"的概念，还应用了实例这一思想。

20 世纪 70 年代施乐 PARC 研究所发明的 Smalltalk 语言将面向对象程序设计的概念定义为，在基础运算中对对象和消息的广泛应用。Smalltalk 的创建者深受 Simula 67 的主要思想影响，但 Smalltalk 中的对象是完全动态的——它们可以被创建、修改并销毁，这与 Simula 中的静态对象有所区别。

此外，Smalltalk 还引入了继承性的思想，它因此一举超越了不可创建实例的程序设计模型和不具备继承性的 Simula。

面向对象程序设计在 80 年代成为了一种主导思想，这主要应归功于 C++——C 语言的扩充版。C++ 语言包含了 C 语言的全部特性，但是 C++ 语言比 C 语言要复杂的多。

实际上，C++ 语言在继承了 C 语言的全部特性的同时，也增加了大量新特性。另外，C++ 语言的不同功能之间的多种组合也进一步增加了语言的复杂程度，因此 C 语言更适合编写简单程序。

C++ 语言所提供的新特性需要编译器做更多的工作，因此 C++ 程序编译速度可能比 C 语言程序慢，而且这些新特性对程序的运行性能也有一定的影响，这种影响对于某些程序来说可能是不可接受的。

对于简单程序以及移植性更高的程序而言，C 语言更适合。对于大型的、功能齐全的程序，尤其是在图形用户界面（GUI）日渐崛起的情况下，以 C++ 语言为代表的面向对象编程语言很好地适应了潮

流,而且 GUI 和面向对象程序设计的紧密关联在 Mac OS X 中可见一斑。现在,面向对象的编程语言必须具有下述的能力已经得到了广泛的共识。

- 封装(Encapsulation)—能够定义新的类型以及一组对这个类型的操作的能力,并且不会暴露类型的具体实现。
  在“面向对象”的环境中,类型的值就是“对象”。C++ 语言通过限制对私有数据成员的访问来支持封装。
- 继承(Inheritance)—能够定义新的类型,并且在新的类型中继承已经存在的类型的属性的能力。
  C++ 语言通过派生类来实现继承。
- 多态(polymorphism)—对于同样的操作,对象能够根据其所属的类来采取不同的响应。
  C++ 语言通过虚函数来支持多态性。

面向对象程序设计的思想也使事件处理式的程序设计更加广泛被应用(虽然这一概念并非仅存在于面向对象程序设计)。在过去的几年中,Java 语言成为了广为应用的语言。除了 Java 与 C 和 C++ 语法上的近似性,Java 的可移植性是它的成功中不可磨灭的一步。

虽然 C++ 语言解决了 C 语言的一些隐患,但是仍然有一些没有涉及,而且 C++ 语言的新特性也引入了一些新的陷阱。在最近的计算机语言发展中,一些既支持面向对象程序设计,又支持面向过程程序设计的语言逐渐发布,它们中的佼佼者有 Python、Ruby 等。

正如面向过程程序设计使得结构化程序设计的技术得以提升,现代的面向对象程序设计方法使得对设计模式的用途、契约式设计和建模语言(如 UML)技术也得到了一定提升。

45.2 Module

当设计一个 C 语言程序(或其他任何语言的程序)时,最好将它看作是一些独立的模块。

模块是一组功能(服务)的集合,其中一些功能可以被程序的其他部分(称为客户)使用。每个模块都有一个接口来描述所提供的功能。

模块的细节(包括模块功能自身的源代码等)都包含在模块的实现中。

在 C 语言环境下,上述提到的这些“功能”就是函数,模块的接口就是头文件(.h),头文件中包含那些可以被程序中其他文件调用的函数的原型,而模块的实现就是包含该模块中函数的定义的源文件(.c)。

以一个作为示例的计算器程序为例,该程序由 calc.c 文件和一个栈模块组成。其中,calc.c 文件包含 main 函数,而栈模块则存储在 stack.h 和 stack.c 中。

文件 calc.c 就是栈模块的客户,而 stack.h 则是栈模块的入口,stack.c 是栈模块的实现(里面包含了操作栈的函数的定义以及组成栈的变量的定义)。

C 函数库本身就是一些模块的集合,库中的每个头文件都会作为一个模块的接口,例如 stdio.h 就是包含字符串处理函数的模块的接口。

将程序分割为模块,就可以将其抽象成一组相互服务的模块,提高模块的可复用性和可维护性。

- 抽象
  合理设计的模块可以被看作抽象,从而不需要了解模块的功能的实现细节。
  通过划分清晰的抽象层次,可以不必为了修改部分程序而了解整个程序,这样可以使团队成员更容易地协同工作来实现程序。
  对模块的接口达成一致后,就可以将实现每一个模块的工作分派给不同的成员,从而可以给每个成员更大程度的独立性。
- 可复用性
  如果模块设计合理,功能划分清楚,那么每一个提供一定功能的模块就都有可能在其他程序中复用。
  模块在不同程序中的应用场景是很难预测的,因此最好将模块设计成可复用的。
- 可维护性

将程序划分为模块后,程序中的错误通常就只会影响一个模块,这样就更容易找到并处理错误。另外,清除错误后再次编译程序时,就只需要重新编译受到影响的模块,然后重新链接整个程序。

对于高可维护性的程序,当为了提高性能或进行跨平台移植时,接口不变甚至允许开发者替换一个完整的模块的实现。

现实中的程序的可维护性是最重要的,在程序的生命周期中可能会不断的发现问题或者增加特性来适应更新的需求。将程序按照模块来设计和开发会使维护工作更容易,例如维护一辆汽车时,修理轮胎应该不需要同时检修引擎。

在一个管理零件的程序中,最初的程序可能将零件记录在一个数组中。在程序使用了一段时间后,为了突破存储的零件数量的上限,可能需要考虑将存储方式修改为链表,这时就需要仔细检查整个程序并找到所有依赖零件存储方式的地方。

如果在开始设计程序时采用了不同的方式——使用一个独立的模块来处理零件的存储,就可能只需要重写该模块来修改程序,而不需要检查整个程序。

在进行模块化程序设计时,需要考虑的问题包括需要定义的模块的种类、模块的功能和模块之间的相互关系等。

45.2.1 Module Relationship

一个好的模块接口并不是随意的一组声明。对于一个具有高可用性的模块,应该至少具有下面的两个性质:

- 高内聚性(High cohesion)
  模块中的元素应该相互紧密相关,并且为了同一目标而相互合作。高内聚性在使模块更易于使用的同时,也使程序更容易理解。
- 低耦合性(Low coupling)
  模块之间应该尽可能相互独立。低耦合性可以使程序更便于修改,也方便以后进行模块复用。

例如,实现栈的模块是具有内聚性的,其功能是实现与栈相关的操作。主程序通过包含头文件来引用栈模块,以及栈模块的实现与头文件的包含关系都是低耦合性的表现。

45.2.2 Module Types

需要具备高内聚性和低耦合性的模块通常被划分为如下几类。

- 数据池
  数据池是一些相关的变量或常量的集合。在 C 语言中,这类模块通常只是一个头文件 (例如 float.h 和 limits.h 等)。
  从程序设计的角度来看,通常不建议将变量包含在头文件中。
- 库
  库是一组相关函数的集合,例如 string.h 就是字符串处理函数库的接口。
- 抽象对象
  在这里,“对象”是一组数据以及针对数据的操作的集合。如果数据是隐藏起来的,那么这个对象就是“抽象”的,因此抽象对象是指对于隐藏的数据结构进行操作的一组函数的集合。
- 抽象数据类型
  将具体实现方式隐藏起来的数据类型称为抽象数据类型。
  作为客户的模块可以使用抽象数据类型来声明变量,但无法知道这些变量的具体数据结构。如果客户需要对变量进行操作,则必须调用抽象数据类型所提供的函数。

45.2.3 Information Hiding

设计良好的模块需要对它们的客户隐藏一些信息，例如栈模块的客户无需了解栈是如何实现的(可能是用数组实现的，也可能是用链表或者其他方式实现的)。

通过模块来对客户隐藏信息的方式称为信息隐藏，这可以为模块带来安全性和灵活性。

例如，客户在不了解栈的存储方式时，就无法通过栈的内部机制来修改栈的数据。模块自身提供的函数都是经过测试的，从而可以保护栈的数据。

在接口固定的情况下，对模块的内部机制进行改动时不会对客户产生影响。例如，只要栈模块是按照正确的方式设计的，那么不管是用数组来实现栈还是用链表或其他方式来实现栈，都不需要改变栈模块的接口。

C 语言提供的可以用于强制信息隐藏的主要工具是 static 存储类型。例如，将一个函数声明为 static 类型可以使函数内部链接，从而阻止其他文件(包括模块的客户)调用这个函数。

将一个带文件作用域的变量声明为 static 类型可以达到类似的效果，从而使变量只能被同一文件中的其他函数访问。

为了说明信息隐藏，下面引入栈模块的两种实现，它们都具有同样的头文件。

```
/* stack.h */
#ifndef STACK_H
#define STACK_H

void make_empty(void);
int is_empty(void);
void push(int i);
int pop(void);

#endif
```

这里，stack_full 的原型没有包含进 stack.h 中，原因是 stack_full 函数在使用数组存储栈时是有意义的，但是在使用链表来存储栈时就没有意义了。

首先，使用数组来实现栈。

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

void make_empty(void)
{
  top = 0;
}

int is_empty(void)
{
  return top == 0;
}

static int is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
```

```
{
  if (is_full()) {
    printf("Error in push: stack is full.\n");
    exit(EXIT_FAILURE);
  }
  contents[top++] = i;
}

int pop(void)
{
  if (is_empty()) {
    printf("Error in pop: stack is empty.\n");
    exit(EXIT_FAILURE);
  }
  return contents[--top];
}
```

在栈模块的数组实现中，用于实现栈的变量（contents 和 top）都被声明为 static 类型，这样就阻止了程序的其他部分直接访问它们。

这里，is_full 函数也被声明为 static 类型，从而可以通过信息隐藏来屏蔽程序的其他部分对 is_full 函数的访问。

宏可以用来指明函数和变量是“公有的”（即可以被程序的其他部分访问）还是“私有的”（即仅限当前文件内访问）。

```
#define PUBLIC /* empty */
#define PRIVATE static
```

通过宏来将 static 写成 PRIVATE 可以更清晰地指明信息隐藏。

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PUBLIC void make_empty(void){...}

PUBLIC int is_empty(void){...}

PRIVATE int is_full(void){...}

PUBLIC void push(int i){...}

PUBLIC int pop(void){...}
```

下面的示例代码使用链表来实现栈。

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;

void make_empty(void)
{
  top = NULL;
}
```

```
int is_empty(void)
{
  return top == NULL;
}

void push(int i)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error in push: stack is full.\n");
    exit(EXIT_FAILURE);
  }

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}

int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty()) {
    printf("Error in pop: stack is empty.\n");
    exit(EXIT_FAILURE);
  }

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```

使用链表来实现栈模块时，栈可以没有固定大小，因此就不再需要 is_full 函数了。

push 函数需要测试 malloc 函数是否返回空指针。如果 malloc 函数返回空指针，则说明没有足够的内存来压入下一个元素。

is_full 函数在栈模块的实现中被声明为 static 类型，因此其他文件就无法调用 is_full 函数，而且它们也无法知道是否有 is_full 函数。

通过信息隐藏，模块的实现方式无关紧要。只要具有同样的接口定义，模块的不同实现版本之间可以相互替换而不会影响程序的其他部分。

45.2.4 Abstract Type

抽象对象的模块不可能对同一对象有多个示例，因此需要进一步创建一个新的类型。

抽象数据类型的引入使得通过定义 Stack 类型来实现任意多个栈。例如，在下面的示例代码中演示了如何在同一个程序中有两个栈。

```
#include <stdio.h>
#include "stack.h"

main()
```

```
{
  Stack s1, s2;

  make_empty(&s1);
  make_empty(&s2);
  push(&s1, 1);
  push(&s2, 2);

  if(!is_empty(&s1))
    printf("%d\n", pop(&s1)); /* prints 1 */
  ...
}
```

对于栈模块的客户，s1 和 s2 是抽象的对象，它只响应特定的操作（make_empty、is_empty、push 和 pop 等）。

接下来在 stack.h 中增加 Stack 类型的定义，这需要给每个函数增加一个 Stack 类型（或 Stack *）的形式参数。

```
#define STACK_SIZE 100

typedef struct{
  int contents[STACK_SIZE];
  int top;
}Stack;

void make_empty(Stack *s);
int is_empty(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

作为函数 make_empty、push 和 pop 参数的栈变量需要定义为指针，从而允许函数来改变栈的内容。

is_empty 函数的参数并不需要定义为指针，而且给 is_empty 函数传递一个 Stack 指针比传递一个 Stack 值更有效。如果给 is_empty 函数传递一个 Stack 值，将导致整个数据结构被复制。

C 语言没有设计专门用于封装（Encapsulation）类型的特性，因此从技术上来说，C 语言库中是没有抽象数据类型的。虽然可以使用技巧来达到类似的目的，但是使用起来相当笨拙，无法解决依赖性。

C 语言只是提供了一些很接近抽象数据类型的类性，例如 FILE 类型。在对一个文件进行操作之前，必须声明一个 FILE * 类型的变量，并在随后把 FILE * 类型的变量传递给不同的文件处理函数。

```
FILE *fp;
```

FILE 可以认为是一个抽象类型，在使用时不需要知道 FILE 的具体实现，而且 C 语言标准并不能保证 FILE 的具体类型。实际上，最好不去关心 FILE 值的存储，因为 FILE 类型的定义对不同的编译器可能（也确实经常）是不一样的。

但是，如果进入 stdio.h 找到 FILE 的实现，就可以编写代码来访问 FILE 的内部机制。例如，如果发现 FILE 结构中使用 bsize 来说明文件的缓冲区大小，那么就可以直接访问特定文件的缓冲区大小。

```
typedef struct{
  ...
  int bsize; /* buffer size */
  ...
}FILE;
```

如果需要输出文件的缓冲区大小，可以使用的代码：

```
printf("buffer size: %d\n", fp->bsize);
```

其他的编译器可能将缓冲区的大小存储在其他变量中，或者使用其他方式来跟踪 buffer 值，因此试图修改 bsize 值可能引起错误。

```
fb->bsize = 1024; /* Not recommended */
```

除非了解文件存储的全部细节，否则上述操作对于不同的编译器或是同一编译器的更新版本都将是非常危险的。

同理，这里定义的 Stack 类型不是抽象数据类型，stack.h 暴露了 Stack 类型的具体实现，无法阻止客户将 Stack 变量作为结构体直接使用。

通过提供对 top 和 contents 的访问，模块的客户可以破坏栈的数据。更糟糕的是，在没有检查是否需要修改客户之前，不能修改栈的存储方式。

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

C 语言不能很好的支持抽象数据类型是后来开发 C++ 语言的原因之一。另外，C++ 语言中最重要的特性是支持类(class)，从而可以实现对数据类型的封装。

C++ 语言是由 AT%T 贝尔实验室的 Bjarne Stroustrup 在 20 世纪 80 年代开发的 C 语言的扩展版本。在现代程序设计理念上，C++ 语言比 C 语言更好的支持了抽象数据类型，并允许隐藏数据类型的细节。

45.3 Differences

相对与 C 语言来说，C++ 语言增加的特性包括类、重载、派生、虚函数、模板以及异常处理等。

45.3.1 Comments

C++ 语言支持单行注释。单行注释由//开头，在之后的第一个换行符处结束。

```
// This is a comment.
```

单行注释不会意外丢失注释结束的标记，因此比 C 语言的注释更安全，但是 C 语言的注释仍然是合法的。

45.3.2 Symbol

在 C++ 语言中，标记(用于标识特定的结构、联合或枚举名字)会自动被认为是类型名。例如，下面的两种类型定义是相同的。

```
typedef struct {double re, im;} Complex;

struct Complex{double re, im;};
```

45.3.3 Void

在声明或定义一个不带参数的 C++ 函数时，可以不使用 void。

```
void draw(void); // no arguments
void draw(); // no arguments
```

45.3.4 Arguments

C++ 语言允许函数的实际参数有默认值。例如，下面的示例代码可以显示任意个数的换行符。如果调用时没有传递任何实际参数，函数会显示一个换行符。

```
void new_line(int n = 1) // default argument
{
  while(n-- > 0)
    putchar('\n');
}
```

调用 new_line 函数时，可以提供一个实际参数，也可以不提供实际参数。

```
new_line(3); // prints 3 blank lines
new_line(); // prints 1 blank line by default
```

45.3.5 Reference

C 语言规定实际参数是按值传递的，如果要修改作为实际参数提供的变量，只能传递指向该变量的指针(数组除外)。

例如，为了实现将两个变量交换的函数，C 语言的实现版本如下：

```
/* swap(C version)*/
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

当调用 swap 函数时，其参数应该是指向变量的指针。

```
swap(&i, &j);
```

C++ 语言在这方面进行了一些改进，允许实际参数被声明成引用，而不是指针。

```
// swap(C++ version)
void swap(int& a, int& b) // a and b are references.
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

在 C++ 中，当调用 swap 函数时就不再需要在实际参数前加 & 运算符。

```
swap(i, j);
```

在 swap 的函数体中，a 和 b 被分别理解为 i 和 j 的别名。

- temp = a—将 i 的值复制给 temp。
- a = b—将 j 的值复制给 i。
- b = temp—将 temp 的值复制给 j。

45.3.6 Allocation

在 C 语言中，可以使用函数 malloc、calloc、realloc 和 free 来动态分配和释放内存。现在在 C++ 语言仍然支持这些函数，但是更好的做法是使用 new 和 delete。

new 和 delete 是运算符，不是函数。其中，new 用来分配空间，而 delete 用来释放分配的空间。

new 的操作数是一个类型说明符。当无法分配所要求的内存时，new 会返回空指针。

```
int *int_ptr, *int_array;
int_ptr = new int; // allocates memory for an int.
int_array = new int[10]; // allocates memory for an array of ten integers.
```

delete 需要用一个指针作为它的操作数。

```
delete int_ptr; // releases memory pointed to by int_ptr.
delete [] int_array; // [] required when deallocating an array.
```

The destructor of A will run when its lifetime is over. If you want its memory to be freed and the destructor run, you have to delete it if it was allocated on the heap. If it was allocated on the stack this happens automatically (i.e. when it goes out of scope; see RAII). If it is a member of a class (not a pointer, but a full member), then this will happen when the containing object is destroyed.

```
class A
{
   char *someHeapMemory;
public:
   A() : someHeapMemory(new char[1000]) {}
   ~A() { delete[] someHeapMemory; }
};

class B
{
   A* APtr;
public:
   B() : APtr(new A()) {}
   ~B() { delete APtr; }
};

class C
{
   A Amember;
public:
   C() : Amember() {}
   ~C() {} // A is freed / destructed automatically.
};

int main()
{
   B* BPtr = new B();
   delete BPtr; // Calls ~B() which calls ~A()
   C *CPtr = new C();
   delete CPtr;
   B b;
   C c;
} // b and c are freed/destructed automatically
```

In the above example, every delete and delete[] is needed. And no delete is needed (or indeed able to be used) where I did not use it.

auto_ptr, unique_ptr and shared_ptr etc... are great for making this lifetime management much easier:

```
class A
{
   shared_array<char> someHeapMemory;
public:
```

```
    A() : someHeapMemory(new char[1000]) {}
    ~A() { } // someHeapMemory is delete[]d automatically
};

class B
{
    shared_ptr<A> APtr;
public:
    B() : APtr(new A()) {}
    ~B() { } // APtr is deleted automatically
};

int main()
{
    shared_ptr<B> BPtr = new B();
} // BPtr is deleted automatically
```

delete 和析构关系是：前者是后者执行充分条件，后者对前者没有必然的影响，delete 会去调用被 delete 的指针所指向对象的析构函数来释放内存，而通常在析构函数中使用 delete 在本对象被析构之前先把对象里使用 new 实例化的一些对象析构，防止内存泄漏。

- 系统调用 delete 时会在 delete 内调用对象的析构函数。例如，new 一个数组时必须要有默认构造函数，delete 数组时其中的所有元素的析构函数均会被调用。
- 对象被销毁时，首先调用析构函数，而且析构函数也可以显式调用 A a; a. A(); 来销毁对象。
- delete 释放内存空间，只是对使用该块内存空间的对象调用析构函数，并且指定该处空间已经无对象使用。
- 如果在对象内是 new 出来的，在析构函数中需要 delete。

对象只在其生命周期结束时自动调用析构函数释放对象资源，而 delete 这个动作是释放对象就是标志生命周期结束所以必然引起析构动作，所以 delete 是析构的充分条件。

析构函数调用只是会可能使用 delete 来释放对象某些成员占用的内存，达到释放整体资源的目的。但是注意只是可能调用 delete 如果成员没有动态占用，那么就没有必要 delete，所以后者对前者不必然联系。

```
#include <iostream>

using namespace std;
class TmpClass{
public:
      int member;
      TmpClass(int a):member(a){
            cout<<"构造函数: int"<<endl;
      }

      TmpClass():member(0){//没有默认构造函数时没法new一个该类的数组
            cout<<"构造函数: 默认"<<endl;
      }

      TmpClass(TmpClass& tmp){
            cout<<"copy 构造函数"<<endl;
      }

      TmpClass& operator=(const TmpClass& tmp){
            if(this==&tmp){
                  cout<<"= 自己"<<endl;
                  return *this;
            }
```

```
            this->member=tmp.member+100;
            cout<<"= 非自己"<<endl;
            //没写return 也编译通过了
      }

//返回值改为void, 则下面的 m=m; 没问题, 但m=m=m;就不行了
//    void operator=(const TmpClass& tmp){
//          if(this==&tmp){
//                cout<<"= 自己了"<<endl;
//                return ;
//          }
//
//          this->member=tmp.member+100;
//          cout<<"= 非自己"<<endl;
//
//    }

      ~TmpClass()
      {
            cout<<"析构函数,member="<<member<<endl;
      }
};



void main(){
      cout<<"***\tbefore new"<<endl;
      TmpClass* tmp=new TmpClass[2];
      cout<<"***\tafter new"<<endl<<endl;
      tmp[0].member=10;
      tmp[1].member=11;

      TmpClass m(1);
      m=(m=m);
      cout<<"***\tbefore push"<<endl;
      tmp[0]=m;
      cout<<"***\tafter push"<<endl<<endl;

      cout<<"***\tbefore delete"<<endl;
      delete []tmp;
      cout<<"***\tafter delete"<<endl;
      cout<<"main函数返回前"<<endl;
}
```

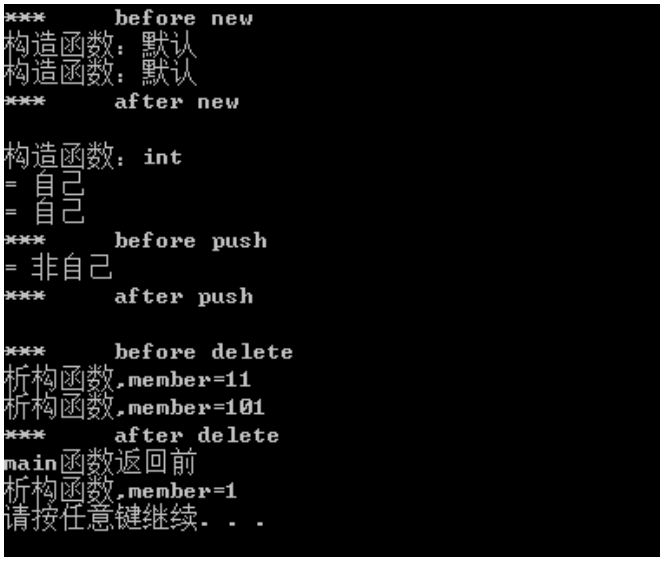
事实上,delete 对象应该完成两个操作:析构和释放空间,它们没有直接联系。当执行 delete 时,它先调用析构函数,然后调用 operator delete 删除这个指针。

另外,有些实现(比如 VC 和 gcc)为了支持多继承,在虚析构函数(而不是 delete 运算符处)的代码段中插入代码调用 operator delete 完成释放空间操作。

C++ 语言在尽可能地保持与 C 语言的兼容的同时,增加了更多强制性限制来增强安全性。

45.4 Classes

从根本上说,一个类就是一个抽象数据类型——一组数据以及操作这组数据的函数。



通过编写一个类,就可以产生一个新的数据类型,而且这个新的数据类型的功能可以和基本数据类型同样强大。

例如,如果要以分数的形式存储数(比如 1/4、3/7 等),通过编写一个类 Fraction 可以很方便地操作这些分数。

首先,可以按照下面的方式来声明 Fraction 类的变量。

```
Fraction f1, f2, f3;
```

随后,可以使用 = 运算符来复制分数,包括把分数传递给函数或编写函数来返回分数等。

在引入运算符重载后,可以将 C++ 运算符用作对 Fraction 对象的操作的名字。例如,通过重载 * 运算符,可以使其实现分数间的乘法,这样就可以使用下面的代码来将 f1 和 f2 相乘。

```
f3 = f1 * f2;
```

假设 f1 的值为 1/2,f2 的值为 2/3,f3 会被赋值为 1/3,因此通过重载运算符可以使类与基本数据类型一样简单易用。

类允许用户构造任何需要的数据类型,例如为了提供一种通常不是原生支持的数值类型,可以设计一个合适的类来将这种类型添加到数据类型列表中。

如果对普通类型的默认行为不能满足需要,也可以使用类来构造自己的类型。例如,自定义的 Array 类中可以有一个变量对下标进行越界检查,而且内置的 String 类型可以根据需要扩展或缩减。

类也适用于构造复杂的数据类型,例如队列、集合、栈等。

45.4.1 Definition

事实上,真正使类具有意义的是它可以用来对现实世界中的对象建模,而不仅仅是作为通用的数据结构。例如,在开发金融程序时,可以定义 Account 类来提供类似 deposit 和 withdraw 等的操作,这样就可以使程序的操作更接近实际,从而使程序逻辑更易读和易实现。

类的定义和结构的定义类似,最简单的类定义和结构结构几乎一样。

```
class Fraction{
  int numerator;
  int denominator;
};
```

这里,numerator 和 denominator 被称为是 Fraction 类的数据成员(data member)。

类标记(class tag)可以直接作为类型名使用,因此使用类来声明变量的方式也和使用结构相同。

```
Fraction f1, f2, f3;
```

这样，编译器就会构造三个变量 f1、f2 和 f3，而且每个变量都有自己的成员 numerator 和 denominator。

在面向对象的编程语言中，使用类来声明的变量有特殊的意义，它们被称作类的实例（instance），而且任何类的实例都是对象（object）。

45.4.2 Member

结构的成员可以用点运算符（.）和指向运算符（->）来访问，但是类中的成员默认是隐藏的，因此下面的示例代码是非法的。

```
f1.numerator = 0; // illegal
denom = f2.denominator; // illegal
```

一般情况下，类的成员都是私有（private）成员，可以通过将成员声明为 public（公有）来提供对它们的访问。

```
class Fraction{
  public:
    int numerator;
    int denominator;
};
```

可以混合使用公有的私有的成员。

```
class Fraction{
  public:
    int numerator;
  private:
    int denominator;
};
```

与结构体不同，类的私有成员不允许从类的外部进行访问。

45.5 Member Functions

45.5.1 Declaration

如果需要修改私有成员或者检查它们的值，则必须把访问类的数据成员的函数声明在类内部，这样的属于类的函数称为成员函数（member function）。

例如，下面的示例代码中把 numerator 和 denominator 声明为类 Fraction 的私有成员，并且给类添加两个成员函数 create() 和 print()。

```
class Fraction{
  public:
    void create(int num, int denom);
    void print();
  private:
    int numerator;
    int denominator;
};
```

这里，create() 和 print() 都是类 Fraction 的公有成员，可以在类的外部调用它们。

和结构体变量一样，成员函数通过对象和点运算符来调用。

```
f1.create(1, 2); // f1 now stores 1/2.
f1.print(); // prints "1/2"
```

f1 是类 Fraction 的对象，f1 调用的是类 Fraction 中的成员函数。在 f1 调用 create() 函数时，会将 1 存储到 f1 的 numerator 成员中，比将 2 存储到 f1 的 denominator 成员中。

f1 是类 Fraction 的对象，f1 调用的是类 Fraction 中的成员函数。在 f1 调用 print() 函数时，首先会显示 f1 的 numerator 成员，接着会显示一个/字符，最后显示 f1 的 denominator 成员。

下面是对 f1 调用 create() 函数的解释：

下面是对 f1 调用 print() 函数的解释：

为了说明成员函数与对象调用的关系，可以把对象理解为放置在成员函数前的实际参数。

在实际使用中，数据成员通常被声明为私有的，但是成员函数并不一定需要是公有的，但是私有的成员函数一般仅仅是用于类的内部实现。

```
class Fraction{
  public:
    void create(int num, int denom);
    void print();
  private:
    void reduce();
    int numerator;
    int denominator;
};
```

45.5.2 Definition

在类的内部声明了成员函数之后，接下来就可以在类定义之外定义每一个函数。

```
void Fraction::create(int num, int denom)
{
  numerator = num;
  denominator = denom;
  reduce();
}
```

在实现成员函数时必须添加类名前缀，否则 C++ 编译器会将函数作为一个普通的函数对待。

通常情况下，成员函数可以访问类的所有成员（包括私有的和公有的）。如果一个成员函数调用了另一个成员函数，默认是从同一个对象中调用的。

换句话说，如果调用 create() 函数来产生一个分数，可以使用下面的调用语句：

```
f1.create(1, 2);
```

实际上，上述的调用语句执行了下面的操作。

```
f1.numerator = num;
f1.denominator = donom;
f1.reduce();
```

另外，除了把成员函数的定义放在类外面之外，也可以把成员函数的定义放在类内部，但是往往只是在函数的实现非常短小时才会这样考虑。

```
class Fraction{
  public:
    void create(int num, int denom)
    {
      numerator = num;
      denominator = denom;
```

```
        reduce();
      }
      void print();
    private:
      void reduce();
      int numerator;
      int denominator;
};
```

下面的示例将继续给 Fraction 类添加乘法函数。

```
class Fraction{
  public:
    void create(int num, int denom);
    void print();
    Fraction mul(Fraction f);
  private:
    void reduce();
    int numerator;
    int denominator;
};
```

接下来，需要编写函数 mul 的定义。

```
Fraction Fraction::mul(Fraction f)
{
  Fraction result;
  result.numerator = numerator * f.numerator;
  result.denominator = denominator * f.donominator;
  result.reduce();

  return result;
}
```

实际使用时，如果使用下面的语句来调用 nul() 函数，可以将其理解为：

```
f3 = f1.mul(f2);
```

f1 是类 Fraction 的对象，因此调用的是 Fraction 类的 mul() 函数。

- 首先，mul() 函数会将 f1 的分子与 f2 的分子相乘，然后将产生的结果存储到 result 的分子中。
- 接着，mul() 函数会将 f1 的分母与 f2 的分母相乘，然后将产生的结果存储到 result 的分母中。
- 最后，mul() 函数返回 result，并且 result 将被复制到 f3 中。

45.6 Constructor

为了确保正确地初始化类的实例，类可以包含一个特殊的函数，称为构造函数（constructor）。

另外，类还可以提供一个析构函数（deconstructor）来对释放的类的实例进行清理。

构造函数和析构函数最方便（也最危险）的地方在于它们通常是被自动调用的，不需要明确的函数调用，编译器会在需要的时候自动调用它们。

一般情况下，构造函数是一个与类同名的函数。

```
class Fraction{
  public:
    Fraction(int num, int denom)
    {
```

```
      numerator = num;
      denominator = denom;
      reduce();
    }
    void print();
    Fraction mul(Fraction f);
  private:
    void reduce();
    int numerator;
    int denominator;
};
```

与其他的成员函数不同的是,构造函数没有指定的返回类型,而且构造函数一般是公有的。

构造函数可以和其他成员一样调用,但是它们通常是在声明实例时隐式调用的。例如,在类 Fraction 的实例 f 的声明中,Fraction 类的构造函数会被隐式调用,调用时的实际参数是 3 和 4,于是 f 的初始值为 3/4。

```
Fraction f(3, 4);
```

上述的实例声明语句可以理解为下面的操作的集合。

```
f.numerator = 3;
f.denominator = 4;
reduce();
```

一般情况下,构造函数通常有默认实际参数。

```
class Fraction{
  public:
    Fraction(int num = 0, int denom = 1)
    {
      numerator = num;
      denominator = denom;
      reduce();
    }
    void print();
  private:
    void reduce();
    int numerator;
    int denominator;
}
```

这里,num 和 denom 有默认值,因此类 Fraction 的构造函数在被调用时可以有两个实际参数。

- 如果类实例化时有一个实际参数,则类 Fraction 的构造函数会提供另一个实际参数 1。

```
Fraction f(3); // same as Fraction(3, 1);
```

- 如果类实例化时没有实际参数,则类 Fraction 的构造函数会提供默认的实际参数。

```
Fraction f; // same as Fraction(0, 1);
```

45.7 Memory Allocation/Deallocation

构造函数和析构函数对那些需要动态分配存储空间(使用 new 和 delete 运算符)的函数特别有用。例如,为了突破普通的 C 字符串的限制,可以创建自己的 String 类来处理字符串操作。

- String 对象可以包含任意长度的字符串。在 C 语言中,字符串的大小受限于数组的长度。
- String 对象的长度可以迅速地确定。要得到 C 语言字符串的长度,需要调用 strlen 函数,而 strlen 函数则需要遍历整个字符串来找到标志字符串结束的空字符。

- 需要时可以给 String 类添加操作。在 C 语言中，很难方便地修改 <string.h> 来增加函数。

下面是在 C++ 语言中声明 String 对象的方式。

```
String s1("abc"), s2("def");
```

在声明 String 对象时可以对其进行初始化，这样 s1 就包含 “abc”，s2 包含 “def”，而且这些对象的值都可以修改。

String 对象对字符串的长度没有限制的事实，使得 String 对象需要包含一个指针来动态地分配内存。另外，出于对运行速度的考虑，还需要增加一个成员来保存字符串的长度。

```
class String{
  private:
    char *text; // pointer to string
    int len; // length of string
};
```

接下来，需要一个构造函数来将普通的字符串转换成 String 对象。

```
class String{
  public:
    String(const char *s); // constructor
  private:
    char *text;
    int len;
};
```

下面就是一个可能的构造函数的实现。

```
String::String(const char *s)
{
  len = strlen(s);
  text = new char[len + 1];
  strcpy(text, s);
}
```

在计算了 s 所指向的字符串的长度后，构造函数使用 new 运算符分配足够的内存来复制字符串。最后，构造函数将字符串复制到刚分配的内存中。

45.8 Deconstructor

下面考虑类与动态分配有关的问题，如果在一个函数内部使用 String 对象，在函数调用返回时内存释放将很难正常进行。

```
void f()
{
  String s1("abc");
  ...
}
```

当函数 f 被调用时，s1 对象开始存在，于是 s1 的构造函数分配了一个 4 字节的字符数组，并将字符串 “abc” 复制到数组中。

当函数 f 返回时，s1 将不再存在。虽然 s1 使用的是自动存储期限，在释放 String 对象所占的内存时，仅仅会释放其成员 text 和 len 使用的内存，而不会释放 text 所指向的内存，于是导致程序出现内存泄漏。

释放动态分配的内存时的问题是 C++ 语言提供析构函数的原因之一。析构函数可以在对象被释放时自动被调用来处理内存释放的问题。

构造函数和析构函数关系密切，其中构造函数在对象实例化时对其进行初始化，析构函数在对象销毁时对其进行清理。如果一个类的构造函数动态分配了内存，那么很可能需要析构函数来释放分配的内存。

与构造函数一样，析构函数也是一个成员函数，名字也与类名一致，只是在开头增加一个～字符。构造函数退出时，仍然需要清理内存。

析构函数没有返回类型，也没有实际参数。例如，下面的示例说明了添加了析构函数的 String 类。

```
class String{
  public:
    String(const char *s);
    ~String()
    {
      delete [] text;
    }
  private:
    char *text;
    int len;
};
```

通过析构函数 ~String 可以释放 text 所指向的字符数组。

45.9 Overload

45.9.1 Function Overloading

C++ 语言支持在同一个作用域内的两个或多个函数具有相同的函数名。当函数以这种方式重载时，编译器会通过检测函数的实际参数来决定哪个函数被调用。例如，下面的示例演示了在同一作用域中有两个不同版本的函数 f 的情况。

```
void f(int);
void f(double); // overloading f()
```

接下来的示例代码说明了 f 调用是如何被转变的。

```
f(1); // a call of f(int)
f(1.0); // a call of f(double)
```

对于执行相同操作但是操作数的类型不同的函数，重载的引入使得可以函数可以具有相同的名字。例如，下面的函数都是计算 x 的 y 次方，区别只是实际参数的类型不同。

```
int pow(int x, int y);
double pow(double x, double y);
```

类中的成员函数可以被重载，这实际上也是函数重载在 C++ 中最常见的使用方式。例如，通过重载可以给 String 类添加另一个构造函数。

```
class String{
  public:
    String(const char *s);
    String()
    {
      text = 0;
      len = 0;
    }
    ~String()
    {
      delete [] text;
    }
```

```
  private:
    char *text;
    int len;
};
```

0 代表空指针，因此 C++ 中更多的是使用 0 而不是 NULL 来进行指针变量的初始化。

不带实际参数的构造函数称为默认的构造函数。例如，上述 String 类的默认构造函数会在声明 String 对象而没有指定初始值时被调用。

```
String s; // invoking default constructor
String s("abc"); // invoking constructor String(const char *s)
```

45.9.2 Operator Overloading

C++ 语言支持的运算符重载可以给传统的 C 语言运算符赋予新的含义，这样可以定义新的数据类型来扩展编程语言本身。

通过运算符重载，可以根据操作数类型的不同来给运算符赋予不同的操作，这样就可以重新定义 C++ 语言中的运算符来应用到类的实例上，从而产生更自然也更易读的程序。

运算符重载需要在声明函数时使用 operato 加运算符来代替函数名。例如，如果将 mul 函数替换为 * 运算符，Fraction 类就会具有更好的可读性，而且类的客户可以使用重载运算符来执行操作。

```
class Fraction{
  public:
    void create(int num, int denom);
    void print();
    Fraction operator*(Fraction f);
  private:
    void reduce();
    int numerator;
    int denominator;
};
```

接下来所做的只是修改函数的名字，函数的内部实现不变。

```
Fraction Fraction::operator*(Fraction f)
{
  Fraction result;
  result.numerator = numerator * f.numerator;
  result.denominator = denominator * f.denominator;

  return result;
}
```

当一个运算符被定义为类的成员函数时，它其中的一个操作数始终是隐含的，因此重载的运算符 * 是一个二元运算符。

例如，在执行下面的语句时，编译器会在对操作对象和运算符进行判断之后才执行对应的操作。

```
f3 = f1 * f2;
```

首先，编译器判断出 f1 是一个 Fraction 对象，于是会查看 Fraction 类并找到名字为 operator* 的函数。

编译器最终会将上述语句中的 * 转换成 operator*，这样就可以和执行一般的成员函数一样进行操作。

```
f3 = f1.operator*(f2);
```

45.10 Input/Output

C++ 语言在继续使用 stdio.h 之外提供了额外的 I/O 库，其中 iostream.h 是 C++ 输入/输出函数库中最重要的头文件。

iostream.h 中定义的类包括 istream(输入流)和 ostream(输出流)等，I/O 通过对 istream 和 ostream 操作来进行。其中，cin 是 istream 类的一个实例，而 cout 对象是 ostream 类的一个实例。

对于从键盘获得输入和向屏幕上显示输出的简单程序，可以使用 cin 对象来进行输入，使用 cout 对象来进行输出。

istream 类和 ostream 类都与运算符重载关系密切。特别是，重载的 C 运算符 « 和 »(向左和向右移位)用于绝大多数的 C++ 读和写操作中。

- istream 类重载了 »，使其可以从输入流中读取数据。
- ostream 类重载了 «，使其可以从输出流中读取数据。

使用 « 和 » 进行交互的形式大致如下：

```
cout << "Enter a number: ";
cin >> n;
cout << "The square is: ";
cout << n * n;
cout << "\n";
```

这里，第一条语句的意义是："cout 是一个 ostream 对象，所以调用的是 ostream 类的 operator« 函数，而 operator« 函数会将字符串 "Enter a number: " 作为它的实际参数。"

在 C++ 引入新的 I/O 库的一个好处是可以扩展它们来读/写类的实例。例如，在下面的示例中使用重载运算符 « 来写一个 Fraction 对象，从而进一步使得类和基本类型的相互通用。

```
Fraction f(3, 4);
cout << f; // prints 3/4
```

45.11 Inheritance

C++ 语言支持面向对象的程序设计，并通过允许从已经存在的类"派生"出新的类来提高代码的复用性。

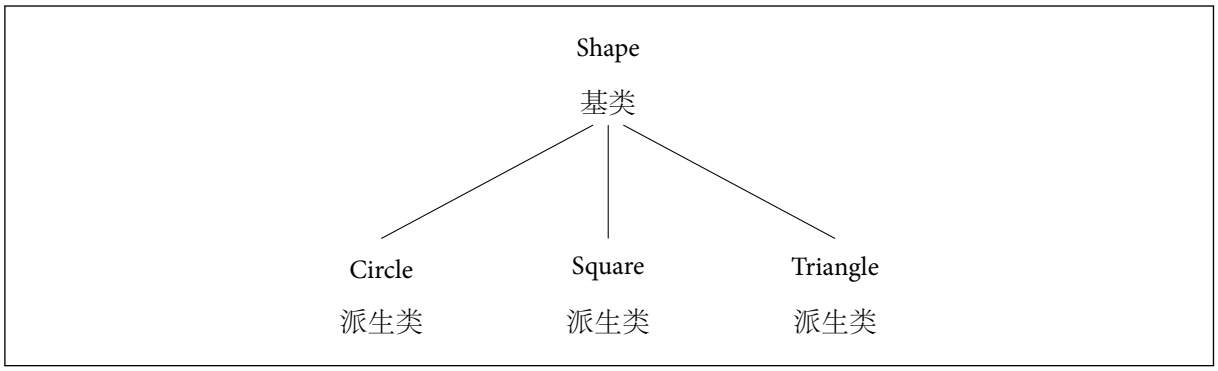
当需要一个新类时，C++ 语言允许从一个已定义的类来派生出新类，从而不需要重写一个新类。例如，一个用来生成图形的程序中可能需要 Circle、Square 以及 Triangle 这三个类。这些类可以都从一个更通用的类 Shape 来派生。

对于这三个类中共同的属性，可以只在 Shape 类中定义一次，同时对于所有形状都适用的操作也可以定义在 Shape 类中。例如，如果每个形状都有颜色和坐标，而且每个形状都可以改变它的颜色和位置，因此可以得到 Shape 类的实现如下：

```
class Shape{
  public:
    void change_color(int new_color);
    void move(int x_change, int y_change);
    ...
  private:
    int x, y; // coordinates of center
    int color; // current color
    ...
};
```

按照继承的层次结构，Shape 类是基类，而类 Circle、Square 和 Triangle 都是派生类。

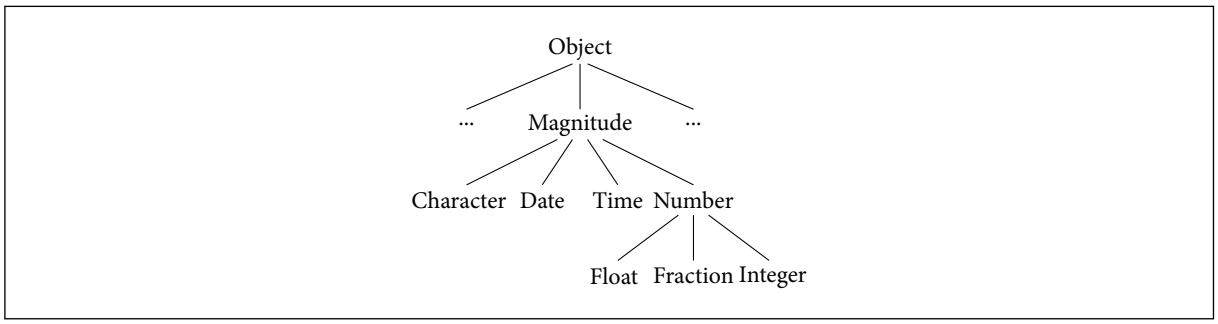
下面的示意图显示了基类与派生类之间的关系。



通过类的继承层次可以复用代码(例如可以从 Shape 类继续派生出 Pentagon(五角星)类),而且派生类可以从基类中继承大多数它们需要的属性和操作等,于是就仅需要继续实现不能共享的属性和操作。

派生也可以简化代码维护工作,例如如果需要给所有派生类都添加一个属性(或操作),只需添加到基类中就可以使所有派生类继承得到。

通过继承可以被用来开发可以扩展的相关类的库。例如,Samlltalk 的所有的类都会直接或间接地从一个单一的 Object 类派生。



每个派生类都会包含它的基类中的所有属性,以及它自身所特有的属性。例如,Magnitude(量级)类的对象有一个共同的特性——使用关系运算符(大于、小于等)进行比较。Number 类的对象就继承了这一特性,同时还提供了对算术运算的支持,但是算术运算不会提供给其他 Magnitude 对象(虽然对日期进行比较是有意义的,但是对日期进行算术运算是没有意义的)。

为了说明 Circle 类是从 Shape 类继承的,在 Circle 类的定义中需要包含一个派生列表。

```
class Circle: public Shape{
  ...
};
```

Circle 类继承了 Shape 类的成员(除了构造函数和析构函数外),因此 Shape 类中的成员也是 Circle 类的成员。

派生类可以声明自己独有的数据成员和成员函数。例如,Circle 类可能会需要一个数据成员来保存圆的半径,于是就可以添加到 Circle 类中。

```
class Circle: public Shape{
  public:
    ...
  private:
    int radius;
    ...
};
```

Circle 类继承自 Shape 类,因此 Circle 类的对象还包含数据成员 x、y 和 color。

当一个类从另一个类继承时，C++ 语言允许基类的指针指向派生类的实例。例如，Shape* 类型的变量可以指向 Circle、Square 或 Triangle 对象。

```
Circle c;
Shape *p = &c; // Shape pointer points to a Circle instance
```

类似地，C++ 语言还允许基类的指针作为形式参数与指向派生类的实例对象的实际参数匹配。例如，Shape& 类型参数也可以匹配任何 Circle、Square 或 Triangle 对象。

在下面的示例中，函数 add_to_list 表面上要求一个 Shape 类型的实际参数，但是实际上也可以使用 Circle、Square 或 Triangle 类型的对象，于是 add_to_list 就可以灵活处理各种派生类的操作。

```
void add_to_list(Shape& s)
{
  ...
}
```

45.12 Virtual Functions

类继承与虚函数结合使用会更有价值。

虚函数可以在基类中声明，然后在各个派生类中提供不同的实现。以 Shape 类为例，每个 Shape 对象都可以增加它的大小，于是可以引入一个 grow 函数。

```
class Shape{
  public:
    void grow();
    ...
  private:
    ...
};
```

为了解决 grow 函数对每个派生类都不同的问题，可以在基类中将 grow 类声明为 virtual。

```
class Shape{
  public:
    virtual void grow();
    ...
  private:
    ...
};
```

接下来，基类 Shape 的派生类 Circle、Square 和 Triangle 就可以提供自己的 grow 实现。例如，Circle 类的 grow 函数的实现大致如下：

```
class Circle: public Circle{
  public:
    void grow()
    {
      radius ++;
    }
    ...
  private:
    int radius;
    ...
};
```

大多数情况下，虚函数与普通的成员函数具有相同的行为。假设 c 是一个 Circle 对象，调用 c.grow() 会增加 c 的半径。

但是，当通过基类的指针（或引用）调用该函数时，虚函数会有特殊的处理：根据指针当前所指向的对象的实际类型来决定所调用的函数的实现版本。

例如，p 是一个指向 Shanp 对象的指针，Circle 和 Square 都是从 Shape 类派生的，于是 p 可以指向其中一个类的实例。当使用 p 调用调用 grow 函数时，如果 p 指向的是一个 Circle 对象，则调用 Circle::grow()。如果 p 指向的是一个 Square 对象，则调用的是 Square::grow()。

```
Shape *p;
Circle c;
Square s;

p = &c; // p points to a Circle
p->grow(); // calls Circle::grow()
p = &s; // p points to a Square
p->grow(); // calls Square::grow()
```

相比结构体，通过指向对象的指针调用成员函数时，需要使用-> 运算符而不是. 运算符。

对于普通函数和重载函数的调用，编译器可以判断出正确的函数版本。但是，编译器并不是总能判断出虚函数的实现版本，因此虚函数的调用必须依赖于“动态绑定”技术。

在下面的示例中，编译器无法判断出应该调用 grow 函数的哪一个版本，知道程序运行时才能确定。

```
if(TRUE)
  p = &c;
else
  p = &s;
p->grow();
```

使用动态绑定技术可以将从同一个基类派生出的类构造出包含不同类的对象的数据结构，然后对每个对象执行相同的操作，而且每一个对象会根据其所属的类采取不同的响应。

例如，把 Shape 对象存储在一个列表中以跟踪当前在屏幕上显示的对象。使用动态绑定技术可以逐个访问每个对象，并调用它们的 grow 函数来改变对象的大小，而不需要知道它具体是什么形状。

```
while(不在列表末尾){
  让p指向当前形状;
  p->grow();

  向前移动至列表中下一个项目;
}
```

使用动态绑定技术之后，就不再需要在执行操作前使用 switch 语句来检测每一个对象，从而可以简化代码实现。

另外，使用动态绑定技术可以不必改动操作数据结构的代码的情况下增加新类和删除旧类。例如，如果增加了一个 Pentagon 类（由 Shape 类派生），就不需要改动扩大每个形状的循环。

45.13 Template

C++ 语言支持的模板（template）可以使我们写出更通用的、高度可复用的类和函数等。

模板是构造类所使用的“模式（pattern）”，而且 C++ 语言也支持函数模板。

除了对类定义的部分内容未加具体说明以外，模板看起来很像一个普通的类，这样使用模板忽略类定义的部分内容可以使类更通用，也更易复用。

如果将 Stack 类型转换为一个 C++ 类，首先会得到如下的定义：

```
class Stack{
  public:
    void make_empty();
```

```
    int is_empty();
    void push(int);
    int pop();
  ...
};
```

但是，Stack 对象只能存储整数。如果需要一个可以存储其他类型的栈时，需要将类定义复制一份并改变类名外，还需要将类的内部数据结构作出修改。

为了解决上述的问题，可以构造一个 Stack 模板。

```
template <class T>
class Stack{
  public:
    void make_empty();
    int is_empty();
    void push(T);
    T pop();
    ...
};
```

首先，template <class T> 指明 Stack 是一个模板，直至将缺少的类 T 补充完整。这里，T 代表一个“模板实际参数”，可以修改成其他名字。

其次，类的成员函数的参数发生了变化。例如，push 函数需要的参数是 T 类型，而且 pop 函数会返回 T 类型的值。

最后，Stack 的成员函数需要修改成下面的形式：

```
template <class T>
void Stack<T>::push(T, x)
{
  ...
}
```

模板类在提供了模板参数后才会被“实例化”。这里将 T 表示成类，但是 Stack 的实际参数不需要一定是类，任何 C++ 类型都可以。例如，下面是 3 种不同的 Stack 类的实例化方式，分别使用了 int、float 和 char 作为模板实际参数。

```
Stack<int> int_stack; // stack of int values;
Stack<float> float_stack; // stack of float values
Stack<char> char_stack; // stack of char values
```

下面对 push 的调用解释了如何使用这三种栈。

```
int_stack.push(10); // push 10 onto int_stack top
float_stack.push(1.0); // push 1.0 onto float_stack top
char_stack.push('a'); // push 'a' onto char_stack top
```

45.14 Exception Handling

异常是指程序运行时可能产生的情况，通常作为出错的结果。例如，使用 Stack 对象时可能产生两种错误：

- 在栈已满时试图向栈中压入数据。
- 在栈为空时试图弹出数据。

C++ 语言通过一种统一的方式来检测和响应错误。例如，在上述的情况中，可以分别使用 StackFull 异常和 StackEmpty 异常来表示相应的错误。

当错误发生时，函数可以“抛出”一个异常。例如，在 Stack 类的例子中，push 函数和 pop 函数可以相应地抛出 StackFull 和 StackEmpty 异常。

```
void Stack::push(int i)
{
  if(栈满)
    throw StackFull();
    ...
}
int Stack::pop()
{
  if(栈空)
    throw StackEmpty();
    ...
}
```

可能会发生的异常的代码放置在测试程序(“try 程序块”)中,异常会被处理程序(“catch 程序块”)捕获。例如,在下面的例子中,一个包含 push 和 pop 调用的 try 程序块跟着两个 catch 程序块。其中,第一个 catch 程序块处理 StackFull 异常并显示“Error: Stack full.”,第二个 catch 程序块处理 StackEmpty 异常并显示“Error: Stack empty”。

```
try
{
  ...
  s.push(y);
  ...
  z = s.pop();
  ...
}
catch(StackFull)
{
  cout << "Error: Stack full.\n";
}
catch(StackEmpty)
{
  cout << "Error: Stack empty.\n";
}
```

注意,在异常处理之后,程序会从最后的 catch 程序块后面的语句继续执行,而不是返回或中止。

对于一个异常,如果在当前的 try 程序块末尾没有相应的 catch 来捕获,C++ 语言并不会放弃。实际上,C++ 编译器会检查被包围的 try 程序块来寻找相应的处理程序。如果查找失败,则会中止当前函数,并将当前的异常传递给调用函数处理。

如果有需要,还会继续向上传递给再上层的调用者,依次类推。最坏的情况是异常会一直传递给 main 函数,如果 main 函数也无法处理这个异常,整个程序会被终止,异常的这种性质可以帮助避免异常被意外地忽略。

Part VII

Interface

第 46 章

Introduction

如果编写代码时要利用已有的代码，则必须定义某种接口或者通信的方法。接口（interface）通常包含了对子程序、对象、类或原型的引用或调用。

接口泛指实体把自己提供给外界的一种抽象化物（可以为另一实体），用以由内部操作分离出外部沟通方法，使其能被修改内部而不影响外界其他实体与其交互的方式，就如面向对象程序设计提供的多重抽象化。

- 人类与计算机等信息机器或人类与程序之间的接口称为用户界面。
- 计算机等信息机器硬件组件间的接口叫硬件接口。
- 计算机等信息机器软件组件间的接口叫软件接口。

接口可能也提供某种意义上的在讲不同语言的实体之间的翻译，诸如人类与计算机之间。因为接口是一种间接手段，所以相比起直接沟通，会引致些额外负担。

软件部件间的接口可以包括常数、数据类型、过程、异常处理和类型签名等。

在某些特殊情况中，定义变量作为接口的一部份可能会很有用。

另外，port 也被译为“接口”或“端口”，而 interface 的中文译名为“界面”或“接口”。

其中，当接口是 port 的译名时，可以指：

- 硬件接口是电脑硬件中可连接两个或两个以上不同之电路装置使之能够传递电子或任何形式信号的装置。
- TCP/UDP 端口

当接口是 interface 的译名时，它指的是用于沟通的中介物的抽象化。

46.1 Hardware Interface

硬件接口可以理解为在计算机等的信息机器的硬件之间通信时的物理连接器形状、传送接收信号的方法（协议）等的规格。

硬件接口主要可分为并行的和串行两种，其中串行相比并行多使用同一电线作为信号控制线和电源线。在个人计算机领域，从并行连接向更高传输速度的发展遇到瓶项，因而在向各接口的串行连接方式迁移。

通用的可热拔插的串行硬件设备包括 USB、IEEE 1394、以太网（100BASE 为止）、ExpressCard、eSATA 等，而并行硬件设备则包括以太网（1000BASE-T）等。

不支持热插拔的并行硬件设备包括 SCSI、IDE 和 PCI 等，另外 PCI-Express、Serial ATA 等串行设备也不支持热拔插。

46.2 Software Interface

软件接口定义了软件间通信时传递消息（message）的规格，例如 API、进程间通信和计算机网络等。

程序设计中的接口可以理解为在程序编写或设计的方法论中程序组件功能的抽象化物，通常接口会通过注释或逻辑断言来指明过程和方法的功能。

任一个软件模块 A 的接口会与该模组的实现保持分离，其中实现包括描述于接口的过程和方法的实现代码，比如其他“私有”变量、过程等。

任何其他软件模块 B（可以归类为 A 的客户）与 A 交互都强制通过接口来进行，这样当修改 A 的实现为符合该接口的相同规范的另一个，应该不会导致 B 故障，只要 B 那些使用到 A 的部份一直遵守该接口的规范。

在面向对象编程中，接口通常定义为一些方法的集合，通常对对象的属性的访问通过属性存取函数来进行。

接口在投入使用之后就不应该被修改。如果接口的实现模块提供了新的功能，而想在其他模块中调用这个功能，那么需要定义新的部份而不是修改现存的接口。

尽管接口的定义没有强制的标准，但是一些标准的 COM 接口的应用十分广泛（例如 IUnknown 和 IDispatch）。例如，在面向对象程序设计中，一些支持动态语言的模块实现了 IDispatch 来支持在运行时“发现”对象提供的函数、方法和事件（通常称为自动化），但是这个通过 IDispatch 来做代理的方法使得程序性能有所降低。

46.3 API

应用程序接口又称为应用编程接口，实际上是软件系统不同组成部分衔接的约定。

由于近年来软件的规模日益庞大，常常需要把复杂的系统划分成小的组成部分，编程接口的设计十分重要。在程序设计的实践中，编程接口的设计首先要使软件系统的职责得到合理划分。

良好的接口设计可以降低系统各部分的相互依赖，提高组成单元的内聚性，降低组成单元间的耦合程度，从而提高系统的维护性和扩展性。

应用程序接口可以理解为操作系统或程序库提供给应用程序调用的代码，其主要目的是让开发人员得以调用一组例程功能，而无须考虑其底层的源代码实现或理解其内部工作机制的细节。

API 本身是抽象的，它仅定义了一个接口，而不涉入应用程序如何实现的细节，因此 API 经常是软件开发工具包（SDK）的一部分。

例如，图形库中的一组 API 定义了绘制指针的方式，可于图形输出设备上显示指针。当应用程序需要指针功能时，可在引用、编译时链接到这组 API，而运行时就会调用此 API 的实现（库）来显示指针。

API 可以是一组数量庞大、极其复杂的函数和子程序，程序员调用 API 可以完成很多任务，比如“读取文件”、“显示菜单”、“在视窗中显示网页”等。

操作系统的 API 还可以用来分配存储器或读取文件，许多系统应用程序都是通过 API 接口来实现的，比如图形系统、数据库、网络、Web 服务和在线游戏等。

API 可以有诸多不同设计，其中用于快速执行的接口通常包括函数、常量、变量与数据结构等。API 的其它方式包括通过解释器或是提供抽象层以屏蔽和 API 实现相关的信息，确保使用 API 的代码无需更改而适应实现变化。

API 又分为（Windows、Linux、Unix 等系统的）系统级 API，及非操作系统级的自定义 API。作为一种有效的代码封装模式，微软 Windows 的 API 开发模式已经为许多商业应用开发的公司所借鉴，并开发出某些商业应用系统的 API 函数予以发布，方便第三方进行功能扩展，例如 Google、苹果电脑公司以及诺基亚等手机开发的 API 等。

API 的应用开发需要按照 API 发布者提供的的规范进行开发。下面的两个示例是 Windows API 在各编程语言中的表达方式。

```
/* Visual Basic */
[Public|Private]
Declare Function|Sub name Lib "libname" [Alias "aliasname"]([[Byval] variable [As type][, [
    Byval] variable [As type]]...])
[As type]
```

```
/* C# */
[DllImport("libname", 'Named Parameters')]
[public|private|internal] [Type] FunctionName(Type parameter1,Type parameter2...);
```

第 47 章

Reuse

代码复用(也被称作软件复用[1])就是指利用已有的代码,或者相关的知识去编写新的代码来构造软件。

可复用的代码以及相关的知识与需求文档、设计、测试用例一样都是软件开发的组织内部所不可或缺的资产。事实上,最早为人所知的复用正是从代码复用开始的。

所谓的代码复用,本质上就是对曾经编写过的代码的一部分或者全部重新加以利用,从而构建新的程序。使用这种方法就可以将程序员从费时费力的重复劳动中解放出来。

为了使代码复用更加方便,更加迅速并且更加体系化,相关的研究者也进行了大量的研究。面向对象程序设计就是以此为目的而衍生出来的方法。

接下来出现的代码复用还包括代码自动生成等,它会基于用户设置的一系列参数来自动生成程序的代码。与此类似的概念被称之为元编程。

用于衡量代码可复用程度的特性通常包括:模块化、低耦合、高内聚、数据封装以及 SOC。

在程序设计领域,由罗伯特·C·马丁在 21 世纪早期引入了 SOLID(单一功能、开闭原则、里氏替换、接口隔离以及依赖反转)原则,指代了面向对象编程和面向对象设计的五个基本原则。

- SRP(Single Responsibility Principle)
  单一功能原则(S)认为对象应该仅具有一种单一功能的概念,一个设计元素只做一件事。
- OCP(Open/Close Principle)
  开闭原则(O)认为“软件体应该是对于扩展开放的,但是对于修改封闭的”的概念。
- LSP(Liskov Substitution Principle)
  里氏替换原则(L)认为“程序中的对象应该是可以在不改变程序正确性的前提下被它的子类所替换的”的概念。
- ISP(Interface Substituation Principle)
  接口隔离原则(I)认为“使用多个特定客户端的接口要好于一个宽泛用途的总接口”的概念。
- DIP(Dependence Inversion Principle)
  依赖反转原则(D)认为一个方法应该遵从“依赖于抽象而不是一个具体的实例”的概念,其中依赖注入是该原则的一种实现方式。

47.1 Library

库(Library)就是一种代码复用的很好的形式,程序员可以创建内部抽象以便程序的部分代码可被复用,或者直接创建一个自定义库给自己使用。

编写库是进行代码复用最常见的方法。很多共通的操作,比如文件的读/写、操作系统信息的获取的动作都会被封装在库中,然后由软件开发人员来调用。

库所提供的操作都是经过充分测试的,但同时库无法对它提供的操作的具体实现进行调整,同时库也要求软件开发人员花大量时间去学习它的用法。

[1]对于以某个已有程序的旧版本作为出发点来进行新版本的开发的做法称为二次开发,它也可被视为一种代码复用。

47.2 Patterns

设计模式对于同类的相似问题提供了通用的解决方法。

术语“设计模式”是由 Erich Gamma 等人在 1990 年代从建筑设计领域引入到计算机科学的，它是对软件设计中普遍存在(反复出现)的各种问题所提出的解决方案。

设计模式并不直接用来完成代码的编写，而是描述在各种不同情况下，要怎么解决问题的一种方案，因此设计模式提供的是概念上的解决方法，对于具体问题往往还需要具体实现一份代码。

利用抽象类或接口往往可以在特定的设计模式中达到代码的复用，例如面向对象设计模式通常以类型或对象来描述其中的关系和相互作用，但不涉及用来完成应用程序的特定类型或对象。

设计模式能使不稳定依赖于相对稳定、具体依赖于相对抽象，避免会引起麻烦的紧耦合，从而增强软件设计面对并适应变化的能力。

并非所有的软件模式都是设计模式，设计模式特指软件“设计”层次上的问题，还有其它非设计模式的模式，例如架构模式。

算法不能算是一种设计模式，因为算法主要是用来解决计算上的问题，而非设计上的问题。

47.3 Frameworks

软件开发人员往往可以通过第三方的应用程序或者框架来实现对一个程序的大范围复用，这可以有效地提高生产效率，只是第三方的应用程序或者框架往往只在某个特定的应用范围内才可以适用。

第 48 章

Header file

在计算机科学中,接口是一个概念性的实体,由实现库的程序员和使用库的程序员之间的理解组成,清楚地说明双方都需要的信息。

当编写一个 C 语言的程序时,必须有一种方法将概念性的接口表示为实际程序的一部分。

C 语言中的接口习惯上被表示为头文件,每个头文件都指出了一个基本的库的接口,而且头文件包含大量的文档以及由库导出的函数原型。

接口的抽象概念和表示接口的实际的头文件之间的区别似乎很小。在许多方面,这个区别类似于算法和实现此算法的程序。算法是一个抽象的策略,而程序是一个算法的具体实现。同样,C 语言用头文件提供接口的具体实现。

上述这种抽象概念和程序表示之间的区别还产生了另外两个术语的定义,它们通常用于有关接口的讨论中。

首先,在计算机科学中引入了软件包(package)这个术语来描述定义一个库的代码。如果要开发一个库,则该工作包括产生一个.h 文件作为库的接口,以及实现该库的一个或多个.c 文件,这些文件合起来组成了软件包。然而,为了深入理解库,还必须注意该软件之外的东西。

库包含了超过软件包本身的特定的概念性的方法,库的概念性的基础就是抽象(abstraction)。

为了说明抽象和库之间的关系,这里以输入/输出操作为例来说明。程序中的所有的输入/输出操作可以使用 stdio.h 接口中的 scanf 和 printf 函数来完成。其中,对输入来讲,可以用 GetInteger、GetReal 和 GetLine,它们是 simpio.h 接口提供的。

stdio.h 同时也提供了接受用户输入的函数 scanf,因而这两个库包含了输入/输出操作的不同方法,其中 stdio.h 接口强调功能和灵活性,而 simpio.h 接口强调的是结构简单和容易使用。

每个接口中所用的方法都是抽象的一个部分,相关的软件包实现了这个抽象并使它成为现实,然后它们就能被程序员使用。

48.1 Interface

在计算机游戏或商用的字处理系统中,现代计算机程序都以更具有创造性的方法使用屏幕,包括图片和奇特的图形显示,这些特性使得计算机更容易使用,也更有趣。

虽然图形显示使用户的工作更加方便,但将这些内容组合到一个程序中将给程序员带来极大的挑战。从整体上考虑,在屏幕上划一条简单的线就是一个非常困难的程序设计问题,而通过引入图形库软件包,就不需要再完整地考虑这个问题。

使用图形库可以忽略计算机绘图的复杂性,并集中考虑一些高级操作,从而可以在屏幕上显示某些线和其他图形特性。细节问题隐藏在接口边界的实现的一端,这里的扩展库 graphics.h 就是一个说明了在屏幕上画图的简单抽象的接口。

要使用图形库,可以在程序的开始通过 #include 行来指定其接口。

```
#include "graphics.h"
```

在使用 graphics.h 接口中可用的过程和函数之前,首先需要理解图形模型,这就需要了解基本的抽象,包括如何指定屏幕上的位置和用什么作为长度单位等,这些问题对理解图形模型是很重要的,它是概念抽象的核心部分。

在使用图形库的程序中,程序的运行结果将在屏幕上显示,而所使用的屏幕的图形显示功能取决于所用的计算机硬件,因而 graphics.h 接口应尽可能被设计得通用一些,但这个通用性也使得难以精确描述在特定系统中如何显示图形。

通常,当启动一个图形软件包时,在屏幕上会创建一个新的矩形窗口,称为图形窗口(graphics window),该窗口作为绘图平面,当调用图形库中的过程或函数时,结果就会显示在图形窗口中。

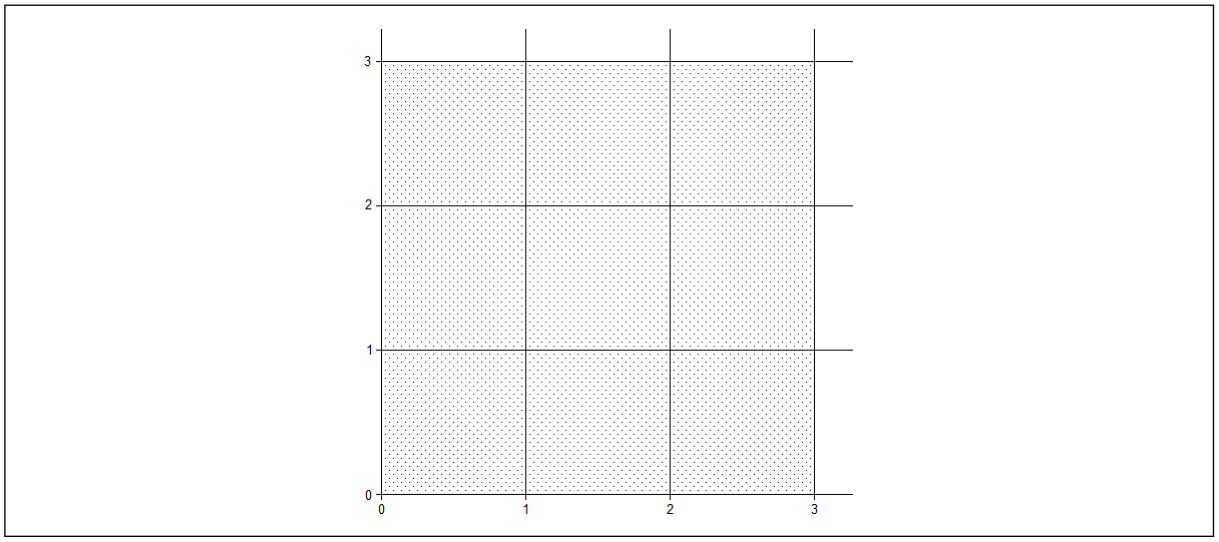
为了指定图形窗口中的点,图形库采用了一种几何或代数中所用的方法,图形窗口中的所有图形占据了其中的一个概念网格。

和传统的几何一样,点是用相对与原点(origin)的位置来标识的,原点位于图形窗口的左下角。从原点出发的沿着图形窗口边界的水平线和垂直线称为轴(axe),x 轴沿着窗口的底线,y 轴沿着窗口的左边。图形窗口中的每个点都能用一对值来标识(coordinates),通常记为(x, y),它指出了该点沿 x 轴和 y 轴的位置。这些值被称为点的坐标。坐标用相对于原点的英寸计量,原点的坐标为(o, o)。从原点开始,x 的值随着往右移而增加,y 的值随着往上移而增加。

图形库中的坐标可以以绝对坐标或相对坐标的形式出现。

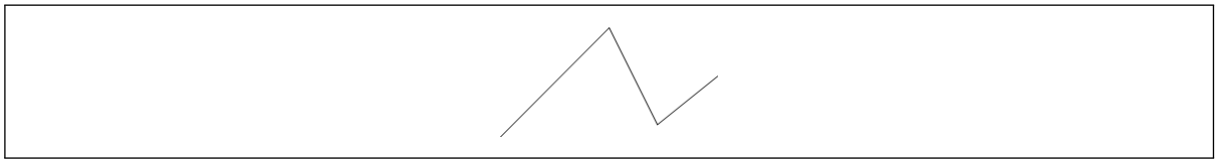
- 绝对坐标用相对于原点的坐标值指定点在窗口中的位置。
- 相对坐标用该点与最后一次指定的位置在每根轴上的距离来指定某一位置。也就是说,从最后一次指定的位置沿着 x 轴、y 轴移动到当前点所经过的距离就是两个点之间所确定的相对坐标。

如果要将某些点连成一条线,标准的方法就是先用绝对坐标指定第一个点,然后用相对坐标的方式指定连接线。这也是几何学中由点到线的原理,图形库中的坐标形式可以表示如下:



在绘图过程中,可以想象成在覆盖在屏幕上的透明图画纸上有一支笔。可以用绝对坐标把笔移到屏幕上的任何位置。一旦到了该位置,可以从当前位置开始,将笔连续在图形纸上移动一个相对位置,画出一条直线。从这个位置开始,又可以画一条从上一个终点开始的直线。

注意,在绘画艺术中,把下面的图案也看成是一条直线,因为它没有折断,而在几何学中,这个图案被认为是由三条线断组成,或者说是三条头尾相连的线组成。



graphics.h 接口仅导出了一些画图函数和过程。为了画出复杂的图片,就需要设计更强大的功能。

graphics 图形库包括下列函数:

1. InitGraphics():初始化图形软件包。
2. MovePen(x, y):把笔移到某一绝对位置。

3. DrawLine(dx, dy)：用相对一坐标画一条线。
4. DrawArc(r, start, sweep)：通过指定半径和两个角度画一段圆弧。
5. GetWindowWidth()：返回图形窗口的宽度。
6. GetWindowHeight()：返回图形窗口的高度。
7. GetCurrentX()：返回笔当前的 x 坐标。
8. GetCurrentY()：返回笔当前的 y 坐标。

这些函数提供了在图形窗口中画一张简单图形所需的功能。为了了解如何使用这些函数，就需要进一步阅读接口中提供的每个函数的说明文档。

图形库接口包含在头文件 graphics .h 中，为了理解这个头文件，需要首先阅读开始的注释，研究该文件，了解它的结构，通常是将接口作为参考指南。对于每一个新引入的函数，都应该查找接口中的相应条目，看看它们是否有意义。

```
/*
 * File: graphics.h
 * Version: 1.0
 * Last modified on Mon Jun 6 11:03:27 1994 by eroberts
 * ----------------------------------------------------
 * This interface provides access to a simple library of
 * functions that make it possible to draw lines and arcs
 * on the screen. This interface presents a portable
 * abstraction that can be used with a variety of window
 * systems implemented on different hardware platforms.
 */

#ifndef _graphics_h
#define _graphics_h

/*
 * Overview
 * --------
 * This library provides several functions for drawing lines
 * and circular arcs in a region of the screen that is
 * defined as the "graphics window." Once drawn, these
 * lines and arcs stay in their position, which means that
 * the package can only be used for static pictures and not
 * for animation.
 *
 * Individual points within the window are specified by
 * giving their x and y coordinates. These coordinates are
 * real numbers measured in inches, with the origin in the
 * lower left corner, as it is in traditional mathematics.
 *
 * The calls available in the package are listed below. More
 * complete descriptions are included with each function
 * description.
 *
 *  InitGraphics();
 *  MovePen(x, y);
 *  DrawLine(dx, dy);
 *  DrawArc(r, start, sweep);
 *  width = GetWindowWidth();
 *  height = GetWindowHeight();
 *  x = GetCurrentX();
 *  y = GetCurrentY();
 */
```

```
/*
 * Function: InitGraphics
 * Usage: InitGraphics();
 * ----------------------
 * This procedure creates the graphics window on the screen.
 * The call to InitGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output. In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics(void);

/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * ---------------------
 * This procedure moves the current point to the position
 * (x, y), without drawing a line. The model is that of
 * the pen being lifted off the graphics window surface and
 * then moved to its new position.
 */

void MovePen(double x, double y);

/*
 * Function: DrawLine
 * Usage: DrawLine(dx, dy);
 * ------------------------
 * This procedure draws a line extending from the current
 * point by moving the pen dx inches in the x direction
 * and dy inches in the y direction. The final position
 * becomes the new current point.
 */

void DrawLine(double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point. The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle. This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on. The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees. If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle. If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point. If sweep is negative, the arc is drawn
 * clockwise from the current point. The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
```

```
 *
 * Examples:
 *  DrawArc(r, 0, 360) Draws a circle to the left of the
 *                     current point.
 *  DrawArc(r, 90, 180) Draws the left half of a semicircle
 *                     starting from the 12 o'clock position.
 *  DrawArc(r, 0, 90) Draws a quarter circle from the 3
 *                     o'clock to the 12 o'clock position.
 *  DrawArc(r, 0, -90) Draws a quarter circle from the 3
 *                     o'clock to the 6 o'clock position.
 *  DrawArc(r, -90, -90) Draws a quarter circle from the 6
 *                     o'clock to the 9 o'clock position.
 */

void DrawArc(double r, double start, double sweep);

/*
 * Functions: GetWindowWidth, GetWindowHeight
 * Usage: width = GetWindowWidth();
 *      height = GetWindowHeight();
 * ----------------------------------------
 * These functions return the width and height of the graphics
 * window, in inches.
 */

double GetWindowWidth(void);
double GetWindowHeight(void);

/*
 * Functions: GetCurrentX, GetCurrentY
 * Usage: x = GetCurrentX();
 *      y = GetCurrentY();
 * ----------------------------------
 * These functions return the current x and y positions.
 */

double GetCurrentX(void);
double GetCurrentY(void);

#endif
```

graphics.h 接口包含一些固定的行，这些行在每个接口中都存在。

首先，在初始的注释后面有以下行：

```
#ifndef _graphics_h
#define _graphics_h
```

在接口中的最后一行是：

```
#endif
```

这些行的作用与编译器有关，可以给编译器指示某些行为，但它们对理解接口是如何工作的却毫无用处。

接口的其余部分仅由注释和函数原型组成，其中绝大部分是注释。即使编译器会忽略这些注释，但它们仍是接口中最重要的部分。

接口实际的读者不是编译器，而是准备编写客户代码的程序员，而注释的作用就是帮助程序员全面理解这个抽象并使用接口提供的功能。

48.1.1 InitGraphics

graphics.h 接口中的第一个过程是 InitGraphics。

正如接口中的注释所指出的，InitGraphics 过程用于初始化图形库，必须在软件包中的其他函数和在屏幕上显示任何输出之前调用它。

库软件包需要一些初始化是很普遍的，因此当使用一个接口时先把它通读一遍，看看是否需要初始化是一个比较好的策略。

48.1.2 MovePen,DrawLine

函数 MovePen 和 DrawLine 是图形库提供的主要的画线工具。

现在，我们从点 (o.5, o.5) 向上画一条一英寸的直线。注意，在任何使用图形库的主程序中，第一步总是初始化绘图软件包。

```
InitGraphics();
```

要画一条线，首先要把笔移到 (0.5, 0.5)。

```
MovePen(0.5, 0.5);
```

从这里开始，所有要做的就是画一条在 x 轴上不变，在 y 轴上向上移一英寸的线。

```
DrawLine(0.0, 1.0);
```

下面就是在屏幕上画一条直线的完整示例程序。

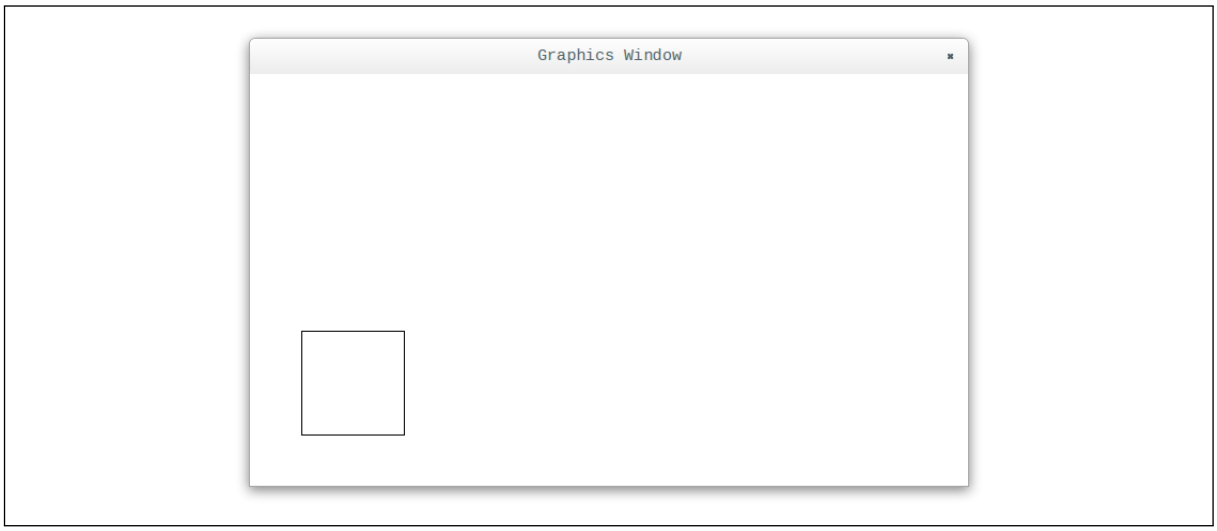
```
/*
* File:oneline.c
* ----------------
* This program draws a single straight line.
*/

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

main()
{
 InitGraphics();
 MovePen(0.5, 0.5);
 DrawLine(0.0, 1.0);
}
```

确保任何使用图形库的程序的第一行要调用初始化函数。作为一条规则，应该牢记库通常需要初始化，所以需要检查每一个接口，看看是否需要进行初始化。

如果要画一个正方形而不是一条直线，可以再在程序 online.c 中加入三个 DrawLine 调用，主程序变为：

```
main()
{
 InitGraphics();
 MovePen(0.5, 0.5);
 DrawLine(0.0, 1.0);
 DrawLine(1.0, 0.0);
 DrawLine(0.0, -1.0);
 DrawLine(-1.0, 0.0);
}
```

注意，每条线都从上一条线的终点开始。这个行为与笔沿着图形窗口表面移动的概念抽象是一致的。



48.1.3 DrawArc

图像库提供的另一个画图功能是 DrawArc，其原型

```
void DrawArc(double r, double start, double sweep);
```

DrawArc 函数用来画组成一个圆的弧。然而通过这个原型并不能确切地了解这个函数要做什么。要得到完整的理解，应该参考 Graphics.h 接口中的注释。

DrawArc 注释中的第一个语句说明：这条弧从笔的当前位置开始。这意味着必须先调用 MovePen 将笔移到所希望画的弧的开始位置，这和开始画一条直线时所做的一样。

这个注释还给出了一个非常重要的信息：角度指的是什么，角度用什么来衡量。作为一个客户，需要知道这些成功使用函数的信息。注释的最后提供了 5 个实例，说明如何使用 DrawArc。这样的实例对客户非常有帮助，因为有一个作为模型的实例，调用函数就比较容易。例如，此注释文档建议可以调用

```
DrawArc(r, 0, 360);
```

在当前点的左边画一个完整的圆。

这条弧半径为 r，从 0 度开始，即 3 点钟的位置。它延伸 360 度，因此创建了一个完整的圆。弧的开始位置是调用时笔的当前位置。相对于所画的圆的位置，这个位置在最右边，这个圆位于左边。

基于上述讨论，应该很容易编写一个画半径为半英寸，圆心在点 (1, 1) 的圆。所需要做的只是将笔移到圆最右边的起始点，然后调用 DrawArc。主程序为：

```
main()
{
  InitGraphics();
  MovePen(1.5, 1.0);
  DrawArc(0.5, 0, 360);
}
```

graphics.h 接口最后导出的四个函数不直接影响图形窗口，但返回有关图形窗口的信息。函数 GetWindowWidth 和 GetWindowHeight 返回图形窗口的尺寸，以英寸为单位。

下面的语句产生一条跨越整个图形窗口的对角线。

```
MovePen(0, 0);
DrawLine(GetWindowWidth(),GetWindowHeight());
```

这些函数也可以帮客户找出屏幕的中心点的坐标，这样可以使所画的图位于窗口的中心，因此可以用下面的语句将笔移到屏幕的中心。

```
MovePen(GetWindowWidth() / 2, GetWindowHeight() / 2);
```

这里，中点的 x 轴的位置是屏幕宽度的一半，中点的 y 轴位置是屏幕高度的一半。

最后，函数 GetCurrentX 和 GetCurrentY 返回笔当前位置的 x 轴和 y 轴的坐标，通过这些函数可以继续开发一些高级的函数。

48.1.4 DrawBox

在使用图形库函数绘图时，可以看到画方框和圆是最常用的操作。

这些基本的绘图操作通常比较简单且乏味，用户完全可以将基本的操作的函数组成一些画方框和圆的工具，而且 C 语言也提供了自定义函数的功能，这些函数同样也可以作为 graphics.h 的接口的一部分。

DrawBox 的作用是沿着坐标轴的方向画一个矩形框。在设计的思考过程中，开发 DrawBox 函数的第一步是定义它的原型，接下来就要考虑其包含的参数。

确定所需要的参数的最有用的策略是确定这个实现需要什么信息，因而不能只给出一个等价于自然语言的命令：“画一个矩形”。实现应该知道矩形有多大，把它放在屏幕的什么地方，而获取这些信息的常用方法是由客户用参数的形式提供。

尽管如此，设计 DrawBox 过程还是有多种方法。一种可能的设计方法是仅用两个参数（width 和 height）指出方框的大小。为了指出方框的位置，应该调用 MovePen。调用了 MovePen 后，就可以以此位置为参照来画方框。

下面的语句实现的功能是在位置 (x, y) 画一个框，但这还不是最终的设计。

```
MovePen(x, y);
DrawBox(width, height); /*这个实例不是最终的设计 */
```

另一种 DrawBox 的设计是采用四个参数，x、y、width 和 height，它将设定位置和设定大小组合在一起。

```
DrawBox(x, y, with, height);
```

这里因为第二种形式对调用程序来说更方便，通常都采用第二种形式，但两种设计方法都是可行的。

除了确定所需的参数个数外，还必须指明前两个参数的解释。

在 (x, y) 位置画一个方框的意义是什么？方框还没有一个明确的起始点。相对于矩形框，点 (x, y) 在哪里？对某些应用来说，比较方便的方法是使点 (x, y) 表示方框的中心。

然而，更常用的策略是将方框的起始点定义在它的左下角，就如同图形窗口的左下角是整个坐标系的原点一样，所以点 (x, y) 指出了起始的位置。不管如何定义点 (x, y) 和方框的关系，需要做的主要工作是确保函数的文档使你的设计决策显得很清晰。

综合上面的构思，得到了下面关于 DrawBox 的一种可能的原型。

```
void DrawBox(double x, double y, double width, double height);
```

其中，x 和 y 指出了方框的起始位置，width 和 height 指出了它的大小。由于这个过程是自己创建的，而不是库的一部分，所以还需要定义它的实现。

这个实现由按参数值画四条线所必需的步骤组成，可以通过 DrawLine 函数来画线，而且可以将其扩展为一个画方框的程序，现在所需要做的只是将直接的坐标值转换为更一般的基于参数的形式。

```
void DrawBox(double x, double y, double width, double height)
{
 MovePen(x, y);
 DrawLine(o, height);
 DrawLine(width, 0);
 DrawLine(0, -height);
 DrawLine(-width, 0);
}
```

在 DrawBox 的实现中，先将绘图点移动到方框的原点，然后绘制形成该方框所需的四条线段，因而通过编写这个过程，就可以改变在屏幕上画方框的程序的实现。

用左下角作为 DrawBox 的原点并不妨碍用不同的原点写其他程序。例如，可以定义一个函数 DrawCenteredBox，它的前两个参数指出方框的中心而不是它的角。如果已经定义了 DrawBox，这个新功能 DrawCenteredBox 可以用下面的语句实现。

```
void DrawCenteredBox(double x, double y, double width, double height)
{
 DrawBox(x-width / 2, y -height / 2, width, height);
}
```

然而，在设计方案中尽可能保持一致是很重要的，用同一个模型将使程序员本人或阅读程序的其他人更容易理解程序的执行过程。

通过对 DrawBox 进行扩展，从而可以达到用某些位置而不是左下角为原点进行画图的目的。

```
/*
 * File: drawbox.c
 * ---------------
 * This program draws a box on the screen.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/* Function prototypes */

void DrawBox(double x, double y, double width, double height);

/* Main program */

main()
{
 InitGraphics();
 DrawBox(0.5, 0.5, 1.0, 1.0);
}

/*
 * Function: DrawBox
 * Usage: DrawBox(x, y, width, height)
 * -----------------------------------
 * This function draws a rectangle of the given width and
 * height with its lower left corner at (x, y).
 */

void DrawBox(double x, double y, double width, double height)
{
 MovePen(x, y);
 DrawLine(0, height);
 DrawLine(width, 0);
 DrawLine(0, -height);
 DrawLine(-width, 0);
}
```

48.1.5 DrawCenteredCircle

在绘图过程中，可以定义一个画完整的圆的函数，接下来实现一个相对于圆心的绘制圆的函数 DrawCenteredCircle。

这个函数需要三个参数：圆心的 x 和 y 轴位置以及它的半径 r，于是就可以得到函数 DrawCenteredCircle 的原型。

```
DrawCenteredCircle(double x, double y, double r);
```

DrawCenteredCircle 可以用下面的语句实现。

```
DrawCenteredCircle(double x, double y, double r)
{
 MovePen(x + r, y);
 DrawArc(r, 0, 360);
}
```

在定义自己的工具的过程中可以看到，虽然有些功能可以用接口中的库函数来实现，但往往使用自己建立的工具更适合工作需要，比如 DrawCenteredCircle 就比 DrawArc 裁剪得适合用户需求。

首先，完整的圆在图形中相当普通，比一段弧出现的机会要多得多。另一方面，用高级函数可以使用户从记住 DrawArc 如何解释角度的过程中解脱出来，而这些问题在画一个圆的时候是不需要考虑的，从而也揭示了使用 DrawCenteredCircle 进行绘图时所提供的方便性和简单性，这两点在程序设计中都有实用价值。

与大多数使用过程的情况一样，库函数和自定义过程都可以在不只是在一种情况下使用。

这样最大的好处是一旦定义了一个新的过程，就可以反复地使用它，而正是这种重复使用已经写好的步骤的能力使过程非常有用。例如，要在图形窗口中画一行正方形，而不只是用 drawbox.c 生成一个正方形，那么可以连续调用 DrawBox，或者把它放在一个 for 循环中，在每次循环中画一个正方形。

48.1.6 Coordinates

MovePen 过程使用绝对坐标指出线的起点，然后用 DrawLine 通过相对坐标画线。对某些应用，将笔移到相对于原始位置的新位置，但不要画线也是很有用的。反过来，有时画一条到特定的绝对坐标位置的线也是很有用的。

函数 GetCurrentX 和 GetCurrentY 提供了写一个 MovePen 的相对版本和 DrawLine 的绝对版本的方便性，新的函数被称为 AdjustPen 和 DrawLineTo。

```
/*
* Function:AdjustPen
* Usage:AdjustPen(dx, dy);
* ---------------------------------
* This procedure adjusts the current point by moving it dx inches from ite current x
* coordinate and dy inches from its current y coordinate. As with MovePen, no line
* is actually draw.
*/

void AdjustPen(double dx, double dy)
{
 MovePen(GetCurrentX() + dx, GetCurrentY() + dy);
}

/*
* Function:DrawLineTo
* Usage:DrawLineTo
* -------------------------
* This function is like DrawLine, except that it uses the absolute coordinate of the
* endpoint rather than the relate displacement from the current point.
```

```
*/

void DrawLineTo(double x, double y)
{
 DrawLine(x -GetCurrentX(), y -GetCurrentY());
}
```

和其他的自定义过程(比如 DrawBox 和 DrawCenteredCircle)一样,这些函数并不是图形库的一个部分。如果要在程序中使用它们,必须拷贝它们的定义。

48.2 Contants

为了逐步加深对图形库中的函数的理解,我们考虑如何解决一个较大的问题——利用图形库绘制复杂的图形,比如房子。

虽然图中会有许多独立的部分,但它仅由两种最基本的图形元素组成:(1)直线,用于房子的框架、门和窗格;(2)圆,仅用于门把手。

如果把这些线和圆按正确的大小和位置放在一起,就可以创建一张完整的图。此外,几乎所有的直线都是用来形成一个方框。

在利用图形库进行绘图时,首先需要将构思好的图形划分成独立的部分,直至将其分为由两种最基本的图形元素组成:线和圆弧。

这里假设要画一幢房子的草图,在开始写实际的程序之前,要注意到这栋特殊的房子有许多定义它形状的特征。例如,该房子宽 3.0 英寸。从地到阁楼的距离是 2.0 英寸,从阁楼到屋顶的最高处为 o.7 英寸。门是一个 0.4×0.7 英寸的矩形,每个窗格也都是 0.2×0.25 英寸的矩形。

为避免因为数字太多而把程序搞乱,最好给这些量分别命名并将其明确地标识,从而便于在程序中使用这些名字,这里在房子草图中会用到下面的常量。

```
#define HouseHeight 2.0
#define HouseWidth 3.0
#define AtticHeight 0.7

#define DoorWidth 0.4
#define DoorHeight 0.7
#define DoorknobRadius 0.03
#define DoorkonbInset 0.07

#define PaneHeight 0.25
#define PaneWidth 0.2

#define FirstFloorWindows 0.3
#define SecondFloorWindows 1.25
```

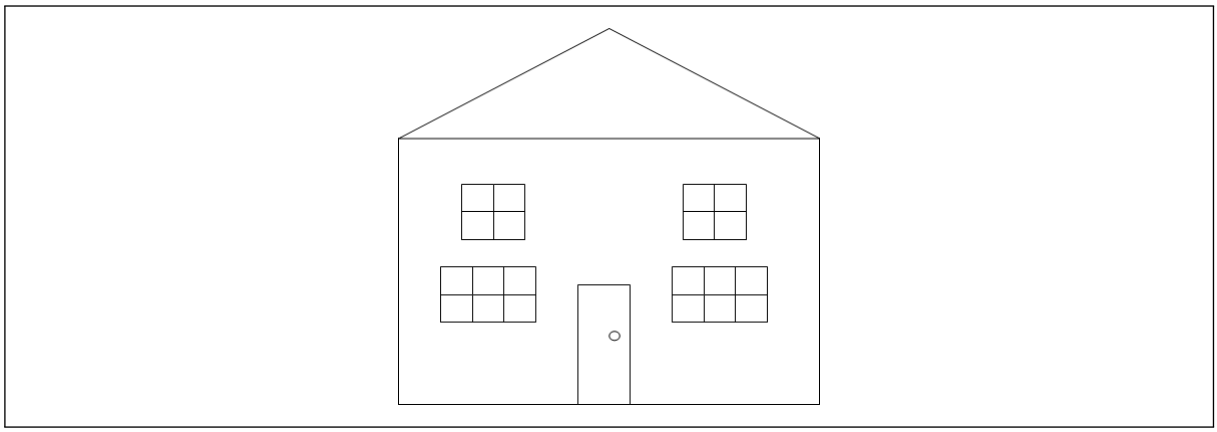
这些值都是实数,表示英寸,名字描述了它们在图中的物理含义。在程序中,用 #define 等定义房子的结构元素,而且用符号化的名字表示这些值使得改变它们的大小更方便,例如在你要使房子更宽一些,窗子更大一些时。

48.3 Decomposition

在使用过程化程序设计方法时,处理大的程序设计问题的最好方法是使用逐步求精策略,将整个问题分解成一组较小的问题。

为了将此策略应用到画房子这个问题,可以从最综合的层次开始:要画一栋房子。

首先,开始实施绘画工作后,要给这个工作取一个名字,如 DrawHouse,并定义它为一个过程,从而将实现 DrawHouse 过程作为绘制房子工程的第一个子问题。



为了完成这个实现，可以将整个问题分解成较小的部分：画轮廓、门和窗户。这些工作是分解过程中的下一层子问题。执行这个策略直到所有的子问题都简化为能用已有工具解决的最简单的操作。

然而，和 DrawBox 过程一样，还要确定 DrawHouse 过程是否需要参数。在初始步骤中，已将房子的大小定义为常数。另外还要指出房子的位置，因此 DrawHouse 过程似乎取 x 和 y 两个坐标值是合适的，这两个坐标值指出了房子在图形窗口中的位置。

为了与 DrawBox 过程一致，使用坐标值指出房子的左下角的位置，因此可以得到 DrawHouse 过程的原型。

```
void DrawHouse(double x, double y);
```

调用此过程将指示计算机画一栋房子，它的左下角位于点 (x, y)。

完成程序原型定义后，接下来就可以开始来完成主程序。首先需要做的是指出房子应该在屏幕上出现的位置。例如，假设要把房子画在图形窗口的中间，可以用函数 GetWindowWidth 和 GetWindowHeight 找出窗口中心的坐标。

现在声明变量 cx 和 cy，可以用如下语句取得中心的坐标。

```
cx = GetWindowWidth() / 2;
cy = GetWindowHeight() / 2;
```

在 DrawHouse 过程中，图形本身是相对于左下角而不是中心的位置绘制的，现在需要做的是把这两个位置关联起来。

房子的宽为 HouseWidth，于是房子的左面的边离中心的位置是这个距离的一半。如果房子左边的边的坐标为 cx – HouseWidth/2，那么房子的中心是在屏幕的中心。同样地，可以对 y 坐标重复同样的处理。唯一的差别在于房子的总高度是矩形和屋顶高度的和，所以房子左下角的 y 坐标为 cy – (HouseHeight + AtticHeight) / 2。

从而现在就得到了房子左下角的坐标，此时可以完成 main 的实现。

```
main()
{
  double cx, cy;

  InitGraphics();
  cx = GetWindowWidth() / 2;
  cy = GetWindowHeight() / 2;
  DrawHouse(cx –HouseWidth / 2, cy –(HouseHeight + AtticHeight) / 2);
}
/*这个定义完成最高一级的分解。*/
```

根据逐步精化的方法，在实现 DrawHouse 过程时，首先要了解其中最基本的操作。概括地讲，DrawHouse 过程看上去应该有如下形式：

```
void DrawHouse(double x, double y)
```

```
{
 DrawOutline( . . .);
 DrawDoor( . . . );
 DrawWindows( . . . );
}
```

在该过程中要完成的工作是确定其中的各个子过程的参数，其中 DrawOutline、DrawDoor 和 DrawWindows 不能访问 DrawHouse 的局部变量 x 和 y 的值，因此必须把坐标信息传给每一个过程。然而，选择所要传递的确切值也是需要仔细考虑的。

轮廓是从房子的左下角开始，因此这里的 x 和 y 值与局部变量的 x 和 y 值完全一样。对于门来讲，需要计算门本身的坐标，然后把这些坐标传给 DrawDoor。

由于相对于房子的框架，需要画几扇窗，DrawWindows 函数要将房子的坐标作为参数，虽然作为实现的一部分，它将为每一扇窗计算更确切的坐标。

如果按上述建议实现 DrawHouse 的话，它看上去应该是：

```
void DrawHouse(double x, double y)
{
 DrawOutline(x, y);
 DrawDoor(x + (HouseWidth - DoorWidth) / 2, y);
 DrawWindows(x, y);
}
```

48.4 Bottom-up

在寻找一个解决大问题的方案时，除了使用逐步求精的策略，另一种非常有用的策略是试图在一个大问题的不同部分中找出共同的部分，从而可以对这些共同的部分应用一个解决方案。

本质上，这种方法由确定完成该任务最好的通用工具组成。例如，如果能用一个完成特定操作的过程解决问题的几个部分，那就值得创建这个过程。

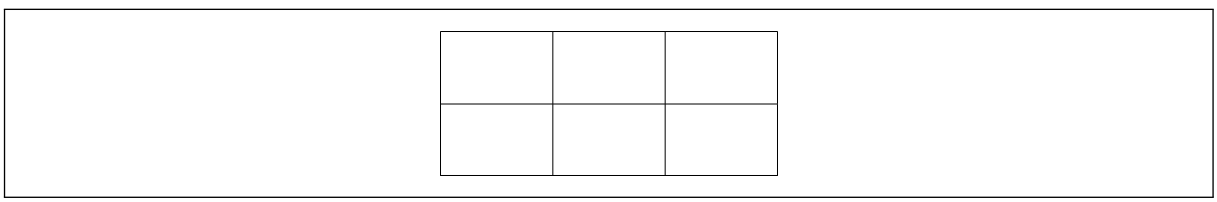
从这个观点考虑画房子的问题，可以将问题的解决方案进一步简化，而且有几个工具可能是正好需要的，其中有一些可能已经完成了。例如，房子的轮廓是方框，门框和窗也是如此，因此，DrawBox 工具应该是非常有用的。门把手是个圆，则可以使用 DrawCenteredCircle。

接下来就要考虑其他有用的工具了。例如，屋顶是个三角形。尽管图中只有一个三角形，但写一个 DrawTriangle 过程也是值得的，因为以后还可以把它用在其他程序中。

在整个问题的求解过程中，更重要的是注意窗的特定结构，可以考虑进一步抽象写一个更通用的过程以便画不同的窗。

为了设计适合于画窗的工具，尽可能考虑将这个问题通用化是不会错的，只有工具越通用，才能在各种情况下越容易地使用它们。保证所创建的工具能够广泛应用的一种方法是跳出当前问题的局限，而从一个更高的、更抽象的层次上认识所需要的操作。

在房子这个问题上，下面的图描述了一个有几个窗格的窗。然而，当我们集中于这个图本身时，此时所见的是由两行组成的矩形网格，每行包括三个方格，因此可以定义一个画一个矩形网格的过程 DrawGrid，就可以用这个过程画每一组窗口。



在确定 DrawGrid 需要的参数时，为了达到必要的通用性，必须保证 DrawGrid 过程不涉及房子问题的特性。

用常量 PaneWidth 和 PaneHeight 会使过程只适用于本例中所指的房子的情况，最好是让调用程序将网格中的每一方框的宽度和高度作为参数来传递，从而使调用程序知道这是画一扇窗，能为这个特定的应用提供 PaneWidth 和 PaneHeight。这个过程本身只是画方框。

除了网格中每个方框的高度和宽度外，DrawGrid 也要知道完整的网格的坐标。为了与其他工具保持一致，DrawGrid 将这些坐标解释为网格的左下角。最后，该过程必须知道网格中的列数和行数，因此可以确定 DrawGrid 的原型。

```
void DrawGrid(double x, double y, double width, double height,
     int columns, int rows);
```

在 DrawGrid 中借助 DrawBox，并且由一对嵌套的 for 循环组成，它们在每一行中对每一列调用 DrawBox。

```
/*
* Function:DrawGrid
* Usage:DrawGrid(x, y, width, height, columns, rows);
* ---------------------------------------------------------------
* DrawGrid draws rectangles arranged in two-dimensional grid.
* As always, (x, y) specifies the lower left corner of the figure.
*/
void DrawGrid(double x, double y, double width, double height,
      int columns, int rows)
{
  int i, j;

  for(i = 0; i < columns; i ++){
    for(j = 0; j < rows; j++){
      DrawBox(x + i * width, y + j * height, width, height);
    }
  }
}
```

给定了 DrawGrid 的实现之后，可以通过用相应的参数调用 DrawGrid 来构造每扇窗口的模型。

尽管通常自顶向下设计问题是最好的方法，但自底向上实现问题通常也是最好的。先实现低层次的工具使它更易于调试程序的每个模块，这通常比一次性调试所有的程序更容易。这个策略称为自底向上的实现(bottom-up implementation)。

在分析完问题并就其中的共同的模式进行处理后，程序中剩下的部分是一个逐步求精的最直接的过程。

```
/*
 * File: house.c
 * -------------
 * This program draws a simple frame house.
 */

#include <stdio.h>

#include "genlib.h"
#include "graphics.h"

/*
 * Constants
 * ---------
 * The following constants control the sizes of the
 * various elements in the display.
 */
```

```
#define HouseHeight  2.0
#define HouseWidth   3.0
#define AtticHeight  0.7

#define DoorWidth    0.4
#define DoorHeight   0.7
#define DoorknobRadius 0.03
#define DoorknobInset 0.07

#define PaneHeight   0.25
#define PaneWidth    0.2

#define FirstFloorWindows 0.3
#define SecondFloorWindows 1.25

/* Function prototypes */

void DrawHouse(double x, double y);
void DrawOutline(double x, double y);
void DrawWindows(double x, double y);
void DrawDoor(double x, double y);
void DrawBox(double x, double y, double width, double height);
void DrawTriangle(double x, double y, double base, double height);
void DrawCenteredCircle(double x, double y, double r);
void DrawGrid(double x, double y, double width, double height,
           int columns, int rows);

/* Main program */

main()
{
  double cx, cy;

  InitGraphics();
  cx = GetWindowWidth() / 2;
  cy = GetWindowHeight() / 2;
  DrawHouse(cx - HouseWidth / 2, cy - (HouseHeight + AtticHeight) / 2);
}

/*
 * Function: DrawHouse
 * Usage: DrawHouse(x, y);
 * -----------------------
 * This function draws a house diagram with the lower left corner
 * at (x, y). This level of the function merely divides up
 * the work.
 */

void DrawHouse(double x, double y)
{
  DrawOutline(x, y);
  DrawDoor(x + (HouseWidth - DoorWidth) / 2, y);
  DrawWindows(x, y);
}

/*
 * Function: DrawOutline
 * Usage: DrawOutline(x, y);
```

```
 * -------------------------
 * This function draws the outline for the house, using (x, y)
 * as the origin. The outline consists of a box with a triangle
 * on top.
 */

void DrawOutline(double x, double y)
{
  DrawBox(x, y, HouseWidth, HouseHeight);
  DrawTriangle(x, y + HouseHeight, HouseWidth, AtticHeight);
}

/*
 * Function: DrawDoor
 * Usage: DrawDoor(x, y);
 * ----------------------
 * This function draws a door, with its doorknob. As usual,
 * (x, y) specifies the lower left corner of the door.
 */

void DrawDoor(double x, double y)
{
  DrawBox(x, y, DoorWidth, DoorHeight);
  DrawCenteredCircle(x + DoorWidth - DoorknobInset, y + DoorHeight / 2, DoorknobRadius);
}

/*
 * Function: DrawWindows
 * Usage: DrawWindows(x, y);
 * -------------------------
 * This function draws all the windows for the house,
 * taking advantage of the fact that the windows are all
 * arranged in two-dimensional grids of equal-sized panes.
 * By calling the function DrawGrid, this implementation
 * can create all of the window structures using a single
 * tool.
 */

void DrawWindows(double x, double y)
{
  double xleft, xright;

  xleft = x + HouseWidth * 0.25;
  xright = x + HouseWidth * 0.75;
  DrawGrid(xleft - PaneWidth * 1.5, y + FirstFloorWindows, PaneWidth, PaneHeight, 3, 2);
  DrawGrid(xright - PaneWidth * 1.5, y + FirstFloorWindows, PaneWidth, PaneHeight, 3, 2);
  DrawGrid(xleft - PaneWidth, y + SecondFloorWindows, PaneWidth, PaneHeight, 2, 2);
  DrawGrid(xright - PaneWidth, y + SecondFloorWindows, PaneWidth, PaneHeight, 2, 2);
}

/*
 * Function: DrawBox
 * Usage: DrawBox(x, y, width, height)
 * -----------------------------------
 * This function draws a rectangle of the given width and
 * height with its lower left corner at (x, y).
 */
```

```
void DrawBox(double x, double y, double width, double height)
{
 MovePen(x, y);
 DrawLine(0, height);
 DrawLine(width, 0);
 DrawLine(0, -height);
 DrawLine(-width, 0);
}

/*
 * Function: DrawTriangle
 * Usage: DrawTriangle(x, y, base, height)
 * ---------------------------------------
 * This function draws an isosceles triangle (i.e., one with
 * two equal sides) with a horizontal base. The coordinate of
 * the left endpoint of the base is (x, y), and the triangle
 * has the indicated base length and height. If height is
 * positive, the triangle points upward. If height is negative,
 * the triangle points downward.
 */

void DrawTriangle(double x, double y, double base, double height)
{
 MovePen(x, y);
 DrawLine(base, 0);
 DrawLine(-base / 2, height);
 DrawLine(-base / 2, -height);
}

/*
 * Function: DrawCenteredCircle
 * Usage: DrawCenteredCircle(x, y, r);
 * -----------------------------------
 * This function draws a circle of radius r with its
 * center at (x, y).
 */

void DrawCenteredCircle(double x, double y, double r)
{
 MovePen(x + r, y);
 DrawArc(r, 0, 360);
}

/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * ----------------------------------------------------
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid. As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid(double x, double y, double width, double height, int columns, int rows)
{
 int i, j;

 for (i = 0; i < columns; i++) {
  for (j = 0; j < rows; j++) {
```

```
        DrawBox(x + i * width, y + j * height, width, height);
      }
    }
  }
```

第 49 章

Development

49.1 Overview

在软件行业里，软件工程在软件开发过程中起着重大的作用，很少有人完整地完成一个独立的大型程序。

程序员都是按团队工作的，现在大规模的应用都不是任何一个人可以单独完成的，因而使许多程序员一起工作是现代软件工程最大的挑战。

在设计程序时，往往会把大程序分解成独立的模块，然后设计这些模块，使它们能作为其他应用的库。

在现代程序设计中，编写一个有意义的程序不可能不调用库函数。按照这个观点，每个程序员都应该相当擅长调用函数，而且每个程序员都应该完善他所编写的每一个函数，使之能放到一个库里，从而被以后的程序反复使用，而不仅仅是将函数作为程序的一部分，然后仅在主程序或作为同一程序文件的一个部分的其他函数来互相调用。

设计好的库并用好它会产生巨大的作用，这个作用的关键部分来自于理解接口的概念。通常，接口（interface）指的是两个独立的实体之间的公共边界，例如池塘的表面就是水和空气的接口。在程序设计中，接口是一个概念性的边界，而不是一个物理的边界。

接口是库的实现和使用库的程序之间的边界。当调用库中的函数的时候，信息会穿越这个边界。

为了理解什么是接口以及它是如何工作的，重点在于能够读懂一个已有的接口，而不是从头开始设计一个接口。可以以写作作为类比，在试图写小说前，作者通常花许多年去读小说。在读小说的时候，他们学到了许多有关小说的体裁，并逐步了解好小说应具备哪些特性。同样地，通过研究已有接口的基本结构，可以更加深入地理解库是如何工作的，它们包含什么内容以及如何用接口描述它们。要做到这一点，最好的方法是研究一个已有的实例。

接口是库和它的使用者之间交换信息的媒介，并且接口还给出了交换信息的结构。从概念上讲，程序设计的接口也表示有关边界特性的共识，提供了库的创建者和使用者之间所需的临界信息。

提供数学运算函数的 math 库中定义了一些函数，使用 math 库的程序员可以调用 sqrt 函数计算平方根，而不需要指出计算平方根的每个步骤。这些步骤是平方根函数实现的组成部分，它是由创建 math 库的程序员设计并编写实现的。

通过数值算法可以了解到，可以使用两种可行的方法实现 sqrt 函数——牛顿法和 Taylor 级数展开法。库的实现者正是使用这些方法中的某一种，或其他能计算出正确结果的算法来实现 sqrt 函数的。

掌握如何调用和实现函数是接口设计的关键，并且认识到调用函数和实现函数完全不相关也很关键。成功的程序员经常使用那些他们并不知道如何编写的函数。反过来，实现库函数的程序员也不能预见该函数的所有潜在应用。

为了强调实现库的程序员和使用库的程序员之间的区别，计算机科学家采用了特定的术语去讨论处于特定角色的程序员。很自然地，实现库的程序员被称为实现者（implementer）。另一方面，因为用户（user）这个词通常指的是运行程序的人，而不是写程序的人，因此调用库函数的程序员被称为这个库的客户（client）[1]。

[1]在计算机科学中，术语客户有时也指使用库的代码，有时则指写这段代码的程序员，因而可以用客户代码表示使用库的代码。

尽管客户和实现者对库有不同的看法,但他们都必须了解库设计中的某些问题。作为客户,不需要知道工作的细节,但必须知道如何调用它。而作为实现者,则不用关心客户程序员会如何使用库。但是,你必须给客户提供调用函数所需要的信息。

对库中的每一个函数,客户必须知道下列内容:

- 函数名。
- 所需的参数以及参数的类型。
- 返回的结果类型。

这些信息正好是函数原型所提供的信息,这绝不是一个巧合。因为在 C 语言中,函数的原型和它的实现是分开的,它们分别传递信息给不同的读者。

客户和实现者必须就函数的原型达成一致,这意味着这些信息是接口的一部分。相比之下,只有实现者是关心函数的实现。在接口中包含这个函数的原型,使客户可以使用该函数的过程称为导出(exporting)这个函数。

Table 49.1: 客户和实现者之间的关系

客户	接口	实现者
负责:如何使用函数	双方约定:函数原型	负责:函数是如何工作的

总之,库中的函数是由实现者编写,由客户调用,客户和实现者交汇的点就被称为接口。

要完全理解接口,还必须学会如何实现它们,但如果仅考虑 C 语言的语法和结构,并没有许多新的规则要学,因为注释和函数原型就是接口的主要组成部分。

然而,一旦有了正确的算法,后面具有挑战性的工作并不是给这个接口编码,而是设计这个接口,所以说重要的问题不是如何写一个接口,而是如何写一个好的接口。

构造一个接口的挑战在于接口的设计而不是编码,设计一个好的接口是一个微妙的问题,需要平衡许多互相冲突的设计标淮,它贯穿于所对应的库软件包的完整的设计和实现过程中。

按照所研究问题的范围,编写的接口可以是非常简单的,也可能是非常复杂的。

49.2 Design

计算机程序反映了它们所要解决的问题的复杂性。随着要用计算机解决的问题的复杂性日益增长,程序设计的过程也将变得更复杂。

在编写一个解决较大的或困难的问题的程序时,程序员必须处理大量的复杂性问题,其中有算法设计、要考虑的特殊情况、要满足用户的需求以及许多要保持正确的细节,因而要使得程序可管理,必须尽可能降低程序设计过程的复杂性。

接口提供了类似于函数和过程的做法来降低一些复杂性,但接口处于一个更高的细节层次上。

函数让它的调用程序访问一组实现某一功能的一段程序,接口让它的客户访问一组实现某一程序设计中抽象行为的函数,而接口对程序设计过程的简化程度在很大程度上依赖于它的设计。

为了设计一个有效的接口,必须在许多标准之间加以权衡。一般来讲,开发的接口应该具有以下特性:

- 同一性:一个接口应该定义一个与某一明确的主题一致的抽象。如果某一函数不适合这个主题,那么它应该定义在另一个接口中。
- 简单性:基本的实现范围本身就很复杂,接口必须对客户隐藏这个复杂性。
- 充分性:当客户要用此抽象行为时,接口必须提供足够的功能来满足他们的要求。如果在接口中缺少某些关键的操作,客户可能会决定放弃它,开发满足自己需要的功能更强的库。和简单性一样重要的是,设计者必须避免把一个接口简化得毫无价值。
- 通用性:一个设计良好的接口应该足够灵活以满足许多不同客户的需求。只完成某一用户所需功能的,包含很少操作的接口不如一个在很多场合都能用的接口有用。

- 稳定性:不管实现的方法如何改变,在一个接口中定义的函数必须保持完全相同的结构和作用。接口行为的变化会迫使客户修改它们的程序,而这将危及接口的价值。

一个定义良好的接口必须具备同一性、简单性、充分性、通用性和稳定性,有时这些标准会互相冲突,因而必须学会如何在接口的设计中达到适当的平衡。下面将详细讨论每一条标准。

49.2.1 Unified

一个良好定义的接口的最重要的特征是它表现的是同一的和一致的抽象行为。

从某一方面讲,这个标准说明同一个库中的函数应该反映相关的主题。例如,数学库是由数学函数组成,标准的 I/O 库提供了执行输入/输出的函数,图形库提供了在屏幕上绘图的函数。由这些接口输出的函数符合这些接口的目的。例如,我们不希望在 graphics.h 接口中找到 sqrt,尽管图形应用也经常调用 sqrt 计算对角线的长度,但是由于 sqrt 函数是数学函数,因此它还是更适合放在数学库中。

同一主题原则也影响库接口中的函数的设计。接口中的函数应该尽可能有一致的行为,函数工作方式上的不同将会使客户难以使用接口。

例如,图形库中的所有函数都用英寸表示坐标,角用度数表示,而如果库的实现者决定增加使用不同计量单位的函数,客户必须记住每个函数所用的计量单位。同样,图形库中的 DrawLine 和 DrawArc 函数都是假定按笔的当前位置开始绘图,这样做意味着基本的概念模型有一致的结构,可以很容易理解这个库和它的操作。

49.2.2 Simple

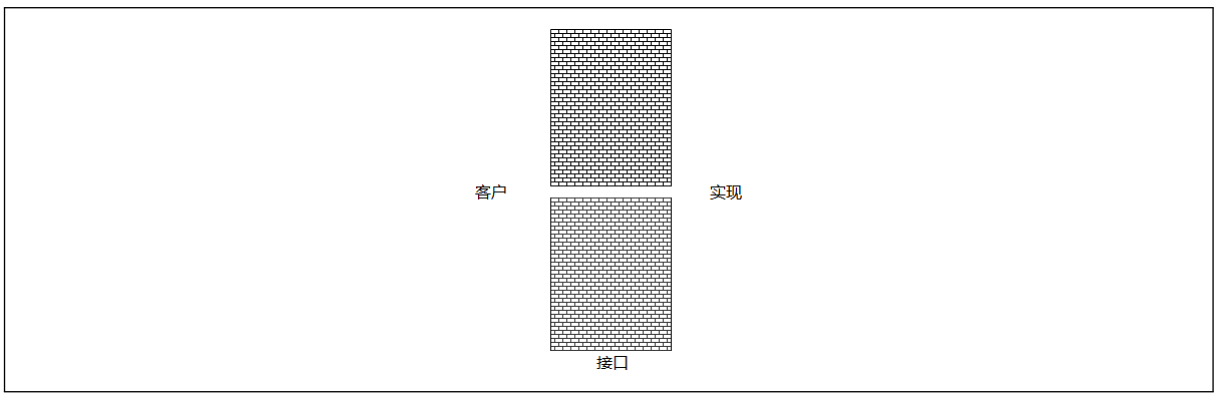
使用接口的主要目的是减少程序设计过程的复杂性,从这个意义上说,简单性是接口设计中的最希望达到的标准。

一般来讲,接口应该尽可能使用方便,基本的实现可能要通过非常复杂的操作来完成,但客户应该以简单的、更抽象的方法考虑这些操作。

从某种程度上说,接口担当着特定库抽象的参考指南。当用户想知道如何使用数学库时,可以从 math.h 接口中找出使用数学库的方法。

接口应该正好包含客户所需要知道的信息(而且没有多余的信息),因为对客户来说,获取太多的信息和获取的信息不够一样糟,多余的信息会使接口更难理解。

通常,接口的真正价值不在于它揭示了某些信息,而在于它隐藏了某些信息。当设计一个接口时,应该尽可能对客户屏蔽实现过程中的许多细节。从这方面来讲,最好认为接口不只是客户和实现者之间的通信信道,而且是分割它们的界限(如下页的图所示)。



和希腊神话中分割情人帕拉莫斯和提斯柏的墙一样,表示接口的墙有一条细缝,它允许客户和实现之间互相通信。然而,墙的主要目的是保持两边分开。

一般地,我们把接口想象为库表示的抽象的边界,所以接口也被称为抽象边界(abstraction

boundary)。理想情况下,库的实现中的所有的复杂性都位于墙右边的实现这一边。如果接口能对客户方屏蔽复杂性,那么接口就是成功的。

把细节限制在实现范畴内称为信息隐藏(information hiding),信息隐藏的原理对接口的设计有重要的实际意义。

当编写一个接口时,应该确信没有显露出实现的细节,甚至在注释中也没有显露实现的细节。特别是,如果你同时写一个接口和一个实现时,可能会不知不觉地在你的接口中记下你在写这个实现时的所有得意之处,但是一定不要这样做。

写接口主要为了供客户使用,应该仅包含客户需要知道的内容。同样,应该在接口中设计这些函数,并使它们尽可能简单。如果能减少参数个数或找到一种方法能消除特殊情况间的混乱,客户就容易理解如何使用这些函数。

此外,限制接口输出的函数个数通常是好的方法,这样客户不至于在大量的函数中迷失方向,找不到整体的意义。

49.2.3 Sufficient

简单性只是接口设计原则的一个部分。抛弃接口中所有困难的或复杂的部分,就可以很容易地设计出一个接口,但这也使得一个接口变得毫无用处。有时,客户需要完成一些具有某种内在复杂性的任务。

虽然拒绝为客户提供他们所需的工具,可以使这个接口变得较简单,但这不是一个有效的策略。接口必须提供足够的功能以满足客户的需要,因此在接口设计中,学会在简单性和完备性之间达到合理的平衡是程序设计最基本的挑战之一。

另外,在许多情况下,接口的客户不仅关心特定的函数是否可用,而且还关心实现的效率。例如,如果某一程序员正在开发一个航空控制系统,需要调用某一库接口提供的函数,这些函数必须迅速返回正确的结果,延迟返回的答案和错误的答案一样糟。

在大多数情况下,效率是实现而不是接口关心的因素。尽管如此,你还是经常会发现在设计接口本身时,考虑实现的策略是有价值的。例如,假设你有两个设计方案,如果能确定其中的一个能更容易更有效地实现,而另一个方案又没有绝对的竞争理由,选择这个设计是有意义的。

49.2.4 General

完全适合某一客户需要的接口对另外一些客户可能是毫无用处的,一个良好的库抽象要满足不同客户的需求。要做到这一点,它必须通用到足以解决很多问题,而不仅限于解决某种非常特殊的问题。

通过选择一个能在客户使用这个抽象时具有足够灵活性的设计,就能创建一个应用范围很广的接口。

确保接口保持通用性有着重要的实用意义。当在写一个程序时,经常会发现需要一个特定的工具。如果确定这个工具非常重要,值得放到一个库里,那么就需要改变当时写程序时的思维方法。当为库设计一个接口时,必须忘记第一次使自己想起要做这个工具的应用,从而为大多数普通的、可能的用户设计这样一个工具。

例如,在 graphics.h 中的 DrawGrid 工具,从当时客户的观点来看,只是需要一个为某一栋房子画窗的函数。然而,为了创建这个工具并能在接口中调用它,接口的设计者就必须考虑更一般的情况,结果产生了函数 DrawGrid,并且它可以在许多不同的情况下使用。

49.2.5 Stable

接口还有另一个对程序设计来说非常重要的特性——接口往往在很长的一段时间内是稳定的。

稳定的接口可以通过建立明确的责任分工,极大地简化维护大的程序设计系统的问题。只要接口不改变,实现者和客户可以相对自由地在抽象界限的自己一边做修改。

例如,数学库的实现者在工作的过程中发现了一个更好的计算 sqrt 函数的新算法,它节省了一半的计算时间。如果告诉使用数学库的客户这个新的 sqrt 的实现,它可以完成相同的功能,但速度更快,他们可能会非常高兴。但如果告诉他们函数的名字改变了或增加了一些新的限制,客户会非常恼怒。

为了使用“改进过的”的实现而必须让客户修改他们的程序时,由于修改程序是非常费时而且容易出错的事情,许多客户都不愿意修改他们的程序,甚至会为此牺牲一些效率。

事实上,仅当接口保持稳定时,才能简化程序维护的工作。当发现新算法或应用需求变化时,程序会频繁变化。然而,在这样的演变过程中,接口必须尽可能保持稳定。

在一个设计良好的系统中,改变实现的细节是一个简单的过程,修改时带来的复杂性位于抽象边界的实现这一端。另一方面,修改接口经常会产生全局性的巨变,导致使用该接口的每一个程序都要修改。因此,接口必须很少变化,而且改变时需要客户的参与。

然而,某些接口的变化比另外一些变化影响更大。例如,在一个接口中增加一个全新的函数通常是一个相对简单的过程,因为还没有客户使用这些函数。

使已有的程序不需要修改就可以继续运行的修改接口的方式称为扩展(extending)这个接口。如果发现需要在接口的生命周期中对接口做修改,最好通过扩展对它进行修改。

第 50 章

Pseudo-random

50.1 Introduction

50.1.1 Randomness

随机在自然科学和哲学上有着重要的地位,术语随机经常用于统计学中来表示一些定义清晰的、彻底的统计学属性(例如缺失偏差或者相关)。

一个随机的过程是一个不定因子不断产生的重复过程,但它可能遵循某个概率分布。随机与任意是不同的,因为"一个变量是随机的"表示这个变量遵循概率分布,但是任意在另一方面又暗示了变量没有遵循可限定概率分布。

随机性(Randomness)这个词是用来表达目的、动机、规则或一些非科学用法的可预测性的缺失,目前的随机性测试方法包括:

1. 频数测试:测试二进制序列中,"0"和"1"数目是否近似相等。如果是,则序列是随机的。
2. 块内频数测试:目的是确定在待测序列中,所有非重叠的长度为 M 位的块内的"0"和"1"的数目是否表现为随机分布。如果是,则序列是随机的。
3. 游程测试:目的是确定待测序列中,各种特定长度的"0"和"1"的游程数目是否如真随机序列期望的那样。如果是,则序列是随机的。
4. 块内最长连续"1"测试:目的是确定待测序列中,最长连"1"串的长度是否与真随机序列中最长连"1"串的长度近似一致。如果是,则序列是随机的。
5. 矩阵秩的测试:目的是检测待测序列中,固定长度子序列的线性相关性。如果线性相关性较小,则序列是随机的。
6. 离散傅里叶变换测试:目的是通过检测待测序列的周期性质,并与真随机序列周期性质相比较,通过它们之间的偏离程度来确定待测序列随机性。如果偏离程度较小,序列是随机的。
7. 非重叠模板匹配测试:目的是检测待测序列中,子序列是否与太多的非周期模板相匹配。太多就意味着待测序列是非随机的。
8. 重叠模板匹配测试:目的是统计待测序列中,特定长度的连续"1"的数目,是否与真随机序列的情况偏离太大。太大是非随机的。
9. 通用统计测试:目的是检测待测序列是否能在信息不丢失的情况下被明显压缩。一个不可被明显压缩的序列是随机的。
10. 压缩测试:目的是确定待测序列能被压缩的程度,如果能被显著压缩,说明不是随机序列。
11. 线性复杂度测试:目的是确定待测序列是否足够复杂,如果是,则序列是随机的。
12. 连续性测试:目的是确定待测序列所有可能的 m 位比特的组合子串出现的次数是否与真随机序列中的情况近似相同,如果是,则序列是随机的。
13. 近似熵测试:目的是通过比较 m 位比特串与 m-1 位比特串在待测序列中出现的频度,再与正态分布的序列中的情况相对比,从而确定随机性。
14. 部分和测试:目的确定待测序列中的部分和是否太大或太小。太大或太小都是非随机的。
15. 随机游走测试:目的是确定在一个随机游程中,某个特定状态出现的次数是否远远超过真随机序列中的情况。如果是,则序列是非随机的。

16. 随机游走变量测试：目的是检测待测序列中，某一特定状态在一个游机游程中出现次数与真随机序列的偏离程度。如果偏离程度较大，则序列是非随机的。

在自然与工程学里一些现象会通过随机性模型来模拟，例如混沌理论、密码学、博弈论、信息论、模式识别、机率论、量子力学、统计学和统计力学等。

在 19 世纪，物理学家使用分子的不规则行动的概念去发展统计力学，以解释热力学和气体定律的现象。另外，根据一些量子力学的标准解释，微观现象是客观地随意。换句话说，在一个所有相关的参量受控的实验中，也会出现任意变化的情况，例如我们无法预计在受控环境中放置一粒不稳定的原子衰败的时间，只能计算在指定的时间内衰败的可能性，所以量子力学计算的是机会率而非单一实验的结果。Hidden variable theories 尝试避开大自然包含不能降低的随机性，这样的理论假定在看上去任意的过程中，有些符合统计分布而暗藏的特性在幕后运作以得出结果。

在生物学中，进化论将观察到的分集性归因于随机突变。由于一些突变的基因带给了拥有它们的个体更高的存活与繁衍的机会，随机突变保留在了基因库中。生物体的特征在某种程度上是确定性地发生的（例如在基因和环境的影响下），在某种程度上是随机发生的。以皮肤色斑为例，基因与曝光量仅仅支配着人体皮肤上出现的色斑密度，而单个色斑的精确位置看来是随机决定的。

在计算机科学中，随机化算法（randomized algorithm）中使用了随机函数，且随机函数的返回值直接或者间接的影响了算法的执行流程或执行结果。古代人做决策的时候依靠占卜，这也被认为是随机化算法，因此随机化算法将某一步或某几步置于运气的控制之下，即该算法在运行的过程中的某一步或某几步涉及一个随机决策，或者说其中的一个决策依赖于某种随机事件。

在通信理论中，一个信号的随机性称作噪声，它对立于由源（信号）所引起的那一部分变化。

50.1.2 Pseudorandomness

伪随机性（Pseudorandomness）是指一个过程似乎是随机的，但实际上并不是。

如果一个序列，一方面它是可以预先确定的，并且是可以重复地生产和复制的，另一方面它又具有某种随机序列的随机特性（即统计特性），我们便称这种序列为伪随机序列。例如，在计算机科学中，伪随机数（或称伪乱数）是使用一个确定性的算法计算出来的似乎是随机的数序，在计算伪随机数时假如使用的开始值不变的话，那么伪随机数的数序也不变，因此伪随机数实际上并不随机。

用来计算伪随机数的函数被称为随机函数，使用随机函数产生随机数的算法称为随机数生成器。一些随机函数是周期性的，虽然一般来说使用非周期性的函数要好得多，但周期性的随机函数往往快得多。有些周期函数的系数可以调整，之后它们的周期非常大，基本上与非周期的函数效果一样。

伪随机数的随机性可以用它的统计特性来衡量，其主要特征是每个数出现的可能性和它出现时与数序中其它数的关系。伪随机数的优点是它的计算比较简单，而且只使用少数数值很难推算出计算它的算法。一般人们使用一个假的随机数，比如电脑上的时间作为计算伪随机数的开始值。

伪随机数的一个特别大的优点是它们的计算不需要外部的特殊硬件的支持，因此在计算机科学中伪随机数依然被使用。真正的随机数必须使用专门的设备，比如热噪讯号、量子力学的效应、放射性元素的衰退辐射，或使用无法预测的现象，譬如用户按键盘的位置与速度、用户运动鼠标的路径坐标等来产生。

在电脑模拟中伪随机数用来模拟产生随机的过程，背景噪音产生器中也可应用伪随机数。由于伪随机数不是真的随机数，在有些方面它们不能被使用。例如，在密码学中使用伪随机数要小心，因为其可计算性是一个可以攻击的地方。

统计学、蒙特·卡罗方法[1]上使用的伪随机数也必须挑选周期极长、随机性够高的随机函数，以确保计算结果有足够的随机性。

20 世纪 40 年代，在 John von Neumann，Stanislaw Ulam 和 Nicholas Metropolis 在洛斯阿拉莫斯国家实

[1]二十世纪四十年代中期，由于科学技术的发展和电子计算机的发明而被提出的一种以概率统计理论为指导的一类非常重要的数值计算方法。

验室为核武器计划工作时，发明了蒙特卡罗方法[2]，后来广泛应用在金融工程学、宏观经济学、生物医学、计算物理学(如粒子输运计算、量子热力学计算、空气动力学计算)等领域。

在解决实际问题的时候应用蒙特卡罗方法主要有两部分工作：

1. 用蒙特卡罗方法模拟某一过程时，需要产生各种概率分布的随机变量。
2. 用统计方法把模型的数字特征估计出来，从而得到实际问题的数值解。

蒙特卡罗方法(Monte Carlo method)又称统计模拟方法，它使用随机数(或更常见的伪随机数)来解决很多计算问题，与之对应的是确定性算法。

通常蒙特卡罗方法可以粗略地分成两类：一类是所求解的问题本身具有内在的随机性，借助计算机的运算能力可以直接模拟这种随机的过程。例如在核物理研究中，分析中子在反应堆中的传输过程。中子与原子核作用受到量子力学规律的制约，人们只能知道它们相互作用发生的概率，却无法准确获得中子与原子核作用时的位置以及裂变产生的新中子的行进速率和方向。科学家依据其概率进行随机抽样得到裂变位置、速度和方向，这样模拟大量中子的行为后，经过统计就能获得中子传输的范围，作为反应堆设计的依据。

另一种类型是所求解问题可以转化为某种随机分布的特征数，比如随机事件出现的概率，或者随机变量的期望值。通过随机抽样的方法，以随机事件出现的频率估计其概率，或者以抽样的数字特征估算随机变量的数字特征，并将其作为问题的解。这种方法多用于求解复杂的多维积分问题。

假设我们要计算一个不规则图形的面积，那么图形的不规则程度和分析性计算(比如，积分)的复杂程度是成正比的。蒙特卡罗方法基于这样的思想：假想你有一袋豆子，把豆子均匀地朝这个图形上撒，然后数这个图形之中有多少颗豆子，这个豆子的数目就是图形的面积。当你的豆子越小，撒的越多的时候，结果就越精确。借助计算机程序可以生成大量均匀分布坐标点，然后统计出图形内的点数，通过它们占总点数的比例和坐标点生成范围的面积就可以求出图形面积。

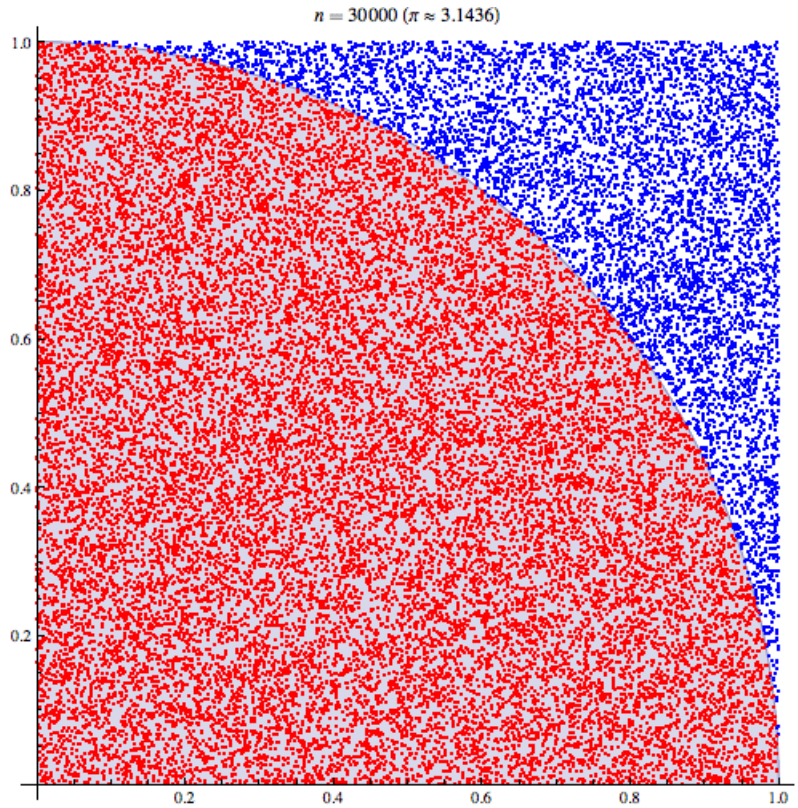


图 50.1: 使用蒙特卡罗方法估算 π 值时，放置 30000 个随机点后，π 的估算值与真实值相差 0.07%。

[2]Ulam 的叔叔经常在蒙特卡罗赌场输钱得名，而蒙特卡罗方法正是以概率为基础的方法。

50.2 Random Number

在统计学的不同技术中需要使用随机数,比如在从统计总体中抽取有代表性的样本的时候,或者在将实验动物分配到不同的试验组的过程中,或者在进行蒙特卡罗模拟法计算的时候等。

早期的计算机主要用于数值计算,用计算机生成随机性的想法通常以生成特定范围内的随机数(random number)来表示。

从理论上看,没有办法事先确定值的数是随机数,它将是一组等概率的值中的一个,例如掷骰子生成 1 ～ 6 之间的随机值。如果骰子是公平的,那么没有办法预测将会出现什么数,6 个值出现的可能性是相等的。

虽然随机数的概念有直观的意义,但却是在计算机内部很难表示的概念。计算机根据一组内存中的指令序列工作,因此以一种确定的模式运行。如何遵循一组确定的规则产生一组不确定的结果? 如果一个数是由一个确定的过程产生的,任何用户应该能够通过同一组规则工作并等待计算机的响应。

事实上,计算机确实能够用确定性的过程产生所需要的随机数。从理论上说,一旦用户能够遵循同一规则,并等待计算机的响应,那么这个策略就可以工作了,没有人会怀疑这个策略。

随机数生成器(Random number generator)是通过一些算法、物理讯号、环境噪音等来产生看起来似乎没有关联性的数列的方法或装置。在类 UNIX 操作系统中,/dev/random 是一个特殊的设备文件,可以用作随机数发生器或伪随机数发生器。它允许程序访问来自设备驱动程序或其它来源的背景噪声。

真随机数生成器并不一定非得是特殊设计的硬件,Linux 操作系统内核中的随机数生成器(/dev/random)维护了一个熵池(搜集硬件噪声,如:键盘、鼠标操作、网络信号强度变化等),使得它能够提供最大可能的随机数据熵,因此同样是高品质的真随机数生成器。

1994 年,美国程序员 Theodore Y. Ts'o(曹子德)第一次在 Linux 内核中实现了/dev/random 以及对应的核心驱动程序,使用 SHA-1 散列算法而非密码来避开法律限制,从而让 Linux 成为所有操作系统中第一个实现了以系统背景噪音产生的真正随机数生成器。/dev/random 可以不用依靠硬件随机乱数产生器来独立运作,提升效能的同时也节省了成本。其他的守护行程 (例如 rngd) 可以从硬件取得随机数提供给/dev/random,应用程序可以经由 /dev/random 取得随机数。在/dev/random 与 /dev/urandom 实现出来之后,很快就成为在 Unix、Linux、BSD 与 Mac OS 的共通标准接口。

Linux 随机数发生器有一个容纳噪声数据的熵池,在读取/dev/random 设备时会返回小于熵池噪声总数的随机字节。/dev/random 可生成高随机性的公钥或一次性密码本。若熵池空了,对/dev/random 的读操作将会被阻塞,直到收集到了足够的环境噪声为止。这样的设计使得/dev/random 是真正的随机数发生器,提供了最大可能的随机数据熵,建议在需要生成高强度的密钥时使用。

/dev/random 的一个副本是/dev/urandom("unlocked",非阻塞的随机数发生器),它会重复使用熵池中的数据以产生伪随机数据,这样对/dev/urandom 的读取操作不会产生阻塞,但其输出的熵可能小于/dev/random 的。/dev/urandom 可以作为生成较低强度密码的伪随机数生成器,不建议用于生成高强度长期密码。

/dev/random 也允许写入,任何用户都可以向熵池中加入随机数据。即使写入非随机数据亦是无害的,只有管理员可以调用 ioctl 以增加熵池大小,Linux 内核中当前熵的值和大小可以通过访问/proc/sys/kernel/random/得到。

并不是所有操作系统中的/dev/random 的实现都是相同的,而 Linux 是第一个以背景噪声产生真正的随机数的实现,后来 FreeBSD 操作系统实现了 256 位的 Yarrow 算法变体以提供伪随机数流。与 Linux 的/dev/random 不同,FreeBSD 的/dev/random 不会产生阻塞,与 Linux 的/dev/urandom 相似,提供了密码学安全的伪随机数发生器,而不是基于熵池。

Yarrow 算法的前提是现代的伪随机数发生器的安全性很高,若其内部状态不为攻击者所知,且比熵估计更易理解。在某些情况下,攻击者可以在某种程度上掌握熵的量。例如,无盘服务器的熵几乎全部来自于网络,使得它可能易受中间人攻击的影响。算法的种子会被规则的重置——在网络和磁盘负载较轻的系统上,一秒内种子可能被重置数次。

FreeBSD 的/dev/urandom 则只是简单的链接到了/dev/random，而且 FreeBSD 也支持硬件随机数发生器，并在安装了类似硬件时会替代 Yarrow 算法。

与 FreeBSD 的实现类似，AIX 采用了它自身的基于 Yarrow 的设计，但 AIX 使用的熵源数量低于标准/dev/random 的实现，并且在系统认为熵池的熵达到足够值时停止填充熵池。

在 Windows NT 中，ksecdd.sys 提供了类似的功能，但读取特殊文件\Device\KsecDD 并不会提供与 UNIX 中相同的功能，用于产生密码用随机数据的函数是 CryptGenRandom 和 RtlGenRandom。

DOS 没有本地的提供类似功能，但有开源的 Noise.sys 提供了类似功能。该实现与/dev/random 的功能和接口类似，即创建了两个设备（RANDOM$ 与 URANDOM$），也可以通过/DEV/RANDOM$ 和/DEV/URANDOM$ 访问。

EGD（熵收集守护进程，entropy gathering daemon）通常可以在不支持/dev/random 设备的 Unix 系统中提供类似的功能。这是一个运行于用户态的守护进程，提供了高质量的密码用随机数据。一些加密软件（例如 OpenSSL，GNU Privacy Guard）和 Apache HTTP 服务器支持在/dev/random 不可用的时候使用 EGD。

EGD 或者类似的软件（例如 prngd）可以从多种来源收集伪随机的熵，对这些数据进行处理以去除偏置，并改善密码学质量，然后允许其它程序通过 Unix 域套接口（通常使用/dev/egd-pool）或 TCP 套接口访问其输出。EGD 程序通常使用创建子进程的以查询系统状态的方式来收集熵，它查询的状态通常是易变的、不可预测的（例如 CPU、I/O、网络的使用率，也可能是一些日志文件和临时目录中的内容）。

在大多数实际应用中，这个数是否为真正随机是无所谓的，真正为用户所关心的是这个数是随机出现的。对随机出现的数，它们的行为应该像统计学中的随机数，而且提前预测它们非常困难，但没有用户为此操心。

由计算机内部的算法产生的“随机”数称为伪随机数（pseudo-random number），强调没有涉及真正的随机活动。但是，在真正关键性的应用中（比如在密码学中），人们一般使用真正的随机数。

在开发应用时，不要使用任何第三方的随机数解决方案，哪怕是一些高级的安全库所提供的声称“非常安全”的随机函数。因为它们都是用户态的密码学随机数生成器，而 urandom 是内核态的随机数生成器，内核有权访问裸设备的熵，从而可以确保不在应用程序间共享相同的状态。

随机数失败案例大多出现在用户态的随机数生成器，而且用户态的随机数生成器几乎总是要依赖于内核态的随机数生成器（如果不依赖，那风险则更大）。除了简化某些开发工作，用户态随机数生成器没有任何额外的好处，反而增加了因引入第三方代码所可能导致的潜在安全风险，因此开发者在需要密码学安全的随机数时应该使用/dev/urandom。

50.3 Pseudo-random Number

大部分计算机程序和语言中的随机函数的确是伪随机数生成器，它们都是由确定的算法，通过一个种子（比如时间），来产生看起来随机的结果。毫无疑问，任何人只要知道算法和种子，或者之前已经产生了的随机数，都可能获得接下来随机数序列的信息。因为它们的可预测性，在密码学上并不安全，所以我们称其为“伪随机”。

在计算机中，随机函数生成都是伪随机数，这对于要求不严格的情况来讲是可以允许的，而且几乎所有的随机数生成程序实际上产生的都是一系列的伪随机数。

从严格的数学意义上而言，伪随机数不是随机的，因为数的整个序列由基于某个初始种子值的确定性函数产生的。但不管怎样，可以生成这样的数的序列，使它们看起来非常接近于随机。

度量随机数发生器质量的最基本值是它的循环长度，发生器周期性地重复同一序列的随机数的个数，因此循环长度越长，发生器越好。如果不能充分肯定正使用的随机数发生器的性质，就应该考虑使用被充分证明了的随机数发生器。

- 在 VB 和 ASP 中通常采用 Rnd 获取伪随机数，但大多数时候配合 Randomize 使用。

Rnd 函数得出的事实上是顺序读取一个随机数列表中的数，Randomize 的功能是重新生成随机数列表，因此一般放置于 Rnd 函数前。

- 在 C 语言中，使用库函数 rand() 可以产生一个 0 ～ 32768 之间的伪随机整数。若要产生带有范围的伪随机数，可以使用 mod 运算符，例如 rand()%15 代表产生一个 0 ～ 14 之间的伪随机整数。

很多开发环境都把随机数发生器作为其内部函数，采用最多的函数是返回 [0,1] 区间均匀分布的值，还有些函数返回具有给定均值和方差的高斯分布值，并非所有的随机数发生器都具有同样的质量。

如果能产生 [0,1] 区间的均匀分布的数，那么也就具备了产生很多其他随机变量的基础。例如，如果想生成具有参数 p 的伯努利随机变量，那么调用 [0,1] 区间的一个均匀随机数，确定它是否小于 p，如果小于 p，则返回值 1.0，否则返回值 0.0。通过连续地调用伯努利随机变量就能够构造二项式随机变量。

也可以用均匀随机变量来构造其他重要的随机变量，比如高斯随机变量和指数随机变量。通过取两个相互独立的均匀随机数 U_1 和 U_2，将它们转换为：$\sqrt{-2\log U_1}\sin(2\pi U_2)$ 和 $-\sqrt{-2\log U_1}\cos(2\pi U_2)$

从而得到两个均值为 0，方差为 1 的高斯随机数。通过转换 $\frac{1}{-\lambda}\log(U_1)$ 可以得到一个具有参数 λ 的指数随机数。

可以从高斯随机变量去生成卡方随机变量，只要取高斯数的平方，就能得到卡方数（自由度为 1）。

两个相互独立的卡方随机变量的和是另一个卡方随机变量，它的自由度等于两个相互独立的卡方随机变量的自由度之和，因此通过累加足够数目的高斯随机变量的平方可以生成任何想要得到的卡方随机变量。

通过简单地取两个相互独立的高斯随机变量的比值，也可以生成一个柯西随机变量。

另一种方法是使用公式 $\tan[\pi(U_1 - 0.5)]$，它仅依赖于一个均匀分布的随机变量。这里要考虑的另一个随机变量是泊松随机变量，生成它的难度要大一些，推荐的过程如下：

对所能观察到的最大数 N 确定某个任意截断值，然后确定 0，1，2，...，N 的出现概率（可以根据统计表或概率质量函数的平滑计算）。随后，产生一个均匀随机数以确定 0，1，2，...，N 中的第一个数，使得累积概率比从均匀分布获得的值要大。如果考虑完所有 N 个数都没有获得一个比均匀分布的样本点更大的累积概率，就简单地选择 N，不过这种情况可能性不大。

50.3.1 Generator

随机数是专门的随机试验的结果，其最重要的特性是它在产生时后面的那个数与前面的那个数毫无关系。产生随机数有多种不同的方法，例如丢硬币、丢骰子、洗牌就是生活上常见的随机数产生方式。

真正的随机数是使用物理现象产生的（比如掷钱币、骰子、转轮、使用电子元件的噪音、核裂变等），这样的随机数生成器叫做物理性随机数生成器，它们的缺点是技术要求比较高。

实际上，严格意义上的真随机可能仅存在于量子力学之中，我们当前所想要的（或者所能要的），并不是这种随机。我们其实想要一种不可预测的、统计意义上的、密码学安全的随机数，只要能做到这一点的随机数生成器，都可以称其为真随机数生成器，因此大部分计算机上的随机数并不是真正的随机数，只是按一定的算法和种子值生成的重复的周期比较大的数列。但是，在实际应用中往往使用伪随机数就足够了，这些数列是“似乎”随机的数。

伪随机数是通过一个固定的、可以重复的计算方法产生的，因此伪随机数并不真正地随机，它们实际上是可以计算出来的，但是它们具有类似于随机数的统计特征，这样的生成器叫做伪随机数生成器。例如，作为 stdlib.h 接口的一部分，ANSI C 库包含了一个产生伪随机数的函数 rand，并在 stdlib.h 接口中给出了 rand 的原型。

```
int rand(void);
```

它指出 rand 没有参数，并返回一个整数，这就是一个伪随机值，即每次调用 rand 返回的结果都不同。

rand 的结果保证是非负的，并且不大于常数 RAND_MAX，该常数也是在 stdlib.h 接口中定义的。

每次调用 rand 时，它返回的是 0 ～ RAND_MAX 的不同的整数，包括 0 和 RAND_MAX。RAND_MAX 的值取决于计算机系统，例如在典型的 Macintosh 环境中，RAND_MAX 的值是 32 767，而在典型的 Unix 工作站上，它的值是 2 147 483 647。

在写一个用到随机数的程序时，不需要对 RAND_MAX 的值作精确的假设。相反，我们所编写的程序应该能在任意的 RAND_MAX 的系统上使用。如果做到了这一点，就可以将一个在某一系统上工作的程序重新编译一下，它就能在另一个系统上工作。

运行下面的程序 randtest.c，将会显示 rand 的行为。

```
/*
* File:randtest.c
* ---------------
* This program tests the ANSI rand function.
*/

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
* Constants
* -------------
* NTrials -- Number of trials.
*/

#define NTrials 10

/* Main Program */

main()
{
  int i, r;

  printf("On this computer, RAND_MAX = %d.\n", RAND_MAX);
  printf("Here are the results of %d calls to rand: \n", NTrials);
  for(i = 0; i < NTrials; i ++){
    r = rand();
    printf("%10d\n", r);
  }
}
```

执行程序 randtest.c 后，可以看到该程序生成的数都是整数，没有一个数大于这个实例运行时显示的当前系统的 RAND_MAX 值。

因为这些数是伪随机数，我们可以知道它一定有某种模式，但这又不太可能是我们认识的那些模式。从用户的角度来看，这些数是随机出现的，因为不知道基本的模式是什么。

50.3.2 Range

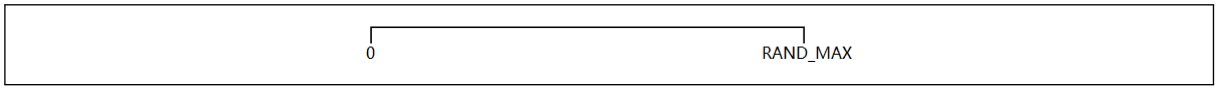
rand 库函数给出了一个生成伪随机数的机制，但它很少能满足特定应用的精确的数值范围。

虽然 rand 函数可以产生在 0 ～ RAND_MAX 之间均匀分布的数，但按照具体应用的要求，我们需要的数可能是一个落在某一范围内的数。

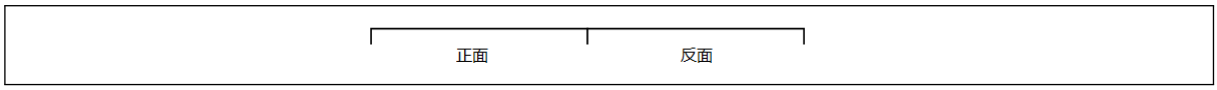
通常这个范围是比较小的，例如如果要模拟抛硬币的过程，需要将这个大范围内的随机数转换为只包含两个值的小范围：正面和反面。类似地，要模拟掷骰子的过程时，需要将 rand 返回的伪随机数转换为 1 ～ 6 之间的值(包括 1 和 6)。

为了完成这类转换，需要重新解释 rand 产生的每一个随机数，使它能覆盖不同的范围。

开始时，rand 函数产生的数位于数轴上 0 和 RAND_MAX 之间的某一位置。



现在，如果要模拟抛硬币的过程，可以把这条线分成两段，一段表示正面，另一段表示反面。



用这个想法可以开发出如下所示的 cointest.c 程序，它可以模拟抛硬币的过程。

```
/*
* File:cointest.c
* ------------------
* This program simulates flipping a coin.
*/

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
* Constants
* -------------
* NTrials —Number of trials
*/

#define NTrials 10

/* Main Program */

main()
{
  int i;

  for(i = 0; i < NTrials; i ++){
    if(rand() <= RAND_MAX / 2){
      printf("Heads\n");
    }else{
      printf("Tails\n");
    }
  }
}
```

该程序打印出字符串“Heads”或字符串“Tails”，每种输出的出现几率大约为 50%。正面和反面有合理的组合，用户不能很容易地找出它的模式。

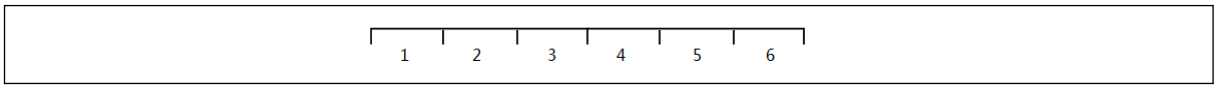
在考虑如何将 rand 的结果转换为两种可能时，最初可能会采取一些比较简单的方法，例如用取余数的运算。如果将 rand 的结果除以 2，并取它的余数，结果是 0 或者 1，然后可以在程序中定义 0 为正面，1 为反面。

但是，这个策略实际上是危险的，因为它不能保证 rand 的结果对偶数和奇数是随机分布的，它仅能保证结果的大小将是沿着数轴在 0 和 RAND_MAX 之间随机分布的。

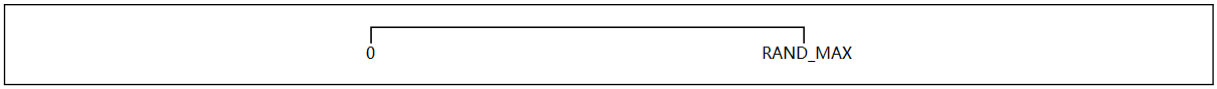
常见错误：当将 rand 的结果转换为更有限的整数范围时，不要试图用取余数运算。在使用 rand 时，只有返回的结果的位呈在数轴上具有随机特性。

rand 的一种常见的实现生动地说明了用这种方法生成随机数会产生多严重的错误。在许多计算机系统上，rand 函数是以一种轮流生成奇数和偶数的方法实现的。这个结果仍然按照在数轴上的距离在数轴上随机分布。但是，采用取余数运算模拟抛硬币的程序将会按严格的交替模式生成正面和反面。

在模拟掷骰子的过程中，如果采用 cointest.c 中的策略，所需要做的只是把输出



覆盖在数轴



假如按 cointest.c 的结构，以一种原始的方式处理这个任务，得到的程序如下：

```
int RollDie(void)
{
  if(rand() < RAND_MAX / 6){
    return (1);
  }else if(rand() < RAND_MAX * 2 / 6){
    return (2);
  }else if(rand() < RAND_MAX * 3 / 6){
    return (3);
  }else if(rand() < RAND_MAX * 4 / 6){
    return (4);
  }else if(rand() < RAND_MAX * 5 / 6){
    return (5);
  }else{
    return (6);
  }
}
```

在 RollDie 函数的实现有一些严重的问题，对于这一类自己编码解决的问题值得仔细考虑。

代码中的第一个问题是：在这里做了一个非常容易、非常自然的假设。

在 *Zen and the Art of Motorcycle Maintenance* 这本书中，Robert Pirsig 称这类错误（一个似乎合理的假设导致错误的结论）为大胆的陷阱（gumption trap），而且在程序设计中经常出现。Pirsig 的书至少提供了许多程序调试的有用的观点，就像修理摩托车的方法一样。

这里，该程序作了下列假设：

- 如果产生的随机数小于最大值的 1/6，返回值 1。
- 否则，如果产生的随机数小于最大值的 2/6，返回 2。
- 否则，如果产生的随机数小于最大值的 3/6，返回 3，以此类推。

问题是，上述的代码没有完全实现这个想法。通过重复调用函数 rand，在每一个 if 语句中都将产生一个新的随机数。该函数的结构依赖于每次产生的随机数都相同的假设。

要了解这种情况，观察一下这个 RollDie 的实现，现在来了解一下在什么条件下它将会返回 2。为了使函数返回 2，第一个 if 语句必须为 FALSE，第二个语句必须为 TRUE。

第一个 if 语句的条件为 FALSE 的概率为 $\frac{5}{6}$，第二个 if 语句会使得返回 TRUE 的概率为 $\frac{1}{3}$，因为调用 rand 返回的是一个全新的随机数。

在统计学上，两个独立事件一起出现的概率为两个独立事件出现的概率的乘积，因此 RollDie 返回 2 的概率是

$$\frac{5}{6} \times \frac{1}{3} = \frac{5}{18}$$

18 次中出现 5 次的机会几乎是 6 次中出现 1 次的机会的 2 倍，这意味着 RollDie 函数返回 2 的机会比应该出现的机会大。为了解决这个问题，需要声明一个变量保存 rand 的结果，以后在每行中测试这个变量。

```
int RollDie(void)
{
  int r;
  r = rand();

  if(rand() < RAND_MAX / 6){
    return (1);
  }else if(rand() < RAND_MAX * 2 / 6){
    return (2);
  }else if(rand() < RAND_MAX * 3 / 6){
    return (3);
  }else if(rand() < RAND_MAX * 4 / 6){
    return (4);
  }else if(rand() < RAND_MAX * 5 / 6){
    return (5);
  }else{
    return (6);
  }
}
```

常见错误：当调用一个产生伪随机数的函数时，重要的是要记住每次调用这个函数都会产生一个不同值。如果要保存一个特定值，必须把函数的结果保存在一个变量中。

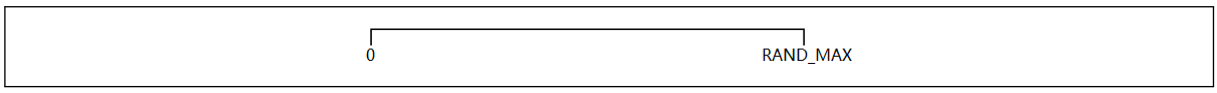
但是，在 RollDie 的第二次实现中，仍然存在问题而且更不容易发现。在大多数系统上，RAND_MAX 的值是按某种原因给出的。通常为 RAND_MAX 选择的值不一定是 rand 函数可能返回的最大值，而是系统中 int 类型的最大的正值。这个限制在所建议的 RollDie 的实现中引起了一个严重的问题，因为该程序中的某些中间结果可能大于最大的整数值。

尽管 RAND_MAX * 2 / 6 最终的结果适合 int 类型，但 C 语言中的优先规则指出 RAND_MAX 先乘 2，再除 6，产生一个所允许范围之外的整数称为运算溢出（arithmetic overflow）。如果出现这样的溢出，程序将不能产生预期的结果。

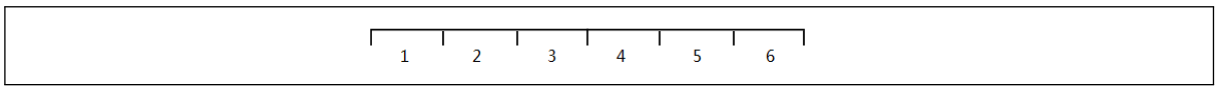
虽然可以把表达式改为 RAND_MAX / 6 * 2 来解决问题，然而，这仍然不是最好的办法。

RollDie 实现的真正问题在于这个过程太复杂。代码将六种可能出现的情况中的每一种作为独立的情况来测试，为此所需要的是用某些数学的方法将所有这些特殊情况一起消除掉。

仍然从几何的观点看这个问题，我们所需要做的只是将数轴



转换为离散的区间。



与使用 if 语句相比，采用算术运算能更好地完成此任务。然而，在确定算术运算如何用于这种情况之前，有必要将此问题通用化，使解决方案的技术可以用在更多的应用中。

50.3.3 Universalization

下面以如何写一个可以做随机选择的程序为例来说明接口设计原则。能够模拟随机行为是很必要的,例如当要写一个涉及抛硬币或掷骰子的计算机游戏程序时,而且在某些更实际的场合,模拟随机行为也是很有用的。

若程序具有确定性(deterministically)行为,这就意味着给定一组输人,程序的行为是完全可预测的。这种程序的行为是可重复的,即不论何时运行程序,产生的都是同样的结果。

而在某些程序设计应用中(如游戏或仿真),程序行为的不可预测性是很重要的。例如,每次产生同样结果的计算机游戏是很乏味的。为了创建一个有随机行为的程序,在程序中需要一些表示随机过程的机制,如抛硬币或掷骰子。模拟这种随机事件的程序称为非确定性(nondeterministic)程序。

使程序具有随机行为具有一定的复杂性,需要将这些复杂性隐藏在接口的背后以方便使用。例如,为了模拟掷骰子的过程,要生成一个 1 ~ 6 之间的随机数。如果想让程序"取任意一张牌",需要选择一个 1 ~ 52 之间的值。要为一个欧洲轮盘赌的轮子建模,需要选取一个 0 ~ 36 之间的数。

一般来说,我们需要的不是选择一个 0 ~ RAND_MAX 之间的数的函数,而是一个在指定的范围之内选择一个随机整型数的函数,因此函数可以用下列原型定义:

```
int RandomInteger(int low, int high);
```

换句话说,如果给了这个函数两个整数,它将返回一个位于这两点之间的一个随机数,包括这两个点。

- 如果要模拟掷骰子,只需要调用

```
RandomInteger(1, 6);
```

- 如果要模型欧洲轮盘赌的轮子的旋转,只需要调用

```
RandomInteger(0, 36);
```

这样的通用工具有很多用途,可以将它放入某一个库中,使之能反复使用。

现在已经知道如何产生 0 ~ RAND_MAX 之间的随机数。一般说来,需要的不是选择一个 0 ~ RAND_MAX 之间的数的算法,而是一个在指定的范围之内选择一个随机整型数(或浮点数)的算法。

算法:将 0 ~ RAND_MAX 之间的随机数转换到更严格的范围内。

- 规范化:将 rand 的整数结果转换为 0≤d<1 之间的浮点数 d。
- 换算:将 d 乘以所希望范围的大小,使它可以包含正确的整型数个数。
- 截断:丢弃小数部分,将此数截断为整型数。这一步给出了一个下界为 0 的随机整型数。
- 转换:将此整型数转换,使之以所期望的下界开始。

在给算法实现的函数命名时,通常可以用如下的命名方式:

```
int RandomInteger(int low, int high);
double RandomReal(double low, double high);
```

在求解随机整型数的算法中,要规范化这个值时,首先需要将结果转换为 double 型,然后除以范围中的元素数。

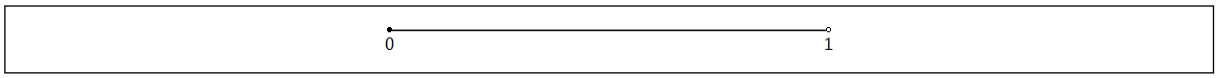
元素数可以从 0 ~ RAND_MAX,包括这两个值,因此可能出现的数的个数是 RAND_MAX 加 1(在 1 ~ RAND_MAX 之间有 RAND_MAX 个值,还需要包括 0)。

可以用强制类型转换明确地将一种类型的值转换为另一种类型的值。强制类型转换是将所希望的新类型括在一对圆括号内,并把它写在要转换的值的前面。例如,可以用下面的语句将 rand 的结果转换为 0 ~ 1 之间的数 d。

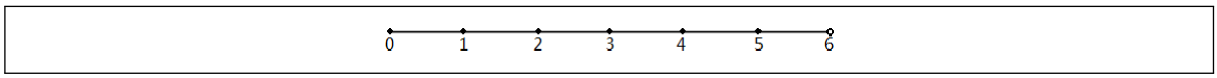
```
d = (double) rand() / ((double) RAND_MAX + 1);
```

在这个分数中,分子必须小于分母,使得最终的结果严格小于 1,这样在这个过程结束时,就可以得到一个至少为 0,但严格小于 1 的随机实数。

在数学中,可以等于一个端点值,但不能等于另一个端点值的实数范围称为半开区间(half-open interval)。在图中,不包括在范围内的端点用空心圆表示,因此变量 d 的可能的范围可以用下图表示:



下一步是换算这个值，使它覆盖正确的整型数的数目。例如，要模拟掷骰子，需要把这个值乘 6，因此新的范围为：



注意，在这个范围中有 6 个整数：0、1、2、3、4 和 5。6 本身在可能值的范围外，因为 d 永远不可能为 1。

在更一般的情况下，要将规范化的随机数乘以这个范围内的元素个数，元素个数可以用表达式 (high – low + 1) 表示。

表达式中多出来的这个 1 是必须的，因为该范围是包括两个端点的。减法给出了两点之间的距离，它比闭区间中包含的整数少 1。对掷骰子来讲，可能出现六个值是 1、2、3、4、5 和 6，但 6 减 1 是 5。为了计算出现的值的个数，需要将最大的数减去最小的数，然后加 1。

下一步，向后截断该实数，使它成为一个整数。在 C 语言中，如果用强制类型转换将 double 转换为 int，那么该转换是通过丢掉小数部分实现的。因此，如果要将一个已知的实型数转换为大于等于 0，但严格小于 6 的数，将得到 0、1、2、3、4 和 5 中的某一个整数。

过程中的最后一步是将结果转换到所期望的范围内。现在已经有了整型结果的正确个数，仅有的问题是它们是从 0 开始。为了得到一组正确的可能性，只要简单地加上下界值即可。

可以把所有的这些步骤放在一起，写出如下的 RandomInteger 函数的实现：

```
/*
* Function:RandomInteger
* ------------------------------
* This function first obtains a random integer in the range [0...RAND_MAX]
* by applying four steps:
* (1) Generate a real number between 0 and 1.
* (2) Scale it to the appropriate range size.
* (3) Truncate the value to an integer.
* (4) Translate it to the appropriate starting point.
*/
int RandomInteger(int low, int high)
{
  int k;
  double d;
  d = (double) rand() / ((double) RAND_MAX + 1);
  k = (int) (d * (high –low + 1));
  return (low + k);
}
```

对于像 RandomInteger 这些有用的函数，可以将它们放入某一个库中，从而将解决某类问题的工具通用化。

在库中保存工具的过程中要建立一个相应的接口并添加接口开发过程中应具备的特性。这个过程的第一步是建立一个接口，一旦完成了这个接口，就可以在另一个文件中写出对应的实现。

在大多数情况下，除了文件类型外，接口和它的实现的文件名是相同的。如果接口被命名为 random.h，一般会用 random.c 作为实现文件的名字。

50.3.4 Structure

接口的基本结构主要由为使用该库的客户所编写的注释和接口项组成。

在设计一个接口时，注释提供了该接口为客户提供使用这个库时所需的信息，千万不要忽略这些注释。

接口输出给它的客户的每个定义称为接口项(interface entry)，接口项有一些不同的形式。

1. 函数原型：接口必须包含客户可用的每个函数的原型。
2. 常量定义：接口通常用 #define 定义客户需要知道的常量。例如，stdlib.h 接口定义常量 RAND_MAX 来告诉它的客户由 rand 函数返回的最大值。
3. 类型定义：使用接口定义一些客户所用的新类型是很有用的。例如，genlib.h 接口定义了类型 bool 和 string。

除了这些项和它们相关的注释外，每一个接口都应该包括三行，这三行可以帮助编译器记录它所读取的接口。

在最初的注释后，在真正的项之前，每个接口应该包括下面两行：

```
#ifndef _name_h
#define _name_h
```

其中，name 是接口文件的名字。

在复杂的程序中，一个接口可能通过各种路径被包含许多次。为了避免编译器重复读取同一个接口，如果符号 _name_h 以前已经被定义过了，那么下面的行可以使编译器跳过接口的所有行，直到 #endif 行。

```
#ifndef _name_h
```

第一次读取这个接口时，这个符号没有定义过，于是编译器继续读。然而，接下来编译器立刻遇到了 #define 行。

```
#define _name_h
```

它定义了符号 _name_h。如果以后编译器又要读同一个接口，_name_h 已经定义过了，它应该忽略接口中的项。

在接口技术中，这条规则是很明确的。在写每一个接口的时候，都要遵循这条规则，即必须包含 #ifndef、#define 和 #endif。

另外，除了模板文件外，接口有时也需要用同样的 #include 行包含其他已经在程序中使用的接口。

当写一个特定类型的文件时，每次必须包含的这类风格模式通常被称为模板文件(boilerplate)。这些行是接口的模板文件，要确保它们总是位于该特定类型文件(如接口文件)中。

语法：接口文件

```
/* comments */
#ifndef _name_h
#define _name_h
任何所需的 #include 行
接口项
#endif
```

其中，name 是库的名字，#include 行部分仅当接口本身需要其他库时才使用，它由标准的 #include 行组成。

接口项表示库输出的函数原型、常量和类型，其中在接口中定义类型是现代程序设计中非常重要的技术。

接口的最后一行必须以 #endif 表示接口结束。

```
#endif
```

50.3.5 Implementation

接下来,在实现 random.h 接口的第一个任务就是写初始的注释,这些注释将解释这个库提供了什么,谁可以使用它。

在注释后面,应该包括接口的模板文件,下面的示例代码是 random.h 的模板文件。

```
#ifndef _random_ h
#define _random_h
```

在 random.h 接口中,下一个要写的是有关 RandomInteger 过程的注释。

```
/*
* Function:RandomInteger
* Usage:RandomInteger(low, high);
* ----------------------------------------
* This function returns a random integer in the range low to high, inclusive.
*/
```

这段注释给客户提供使用这个函数所需的信息,例如 Usage 这一行给出了调用函数的实例,并且注释中也包括这个函数功能的说明,但没有说明函数是如何实现的。

接口的下一部分是函数本身的原型。

```
int RandomInteger(ink low, int high);
```

这一行是接口中仅有的对编译器有实际意义的部分,其他都是注释或模板文件。

接口中的最后一行是简单的 #endif 行,它也是接口的模板文件的一部分,于是得到了 random.h 的最初版本,后面还会加入新的函数来扩展它的功能。

```
/*
* File:random.h
* ------------------
* This file contains a preliminary version of a library interface to produce
* pseudo-random numbers.
*/

#ifndef _random_h
#define _random_h

/*
* Function:RandomInteger
* Usage:RandomInteger(low, high);
* ----------------------------------------
* This function returns a random integer in the range low to high, inclusive.
*/

int RandomInteger(int low, int high);

#endif
```

random.h 接口的实现在另一个文件 random. c 中。对于已有的这个接口,对应的实现文件如下。

```
/*
* File:random.c
* ------------------
* This file implements the preliminary random.h interface.
*/

#include <stdio.h>
#include <stdlib.h>
```

```
#include “genlib.h”
#include “random.h”

/*
* Function:RandomInteger
* ------------------------------
* This function first obtains a random integer in the range [0..RAND_MAX] by
* applying four steps:
* (1) Generate a real number between 0 and 1.
* (2) Scale it to the appropriate range size.
* (3) Truncate the value to an integer.
* (4) Translate it to the appropriate starting point.
*/

int RandomInteger(int low, int high)
{
  int k;
  double d;
  d = (double) rand() / ((double) RAND_MAX + 1);
  k = (int) (d * (high –low + 1));
  return (low + k);
}
```

接口对应的实现也是以初始的注释开始，接下来的部分列出了编译所需的 #include 文件。stdio.h 和 genlib.h 是必需的，访问 rand 函数还需要 stdlib.h。

最后，每个实现需要包含它自己的接口，使编译器能针对真正的定义检查原型。

在 #include 行后，下一部分是由接口输出的函数的实现，以及任何对将来维护这个程序的程序员有用的注释组成。

和其他所有的说明形式一样，编写注释时必须考虑客户的立场。当写注释时，必须把自己放在读者的位置，这样才能理解读者想看到什么信息。

.c 文件中的注释与.h 文件中的注释相比有不同的读者，因为实现中的注释是为另一个实现者而写的，另一个实现者必须以某种方式修改这个实现。因此，编写者必须解释这个实现是如何工作的，并提供所有以后维护时应该知道的任何细节。另一方面，接口中的注释是为客户而写的。客户不需要读实现中的注释，接口中的注释应该已经足够了。

50.3.6 Usage

可以写一个如下所示的测试程序 dicetest.c 来测试 random.c 的实现，其中的主程序使客户可以使用新的随机函数库。

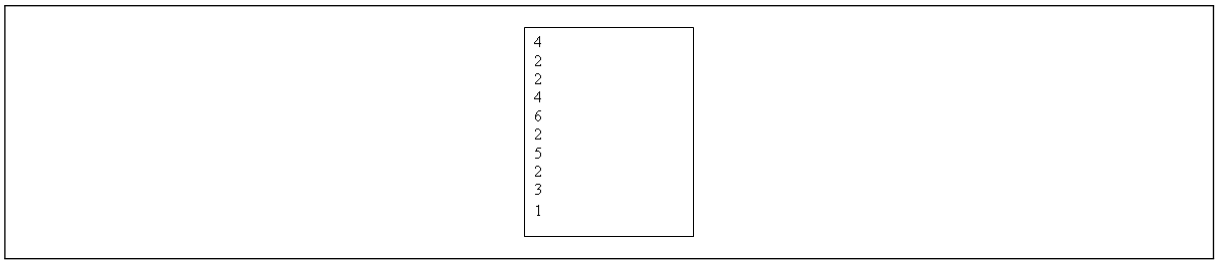
```
#include "random.h"
```

在 dicetest.c 文件中需要使用 #include 行使它能使用 RandomInteger 函数。

```
/*
 * File: dicetest.c
 * ----------------
 * This program simulates rolling a die.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
```

```
 * ------------
 * NTrials -- Number of trials
 */

#define NTrials 10

/* Function prototypes */

int RollDie(void);

/* Main program */

main()
{
  int i;

  for (i = 0; i < NTrials; i++) {
    printf("%d\n", RollDie());
  }
}

/*
 * Function: RollDie
 * Usage: die = RollDie();
 * ---------------------------
 * This function generates and returns a random integer in the
 * range 1 to 6, representing the roll of a six-sided die.
 */

int RollDie(void)
{
  return (RandomInteger(1, 6));
}
```

然后通过运行程序来测试并确保它能工作。运行此程序的结果如下：



再一次可以看到，这些数字都处于正确的范围(1 6)内，也是随机出现的。尽管其中数字 2 比其他数字出现得更频繁，但在统计上有这样的可能性，这些数字的出现概率是相同的，数字 2 出现四次完全是偶然的。

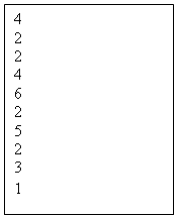
尽管如此，可能就需要通过再运行一次程序来作调查，得到下面的的结果。

第二次运行中，数字 2 还是出现了 4 次，而且我们比较运行结果发现。

事实上，每次运行这个程序，得到的都是完全相同的结果，也就是生成的随机数都是相同的。

尽管这个程序证明了可以产生随机数，但是却在每次运行的过程中产生了完全相同的随机数，而要写出一个有趣的计算机游戏程序，那么程序一定不能有这样的行为。

dicetest.c 程序每次都产生相同的随机数字序列的事实并不是由于 RandomInteger 实现中的任何错误所引起的，而是由标准的 ANSI 库中的 rand 函数的定义所引起的。



尽管调用程序采取特定的行为改变了操作的标准模式,但在每次执行调用它的程序时,rand 函数总是返回相同的数字序列,而且到目前为止,程序在调用库函数 rand 时都会返回相同的结果。

于是便产生了实际运行结果与标准库函数 rand 的理论用途之间的矛盾,因为 rand 函数是模拟非确定性过程的,它应该不可能有这样的行为。又按照 stdlib.h 接口的定义可知,rand 的行为是完全确定的,然而以这种方式定义并实现的 rand 函数有一个非常好的地方:有确定性行为的程序容易调试。

为了说明这个问题,假定刚刚完成一个复杂的程序,作为一个新写的程序,程序有较少的的错误总是好的。在一个复杂的程序中,错误可能相对比较难发现,它们仅出现在很少出现的情况下。假设在运行这个程序(它可能是个游戏程序)的过程,发现该程序有不太正常的行为,但是我们没有注意到相关的征兆,于是这时我们可以重新运行该程序,更仔细得观察。

如果程序确实是以非确定的方式运行,程序第二次运行的行为将与第一次不同。在第一次中出现的错误在第二次中可能不出现。一般来讲,如果程序的行为是真正的随机行为,那么重新产生引起程序出错的条件是非常困难的。另一方面,如果程序以确定的方式工作,那么每次运行都将做同样的工作,这个行为使我们有可能重新产生某个问题。在调试阶段,rand 函数每次都返回同一序列值。

50.3.7 Initialization

尽管 rand 的系统定义有利于调试,但一旦程序开始工作仍能够改变它的行为也是很重要的。而了解如何进行改变需要知道一些 rand 的实现的知识。

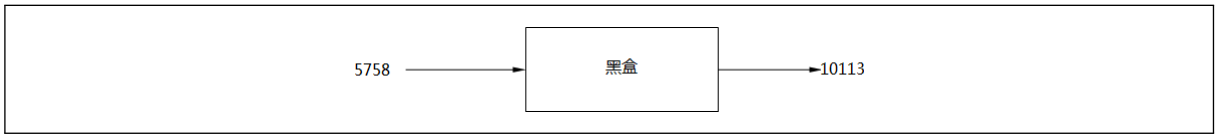
ANSI 库通过记录最近生成的数产生伪随机数。每次调用 rand 时,它取最近生成的一个数对这个数执行一系列的操作,产生下一个数。因为我们不知道到底是在其中做了哪些运算,因此最好把整个操作看成一个黑盒子,旧的数从盒子的一边进入,新的伪随机数从另一边弹出。

在 randtest.c 程序中提供了 rand 内部操作的说明。在运行该程序时,前十次调用 rand 生成的数可能如下所示:

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls t rand:
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

第一次调用 rand 产生 16838。第二次调用对应于将 16838 放入表示内部实现的黑盒的一端,使 5758 从另一边弹出来,见下图。

同样,下一次调用 rand 时,该实现将 5758 放入黑盒子中,返回 10113,见下图。



在每次调用 rand 时,都重复同样的过程。黑盒内部的计算要满足以下条件:

1. 数值在合法的范围内均匀分布。
2. 该序列能持续很长时间才开始重复。

但第一次调用 rand(返回 16838 的那次)的过程又是怎样呢?该实现必须有一个开始点,必须有一个整型数 s 被输入到黑盒子以产生 16838,见下图。

这个用来使整个过程开始的初始值被称为随机数生成器的种子(seed)。每次程序开始时,ANSI 的库实现将初始种子设为一个常量,因此总是产生同一序列值,将种子设为不同的值就能改变这个序列。

要做到这一点,需要调用函数 srand,它将一个新的种子作为参数。为了保证新的种子值在每次程序运行时都变化,标准的做法是用内部系统时钟值作为初始种子。因为时间一直在变,所以随机数序列也将变化。

可以调用函数 time 获取系统时钟的当前值,该函数在 ANSI 库接口 time.h 中定义,然后将返回值转换为整型数。这个技术允许我们利用下列语句将伪随机数生成器初始化为某一个不可预测点:

```
srand((int) time(NULL));
```

该语句只有一行,但基于系统时钟将随机种子设为一个不可预测值的操作是相对模糊的。如果这一行出现在客户程序中,客户必须理解随机数种子的概念、time 函数以及常量 NULL 的意义。

为了简化客户的工作,最好是给这个操作取一个简单的名字(如 Randomize),并把它加到随机数库中。如果做了这样的改变,客户需要做的只是调用

```
Randomize();
```

Randomize() 的实现如下所示:

```
void Randomize(void)
{
  srand((int) time(NULL));
}
```

常见错误:当写一个使用随机数的程序时,在调试时最好不要调用 Randomize。当程序似乎工作正常时,可以在主程序中插入时 Randomize 的调用,从此,程序每次执行都会改变它的行为。

50.3.8 Evaluating

作为设计接口过程的一部分,应该时刻牢记这个设计所要遵循的通用原则。例如,在 random.h 接口的演变过程中,考虑当前的接口是否满足本章前面概括过的五个基本标准是很重要的。

- 它同一吗?
  函数 RandomInteger 和 Randomize 都适合提供访问随机数抽象的同一模式。因此,该接口是同一的。
- 它简单吗?
  通过编写测试程序,比如 dicetest.c 可以有效地检测接口是否简单。此外,还要检查接口是否掩盖了大量的复杂性。例如,调用 RandomInteger 使客户可以不必被规范、换算、截断和转换这些内部步骤所困扰,因为所有的这些操作都由库的实现来完成。同样,Randomize 函数使客户不必知道随机数发生器的种子的内部细节。因此,这个接口确实提供了简化的作用。
- 它充分吗?
  这个问题总是很难回答的,因为它又提出了与之相关的问题:它对什么是充分的?虽然不期望它能满足所有用户的需要,但从这个角度衡量是一个好的想法。
  这个软件包的当前版本对使用随机整数的客户很有用,但对一些需要其他操作的客户就不一定有用了,如对要在一个连续范围内模拟随机实型数的客户来说。因此需要进一步作一些设计工作以满足这个需要。
- 它通用吗?
  通用性问题与充分性问题是密切相关的,但也包含有关接口设计中是否无意识地做了某些与特定的客户领域有关的工作,因此降低了对其他人的可用性的问题。例如,如果直接定义接口使之包括一个模拟掷骰子的函数,而不是让客户在 RandomInteger 的基础上建立这个函数,那么这个接口在设计上应用面就比较狭窄。从而,这也说明现在的接口中的函数似乎满足通用性的要求。
- 它稳定吗?
  与软件包的长期维护相比,稳定性问题在设计阶段并不是一个大问题。在这一点上,重要问题在于接口设计是否以某种方式促进了长期的稳定性。一般来讲,满足其他需求的接口应该尽可能保持稳定,虽然保持这样的稳定性需要对负责库维护的人有某种规定。
  可以确定现在仅有的尚未解决的问题是 random.h 接口没有提供客户所需的所有函数。特别是,在接口的设计分析中建议提供一个随机实数以便提高随机数库对某些客户的可用性的需求,因此,值得再定义一个函数以便提供所需的功能,与 RandomImerger 相对应,可以称为 RandomReal。

50.4 Pseudo-random Real Number

作为 RandomInteger 实现的一部分,可以使用 rand 函数来产生随机实数。

生成随机整数过程的第一步就是生成一个 0 ～ 1 之间的随机浮点数,因此要实现 RandomReal 函数,一种方法是做同样的计算,返回结果。

然而,这样的设计在某种程度上违反了保证库一致性的同一性原则。用这种方法,RandomReal 没有参数,并返回一个处于预先设定范围内的浮点值,因此为了保持一致性,最好让 RandomReal 有同样的设计。如果这样的话,了解 RandomInteger 函数是如何工作的客户就能正确预测 RandomReal 的结构。

RandomInteger 有两个参数,并返回由输入定义的范围内的值。为此,RandomReal 应该有如下所示的原型:

```
double RandomReal(double low, double high);
```

它的实现基本上是由 RandomInteger 实现的前两行组成,只是换算因子是确切的两个端点之间的距离而不是包含在此范围内的整型数个数。

```
/*
* Function:RandomReal
* ---------------------------
* The implementation of RandomReal is similar to that of RandomInteger,
* without the truncation step.
```

```
*/
double RandomReal(double low, double high)
{
  double d;

  d = (double) rand() / ((double) RAND_MAX);
  return (low + d * (high - low));
}
```

50.5 Simulating Probabilistic Event

除了随机实数以外，在模拟随机行为的一般抽象中还有另一种有用的随机变量的类型。

假设要写一个程序，其中需要一个事件以某种随机概率出现。例如，假设这个程序要模拟一条装配线，这条装配线平均每传送 1000 个零件就要出一次故障。用仿真的术语来讲，是每个零件有千分之一的故障率。在数学和统计学中，概率是 0 ～ 1 之间的数，因此这个例子中故障率是 0.001(1/1000)。

在这个例子中，结果只有两种可能：有故障或者没有故障。表示某个条件出现或不出现的只有两个结果的事实用布尔值表示是最恰当的。用 TRUE 表示检测到故障时，它出现的概率是 0.001。

在这种情况下，有一个以某种概率返回 TRUE 的谓词函数是很有用的，于是可以在 random.h 中引入一个函数，称为 RandomChance，可以用下列的代码表示这条装配线：

```
if(RandomChance(.001)){
  printf("A defect has occurred.\n");
}
```

作为第二个实例，可以用 RandomChance 模拟抛硬币落下时的正反面：

```
if(RandomChance(.5)){
  printf("Heads\n");
}else{
  printf("Tails\n");
}
```

这种实现的优点在于它不需要客户理解 rand 函数本身的操作，客户仅依赖于更高级的 random.h 接口中定义的函数，rand 函数的存在仅仅被认为是实现一边的细节问题。

RandomChance 函数的原型和实现都非常简单。作为接口一部分的原型是：

```
bool RandomChance(double p);
```

利用 RandomReal 可以用如下的方式实现 RandomChance。

```
bool RandomChance(double p)
{
  return (RandomReal(0, 1) < p);
}
```

在将 RandomChance 加入到 random.h 接口中时，引入了一个非常重要的问题。

RandomChance 是一个谓词函数，因此它的返回类型是 bool，而 bool 类型不是 C 语言的定义中的一部分，而是在 genlib.h 接口中定义的。为了使编译器读入 random.h 接口时能正确地解释 RandomChance 原型，需要访问 genlib.h 中的 bool 类型的定义。

为了给编译器提供必须的信息，在此接口中应该包括 genlib.h 头文件。

```
#ifndef _random_h
#define _random_h
#include "genlib.h"
```

它指示编译器读取 genlib.h 中的定义，因此编译器在文件后面遇到 RandomChance 时，就已经知道了 bool 类型的定义。

常见错误:每个接口必须包含编译器理解接口本身所需要的任何头文件。然而,接口不包括其对应的实现中所需用到的头文件。这些头文件仅包含在实现该接口的.c 文件中。

例如,为了使用函数 rand、srand 和 time,实现文件 random.c 需要访问 stdlib.h 和 time.h。然而,这些函数仅出现在实现中,在接口中不出现。因此,尽管 random.c 实现中包括了这些头文件,但 random.h 并不包含这些头文件。与之相比,类型 bool 明确地出现在 random.h 接口中,这意味着接口必须包含 genlib.h。

50.6 Random-number Package

完成 random.h 接口定义和对应的 random.c 实现剩下的任务是将增加的新的函数加入到相应的代码版本中,并加入相应的注释,使客户能理解这个接口。

```
/*
 * File: random.h
 * Last modified on Mon Sep 13 10:42:45 1993 by eroberts
 * --------------
 * Library package to produce pseudo-random numbers.
 */

#ifndef _random_h
#define _random_h

#include "genlib.h"

/*
 * Function: Randomize
 * Usage: Randomize();
 * -------------------
 * This function sets the random seed so that the random
 * sequence is unpredictable. During the debugging phase,
 * it is best not to call this function, so that program
 * behavior is repeatable.
 */

void Randomize(void);

/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * ------------------------------------
 * This function returns a random integer in the range
 * low to high, inclusive.
 */

int RandomInteger(int low, int high);

/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * ---------------------------------
 * This function returns a random real number in the
 * half-open interval [low .. high), meaning that the
 * result is always greater than or equal to low but
 * strictly less than high.
 */
```

```
double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * ---------------------------------
 * The RandomChance function returns TRUE with the probability
 * indicated by p, which should be a floating-point number
 * between 0 (meaning never) and 1 (meaning always). For
 * example, calling RandomChance(.30) returns TRUE 30 percent
 * of the time.
 */

bool RandomChance(double p);

#endif


/*
 * File: random.c
 * Last modified on Mon Sep 13 10:42:45 1993 by eroberts
 * --------------
 * Implements the random.h interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "genlib.h"
#include "random.h"

/*
 * Function: Randomize
 * -------------------
 * This function operates by setting the random number
 * seed to the current time. The srand function is
 * provided by the <stdlib.h> library and requires an
 * integer argument. The time function is provided
 * by <time.h>.
 */

void Randomize(void)
{
   srand((int) time(NULL));
}

/*
 * Function: RandomInteger
 * -----------------------
 * This function first obtains a random integer in
 * the range [0..RAND_MAX] by applying four steps:
 * (1) Generate a real number between 0 and 1.
 * (2) Scale it to the appropriate range size.
 * (3) Truncate the value to an integer.
 * (4) Translate it to the appropriate starting point.
 */
```

```
int RandomInteger(int low, int high)
{
   int k;
   double d;

   d = (double) rand() / ((double) RAND_MAX + 1);
   k = (int) (d * (high - low + 1));
   return (low + k);
}

/*
 * Function: RandomReal
 * --------------------
 * The implementation of RandomReal is similar to that
 * of RandomInteger, without the truncation step.
 */

double RandomReal(double low, double high)
{
   double d;

   d = (double) rand() / ((double) RAND_MAX + 1);
   return (low + d * (high - low));
}

/*
 * Function: RandomChance
 * ----------------------
 * This function uses RandomReal to generate a number
 * between 0 and 100, which it then compares to p.
 */

bool RandomChance(double p)
{
   return (RandomReal(0, 1) < p);
}
```

随机数软件包可以为涉及随机数行为的计算机游戏或其他应用提供了一组工具，而且在使用随机数软件包时，不需要记住细节问题。使用随机数库所需要做的是将头文件 random.h 包含在你的程序中，当编译和运行程序时，确保库是可用的。

为了说明此软件包的应用，下面的 craps.c 的程序将模拟一种掷两个骰子的游戏，游戏过程如下：

开始掷两个 6 面的骰子，看所得的总数。然后按第一次掷骰子的结果，分解出下列几种情况：

1. 掷出 2, 3 或 12，第一次掷骰子得到这些结果称为放弃，意味着玩家输了。
2. 掷出 7 或 11，第一次掷骰子得到这些结果称为自然赢，表示玩家赢了。
3. 如果是其他数(4、5、6、8、9 或 10)。在这些情况下，掷到的数称为玩家的点，玩家可以继续掷骰子直到第二次掷到玩家的点或者是 7，如果掷到玩家的点，玩家赢了，如果掷到 7，玩家输了。如果掷到其他数(包括 2、3、11 和 12，这些不再特殊处理)，继续掷骰子直到出现玩家的点或 7。

```
/*
 * File: craps.c
 * -------------
 * This program plays the dice game called craps. For a discussion
 * of the rules of craps, please see the GiveInstructions function.
 */

#include <stdio.h>
```

```
#include "genlib.h"
#include "random.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

void GiveInstructions(void);
void PlayCrapsGame(void);
int RollTwoDice(void);
bool GetYesOrNo(string prompt);

/* Main program */

main()
{
   Randomize();
   if (GetYesOrNo("Would you like instructions? ")) {
      GiveInstructions();
   }
   while (TRUE) {
      PlayCrapsGame();
      if (!GetYesOrNo("Would you like to play again? ")) break;
   }
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * ----------------------
 * This function welcomes the player to the game and gives
 * instructions on the rules to craps.
 */

void GiveInstructions(void)
{
   printf("Welcome to the craps table!\n\n");
   printf("To play craps, you start by rolling a pair of dice\n");
   printf("and looking at the total. If the total is 2, 3, or\n");
   printf("12, that's called 'crapping out' and you lose. If\n");
   printf("you roll a 7 or an 11, that's called a 'natural' and\n");
   printf("you win. If you roll any other number, that number\n");
   printf("becomes your 'point' and you keep on rolling until\n");
   printf("you roll your point again (in which case you win)\n");
   printf("or a 7 (in which case you lose).\n");
}

/*
 * Function: PlayCrapsGame
 * Usage: PlayCrapsGame();
 * ----------------------
 * This function plays one game of craps.
 */

void PlayCrapsGame(void)
{
   int total, point;
```

```
   printf("\nHere we go!\n");
   total = RollTwoDice();
   if (total == 7 || total == 11) {
      printf("That's a natural. You win.\n");
   } else if (total == 2 || total == 3 || total == 12) {
      printf("That's craps. You lose.\n");
   } else {
      point = total;
      printf("Your point is %d.\n", point);
      while (TRUE) {
         total = RollTwoDice();
         if (total == point) {
            printf("You made your point. You win.\n");
            break;
         } else if (total == 7) {
            printf("That's a seven. You lose.\n");
            break;
         }
      }
   }
}

/*
 * Function: RollTwoDice
 * Usage: total = RollTwoDice();
 * -----------------------------
 * This function rolls two dice and returns their sum. As part
 * of the implementation, the result is displayed on the screen.
 */

int RollTwoDice(void)
{
   int d1, d2, total;

   printf("Rolling the dice . . .\n");
   d1 = RandomInteger(1, 6);
   d2 = RandomInteger(1, 6);
   total = d1 + d2;
   printf("You rolled %d and %d -- that's %d.\n", d1, d2, total);
   return (total);
}

/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * ------------------------------------
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response. If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

bool GetYesOrNo(string prompt)
{
   string answer;

   while (TRUE) {
```

```
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
      }
   }
```

程序本身是将自然语言描述的规则直接转换为 C 代码。当通读 craps.c 程序时，应注意下列特征：

该程序包括接口 random.h，因此，它能使用这个库中的函数。此外，这个程序仅应用那个库中的函数，从来没有直接调用 rand(或 srand，time)。随机数序列是调用 Randomize 来初始化的，每次掷骰子是调用 RandomInteger 产生的。

程序被分解成更小的单元，这些单元给出更详细的信息。这种分解有助于突出程序的结构，使我们能理解这些单元是如何组合起来的。

掷两个骰子的问题出现在程序中的多个地方，因此模拟掷两个骰子、显示结果和记住总和这组动作被封装在一个函数 RollTwoDice 中，它也能被用在其他的地方。

Part VIII

Module

第 51 章

Introduction

现代软件开发往往利用模块(Module)作为合成的单位,模块可以理解为一套一致而互相有紧密关连的软件组织,分别包含了程序和数据结构两部份。

模块的接口表达了由该模块提供的功能和调用它时所需的元素,从而可以使模块具有复用性,并允许开发人员之间同时协作、编写及研究不同的模块。

使用 C 语言进行编程时,模块化开发的过程中要注意下面的这些问题:

- 把单个程序分为多个单独模块的重要性。
- 在一个模块里需要在多个函数调用之间保存状态信息。
- 使用全局变量表示在跨函数调用中所需维护的状态信息。
- 过度使用全局变量的危险性。
- 使用 static 关键字来保证一个模块中的函数和全局变量的私有性。

在程序设计过程中,短小的程序可以用来说明程序设计语言的一些特殊方面。关注单个语句格式或者其他一些语言细节可以使程序员能够单独考虑每个概念,并且了解它是如何工作的,而不必涉及大程序中的内在的复杂性。

但是,程序员职业生涯中真正具有挑战性的则正是如何掌握这种复杂性。为了达到这个目的,就必须写一些大的程序,只有在这样的程序中,才可以将一些单独的概念和工具结合起来。

管理大程序的最重要的技术就是逐步求精策略,即当遇到一个需要复杂解决方案的较大的间题时,把问题分解成更为容易解决的几个小部分。

把一个大的问题分解成小片段的策略的方法在程序设计过程的很多阶段都会用到。然而,当程序变得更长的时候,要在一个源文件中处理如此众多的函数会变得困难。就像要一下子理解一个 50 行的函数也是比较困难的一样。

在这两种情况下,可以引入一些额外的结构。比如碰到一个 50 行的函数时,最好的方法是把它分成几个相互调用的小函数来完成任务。而当碰到一个包含 50 个函数的程序时,最好的办法就是把程序再分成几个更小的源文件。每个源文件都包含一组相关的函数。

由整个程序的一部分组成的较小的源文件称为模块(module)。每个独立的模块比整个程序要简单。而且,如果在设计阶段设计得当,可以把同一模块通用化,从而作为许多不同的应用程序一部分。

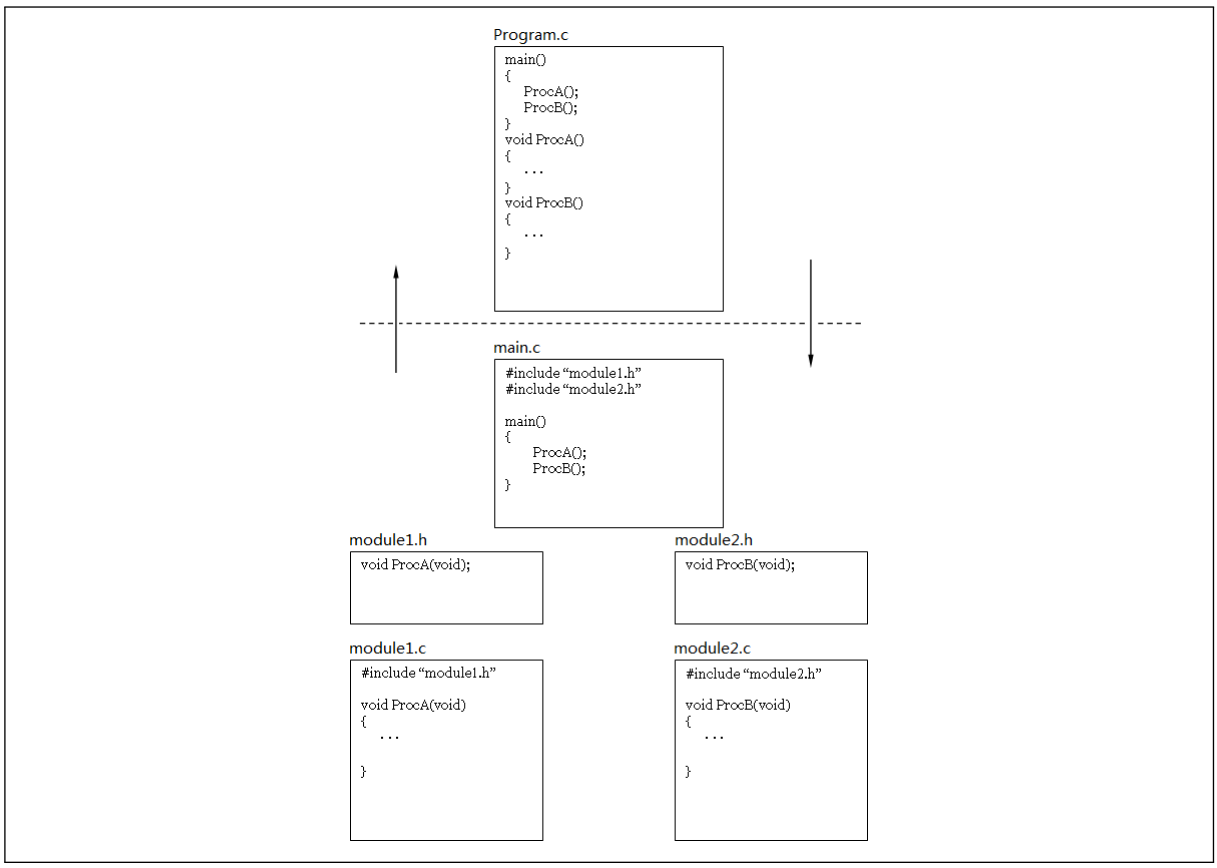
当把一个大的程序分成模块的时候,选择合适的分解方法来减少模块之间相互依赖的程度是很重要的。其中包含 main 函数的模块叫主模块(main module),在分解层次中处于最高层。每个其他模块代表一个独立的抽象,其操作在一个接口中定义。

为了说明模块化设计原则,假设编写了一个如下所示的程序 program.c,这个程序由一个文件组成,该文件包含一个主程序和两个过程 ProcA 和 ProcB。

Program.c

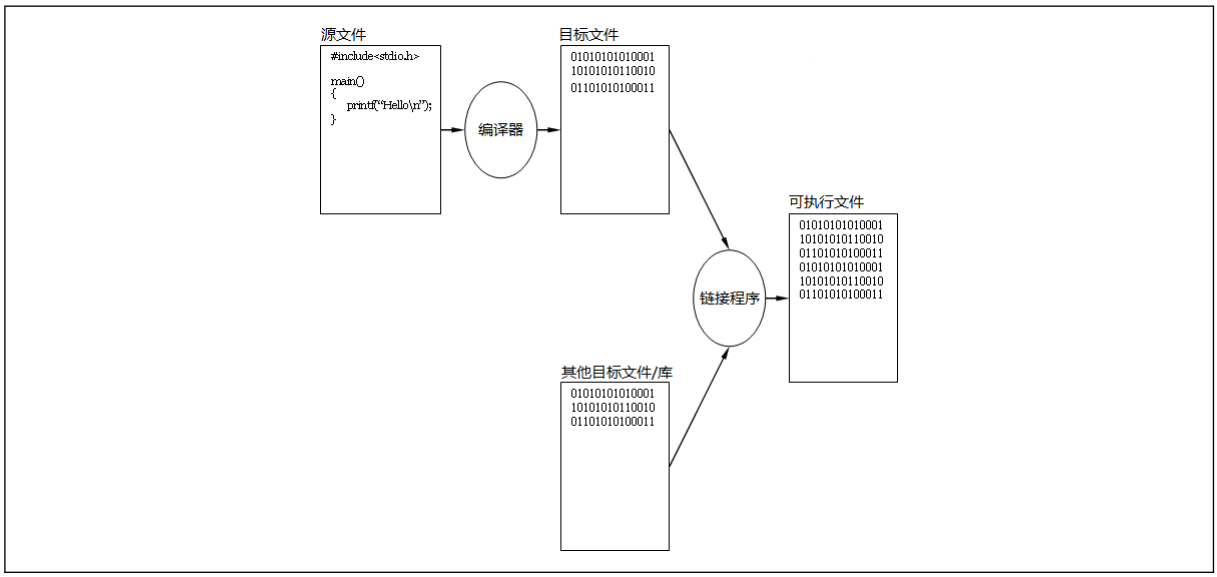
```
main()
{
   ProcA();
   ProcB();
}
void ProcA()
{
   ...
}
void ProcB()
{
   ...
}
```

通过引入模块化开发，可以把这个源文件分解成包括 main 函数的主模块和两个子模块，第一个子模块包括 ProcA，第二个子模块包括 ProcB，如下图所示。



经过分解后，主模块现在只包含整个程序的部分代码。函数 ProcA 和 ProcB 的实现是在单独的模块中。

在程序能够运行前，这些模块的代码必须被链接成一个可执行的程序。把程序片段链接在一起的任务是程序执行的标准流程的一部分，如下图所示。



从某种意义上说，每个子模块都作为主模块的库。和任何库一样，每个模块必须有一个接口，为编译器提供关于模块所包含的函数的必要信息。在上述的模块化开发实例中，这些接口表示为头文件 module1.h 和 module2.h。

在更典型的应用中,每个模块包含一系列相关的函数,这些函数可以形成一个可容易识别的作为一个整体的应用的组件。把一个程序分成多个模块的技术称为模块化开发(modular development)。

51.1 Library

库(Library)[1]就是一种代码复用的很好的形式,程序员可以创建内部抽象以便程序的部分代码可被复用,或者直接创建一个自定义库给自己使用。

包含常用功能的多个目标文件通常会被组合为单一库文件,这样库文件就可以被连接进多个应用程序。其中,在编译时被连接的库称为静态库,在应用程序运行时被加载的库称为动态库。

编写库是进行代码复用最常见的方法。很多共通的操作,比如文件的读/写、操作系统信息的获取的动作都会被封装在库中,然后由软件开发人员来调用。

库所提供的操作都是经过充分测试的,但同时库无法对它提供的操作的具体实现进行调整,同时库也要求软件开发人员花大量时间去学习它的用法。

51.2 Program

高级语言使用特殊的保留关键字来定义变量(数据的内存位置)、创建循环和处理程序的输入和输出等。但是,处理器实际上根本不知道如何处理高级语言代码,因此必须通过某种机制把高级语言代码转换为处理器能够处理的简单的指令码形式。

现在,程序员可以使用简单的关键字和术语定义一个或多个指令码,而不是使用原始的处理器指令码,这使程序员可以把精力集中在应用程序的逻辑上,而不必担心底层处理器指令码的细节问题。

许多编译器都包含汇编器的处理过程,虽然不是所有编译器都这样,因此高级语言程序在连接为可执行程序之前,都会被编译为汇编语言程序——汇编语言是处理器指令的抽象。

编译器把高级语言程序转换为处理器能够执行的指令码,但大多数编译器会产生一个中间步骤,此时编译器把源代码转换为汇编语言代码。接下来使用汇编器把汇编语言代码转换为指令码。

当把 C/C++ 程序源代码转换为汇编语言后,编译器会使用汇编器生成链接器需要使用的指令码。如果在汇编和链接之间检查从 C 或 C++ 源代码生成的汇编语言代码,就可以优化某些汇编语言代码部分,从而把代码汇编为新的指令码。

高级语言又继续演化为编译型语言和解释型语言等,因此相同的程序设计语言的不同实现可能是编译的,也可能是解释的。

51.2.1 Construction

在构建简单 C 语言程序时,C 语言为编排程序元素制定了相应的方法。

- 预处理指令直到行出现时才会起作用。
- 类型名需要先定义才能使用。
- 变量需要先声明才能被引用。
- 函数声明和全局变量声明要先于函数定义和引用。
- 函数定义或声明要先于函数调用。

为了遵守上述规则,下面是一种可能的构建程序的编排顺序。

1. #include 指令

   #include 指令带来的信息将可能在程序中的某些地方都需要,因此最先放置调用库的指令。

2. #define 指令

   #define 指令产生宏,对这些宏的使用通常遍布整个程序。

[1]在计算机科学中,库是用于开发软件的子程序集合。库和可执行文件的区别是,库不是独立程序,它们是向其他程序提供服务的代码。

3. 类型定义
   全局变量的声明可能会引用扩展类型定义，因此类型定义放置在全局变量的上部是合理的。
4. 全局变量声明
   全局变量声明对于跟随其后的所有函数都是有效的。
5. 除 main 函数之外所有的函数原型
   在编译器读入函数原型之前调用函数可能会产生问题，通过预先声明除 main 函数之外的所有扩展函数可以避免这些问题，而且有助于定位函数的起始点。
6. main 函数的定义
7. 其他函数的定义

另外，在每个函数定义前或接口中的函数原型前放置注释，可以给出函数名和返回值、描述函数的目的、明确每个形式参数的含义，以及列举任何的副作用。

在下面的示例程序中将读取一手 5 张的牌，然后根据下列类别把牌进行分类(列出的顺序从最好到最坏)。

- 每张牌都有花色(方块、梅花、红桃和黑桃)和等级(2、3、4、5、6、7、8、9、10、J、Q、K 和 A)。
- 不允许使用王牌(纸牌中可当任何点数用的一张)。
- 假设 A 是最高等级的。
  1. 同花顺的牌(即顺序相连又都是同花色)；
  2. 4 张相同的牌(4 张牌等级相同)；
  3. 3 张相同和 2 张相同的牌(3 张牌是同样的花色，而另外 2 张牌是同样的花色)；
  4. 同花色的牌(5 张牌是同花色的)；
  5. 同顺序的牌(5 张牌的等级顺序相连)；
  6. 3 张相同的牌(3 张牌的等级相同)；
  7. 2 对子；
  8. 1 对(2 张牌的等级相同)；
  9. 其他牌(任何其他情况的牌)。

如果一手牌有两种或多种类别，程序将选择最好的一种。为了便于输入，可以把牌的等级和花色简化(字母大小写不敏感)。

- 等级：2 3 4 5 6 7 8 9 t j q k a
- 花色：c d h s

如果用户输入非法牌或者输入同张牌 2 次，程序将把此牌忽略并产生报错信息，然后要求输入另外一张牌。如果输入为 0 而不是一张牌，程序终止。

根据自顶向下、逐步求精的策略，可以把这个程序分解为 3 个子任务。

- 读入一手 5 张牌；
- 分析一手牌的对、顺序和其他；
- 显示一手牌的分类。

这样就可以把程序分为 3 个函数来分别完成上述 3 个任务，即 read_cards 函数、analyze_hand 函数和 print_result 函数。

主程序只负责在无限循环中调用上述这些函数，这些函数需要共享大量的信息，因此需要让它们通过全局变量来进行交互。其中，read_cards 函数将存储全局变量的信息，接下来 analyze_hand 函数将检查这些全局变量，并把结果值存储到其他全局变量并传递给 print_result 函数。

基于上述初步分析可以规划出程序的主体如下：

```
/* Classifies a poker hand */

/* #include directives */

/* #define directives */
```

```
/* declarations of external variables */

void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*************************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *     repeatedly.                              *
 *************************************************************/
main()
{
  for (;;) {     /* infinite loop */
    read_cards();
    analyze_hand();
    print_result();
  }
}

/*************************************************************
 * read_cards: Reads the cards into the external *
 *          variables num_in_rank and num_in_suit; *
 *          checks for bad cards and duplicate cards. *
 *************************************************************/

void read_cards(void)
{
  ...
}

/*************************************************************
 * analyze_hand: Determines whether the hand contains a *
 *           straight, a flush, four-of-a-kind, *
 *           and/or a three-of-a-kind; determines the *
 *           number of pairs; stores the results into *
 *           the external variables straight, flush, *
 *           four, three, and pairs.        *
 *************************************************************/
void analyze_hand(void)
{
  ...
}

/*************************************************************
 * print_result: Notifies the user of the result, using *
 *           the external variables straight, flush, *
 *           four, three, and pairs.        *
 *************************************************************/
void print_result(void)
{
  ...
}
```

主程序中最主要的问题是如何表示一手牌，其中在分析这手牌期间，analyze_hand 函数将需要知道每个等级和每个花色的牌的数量。

把 0 ～ 12 的数编码为等级，把 0 ～ 3 的数编码为花色后，建议分别使用数组 num_in_rank 存储等级为 r 的牌的数量，num_in_suit 存储花色为 s 的牌的数量。

为了便于 read_cards 函数检查重复的牌，还需要第 3 个数组 card_exists，这样每次读取等级为 r 且花色为 s 的牌时，read_cards 函数都会检查 card_exists[r][s] 的值是否为 TRUE。如果为 TRUE，就表示此牌以及输入过，否则 read_cards 函数把 TRUE 赋值给 card_exists[r][s]。

read_cards 函数和 analyze_hand 函数都需要访问数组 num_in_rank 和 num_in_suit，因此这两个数组都必须是全局变量。数组 card_exists 只用于 read_cards 函数，因此该数组是 read_cards 函数的局部变量。

```
/* Classifies a poker hand */

#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5
#define TRUE 1
#define FALSE 0

typedef int Bool;

int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
Bool straight, flush, four, three;
int pairs; /* can be 0, 1, or 2 */

void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*********************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *     repeatedly.                                *
 *********************************************************/
main()
{
  for (;;) {     /* infinite loop */
    read_cards();
    analyze_hand();
    print_result();
  }
}

/*********************************************************
 * read_cards: Reads the cards into the external *
 *          variables num_in_rank and num_in_suit; *
 *          checks for bad cards and duplicate cards. *
 *********************************************************/

void read_cards(void)
{
  Bool card_exists[NUM_RANKS][NUM_SUITS];
  char ch, rank_ch, suit_ch;
  int rank, suit;
  Bool bad_card;
  int cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
```

```
    for (suit = 0; suit < NUM_SUITS; suit++)
      card_exists[rank][suit] = FALSE;
  }

  for (suit = 0; suit < NUM_SUITS; suit++)
    num_in_suit[suit] = 0;

  while (cards_read < NUM_CARDS) {

    bad_card = FALSE;

    printf("Enter a card: ");

    rank_ch = getchar();
    switch (rank_ch) {
      case '0':      exit(EXIT_SUCCESS);
      case '2':      rank = 0; break;
      case '3':      rank = 1; break;
      case '4':      rank = 2; break;
      case '5':      rank = 3; break;
      case '6':      rank = 4; break;
      case '7':      rank = 5; break;
      case '8':      rank = 6; break;
      case '9':      rank = 7; break;
      case 't': case 'T': rank = 8; break;
      case 'j': case 'J': rank = 9; break;
      case 'q': case 'Q': rank = 10; break;
      case 'k': case 'K': rank = 11; break;
      case 'a': case 'A': rank = 12; break;
      default:       bad_card = TRUE;
    }

    suit_ch = getchar();
    switch (suit_ch) {
      case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:       bad_card = TRUE;
    }

    while ((ch = getchar()) != '\n')
      if (ch != ' ') bad_card = TRUE;

    if (bad_card)
      printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
      printf("Duplicate card; ignored.\n");
    else {
      num_in_rank[rank]++;
      num_in_suit[suit]++;
      card_exists[rank][suit] = TRUE;
      cards_read++;
    }
  }
}

/**********************************************************
```

```
 * analyze_hand: Determines whether the hand contains a *
 *           straight, a flush, four-of-a-kind, *
 *           and/or a three-of-a-kind; determines the *
 *           number of pairs; stores the results into *
 *           the external variables straight, flush, *
 *           four, three, and pairs.        *
 ************************************************************/
void analyze_hand(void)
{
  int num_consec = 0;
  int rank, suit;

  straight = FALSE;
  flush = FALSE;
  four = FALSE;
  three = FALSE;
  pairs = 0;

  /* check for flush */
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
      flush = TRUE;

  /* check for straight */
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank]; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = TRUE;
    return;
  }

  /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = TRUE;
    if (num_in_rank[rank] == 3) three = TRUE;
    if (num_in_rank[rank] == 2) pairs++;
  }
}
/************************************************************
 * print_result: Notifies the user of the result, using *
 *           the external variables straight, flush, *
 *           four, three, and pairs.        *
 ************************************************************/
void print_result(void)
{
  if (straight && flush) printf("Straight flush\n\n");
  else if (four)   printf("Four of a kind\n\n");
  else if (three &&
         pairs == 1) printf("Full house\n\n");
  else if (flush)  printf("Flush\n\n");
  else if (straight) printf("Straight\n\n");
  else if (three)  printf("Three of a kind\n\n");
  else if (pairs == 2) printf("Two pairs\n\n");
  else if (pairs == 1) printf("Pair\n\n");
  else             printf("High card\n\n");
}
```

51.2.2 Compilation

大多数程序都是通过编程语言的一般语句来表示程序逻辑来创建的，然后通过编译器把源代码编译为可以在处理器上运行的指令码集合。

源代码中的每一个代码行都和要运行应用程序的特定处理器相关的一个或多个指令码相匹配。例如，下面的 C 语言代码会被编译为对应的 IA-32 指令码。

```
int main()
{
  int i = 1;
  exit(0);
}
```

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

在上述这个过程中会生成一个中间文件，称为目标代码文件（object code file）。目标代码文件包含表示应用程序功能核心的指令码，但目标代码文件本身不能由操作系统运行。

通常，宿主操作系统需要的是可执行文件，并且开发程序时可能需要其他目标文件的功能，所以需要另一个步骤来添加这些功能。

gcc 使用-S 参数让编译器创建中间汇编语言文件，从而可以显示编译器实现 C 语言程序的汇编语言指令，可以针对性地对汇编语言程序进行优化。

```
#include <stdio.h>

main()
{
  printf("hello, world.\n");
}
$ gcc -S hello.c
$ cat hello.s
      .file "hello.c"
      .section    .rodata
.LC0:
      .string "hello,world."
      .text
      .globl main
      .type main, @function
main:
.LFB0:
      .cfi_startproc
      pushq %rbp
      .cfi_def_cfa_offset 16
      .cfi_offset 6, -16
      movq  %rsp, %rbp
      .cfi_def_cfa_register 6
      movl  $.LC0, %edi
      call  puts
      popq  %rbp
```

```
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident "GCC: (GNU) 4.8.2 20131212 (Red Hat 4.8.2-7)"
        .section    .note.GNU-stack,"",@progbits
```

另外,如果需要使用调试器监视程序,就必须使用-g 参数。对于 Linux 系统,-gstabs 参数在可执行程序中为 GNU 调试器提供额外的调试信息,以便让调试器知道指令码与源代码文件的位置如何关联。

```
$ gcc -gstabs -o hello hello.c
$ gdb hello
```

可以使用不同的命令行格式调用 GNU 编译器,具体参数取决于要编译的源代码和操作系统的底层硬件。

Table 51.1: gcc 命令行参数

参数	描述
-c	编译或者汇编代码,但是不进行连接
-S	编译后停止,但是不进行汇编
-E	预处理后停止,但是不进行编译
-o	指定要使用的输出文件名
-v	显示每个编译阶段使用的命令
-std	指定使用的语言标准
-g	生成调试信息
-pg	生成 gprof 制作简档所需的额外代码
-O	优化可执行代码
-W	设置编译器警告消息级别
-pedantic	按照 C 标准发布强制性诊断清单
-I	指定包含文件的目录
-L	指定库文件的目录
-D	预定义源代码中使用的宏
-U	取消任何定义了的宏
-f	指定用于控制编译器行为的选项
-m	指定与硬件相关的选项

GNU 编译器 gcc 不仅提供了编译 C/C++ 程序源代码的功能,还提供了在系统上运行 C 和 C++ 应用程序所需的库,这些库也和运行在系统上的其他程序兼容。

51.2.3 Interpretation

编译语言自己运行在处理器上,而解释语言则相反。

解释语言是由单独的程序读取和运行的,这个单独的程序是应用程序的宿主,宿主在进行处理时读取并解释程序代码,生成适应处理器的正确指令码。

显然,使用解释语言的不足是速度慢,程序没有直接被编译为运行在处理器上的指令码,而是由宿主程序读取程序代码的每一行并执行必须的功能,这样宿主程序读取代码并解释执行代码所花费的时间会给应用程序的执行增加额外延迟。

解释语言程序的优点在于其便利性，免去了以前每次对程序作出改动后必须重新编译程序以及链接代码库等的操作，从而可以快速对源代码文件进行修改并重新运行程序以检查错误。

另外，使用解释语言时，解释器可以自动地确定核心代码需要包含的功能。

后来，现代程序设计语言的发展模糊了编译语言与解释语言之间的界限，没有一种特定的语言可以被确定为属于某一分类。相反地，不同高级语言的实现是可以分类的，例如很多 BASIC 语言实现需要解释器把 BASIC 代码解释为可执行代码，但也有很多 BASIC 实现允许把 BASIC 程序编译为可执行指令码。

混合语言把编译语言的特性和解释语言的通用性结合在一起，例如 Java 语言程序被编译为字节码(byte code)的形式。

Java 字节码和处理器指令码类似，但是它本身不和当前的任何处理器兼容，它必须通过 Java 虚拟机(Java Virtual Machine)进行解释。

Java 虚拟机运行在宿主计算机上，Java 字节码可以通过任何类型的宿主计算机上的 JVM 来运行，而且不同的平台可以使用它们自己特定的 JVM。

JVM 可以解释 Java 字节码并产生相同的输出，从而无需从原始的源代码进行重新编译。

与其他程序设计语言不同的是，汇编语言汇编器没有被标准化，因此并不是所有汇编器都使用一种标准格式。

虽然基础是一样的，但是不同的汇编器可能使用不同的语法编写程序语句，因此首先要决定在现有环境中希望(或者需要)使用什么类型的汇编语言，或者通过交叉编译或直接在新的平台上重新编译，系统会自动使用目标处理器的代码重写程序，这样才能使高级语言程序移植到其他操作系统和其他处理器平台上。

当然，实践中引入里成熟的框架后，使用操作系统 API 的复杂程序要简单地重新编译为另一种平台地代码也是困难的。

另外，高级语言都遵循相关的标准规范，从而使得使用标准编译器在一种操作系统和处理器上编译源代码得到的结果和在另一种操作系统和处理器上编译的结果应该是相同的。

51.3 Instruction

虽然高级语言的出现简化了程序开发，但是这并不一定意味着产生的程序更有效率。

在操作系统的最低层，所有计算机处理器都按照其内部定义的二进制指令操作数据，这些预置的指令代码定义了处理器如何处理数据。

不同的处理器包含不同类型的指令码(instruction code)集合——指令集(instruction set)，通常按照处理器芯片支持的指令码的数量和类型对它们进行分类。

为了提高可移植性和标准化，很多编译器被编码为“最小公分母”，创建指令码时可能不使用某些处理器独有的特殊指令码，因此也就不能创建更快的应用程序。

例如，很多新的处理器都提供的一个特性是高级的数学处理指令码，通过使用长度超长的字节表示数字(64 位或 128 位)，这些指令码可以提高处理复杂数学表达式的速度，但作为“最小公分母”的编译器却没有利用这些高级指令码的优势，因此在许多对执行速度要求极高的环境中就需要使用汇编语言等来对应用程序进行优化[2]

高级语言依赖编译器把程序逻辑转换为处理器运行的指令码，这无法保证生成的指令码是否就是实现程序逻辑的最有效率的方法。

对于希望提高程序效率或者希望对处理器如何处理程序具有更大控制力的程序员来说，汇编语言提供了另外一种选择。

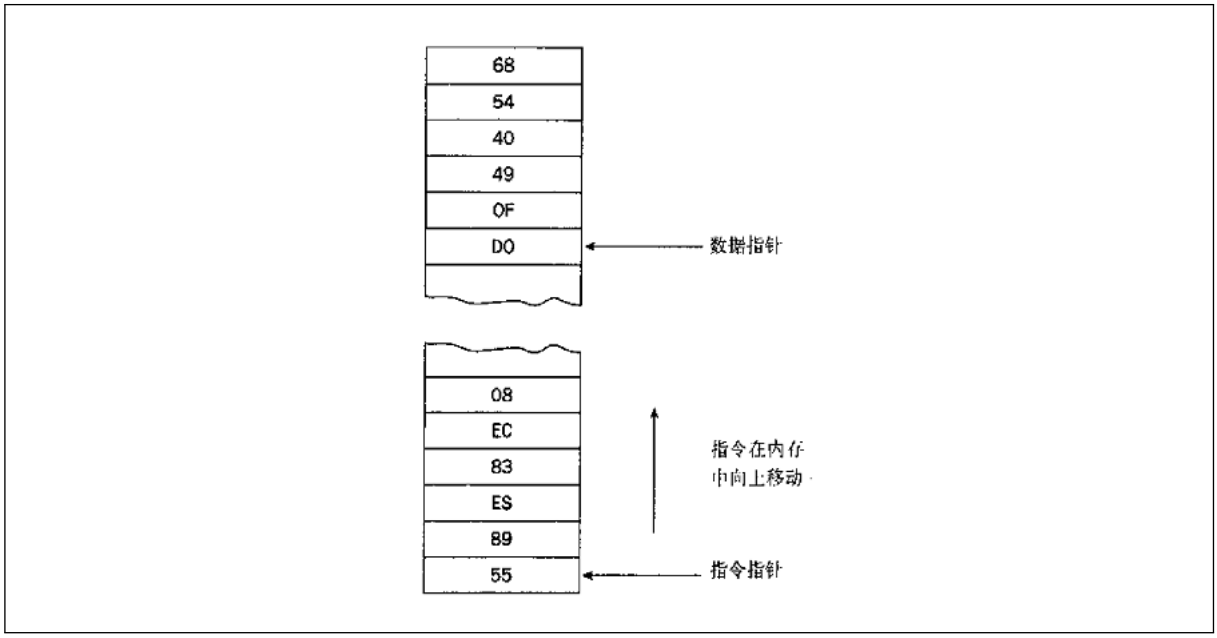
汇编语言允许程序员直接创建指令码程序，或者通过助记符来尽可能地靠近原始处理器指令码，而无需担心处理器上众多指令码集合的组合，这就向程序员同时提供了高级语言的简易性和使用指令码提供的控制能力。

[2]当然，提高应用程序执行速度的首要步骤是确保使用了最有效的算法，对不良算法进行优化是比不上使用更快的算法的。

虽然不同类型的处理器可能包含不同类型的指令码，但它们处理指令码程序的方式是类似的。例如，在执行程序时，处理器会读取存储在内存中的程序指令，每条指令可能包含一个或多个字节的信息，这些信息知识处理器完成特定的任务。

计算机处理的指令和数据都是存储在内存中，并且由内存读入处理器进行处理，指令和内存的编码是一致的。

为了区分数据和指令，需要使用专门的指针（pointer）来帮助处理器跟踪数据和指令在内存中的存储位置。



- 指令指针（instruction pointer）[?] 用于帮助处理器了解哪些指令码已经处理过了，以及接下来要处理的是哪条指令。有些专门的指令能够改变指令指针的位置，比如跳转到程序的特定位置。
- 数据指针（data pointer）用于帮助处理器了解内存中数据区域的起始位置。这个区域称为堆栈（stack），当新的数据元素被放入到堆栈中时，指针在内存中“向下”移动。当数据被读取出堆栈时，指针在内存中“向上”移动。

每条指令都必须至少包含 1 个字节的操作码（operation code），操作码定义处理器应该完成什么操作。例如，下面的这些指令码字节（十六进制）

```
C7 45 FC 01 00 00 00
```

会通知 Intel IA-32 系列处理器[3]把十进制值 1 加载到一个处理器寄存器定义的内存偏移位置。指令码包含若干信息片段，它们明确地定义处理器要完成什么操作。在处理器完成了一个指令码集合的处理之后，它读取内存中的下一个指令码集合（就是指令指针所指的）。在内存中，指令必须按照正确的格式和顺序排列，这样使处理器能够正确地按顺序执行程序代码。

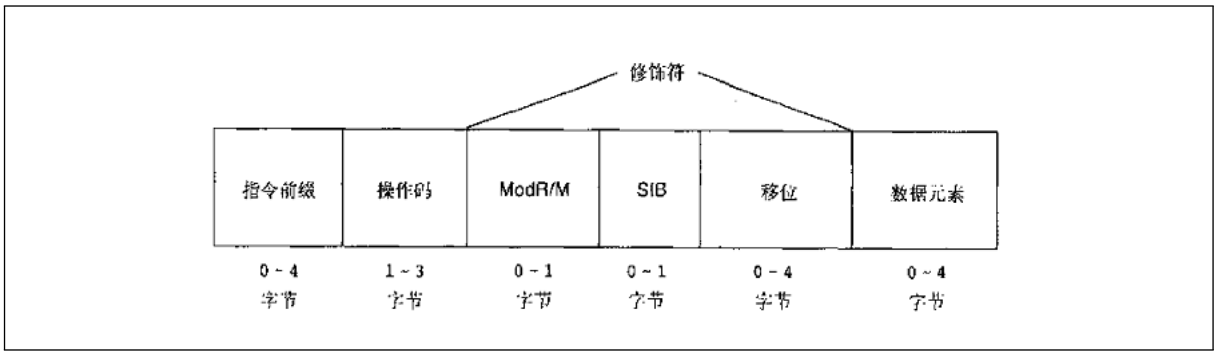
每个处理器系列都具有其自己的预定义好的指令集，它们定义所有可用的功能。

IA-32 指令码格式由 4 个主要部分构成：

- 可选地指令前缀
- 操作码（opcode）
- 可选地修饰符
- 可选地数据元素

每个部分都用于完整地为处理器定义要执行的特定指令。

[3]Intel Pentium 系列处理器不是使用 IA-32 指令码格式的唯一处理器芯片系列，AMD 公司生产的处理器也完全兼容 Intel IA-32 指令码格式。



51.3.1 opcode

IA-32 指令码格式中唯一必须的部分就是操作码。每个指令码都必须包含操作码，它定义了处理器执行的基本功能或任务。

操作码的长度在 1 到 3 字节之间，它唯一地定义要执行的功能。例如，2 字节的操作码 OF A2 定义了 IA-32 CPUID 指令。当处理器执行这个指令码时，它返回不同寄存器中关于微处理器的特定信息，然后程序员可以使用其他的指令码从处理器寄存器中提取信息，以便确定运行程序的微处理器的类型和型号。

寄存器是处理器芯片之内的组件，用于临时存储处理器正在处理的数据。

51.3.2 prefix

指令前缀可以包含 1 个到 4 个修改操作码行为的 1 字节前缀。

按照前缀的功能，这些前缀被分为 4 个组，这 4 个前缀组如下：

- @a1@ 锁定前缀和重复前缀
  - 锁定前缀表示指令将独占的使用共享内存区域。这对于多处理器和超级线程系统非常重要。
  - 重复前缀用于表示重复的功能(常常在处理字符串时使用)。
- @a2@ 段覆盖前缀和分支提示前缀
  - 段覆盖前缀定义可以覆盖定义了的段寄存器的指令。
  - 分支提示前缀尝试向处理器提供程序在条件跳转语句中最可能的路径的线索(这同预报分支的硬件一起使用)。
- @a3@ 操作数长度覆盖前缀

  操作数长度覆盖前缀通知处理器，程序将在这个操作码之内切换 16 位和 32 位的操作数长度。这使程序可以再使用大长度的操作码时警告处理器，帮助加快对寄存器的数据赋值。
- @a4@ 地址长度覆盖前缀

  地址长度覆盖前缀通知处理器，程序将切换 16 位和 32 位的内存地址。这两种长度都可以被声明为程序的默认长度，这个前缀通知处理器程序将切换到另一个长度

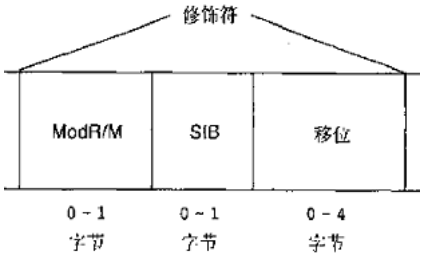
修改操作码时，每个组的前缀一次只能使用一个(因此最多有 4 个前缀字节)。

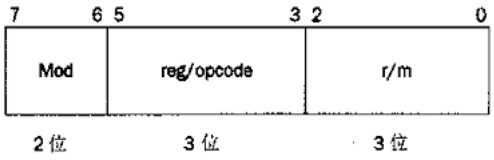
51.3.3 modifier

一些操作码需要另外的修饰符来定义执行的功能中涉及到什么寄存器和内存位置。修饰符包含在 3 个单独的值中：

1. @a1@ 寻址方式说明符(ModR/W)字节

   ModR/W 字节由 3 个字段的信息构成，如下：





- mod 字段和 r/m 字段一起使用,用于定义指令中使用的寄存器或者寻址模式。在指令中,可能的寻址模式有 24 个,加上 8 个可以使用的通用寄存器,所以有 32 个可能值。
- reg/opcode 字段用于运行使用更多的 3 位进一步定义操作码功能(比如操作码子功能),或者可以用于定义寄存器。
- r/m 字段用于定义用作该功能的操作数的另一个寄存器,或者可以把它和 mod 字段组合在一起定义指令的寻址模式。

2. @a2@ 比例-索引-基址(SIB)字节

   SIB 字节也由 3 个字段的信息构成,如下:

- 比例字段指定操作的比例因子。
- 索引字段指定内存访问中用作索引寄存器的寄存器。
- 基址字段指定用作内存访问的基址寄存器的寄存器。

   ModR/M 和 SIB 字节的组合创建一个表,它可以定义用于访问数据的众多可能的寄存器组合和内存模式。

3. @a3@ 1,2 或 4 个的地址移位字节

   地址移位字节用来指定对于 ModR/W 和 SIB 字节中定义的内存位置的偏移量。可以使用它作为基本内存位置的索引,用于存储或者访问内存的数据。

51.3.4 data

指令码的最后一部分是该功能使用的数据元素。

一些指令码从内存位置或者处理器寄存器读取数据,而一些指令码在其本身之内包含数据。这个值经常被用于表示静态数字值(比如要加的数字)或者内存位置。根据数据长度,这个值可以包含 1,2 或者 4 字节的信息。

例如,下面的指令定义操作码 C7,这个操作码是把值传送到内存位置的指令。

```
C7 45 FC 01 00 00 00
```

接下来,内存位置由修饰符 45 FC 定义(它定义从 EBP 寄存器中的值(值 45)指向的内存位置开始的 4 字节(值 FC)),最后 4 字节定义存储到这个内存位置的整数值(在这个示例中这个值是 1)。

从这个例子可以看出,值 1 被写为 4 字节的十六进制值 01 00 00 00,数据流中的字节的顺序取决于使用的处理器的类型。

IA-32 处理器使用“小尾数(little-endian)”表示法,其中低值的字节首先出现(当从左到右读时),其他处理器使用“大尾数(big-endian)”表示法,其中高值字节首先出现。使用汇编语言指定数据和内存位置值时,这一概念非常重要。

GNU objdump 程序不仅能够显示汇编语言代码,还能够显示生成的原始指令码。

```
$ gcc -c hello.c
$ objdump -d hello.o
hello.o:  file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
  0:  55                push %rbp
  1:  48 89 e5          mov  %rsp,%rbp
  4:  bf 00 00 00 00    mov  $0x0,%edi
  9:  e8 00 00 00 00    callq e <main+0xe>
  e:  5d                pop  %rbp
  f:  c3                retq
```

这里,程序中引用的内存地址被标记为零,说明在链接器链接应用程序并且使其准备好在系统上执行之前,这些值还无法确定下来。

除了处理目标代码文件外,objdump 还可以解码很多不同类型的二进制文件。

参数	描述
-a	如果是存档文件,则显示存档头信息
-b	指定目标代码文件的目标代码格式
-C	将低级符号还原为用户级别的符号
-d	把目标代码反汇编为指令码
-D	把所有段反汇编为指令码(包括数据)
-EB	指定大尾数目标文件
-EL	指定小尾数目标文件
-f	显示每个文件头的摘要信息
-G	显示调试段的内容
-h	显示每个文件段头的摘要信息
-i	显示所有架构和目标格式的清单
-j	只显示指定段的信息
-l	使用源代码行号标记输出
-m	指定进行反汇编时使用的架构
-p	显示目标文件格式特有的信息
-r	显示文件中的重定位条目
-R	显示文件中的动态重定位条目
-s	显示指定段的完整内容
-S	交错显示源代码和反汇编后的代码
-t	显示文件的符号表条目

参数	描述
-T	显示文件的动态符号表条目
-x	显示文件所有可用的头信息
--start-address	开始显示在指定地址上的数据
--stop-address	停止显示在指定地址上的数据

51.4 Assembler

汇编语言程序使用助记符(mnemonics)表示指令码,汇编器可以很容易地把助记符转换为用于运行应用程序的处理器的原始指令码。

汇编语言程序由 3 个组件构成,它们用于定义程序操作。

- 操作码助记符
- 数据段
- 命令

其中,汇编语言程序的核心是用于创建程序的指令码。

下面是把汇编语言程序 test.s 转换为 test.o 的示例。

```
as -o test.o test.s
```

这个命令创建目标文件 test.o,其中包含汇编语言程序的指令码。

为了简化指令码的编写,汇编器把助记符词汇和指令码功能(比如传递或添加数据元素)等同对待,因此下面的指令码示例可以写成相应的汇编语言代码。

```
55                                      push  %ebp
89 E5                                   mov   %esp, %ebp
83 EC 08                                sub   $0x8, %esp
C7 45 FC 01 00 00 00                    movl  $0x1, -4(%ebp)
83 EC 0C                                sub   $0xc, %esp
6A 00                                   push  $0x0
E8 D1 FE FF FF                          call  8048348
```

不同的汇编器使用不同的助记符表示指令码,相同处理器的指令码集合在不同的汇编器中也可能是不同的。

GNU 汇编器在处理汇编语言代码时遵循 AT&T 助记符语法,但 Intel 使用的是另外的语法,二者大多数区别出现在特定的指令格式中。

- AT&T 使用 $ 表示立即操作数,而 Intel 的立即操作数是不需要界定的。
- AT&T 在寄存器名称前加上前缀%,而 Intel 不这样做。
- AT&T 语法处理源和目标操作数时使用相反的顺序。例如把十进制数 4 传送给 EAX 寄存器,AT&T 的语法是 movl $4, %eax,而 Intel 的语法是 mov eax, 4。
- AT&T 语法在助记符后面使用一个单独的字符来引用操作中使用的数据长度,而 Intel 语法中数据长度被声明为单独的操作数,因此 AT&T 的指令 movl $test, %eax 等同于 intel 语法的 mov eax, dword ptr test。
- 长调用和跳转使用不同语法定义段和偏移值。AT&T 语法使用 ljmp $section, $offset,而 Intel 语法使用 jmp section:offset。

根据操作系统使用的硬件平台类型,as 的命令行参数是不同的,下面列出的是对于所有硬件平台都通用的命令行参数。

Table 51.3: GNU ASsembler 命令行参数

参数	描述
-a	指定输出中包含哪些清单
-D	用于向下兼容,已被忽略
`--defsym`	在汇编源代码之前定义符号和值
-f	快速汇编,跳过注释和空白
`--gstabs`	包含每行源代码的调试信息
`--gstabs+`	包含专门的 gdb 调试信息
-I	指定搜索包含文件的目录
-J	不警告带符号溢出
-K	用于向下兼容,已被忽略
-L	在符号表中保存本地符号
`--listing-lhs-width`	设置输出数据列的最大宽度
`--listing-lhs-width2`	设置连续行的输出数据列的最大宽度
`--listing-rhs-width`	设置输入源代码行的最大宽度
`--listing-cont-lines`	设置输入的单一行在清单中输出的最大行数
-o	指定输出目标文件的名称
-R	把数据段合并进文本段
`--statistics`	显示汇编使用的最大时间和总时间
-v	显示 as 的版本号
-W	不显示警告信息
`--`	对于源文件使用标准输入

GNU 汇编器提供了使用 Intel 语法替换 AT&T 语法的方法,即使用.intel_syntax 通知 as 使用 Intel 语法汇编指令码助记符,而不使用 AT&T 语法。

除了指令码外,大多数程序还需要使用数据元素来定义程序当中用到的变量和常量值的内存位置。

高级语言使用变量来定义保存数据的内存段,例如在下面的 C 语言示例程序中的不同数据类型的变量。

```
long testValue = 150;
char message[22] = {"This is a test message"};
float pi = 3.14159;
```

这些语句中的每一个变量都被编译器理解为保留指定数量的字节的内存位置,这些位置用于存储程序执行期间可能改变的值。程序在每次引用变量名时,编译器就知道要去访问内存中指定的位置来读取或者改变字节的值。

汇编语言也允许自定义将存储在内存中的数据项目,而且它提供里更大的控制权来决定在内存中的存储数据的位置和如何存储数据。

和高级语言定义数据的方法类似,汇编语言允许声明指向内存中特定位置的变量。使用这种方法时,在汇编语言中定义变量包括两个部分。

- 指向一个内存位置的标记;
- 内存字节的数据类型和默认值。

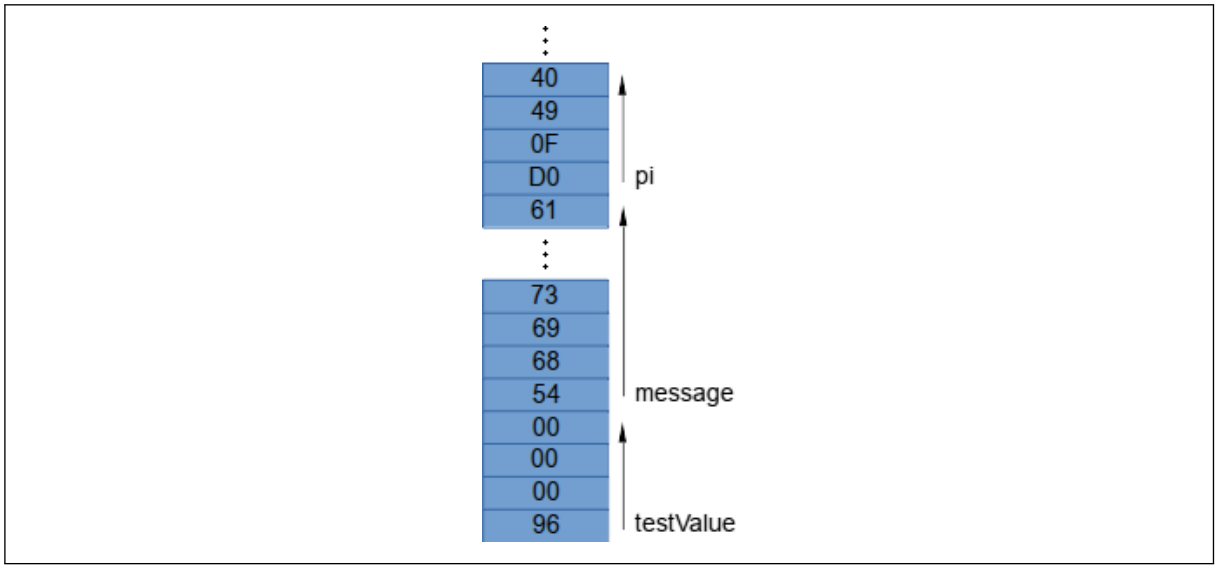
汇编语言允许声明存储在内存位置中的数据的类型以及默认值,通过数据类型来决定为变量保留多少字节。

```
testValue:
```

```
  .long 150
message:
  .ascii "This is a test message"
pi:
  .float 3.14159
```

这里，数据类型是使用 GNU 汇编器中的汇编器命令声明的，这里命令和标记的不同在于命令前面有一个点号(.)，因此.long、.ascii 和.float 命令用于通知汇编器正在声明一个特定的数据类型。

每种数据类型都占用特定数量的字节，从为标记保留的内存位置开始。



这里，声明的第一个数据元素 testValue 按照小尾数顺序作为 4 字节的十六进制值(96 00 00 00)被存放在内存中。下一个数据元素 message 紧跟在数据元素 testValue 的最后一个字节后面，以此类推。

数据元素 message 是字符值，它在内存中存放的顺序是按照字符串中出现的字符的顺序。

在汇编语言程序中，按照定义开始位置的标记来引用内存位置。

```
movl  testValue, %ebx
addl  $10, %ebx
movl  %ebx, testValue
```

在上述汇编语言程序的示例中，第一条指令把 testValue 标记指向的内存位置的 4 字节值(当前它的值被定义为 150)载入 EBX 寄存器。下一条指令把 EBX 寄存器中存储的值加上 10(十进制)，然后把结果存储回 EBX 寄存器。最后，寄存器的值被存储到 testValue 标记引用的内存位置中，此时它的值将是 160。在后面的程序中，可以再次使用 testValue 标记引用这个新的值。

除了通过内存位置外，汇编语言提供的另一种存储和检索数据的方法是使用堆栈(stack)。

堆栈是在计算机中为应用程序保留的内存范围的结尾位置的特殊的内存区域，经常用于在程序中的函数之间传递数据，也可以使用它临时地存储和检索数据元素。

堆栈指针(stack pointer)用于指向堆栈中的下一个内存位置以便放入或者取出数据。可以把堆栈理解为一叠纸，在把数据元素放到堆栈中时，它就成为可以从堆栈中删除的第一个项目(这里假设只能从这叠纸的上面把纸拿走)。

在汇编语言程序中调用函数时，经常把希望传递给函数的任何数据元素放到堆栈的顶端，这样当函数被调用时，它可以从堆栈查找数据元素。

汇编器保留有专门的关键字用于在助记符被转换为指令码时，指示汇编器如何执行专门的函数。

汇编器命令用于简化创建指令码的操作，而命令也是不同汇编器之间最大的区别。操作码助记符和处理器指令码的关系密切，而各个汇编器命令对于不同的汇编器都是独特的。一些现代汇编器把命令提高到了更高的水平，提供支持对高级语言特性(例如 while 循环和 if-then 语句)的支持，而传统

的汇编器仅保持最少的命令,因此需要使用助记符来创建程序逻辑(比如声明数据类型和定义内存区域等)。

- MASM(Microsoft Assembler)与 Visual Studio 集成,也可以从命令行汇编程序。
- MASM32 结合了原始的 MASM 汇编器和 Win32 API,可以在 Windows 环境中创建成熟的 C 和 C++ 程序。
- NASM 和所有的 Intel 指令码集合完全兼容,可以生成 UNIX、MS-DOS 和 Microsoft Windows 格式的可执行文件。
- GNU 汇编器(gas)可以在不同的处理器平台上运行,而且 gas 能够自动检测底层硬件平台并且创建适合该平台的正确的指令码。

  GNU 汇编器支持交叉汇编,GNU C 也使用它把编译后的 C 和 C++ 程序转换为指令码,从而可以在现有的 C 和 C++ 应用程序中引入汇编语言功能。

汇编语言程序中使用的最为重要的命令之一是.section 命令。

.section 命令用于定义内存段,汇编语言程序在其中定义元素,而且所有汇编语言程序都至少具有 3 个必须声明的段落。

- 数据段
- bss 段
- 文本段

其中,数据段用于声明为程序存储数据元素的内存区域。在声明数据元素之后,这一段落不能扩展,并且它在整个程序中保持静态。

bss 段也是静态的内存段,它包含用于以后在程序中声明的数据的缓冲区,这一段落的特殊之处是缓冲区内存区域是由 0 填充的。

文本段是内存中存储指令码的区域。同样地,这一区域也是固定的,其中只包含汇编语言程序中声明的指令码。

尽管汇编语言程序设计经常被归类为单一的程序设计语言,但实际上存在多种不同类型的汇编器。

在汇编出最终的程序时,每种汇编器用来表示指令码、数据和专门命令的格式都可能不同,因此在进行汇编语言程序设计时的第一个步骤是决定要使用的汇编器及其格式。

51.5 Link

大多数汇编器不会自动链接目标文件来生成可执行程序文件,因此源代码被编译为目标文件之后,就需要使用链接器(linker)来把应用程序的目标文件及其依赖的目标文件连接起来,从而创建出可执行程序或库文件。

很多高级语言可以使用单一命令执行编译和链接两个步骤,链接目标代码的过程涉及到解析程序代码中声明的所有定义好的函数和内存地址引用。

为了达到这一目的,任何外部函数(例如 C 语言的 printf 函数)都必须包含在目标代码中(或者提供对外部动态库的引用),这就要求链接器还必须知道常用目标代码库的位置,或者由用户使用编译器命令行参数来手动指定。

每个汇编器包都包含自己的链接器,使用与汇编器适配的链接器有助于确保把函数链接在一起所使用的库文件是相互兼容的,并且保证输出文件的格式对于目标平台是正确的。

在最简单的情况下,要从汇编器生成的目标文件创建可执行文件时,可以使用下面的命令:

```
$ ld -o test test.o
$ ls -al test
$ ./test
```

上述示例程序从目标代码文件 test.o 创建可执行文件 test，并且创建出的可执行文件具有适当的文件权限[4]，以允许从命令行中运行它。

当手动调用链接器时，开发者必须知道完整地解析应用程序使用的所有函数需要的库，并通知链接器函数库的位置以及需要连接到一起的目标代码文件来生成最终文件。

Table 51.4: GNU ld 命令行参数（Intel 平台）

参数	描述
-b	指定目标代码输入文件的格式
-Bstatic	只使用静态库
-Bdynamic	只使用动态库
-Bsymbolic	把引用捆绑到共享库中的全局符号
-c	从指定的命令文件读取命令
`--cref`	创建跨引用表
-d	设置空格给通用符号，即使指定了可重定位输出
-defsym	在输出文件中创建指定的全局符号
`--demangle`	在错误消息中还原符号名称
-e	使用指定的符号作为程序的初始执行点
-E	对于 ELF 格式文件，把所有符号添加到动态符号表
-f	对于 ELF 格式共享对象，设置 DT_AUXILIARY 名称
-F	对于 ELF 格式共享对象，设置 DT_FILTER 名称
-format	指定目标代码输入文件的格式（和-b）
-g	用于提供和其他工具的兼容性，已被忽略
-h	对于 ELF 格式共享对象，设置 DT_SONAME 名称
-i	执行增量链接
-l	把指定的存档文件添加到要链接的文件清单
-L	把指定的路径添加到搜索库的目录清单
-M	显示连接映射，用于诊断目的
-Map	创建指定的文件来包含连接映射
-m	模拟指定的连接器
-N	指定读取/写入文本和数据段
-n	设置文本段为只读
-noinhibit-exec	生成输出文件，即使出现非致命连接错误
-no-keep-memory	为内存使用优化连接
-no-warn-mismatch	允许连接不匹配的目标文件
-O	生成优化了的输出文件
-o	指定输出文件的名称
-oformat	指定输出文件的二进制格式
-R	从指定的文件读取符号名称和地址
-r	生成可重定位的输出（称为部分连接）
-rpath	把指定的目录添加到运行时库搜索路径
-rpath-link	指定搜索运行时共享库的目录

[4]链接器 ld 自动使用 755 权限模式来创建可执行文件，这允许使用系统的任何人运行它，但只有文件的所有者才能修改它。

参数	描述
-S	忽略来自输出文件的调试器符号信息
-s	忽略来自输出文件的所有符号信息
-shared	创建共享库
-sort-common	在输出文件中不按照长度对符号进行排序
-split-by-reloc	按照指定的长度在输出文件中创建额外的段
-split-by-file	为每个目标文件在输出文件中创建额外的段
`--section-start`	在输出文件中指定的地址定位指定的段
-T	指定命令文件(和-c 相同)
-Ttext	使用指定的地址作为文本段的起始点
-Tdata	使用指定的地址作为数据段的起始点
-Tbss	使用指定的地址作为 bss 段的起始点
-t	在处理输入文件时显示它们的名称
-u	强制指定符号在输出文件中作为未定义符号
-warn-common	当一个通用符号和另一个通用符号结合时发出警告
-warn-constructors	如果没有使用任何全局构造器,则发出警告
-warn-once	对于每个未定义符号只发出一次警告
-warn-section-align	如果为了对齐而改动了输出段地址,则发出警告
`--whole-archive`	对于指定的存档文件,在存档中包含所有文件
-X	删除所有本地临时符号
-x	删除所有本地符号

链接器输出的可执行文件只能运行在当前的操作系统上,而且每种操作系统使用的可执行文件的格式是不同的,因此这又涉及到交叉编译的问题。

51.6 Debug

和汇编器类似,调试器也是需要与特定的软硬件开发平台相对应的,还必须了解硬件平台的指令集以及操作系统使用寄存器和内存的方法。

大多数调试器为开发者提供了 4 个基本功能。

- 在一个受控环境下运行程序,并指定任何必须的运行时参数;
- 在程序内的任何位置停止程序;
- 检查数据元素(例如内存位置和寄存器);
- 在程序运行时改变程序中的元素以便帮助消除缺陷。

GNU 调试器 gdb 可以通过步进方式执行程序来调试 C 和 C++ 应用程序来查找程序中的错误,也可以用于调试汇编语言程序。

Table 51.5: gdb 命令行参数

参数	描述
-b	设置远程调试时串行接口地线路速度
-batch	以批处理模式运行
-c	指定要分析地核心转储文件

参数	描述
-cd	指定工作目录
-d	指定搜索源文件的目录
-e	指定要执行的文件
-f	调试时以标准格式输出文件名和行号
-nx	不执行来自.gdbinit 文件的命令
-q	安静模式,不输出 gdb 介绍信息
-s	指定符号的文件名
-se	指定符号和要执行的文件名
-tty	设置标准 I/O 设备
-x	从指定的文件执行 gdb 命令

调试器在其自己控制下的“沙箱”内运行程序,沙箱允许程序访问内存区域、寄存器和 I/O 设备,监视每条指令如何修改寄存器和内存位置,这些操作都处于调试器的控制下。

在 gdb 命令提示下,通过输入调试命令可以控制程序如何访问项目并能够提供信息来显示程序如何以及何时访问项目。

命令	描述
break	在源代码中设置断点以便停止执行
watch	设置监视点,当变量到达特定值时停止执行
info	观察系统元素,例如寄存器、堆栈和内存
x	检查内存位置
print	显示变量值
run	在调试器内开始程序的执行
list	列出指定的函数或行
next	执行程序中的下一条指令
step	执行程序中的下一条指令
cont	从停止的位置继续执行程序
until	运行程序,直到到达执行的源代码行(或者更大的)

下面是 gdb 会话的一个示例。

```
$ gdb hello
GNU gdb (GDB)
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...done.
(gdb) list
```

```
1     #include <stdio.h>
2
3     main()
4     {
5       printf("hello,world.\n");
6     }
(gdb) break main
Breakpoint 1 at 0x400534: file hello.c, line 5.
(gdb) run
Starting program: /home/guoyu/Documents/Git/TCPL/test/test/hello

Breakpoint 1, main () at hello.c:5
5       printf("hello,world.\n");
(gdb) next
hello,world.
6     }
(gdb) quit
```

首先,使用 list 命令显示源代码行号。接着,使用 break 命令在 main 标记处设置断点,然后使用 run 命令启动程序。

在 main 处设置断点后,程序在 main 之后的第一个源代码语句之前立即停止。使用 next 命令可以执行下一行源代码,即执行 printf 语句。

应用程序运行结束后,由于调试过程并未结束,因此程序仍处于调试器环境中,而且可以选择再次运行程序。

在程序执行过程中的任何位置,调试器都能够停止程序并且指出执行停止在源代码的什么位置。为了做到这一点,调试器必须了解源代码,以及从哪些源代码行生成了什么指令码。调试器需要额外的信息被编译到可执行文件中以便识别这些元素,通常在编译或汇编程序时使用特定的命令行参数完成这一任务。

程序在执行过程中停止时,调试器能够显示与程序相关的任何内存区域或者寄存器值。同样,这也是通过在调试器的沙箱内运行程序做到的,这使得调试器在程序执行时能够了解程序的内部情况。通过了解各个源代码语句如何影响内存位置和寄存器的值,可以发现程序中的错误发生在什么位置。

调试器向开发人员提供在程序运行时改变程序中的数据值的途径,从而可以在程序运行时改动程序以查看这些改动如何影响程序的输出,这节省里修改源代码、重新编译源代码和重新运行可执行文件的时间。

51.7 Disassembler

试图优化高级语言时,需要了解代码如何在处理器上运行,GNU 编译器允许在对生成的汇编语言代码进行汇编之前进行查看和修改。

当目标文件已经创建时,可以使用反汇编器(Disassembler)处理完整的可执行程序或者目标代码文件,并且显示将运行在处理器上的指令码或汇编语言代码。

另外,很多编译器也具有显示从源代码生成的指令码的能力,从而可以检查源代码语句与实际指令码之间的转换。这样,在检查了编译器生成的指令码后,可以确定编译器生成的指令码是否进行了充分的优化。如果不是,能够创建自己的指令码函数来替换编译器生成的函数,从而提高应用程序的性能。

51.8 Profiler

程序优化可以从确定程序中消耗执行时间最多的函数入手,通过查找处理密集型的函数[5],可以缩小需要优化的函数的范围。

为了确定每个函数消耗的时间,可以使用分析器(profiler)来跟踪分析应用程序的性能,监视每个函数在程序执行过程中的运行。

GNU 分析器(gprof)用于分析程序的执行和确定应用程序中的“热点”的位置,找到消耗时间最多的函数之后,就可以使用反汇编器查看生成的指令码并分析使用的算法进行优化,这样就能够使用高级处理器指令来手动生成指令码来优化函数。

gprof 的参数可以分为三组。

- 输出格式参数:用于修改 gprof 生成的输出。

参数	描述
-A	显示所有函数的源代码,或者只显示特定函数的
-b	不显示解释分析字段的详细输出
-C	显示所有函数的总计数,或者只显示指定函数的
-i	显示简档数据文件的摘要信息
-I	指定查找源文件的搜索目录清单
-J	不显示注解的源代码
-L	显示源文件名的完整路径名称
-p	显示所有函数的一般简档,或者只显示指定函数的
-P	不输出所有函数的一般简档,或者不显示指定函数的
-q	显示调用图表分析
-Q	不显示调用图表分析
-y	在单独的输出文件中生成注解的源代码
-Z	不显示函数的总计数和被调用的函数
`--function-reordering`	按照分析显示建议的函数的重排序
`--file-ordering`	按照分析显示建议的目标文件重排序
-T	按照传统的 BSD 样式显示输出
-w	设置输出行的宽度
-x	在函数内显示被注解的源代码中的每一行
`--demangle`	在显示输出时 C++ 符号被还原

- 分析参数:用于修改 gprof 分析包含在分析文件中的数据的方式。

参数	描述
-a	不分析静态声明(私有)的函数的信息
-c	分析程序中永远不会被调用的子函数的信息
-D	忽略已知不是函数的符号(只在 Solaris 和 HP 操作系统上)
-k	不分析匹配开头和结尾的 symspec 的函数
-l	按行分析程序,而不是按函数
-m	只分析被调用超过指定次数的函数

[5] I/O 密集型的函数也会增加处理时间。

参数	描述
-n	只分析指定的函数的时间
-N	不分析指定的函数的时间
-z	分析所有函数,即使是从不被调用的那些函数

- 杂项参数:修改 gprof 的行为。

参数	描述
-d	使 gprof 处于调试模式中,指定数字化的调试级别
-O	指定简档数据文件的格式
-s	使 gprof 只在简档数据文件中汇总数据
-v	输出 gprof 的版本

为了对应用程序使用 gprof,必须确保使用-pg 参数编译希望监视的函数。使用-pg 参数编译源代码时,会为程序中的每个函数插入对 mcount 子例程的调用。当应用程序运行时,mcount 子例程会创建一个调用图表简档文件——gmon.out,它包含应用程序中每个函数的计时信息。

另外,运行应用程序时要小心,因为每次运行都会覆盖 gmon.out 文件。如果希望进行多次采样,就必须在 gprof 的命令行中包含输出文件的名称,并且在每次采样时使用不同的文件名。

程序测试结束后,使用 gprof 程序查看调用图表简档文件,可以分析出每个函数消耗的时间。

gprof 的输出包含 3 个报告,分别是:

- 一般简档报告,列出了总执行时间和所有函数的调用次数。
- 按照每个函数及其子函数消耗的时间进行排序的函数清单。
- 循环清单,用于显示循环成员和它们的调用次数。

默认设置下,gprof 的输出直接发送到控制台的标准输出。如果希望保存输出,就必须把它重定向到文件。

下面以一个 C 语言示例程序来说明 gprof 的使用。

```
#include <stdio.h>

void function1()
{
  int i, j;
  for(i = 0; i < 100000; i ++)
    j += i;
}

void function2()
{
  int i, j;
  function1();
  for(i = 0; i < 200000; i ++)
    j = i;
}

int main()
{
  int i, j;
  for(i = 0;i < 100; i ++)
    function1();

  for(i = 0; i < 50000; i ++)
```

```
    function2();

  return 0;
}
```

这个示例程序的主程序中有两个循环:一个调用 function1()100 次,另一个调用 function2()5000 次。每个函数只执行简单的循环,但是每次调用 function2() 时,它也调用 function1()。

在编译该示例程序时,需要使用-pg 参数。

```
$ gcc -o demo demo.c -pg
$./demo
```

在示例程序执行结束后,在相同的目录下会创建 gmon.out 调用图表简档文件,然后可以对示例程序运行 gprof 程序,并把输出保存到新文件中。

```
$ gprof demo > gprof.txt
```

这里,命令行中没有引用 gmon.out 文件,gprof 会自动使用位于相同目录下的 gmon.out 文件,然后把 gprof 的输出重定向到 gprof.txt 文件。

在 gprof 报告中,首先显示总的处理器时间和 main 调用每个函数的时间。

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 61.47     25.61    25.61    50000   512.30   842.78  function2
 39.73     42.17    16.56    50100   330.48   330.48  function1

 %           the percentage of the total running time of the
time         program used by this function.

cumulative   a running sum of the number of seconds accounted
 seconds     for by this function and those listed above it.

 self        the number of seconds accounted for by this
seconds      function alone. This is the major sort for this listing.

calls        the number of times this function was invoked, if
             this function is profiled, else blank.

 self        the average number of milliseconds spent in this
ms/call      function per call, if this function is profiled, else blank.

 total       the average number of milliseconds spent in this
ms/call      function and its descendents per call, if this
             function is profiled, else blank.

name         the name of the function. This is the minor sort
             for this listing. The index shows the location of
             the function in the gprof listing. If the index is
             in parenthesis it shows where it would appear in
             the gprof listing if it were to be printed.
```

在接下来的图表报告中显示出每个函数的细分时间以及函数是如何被调用的。

```
                Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.02% of 42.17 seconds
```

```
index % time self children called name
                                     <spontaneous>
[1]  100.0  0.00 42.17                 main [1]
           25.61 16.52 50000/50000   function2 [2]
            0.03  0.00   100/50100   function1 [3]
-----------------------------------------------
           25.61 16.52 50000/50000       main [1]
[2]   99.9 25.61 16.52 50000        function2 [2]
           16.52  0.00 50000/50100   function1 [3]
-----------------------------------------------
            0.03  0.00   100/50100       main [1]
           16.52  0.00 50000/50100   function2 [2]
[3]   39.3 16.56  0.00 50100        function1 [3]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the

```
            function `/' the total number of times the function
            was called. Recursive calls to the function are not
            included in the number after the `/'.

   name     This is the name of the parent. The parent's index
            number is printed after it. If the parent is a
            member of a cycle, the cycle number is printed between
            the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

   self     This is the amount of time that was propagated directly
            from the child into the function.

   children This is the amount of time that was propagated from the
             child's children to the function.

   called   This is the number of times the function called
            this child `/' the total number of times the child
            was called. Recursive calls by the child are not
            listed in the number after the `/'.

   name     This is the name of the child. The child's index
            number is printed after it. If the child is a
            member of a cycle, the cycle number is printed
            between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

  [3] function1        [2] function2
```

调用图表的每个部分显示被分析的函数(即带有索引号码的行中的函数)、调用它的函数及其子函数,从而可以跟踪整个程序中的时间流。

第 52 章

Pig Latin

问题：写一个程序用来从终端读入一行文本，并把这行文本中的英文转换成 Pig Latin。

Pig Latin 是按照如下简单规则转换每个英文单词的一种自发明语言：

1. 如果单词以辅音开头，那么把起始辅音字符串（即直到第一个元音字母的所有字母）从单词开始移到单词尾部，并加上后缀 ay。
2. 如果单词以元音开头，则加后缀 way。

例如，对于以辅音开头的单词 scram，需要把它分为两部分。第一部分由第一个元音之前的所有字符组成，另一部分包括那个元音以及之后的字母。

```
scr  am
```

然后交换这两部分的位置，并且在末尾加上 ay。

```
am  scr  ay
```

这样就创建了 Pig Latin 单词 amscray。对于元音开头的单词，如 apple，则在末尾加上 way，从而得到 appleway。

52.1 Top-down design

在开始设计某一个问题的解决方案时，可以先不立即决定是否需要把程序分割成的模块。通常在比较了不同的解决方法以后，才可以决定一个程序是否需要一个特定模块分解，因此一般最好的方法是先应用自顶向下的设计方法。

通过问题陈述现在可以了解问题需求是：把整行英文翻译成 Pig Latin，那么程序应该能够产生如下的运行示例：

```
Enter a line: This is pig latin.
isthay isway igpay atinlay
```

对 Pig Latin 问题采用自顶向下的设计方法，从主程序的层次着手，然后依次写下一系列函数，每个函数解决整个问题中的一个部分。

在初始阶段，可以用一系列有待具体实现的高层步骤来定义 main 函数。虽然可以将这些步骤直接编码为函数调用，但是通常先描述出其功能更为容易。

例如，当在纸上设计程序时，可以写出如下的 Pig Latin 程序的初稿：

```
main()
{
  Read in a line of text from the user.
  Translate the line of text into Pig Latin.
}
```

这里，函数头和花括号是 C 语言标准语法的一部分，而语句本身则写成英文语句，用来描述程序要做什么事情。由英文和 C 语言混合构成的程序称为伪代码（pseudo code）。

52.2 Using Pseudocode

伪代码对编译器来说没有意义，但是对于程序员非常有用，因为它可以记录逐步求精的过程。在用一系列英文步骤写出整个程序之后，可以再回到伪代码语句，用实际的 C 语言代码来实现它们。

例如，在 Pig Latin 的设计过程中，可以很容易地把第一句伪代码翻成 C 语句，因为它和读取一个字符串的习语非常吻合。在填写完从用户处读入一行文本的操作之后，程序的伪代码版本如下所示：

```
main()
{
  string line;
  printf( "Enter a line:" );
  line = GetLine();
  Translate the line of text into Pig Latin.
}
```

结果仍然是伪代码，但是已经更进一步了。剩下的过程更难编码，最好的方法是实施逐步求精的策略，用一个效果和这行英文语句同样的新的函数来替换这行伪代码。在这个例子中，需要一个“将这一行转换为 Pig Latin”的函数，可以将函数命名为 TranslateLine。

使用这个名字，现在就可以完成 main 函数的实现如下：

```
main()
{
  string line;
  printf( "Enter a line:" );
  line = GetLine();
  TranslateLine(line);
}
```

在这个层次，程序的流程显得非常简单，可以描述为：

显示一个提示，读入一行数据，然后调用 TranslateLine 来完成工作。虽然还未编写 TranslateLine 函数，但是可以从调用程序的角度来说出它的行为。

事实上，现在我们已经拥有了足够的信息来为 TranslateLine 定义描述和原型：

```
/*
* Function:TranslateLine
* Usage:TranslateLine(line);
* ------------------------------
* This function takes a line of text and translate the words in the line to Pig Latin,
* displaying the translation as it goes.
*/

void TranslateLine(string line);
```

52.3 Implementing TranslateLine

在进一步分解 TranslateLine 时，和程序设计领域中通常的情况一样，有很多策略可以完成这个目标，其中一些策略的工作情况可能更好一些。但在大多数情况下，没有一个特定的分解策略是绝对正确的，因此通常需要考虑几种分解问题的方法，然后看看哪一种策略最好。

在实现 TranslateLine 的时候，需要先解决如何把一个字符串分为单词，把每个单词翻译成 Pig Latin，然后在屏幕上显示每个 Pig Latin 单词的问题。根据这个问题的陈述，可以写出如下的概念分解：

```
void TranslateLine(string line)
{
  Divide the line into words.
  Translate each word into Pig Latin.
```

```
    Display each translated word.
    Display a newline character to complete the output line.
}
```

继续考虑,从理论上说,这个分解是合理的,但会导致某些实际的问题。在第一步中,将语句行分解为单词,便引起了如何存储结果的问题。

实现这个概念的函数返回的不是一个单词,而是一个单词列表。

如果这里不使用处理单词列表的工具,就需要仔细考虑这个问题,进而发现不需要立即记录所有的单词。一旦找到一个单词,就可以马上翻译并且正确显示。一旦显示出这个单词,就可以丢开它继续处理下一个单词。根据这个观察结果,可以提出第二种策略:

```
void TranslateLine(string line)
{
  while(there are any words left on the line){
    Get the next word.
    Translate that word into Pig Latin.
    Display the translated word.
  }
  Display a newline character to complete the output line.
}
```

虽然这个策略的伪代码里还有很多细节问题没有考虑,但是总体思想看起来是有意义的,并且可以避免记录整个单词列表的问题。

52.4 Space and Punctuation

现在 TranslateLine 的伪代码版本所用的策略并不完善。例如,在程序运行时,假设用户输入了以下数据行

```
Enter a line: this is pig latin.
```

如果把输入看作四个单词 this、is、pig 和 latin,如果所有的英文表示的步骤都如预期的那样正常运行,那么程序输出将会是:

```
Enter a line: this is pig latin.
isthayiswayigpayatinlay
```

这个输出并不是我们真正想要的。因为没有空格和标点符号,所以所有的单词都挤在一起了。

考察设计,原因在于这个伪代码版本没有把空格和标点符号考虑进去。而原来对问题的英文陈述也没有考虑这个问题。这个问题没有被完整地定义过。

程序设计中的一个现实问题是问题的英语描述通常是不完整的。作为一个程序员,常常会被一些所忽略的或者认为太明显而没有提及的细节所难倒。在某些情况下,被省略的部分是非常重要的,需要跟给你布置任务的人进一步讨论。然而,在很多时候,程序员必须自己选择一种合理的策略。但是,判断什么是合理的策略也有一些技巧。

在这个例子中,发现问题后需要决定的是在输出的每个单词之间打印空格,而忽略其他标点符号。这个策略很简单,而且在本例中很合理。但另一方面它又可能不是最好的。标点符号可以提高输出的可读性。因为标点符号和空格都能表达出某种意义,所以最好让它们在输出中的位置和在输入中的位置相同。因此,需要的输出为:

```
Enter a line: this is pig latin.
isthay isway igpay atinlay
```

因此需要想办法重新设计该程序,使得标点符号正确地出现在输出中。例如,一种方法是改变主循环,使得它按字符而不是按单词为单位来处理。如果使用这种策略,伪代码便可以按如下结构实现:

```
void TranslateLine(string line)
{
  for(i = 0; i < StringLength(line); i ++){
    if(the ith character in the line is some kind of separator){
      Display that character
    }else if(the ith character is the end of a word){
      Extract the word as a substring
      Translate the word to Pig Latin
      Display the translated word
    }
  }
  Display a newline character to complete the output line.
}
```

现在这个策略可以工作了，虽然这个策略还是有某些缺点。一个缺点就是程序结构变得更加复杂了。原来的伪代码设计更简短些，部分原因是因为它允许以更大的单元去处理字符串。

不过，在分解中暴露出一个更严重的问题，上一个伪代码版本包含了一个英文语句：

```
Get the next word
```

在这里被省略了。而从程序员的角度考虑，会认为“get the next word”这个操作是一个非常有用的工具，它的应用范围远远超出简单 Pig Latin 程序。

许多问题都需要把文本分割成单词，因而有必要开发出一个通用的函数来执行这个操作，从而用来解决此类相关问题。

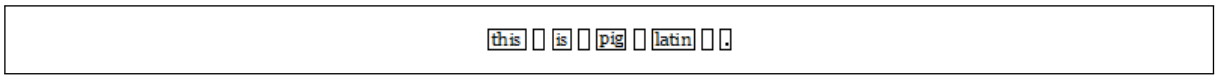
另一方面，简单地做到取下一个单词还没有解决问题。为了将对单词的处理应用到当前的策略中，返回下一个词的函数还必须能够返回空格和标点符号，以使输出行中能包括这些符号。

52.5 Refining Words

当前需要做的是对一个单词的概念进行精化。对这样的输入行：

```
this is pig latin
```

可以将它看作四个单词：this、is、pig、latin，或者也可以将这个输入行看作由以下 8 个单独的片段组成：



和单词一样，上述方法把空格和标点解释为单独的实体。在计算机科学中，作为一个相关单元的字符序列被称为一个记号(token)。在上面这个图中，每个方框都表示一个记号。

把空格和标点符号作为独立的记号使得程序员可以修改 TranslateLine 这个方案，使这些符号也显示出来。修改过的伪代码策略如下：

```
void TranslateLine(string line)
{
  while(there are any tokens left on the line){
    Get the next token
    if(the token is a regular English word){
      Replace the token by its pig latin translation.
    }
    Display the token.
  }
  Display a newline character to complete the output line.
}
```

这个策略中获取一个记号并测试这些记号是否留在该行中的操作很可能在不同的应用程序中都是有用的。

把一行分成一些独立的记号的想法在计算机科学中非常常见。例如，当 C 编译器将一段程序翻译成机器代码的时候，此过程的第一步就是将输入文件分割成 C 语言所用的记号，即变量名、数字、运算符等等。将输入分解成记号的过程称为词法分析（lexical analysis），或者更通俗地说，把它称为记号扫描（token scanning）。

52.6 Designing Token Scanner

要使用新的策略来完成 TranslateLine 的实现，必须首先设计能够分割输入行的记号扫描器。而且，要记住它很可能是个通用的工具，除了将字符串转换成 Pig Latin 外，记号扫描器还能用于许多其他问题。因此，将记号扫描器设计成一个独立的模块就十分有意义。起初，设计过程需要考虑如何组织扫描器模块，特别是它该包含什么函数。

在目前最新的 TranslateLine 伪代码版本中，扫描器模块有两个不同的作用。第一，扫描器必须提供一个函数返回当前行中的下一个记号。第二，当扫描完最后一个记号时要让客户知道。

在设计这个接口时，要设定好这些函数的名字。负责返回下个记号的函数可以命名为 GetNextToken。要报告所有记号都已经被扫描，一种选择是定义一个叫 AtEndOfLine 的谓词函数，它在读到最后一个记号时返回 TRUE。

接下来要确定的一个问题就是这些函数的参数。初一看，一方面调用程序必须将输入行传递给 GetNextToken，因为记号是来自于那一行的。

另一方面，调用 GetNextToken(line) 的想法在概念上有矛盾之处。如果 GetNextToken 像通常的函数那样运作，那么 GetNextToken(line) 每次都返回同样的结果，因为 line 的值没有改变。

为了解释这个问题，我们假设把字符串 “Hello there” 分成三个记号：单词 “Hello”，跟在它后面的空格，以及单词 “there”。

另外，假设已经用赋值语句

```
line = "Hello there";
```

把字符串 “Hello there” 存储在变量 line 中。如果 GetNextToken 将 line 作为参数，很容易就能想到调用 GetNextToken(line) 将得到第一个单词，接下来的问题是如何得到下一个记号。

如果再次调用 GetNextToken(line)，变量 line 仍然包含着整个字符串 “Hello there”。既然这个函数调用跟第一次出现的形式完全相同，那么它将返回同一个值。

为了改正这个问题，必须将扫描器模块设计为能够追踪划分行的进度。在 GetNextToken 从行返回一个记号后，必须记住已经扫描过的那些记号，以便在下次调用时返回一个不同的结果。一个模块中的多次函数调用之间的信息称为内部状态（internal state）。

当模块维护内部状态时，模块的接口通常导出一个用来初始化此状态信息的函数。例如，在扫描器模块中，提供一个函数 InitScanner(line) 是有用的，它使得客户能够告诉扫描器在字符串 line 的开头开始返回记号。为了得到这些记号，客户只要调用 GetNextToken 即可，不需要参数。哪个记号该被返回的信息是扫描器模块要维护的内部状态的一部分。

第一次调用 GetNextToken 返回第一个记号，下次调用返回第二个记号，如此继续直到所有的记号都被读取为止。现在，同样不带参数的 AtEndOfLine 函数会返回 TRUE。

现在可以用函数 InitScanner、GetNextToken 和 AtEndOfLine 来更新 TranslateLine 的实现，如下所示：

```
void TranslateLine(string line)
{
  string token;

  InitScanner(line);
  while(!AtEndOfLine()){
```

```
    token = GetNextToken();
    if(the token is a legal word){
      Replace the token by its Pig Latin translation.
    }
    Display the token.
  }
  Display a newline character to complete the output line.
}
```

这个函数仍然有一些未完成的片段,但是循环结构本身现在是完整的。这个实现首先告诉扫描器通过调用 InitScanner 从变量 line 中提取记号。然后进入循环,在循环中调用 GetNextToken 依次得到每个新的记号,直到所有的记号都被读取为止。

在更新扫描器接口的设计和实现它之前,要先在设计层次上整理目前 TranslateLine 的伪代码实现。如果要继续进行逐步求精的策略,可以用函数调用替换剩下的伪代码语句来完成这个实现。在这个例子中,函数要么调用 printf,要么调用在下一个精化层次中有待实现的函数。

在这两种情况下,函数调用本身是一个直接的英文翻译,于是得到 TranslateLine 的实现如下:

```
void TranslateLine(string line)
{
  string token;
  InitScanner(line);
  while(!AtEndOfLine()){
     token = GetNextToken();
     if(IsLegalWord(token)) token = TranslateWord(token);
     printf( "%s" , token);
  }
  printf( "\n" );
}
```

这样,在目前这个分解层次上实现 TranslateLine 后,这个解决方案中还有两个函数仍然没有实现:

```
IsLegalWord和TranslateWord
```

谓词函数 IsLegalWord 确定 GetNextToken 返回的记号是一个需要转换成 Pig Latin 的单词还是一个简单的标点符号。Pig Latin 规则仅当一个单词全部由字母组成时才有意义。因此如果 token 中的每个字符都是字母,让 IsLegalWord(token) 返回 TRUE 是合理的,因此用字符串实现这个函数,如下所示:

```
bool IsLegalWord(string token)
{
  int i;
  for(i = 0; i < StringLength(token); i ++){
    if(!isalpha(IthChar(token, i))) return (FALSE);
  }
  return (TRUE);
}
```

在确认 token 是单词后,下面就可以调用 TranslateWord 来翻译这个单词了。

在伪代码实现中,TranslateWord 的结构对应了 Pig Latin 的规则。

```
string TranslateWord(string word)
{
  Find the position of the first vowel.
  if(the vowel appears at the beginning of the word){
    Return the word concatenated with "way"
  }else{
    Extract the initial substring up to the vowel and call it the "head"
    Extract the substring from that position onward and call it the "tail"
    Return the tail, concatenated with the head, concatenated with "ay"
  }
```

```
}
```

伪代码中的第一条英文语句是最难实现的，而且是唯一对于定义一个新函数有意义的语句。函数 FindFirstVowel(word) 返回 word 的第一个元音的下标位置。

这样，在 TranslateWord 实现中的第一条语句可以写成如下形式，其中 vp 是个用来记录元音位置的整型变量：

```
vp = FindFirstVowel(word);
```

如果在设计中非常仔细，就会注意到函数 FindFirstVowel(word) 还没有完整的定义。非形式化的描述没有包含所有可能的情况，因此在 FindFirstVowel(word) 中还要考虑如下的情况：

如果 word 中没有元音，如何处理？

在这种情况下，FindFirstVowel 仍然要返回必要的结果。库函数 FindChar 和 FindString 使用-1 作为一个特殊的标志，用来指出函数查询的值在字符串中不存在。这里也可以采用相同的策略。

当 FindFirstVowel 返回-1 时，TranslateWord 必须有合理的响应，因为转换一个不包含元音的单词的结果还没有明确定义过，因此需要确定 TranslateWord 的对应的处理步骤。这时最简单的办法是让 TranslateWord 返回原来的单词。

结合这个设计决策意味着 TranslateWord 的代码将以如下行开始：

```
vp = FindFirstVowel(word);
if(vp == -1){
  return (word);
}else . . .
```

TranslateWord 接下来的步骤只是从 strlib.h 接口中调用合适的函数来完成该过程。

```
string TranslateWord(string word)
{
  int vp;
  string head, tail;
  vp = FindFirstVowel(word);
  if(vp == -1){
    return (word);
  }else if(vp == 0){
    return (Concat(way, "way"));
  }else{
    head = SubString(word, 0, vp -1);
    tail = SubString(word, vp, StringLength(word) -1);
    return (Concat(tail, Concat(head, "ay")));
  }
}
```

下面完成 FindFirstVowel 函数的设计，调用函数 IsVowel 来判断一个字符是否是元音，FindFirstVowel 的实现如下：

```
int FindFirstVowel(string word)
{
  int i;
  for(i = 0; i < StringLength(word); i ++){
    if(IsVowel(IthChar(word, i))) return (i);
  }
  return (-1);
}
```

在实现了 FindFirstVowel 后就完成了主模块的最后一部分，然后可以重点关注扫描器模块。

52.7 Specifying Interface

现在已经有了一个非正式的扫描器模块的接口，而且已经知道该模块要导出三个函数：

- InitScanner
- GetNextToken
- AtEndOfLine

现在也知道了每个函数所需要的参数，下一步就是把非正式的设计翻译成接口。

既然已经知道扫描器模块中的函数的作用，那么 scanner.h 接口的工作主要包括给每个函数编写注释，因此最终的结果如下所示：

```
/*
 * File: scanner.h
 * ---------------
 * This file is the interface to a package that divides
 * a line into individual "tokens". A token is defined
 * to be either
 *
 * 1. a string of consecutive letters and digits representing
 *   a word, or
 *
 * 2. a one-character string representing a separator
 *   character, such as a space or a punctuation mark.
 *
 * To use this package, you must first call
 *
 *     InitScanner(line);
 *
 * where line is the string (typically a line returned by
 * GetLine) that is to be divided into tokens. To retrieve
 * each token in turn, you call
 *
 *     token = GetNextToken();
 *
 * When the last token has been read, the predicate function
 * AtEndOfLine returns TRUE, so that the loop structure
 *
 *     while (!AtEndOfLine()) {
 *        token = GetNextToken();
 *        . . . process the token . . .
 *     }
 *
 * serves as an idiom for processing each token on the line.
 *
 * Further details for each function are given in the
 * individual descriptions below.
 */

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"

/*
 * Function: InitScanner
 * Usage: InitScanner(line);
 * -------------------------
```

```
 * This function initializes the scanner and sets it up so that
 * it reads tokens from line. After InitScanner has been called,
 * the first call to GetNextToken will return the first token
 * on the line, the next call will return the second token,
 * and so on.
 */

void InitScanner(string line);

/*
 * Function: GetNextToken
 * Usage: word = GetNextToken();
 * -----------------------------
 * This function returns the next token on the line.
 */

string GetNextToken(void);

/*
 * Function: AtEndOfLine
 * Usage: if (AtEndOfLine()) . . .
 * -------------------------------
 * This function returns TRUE when the scanner has reached
 * the end of the line.
 */

bool AtEndOfLine(void);

#endif
```

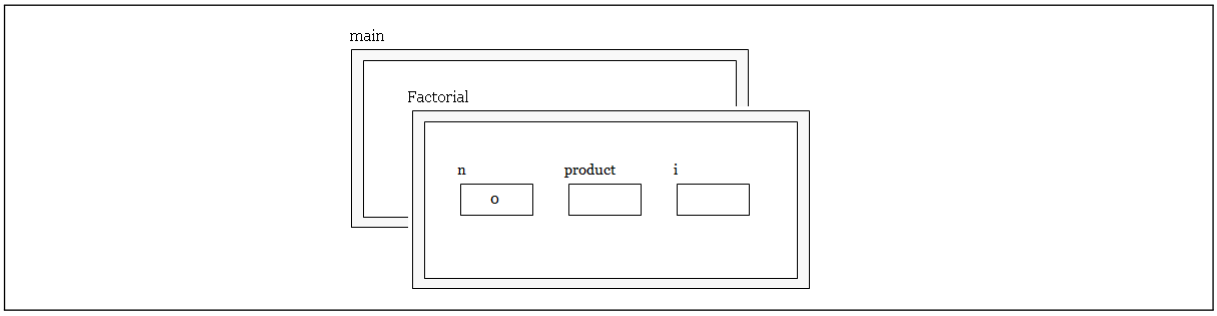
第 53 章

Internal State

为了记录内部状态,导致扫描器模块中的函数与之前谈到的那些函数的实现有很大的区别。

一般地,当调用一个典型的函数时会声明一些局部变量,但是当函数返回时,这些变量值就被丢弃了。

根据函数调用的机制,如果一个函数被调用,它声明的变量就在一个叫栈帧的独立内存区域中被创建,可以用索引卡片图的形式来说明栈帧。



- 调用一个函数相当于创建一个新的索引卡片,并且将其放在代表其他活动函数的卡片堆的上面。
- 从函数返回相当于拿掉它的索引卡片,并继续在调用程序中执行。

在函数内部声明的变量叫局部变量,局部变量仅存在于一个栈帧中。当函数返回时,其栈帧中的变量就完全消失了。代表栈帧的索引卡片会被扔掉,这些变量的值也就丢失了。

在这里,扫描器的实现是不允许其中的函数每次返回时都丢弃所有信息的。例如,当调用下面的函数后,传递到 InitScanner 的字符串必须保存原来的字符串 “Hello there” 以便接下来调用 GetNextToken 时可以按序返回行的每个片段。

```
InitScanner("Hello there");
```

同样,扫描器模块也必须记录行中的当前位置,这样当 GetNextToken 返回 “Hello” 后,该实现必须记录下一个记号是单词 “Hello” 和 “there” 之间的空格这一事实。

这个信息(行本身和行当前位置)就代表了扫描器模块的内部状态,与函数中的局部变量不一样,内部状态必须在调用之间保留信息。

下面列出的是在 C 语言程序中需要维护内部状态信息的库软件包的实例。

- 在图形库中,每一条线都是从上一条线的结束处开始。图形库中的当前点就是其内部状态的一部分,每次调用图形库时要保持这些信息。
- 在 stdlib.h 中定义的随机数函数 rand 必须记住先前的随机数以便产生下一个随机数。

简单来说,维护内部状态就是实现这些库的模块中,在函数调用之间需要保存一些变量的值。

如果模块需要在函数调用之间维护一些内部状态,就不能使用局部变量来实现,此时必须使用全局变量(Global Variable)。

53.1 Global Variable

全局变量和局部变量相对，局部变量是在一个函数中声明的，参数名和在组成函数体的块首声明的变量名仅在函数内部可以引用。

然而，全局变量声明在函数定义之外出现，以这种方式声明的变量称为全局变量（global variable）。全局变量的声明看起来跟局部变量一样，只是它们是出现在文件开始处。例如，在代码段中，变量 g 是全局变量，而变量 i 是局部变量。局部变量 i 仅在函数 MyProcedure 内有效，而全局变量 g 可以在模块中随后声明的任何函数中使用。

```
int g;
void MyProcedure()
{
  int i;
  . . .
}
```

可以使用变量的程序部分叫变量的作用域。这样，局部变量的作用域是定义它的函数中。全局变量的作用域则是源文件中在它所出现后的其余部分。

和局部变量不同，全局变量以某一种方式保持在内存中，在这种方式下它的值不受函数调用的影响。

仍以索引卡片来比喻，全局变量被保存在一个始终有效的独立的卡片上，好像被粘在桌面上，永远都不会被包含局部变量的卡片覆盖一样。模块中每个函数都可以看到全局变量卡片的那些变量。而且，当函数返回时，这些变量值不会丢失。全局变量保持相同的值直到给它赋予新值为止。

全局变量的想法非常有吸引力，而且全局变量可以被程序的所有部分看到，从而不需要将它们作为参数来传给某个函数。

同时，程序员很快会发现使用全局变量会使得代码很难读懂，而且在设计程序的过程中，往往新手觉得优越的特性往往是有经验的程序员非常认真对待的问题。

新手喜欢全局变量可以被源文件中任何函数操作的特点，对有经验的程序员来说，这却是一个危险的信号。

例如，在查找一个由于变量被错误赋值而导致的程序错误。如果是个全局变量，由于模块中的每个函数都能操作那个变量，那么问题可能存在于源文件的任何位置。而局部变量的错误位置则更容易确定。如果局部变量值错误，程序员只需要查找应用该变量的函数即可。

为了避免这样的问题，在结构良好的程序中，一般很少使用全局变量。

全局变量的主要优点在于使得像扫描器模块这样的模块可以维护内部状态。既然全局变量在函数调用之间可以维护它们的值，那么对于这样的用途非常理想。于是，在扫描器模块中，可以用全局变量来追踪传给 InitScanner 的行和此行的当前位置。

53.1.1 Initializing Global Variable

使用一个初始化函数来给代表模块内部状态的全局变量赋值的方式称为动态初始化（dynamic initialization）。在 scanner.h 接口中，通过调用 InitScanner 函数来初始化全局变量。

动态初始化的主要特点是在程序运行时候执行。客户程序调用模块的初始化函数，该函数使用赋值语句来给全局状态变量赋初值。

在 C 语言中，也可以在程序运行前给全局变量赋初值。编译器生成执行该程序所需指令的目标文件，这些指令是用这台计算机的内部语言表示的。除了这些指令外，目标文件也可以包含定义全局变量初始内容的数据值。由于这种初始化在执行程序之前发生，因而称为静态初始化（static initialization）。

为了定义一个全局变量的静态初始化，要在声明中的变量名之后加上一个等号，然后跟上初值。

初值必须是一个常量，例如在下面的语句中不仅声明 startingValue 为此模块的私有全局变量，而且保证当程序开始运行时变量的内容是 1。

```
static int startingValue = 1;
```

在某些情况下，同样的初始化的语法也能用于局部变量。如果初始化一个局部变量，效果与执行一个赋值完全相同，只是写起来容易一些。为了简化初始化的讨论，本书在局部变量中不用初始化。

这里，大多数维护内部状态的接口使用动态初始化，并包括一个函数来显式执行它。比如 scanner.h 接口中的 InitScanner 函数。然而对于以下两种情况，静态初始化是更好的选择。

1. 变量值在程序整个生命周期内都是常量。
   这种情况很少出现在简单变量中，但是当使用更复杂的数据结构的时候，会变得很重要。
2. 变量的初值只有少数几个客户想要改变。
   在这种情况下，最好的办法是使用静态初始化设置标谁选项，然后提供一个让客户在必要时候改变其值的函数。

下面用例子来说明第二种情况。假设一个扫描器模块的客户要求改变扫描器接口以便所有的包含字母的记号都以大写形式返回。这样，在调用下面的语句之后，用户希望记号为“Hello”，“”和“THERE”，这个行为可能不会对所有的客户都有用，而只对其中某一些客户才有用。

```
InitScanner("Hello there");
```

为了满足这一部分客户的需要，同时又不会让其他客户不满意，需要做的就是在从 GetNextToken 返回之前调用函数 ConvertToUpperCase。另一方面，仅当用户有此要求时才这样做。

要追踪客户是否想要大写记号，可以声明如下所示的全局布尔变量：

```
static bool uppercaseFlag;
```

如果 uppercaseFlag 为真，扫描器全部返回大写记号，如果为假，则照原样返回。

至于如何初始化 uppercaseFlag？如何设计接口来让客户改变这个标志的值？这些问题引出了一些接口设计的重要问题。

一种途径是使用动态初始化。这样，客户通过传递额外的布尔参数给设置 uppercaseFlag 选项的 InitScanner 来选择行为方式。

调用下面的语句来指出客户想要返回大写记号。

```
InitScanner("Hello there", TRUE);
```

相反，调用

```
InitScanner("Hello there", FALSE);
```

来将布尔参数设为 FALSE 来指定客户想要维持扫描器的常规行为，即返回大小写混合的记号，就和输入行中出现的这些字符一样。

然而，这种方法有两个严重缺点。首先，程序阅读者很难知道在调用 InitScanner 时 TRUE 和 FALSE 参数的含义。为了理解这些参数的用途，任何客户都不得不阅读接口注释。

第二，新的设计改变了已有的接口而破坏了稳定性。如果扫描接口已经有客户，那么这些客户不得不修改程序。

避免以上两种问题的更好的办法是扩展扫描器接口而非改变它。所有老的函数都像以前那样工作。为了提供返回大写记号的选项，可以增加一个新的函数 ReturnUppercaseTokens，它带有一个布尔值，用它来设置 uppercaseFlag。因此，调用下面的语句可以满足客户选择返回大写记号的新的行为模式。

```
ReturnUppercaseTokens(TRUE);
```

要注意的是，如果 ReturnUppercaseTokens 从来未被调用，那么模块必须按照以前那样继续工作。因此，即便没有明确的调用来设置它的值，变量 uppercaseFlag 必须用 FALSE 作为其初值。为了保证它有恰当的值，需要使用静态初始化：

```
static bool uppercaseFlag = FALSE;
```

毕竟，使用老的 scanner.h 接口的现有程序没有一个会调用 ReturnUppercaseTokens。甚至写这些客户程序时，这个函数还不存在。除非客户专门采取行动去改变否则一直在程序里使用的值称为默认值(default value)。

在一个典型的模块中，指定客户可以设置的选项的全局变量通常被静态初始化为其默认值。客户可以通过调用接口提供的函数来改变这些值。

53.2 Private Variable

在编写程序时，要避免使用全局变量，因为这样会使程序难以理解、难以调试。只有当模块必须维护函数调用之间的内部状态时，才必须使用全局变量。

全局变量在一个模块的任何地方都可见只是使用它们的问题之一。除非明确地声明，否则 C 编译器假定其他模块也可以看到全局变量。这样，当声明了一个全局变量后，可能改变其值的函数不只是局限在一个模块中。该变量可能被整个程序的任何模块引用。

在一个结构良好的程序中，独立的模块之间通过在模块间传递参数的函数调用来交换数据。在大多情况中，尤其是刚开始养成设计程序习惯的时候，最好确保每个全局变量不会被一个以上的模块引用。为了避免两个模块引用同一个全局变量的可能性，应该在声明前用关键字 static 来彻底避免这种危险。

```
static int cpos;
```

这个声明定义 cpos 为一个全局整型变量，在所定义的模块里的任何地方都可见。然而，cpos 对于别的模块是无效的，因此它是当前模块私有的。

在 C 语言中，单词 static 用来指示变量如何存储，而且在大多数实际应用中，最好认为 static 就是 private 的同义词，这样能更贴切地描述它的用途。

用关键词 static 声明变量可以使它们对于应用它们的函数来说是私有的。

53.3 Private Function

关键字 static 除了用于全局变量外，还可以用来指示某函数是某个特定模块的私有函数。

定义接口时，接口导出的函数不是私有的，而且接口的要点就是让这些函数可以在其他模块中调用。

在很多情况下，接口还包括一些只能在当前模块中调用的函数。要指出某一函数是否被限制在一个特定的模块中，可以把关键字 static 放在函数原型和其实现的前面。这样使得客户无法调用这些函数，从而使接口与用户间的抽象边界更加坚固稳定。

声明函数为 static 在由多个程序员参与开发的大型程序环境中也有好处。如果函数或者全局变量没有声明为 static，组成整个程序的模块集合中的其他模块就不能使用这些名字。

在不同函数中，避免名字相互干扰就需要独立模块的开发者之间必须相互沟通，确保不使用相同的名字，或者使用 static 关键字来保证他们使用的名字对于自己的模块的私有化。

如下的规则对于模块化开发来说是极好的指导：

> 静态声明规则：除了 main 函数和接口明确导出的函数之外，所有的函数必须被声明为 static。

一旦理解模块的内部状态可以用全局变量保存，实现扫描器抽象的实际流程就变得很简单。扫描器模块必须追踪传递给 InitScanner 的行以及行的当前位置(很可能会是个整型的下标)。

计算机科学中常使用术语缓冲区(buffer)来指出一个内部存储区，于是行的私有拷贝可以保存在

一个叫 buffer 的字符串变量中，当前位置保存在整型变量 cpos 中。记录缓冲区字符串的长度也很容易，这样函数无需每次重新计算字符串的长度。用于这个目的的变量也是个整数，可以命名为 buflen。

这些变量都得在扫描器模块中声明为全局且私有，这样别的模块无法访问这些变量。因此扫描器模块使用的全局变量有：

```
static string buffer;
static int buflen;
static int cpos;
```

声明了这些变量后，InitScanner 的实现就是将其初始化为正确的值。

```
void InitScanner(string line)
{
  buffer = line;
  buflen = StringLength(buffer);
  cpos = 0;
}
```

调用程序传入的行存在变量 buffer 中以追踪扫描器模块的行，变量 buflen 被设为缓冲区的长度，变量 cpos 设为 0 来指示扫描器处于行首。

GetNextToken 的实现精确地遵循接口中指定的函数定义。函数从查找行的下一个字符开始，如果字符是字母或者数字，函数搜索并找到同类字符的一个不间断的字符串并返回整个串；如果当前字符不是字母或者数字，函数就返回一个包含那个字符的单字符的字符串。因此，GetNextToken 的实现为

```
string GetNextToken(void)
{
  char ch;
  int start;
  if(cpos >= buflen) Error( "No more tokens" );
  ch = IthChar(buffer, cpos);
  if(isalnum(ch)){
    start = cpos;
    while(cpos < buflen && isalnum(IthChar(buffer, cpos))){
      cpos ++;
    }
    return (SubString(buffer, start, cpos - 1));
  }else{
    cpos ++;
    return (CharToString(ch));
  }
}
```

当下标 cpos 到达字符串的结尾时，记号流就完成了，因此得到下面关于 AtEndOfLine 的实现。

```
bool AtEndOfLine(void)
{
  return (cpos >= buflen);
}
```

下面是 scanner.c 的实现：

```
/*
 * File: scanner.c
 * ---------------
 * This file implements the scanner.h interface.
 */

#include <stdio.h>
#include <ctype.h>
```

```
#include "genlib.h"
#include "strlib.h"
#include "scanner.h"

/*
 * Private variables
 * -----------------
 * buffer -- Private copy of the string passed to InitScanner
 * buflen -- Length of the buffer, saved for efficiency
 * cpos  -- Current character position in the buffer
 */

static string buffer;
static int buflen;
static int cpos;

/*
 * Function: InitScanner
 * ---------------------
 * All this function has to do is initialize the private
 * variables used in the package.
 */

void InitScanner(string line)
{
   buffer = line;
   buflen = StringLength(buffer);
   cpos = 0;
}

/*
 * Function: GetNextToken
 * ----------------------
 * The implementation of GetNextToken follows its behavioral
 * description as given in the interface: if the next character
 * is alphanumeric (i.e., a letter or digit), the function
 * searches to find an unbroken string of such characters and
 * returns the entire string. If the current character is not
 * a letter or digit, a one-character string containing that
 * character is returned.
 */

string GetNextToken(void)
{
   char ch;
   int start;

   if (cpos >= buflen) Error("No more tokens");
   ch = IthChar(buffer, cpos);
   if (isalnum(ch)) {
      start = cpos;
      while (cpos < buflen && isalnum(IthChar(buffer, cpos))) {
         cpos++;
      }
      return (SubString(buffer, start, cpos - 1));
   } else {
      cpos++;
      return (CharToString(ch));
```

```
   }
}

/*
 * Function: AtEndOfLine
 * ---------------------
 * This implementation compares the current buffer position
 * against the saved length.
 */

bool AtEndOfLine(void)
{
   return (cpos >= buflen);
}
```

Part IX

Pointer

第 54 章

Introduction

在计算机科学，指针（Pointer）是一个用来指示一个内存地址的计算机语言的变量或中央处理器（CPU）中的寄存器（Register）。

程序设计的实质可以理解为改变指令中的位，因此指针一般出现在接近机器语言的语言（如汇编语言或 C 语言）中，面向对象语言（例如 Java 等）一般避免用指针，而是使用引用（reference）。

在 C 语言中，指针是计算机内存中一个数据值的地址，一般指向一个函数或一个变量。

在使用一个指针时，既可以直接使用这个指针所储存的内存地址，又可以使用这个地址所指向的变量或函数的值。

现实中，指针几乎能不受限制的在各种存储器地址间活动，所以一旦有任何重复、重叠、溢出的情形发生时，计算机便直接死机，这成为指针功能上的最大缺憾。后来，新的网络编程语言（如 Java、C# 等）的开发上，已经取消了指针的无限制使用形式。

- C# 允许指针的有限功能的使用，指针和运算指针在一个操作的环境中是存在潜在的非安全性的，因为它们的使用可以避开对象的一些严格访问规则。
- C# 中使用指针的代码段或者方法的地址要用 unsafe 关键字进行标记，这样这些代码的用户就会知道这个代码相比其他的代码而言是不具有安全性的。
- C# 编译器也需要 unsafe 关键字将使用此代码的程序转换成是允许被编译的。

一般来说，不安全代码的使用可能是为了非托管的 API（应用程序编程接口）的更好互用，或者是为了（存在内在不安全性的）系统调用，也有可能是出于提高性能等方面的原因，而 Java[1] 则取消了指针或者算术指针。

54.1 Formal Description

计算机中的内存都是编址的，每个地址都有一个符号，就像家庭地址或者 IP 地址一样。

根据 Pascal 的创始者 Niklaus Wirth 的说明，可执行程序由算法（代码）和数据组成。

- 算法是原始 C 语言程序中与语句对应的机器指令。
- 数据可以理解为原始程序中的变量。

可执行程序中的每个变量占用一个或多个内存字节，把第一个字节的地址称为 i 变量的地址，这就是指针的出处。

实际上，指针（Pointer）是一个数据项[2]，它的值就是其他值在内存中的地址。C 语言的设计意图是让程序员尽可能多地访问由硬件本身提供的功能，因而指针的使用非常普遍。

许多现代高级程序设计语言已经很少使用指针，这些语言提供了其他的机制避免使用指针。

32 位系统的寻址能力（地址空间）是 4GB（$0 \sim 2^{32}-1$），二进制表示长度为 32 比特，也就是 4Bytes。不难验证，在 32 位系统的大多数实现里，int 类型也正好是 4Bytes（32-bit）长度，可以用来编址上述范围。同理，64 位系统取值范围为 $0 \sim 2^{64}-1$，int 类型长度为 8Bytes。

```
#include <stdio.h>
```

[1] 实质上 Java 在传递对象的时候用的是按指针（这里认为指针和引用没有本质区别）传递，在传递基本类型（如 int）时用的是按值（副本）传递。

[2] 在 C 语言的多数实现中，指针值等同于一个无符号整数（unsigned int，因不致歧义，下面简称“整数”），它是一个以当前系统寻址范围为取值范围的整数。

```
main()
{
   char *pT;
   pT=(char *)1245048;
   putchar(*pT);
}
$ gcc -c test.c
$ ./a.out
Segmentation fault (core dumped)
```

char * 声明过的类型，一次访问 1 个 sizeof(char) 长度，double * 声明过的类型，一次访问 1 个 sizeof(double) 长度，因此程序中第 6 行加上“(char *)”是因为毕竟 unsigned int 和 char * 类型不同，需要强制转换，否则会有警告。

理论上声明一个无符号整数并使它的值等于对象的地址值时，也能使之有指针的作用。实际上，指针的取值范围可能不同于整数的范围，所以不能用普通整型变量存储地址，但是可以用指针变量来存储地址。

```
int i, *pT;
pT = &i;
```

在用指针变量 pT 存储变量 i 的地址时就是将指针 pT“指向”i，或者说指针就是地址，而且指针变量是只存储地址的变量。

汇编语言中没有数据类型这一概念，整数类型和指针就是一回事。不论是整数还是指针，在执行自增的时候，都是将原值加 1。

- 如果声明 char *pT;，汇编语言中 pT 自增(INC)之后值为 1245049，可是 C 语言中 pT++ 之后 pT 值为 1245049。
- 如果 32 位系统中声明 int *pT;，汇编语言中 pT 自增之后值为 1245049，可是 C 语言中 pT++ 之后 pT 值为 1245052。

编译器在编译源程序时决定了地址偏移量，操作系统在程序执行时决定了地址段的初始值和可访问长度，也就是说数据类型长度取决于编译器的位长。

C 语言的指针可以有多种用途，下面列出的是一些最重要的用途：

1. 指针允许程序员以更简洁的方式表示大的数据结构。
   程序中的数据结构可以任意大，但无论如何增长，数据结构总是位于计算机的内存中，因此必然会有地址。利用指针就可以使用地址作为一个完整值的速记符号，一个内存地址可以在操作系统内部表示为一个整数。当数据结构本身很大时，这种策略能节约大量内存空间。
2. 指针使程序的不同部分能共享数据。
   如果将某一个数据值的地址从一个函数传递到另一个函数，这两个函数就能使用同一份数据。
3. 利用指针能在程序执行过程中分配新的内存空间。
   在程序中能使用的内存可以通过显式声明分配给变量。引入指针后，在许多应用中就都能在运行时获得新的内存空间，并让指针指向这一内存。
4. 指针可用来记录数据项之间的关系。
   在高级程序设计应用中，指针被广泛用于构造单个数据值之间的联系。比如，通常在链表中的第一个数据的内部表示中包含指向下一个数据项的指针，从而可以说明这两个数据项之间有概念上的顺序关系。

54.2 Address Value

程序中的数据不仅能存储在简单变量中，还可以存储在更加复杂的数据结构(比如数组、枚举、结构、联合和堆栈等)中。

在 C 语言中，任何一个指向能存储数据的内存位置的表达式称为左值(lvalue，发音为 ell-value)，在左值开头的 l 表示在 C 语言中左值可以出现在赋值语句的左边。

左值是 C 语言中的任意表达式，左值有一个内存位置，所以具有地址。比如简单变量就是左值，因为可以写如下这样的语句：

```
x = 1.0;
```

同样地，选择表达式也是左值，从而可以直接为其赋值，比如：

```
intArray[2] = 17;
```

左值的地址称为指针，它可以像数据一样由程序操作，但是 C 语言中的很多值并不是左值(比如常量就不是左值)，其中常量就是不能够改变的。另外，算术表达式的返回值是值，但并不是左值，因为把值赋给算术表达式的结果是非法的。

利用左值的概念可以将变量的三条原则改写成更一般的形式：

1. 每个左值都存储在内存中，因此必定有地址。
2. 一旦声明左值，尽管左值的内容可以改变，但它的地址永远不能改变。
3. 按照所保存的数据类型，不同左值需要不同大小的内存。

再加上第 4 条原则可以更好地讨论指针：

4. 左值的地址本身也是数据，也能在内存进行操作和存储。

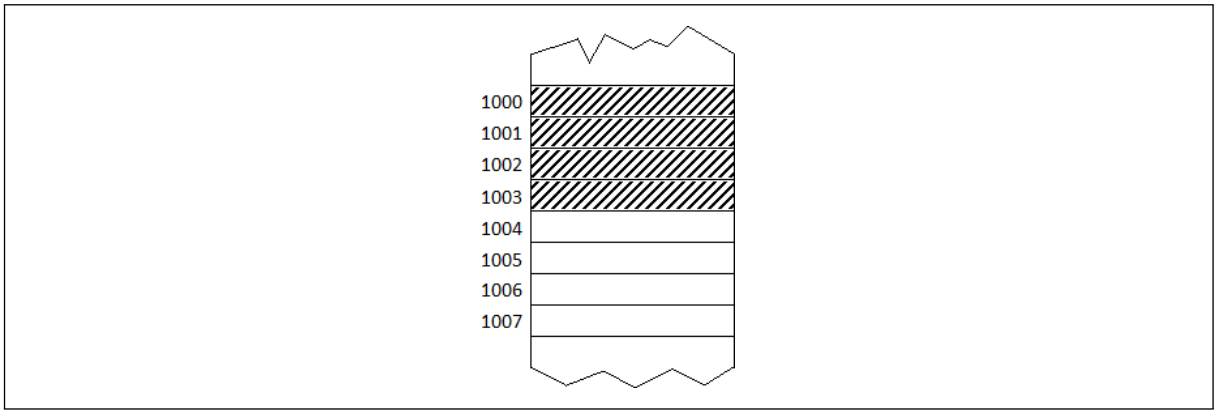
其中，最后一条原则对于程序设计时会显示出深远的意义。作为说明，考虑如下这样一个声明：

```
int i;
```

上述声明为整数 i 在内存的某处保留了一个存储空间来存储 i 中的数据。

大多数现代计算机使用字节(byte)来分割内存，每个字节都有唯一的地址(address)来和其他字节进行区分，一般每个字节可以存储 8 位的信息。

如果内存中有 n 个字节，那么可以认为作为地址的数的范围是 0 ～ n-1。例如，如果在运行程序的计算机上保存整数需要 4 个字节[3]的空间，那么变量 i 可能会得到从 1000 ～ 1003 的位置，如下图阴影部分所示。

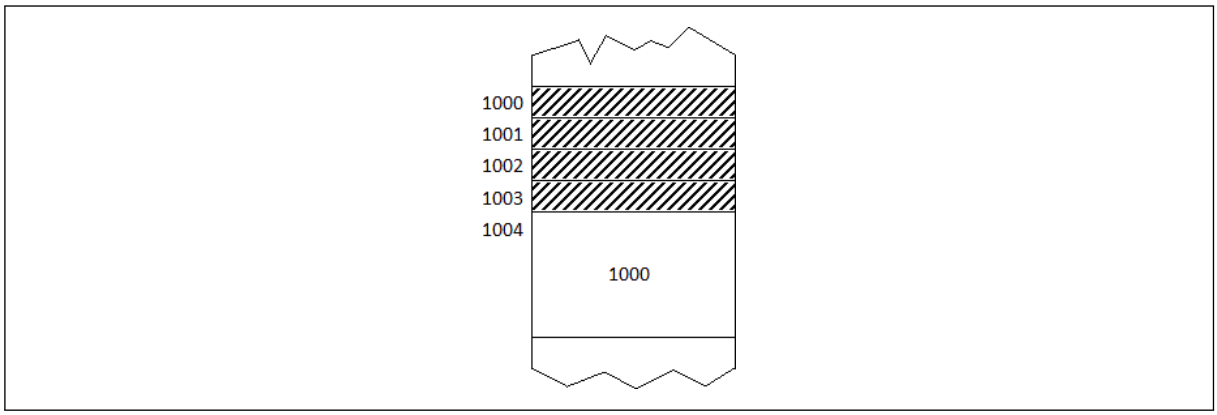


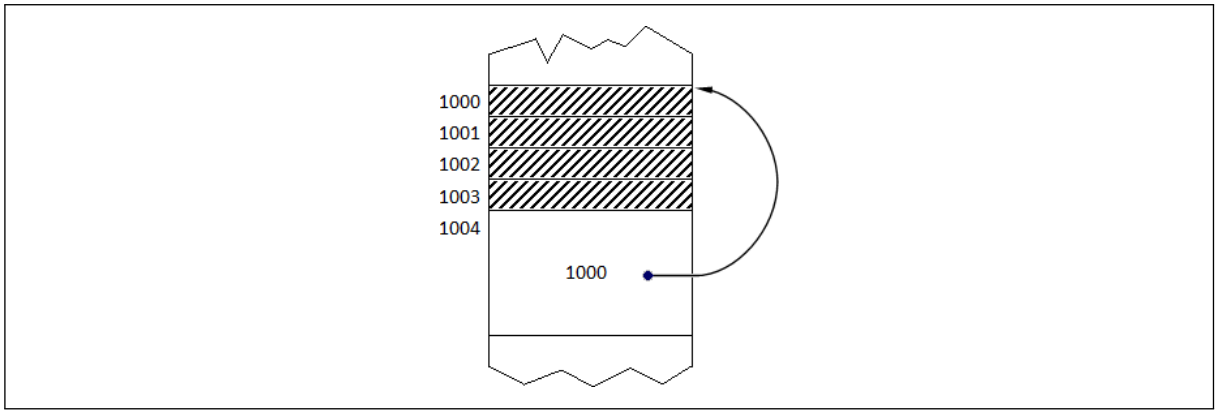
根据第 4 条原则，和变量 i 相关的地址 1000 本身也是一个数据值。毕竟，根据字面意义，值 1000 只是一个整数，可以存入内存。而在程序运行时，1000 正巧代表另一个值的地址，这对程序设计过程来说是很重要的，但并不影响值 1000 的内部表示。

整型变量 i 在内存中的存储方式和其他整型数一样。比如，可以把变量 i 的地址存人下一个内存字中，即地址 1004 ～ 1007 的字节。

出现在地址 1004 的值 1000 可以用来指向存放在阴影部分的变量 i 的地址。为了强调诸如位置在 1004 的地址和位置在 1000 的变量 i 之间的关系，可以在内存图上画上箭头(当然，在计算机内部是没有箭头的。)，如下图所示。

[3]这里假设保存整数需要 4 个字节，这样的变化是为了说明存储一个整数所需的空间会随着计算机的不同而改变。对任何对象的大小做出假设会限制程序在其他机器下的运行。





地址 1004 里的字只是包含了与数值 1000 对应的位串。

同样的位串作为整数使用还是作为地址使用取决于变量在程序中是怎样声明的。如果将变量声明为指针，就可以把存放在位置 1004 的值 1000 理解成内存中变量 i 的地址，并使用指针检索或操作 i 的值。

C 语言指针通常和地址是一样的，但是在用字而不是字节划分内存的计算机中，可能需要用不同于其他指针的格式存储地址。

一般情况下，字可以包含 36 位、60 位或者更多位。如果假设使用 36 位来保存字，那么当用字来划分内存时，每个字的地址可能有不同的格式。

- 如果整数占一个字长度，则指向整数的指针可能和地址格式一致。
- 如果字存储多于一个的字符，可能就需要用不同于其他指针的格式存储指向字符的指针。

  指向字符的指针可以由地址(存储字符的字)加上一个小整数(字符在字内的位置)组成。

在另外一些计算机上，指针可能是“偏移量”而不完全是地址。例如，Intel 处理器具有复杂的模式，其中的地址有时用单独的 16 位数(偏移量)来表示，有时用两个 16 位数(段:偏移量对)来表示。

偏移量不是真正的内存地址，处理器必须把偏移量和存储在特殊寄存器中段的值联合起来才能得到真正的内存地址。

C 语言编译器通过提供两种指针的方式处理 Intel 的分段结构:近指针(16 位偏移量)和远指针(32 位段:偏移量对)，而且通常保留关键字 near 和 far 用于指针变量的声明。

54.3 Memory Allocation

在程序编译或者运行时，操作系统会开辟一张表。每遇到一次声明语句(变量的声明、函数的声明和传入参数的声明等)都会开辟一个内存空间，并在表中增加一行记录来记载一些对应关系。

C 语言的结构体(汇编语言对应为 Record 类型)按顺序分配空间。涉及到内存对齐的问题时，编译器会把结构体的大小规定为结构体成员中大小最大的那个类型的整数倍。

下面的声明用一个整数来代表一棵树的结点。

```
typedef struct BiTree
{
  int value;
  struct BiTree *LeftChild;
  struct BiTree *RightChild;
}BTree;
```

通过把它赋给某个结点的 LeftChild/RightChild 的值，就形成了上下级关系。如果无法找到一个路径，使得 A->LC/RC->LC/RC···->LC/RC==A（A 泛指某一结点），就构成了一棵二叉树，反之就构成了图。

如果没有指针，我们很难用一个统一的模式去 A 的定位并修改一棵树的结点。例如，不用指针要修改 A 的左子树的左子树的右子结点，只有“A.LC.LC.RC=···”一种表达方式，不能通过赋值而简化。

另外，C 语言函数调用是按值传递的，传入参数在子函数中只是一个初值相等的副本，无法对传入参数作任何改动。为了修改传入参数的值，即传入参数的地址而不是原参数本身，可以通过对传入参数（地址）取“*”运算来直接在内存中修改，从而可以修改传入参数的参数值。

```
#include <stdio.h>
void inc(int *val)
{
  (*val)++;
}
int main()
{
  int a = 3;
  inc(&a);
  printf(''%d", a);
  return 0;
}
$ gcc -c test.c
$ ./a.out
4
```

在执行 inc(&a); 时，系统在内存分配表里增加了一行“val@inc”，其地址为新地址，值为 &a。操作 *val 就是通过传递地址在操作 a。

在下面的例子中，main() 内的变量从来没有改变，改变的只是 sw() 内的变量。

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

void sw(int x, int y)
{
  int Temp;
  Temp = x;
  x = y;
  y = Temp;
}

int main()
{
  int a = 1;
  int b = 2;
  cout << a << b << endl;
  sw(a, b);
  cout << a << b << endl;
  return 0;
}
$ g++ -o test test.cpp
```

```
$ ./test
12
12
```

当 sw() 函数执行完毕后，其内容会自动删除。如果传递给 sw() 函数的是指针变量，执行时通过交换指针变量的地址来达到交换两个值的效果。

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

void sw(int *x, int *y)
{
  int Temp;
  Temp = *x;
  *x = *y;
  *y = Temp;
}

int main()
{
  int a = 1;
  int b = 2;
  cout << a << b << endl;
  sw(&a, &b);
  cout << a << b << endl;
  return 0;
}
$ g++ -o test test.cpp
$ ./test
12
21
```

第 55 章

Manipulation

C 语言具有操作指针的能力，就和 C 语言具有的操作其他数据类型的能力一样，可以存储指针变量中的地址值，或是将地址值作为参数传递给函数。

55.1 Declaration

和 C 语言中其他变量一样，使用指针变量前必须先对其进行声明。

对指针变量的声明与对普通变量的声明基本一样，唯一的不同是必须在指针变量名字前放置星号 *。比如，下面的声明

```
int *p;
```

将变量 p 声明为概念上指向整型变量的指针类型。

类似的，声明

```
char *cp;
```

表示声明变量 cp 为指向字符型变量的指针类型。

虽然所有类型的指针变量在计算机内部都是以地址形式表示的，但指向不同对象的指针在 C 语言中还是有区别的。特别地，指针变量可以指向另一个指针，即指向指针的指针。

要使用某地址中的数据，编译程序必须知道如何解释地址中存储的数据，因此就要求显式地说明指针所指向的数据的类型。

指针指向的值的类型称为指针的基本类型（base type），所以指向整型的指针的基本类型为 int，指向字符型的指针的基本类型为 char，但是指针也可以指向不用作变量的内存区域。

语法：指针声明

```
base-type *pointer-variable
```

其中：

base-type是指针指向的值的类型；

pointer-variable就是正在被声明的指针变量。

必须注意，表示变量为指针的星号在语法上属于变量名，不属于基本类型。

如果使用同一个声明语句来声明两个同类型的指针，必须给每个变量都加上星号标志。

```
int *p1, *p2;
```

而声明

```
int *p1, p2;
```

则表示声明 p1 为指向整型的指针，声明 p2 为整型变量。当然，这也说明指针变量可以和其他变量一起声明。

C 语言要求每个指针变量每次都指向唯一的特定类型（引用类型）的对象，而且对于引用类型没有限制。

常见错误:新程序员有时会忘记在声明指针时在每个变量前都必须加上 *。

指针不仅可以指向程序中的数据,还可以通过指向函数的指针来指向程序代码,也就是说指针可以指向不用作变量的内存区域。

55.2 Operation

C 语言定义了两种操作指针值的运算符:& 和 *。

- 取地址运算符 & 可以取一个左值并返回一个指向该左值的指针。
  & 运算符把对应于某个内存中的值的表达式作为操作数,这个操作数通常是一个变量或者是一个数组引用。操作数写在 & 后,且必须是一个左值。
  给定一个特定的左值,& 运算符会返回存储该左值的内存地址。
- 间接寻址运算符(*)可以取一个指针并返回该指针所指向的左值。
  * 运算符取任意指针变量的值,并返回其指向的左值。* 操作产生一个左值,说明可以赋一个值给一个间接引用的指针。
  将指针移至它所指向的左值,这一操作称为对指针间接引用(dereferencing)。

如果 p 是指针,那么对指针的间接引用 *p 表示 p 当前指向的对象,而对指针的寻址操作 &p 则表示指针 p 的地址。

考虑以下声明:

```
int x, y;
int *p1, *p2;
```

上述这些声明为 4 个变量预留了内存空间,其中 x 和 y 是 int 类型,p1 和 p2 是指向整型变量的指针变量,但是并没有指向确定的对象。

为了更具体一些,可以假设这些值在机器中存放的地址如下图(a)所示,并通过赋值语句给 x 和 y 赋值。

比如,执行赋值语句:

```
x = -42;
y = 163;
```

可以对普通变量 x 和 y 进行赋值,并产生如下图(b)所示的内存状态。

为了初始化指针变量 p1 和 p2,需要将那些普通变量的地址值赋给它们。

55.3 Initialization

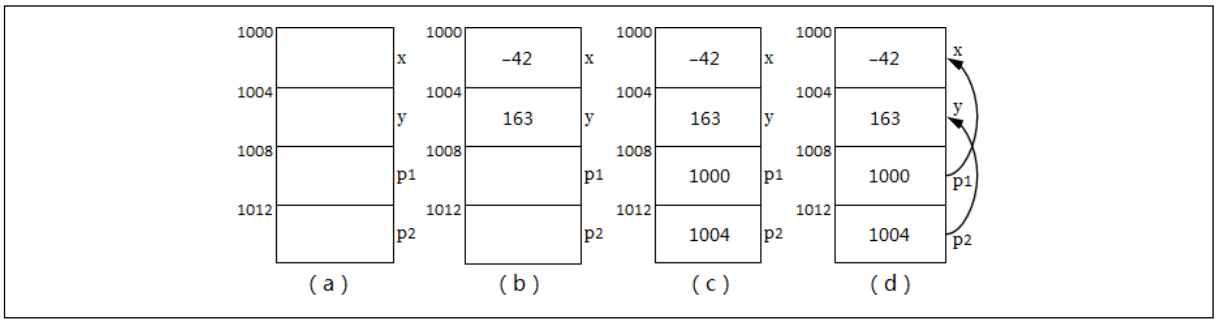
和普通变量一样,声明指针变量可以指示编译器为指针预留出空间,但是并没有把它指向具体的对象,因此在使用前初始化指针变量是很重要的。

可以通过把某个变量的地址赋给指针变量,或者采用左值,因此可以用取地址运算符 & 把 x 和 y 的地址分别赋给 p1 和 p2:

```
int x, y, *p1, *p2;
p1 = &x;
p2 = &y;
```

上述赋值操作将内存变成了下图(c)所示的状态,其中 p1 和 p2 中的指针值再一次具有了指向它们所引用的变量的直观效果,可以用箭头把这种关系用下图(d)表示。

取地址运算符可以出现在声明中,因此在声明指针的同时对它进行初始化是可行的。



55.3.1 Addressof

对变量使用 & 运算符可以产生指向变量的指针，对指针使用 * 运算符则可以返回原始变量。假设内存结构如上面的图所示，那么 *p1 就是变量 x 的另外一个名字。

```
int x = *p1;
int y = *&i;
```

注意，不要把间接寻址运算符用于未初始化的指针变量。如果指针变量没有被初始化，那么使用间接寻址运算输出的值是未定义的。

```
#include <stdio.h>
main()
{
  int i;
  int *p = &i;
  printf("%d\n", *p);
}
$ gcc -c test.c
$ ./a.out
32767
```

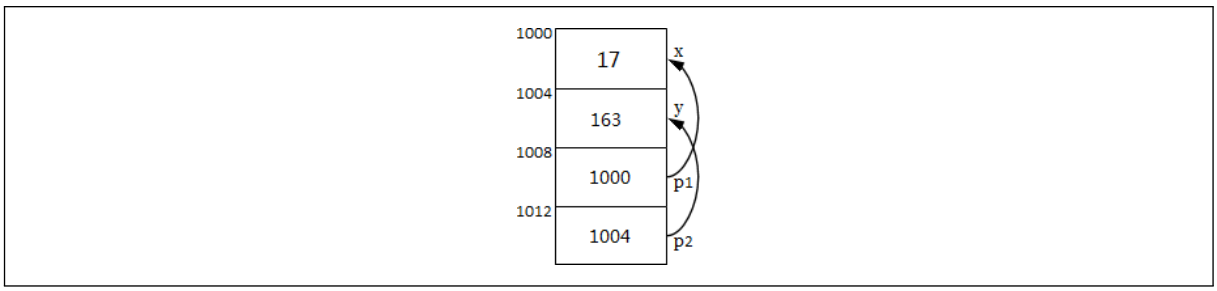
或者，可以把普通变量 i 的声明及其指针变量 p 的声明合并，但是需要首先声明普通变量 i。

```
int i, *p = &i;
```

和简单变量 x 一样，指出 *p1 和 x 的关系后，表达式 *p1 是一个左值，而且可以给它赋新值。执行赋值语句

```
int x = *p1;
*p1 = 17;
```

会改变变量 x 的值，因为 p1 指向变量 x。对 *p1 赋值之后，内存结构就变成如下所示。



可以看到，p1 的值本身没有因赋值而受到影响，它的值仍然为 1000，即仍然指向变量 x。

55.3.2 Dereference

为了访问存储在指针所指向的变量，可使用 *（间接寻址）运算符。例如，在下面的示例中声明了整型指针变量 p1。

```
int *p;
printf("%d", *p); /* prints garbage */
```

那么，通过 printf 函数将会输出 p 指向的内存位置里所存放的值。

- 如果 p 被声明为指向整数的指针，那么编译器知道表达式 *p 一定是一个整数。
- 如果指针变量 p 没有初始化，那么 *p 的值是未定义的。

未初始化的指针变量 p 可以指向内存中的任何地方，对 *p 进行赋值时就可能会改变某些未知的内存单元。

下面的赋值语句改变的内存单元可能属于程序（可能导致未定义的行为）或者属于操作系统（可能导致系统崩溃），因此不要把间接寻址运算符用于未初始化的指针变量。

```
int *p;
*p = 1; /* wrong */
```

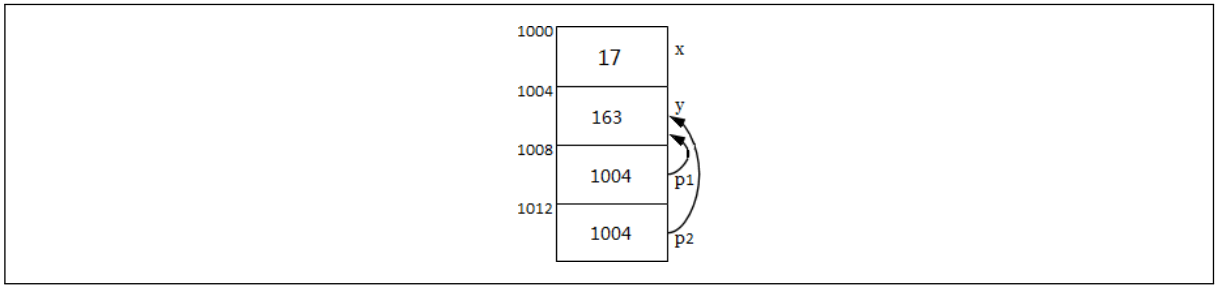
55.4 Assignment

在指针变量具有相同的类型的前提下，C 语言允许使用赋值运算符来给指针变量赋新值。

指针赋值使两个指针指向同一位置，值赋值将一个由指针确定的地址中的值复制到另一指针确定的地址中去。例如，语句

```
int i, j;
int *p1, *p2;
p1 = &i;
p2 = &j;
p1 = p2;
```

指示操作系统取出（指针）变量 p2 的值并将该值复制给（指针）变量 p1。p2 的初始值是指针值 1004，通过上述赋值语句将 p2 的值赋给 p1，结果就是 p1 和 P2 就会同时指向变量 y，如下图所示。



C 语言允许任意数量的指针变量都可以指向同一个对象，而且就发生在机器内部的操作来说，复制一个指针和复制一个整数是完全一样的，指针的值被原封不动地复制到目的地。

从概念角度来说，复制指针的效果就是将目的地指针处的箭头替换成与原指针指向同一位置的箭头。下面的赋值语句

```
p1 = p2;
```

的效果就是改变了从 p1 出发的箭头，使其和从 p2 出发的箭头指向同一内存。能够区分指针赋值和值赋值是很重要的。

形如

p1 = p2;

的指针赋值语句使 p1 和 p2 指向同一位置。

形如

*p1 = *p2;

的值赋值语句是把以 p2 为地址的内存单元里的值复制到以 p1 为地址的内存单元里去。

第 56 章

Parameter

56.1 Actual Argument

在调用函数时传递指向变量的指针，可以使得函数改变变量的值，而不再是改变变量的拷贝的值。

C 语言用值进行参数传递，在函数调用中用变量作为实际参数时会阻止对变量值的改变。如果函数需要改变变量自身而非拷贝的值，就只能采用指针。

使用指针可以不用再传递变量自身作为函数的实际参数，而是使用指向变量的指针。

假设变量为 x，如果使用指向 x 的指针作为函数的实际参数，那么函数体内对指针的间接引用允许函数对变量进行读取和修改。

下面通过把形式参数 int_part 和 frac_part 声明成指针来修改 decompose 函数。

```
void decompose(float x, int *int_part, float *frac_part)
{
  *int_part = (int) x;
  *frac_part = x - *int_part;
}
```

其中，decompose 函数的原型既可以是

```
void decompose(float x, int *int_part, float *frac_part)
```

也可以是

```
void decompose(float x, int *, float *)
```

例如，当以下列方式调用 decompose 函数时，实际参数不再是 i 和 f 的值，而是指向 i 和 f 的指针。

```
decompose(3.14159, &i, *f);
```

在接下来的函数执行过程中，首先把值 3.14159 赋值给 x，把指向 i 的指针赋给 int_part，并把指向 f 的指针赋给 frac_part 中。

首先，

```
*int_part = (int) x;
```

把 x 的值转换为 int 型，并把该 int 型值存储到 int_part 指向的内存单元中。

接着，

```
*frac_part = x - *int_part;
```

把 int_part 指向的值取出并转换为 float 类型，然后与 x 进行相减运算，再把得到的 float 型结果值存储到 frac_part 指向的内存单元中。

这样，当 decompose 函数返回时，x 不变，但是 i 和 f 所指向的内存单元中的值都以被新值填充。

在标准库 stdio.h 中提供的 scanf 函数也是以指针作为函数的实际参数来运作的。

```
int i;
scanf("%d", &i);
```

传递给 scanf 函数的实际参数 &i 是 i 的地址，从而可以指示 scanf 函数要读取的值存放的位置。

虽然要求 scanf 函数的实际参数是指针，但是并不是每个实际参数都需要 & 运算符。在下面的例子中，指针变量 p 包含了整型变量 i 的地址，因此 p 就是要传递给 scanf 函数的指针变量。

```
int i, *p;
p = &i;
scanf("%d", p);
```

向函数传递需要的指针却失败了可能会产生严重的后果。假设传递给 decompose 函数的不是指针而是普通变量。

```
decompose(3.14159, i, f);
```

本来 decompose 函数期望指针作为其第二和第三个实际参数，但却被 i 和 f 的值替换掉，这样函数就会把值写入到未知的内存单元中，而不是修改 i 和 f。

如果已经提供了 decompose 函数的原型，那么编译器将会提示正在试图传递错误的实际参数类型，然而编译器通常不会检查在 scanf 函数中指针传递失败的信息。

C++ 语言提供了可以不需要传递指针就能修改函数的实际参数的方法。

56.2 Call By Reference

C 语言中指针的最常见的应用之一就是通过传递指针给函数来允许该函数操作其调用程序的数据。

在 C 语言中，把一个简单变量从一个函数传递到另一个函数时，该函数就得到了一个调用值的拷贝。在函数的语句中给参数赋一个新值会改变参数在这个函数中的局部副本，但不会影响调用参数。

例如，如果要实现一个用以下方法将某个变量初始化为 0 的函数：

```
void SetToZero(int var)
{
  var = 0; /*（此程序是有错的，事实上无效）*/
}
```

实际运行时，该函数不起任何作用。

如果调用

```
SetToZero(x);
```

参数 var 会被初始化为任何存储在 x 中的值的副本值。函数中的赋值语句

```
var = 0;
```

只是将函数参数副本的值改变成 0，但调用函数中的 x 的值是不变的。

解决这个问题的方法之一就是给函数传递指向变量的指针，而不是传递变量本身。尽管采用此方法会改变函数结构，但为了使函数能发挥效用，这是必要的。

```
void SetToZero(int *ip)
{
  *ip = 0;
}
```

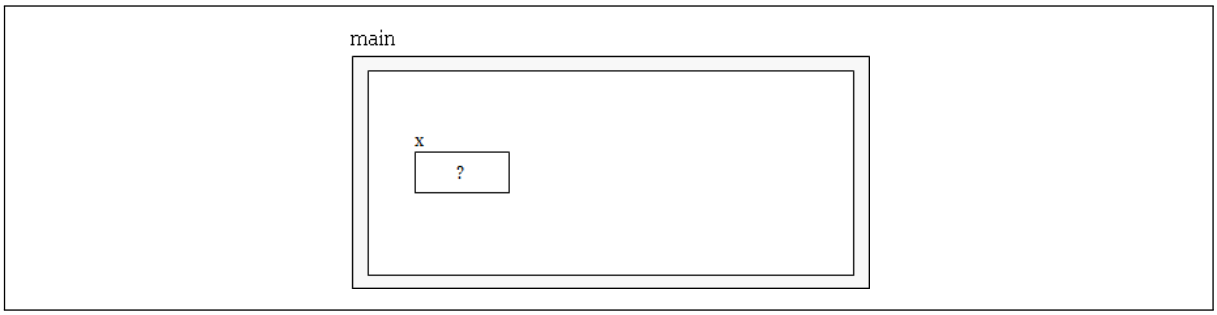
为了使用这个函数，调用程序必须提供一个指向整型变量的指针。例如，要将 x 置为 0，需要作出如下调用

```
SetToZero(&x);
```

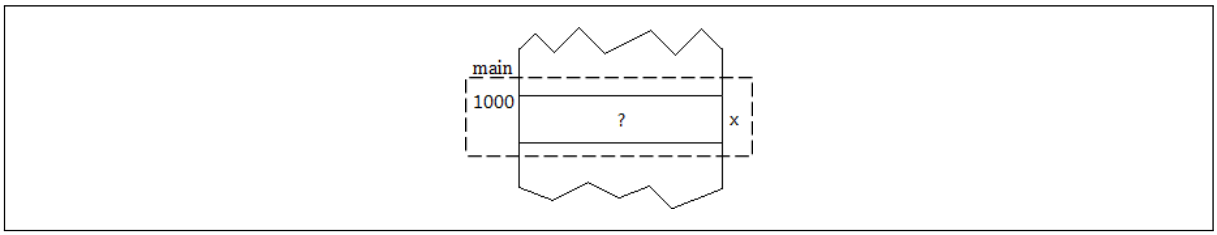
如果遗漏 & 会产生错误，因为只是 x 的话传递的不是需要的类型，SetToZero 要求的是一个指向整数的指针，而不是整数本身。

为了说明得更清楚，假设 main 函数调用了 SetToZero，main 函数的帧包括一个名为 x 的整型变量。

在主程序调用 SetToZero 之前，main 函数的帧如下图所示：



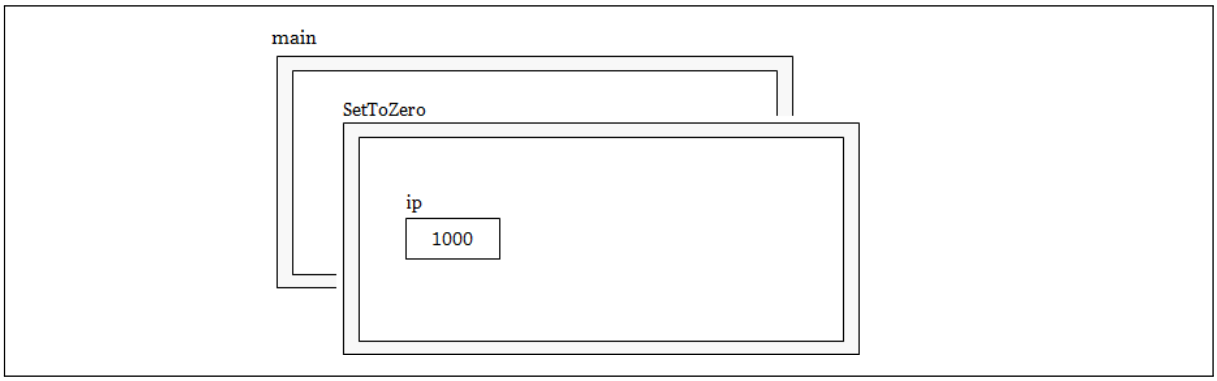
尽管帧图中没有包括地址，但重要的是要认识到变量 x 就处于内存的某处。例如，假设变量存储在地址 1000 中，在内存中的具体表示就是包含该地址的内容的帧，如下图所示。



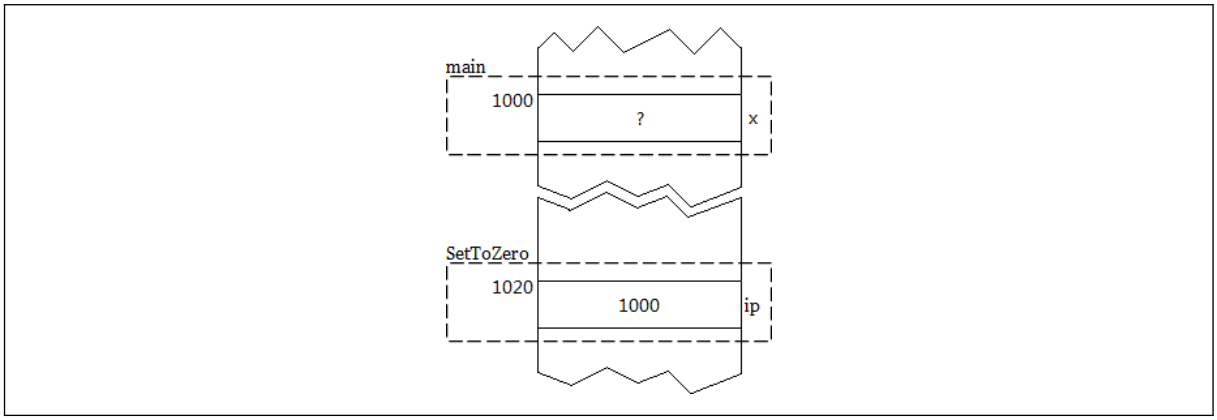
当 main 用以下语句调用 SetToZero 时，就会为 SetToZero 函数建立一个新帧，它的参数是变量 ip，这是一个指向整型的指针。

```
SetToZero(&x);
```

调用 SetToZero 函数后，ip 被初始化为调用参数的值 &x——即 x 的地址，此时 SetToZero 的帧如下图所示：



而且，这个帧同样也处于内存的某处，因此这个时候内存应该如下图所示。

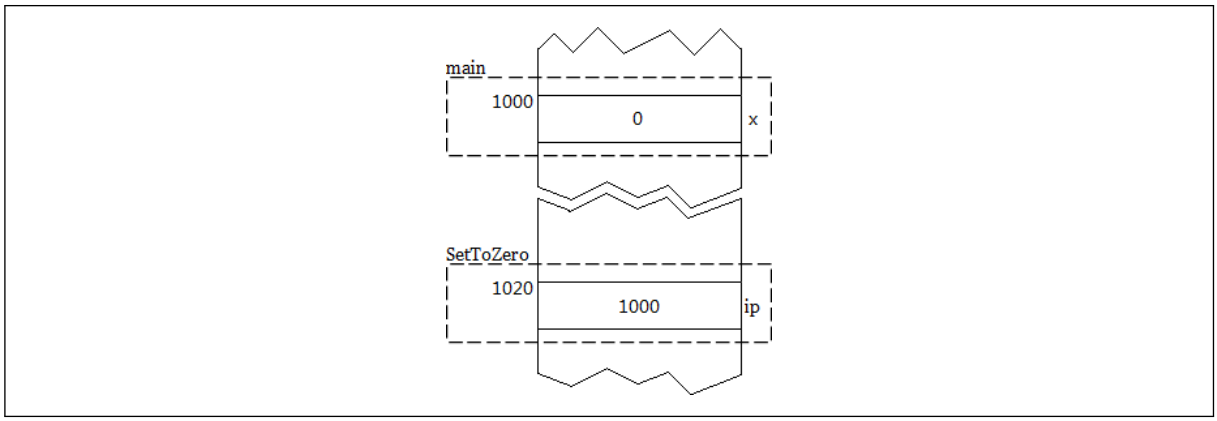


实际上哪个地址会分配给这些帧是没有办法预测的，而且这两个帧的相对方位也是无法预测的。在大多数现代的计算机中，一般 SetToZero 帧的内存地址比 main 帧的内存地址小，但理解内存结构并不依赖于任何这样的假定。

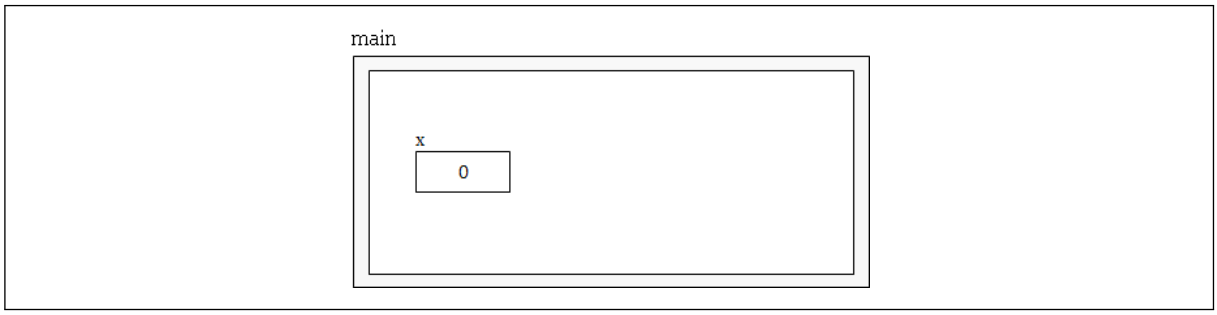
语句

```
*ip = 0;
```

能把地址为 ip 的整型字的值置为 0，如下图所示：



当 SetToZero 函数返回时，地址 1000 中值的改变还是有效的，所以 main 函数的帧现在如下图所示。



因此，指针作为参数使函数能够改变其调用函数的帧中的值。

在 C 语言中，指针可以用于使一个函数与其调用函数共享数据，这一过程称为引用调用（call by reference）。在很多其他语言中，引用调用机制是语言定义的一部分。

C 语言必须通过声明参数值为指针类型明确说明想要改变值，然后将地址作为参数传递，这种做法的优势在于函数调用自身的句法指出了参数变量值在函数执行的过程中是否能被改变。

例如，在完全不了解函数 Mystery 的情况下，可以知道变量 x 的值在进行如下调用后没有发生改变：

```
Mystery(x);
```

如果 Mystery 要改变 x 的值的话，函数必须重新定义为有一个指针参数，应用这一规则使得我们容易预测某个函数调用的结果。

```
Mystery(&x);
```

56.3 SwapInteger

为了说明引用调用是怎样影响程序设计的，下面讨论使用指针重新实现选择排序算法。

原先 SortIntegerArray 过程的实现如下：

```
void SortIntegerArray(int array[], int n)
{
  int lh, rh;

  for(lh = 0; lh < n; lh ++){
    rh = FindSmallestInteger(array, lh, n -1);
    SwapIntegerElements(array, lh, rh);
  }
}
```

在 for 循环的每个周期中，过程都会找出表中剩余值中的最小值的下标，并将它与下标为 lh 的值交换。for 循环中的最后一个语句通过以下函数调用完成了交换操作：

```
SwapIntegerElements(array, lh, rh);
```

这里的函数调用似乎设计得并不好，调用该函数的目的是交换两个整数，如果换成如下调用，似乎这段代码传达的基本概念才更明确一些：

```
SwapIntegerElements(array[lh], array[rh]); /*无效*/
```

但是，这种调用在 C 语言中是无效的，因为无法定义如 SwapInteger 这样的函数。根据数组元素的性质，无法改变调用参数的值，而这个函数则要求必须改变调用参数的值。

引入指针之后，可以使用引用调用来达到同样目的，如果传递给 SwapInteger 的是参数的地址而非参数的值，那么就可能实现如下的函数：

```
static void SwapInteger(int *p1, int *p2)
{
  int tmp;
  tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
}
```

SwapInteger 函数有两个指向整数的指针，并交换两个指针指向的内存单元的值。

56.4 Returns

56.4.1 Return Pointer

C 语言中可以为函数传递指针，也可以开发返回指针的函数。例如，可能希望函数返回结果的内存位置而不是返回值。

当给定指向两个整数的值时，下面的 max 函数返回指向两个整数中较大数的指针。

```
int *max(int *a, int *b)
{
  if(*a > *b)
    return a;
  else
    return b;
}
```

调用 max 函数时，将传递两个指向整型变量的指针，并且把结果存储在指针变量中。

```
int *p, x, y;
p = max(&x, &y);
```

调用 max 函数后，x 和 y 的地址将会被作为实际参数传递给 max 函数，因此 *a 是 x 的别名，而 *b 是 y 的别名。如果 x 大于 y，那么 max 返回 x 的地址，否则返回 y 的地址，并把返回的指针赋给指针变量 p。

上述的示例中，max 函数返回的指针是作为实际参数传递的两个指针中的一个。

另外,一些函数返回的指针可以作为实际参数传递的数组中的一个元素,也可能返回指向外部变量或静态局部变量的指针。

在 C 语言中,返回指针的函数永远不会返回指向自动局部变量的指针。

```
int *f(void)
{
  int i;
  ...
  return (&i);
}
```

在函数的生存期结束并返回后,局部自动变量 i 就被销毁了,所以指向变量 i 的指针将是无效的。

56.4.2 Multiple Results

使用引用调用的最常见情况就是当一个函数需要给调用程序返回多个值时。

单个结果可以很简单地作为函数本身的值返回。如果需要从一个函数返回多个结果时,返回值就不再合适了,解决此问题的标准方法是把函数变成一个过程,并通过参数表来回传递值。

例如,假设想要写一个函数,用于将用分钟表示的时间转换成合适的以小时和分钟表示的时间。例如,235 分钟等于 3 小时 55 分钟。

作为一个函数,这一计算取一个以分钟表示的总时间,并且返回了两个值:表示小时的数字和剩余的表示分钟的数字,通常范围为 0 ～ 59。因为这一操作有两个结果,所以可以把它写成使用引用调用的过程。

这个需求的实现以及测试程序中,过程 ConvertTimeToHM 有三个参数,第一个参数(time)给函数提供输入数据,后两个参数(pHours,pMinutes)使函数能把结果传递给调用程序。

在接下来的调用程序中,后两个参数必须指定为地址,数据可以存入这些地址。

```
ConvertTimeToHM(time, &hours, &minutes);
```

具体的实现代码如下:

```
/*
 * File: hours.c
 * -------------
 * This program converts a time given in minutes into
 * separate values representing hours and minutes. The
 * program is written as an illustration of C's mechanism
 * for simulating call by reference.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Constants */

#define MinutesPerHour 60

/* Function prototypes */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes);

/* Test program */

main()
{
   int time, hours, minutes;
```

```
   printf("Test program to convert time values\n");
   printf("Enter a time duration in minutes: ");
   time = GetInteger();
   ConvertTimeToHM(time, &hours, &minutes);
   printf("HH:MM format: %d:%02d\n", hours, minutes);
}

/*
 * Function: ConvertTimeToHM
 * Usage: ConvertTimeToHM(time, &hours, &minutes);
 * -----------------------------------------------
 * This function converts a time value given in minutes into
 * an integral number of hours and the remaining number of minutes.
 * Note that the last two arguments must be passed using their
 * addresses so that the function can correctly set those values.
 */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes)
{
   *pHours = time / MinutesPerHour;
   *pMinutes = time % MinutesPerHour;
}
```

在这个实现中,这些值被表示为指向实际变量的指针,所以为了计算小时数,合适的赋值语句为:

```
*pHours = time / MinutesPerHour;
```

要使用间接引用运算符(*)来保证该结果存储在 pHours 指向的变量中。

56.5 Overusing

尽管引用调用的策略具有很高的应用价值,但也很容易被过度使用。

56.5.1 Disable Pointer

在大多数情况下,特别是在后续引入记录之后,就可以重新设计程序,从而让所有的结果都作为函数值返回。有返回值的函数通常比过程更易使用,主要是因为函数调用可以嵌套。

可以把一个函数的结果作为参数传递给另一个函数,继续这一过程,直到应用不再需要为止。当使用过程时,必须把它们作为单独的语句一个一个地调用。任何从一个过程传递到另一个过程的值都必须存储在一个变量里并通过参数表传递。

例如,为了避免使用引用调用,可以定义两个独立的函数来代替过程 ConvertTimeToHM:一个是 Hours 函数,用于返回表示小时的整数,另一个是 Minutes 函数,返回剩余的分钟。

这一方法的主要优势在于没有一个函数用到指针。比如,Hours 函数仅仅是:

```
int Hours(int time)
{
  return (time / MinutesPerHour);
}
```

从而可以将主程序 hours.c 简化为:

```
main()
{
  int time;

  printf( "Test program to convert time values\n" );
```

```
  printf( "Enter a time duration in minutes:" );
  time = GetInteger();
  printf( "HH:MM format: %d:02d%\n" , Hours(time), Minutes(time));
}
```

其中两个局部变量(hours 和 minutes)在这个程序中已经不复存在了,因为现在可以直接使用函数结果而不必事先将它们赋给变量。

56.5.2 Protect Argument

当调用函数并且传递给它指向变量的指针时,通常会假设函数将修改变量。

对于仅需要检查变量的值而不是修改它的值的情况,使用指针变量要比使用普通变量高效,原因在于如果变量需要的大量的存储空间,那么传递变量的值可能会浪费时间和空间。

使用使用类型限定符 const 可以避免函数修改传递给函数的指针所指向的对象。为了允许函数检查传递的指针所指向的实际参数而不加以修改,可以在参数声明中把类型限定符 const 放置在形式参数的类型说明前面。

```
void f(const int *p)
{
  *p = 0; /* wrong */
}
```

使用 const 修饰形式参数说明 p 是指向"整型常量"的指针,这样就不能改变指针 p 指向的整数,试图改变 *p 将会引发编译器发出特定的信息。

在 C 语言中,实际参数是按值传递的,因此通过使指针指向其他地方的方法给指针变量赋新值不会对函数外产生任何影响,也就是说 C 语言并不阻止改变指针自身。

```
void f(const int *p)
{
  int j;
  p = &j; /* legal */
}
```

另外,当声明指针类型的形式参数时,在参数名前面放置 const 也是合法的。

在指针的类型前面放置 const 可以保护指针指向的对象,而在指针的类型后面放置 const 可以保护指针本身。

```
void f(int * const p)
{
  int j;

  *p = 0; /* legal */
  p = &j; /* wrong */
}
```

指针很少是另一个指针(调用函数时的实际参数)的副本,因此上述特性并不经常用到。

还有一类极少出现的情况是需要同时保护指针及其指向的对象,这可以通过在指针类型和前面和后面都放置 const 来实现。

```
void f(const int * const p)
{
  int j;
  *p = 0; /* wrong */
  p = &j; /* wrong */
}
```

第 57 章

Array & Pointer

在 C 语言中，指针和数值密切相关，可以将指针像数组一样使用，反之亦然。指针和数组之间的关系使得算术运算符 + 和-也可以用于指针领域。

另外，在指针算术运算中，运算符 ++ 和--既可以写在它们作用的操作数之前，也可以写在操作数之后。

57.1 Array Address

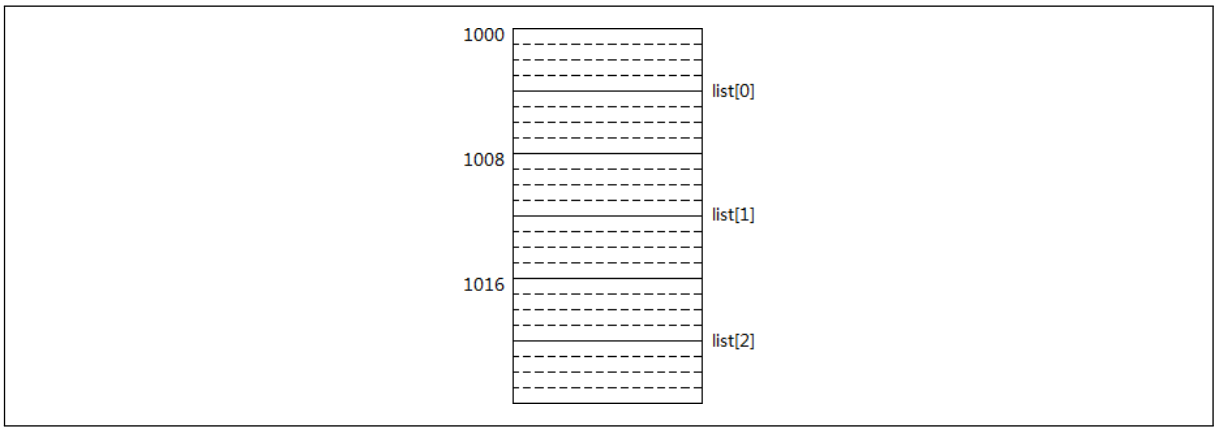
指针可以用作函数实际参数来实现同时修改源地址数据的目的，而且指针也可以用作函数的返回值。但是，指针的应用并不仅限于指向某一简单变量的情况，实际上指针可以指向任何左值。特别是，数组元素是左值，所以也有地址。

使用指针处理数组可以提高程序效率，因此理解指针和数组之间的关联可以深入理解 C 语言的设计过程，并帮助理解现有的程序。

例如，下列数组声明

```
double list[3];
```

预留了三个连续的内存单元，每个单元的大小都足以存放一个 double 类型的值。假设 double 类型为 8 个字节长，该数组占用的内存如下图所示。



三个数组元素都有地址，这些地址可使用 & 运算符得到。例如，表达式的指针值为 1008，因为元素 list[1] 存储于该地址。

```
&list[1]
```

此外，下标值可以不是常数，例如选择表达式 list[i] 是一个左值，所以书写 &list[i] 是合法的，它表示 list 中第 i 个元素的地址。

由于 list 中第 i 个元素的地址取决于变量 i 的值，所以 C 语言编译器在编译程序时是无法计算这一地址的。

为了确定数组元素的地址，编译器首先产生一段指令取数组的基地址，再加上适当的偏移量，偏移量是将 i 的值乘以每个数组元素的字节数而得到的。所以，计算 list[i] 的地址可以由下面的公式得

到。

$$1000 + i \times 8$$

例如，如果 i 为 2，地址计算的结果即为 1016，与图中显示的 list [2] 的地址是一致的。因为计算任何一个数组元素地址的过程是自动的，所以在写程序时无需考虑计算的细节问题。

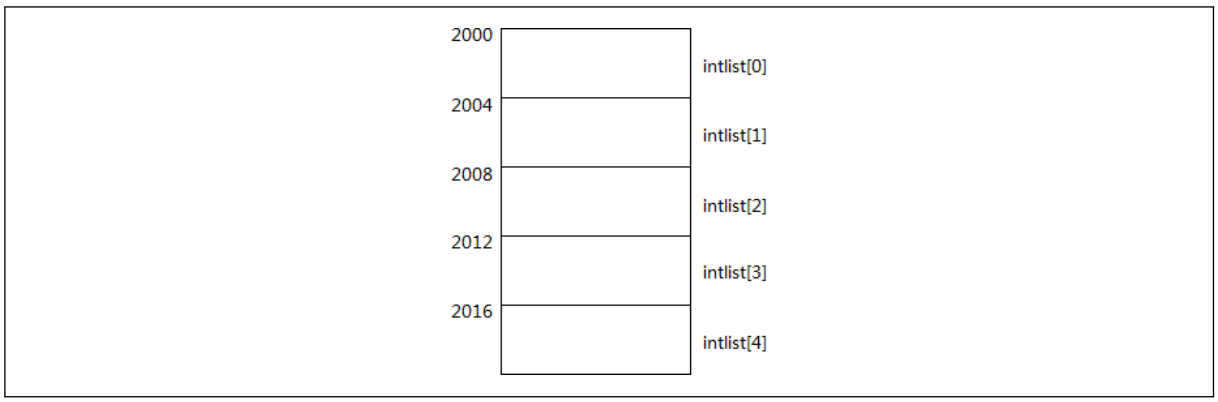
57.2 Array Name

实际上，C 语言最不寻常的特征中最有趣的就是，可以用数组名作为指向该数组中第一个元素的指针的同义词，并且这个事实也说明了数组型实际参数是如何具体工作的。

这个概念很容易用例子解释，下面的声明

```
int intlist[5];
```

为一个含有 5 个整型数的数组分配了空间，分配的空间位于计算机内存的某处。



这里，在数组的内存示意图中左边的地址是假设的，数组的起始地址是 2000，但也可以是其他的任意地址。

intlist 代表一个数组，但也可以直接用作指针值。当它作为指针使用时，intlist 定义为数组中第一个元素的地址。

对于任意数组 arr 来说，下述等式在 C 语言中是永远成立的：

arr 等同于 &arr[0]

给定任何数组名，可以将它的地址直接赋给任何指针变量。当一个数组从一个函数传递到另一个函数时通常可以用到这个等式。典型地，被调用函数可以利用下面的句法来声明这个数组：

```
void SortIntegerArray(int array[], int n);
```

根据指针和数组的关系，另一种定义函数原型的句法如下：

```
void SortIntegerArray(int *array, int n);
```

同样也会达到一样的效果。

- 在第一种声明句法中，若要用 intlist 数组调用 Sort 函数，那么传递给 Sort 的形式参数 array 的值将会是 intlist 中第一个元素的地址。
- 在第二种声明句法中，第一个参数被声明为指针，但效果是相同的。intlist 中第一个元素的地址赋值给了形式参数 array，并且通过指针运算进行处理。

这些声明在机器内部都是相等的，同样的操作在上面两种情况下都可应用。作为一般的规则，声明参数时必须能体现出它们的用途。

- 如果想要将一个参数作为数组使用并从中选择元素，那么应该将该参数声明为数组。
- 如果想要将一个参数作为指针使用，并且对其间接引用，那么应该将该参数声明为指针。

在以数组作为实际参数传递给函数时，数组名始终作为指针。在下面的示例程序中，函数返回整型数组中最大的元素。

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for(i=1; i<n; i++)
    if(a[i]>max)
      max = a[i];
  return max;
}
```

如果使用下面的格式调用 find_largest 函数，那么会把数组 b 的第一个元素赋值给 a，数组本身并没有进行复制。

```
largest = find_largest(b, N);
```

- 在给函数传递普通变量时，变量的值被复制给调用函数，任何对相应的形式参数的改变都不会影响到变量本身。
- 在给函数传递数组时，函数操作的是作为实际参数的数组。
  例如，通过在数组的每个元素中存储零的方式可以对数组进行初始化。

  ```
  void store_zeros(int a[], int n)
  {
    int i;

    for(i=0; i<n; i++)
      a[i] = 0;
  }
  ```

  如果要指明作为形式参数的数组不会被修改，可以在它的声明中加上 const。

  ```
  int find_largest(const int a[], int n)
  {
    ...
  }
  ```

- 可以把作为形式参数的数组声明为指针。
  例如，在形式参数的声明中 *a 和 a[] 是一样的。在这两种情况下（特别是指针算术运算和数组下标运算）可能在 a 上的操作是相同的，而且都可以在函数内给 a 本身赋于新值。

  ```
  int find_largest(int *a, int n)
  {
    ...
  }
  ```

  这里，声明 a 是指针就相当于它是数组，编译器在处理这两类声明时是一样的。
  虽然 C 语言允许数组变量的名字只作为“常量指针”，但是对于对于作为形式参数的数组名没有这类限制。
- 可以给形式参数为数组的函数传递数组的“片断”——即将连续元素的序列传递给函数。
  例如，为了用函数 find_largest 来定位数组 b 中某一序列内的最大元素，可以传递起始元素和序列的长度给函数。
  这里，将通过传递 b[5] 的地址和数字 10 来使用 find_largest 函数检查从 b[5] 开始的 10 个数组元素中的最大值。

  ```
  largest = find_largest(&b[5], 10);
  ```

57.3 Array Initialization

在 C 语言中，当变量不是在作为参数传递时声明而是原始的声明时，数组和指针间最关键的区别就体现出来了。

声明

```
int arr[5]
```

和声明

```
int *arr;
```

最根本的区别在于内存的分配。

- 第一个声明指示编译器为数组预留了五个字的连续内存来存放数组元素。
- 第二个声明指示编译器为指针变量分配空间，这里只分配了一个字的内存空间，其大小只能存放一个机器地址。

  如果试图用未初始化的指针变量 a 作为数组可能会导致致命错误。例如，在未初始指针变量 a 时，赋值语句

  ```
  *a = 0;
  ```

  将在 a 指向的位置存储 0，但是 a 指向的地址未知，所以对程序的影响是无法预料的。

理解这一区别对于程序员来说至关重要。

- 如果声明一个数组或其他普通变量，则需要有工作空间。
- 若声明一个指针变量，那么变量在显式地初始化之前和任何内存空间都无关。

可以通过将数组的基地址赋给指针变量来初始化指针，从而达到将指针作为数组使用的目的。在进行过上述声明后，可以通过下述代码

```
int arr[5];
int *p;
p = arr;
```

从而使指针变量 p 和 arr 指向同样的地址，两者可以互换使用。

使用数组名作为指针后，编译器在标记数组元素时，对待 p[i] 和对应 *(p+i) 是一样的，这是指针算术运算非常正规的用法。

总的来说，将指针指向一个已经存在的数组的地址的技术是有很大局限性的。毕竟，如果已经有一个数组名的话就可以直接使用了，把数组名赋给指针实际上并没有什么好处。

虽然可以把数组名用作指针，但是不能给数组名赋新的值，试图使数组名指向其他地方也是错误的。

```
int a[10];
while(*a != 0)
  a++; /* wrong */
```

在实际应用中应该始终把 a 赋值给指针变量，然后改变指针变量，这并不会产生性能损失。

```
int a[10], *p;
p = a;
while(*p != 0)
  p++;
```

下面的示例重新实现反向输出数列中的数，程序读数时首先把输入的数存入数组，在输入结束时再反向从尾到头单步浏览数组并输出。

```
/* Reverses a series of numbers (pointer version) */
#include <stdio.h>

#define N 10
```

```
main()
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for(p = a; p < a+N; p++)
    scanf("%d", p);

  printf("In veverse order: ");
  for(p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

将指针作为数组使用的实际优势在于,可以把指针初始化为未声明的新内存,从而能在程序运行时建立新的数组,这是一个重要的程序设计技巧。

57.4 Array Parameter

当需要将数组作为函数的形式参数时,要确定使用 *a 还是 a[] 是一件很棘手的问题。

从一种观点来看,*a 无法明确的说明函数是需要多对象的数组还是指向单独对象的指针,所以 a[] 是有优势的。

另一方面,*a 只是说明传递指针而不是复制数组。根据函数是使用指针算术运算还是下标运算来访问数组的元素,其他函数可以在 *a 和 a[] 之间进行切换,因此 *a 比 a[] 更通用。

虽然指针和数组之间有紧密的联系,但是也无法精确地说明它们可以互换。

首先,在 C 语言中,数组型形式参数和指针型形式参数是可以互换的,但是数组变量是不同于指针变量的。

其次,从技术上说,数组的名字不是指针,只是在需要时 C 语言编译器会把数组的名字转换成指针。

考虑下面的测试,当对数组 a 使用 sizeof 运算符时,sizeof(a) 的结果是数组中字节的总数,即每个元素的大小乘以元素的数量。对于指针变量 p,那么 sizeof(p) 的结果则是用来存储指针值所需的字节数量。

```
int a[10], *p;
printf("%d", sizeof(a));
printf("%d", sizeof(p));
```

第 58 章

Pointer Arithmetic

当指针指向数组元素时，C 语言允许对指针进行算术运算——即加法和减法，这种运算引出了一种对数组进行处理的替代方法，从而可以使指针代替数组下标进行操作。

- 当 + 和-的操作数为数字时，结果由简单的数学运算决定。
- 在算术运算符（+ 和-等）应用于指针时，在某些方面其结果与数学运算相似，但在另外一些方面又有所不同。

对指针值应用加减的过程称为指针算术运算（pointer arithmetic）或者地址算术运算，而且数组名和指针的关系进一步简化了指针算术运算，并使得数组和指针都更加通用。

下面的规则对理解指针运算工作机制是很有必要的。

> 指针运算规则：如果指针 p 指向数组 arr 的第一个元素且 k 为整型数，以下的等式总是成立的：
> p + k 定义为 &arr[k]

换句话说，如果在指针值上加上一个整数 k，则结果就是下标为 k 的元素的地址，因此通过在数组指针上进行算术运算来访问数组元素。

标量用于指针算术运算地整数要依赖于指针的类型。例如，如果 p 的类型是 int *，那么 p+j 通常既可以用 2×j 加上 p，也可以用 4×j 加上 p，依据就是 int 型的值要求的是 2 个字节还是 4 个字节。如果 p 的类型是 double *，而 double 类型的值通常都是 8 个字节长，那么 p+j 可能就是 8×j 加上 p。

C 语言支持 3 种（并且仅有 3 种）格式的指针算术运算。

1. 指针加上整数。
2. 指针减去整数。
3. 两个指针相减。

为了解释指针运算规则，假设有一个函数包含如下所示的变量声明，并且在函数帧内给每个变量都分配了空间。

```
double list[3];
double *p;
```

如果帧从地址 1000 开始，内存分配可能如下图所示。

- 对数组变量 list，编译器为它的三个元素都分配了足够的空间以便存放一个 double 型的数据。
- 对指针 p，编译器为其分配了足够空间来存放某个 double 类型的左值的地址。

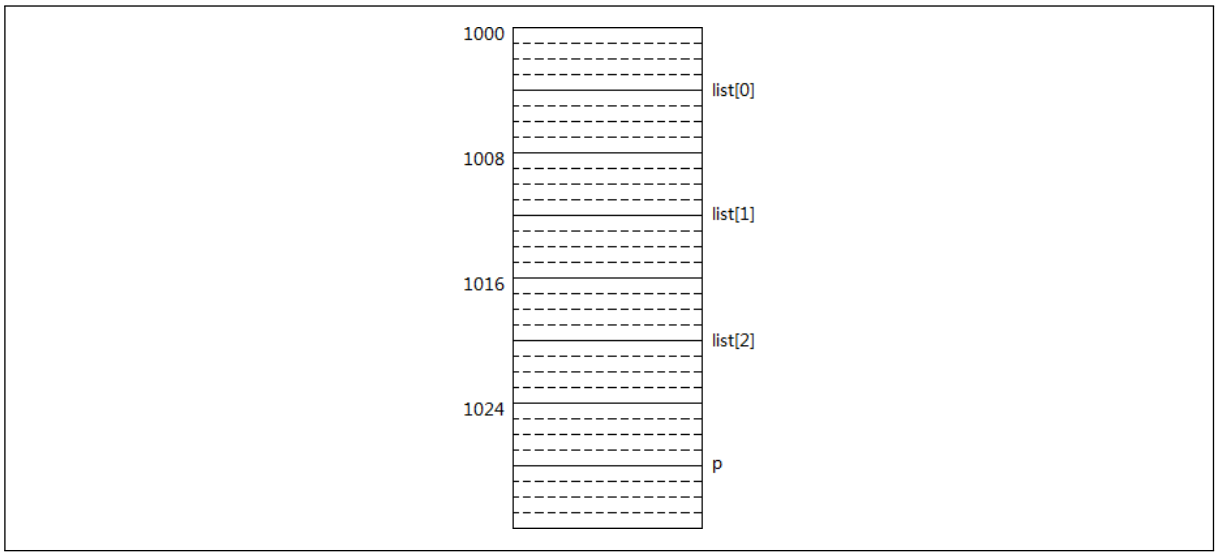
由于还未给这些变量赋值，所以无法确定其初始内容。例如，可以使用以下的赋值语句将任意值存储到每个数组元素中去：

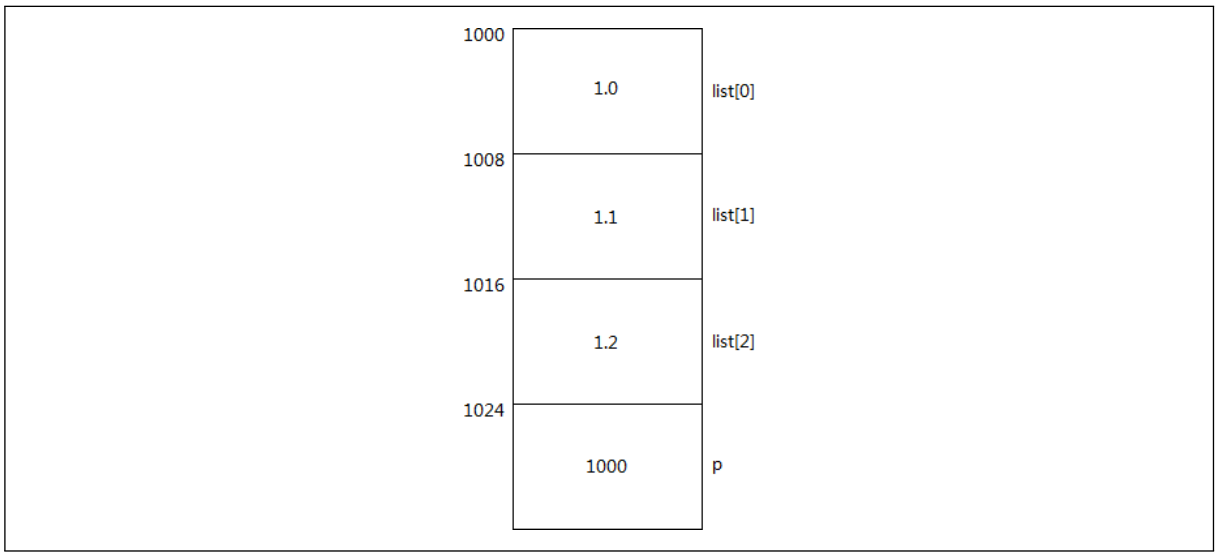
```
list[0] = 1.0;
list[1] = 1.1;
list[2] = 1.2;
```

要初始化指针变量 p，使其指向数组的起始地址，可以执行以下赋值语句：

```
p = &list[0];
int j;
```

完成以上这些赋值后，内存单元中存放的是下图所示的值。





这样，在数组 list 的示意图中，指针 p 指向了数组 list 的起始地址。

如果给指针 p 加整数 k，将产生指向特定元素的指针，其结果就和下标为 k 的数组元素的地址相对应。例如，如果程序包含表达式：

```
p + 2
```

表达式求得的值将是指向 list[2] 的新指针值。

- 如果指针 p 指向数组 a，那么 a+i 同于 &a[i]（两者都是表示指向数组 a 中元素 i 的指针），而且 *(a+i) 就等于 a[i]（两者都表示元素 i 本身）。

```
int a[10];
*a = 7; /* stores 7 into a[0] */
*(a+1) = 12; /* stores 12 into a[1] */
```

换句话说，可以把数组的下标看成是指针算术运算的格式。

- 如果指针 p 指向数组元素 a[i]，那么 p+j 将指向 a[i+j]（当然前提是 a[i+j] 必须存在）。

在前面的图中，p 指向地址 1000，p+2 指向数组中该元素后出现的第 2 个元素 list[0+2] 的地址——即地址 1016。

需要注意的是，指针加法和传统加法是不同的，因为指针运算必须考虑到基本类型的大小。这里，由于每个 double 型需要 8 个字节，所以指针值每增加 1 个单位，内部数值必须增加 8。

C 语言编译器解释从指针减去整数时也采用类似的方法，如果 p 指向数组元素 list[i]，那么 p-j 指向 list[i-j]。比如下面的表达式：

```
p –k
```

这里，p 是一个指针，k 是一个整数，计算的是数组中 p 的当前值指向的地址前的第 k 个元素的地址。

如果 p 指向 list[1] 的地址，则与 p - 1 及 p + 1 相应的地址分别为 list[0] 和 list[2] 的地址。

```
p = &list[1];
p –= 1; /* &list[0] */
p += 1; /* &list[2] */
```

从程序员的角度来看，重要的是认识到指针运算被定义为自动考虑基本类型的大小，因此对于给定的任何指针 p 和整数 k，则表达式

```
p + k
```

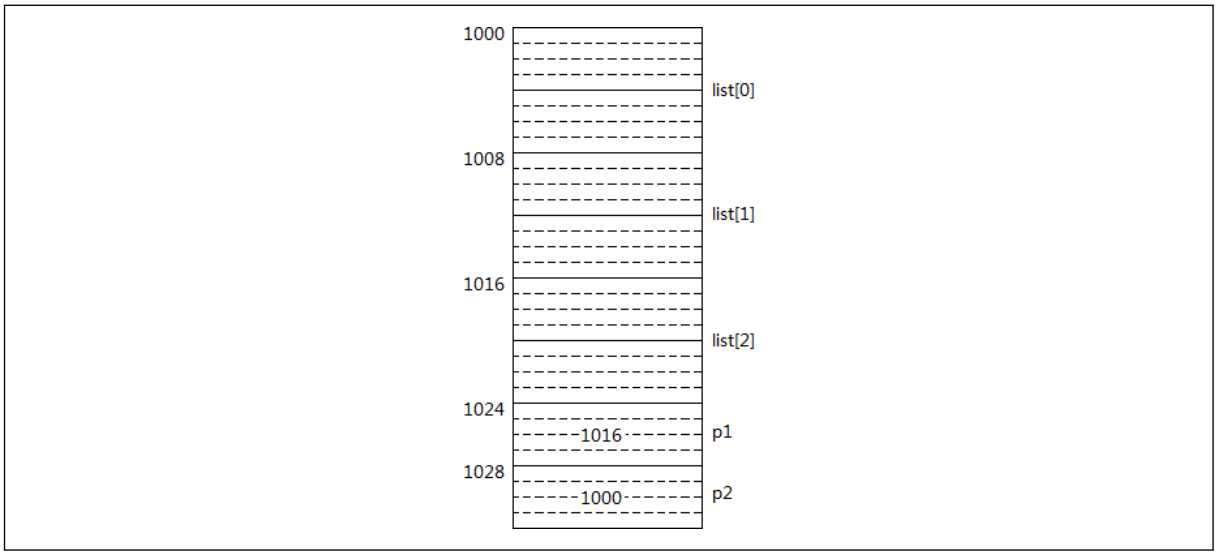
意味着不管每个元素需要多少内存，结果总是数组中 p 目前指向的地址后第 k 个元素的指针。数组元素所需的内存大小只在理解计算机内部如何进行计算操作时才需要考虑。

对指针而言，算术运算符 *、/和% 对指针来说是没有意义的，不能和指针操作数一起使用。

在指针运算中 + 和-的使用也是有限的，可以给一个指针加上或减去一个整数偏移量，但不能将两个指针相加，其他唯一可用于指针的算术操作是将两个指针相减。

当两个指针相减时，结果为指针之间的距离，因而可以以此来计算数组中元素的个数。例如，如果 p1 和 p2 是指向同一数组的指针，那么表达式 p1 – p2 可以返回 p2 和 p1 当前值之间的数组元素的个数。

如果 p 指向 list[i] 且 q 指向 list[j]，那么 p-q 就等于 i-j。这里，假设指针 p1 指向 list[2]，p2 指向 list[0]，那么表达式 p1 – p2 的值为 2，因为当前两指针值之间正好有 2 个元素。



理解这个定义的另一种方法是将指针相减所得的值通过赋值语句赋给一个整型变量 k：

```
k = p1 –p2;
```

它将满足以下关系：

```
p1 == p2 + k;
```

只有在指针指向数组元素时，指针上的算术运算才会获得有意义的结果，而且仅当在两个指针指向同一个数组时，指针相减才有意义。

根据 C 语言标准的规定，对于必非指向数组的指针，指针的算术运算是“未定义的”，也就是说不能保证对未指向数组的指针进行算术运算的结果。

58.1 Increase and Decrease Operators

自增和自减运算符(即 ++ 和--运算符)除了可以用于操作数,对它们所操作的左值加 1 或者减 1 之外,在实际情况中也可以执行更加复杂的操作。

- 首先,自增和自减运算符都可以用两种方式写,其中一种写法是将运算符写在操作数后,即

```
x++
```

- 或者将运算符写在操作数前,即

```
++x
```

第一种形式,即运算符在操作数之后,称为后缀(postfix)式,而第二种形式称为前缀(prefix)式。

如果所要做的只是执行单独 ++ 运算,就像把它作为一个单独的语句,或作为 for 循环中的自增运算符那样,前缀运算符和后缀运算符的效果是完全一样的。

只有在把这两个运算符作为更长的表达式的一部分时,区别才显现出来。和所有的运算符一样,++ 运算符会返回一个值,但为何值取决于运算符相对于操作数的位置。

- x++ 先计算 x 的值再将它加 1。返回给外部表达式的值是加 1 前的原始值。
- ++x 先将 x 的值加 1 再使用新值作为整个 ++ 操作所得的值。

--运算符也类似,但为减 1 而非加 1,假如执行以下程序:

```
main()
{
  int x, y;

  x = 5;
  y = ++x;
  printf( "x = %d, y = %d\n" , x, y);
}
```

结果为:

```
x = 6, y = 6
```

如果程序改写为:

```
main()
{
  int x, y;

  x = 5;
  y = x ++;
  printf( "x = %d, y = %d\n" , x, y);
}
```

此时结果为:

```
x = 6, y = 5
```

语句

```
y = x ++;
```

确实给 x 的值增加了 1,即它的值为 6,但是赋给 y 的值是在自增操作之前的值,所以 y 值为 5。

注意,++ 和--运算符并不是必须的。此外在其他许多情况下,程序将这些运算符嵌在一个大的表达式中会比使用简单方法更显示出优势。但是在另一方面,++ 和--已成为习惯用语而被程序员频繁使用,C 语言程序员已牢牢地确立使用 ++ 和--的传统。

因为这些运算符很通用,所以必须先理解它们才能读懂一些现有的代码。举例来说,假设想要把一个数组 arr 的前 n 个元素置为 0,最直接的方法是使用 for 语句:

```
for(i = 0; i < n; i ++) arr[i] = 0;
```

但有一些程序员可能注意到自增运算可以和选择操作结合使用,于是将其写做:

```
for(i = 0; i < n;) arr[i ++] = 0;
```

这个例子要求的是 i ++ 的形式而非 ++ i 的形式想要让 i 自增,但同时也想选择数组中编号为 0、1、2 以及以下的元素,所以在选择表达式中需要的是自增前的 i 值。

在某些机器(特别是设计 C 语言的 PDP-11 机器)上,采用指针算术运算的算法的确比数组下标更有效一些。即使如此,由于后一种编码方法违背了 for 循环的精神,由此而带来的效率并不足以进行这种修改。

使用 for 循环头能够很确切地告诉读者每个循环周期中下标变量的行为,因此在现代编译器和影件架构下,数组下标方法往往是更好的。

58.2 Pointer Ambiguity

在引用数组 a[i] 的元素时,i[a] 和 a[i] 的结果是一样的。

对于编译器而言,i[a] 等同于 *(i+a),指针加法也是可以交换的,因此 *(i+a) 等同于 *(a+i),而 *(a+i) 也就是 a[i]。

另外,将 ++ 运算符作为表达式的一部分来使用也可能会造成歧义。举例来说,假如想把 arr 数组中的每一个元素值都设为其下标值,即 arr[0] 为 0,arr[1] 为 1,以此类推。

下面的示例代码用来表示 arr[i] 被置为 i,随后 i 自增并为下个循环周期做准备,但这一解释在 C 语言中并不能得到保证。

```
for(i = 0; i < n; i ++) arr[i] = i ++; /*该语句是有歧义的*/
```

在一个含二元运算符的表达式中,C 语言程序会使用任何一种次序计算运算符对应的操作数,通常选择那种对机器来说最方便的顺序。

上述的示例代码设想程序会首先计算 arr[i] 的地址,再计算 i++,最后才将 i++ 的值(i 的旧值)赋给已计算出的地址。

但是,程序很有可能先计算 i++ 的值再计算 arr[i] 的值。作为表达式 i++ 的结果返回的值仍是 i 的旧值,但到计算机设法算出 arr[i] 所指的值时,i 已经自增过了,所以程序会给编号为 i 的数组元素赋值为 0。

为了避免产生歧义,需要保证任何一个和 ++ 或--一起使用的变量都不再在表达式的其他地方出现,前面 for 语句的主体违反了这一原则,因为 i 同时出现在赋值的左边和右边。

避免这一歧义的最好方法是限制自增和自减运算符只以单独形式出现,并避免在表达式内使用它们的值。

> 常见错误:使用运算符 ++ 和-时,注意必须避免写出有歧义的表达式,否则在不同的机器上会计算出不同的结果。一般规则是,使用这些运算符进行自增或自减的变量不应在同一个表达式中出现两次。

58.3 Pointer Comparation

C 语言允许用关系运算符(<、<=、>、>=)和判等运算符(== 和!=)进行指针比较,但只有在两个指针指向同一数组时,用关系运算符进行的指针比较才有意义,而且比较的结果依赖于数组中两个元素的相对位置。

例如,在给下面的指针赋值后,p<=q 的值是 0,而 p>=q 的值是 1。

```
p = &list[4];
q = &list[1];
```

58.4 Incrementing and Decrementing Pointers

现在可以引入 C 语言中最常用的一种处理数组元素的结构——即指针的自增和自减表达式。

在处理数组元素的语句中经常会组合 *(间接寻址)运算符和自增/自减运算符。在下面的示例中,首先把值存入数组元素中,然后推进到下一个元素。

```
arr[i++] = j;
```

如果 p 指向数组元素,那么相应的指针实现语句为:

```
*p++ = j;
```

如果要确定指针自增表达式的意义,首先需要解决的问题就是操作的顺序是什么。按照语言的优先级规则和结合性规则,*(间接寻址)运算符和 ++(自增)运算符在争夺操作数 p 时,该表达式可能会等同于

```
(*p) ++
```

或者是

```
*(p ++)
```

当这种情况发生时,C 语言中的一元运算符按照从右至左的顺序进行运算,++ 运算优先于 * 运算,第二种解释是正确的。

这里使用后缀 ++,所以 p 只有在表达式计算出来后才可以自增,因此 *(p++) 的值将是 *p——即指针 p 当前所指向的对象。

当然,*p++ 不是唯一合法的 * 和 ++ 的组合。例如,可以通过 (*p)++ 来返回指针 p 指向的对象的值,然后对象进行自增(p 本身不会变化)。

表达式	含义
*p++ 或 *(p++)	自增前表达式的值是 *p,然后自增 p
(*p)++	自增前表达式的值是 *p,然后自增 *p
*++p 或 *(++p)	先自增 p,自增后表达式的值是 *p
++*p 或 ++(*p)	先自增 *p,自增后表达式的值是 *p

下面的示例程序用来对数组 arr 的元素进行求和。

```
#define N 10

int arr[N], sum, *p;
sum = 0;
for(p=&arr[0]; p<&arr[N]; p++)
  sum += *p;
```

首先,指针变量 p 指向 arr[0],每次循环结束时 p 自增,以此类推。在 p 执行到数组 arr 的最后一个元素后循环终止。

另外,数组名可以用作指针的事实使得可以实现单步操作数组的循环,这里可以用 arr 替换 &arr[0],同时用 a+N 替换 &a[N]。

```
#define N 10

int arr[N], sum, *p;
sum = 0;
```

```
for(p=a; p<a+N; p++)
  sum += *p;
```

如以前所见，后缀式 ++ 运算符让 p 值先自增，随后再返回自增操作前的 p 值。由于 p 是一个指针，必须根据指针运算定义自增操作，所以在计算 p + 1 时，知道结果值应该指向数组中的下一个元素。

上述示例也可以写成

```
#define N 10

int arr[N], sum, *p;
sum = 0;
p = &arr[0];
while(p < &a[N])
  sum += *p++;
```

这里，在第一次循环中，后缀式 *p++ 先计算 *p，然后才对 *p 进行自增从而指向 arr[1]，在接下来的第二次循环中进行类似的操作，最终完成对数组元素求和。

针对 for 语句的条件判断式中 &arr[N]，标准 C 规定即使元素 arr[N] 不存在（数组 arr 的下标从 0 到 N-1），但是对它使用取地址运算符是合法的。

for 循环不会去尝试检查 arr[N] 的值，所以在上述示例代码中使用 arr[N] 是非常安全的，实际上当指针 p 等于 &arr[0]、&a[1]、……、&arr[N-1] 时可以执行循环体，但是当 p 等于 &a[N] 时，循环终止。

如果 p 原来指向 arr[0]，那么自增操作就会使其指向 arr[1]，于是表达式 *p ++ 表示下面的含义：

> 间接引用指针 p，使其当前指向的对象作为左值返回。其负面影响是，p 值进行了自增，所以如果原左值是数组的一个元素的话，新的 p 值指向数组中的下一个元素。但是为了理解该运算符的有用之处，必须更细致地考虑指针和数组之间的关系。

* 运算符和--运算符的混合方法类似于 * 和 ++ 的组合。在实现栈的示例中可以使用指针变量来替换整型变量 top，原始版本的栈依赖整型变量 top 来跟踪 contents 数组中“栈顶”的位置。

在初始化栈时，指针变量初始指向 contents 数组的第 0 个元素，并重新实现了新的 push 函数和 pop 函数。

```
int *top_ptr = &contents[0];

void push(int i)
{
  if(is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}
int pop(void)
{
  if(is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

这里，为了使 pop 函数在取回 top_ptr 指向的值之前对 top_ptr 进行自减，需要写成 `*--top_ptr`，而不是 `*top_ptr--`。

虽然使用下标代替可以很容易地写出不使用指针的循环，但是实际参数常用于支持指针的算术运算，这样可以节约执行时间。随着编译器效率的提高，与依赖指针的实现相比，已经可以在不牺牲效率的前提下，依赖下标的实现实际上会产生更好的循环代码。

第 59 章

Multidimensional Array

C 语言始终按行的顺序存储二维数组，即先是第 0 行的元素，接着是第 1 行的元素，以此类推。

指针不仅可以指向一维数组的元素，也可以指向多维数组的元素。如果使指针指向二维数组中的第一个元素(即第 0 行第 0 列的元素)，就可以通过重复自增指针的方式访问数组中每一个元素。

下面的示例中，首先使用数组下标把二维数组的所有元素初始化为 0。

```
int a[NUM_ROWS][NUM_COLS];
int row, col;

for(row=0; row<NUM_ROWS; row++)
  for(col=0; col<NUM_COLS; col++)
    a[row][col] = 0;
```

如果把数组 a 看作是一维数组，那么可以用单独一个 for 循环来实现上述程序。

```
int *p;
for(p=&a[0][0]; p<=&a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

循环从 p 指向的 a[0][0] 开始，对 p 的连续自增可以使指针指向数组 a 中的所有元素。其中，当 p 自增到 a[0][NUM_COLS-1] 时(即 0 行的最后一个元素)，此时再次对 p 自增将使得它指向 a[1][0]，也就是 1 行的第一个元素。处理过程持续到 p 自增到 a[NUM_ROWS-1][NUM_COLS-1] 为止，也就是到达数组中的最后一个元素。

上述使用指针遍历二维数组的做法在 C 语言中是合法的，

59.1 Array Rows

为了访问二维数组中某一行的元素，首先初始化指针指向数组中某一行的元素 0。

```
p = &a[i][0];
```

对于二维数组来说，表达式 a[i] 代表指向 i 行中第一个元素的指针，那么上述的语句可以简写成：

```
p = a[i];
```

根据数组下标与指针算术运算的关系，对于任意数组 a 有，表达式 a[i] 等价于 *(a+i)，因此 & 和 * 运算符可以取消。

&a[i][0] 等同于 &(*(a[i]+0))，也就等同于 &*a[i]，即 a[i]。下面把这种简化公式用于 for 循环中来初始化二维数组 a 的 i 行。

```
int a[NUM_ROWS][NUM_COLS], *p, i;
...
for(p=a[i]; p<a[i]+NUM_COLS; p++)
  *p = 0;
```

a[i] 是指向数组 a 的 i 行的指针，可以把 a[i] 传递给期望一维数组作为实际参数的函数。换句话说，设计使用一维数组的函数也将可以用二维数组中的一行工作，从而使函数更加通用。

对于用于找到一维数组中最大元素的函数 find_largest，改造后仍然可以简单地用于确定二维数组一行中的最大元素。

```
largest = find_largest(a[i], NUM_COLS);
```

59.2 Array Columns

使用相似的方法处理多维数组的列中的元素也是可行的。

数组是按行存储的，下面的循环用来对数组 a 中 i 列的元素进行初始化。

```
int a[NUM_ROWS][NUM_COLS], i (*p)[NUM_COLS];

for(p = a; p <= &a[NUM_ROWS-1]; p ++)
{
  (*p)[i] = 0;
}
```

首先声明 p 是指向长度为 NUM_COLS 的数组的指针，且此数组的元素为整型的。

在表达式 (*p)[NUM_COLS] 中要求 *p 周围有圆括号。如果没有圆括号，那么编译器将会把 p 作为指针的数组而不是指向数组的指针来处理。

表达式 p++ 在下一行开始时提前处理 p，在表达式 (*p)[i] 中的 *p 表示数组 a 的完整一行，所以 (*p)[i] 选择了此行 i 列的元素。

在 (*p)[i] 中的圆括号是必需的，因为编译器将把 (*p)[i] 解释为 *(p[i])。

59.3 Array Name

在 C 语言中，可以忽略数组维数而采用任意数组的名字作为指针。

考虑下面的数组，虽然可以使用 a 作为指针指向数组元素 a[0]，但是不能使用 b 指向数组元素 b[0][0]，而只能是说 b 是指向 b[0] 的指针。

```
int a[10], b[10][10];
```

C 语言认为二维数组名是指向的每一维都是一维数组，这样上述 a 可以用作是 int * 的指针，而 b 用作指针时则是具有 int ** 类型的指针(指向整型指针的指针)。

例如，为了使用 find_largest 函数找到二维数组的最大元素，可以把 a 考虑成一维数组，a(数组的地址)将作为 find_largest 函数的第一个实际参数进行传递，然后把 NUM_ROWS*NUM_COLS(数组 a 的总元素数)作为第二个实际参数进行传递。

```
int a[NUM_ROWS][NUM_COLS];
largest = find_largest(a, NUM_ROWS*NUM_COLS); /* wrong */
```

上述语句不能编译的原因是 a 的类型是 int **，而 find_largest 函数期望的实际参数是 int *，因此正确的调用应该是：

```
largest = find_largest(a[0], NUM_ROWS*NUM_COLS);
```

a[0] 指向第 0 行的第 0 个元素，而且它的类型为 int *，上述调用将正常执行。

这里，a 和 a[0] 都指向同一位置——即元素 a[0][0]，但是 a 的类型为 int **(这不是 find_largest 函数所希望的)，而 a[0] 的类型为 int *，满足函数和编译器的要求。

第 60 章

Storage Allocation

当声明一个全局变量时，编译器给在整个程序中持续使用的变量分配内存空间。这种分配方式称为静态分配(static allocation)，因为变量分配到了内存的固定位置。

当在函数中声明一个局部变量时，给该变量分配的空间在系统栈中。调用函数时给变量分配内存空间，函数返回时释放该空间。这种分配方式称为自动分配(automatic allocation)。

还有第三种内存分配方式，能在需要新内存的时候得到内存，不需要内存时就显式释放这部分内存，这种在程序运行时获取新内存空间的过程称为动态分配(dynamic allocation)。

当程序载入内存时，通常只占用可用空间的一部分。在大多数系统中，当程序需要更多内存时，可以将一些未使用的内存空间分配给程序。

在计算机科学中，程序可用的未分配的内存资源称为堆(heap)，可以从堆中动态地分配新内存。例如，如果程序运行时需要一个新的数组空间，可以预留部分未分配的内存，将其余的内存留到后面分配。

作为库接口 stdlib.h 的一部分，ANSI C 语言的环境提供了一些从堆中分配新的内存的函数，其中最重要的一个函数为 malloc。

具体来说，malloc 函数能分配一块固定大小的内存，待分配的内存块的大小以字节为单位给出(一个字节是足够存放一个字符值的内存单元)。例如，如果要分配 10 个字节的内存，可调用 malloc(10)来返回一个指针，指向一个 10 字节大小的内存块。

为了使用新分配的空间，必须将 malloc 的结果存放在一个指针变量内，以后就可以像使用数组一样使用该指针变量了。

60.1 void *

首先讨论一个很可能会十分棘手的问题。

在 C 语言中，指针是具有类型的。如果通过

```
int *p;
```

声明一个变量 ip，该变量在概念上是一个指向整型的指针。而声明

```
char *cp;
```

则引入了一个指向字符的指针变量 cp。

在 ANSI C 中，指向不同类型的指针是不同的，如果想要把其他类型赋给这些指针变量，编译器会发出警告信息。

> 指针具有类型的概念引发了一个很有趣的问题，即 malloc 函数返回的是什么类型的指针?

malloc 函数用于为调用函数需要的任何类型的值分配新的内存空间，所以必须返回一个未确定类型的“通用”指针。

在 C 语言中，通用指针类型是一个指向空类型 void 的指针，void 也用于指示无返回值或参数列表为空的函数。

如果声明一个指向 void 类型的指针，例如

```
void *vp;
```

就可以将任何类型的指针值存入该变量，但不允许用 * 运算符间接引用 vp。编译器不知道 vp 的基本类型是什么，所以没有办法谈论 vp 指向的值。

虽然如此，void * 类型仍是有用的。作为表示与所有其他指针类型兼容的通用指针类型，void * 类型允许函数(特别是 malloc 函数)返回以后由调用函数创建实际类型的通用指针。

malloc 函数返回类型为 void * 的指针值，表明其原型为：

```
void *malloc(int nBytes);
```

注意，这个函数原型的结果类型与指针变量的声明方法非常相似。* 表示结果是和函数名而非基本类型相联系的指针，尽管在概念上的结果是与基本类型而非函数名联系的。除非习惯 C 语言中返回指针值的函数句法，否则是很容易混淆的。

malloc 的参数类型实际上是在头文件 stddef.h 中定义的 size_t 类型。在一些机器上，int 类型的大小可能不足以表示内存块的大小。不论在哪种情况下，参数在概念上是整数。

ANSI C 能在指向 void 的指针类型和指向基本类型的指针类型间自动进行转换。例如，如果声明字符指针 cp 为

```
char *cp;
```

就可以用语句

```
cp = malloc(10);
```

将 malloc 的结果直接赋给 cp。

一般情况下，在赋值之前会使用下列强制类型转换将 malloc 返回的结果转换成指向字符的指针。这样做的一部分原因是因为历史因素，另一部分原因是这样做能使指针类型间的转换更清楚。

```
cp = (char *)malloc(10);
```

不管有没有明确使用强制类型转换，这条语句均可分配 10 个字节的新内存空间，并将第一个字节的地址存放在 cp 中。

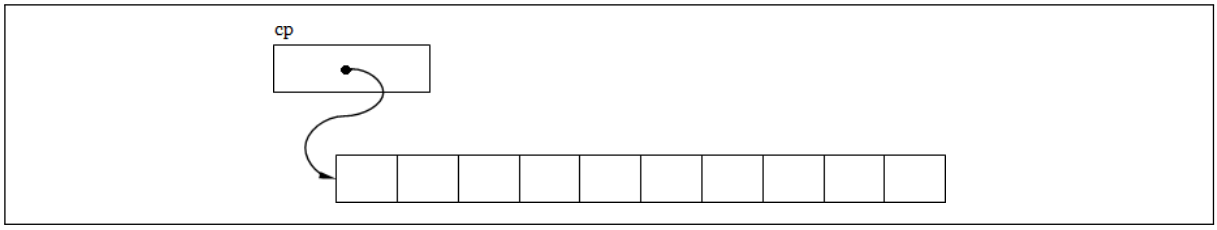
> 常见错误：一定要分清过程原型“void f(...);”和函数原型“void *f(...);”，后者声明了一个返回通用指针的函数。

60.2 Dynamic Array

从概念上讲，赋值语句

```
cp = (char *)malloc(10);
```

建立了如下内存配置：



这里，变量 cp 指向已经在堆内分配的连续 10 个字节。因为指针和数组在 C 语言中能自由地相互转换，所以变量就好像声明为一个含 10 个字符的数组一样。

分配在堆上并用指针变量引用的数组称为动态数组(dynamic array),可以使用动态分配来为在程序运行时,而不是在编译时确定大小的动态数组分配内存。

一般来说,分配一个动态数组包含以下步骤:

1. 声明一个指针变量,用以保存数组基地址。
2. 调用 malloc 函数为数组中的元素分配内存。
   由于不同的数据类型要求不同大小的内存空间,所以 malloc 调用必须分配的字节大小等于数组元素数乘以每个元素字节大小的内存空间。
3. 将 malloc 的结果赋给指针变量。

例如,要给一个含 10 个元素的整型数组分配空间,然后将该内存赋给变量 arr,必须先用

```
int *arr;
```

声明 arr,随后使用

```
arr = malloc(10 * sizeof(int));
```

来分配空间。

声明数组和动态数组的主要区别如下:

1. 与一个已声明的数组相关的内存是作为声明过程的一部分自动分配的。当声明数组的函数帧建立时,数组中所有的元素都作为帧的一部分进行分配。在动态数组的情况下,实际内存在调用 malloc 函数前是不会被分配的。
2. 在程序中,已声明的数组大小必须是不变的。而由于动态数组的内存来自于堆,所以它们的大小可以是任意的。而且,可以根据数据量推断数组的大小。如果知道需要一个含 N 个元素的数组,可以保留大小正好的内存空间。

由于计算机内存系统的大小是有限的,堆的空间终会用完。此时 malloc 返回指针 NULL 表示分配所需内存块的工作失败。

谨慎的程序员应该在每次调用 malloc 时都检查失败的可能性,因此分配一个动态数组后,需要写如下语句:

```
arr = malloc(10 * sizeof(int));
if(arr == NULL) Error("No memory available");
```

动态分配往往会频繁用于多种程序,因此错误检查尽管重要,却也是极其单调乏味的,而且常常因为出错消息使代码变得混乱不堪,搞乱算法结构。

实际上,当空间用完时就无计可施了,程序显示出错消息并且停止运行通常是唯一可行的方法,所以调用 malloc 并将其嵌入一个包括内存不够测试的新抽象层中是很有用的。

genlib.h 接口输出函数 GetBlock,该函数结合了 malloc 并对 NULL 结果进行测试。如果 GetBlock 函数检测到内存不够的情况,就会调用 Error 函数,所以如果以上两行代码用

```
arr = GetBlock(10 * sizeof(int));
```

代替的话,GetBlock 函数的执行就结合了分配内存和检测错误的操作,因此 GetBlock 使内存分配的操作更清晰,程序更易懂。但在概念上,这两个函数的功能是相同的。

为了进一步简化分配动态数组的过程,genlib.h 库还定义了函数 NewArray。NewArray 取元素数和其基本类型后返回一个指针,并指向特定大小的动态数组。

如果要分配一个含 50 个字符串的动态数组,应使用如下的语句:

```
arr = NewArray(50, string);
```

60.3 Memory Allocation

60.4 Memory Deallocation

保证不会发生内存不够的一种方法是一旦使用完已分配的空间就立刻释放它。

标准 ANSI 库提供了函数 free，用于当不再需要这些内存时，由 free 函数释放以前由 malloc 分配出去的堆内存并归还给堆。例如，如果能肯定已经不再使用分配给 arr 的内存，就可以通过调用

```
free(arr);
```

来释放该空间。

由于在 genlib.h 引入了 GetBlock 执行动态分配，所以为分配和释放建立一个包括兼容的命名机制的统一抽象体系尤为重要。为此，genlib.h 接口也包括一个 FreeBlock 函数，其操作方法和 free 完全相同。

但事实证明，知道何时释放一块内存并不那么容易，根本问题在于分配和释放内存的操作分别属于接口两边的实现及客户。

实现知道何时该分配内存，返回指针给客户，但它并不知道何时客户结束使用已分配的对象，所以释放内存是客户的责任，虽然客户可能并不十分了解对象的结构。

对现在的大多数计算机的内存来说，我们可以随意分配所需内存而不需要考虑释放内存的问题。这种策略几乎对所有运行时间不长的程序还是有效的。

内存有限的问题只有在设计一个需要运行很长时间的应用（比如所有其他系统所依靠的操作系统）时，才变得有意义。在这些应用中，不需要内存时释放它们是很重要的。

某些语言支持那些能主动检查正在使用的内存，并释放不再使用的内存的动态分配系统，该策略称为碎片收集（garbage collection）。

C 语言中也有碎片收集分配器，即使在长时间运行的程序中，也可以忽略释放内存的问题，因为可以依靠碎片收集器自动执行释放内存的操作。

在大部分情况下，都应该假定无论何时都可以给应用分配内存，这样大大简化了程序，并能使我们专注于算法细节。

60.5 NULL pointer

更进一步，通过调用 printf 函数并在格式串中采用%p 转换说明可以返回指针的值。

C 语言提供了一个特殊指针值 NULL 来表示指针当前不指向任何数据。在下面的示例中返回的是未初始化的指针的值（nil）[1]。

```
#include <stdio.h>
main()
{
  int *p;
  printf("%p", p);
}
$ gcc -c test.c
$ ./a.out
(nil)
```

[1] nil 表示无值，任何变量在没有被赋值之前的值都为 nil。对于真假判断，只有 nil 与 false 表示假，其余均为真。

另外，在 Objective-C、Ruby 和 Lua 中的关键字 NULL 是一种类型，它只有一个值 nil，它的主要功能是用于区别其他任何值。一个全局变量在第一次赋值前的默认值就是 nil，将 nil 赋予一个全局变量等同于删除它，Lua 将 nil 用于表示一种“无效值 (non-value)”的情况——即没有任何有效值的情况。

在 C 语言编程中，如果给指针赋予一个初始值，那么会根据实际情况返回指针当前指向的内存地址。

```
#include <stdio.h>
main()
{
  int i, *p = &i;
  printf("%p", p);
}
$ gcc -c test.c
$ ./a.out
0x7fff1af2aea4
$ ./a.out
0x7fff3157bd44
$ ./a.out
0x7fffb698d344
```

在指针的应用中，可以在一个指针变量中存储一个特殊值表示变量实际上并不指向任何有效数据是很有用的。

为了达到这个目的，C 语言定义了一个特殊的常量 NULL[2]。常量 NULL 可以赋给任何指针变量，在机器内部表示为地址值 0。

```
#include <stdio.h>
#include <stdlib.h>

main()
{
  int i, *p, *q;
  p = NULL;
  q = &i;
  printf("%p\n", p);
  printf("%p\n", q);
}
$ gcc -c test.c
$ ./a.out
(nil)
0x7fffc850a82c
```

事实上，常量 NULL 是在 stdlib.h 头文件中定义的，当程序包含 stdio.h 头文件时就会自动包含这个头文件，从而可以把 NULL 当成是 C 语言内置的常量。

```
The <stdlib.h> header shall define the following macros:

EXIT_FAILURE
Unsuccessful termination for exit(); evaluates to a non-zero value.

EXIT_SUCCESS
Successful termination for exit(); evaluates to 0.

NULL Null pointer.

{RAND_MAX}
Maximum value returned by rand(); at least 32767.
```

[2]C/C++ 里的 NULL 是一个值为 0 的宏定义，其结果 nil 表示无值。

```
{MB_CUR_MAX}
Integer expression whose value is the maximum number of bytes in a character specified by
    the current locale.
```

如果一个指针变量的值为 NULL，很重要的一点就是不要用 * 运算符来间接引用此变量，NULL 值的目的是表示指针不指向有效数据，所以想要找出和 NULL 指针有关的数据是没有意义的。

遗憾的是，大多数编译器没有查找这一错误的程序。如果间接引用 NULL，通常计算机会寻找存放在内存地址 0 中的值。如果正巧使用 NULL 指针赋值来改变那个值的话，程序会崩溃，也不会有任何关于问题如何产生的线索。未初始化的指针变量也会发生同样的问题。

> 常见错误：小心不要间接引用那些没有初始化的指针或值为 NULL 的指针，这样做会引用不属于当前程序的内存空间，很可能会使程序崩溃。

Part X

String

第 61 章

Enumeration Type

61.1 Overview

绝大多数的程序设计都是用数字作为基本的数据类型。正如 Duster 在 Mathemagician 中所坚持的,数字确实很重要,但在世界上还有许多其他类型的数据。

目前,计算机较少用于数值型的数据,而较多用于文本数据(text data),即由键盘和屏幕上出现的独立的字符组成的信息。现代计算机处理文本数据的能力推动了字处理系统、在线参考库、电子邮件以及其他许多非常有用的应用的发展。

为了发挥文本数据的所有能力,需要知道如何以更复杂的方式操作字符串。因为字符串是由独立的字符组成,因此了解字符是如何工作的,以及它们如何在计算机内部表示是非常重要的。然而,在研究这些类型的细节之前,需要从更一般的观点了解数据的表示。

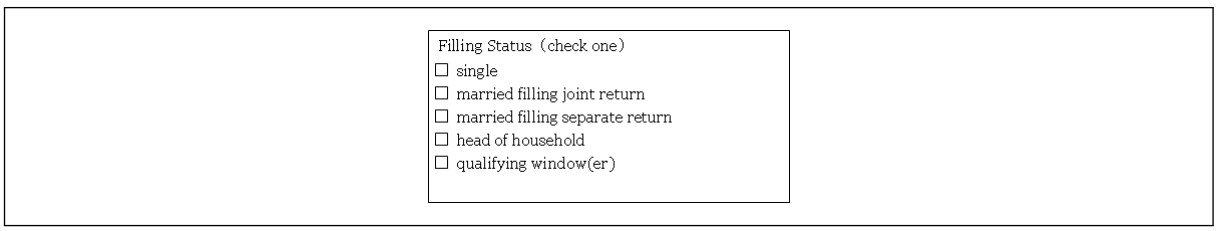
61.2 Principle

随着计算机技术的发展,越来越多的信息要以电子化的方式存储。为了将信息存储在计算机中,需要将数据表示为机器能使用的形式。特定内容的表示取决于它的数据类型。

整数在计算机内有一种表示方式,而浮点数有另一种表示方式。尽管我们不了解这些数据究竟是怎样表示的,但是这取决于计算机能在它的内部存储器中存储数据这个事实。

数值型数据中,整型数、浮点数各自都有自己的表示方式。除了数字之外,计算机也能表示非数值型的数据,因此计算机必须也能表示并处理这些非数值型的数据,特定内容的表示取决于它的数据类型。

为了了解非数值型数据的特性,考虑一下你自己在一年中提供给学校和机关的信息。以生活在美国的公民在一年中提供给学校和机关的信息为例,来讨论非数值型数据的特性,比如提供给国内税收服务部的年退税数据中的信息,其中有些是数值型的,包括工资、扣除额、税、代扣所得税等,另一些是由文本数据组成的,比如名字、地址、职务等。还有一些项目不能简单地归入这两类,例如问题之一是:



此时,你的响应既不是数值型,也不是文本数据。描述这种数据类型的最好的方法是称它为报税身份数据,这是一种全新的数据类型,它的域值由五个值组成:single、married filing joint return、married filing separate return、head of household 和 qualifying widow(er)。

而且在其他的应用中,也有类似结构的其它数据类型。例如,其他表格可能调查有关性别、民族或学生状态的信息。在每种情况下,都可以从一组由不同概念类型组成的可能值的表中选择响应。

列出类型值域中所有元素的处理方式称为枚举(enumeration)。通过列出它的所有元素来定义的类型称为枚举类型(enumeration type)。

在结构上字符类型与枚举类型很相似,理解枚举类型是如何工作的有助于理解字符的工作原理。

61.3 Implementation

值不是数字的类型在计算机内部通常通过对该类型的值域中的元素编码,然后用这些编码作为原先值得代码来表示。

通过对元素进行计数而定义的类型称为枚举类型。枚举类型作为一个抽象的概念,要理解它们如何被用于程序设计,需要学习在计算机内部如何表示这些值以及如何在 C 程序中使用枚举类型。

继续讨论公民的年退税数据的处理,国内税收服务部在审核公民的退税信息时,该过程的第一步就是将退税数据输人计算机系统。

为了存储枚举的保税人身份数据,计算机必须有一种表示每一个不同数据项的方法,包括公民的报税身份,于是就要解决如何规划出一个记录纳税人报税身份的方法的问题。

在解决这个问题之前,我们需要考虑一下计算机的能力。计算机擅长于处理数值型数据,它们就是为处理数值型数据而制造的。作为基本硬件操作的一部分,它们能存储、加、减、比较以及各种数值型数据可以做的事情。

计算机擅长于处理数值型数据的事实给出了一个枚举类型表示问题的解决方案。为了表示一个任何类型数值的有限集合,所需要做的就是给每个值分配一个数字。例如,有一组允许的报税身份值,可以简单地对它们计数,设 single 为 1,married filing joint return 为 2,married filing separate return 为 3,依次类推。(事实上,这些数字编码已经直接列在税单上了。)

为每一个可能值分配一个整数值意味着我们能用整数表示对应的报税身份。因此,定义任何枚举类型的表示所需要做的只是对它的元素进行编号。为枚举类型的每个元素赋一个整数值的过程称为整数编码(integer encoding),即将整数作为原来值的代码。

61.4 Definition

61.4.1 macro

在 C 语言中可以用不同的方法表示枚举类型,一种方法是显式地用 int 类型,然后用 #define 功能引入一些新的常量名,比如在 calendar.c 程序中就有用整数表示枚举类型的例子。

下面的 calendar.c 程序记录了一星期中的每一天。为了定义每周中的每一天的名字,程序为每一天的名字分配一个数字:星期日是 0,星期一是 1,星期二是 2,依次类推。然后用 #define 功能将这些数字定义为常量,如下所示:

```
#define Sunday          0
#define Monday          1
#define Tuesday         2
#define Wednesday       3
#define Thursday        4
#define Friday          5
#define Saturday        6
```

为了用这种方法表示枚举类型,必须显式指定整数编码。如果要引人一个变量保存这种类型的值,可以把它定义为 int 型。

也可以用这种方法表示退税单上的报税身份,可以定义如下常量:

```
#define Single                          1
#define MarriedFilingJointReturn        2
#define MarriedFilingSeparateReturn     3
#define HeadOfHouseHold                 4
#define QualifyingSurvivingSpouse       5
```

最后一个常量的名字已经从税单上的名字改为符合 C 语言规则的名字，因为 C 语言的名字中不允许出现圆括号。

一旦定义了这些常量，就可以声明一个 int 类型的变量来表示报税身份：

```
int filingStatus;
```

使用 #define 机制在 C 语言中将枚举类型表示为整数，并没有完全发挥该语言提供的功能的所有优势。如果不使用 int 类型表示柞数字值，在 C 语言中，可以定义一个真正的类型名去表示一种枚举类型。

在机器内部，这两种策略产生的结果完全相同：枚举类型中的每个元素都用一个整数代码表示。从程序员的观点来看，定义单独的枚举类型有三个优势：

- 编译器能选择整数代码，从而将程序员解脱出来。
- 有一个独立的类型名使程序容易读，这样声明时可以使用有意义的名字，而不是通用的符号 int。
- 在许多计算机系统中，使用显式定义的枚举类型的程序容易调试，因为编译器可以为调试系统提供有关该类型行为的额外信息。

61.4.2 typedef

C 语言提供了多种定义新的枚举类型的语法形式。在本书中，所有新的枚举类型都是用如下所示的语法定义的。

语法：枚举类型的定义

```
typedef enum{
  list of elements
}typename;
```

该定义以关键词 typedef 开头，它表示引人了一种新的类型名，因此关键词 typedef 也可以用在其他类型定义中。

C 语言用关键字 typedef 和 enum 定义新的枚举类型，因此关键词 typedef 后面紧跟着关键词 enum 来指出新的类型是枚举类型。

枚举类型中的每个元素的名字列在一对花括号中，再接下去是类型的名字和一个分号。

其中：

- *list of elements* 是由组成枚举类型的每个值的名字组成的，表中的元素用逗号分开。每个元素后面也可以紧跟一个等号和一个整数常量，它指出了特定的内部表示。
- *typename* 指出新的枚举类型的名字。

用这种方法，可以在 calendar.c 程序中引入下面的枚举类型来定义一周中的每天的名字。

```
typedef enum{
   Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
} weekdayT;
```

这个定义引人了一个新的类型 weekdayT，它在程序中可以和其他类型一样使用，如 int 和 bool。名字结尾处的大写字母 T 强调名字 weekdayT 是一种类型，而不是一个变量。

可以形成如下的约定：

> 所有定义的类型名（除了 bool 和 string 等，它们被认为是基本类型集合的一部分）都用大写字母 T 结尾。这种格式上的约定增加了程序的可读性。

在 calendar.c 程序中，表示一周中每天的名字的变量声明需要改变，使之能用 weekdayT 类型取代 int 类型。

在程序中使用枚举类型的好处在于该类型名能传递一些有用的信息。将一个变量声明为 int 类型会使任何一个读此程序的人认为该变量保存的是整数，该整数代表的是一个星期中的某一天的功能完全丧失了。声明一个变量为 weekdayT 类型，则立即告诉读者该变量包含的值的类型。

weekdayT 的定义也定义了对应于一周中每一天的七个常量，这些常量类似于用 #define 引人的常量。事实上，它们有同样的内部表示。

只要定义一个新的枚举类型，其中的元素将被赋以从 0 开始的连续的整数。因此，在 weekdayT 这个实例中，和以前一样，Sunday 的值还是 0，Monday 的值为 1，依次类推。

作为定义的一部分，C 语言的编译器也允许程序员显式指出枚举类型的元素的内部表示。例如，对国内税收服务部来说，用于表示报税身份的内部代码要有一个确切的值，它们要被打印在税单上。毕竟，他们已经保存了许多年的数据，在这些数据中，报税身份被编码为 single 等于 1，等等。在定义报税身份类型时，需要指定常量的值，如下所示：

```
typedef enum{
   Single = 1,
   MarriedFilingJointReturn = 2,
   MarriedFilingSeparateReturn = 3,
   HeadOfHouseHold = 4,
   QualifyingSurvivingSpouse = 5
} filingStatusT;
```

而且这个定义可以继续简化，去掉等号，编译器仍然会对每一个元素依次赋值，因此下面的定义与前面的定义有完全相同的作用。首先，元素 Single 被明确地赋值为 1，其余的名字从 1 开始连续编号。

```
typedef enum{
   Single = 1,
   MarriedFilingJointReturn,
   MarriedFilingSeparateReturn,
   HeadOfHouseHold,
   QualifyingSurvivingSpouse5
} filingStatusT;
```

可以对任意类型的值使用枚举。例如可以用下面的语句定义彩虹的颜色。

```
typedef enum{Red, Orange, Yellow, Green, Bule, Violet} colorT;
```

或者，可以用下面的语句来定义指南针上的四个基本方向。

```
typedef enum{North, East, South, West} directionT;
```

每个这样的定义都引入了一个新的类型名和对应于该类型元素的一组常量。

如果使用 #define 指令通过宏来定义布尔型数据类型，可以使用如下的语法：

```
#define BOOL int;
```

但是，一个更好的设置布尔型的方法是使用类型定义的特性。

```
typedef int Bool;
```

这样，采用 typedef 定义 Bool 会使编译器在它所识别的类型名列表中加入 Bool，从而 Bool 类型可以和内置的类型名一样用于变量声明、强制类型转换等。

可以使用 Bool 声明下面的布尔变量 flag，编译器将会把 Bool 类型看成是 int 类型的同义词，这样变量 flag 实际就是一个普通的 int 型变量。

```
Bool flag; /* same as int flag */
```

事实上，genlib.h 中的 bool 类型的定义就是一个枚举类型。

```
typedef enum{FALSE, TRUE} bool;
```

在一个表示布尔条件的 C 语言表达式中，整数 0 表示条件为假。尽管整数 1 是常量 TRUE 的“官方”值，但实际上任何非 0 的值都被解释为真。

C 程序员用经常用 C 语言解释非 0 整数为真的条件，因为限制在条件中使用明确的布尔值，将有利于逐渐养成良好的程序设计习惯，也使得编写的程序比较容易阅读和调试。

类型定义使得程序更加易于理解，例如通过类型定义美元数据类型后，就可以使用变量 case_in 和 cash_out 可以用于存储美元数量。

```
typedef float Dollars;
Dollars case_in, cash_out;
```

相比而言，这样的写法比下面的一般写法更有实际意义。

```
float cash\_in, cash\_out;
```

类型定义还可以使程序更容易修改，例如如果需要修改 Dollars 的数据类型，可以在类型定义中修改。

```
typedef double Dollars;
```

这样，Dollars 变量的声明不需要进行改变。如果不使用类型定义，则需要修改所有用于存储美元数量的 float 类型变量并且修改它们的声明。

在实际程序开发中，不同计算机上的类型取值范围可能不同，因此可以使用 typedef 来定义新的整型名来获得更大的可移植性。

如果 i 是 int 型的变量，那么赋值语句

```
int i=100000;
```

在使用 32 位整数的计算机上没有问题，但是在 16 位整数的计算机上就会出错。

相比 long int 等类型，算术运算时 int 型值比 long int 型值运算速度快，而且 int 型变量可以占用较少的空间，因此用户更愿意使用 int 型的变量。

通过自定义的类型，可以避免直接使用 int 类型声明变量。

```
typedef int Quantity;
Quantity q;
```

当把程序转到使用小值整数的硬件平台上时，只需改变 Quantity 类型的定义既可。

```
typedef long int Quantity;
```

C 语言标准库自身使用 typedef 为那些可能依据 C 语言实现的不同而异的类型创建类型名，这些类型名通常以 _t 结尾(例如 ptrdiff_t、size_t 和 wchar_t 等)，编译器可能在对应的库中有下列类型定义。

```
typedef int ptrdiff_t;
typedef unsigned size_t;
typedef char wchar_t;
```

其他编译器可能采用不同的方式定义这些类型，例如 ptrdiff_t 可能在某些机器上是 long int 类型。

在类型定义和宏定义之间存在着两个重要的不同点。

- 首先，类型定义比宏定义功能更强大，特别是数组和指针类型是不能定义为宏的。
  下面的示例试图使用宏来定义“指向整数的指针”类型。

  ```
  #define PTR_TOINT int *
  ```

  声明

  ```
  PTR_TO_INT p, q, r;
  ```

  在处理以后，将会变成

  ```
  int * p, q, r;
  ```

  这样就只有 p 是指针，q 和 r 都成了普通的整型变量，而使用类型定义就不会有这样的问题。

```
typedef (int *) PTR_TO_INT;
```

- 其次，typedef 命名的对象具有和变量相同范围的规则，定义在函数体内的 typedef 名字在函数外是无法识别的，而宏的名字在预处理时会在任何出现的地方被替换。
-

61.5 Operation

在表达式中用到枚举类型的值时，C 编译器自动地立刻将它转换为整数。因为每个枚举常量都用一个整数值表示，所有涉及枚举类型的计算只是用到它的整型代码代替。

例如，如果声明一个变量 weekday 为 weekdayT 类型，仍然可以在 calendar.c 程序中使用下面的表达式：

```
weekdayT weekday;
weekday = (weekday + 1) % 7
```

变量 weekday 可以是 0 6 之间的某一个内部整数值。该语句将此整数值加 1，然后通过模 7 取余操作保证结果还在 0 6 之间，从而说明，枚举类型的所有算术运算都与整数一样。

然而，在 C 语言中，使用枚举类型时要非常小心，因为编译器不检查计算的结果是否是某一枚举类型的合法成员。例如，在下面的语句中

```
weekdayT weekday;
weekday = weekday + 1;
```

而若此时 weekday 的值碰巧为 Saturday，就会遇到麻烦。计算机将取表示 Saturday 的值 6，对它加 1，把这个值 7 存回到变量 weekday，此时便产生了错误，因为 7 不是 weekdayT 类型的合法元素。

C 编译器应该生成一段额外的代码去检查这个条件，并报告错误，但很少有编译器这样做。当然，如果将 weekday 作为一个整数，不考虑取余数的操作也是会产生错误的，因此，选择使用枚举类型不是问题的起因。

第 62 章

Scalar Type

行为类似于整数的类型(如枚举类型)称为标量类型(scalar type),它们具有保存单一数据项的能力。

枚举类型是标量类型的子类,标量类型的行为与整数的行为完全一致。

在 C 语言中,在表达式中使用标量类型时,它们被自动转换为整数,因此标量类型和整数类型之间不需要显式的转换。

此外,标量类型可以用在任何整数类型出现的地方。例如,一个枚举类型的变量能被用在 switch 语句的控制表达式中。假设 directionT 被定义为

```
typedef enum{North, East, South, West} directionT;
```

下面的函数返回参数指定的方向的反方向:

```
directionT OppositeDirection(directionT dir)
{
  switch(dir){
    case North:   return (South);
    case East:    return (West);
    case South:   return (North);
    case West:    return (East);
  }
}
```

因此,调用 OppositeDirection(North) 将返回 South。

第 63 章

Character Type

字符是所有文本数据处理的基础,只是字符串在程序中比单个字符出现得更频繁。

字符是基础类型,是构建所有其他形式的文本数据的“原子”,因此理解字符是如何工作的是理解所有其他文本处理的关键。

在某种意义上,字符组成了一种内置的枚举类型,虽然术语枚举类型通常是指用关键词 enum 指定的类型。然而,众所周知,字符是一种标量类型,因此和用户定义的枚举类型一样,字符属于同一个通用类型。

63.1 Char type

在 C 语言中,单个字符用数据类型 char 来表示,它是预定义的数据类型中的一种。与其他的基本类型一样,类型 char 由一个合法值的值域和一组操作这些值的运算组成。

通俗地说,数据类型 char 的值域是一组能在屏幕上显示或能在键盘上输入的符号,这些符号(包括字母、数字、标点符号、空格、回车键等等)是所有文本数据的基本构件。

char 是标量类型,因此字符可用的运算集合与整数是一样的,可以用标准的算术运算来处理 char 类型数据。然而,理解这些运算在字符域中的含义需要更进一步认识字符在机器内部的表示。

单个字符在机器内部的表示方法和其他标量类型一样。从概念上说,基本思想就是为每个字符赋一个编号,通过把这些字符写在一个表中,然后对它们计数来实现编号过程。

用于表示特定字符的代码称为它的字符代码(character code)。例如,可以用整数 1 代表字母 A,整数 2 代表字母 B,等等。在用 26 表示字母 z 后,可以继续用整数 27、28、29 等编码小写字、数字、标点符号和其他的字符。

尽管设计一台用数字 1 表示字母 A 的计算机在技术上是可行的,但这样做确实是错误的。在当今世界,信息通常在不同的计算机之间共享,为了使这种计算机之间的通信成为可能,计算机必须能以某种公共的语言“互相交谈”。这种公共语言的基本特性在于计算机用同样的代码表示字符,这样,一台机器上的字母 A 不会变成另一台机器上的字母 Z,它仍被显示为字母 A。

在早期的计算机中,不同的计算机确实用不同的字符代码。字母 A 在一台计算机上有特定的表示,但在由另一个生产商生产的计算机上有完全不同的表示。甚至可用字符集也会改变。例如,一台计算机的键盘上可能有字符 Q,而另一台计算机则完全不能表示这个字符。计算机之间的通信困难和说不同语言的人进行交流所遇到的困难一样。

然而,随着时间的推移,计算机之间相互通信带来的许多好处使许多计算机生产商采用了统一的字符表示标准,这种标准称为 ASCII(发音为“as-key”)字符编码系统,它表示信息交换的美国标准代码(American Standard Code for Information Inter-change)。

按照预先定义的字符编码,字符在计算机系统内部被表示为整数,虽然某些计算机使用不同的字符编码系统,但大多数现代计算机系统都会采用 ASCII 编码系统。后来有些计算机把 ASCII 码扩展为 8 位代码以便可以表示 256 个字符,这称为 ASCII 的扩展版本。

下表显示的是表示每个字符的 ASCII 代码值。

以下是 ASCII 表的两个结构特性:

(1)表示数字 0 ～ 9 的字符的代码是连续的。

0	\000	\001	\002	\003	\004	\005	\006	\a	\b	\t
10	\n	\v	\f	\r	\016	\017	\020	\021	\022	\023
20	\024	\025	\026	\027	\030	\031	\032	\033	\034	\035
30	\036	\037	空格	!	“	#	$	%	&	‘
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	\177		

根据枚举类型的性质，程序员可以不关心哪个代码对应对应数字字符 ‘0’，只要知道数字 ‘1’ 的代码是比 ‘0’ 的代码大 1 的整数，同样，如果给 ‘0’ 的代码加 9，就是字符 ‘9’。

(2) 字母按字母顺序分成两段：一段是大写字母 (A ～ Z)，另一段是小写字母 (a ～ z)，然而在每一段中，ASCII 是连续的。

根据枚举类型的性质，为了得到字母的 ASCII 码，可以按照字母顺序进行计算。事实上，ASCII 没有保证第二条特性，尽管它几乎总是对的，仍然有许多采用字符编码系统的计算机系统中的字母是不连续的，但在这些计算机上很少使用 C 语言编程，而字母的连续集合的假设可以大大简化某些程序设计问题。

除了出现在键盘上的许多项，但还有一些用反斜杠 (\) 开头，后面接一个字母或一组数字的用户不太熟悉的项，这些项称为特珠字符。

要计算上表中任何字符的 ASCII 值，只要将该项行和列相关的数字相加即可。例如，在图中心位置上的字母 A 所在的行标记为 60，列标记为 5。因此，字母 A 的 ASCII 值是 60 + 5，65。以同样的方法，能找出任何字符的编码。然而，在大多数情况下，并不需要这样做。

虽然知道字符在计算机内部用什么数字编码很重要，但知道哪个数字值对应于哪个特定字符并不是很有用。当键入字母 A 时，键盘中的硬件自动将此字符翻译成 ASCII 代码 65，然后把它发送给计算机。同样，当计算机把 ASCII 代码 65 发送到屏幕时，屏幕上会出现字母 A。

63.2 Character set

char 类型的值可以根据计算机的不同而不同，原因在于不同的机器可能会有不同的字符集。

ASCII 字符集是最常用是字符集，使用 7 位代码表示 128 个字符，其取值范围是 0000000 ～ 1111111，因此字符可以看成是 0 ～ 127 的整数。

具体来说，在 ASCII 码中所有字符都是以二进制的形式编码的，数字 0 ～ 9 用 0110000 ～ 0111001 来表示，大写字母 A ～ Z 用 1000001 ～ 1011010 表示，这导致在 C 语言中字符的操作非常简单。

其他一些计算机则使用完全不同的字符集，例如 IBM 主机依赖早期的 EBCDIC 代码。不过，现代计算机开始遵循 Unicode 并使用 16 位代码来来表示 65536 个字符。

63.3 Character Constant

在 C 语言程序中用到某一特定字符时，标准的方法是指定一个字符常量（character constant），即将所需的字符括在一对单引号中。

可以使用计算机能表示的任意字符给 char 类型的变量赋值。

```
char ch;
ch = 'a'; /* lower-case a */
ch = 'A'; /* upper-case A */
ch = '0'; /* zero */
ch = '9'; /* nine */
ch = ' '; /* space */
```

当程序中出现字符时，C 编译器是以其对应的整数值来处理字符的，而且 char 类型也相应地划分为有符号型和无符号型。通常，有符号型字符的取值范围是-128 ～ 127，无符号型字符的取值范围是 0 ～ 255。

C 语言标准没有说明普通字符型数据是有符号型还是无符号型，一些编译器把它们按照有符号型数据来处理，而另外一些编译器则将它们处理成无符号型数据，也有的编译器允许程序员通过编译器选项来选择字符类型是有符号型还是无符号型。

- 如果在变量中只存储 7 位的字符，因为符号位为 0，那么不需要考虑字符型数据有无符号。
- 如果计划存储 8 位字符，那么将希望是 unsigned char 类型。

思考下面的例子：

```
ch = '\xdb';
```

如果已经把变量 ch 声明成 char 类型，那么编译器可能选择把它看作是有符号的字符来处理。只要变量 ch 只是作为字符来使用，就不会有什么问题。但是如果 ch 用在一些需要编译器将其值转换位整数的上下文中，那就可能有问题了：转换为整数的结果将是负数，因为变量 ch 的符号位为 1。

还有另外一种情况：在一些程序中，习惯上使用 char 型变量存储单字节的整数。如果编写了这类程序，就需要决定每个变量应该是 signed char 类型的还是 unsigned char 类型的，这就像需要决定普通整型变量应该是 int 类型还是 unsigned int 类型一样。

大多数情况下，字符型数据是被存储为小值整数的，因此标准 C 允许使用 signed 和 unsigned 来修饰 char 类型，而且可以对字符进行比较。

下面的示例测试 ch 是否含有小写字母。如果有，那么它会把 ch 转化为相应的大写字母。

```
char ch
signed char sch;
unsigned char uch;

if('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

这里比较表达式中 'a' <= ch 使用的就是字符所对应的整数值，这些整数值依据所使用的字符集有所不同，所以使用 <、<=、> 和 >= 等来进行字符比较可能不易移植。

在 C 语言中，若要指出字母 A 的 ASCII 值，所需要做的只是输入 'A'，C 编译器知道这个符号指的是使用字母 A 的 ASCII 值——即 65。同样，可以用 ' ' 指出空格，或用 '9' 指出数字 9。

注意，常量 '9' 指的是一个字符，不应该与整数值 9 混淆。作为一个整数，值 '9' 是 ASCII 表中给出的这个字符的值——即 57。

只要计算机使用 ASCII 字符集，就应该用整数 65 代替字符常量 'A'。程序在处理字符时也是按照字符的 ASCII 值进行文字处理的，但这样程序将会很难读。

字符拥有和数相同的属性，这一事实会带来一些好处，例如在 for 语句中的控制变量可以使用大写字母。

```
for(ch = 'A'; ch <= 'Z'; ch ++)
  ...
```

使用字符常量可以更直接地传递出字符的意义，因为会有其他程序员要读你写的程序。ASCII 字符集的统一使得程序员更注重字符本身，而不是字符的特定 ASCII 代码。

另一方面，以数的方式处理字符可能会导致编译器无法检查出来的多种程序错误，还可能会导致一些无意义的表达式。此外，这样做同时也会影响程序的可移植性，因为这里程序实际上是基于一些对字符集的假设基础上的。

常见问题：避免在程序中用整数常量表示 ASCII 字符。所有的字符常量应该用单引号将此常童括起来表示，如 'A' 或 '9' 等。

> 可移植性技巧：不要假设 char 类型默认为 signed 还是 unsigned。如果需要注意，要用 signed char 或 unsigned char 来代替 char。

63.4 Escape Sequence

ASCII 表中的大多数字符都能显示在屏幕上，这些字符称为可打印字符（printing character），可以用单引号来指定字符。

ASCII 表也包括许多特殊字符（special character），它们不能被显示在屏幕上，而是用来完成某一特定的动作。

换行符是常用的一个特殊字符，它是由两个字符组成的序列\n 指出的。

换行符大多数出现在 printf 调用中，用来将光标移到屏幕上下一行的开始位置。除了换行字符，还有许多具有预定义功能的特殊字符。

为了使 C 程序可以处理字符集中的每一个字符，C 语言提供了使用反斜杠后跟一个字母或数值来表示特殊字符，这种反斜杠和它后面的字符的组合被称为转义序列（escape sequence）。

在 C 语言中的转义序列分为两种，分别是字符转义序列（character escape）和数字转义序列（numeric escape）。

转义序列	功能
\a	报警声（嘟一声或打铃）（alert）
\b	退格（backspace）
\f	换页（开始一新页）
\n	换行（移到下一行的开始）（next row）
\r	回车（回到当前行的开始）（return）
\t	Tab（水平移到下一个 tab 区）
\v	垂直移动（垂直移到下一个 tab 区）
\o	空字符（ASCII 代码为 0 的字符）
\\	字符 “\” 本身
\'	字符 “'” （仅在字符常量中需要反斜杠）
\"	字符 “"” （仅在字符常量中需要反斜杠）
\ddd	ASCII 代码为八进制值 ddd 的字符

其中，\a、\b、\f、\r、\t 和\v 表示了通用的 ASCII 码控制字符，\n 表示了 ASCII 码的换行符，\\允许字符常量或字符串包含字符\，\' 允许字符常量包含字符'，而转义序列\" 则允许字符串包含字符"。

作为 C 语言的 UNIX 继承部分，一直把行的结束位置标记作为单独的回行字符来看待，单独一个回行符（但不是回车符）会出现在每行的结束处。

C 语言函数库会把用户的按键翻译成回车符，当程序读文件时，输入/输出函数库将文件的行结束标记翻译成单独的回行符，与之相对应的反向转换发生在将输出往屏幕或文件中写的时候，这样可以使程序不受不同操作系统的影响。

将转义序列作为字符常量的一部分就可以在字符常量中包含特殊字符。虽然每个转义序列由几个字符组成，但在机器内部，每个序列被转换为一个 ASCII 代码。这些特殊字符依照其 ASCII 代码值完成其对应的功能，例如，换行符的内部表示为整数 10。

当编译器看见反斜杠字符时，它把该字符看成是转义序列的第一个字符。如果你要表示反斜杠本身，就必须在一对单引号中用两个连续的反斜杠，即“\\”。同样，当单引号被用作字符常量时，必须在前面加上一个反斜杠“\”。

转义序列“\?”与以?? 开头的三字符序列有关，如果需要在字符串中加入??，那么编译器很可能会把它误当作三字符序列的开始，因此需要使用\? 代替第二个? 来解决这个问题。

特殊字符也能用于字符串常量。由于双引号作为字符串结束标记，因此，当双引号作为字符串的一部分时，也必须将其标记为特殊字符。

例如，下面的这个包含 printf 行的程序

```
printf("\"Bother, \"said Pooh.\n");
```

该语句的输出为：

```
"Bother," said Pooh.
```

ASCII 表中的许多特殊字符都没有明确的名字，在程序中使用它们的内部代码来表示，字符转义序列无法包含所有无法打印的 ASCII 字符，也无法用于表示基本的 ASCII 码字符以外的字符。

数字转义字符可以表示任何字符，所以它解决了字符转义序列的不足。

在此过程中，唯一的不便之处是这些特殊字符的数字代码是用八进制表示的，通常称为八进制计数法（octal notation）。

- 八进制转义序列由字符\和跟随其后的一个最多含有 3 位数字的八进制数组成。
  转义序列中的八进制数不一定要用 0 开头，但是必须表示为无符号字符型，因此最大值通常是八进制的 377
- 十六进制序列由\x 和跟随其后的一个十六进制数组成。
  标准 C 对十进制数中的数字个数没有限制，但是必须可以将数表示成无符号型字符，因此如果字符是八位长度，那么十六进制数中的数不能超过 FF。
  若采用十六进制转义序列，可以把转义字符写成\x1b 或\x1B 的形式，即字符 x 必须小写，但是十六进制的数字不限制大小写。

作为字符常量使用时，转义序列必须用一对单括号括起来，因此'\33' 和'\x1b' 意义相同。

在八进制计数法中，每一位数字是它右边数字的八倍。例如，字符常量 '\177' 表示 ASCII 值为八进制数 177 的字符（这个字符对应于表示 Delete 键的代码，在某些键盘上标记为 Rubout）。

在数值上，八进制值 177 对应于整数

$$1 \times 8^2 + 1 \times 8^1 + 7 \times 8^0 = 127$$

另外，在实际程序开发过程中，转义序列往往意义隐晦，因此可以采用 #define 的方式来给它们命名，从而可以将转义序列嵌入到字符串中使用。

```
#define ESC '\33' /* ASCII escape character */
```

转义序列不是用于表示字符的唯一一种特殊符号，C 语言中加入里另外一些表示字符的方式。

- 三字符序列（trigrph sequence）是一些特殊的 ASCII 字符的代码。
- 多字节字符（multibyte character）和宽字符（wide character）用于某些无法用一个字节存储编码的字符集中。

第 64 章

Character Operation

在 C 语言中，字符值能像整数一样计算，不需要特别转换。结果是根据其内部 ASCII 代码定义的。例如，字符'A' 在计算机内部用 ASCII 代码 65 表示，在运算时被当作整数 65 处理。

因为整数和字符能自由地相互转换，所以可以很容易地定义一个函数 RandomLetter，返回一个随机选择的大写字母。通过使用 random.h 接口导出的 RandomInteger 函数，它的实现可写为

```
char RandomLetter(void)
{
  return (RandomInteger('A', 'Z'));
}
```

尽管对 char 类型的值应用任何算术运算都是合法的，但在它的值域内，不是所有运算都是有意义的。例如，在程序中将'A' 乘以'B' 是合法的。为了得到这个结果，计算机取它的内部代码，65 和 66，将它们乘起来，得到乘积 4290。问题在于这个整数作为字符毫无意义，事实上，它超出了 ASCII 字符的范围。

当对字符进行运算时，仅有很少的算术运算是有用的。这些有意义的运算通常有：

1. 给一个字符加上一个整数
   如果 c 是一个字符，n 是一个整数，表达式 c + n 表示代码序列中 c 后面的第 n 个字符。例如，如果 n 在 0 ～ 9 之间，表达式 '0'+n 得到的是第 n 个数字的字符代码。因此 '0'+5 是 '5' 的字符代码。同样，如果 n 在 1 ～ 26 之间，那么 'A' + n-1 表示字母表中第 n 个字母的字符代码。
2. 从一个字符中减去一个整数
   表达式 c-n 表示代码序列中 c 前面的第 n 个字符。例如，表达式 'Z' - 2 的结果是 'X' 的字符代码。
3. 从一个字符中减去另一个字符
   如果 c1 和 c2 都是字符，那么表达式 c1-c2 表示两个字符在代码序列中的距离。例如，在 ASCII 表中，可以确定 'a' - 'A' 是 32。更重要的是，小写字母和与之对应的大写字母之间的距离是固定的，因此 'z' - 'Z' 也是 32。
4. 比较两个字符
   用某个关系运算比较两个字符的值是常用的运算，经常用来确定字母的次序。例如，如果在 ASCII 表中，c1 在 c2 前面，那么表达式 c1 < c2 是 TRUE。

为了了解在实际问题如何应用这些关于字符的运算，考察计算机是如何执行 GetInteger 这样的函数的。

当用户输入一个数字（如 102）时，计算机将每一次击键作为一个字符接收，因此输入的值为 '1'，'0'，'2'。因为 GetInteger 函数必须返回一个整型数，需要将字符转换为相应的整数。要做到这一点，GetInteger 利用数字在 ASCII 表中连续的特点。例如，假设 GetInteger 已经从键盘读入了一个字符，并把它保存在变量 ch 中。它可以用表达式

$$ch - '0'$$

将字符转换为数字形式。

假设 ch 包含一个数字字符，它的 ASCII 代码和 '0' 的 ASCII 代码之间的差正好对应于这个数字的数值。例如，假设变量 ch 包含字符 '9'，字符 '9' 的代码为 57，数字 '0' 的 ASCII 代码为 48。

$$57 - 48 = 9$$

上述的关键在于该函数没有假设'0' 的 ASCII 代码为 48，这意味着同一个函数可以用于具有不同字符集的其他计算机上，只需要假设在此字符集中数字的代码是连续的。

下面讨论 GetInteger 函数如何确定字符 ch 是否为数字。

利用 ASCII 表中数字是连续的事实，语句

```
if(ch >= '0' && ch <= '9') . . .
```

将数字字符与 ASCII 字符集中的其他字符区分开了。同样地，语句

```
if(ch >= 'A' && ch <= 'Z') . . .
```

标识出了大写字母，而语句

```
if(ch >= 'a' && ch <= 'z') . . .
```

标识出了小写字母。

64.1 ctype.h

C 语言的设计者将检查一个字符是否为数字或字母的操作放入一 ctype 库中。

ctype.h 导出一些为单个字符分类以及改变单个字符大小写的函数。与任何其他接口一样，可以通过包含行

```
#include <ctype.h>
```

来获得这些函数的访问权。ctype.h 接口声明了一些确定字符类型的谓词函数，下面是最重要的一些函数。

islower(ch)	如果字符 ch 是小写字母，则返回 TRUE。
isupper(ch)	如果 ch 是大写字母，则返回 TRUE。
isalpha(ch)	如果 ch 是字母（不管大写还是小写），则返回 TRUE。
isdigit(ch)	如果 ch 是数字，则返回 TRUE。
isalnum(ch)	如果 ch 是字母数字（alphanumeric），即或者是字母或者是数字，则返回 TRUE。
ispunct(ch)	如果 ch 是标点符号，则返回 TRUE。
isspace(ch)	如果 ch 是下列字符之一，则返回 TRUE。这些字符是' '（空格字符）、'\t'、'\n'、'\f'或'\v'，所有这些字符在屏幕上都显示为空格。

此外，ctype.h 还定义了以下的转换函数：

1. tolower(ch)
   如果 ch 是大写字母，返回它对应的小写字母；否则，返回 ch 本身。
2. toupper(ch)
   可以使用下面的语句来将小写字母转换成大写字母。

   ```
   if('a' <= ch && ch <= 'z')
     ch = ch - 'a' + 'A';
   ```

   这种方式并不是最好的，ctype 库提供了更快捷、更易于移植的 toupper 函数。

   ```
   ch = toupper(ch); /* converts ch to upper case */
   ```

   被调用时，toupper 函数检测自身的参数是否是小写字母，并返回它对应的大写字母，否则返回 ch 本身。

虽然 ctype.h 接口已经提供了 tolower 和 toupper 两个函数，但如果从头实现它们，则能更进一步了解它们的操作。同样，可以忽略确切的 ASCII 代码，仅依赖于连续的假设进行实现。

如果 ch 包含一个大写字母的字符代码，可以加上一个大写字母和小写字母的内码之间的差，将它转换为小写。然而，这里并没有把这个差作为一个确定的常量，如果把它表示成字符运算 'a' - 'A'，则程序的可读性更好。因此，可以用如下代码实现这个 tolower 函数：

```
char tolower(ch)
{
  if(ch >= 'A' && ch <= 'Z'){
    return (ch + ('a' - 'A'));
  }else{
    return (ch);
  }
}

char toupper(ch)
{
  if(ch >= 'a' && ch <= 'z'){
    return (ch + ('A' - 'a'));
  }else{
    return (ch);
  }
}
```

尽管在标准库 ctype.h 中定义的函数都很容易实现，但使用库函数而不是自己编写的函数是良好的程序设计习惯，这样做有三个主要的理由。

1. 因为库函数是标准的，使得使用库函数的程序容易阅读和维护，而且不同的程序员都有 C 语言的经验，他们了解 ctype.h 中的函数，并确切知道它们的含义。
2. 库函数比我们自己写的函数更能保证正确性。因为 ANSI C 的库被成千上万个客户程序员所使用，实现者为保证函数的正确性作了大量的工作。如果你自己重写库函数，那么带来错误的机会更大。
3. 函数的库实现通常比我们自己写的函数效率更高。例如，在 ctype.h 接口中，库中的机制比上述给出的实现运行速度更快，通常是快三或四倍。重要的是，可以使用库可以提高程序的效率。

64.2 Control statements involving characters

char 是标量类型，因而可以把它用在可以出现整数的所有语句中。例如，如果 ch 声明为 char 类型，可以用下列的 for 语句执行 26 次循环，按字母顺序对每个大写字母执行一次：

```
for(ch = 'A'; ch <= 'Z'; ch ++)
```

同样，也可以将字符用在 switch 语句的控制表达式中。例如，如果参数为元音的话，下面的谓词函数返回 TRUE：

```
bool IsVowel(char ch)
{
  switch(tolower(ch)){
    case 'a': case 'e': case 'i': case 'o': case 'u':
      return (TRUE);
    default:
      return (FALSE);
  }
}
```

注意，对大写和小写的形式，实现中都用 tolower 函数识别元音。

64.3 Character input and output

习惯上，字符的输入输出是用标准 I/O 库函数 getchar 和 putchar 完成的。另外，可以调用自定义的 GetLine 读入完整的一行，然后在此行中选择一个字符。

转换说明%c 允许 scanf 函数和 printf 函数对单独一个字符进行读/写操作，因此可以用带有格式码%c 的 printf 函数在屏幕上显示一个字符。例如，下面的主程序用 IsVowel 谓词函数以大写字母的形式列出英语的元音：

```
main()
{
  char ch;
  printf( "The English vowel are:" );
  for(ch = 'A'; ch <= 'Z'; ch ++){
    if(IsVowel(ch)) printf( "%c" , ch);
  }
}
```

在读入字符前，scanf 函数不会跳过空白字符。如果下一个未读字符是空格，那么 scanf 函数返回的变量值将包含一个空格。

scanf 函数可以用来检查到输入行的结尾以确定最后读入的字符是否为换行符。例如，在下面的循环示例中将读入并忽略掉所有当前输入行中其余的字符。

```
do{
  scanf("%c", &ch);
}while(ch != '\n');
```

这样，当下次调用 scanf 函数时，将读入下一输入行中的第一个字符。

为了强制 scanf 函数在读入字符前跳过空白字符，需要在格式串转换说明%c 前面加上一个空格，这意味着“跳过零个或多个空白字符”。

```
scanf(" %c", &ch);
```

C 语言还提供了读/写单独一个字符的其他方法，特别是使用 getchar 函数和 putchar 函数来代替 scanf 函数和 printf 函数。

每次调用 getchar 函数，它会读入一个字符，并返回这个字符。为了保存 getchar 函数返回的字符，需要使用赋值操作将返回值存储在变量中。

```
ch = getchar(); /* reads a character and stores it in ch */
```

getchar 函数和 scanf 函数一样，都不会在读取时跳过空白字符。

每次调用 putchar 函数，可以向输出设备写一个字符。

```
putchar(ch);
```

相比 scanf 函数和 printf 函数，使用 getchar 和 putchar 函数执行速度快有两个原因。第一个原因是这两个函数比 scanf 函数和 printf 函数简单，而 scanf 函数和 printf 函数是设计用来读/写多种不同格式类型数据的。第二个原因是，为了额外的性能提升，通常 getchar 函数和 putchar 函数是作为宏来使用实现的。

另外，getchar 函数还有一个优于 scanf 函数的地方：因为返回的是读入的字符，所以 getchar 函数可以应用在多种不同地 C 语言惯用法中，包括用循环搜索字符或跳过所有出现的同一字符。

考虑下面的 scanf 函数循环，它用来跳过输入行的剩余部分。

```
do{
  scanf("%c", &ch);
}while(ch != '\n');
```

如果用 getchar 重写上述循环，允许把 getchar 函数的调用放入循环的控制表达式中，因此下面两种实现意义相同，但后者使循环更精简。

```
do{
  ch = getchar();
}while(ch != '\n');

while((ch = getchar()) != '\n')
  ;
```

后一种实现的循环读入一个字符，把它存储在字符变量 ch 中，然后将变量 ch 与换行符对比。如果不是换行符，那么执行循环（循环体实际为空），接着再次测试循环条件，从而读入新的字符。这里实际上不是真的需要变量 ch，可以只把 getchar 函数的返回值与换行符进行比较。

这其实是非常著名的 C 语言惯用法，虽然这种用法的含义是十分隐晦的。

```
while((getchar()) != '\n') /* skips rest of line */
  ;
```

getchar 函数在用于循环中搜寻字符和跳过字符同样有效，在下面的示例中使用 getchar 函数跳过无限数量的空格字符。

```
while((ch = getchar()) == ' ') /* skips blanks */
  ;
```

当循环终止时，变量 ch 将包含 getchar 函数遇到的第一个非空字符。

如果在同一个程序中混合使用 getchar 函数和 scanf 函数，需要注意 scanf 函数有留下后面字符的趋势，也就是说对于输入后面的字符（包括换行符）只是“看了一下”而已，并没有读入。如果试图先读入数再读入字符，下面的示例程序在读入 i 的同时，scanf 函数调用将会留下后面没有消耗掉的任意字符（包括换行符但不仅限于换行符），而 getchar 函数随后将取回第一个剩余字符，但这不是我们期望的结果。

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

为了说明字符的读取方式，下面的示例程序用来计算消息的长度。

消息的长度包括空格和标点符号，但是不包括消息结尾处的换行符。程序需要采用循环结构来实现读入字符和计数器自增操作，循环遇到换行符时立刻终止。

```
/* Determines the length of a message */
#include <stdio.h>

main()
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while(ch != '\n'){
    len ++;
    ch = getchar();
  }
  printf("Message is %d characters long.\n", len);

  return 0;
}
```

下面使用 C 语言惯用法来优化程序。

```
/* Determines the length of a message */
```

```
#include <stdio.h>

main()
{
  int len = 0;

  printf("Enter a message: ");
  while(getchar() != '\n')
    len ++;
  printf("Message is %d characters long.\n", len);

  return 0;
}
```

scanf 函数执行速度虽然不及 getchar 函数，但是 scanf 函数更灵活，例如通过%c 可以使 scanf 函数读入下一个非空白字符。

另外，scanf 函数也很擅长读取混合了其他数据类型的字符。假设输入数据中包含有一个整数、一个单独的非数值型字符和另一个整数，那么可以通过使用格式串%d%c%d 来读取这全部三项内容。

64.4 sizeof

在编写 C 语言程序时，可以使用运算符 sizeof 取得变量占用内存空间的信息，从而可以确定用来存储指定类型值所需空间的大小。

```
sizeof(type-name)
```

sizeof 只有一个操作数，这个操作数必须是一个放在圆括号内的类型名或是一个表达式。

- 如果操作数是一个类型，sizeof 运算符将会返回存储这个类型的数据所需要的内存字节数；
- 如果操作数是一个表达式，那么 sizeof 运算符将会返回存储这个表达式的值所需要的内存字节数。

sizeof 表达式的结果值是无符号整数，表示用来存储属于*类型名*的值所需要的字节数。

例如，表达式

```
sizeof (int)
```

将返回存储一个整型数据所需要的字节数。

与应用于类型时相反，当应用于表达式时 sizeof 不要求圆括号，即 sizeof i 和 sizeof(i) 结果相同，使用圆括号的原因是为了免受运算符优先级的干扰。例如，表达式

```
sizeof (x)
```

将返回存储变量 x 所需要的字节数。

表达式 sizeof(char) 的值始终为 1，但是对其他类型计算出的值可能会有所不同。在 16 位机上，表达式 sizeof(int) 的值通常为 2；在大多数 32 位机上，表达式 sizeof(int) 的值为 4。

通常情况下，运算符 sizeof 也可以应用于常量、变量和表达式。如果 i 和 j 是整型变量，那么 sizeof(i) 的在 16 位机上的值为 2，而且 sizeof(i+j) 的值也是 2。

sizeof 表达式的类型是由实现定义的，在显示前可以把表达式的值转换成一种已知的类型。

根据规定，sizeof 返回无符号整型数，所以最安全的方式是把 sizeof 表达式转换成 unsigned long 类型(最大的无符号类型)，然后用转换说明%ld 进行输出。

下面是采用强制方式显示 int 类型大小的方式。

```
printf("Size of int: %lu\n", (unsigned long)sizeof(int) );
```

这里，符号(unsigned long)告诉编译器将随后的表达式(sizeof(int))的值转换为无符号的长整型数。

编译器本身也可以计算 sizeof 表达式的值，所以运算符 sizeof 是一种特殊的运算符。

第 65 章

String Type

65.1 Overview

计算机最初是用于数值计算的,于是数字便成为计算机程序中基本的数据类型。现代计算机处理文本数据(text data)的能力也大大增强。文本数据即由键盘和屏幕上出现的独立的字符组成的信息,而且现代计算机处理文本数据的能力推动了字处理系统、在线参考库、电子邮件以及其他应用的发展。

字符串是由独立的字符组成,因此首先要了解字符是如何工作的,以及它们如何在计算机内部表示是非常重要的。

使用字符的真正的作用在于可以把它们串在一起形成一个字符序列,这称为字符串(string)。

虽然可以通过调用 GetLine 从用户那里读入一个字符串,调用 printf 将字符串显示在屏幕上,调用 StringEqual 确定两个字符串是否完全相同。然而,为了充分利用字符串为程序设计带来的诸多好处,还需要学习更多有关字符串的知识。为了完全了解字符串,你需要从不同的角度、不同的层次仔细研究它。

从归约和整体两方面考虑程序设计的问题时,如果关心数据表示的内部细节时,可以采取归约的方法。从这个方面来讲,是为了理解字符在计算机的内存中是如何存储的,这些字符序列如何被存储为一个字符串,以及 200 个字符的字符串如何放入保存两个字符的字符串的变量中这样的问题。

当从整体的观点考虑字符串时,是为了理解如何将字符串作为一个逻辑单位来操作。通过关注字符串的抽象行为,可以学会如何有效地使用它,而不必沉溺于细节问题。

理想情况下,最好能将归约与整体分开,把每个方面搞清楚。然而,理解字符串的内部结构需要先熟悉 C 语言中的一些更高级的主题,如数组和指针。

过早将注意力集中在字符串的表示上意味无法从抽象的角度(如字符串如何使用以及为什么要有字符串等)理解字符串。

为了保证能从整体上理解字符串,可以采用分阶段的方法。首先,利用字符串库了解字符串的抽象行为,这个库隐藏了许多复杂性。接着过渡到字符串表示的更细节的问题,最后能自己写一个完整的字符串库。除了可以逐步了解字符串以外,这个方法也提供了另一个演示有效接口设计和信息隐藏原则的实例。

65.2 C/C++ String

C 风格字符串[3] 特指在 C 语言中字符串的存储方式。在编程语言中,常常需要表示一段字符,如“今天你吃了么”,“how are you?”,“afjsa234234(*&(*(”等等。同一种字符串的写法在不用的编程语言中表示的字面值都是一样的,即引号中间的内容,但是在存储的处理上往往不一样。

在 C 语言中,字符串是以字符数组的形式进行存储的,且在数组中以‘\0’作为终结符。由于’\0’在 ASCII 中表示空字符(NULL),即在语义上不可能有有效字符与之重复,故用其来表示字符串的结尾至少在 ASCII 编码下是合理的。

在 C++ 语言中,除了继承了 C 语言中的这种字符串表达形式外,还新添了 string 类用来表达字符串。就表义来说,这两种字符串存储方式是等价的,但在处理的过程中却有显著的区别。在 string 类

中，所有的对字符串的操作都被封装为成员函数，因此只要 string 内部有统一的约定，可以不再使用 '\0' 作为结尾标志。但对于 C 语言中的字符串，所有的操作都是来源于 <string.h> 中的以 str 开头的函数，这些函数的特点就是都以 '\0' 作为所处理的字符串的结尾标志。

由于这些显著的特点，为了区分 C++ 中这两种不同的字符串，使用“C 风格字符串”来特指来源于 C 语言的字符串存储方式。

ASCII 编码及其扩充规范中，每个字符长度都不超过 1Byte，因此，在 C 风格字符串中用 '\0' 表示结尾是合法的。但在 UTF16 编码中，每个字符使用 2Byte 进行编码，故会出现其中一个字符为 0x00 的情况，此时如果仍使用 C 风格字符串，则在使用相关函数进行处理时，会在第一个 0x00 出现的位置就被认为是字符串已经结束，但其实字符串并不在此处终止。

UTF8 是一种很好的解决方案，UTF8 中字符的编码非定长，可能是 1Byte 或者是 2Byte，但是这种编码方案中用每个字符的前缀来表示当前字符的长度，因此既有足够的空间来存储较多的字符，又不会出现 0x00 导致字符串在被以 C 风格字符串处理时异常结束。

第 66 章

String Definition

在 C 语言中，可以通过扩展定义 string 类型。

```
typedef char *string;
```

作为一种抽象数据类型，string 类型的值是一个完整的、不可分割的实体。

可以将字符串传递给函数，也可以将字符串作为结果返回，还可以将一个字符串的值赋给另一个字符串。如果想要从字符串中选出一个字符或对字符串进行复杂的操作，就必须调用 strlib.h 等接口导出的函数。

为了在较低的细节层次上使用字符串，需要理解字符串的工作原理，并能在不同的抽象层次上操作字符串。

66.1 String Conceptes

在 C 语言中，为了更有效地使用字符串，必须能从三个角度理解字符串。

- 字符串作为字符数组
- 字符串作为指向单个字符的指针。
- 字符串作为具有整体的概念完整性的完整实体。

这三个观点不是矛盾的，而是互补的，每一个观点都描述了一个现实的角度。对于一个存储在计算机中的字符串，可以考虑这三种观点中的任何一种来考虑。

从不同的角度来考虑字符串时，字符串本身并不发生变化，唯一改变的只是如何表示字符串的概念。

66.1.1 Char Array

从根本上来说，字符串在计算机内部表示为一个以空字符结尾的字符数组，可以使用数组选择符号初始化字符串，或者从一个字符串中选取字符。

```
char str[6];

str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```

这里，从字符串中选择字符类似于从 strlib 库调用 IthChar 函数，相应可以得出用于处理字符串中每一个字符的标准习语。

```
for(i = 0; str[i] != '\0'; i++){
  ... /* 操作str[i]的循环体 */
}
```

C 语言可以将零值作为 FALSE 处理，并将所有非零值作为 TRUE 处理，而且将比较操作放在 for 循环控制语句外面，因此上述习语可以做相应的简化。

```
for(i = 0; str[i]; i++)
```

在每个循环周期中，选择表达式 str[i] 的值是字符串中的单个字符，循环表达式一直持续到 str[i] 选择了标志字符串结束的空字符为止，从而可以使用 strlib 扩展库中的 StringLength 函数和 IthChar 函数来重写上述习语。

```
for(i = 0; i < StringLength(str); i++){
  ... /* 操作IthChar(str, i)的循环体 */
}
```

相比而言，利用数组选择形式比使用 strlib.h 接口中的函数有效地多。

首先，数组选择避免了调用 IthChar 函数所需付出的代价。

另外，通常字符串长度是不变的，但是在 for 循环控制语句中使用 StringLength(str) 作为测试条件的一部分时，程序必须在每个循环周期中都重新计算一次字符串长度。如果使用检查结束标记的方法，可以避免这些冗余的计算。

为了对这两种方法进行比较，可以不采用任何 strlib.h 接口中的函数来重写 Pig Latin 的 FindFirstVowel 函数。

```
int FindFirstVowel(char word[])
{
  int i;

  for(i = 0; word[i] != '\0'; i++){
    if(IsVowel(word[i])) return (i);
  }
  return (-1);
}
```

相比 FindFirstVowel 函数的非数组实现，在算法上是完全相同的，对检测字符串结束和选择每个字符的方法进行了封装。

```
int FindFirstVowel(string word)
{
  int i;

  for(i = 0; i < StringLength(word); i++){
    if(IsVowel(IthChar(word, i))) return (i);
  }
  return (-1);
}
```

66.1.2 Array Pointer

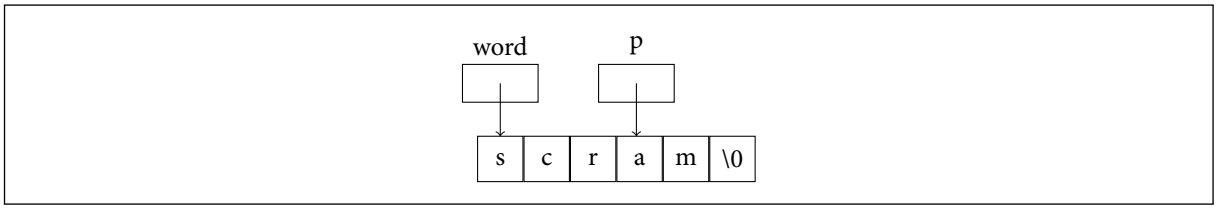
C 语言中的任何数组都可以被解释为指向其第一个元素的指针，因此字符数组也可以解释为指向首元素的指针，从而可以将字符串作为指针。

给定一个指向字符串值的指针，可以用指针运算来操作字符串的值。

```
int FindFirstVowel(char *word)
{
  char *p;

  for(p = word; *p != '\0'; p++){
    if(IsVowel(*p)) return (cp - word);
  }
  return (-1);
}
```

在基于指针的 FindFirstVowel 实现的 for 循环控制行中，首先初始化指针变量 p 使其指向 word 的基地址，然后沿着字符串逐个向前移动，直到指针指向结束处的空字符为止。

例如，如果对字符串“scram”调用 FindFirstVowel 函数，将会找到第一个元音时返回按元素位置定义的两个指针之间的差——即第一个元音字符的下标。



在 for 循环里，函数会检查每一个字符并判断它是否为元音。如果找到了元音字符，函数就用 p 的当前值减去数组的基地址，从而计算得到该字符的下标。

66.1.3 Abstract Type

在引用字符串中的某一个字符时，就需要注意其表示方法，但是在很多情况下把字符串想象成一个统一的整体，就可以忽略掉很多细节，从而使程序更容易理解。

在扩展库 genlib.h 中定义的抽象数据类型 string 强调字符串是一个在概念上独立的类型。

```
typedef char *string;
```

这样，使用 typedef 定义的 string 和 char * 类型完全相同。

对于编译器来说，上述两种字符串类型的意义是完全相同的，但是两种类型名表达的是信息是不同的。

- 如果将一个字符串变量声明为 char * 类型，说明其底层表示方法为指针。
- 如果将一个字符串变量声明为 string 类型，说明将其作为整体看待。

66.2 String Parameter

从调用程序的角度来看，FindFirstVowel 函数的不同实现具有相同的效果，都是由调用程序提供一个待处理的字符串值，由 FindFirstVowel 函数返回一个对应于第一个元音位置的整数。

根据实现原理的不同，FindFirstVowel 函数具有不同的原型，而且相应的字符串处理的方法也是不同的。

- 在 FindFirstVowel 函数的原始实现版本中是将字符串作为抽象数据类型，并通过 strlib.h 中定义的函数来引用单个字符。

  ```
  int FindFirstVowel(string word);
  ```

- 在 FindFirstVowel 函数的数组实现版本中，将字符串作为字符数组来看待，因此会在 for 循环中使用数组下标选择各个字符。

  ```
  int FindFirstVowel(char word[]);
  ```

- 在 FindFirstVowel 函数的指针实现版本中，将字符串作为指针来看待，并在 for 循环中使用指针自增运算来引用单个字符。

  ```
  int FindFirstVowel(char *word);
  ```

对于 C 语言编译器而言，上述三个函数原型中的参数声明是完全一样的，可以互换使用。

字符串可以从不同的角度来考虑，选择某个表示方法的价值在于这种表示方法传达的额外信息。通常情况下，声明的形式参数应该和它们的使用风格一致。

- 如果把字符串作为一个字符数组考虑，并用方括号选择每个字符，应该将该字符串声明成一个数组。
- 如果把字符串作为一个指针考虑，并用 * 间接引用这个指针来选择每个字符，应该将该字符串声明为一个指针。
- 如果把字符串作为一个完整实体来考虑，而且这个完整实体比其各个组成部分更重要，最好的方法是将该字符串声明为抽象数据类型 string。

第 67 章

String Abstractions

以程序设计中更抽象的概念来看，可以采用更高级方法通过一系列各个层次上的字符串操作的抽象实现验证整体论。

不同的抽象形成一个层次结构，底部是最原始的功能，每个新的抽象都是建立在前一层抽象的基础之上，进而提供字符串概念的更复杂的视角。

建立在不同层次结构上的抽象称为分层抽象（layered abstraction），用来表示字符串的分层抽象结构如下图所示：

完成输入输出的硬件设备自动在 ASCII 代码和屏幕或键盘上的符号之间进行转换。计算机对表示字符的整数代码应用算术运算可以处理每个字符，这些功能组成了可用于字符串的机器级的操作，形成层次结构中的最低一层。

在由硬件提供的基本功能的上面，程序设计语言通常也包括一些对字符串操作的支持。可用于字符串的内嵌操作形成了层次结构中的第二层。在许多语言中，这些功能是非常强大的，可以直接在语言级完成复杂的字符串操作。

然而，ANSI C 在语言本身几乎没有提供任何对字符串的支持。仅有的支持是提供定义字符串常量功能。所有其他的字符串操作都是由库提供的，而且 C 语言中没有定义字符串类型，类型名 string 也不是语言的一部分，而是在 genlib.h 接口中作为扩展定义的。

67.1 String Literal

在 C 语言标准中，字符串常量又被称为字符串字面量（string literal）[1]，其含义是在程序执行过程中保持不变的数据。

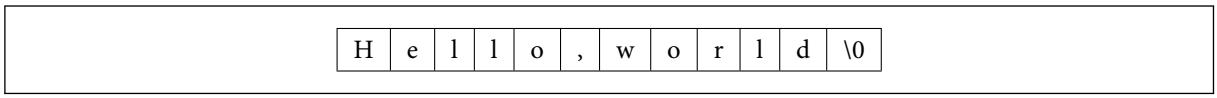
字符串字面量是用一对双括号括起来的字符序列，字符串字面量作为格式串经常被用于 printf 函数和 scanf 函数调用中。

```
#include <stdio.h>

main()
{
  printf("Hello, world.\n");
}
```

字符串变量可以在程序运行过程中发生改变，它并不等同于字符数组，字符串变量使用特殊的空字符\0 来标示字符串的末尾。

只包含一个字符的字符串字面量与字符常量是不同的。

[1]在 C++ 语言中，字符串字面量常被称为字符串字面值，或称为常值，或称为字面量。



- 单字符的字符串字面量是用指针来表示的，这个指针指向存放该单个字符(以及后面的空字符)的内存单元。
- 字符常量是用整数(字符的 ASCII 码)来表示的。

```
const char ch = 'a';
char ch[] = "a";
char *p;
p = ch;
```

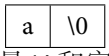


图 67.1: 字符常量 'a' 和字符串字面量 "a"

C 语言不允许在需要字符串的时候使用字符(或者反之亦然)，因此下面的 printf 函数调用是合法的。

```
printf("\n"); /* illegal */
```

但是，使用 printf 来直接输出字符常量是不合法的。

```
printf('\n'); /* wrong */
```

- 如果要使用 printf 函数输出字符，需要使用%c 转换说明。

```
printf("%c", '\n'); /* legal */
```

- 如果要使用 printf 函数输出字符的 ASCII 码值，需要使用

```
printf("%d", '\n'); /* legal */
```

按照 C 语言的标准，编译器必须最少支持 509 个字符长的字符串字面量，许多编译器会允许更长的字符串字面量。

67.1.1 Escape Sequence

字符串字面量可以像字符常量一样包含转义序列。

在字符串字面量中包含八进制和十六进制的转义序列也是合法的。

八进制的转义序列在 3 个数字之后结束，或者在第一个非八进制数字字符处结束。例如，字符串"\1234"包含 2 个字符(\123 和 4)，而字符串"\189"包含 3 个字符(\1、8 和 9)。另一方面，十六进制的转义序列则不限制为 3 个数字，而是直到第一个非十六进制数字字符结束。

现代计算机通常把十六进制数的转义序列限制在\x0 ～\x7f(或可能为\x0 ～\xff)范围之内。

67.1.2 Extend String

如果字符串字面量太长而无法放置在单独一行内，C 语言允许使用\进行截断，从而在下一行延续字符串。除了换行符，在同一行不可以有其他字符跟在\后面。

\并不只是用来截断字符串，还可以用来分割其他任何长的符号。

使用\截断字符串时，字符串字面量必须从下一行的起始位置继续，这样就破坏了程序的缩进风格，因此在标准 C 语言中引入了更好的处理长字符串字面量的方法。

根据 C 语言标准，当两个或更多个字符串字面量相连时(仅用空白字符分割)，编译器必须把它们合并成单独一个字符串，从而就可以把字符串分割后放在两行或更多行中。

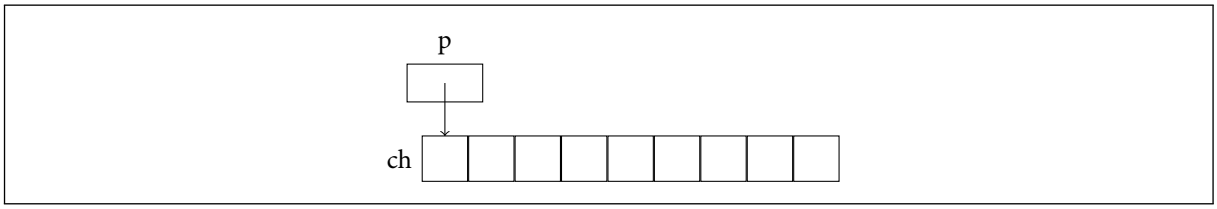
```
printf("How are you. "
      "Fine, thank you.");
```

67.1.3 String Storage

在 stdio.h 提供的 printf 和 scanf 函数调用中会使用字符串字面量作为参数，这里就需要了解 C 语言中字符串字面量的存储。

```
char ch[5] = " ";
char *p = ch;
```

从本质上而言，C 语言把字符串字面量作为字符数组来处理，这样当把字符串存储到内存中时，字符串中的字符就都被分别存储到连续的字节空间中去。

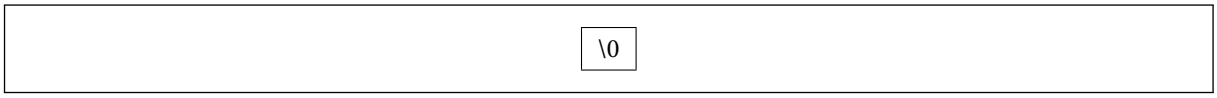


实际上，存放字符本身并不足以表示所有与字符串有关的重要信息，操作字符串的程序必须具有某种确定字符串结束的方法。

当 C 语言编译器在程序中遇到长度为 m 的字符串字面量时，它会为字符串字面量分配长度为 m +1 的内存空间，用以存储字符串字面量中的字符以及代表字符串结束的空字符\0。

空字符被编译器放置在字符数组的最后，可以表示上一个字符串的结束，以及下一个字符串的开始。

- 空字符用来标志字符串的结束，它是 ASCII 字符集中真正的第一个字符。
- 空字符的 ASCII 码值为 0，而零字符的 ASCII 码值为 48，因此用转义序列\0 来表示空字符。
- 字符串字面量可以为空，因此 “ ” 可以作为单独一个空字符来存储。



字符串字面量作为数组来存储，编译器会把它转换为 char * 类型的指针。例如，printf 函数和 scanf 函数都接收来自 char * 类型的值作为它们的第一个参数。

```
#include <stdio.h>

int printf(const char *format, ...);
int scanf(const char *format, ...);
```

当通过下面的语句调用 printf 函数时，会传递 “abc” 的地址(即指向字母 a 的存储单元的指针)。

```
printf("abc");
```

从 C 语言标准库函数 printf 和 scanf 的原型可以看出，它们都需要 char * 类型的变量作为它们的第一个实际参数，这也使用字符串变量代替字符串字面量作为实际参数成为可能。

在下面的示例中，将格式串声明为字符串变量，从而可以把格式串作为实际参数传递给 printf 函数。

```
char fmt[] = "%d\n";
int i;

printf(fmt, i);
```

67.1.4 String Operation

通常情况下，可以在任何允许使用 char * 类型指针的地方使用字符串字面量。例如，字符串字面量可以出现在赋值运算符的右边。

```
char *p;
p = "abc";
```

上述语句不是复制 “abc” 中的字符，而仅仅是将 p 指向字符串中的第一个字符，但是这样就可以通过指针来改变字符串字面量中的字符。

```
char *p = "abc";
*p = 'b'; /* string literal is now bbc */
```

使用指向字符串字面量的指针来直接修改字符的做法是不推荐的，在某些编译器中可能会导致程序程序运行异常。

C 语言允许对指针添加下标，因此可以给字符串字面量添加下标。

```
char ch;
ch = "abc"[1];
```

这样将把新值——字符 b 赋给 ch，其他可能的下标是 0（字符 a），2（字符 c）和 3（空字符）。

下面的函数可以用来把 0 ～ 15 的数转换成等价的十六进制的字符形式。

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

67.2 String Constant

字符串字面量不同于“字符串常量”。

C 语言允许通过指针访问字符串字面量，因此没有限制程序修改字符串字面量中的字符，但标准 C 禁止修改字符串字面量。

一些编译器试图通过存储相同字符串字面量的单独一份副本来节约内存。考虑下面的例子：

```
char *p = "abc", *q = "abc";
```

在这种情况下，某些编译器只存储 “abc” 一次，并且把 p 和 q 都指向该字符串字面量。如果试图通过指针 p 改变 “abc”，那么 q 所指向的字符串也会受到影响，而且还降低了程序的可移植性。

67.3 String Variable

在 C 语言中，如果声明了一个指针变量，其作用就会和任何其他变量的作用相似。从概念上说，它代表了一个已命名的盒子，用于保存其他值的地址。

从一方面来看，指针变量与普通变量一样，因此可以用赋值语句给指针变量赋值。

```
char *cptr;
cptr = NULL;
```

在上述的语句中，可以将 NULL 赋给对应于变量 cptr 的内存单元，这种情况下的指针变量的行为和其他基本类型的变量是一样的。

从另一方面来看，指针变量与数组变量则完全不同。从概念上来说，数组声明指示编译器为数组中的每个元素都分配相应的内存单元。

数组名不是和某个单独的内存单元对应，而是和整个内存单元集合对应，因此数组名不是简单变量，无法和简单变量一样操作数组。

特别地，数组名在 C 语言中不是左值，不能出现在赋值运算的左边，把值赋给任何数组名都是非法的，只能将值赋给单个数组元素。

如果把一个字符串声明为字符数组，就不能直接将字符串赋值给该数组。

```
char carray[6];
carray = "Hello"; /* wrong */
carray[0] = 'H'; /* right */
carray[1] = 'e'; /* right */
carray[2] = 'l'; /* right */
carray[3] = 'l'; /* right */
carray[4] = 'o'; /* right */
carray[5] = '\0'; /* right */
```

数组不是左值对于字符串操作有很重要的意义，不能将字符串字面量直接赋值给数组名，但是将字符串字面量声明为指针就可以解除这一限制。

```
char *cptr;
cptr = "Hello";
```

同样地，调用函数时也有类似的限制，尽管函数可以将一个数组作为参数，但是在 C 语言中从函数返回一个数组是非法的。但是，C 语言允许函数返回指针，因而可以通过让函数返回一个字符指针来得到和返回数组同样的效果。

如果在函数里使用动态分配为某个数组分配内存，可以将指向该内存的指针作为函数的值返回，将结果赋给一个指针变量，接着就可以和使用数组一样使用指针变量。

举例来说，strlib 库中的函数 CharToString 可以用如下方式实现。

```
string CharToString(char ch)
{
  char *cptr;

  cptr = GetBlock(2);
  cptr[0] = ch;
  cptr[1] = '\0';
  return (cptr);
}
```

在 CharToString 函数体中，首先从堆(而不是函数的栈帧)中分配了两个字节的内存：一个字节给字符，另一个字节给每个字符串末尾的空字符。

在接下来的语句中，把对应的值分别赋给这两个字节，最后将一个指向新分配内存的指针作为函数值返回。

假设 test 被声明为字符指针，或者(等同地)被声明为抽象数据类型 string，下面的 CharToString 函数调用语句是合法的。

```
test = CharToString('*');
```

上述的实现只适用于动态分配数组空间的情况。如果要返回一个指向在函数中声明的数组的指针，程序将会失败，原因在于分配给声明成局部变量的数组的内存在函数返回时就自动释放了。

如果函数返回的是一个指向函数帧内声明的内存，数组变量的内容在函数返回时是正确的，但是分配给数组的内存位于函数的栈帧中。

```
string CharToString(char ch)
{
  char carray[2];

  carray[0] = ch;
```

```
    carray[1] = '\0';
    return (carray);
}
```

在上述的示例代码中，当函数返回时会自动释放栈帧中所有的内存，导致指针指向的内存的内容会丢失。

> 将指针作为函数结果返回时，要确保指针指向的内存地址不是当前栈帧的一部分。当前栈帧中的内存会在函数返回时自动释放，所以客户不能再使用。

只要在程序中使用字符串，就必须确定声明保存字符串的变量的方式。将字符串变量声明为字符数组和指向首字符的指针具有的不同的效果。

- 可以将字符串变量声明为数组，并为每个元素保留内存空间；
- 或者可以将字符串变量声明为指针，只为该指针保留足够的空间，在程序运行时才给字符串中的字符分配空间。

具体来说，字符串变量的存储取决于用来操作字符串的工具，声明成指针的字符串必须先初始化后才能选择其中的字符。

不同的库抽象概念描述了字符串行为的不同模式，从而决定了哪种声明方式能更好地使用内存空间。

- 如果库支持字符串的分配，那么应该采用指针声明方式。
- 如果库要求客户自己分配字符串空间，那么必须保证在调用库导出的函数前内存是可用的。

在大多数情况下，保证字符串内存存在的最简单方法是将字符串变量声明成数组，而有效使用字符串的关键在于确保字符串变量的声明方式符合相应的库规定。

67.3.1 String Declaration

现代编程语言中的大多数都为声明字符串变量提供了 string 数据类型，但是 C 语言采取了不同的方式，只要保证字符串是以空字符结尾，任何一维的字符数组都可以用来存储字符串。

尽管在 C 语言中数组和指针是可以互换使用的，但仅限于具有相同意义的形式参数的情况，否则将变量声明为数组和将其声明为指针是截然不同的。对于所有其他的变量类型来说，使用的声明方式决定了其内存的分配方式。

通过逐个字符的搜索空字符可以快速确定字符串的长度，在字符串处理函数时需要声明字符数组来存储字符串时，一定要在字符串的实际字符数基础上加 1 才能正确地处理空字符。

```
#define STR_LEN 10
char str[STR_LEN+1];
```

通过将声明字符串长度的宏加 1 来强调空字符，也就是说当声明用于存放字符串的字符数组时，始终要保证数组的长度比字符串的长度多一个字符。

如果没有给空字符预留位置，可能会导致程序运行时出现不可预知的结果，因为 C 语言的标准库函数都假设字符串都是以空字符结束的。

在上述声明数组的过程中，数组长度是确定的，内存也就是显式分配的，从而可以直接给数组元素赋值，或者将数组传递给其他函数，这样就可以自由操作分配到的内存。

如果换一种方式，将一个变量声明为指向字符的指针 cptr，或者声明为等同的 string 类型时，实际上并没有为字符分配内存。

```
char *cptr;
```

变量 cptr 只是一个字符指针，而且没有经过显式的初始化，所以其内容也是不确定的。

```
typedef char *string;
string cptr;
```

声明一个指针变量或字符串变量时，必须确定已对它进行了初始化，这样才能让它指向内存的真实地址。未成功初始化指针是造成错误的常见原因，这些错误是很难发现的。

未初始化的指针变量的值是不确定的，它可能是某种系统的默认值，或者是任何以前此位置被使用时留下的随机值，这可能会导致关键数据受到破坏并使程序崩溃。

编译器无法捕捉未初始化指针导致的错误，因此最好是通过给指针赋值来完全避免指针未初始化的问题。

67.3.2 String Initialization

声明字数数组后并不是意味着它始终包含特定长度的字符串，字符串的长度取决于空字符的位置，并不受用于存放字符串的字符数组的限制。例如，长度为 STR_LEN+1 个字符的数组可以存放多种长度的字符串，范围从空字符串到长度为 STR_LEN 的字符串。

字符串变量可以在声明时进行初始化。

```
char date[] = {"May 12"};
char date[] = "May 12";
```

编译器将把字符串 “May 12” 中的字符复制到数组 date 中，然后追加一个空字符从而可以使 date 可以作为字符串使用，初始化语句中的大括号可以省略。

字符串变量和字符串字面量是不同的，实际上也可以把字符串变量的初始化式写成：

```
char date[] = {'M', 'a', 'y', ' ', '1', '2', '\0'};
```

大体上来说，字符串初始化行为与 C 语言处理数组初始化式的方法一致。

- 如果字符串初始化式太短以至于无法填满字符串变量的空间，编译器会增加空字符以补齐字符串。
- 如果数组的初始化式比数组本身短时，会把不足的数组元素空间初始化为 0。

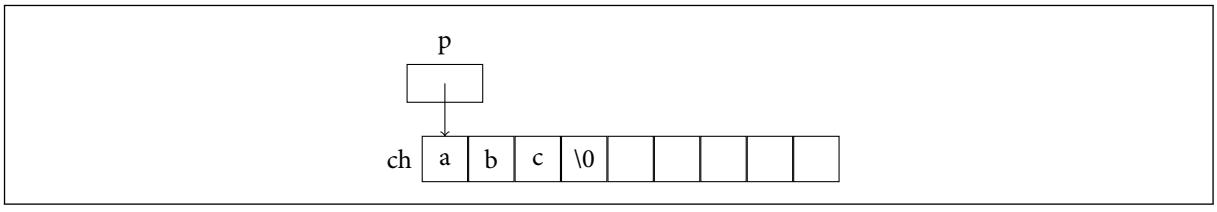
因此，在把字符数组额外的元素初始化为\0 这一点上，编译器对字符串和数组遵循相同的规则。

初始化字符串变量并获得内存的一般策略是采用标准库函数 malloc 或扩展的 GetBlock 函数对内存进行动态分配。动态分配与静态初始化的区别在于字符的内存是从堆(heap)中动态分配来的。

```
char *p;
p = (char *) malloc(9 * sizeof(char));
p[0] = 'a';
p[1] = 'b';
p[2] = 'c';
p[3] = '\0';
```

使用 malloc 等函数动态分配的内存和数组请求分配的内存的使用没有区别，而且只要没有引用超出已分配内存区域界限的元素，都可以安全地引用各个元素。

- 通俗地说，静态初始化就是在定义字符串时由操作系统为字符串分配内存空间。
- 动态数组(普通数组和字符数组等)在定义时只是指针，不能被初始化，需要使用 malloc 等函数来为其动态分配内存。



在下面的声明中，字符数组长度为 9，而字符串初始化只有 6 位，因此余下的字符空间将会被初始化为\0。

```
char Date[9] = "May 12";
char Date[9] = {'M', 'a', 'y', ' ', '1', '2', '\0', '\0', '\0'};
```

如果初始化式比字符串变量长,这对字符串而言是非法的,同样对数组初始化也是非法的。

C 语言允许初始化式(不包括空字符)与变量有完全相同的长度。实际上,在没有空间给空字符时,编译器也不会试图存储一个空字符。

```
char date[6] = "May 12";
char date[6] = {'M', 'a', 'y', ' ', '1', '2'};
```

当字符数组用于存储字符串时,如果无法确保字符数组的长度长于字符串初始化式的长度,导致空字符丢失或编译器忽略空字符,都将使得数组无法作为字符串使用。

> 为空字符预留空间(并实际在字符数组中存储一个空字符)的情况只针对于计划调用以空字符结尾的字符串的函数情况。

在使用 C 语言进行实际编程时,并不是所有的字符数组都作为字符串使用,因此在每个字符数组中都包含空字符的空间并不是必需的。

M	a	y		1	2

如果只对独立的字符进行处理,那么就不需要空字符,这样对这个字符数组唯一可以执行的操作就是使用下标,不能将其看成是字符串,也不能对其执行任何字符串操作。

字符串变量的声明可以忽略长度,编译器可以自动计算长度。

```
char date[] = "May 12";
char date[] = {'M', 'a', 'y', ' ', '1', '2', '\0'};
```

但是,不指明字符数组的长度并不意味着后续可以改变数组的长度,因为在程序编译完毕后数组的长度就固定了。如果初始化式很长,那么忽略字符数组的长度可以防止可能的手工计算错误。

67.3.3 String Pointer

在声明字符串变量时,根据数组和指针的关系,同样可以采用数组形式和指针形式。

下面的声明表明 date 是字符数组。

```
char date[] = "May 12";
char *cptr;
cptr = date;
```

与数组形式声明相似的指针形式声明则表明 date 是指向字符串变量的指针。

```
char *date = "May 12";
```

在上述两种声明中,都是将某一个已经存在的字符数组的地址赋给指针变量,因此都可以作为字符串使用。

任何期望传递字符数组或字符指针的函数都可以接收数组名或指针作为参数,但是二者之间具有显著的差异,不可以互换。

- 声明为字符数组时,可以修改其中的字符。但是,声明为指向字符串字面量的指针时,不能修改字符串字面量中的字符。
- 声明为字符数组和字符串指针时,分别是数组名和指针变量,因此指针变量可以在程序执行期间指向其他字符串。
- 如果需要可以修改的字符串,就需要建立字符数组来存储字符串,不能声明为字符串指针。

下面的声明使编译器为指针变量分配了足够的内存空间。

```
char *p;
```

但是，编译器无法为指针指向的字符串的分配空间，而且在使用 p 作为字符串之前，必须把 p 指向字符数组。

C 语言允许把指针变量指向已经存在的字符串变量，下面的声明将把指针 p 指向 str 的第一个字符，从而可以将 p 作为字符串使用。

```
char str[STR_LEN], *p;
p = str;
```

在 C 语言中，使用未初始化的指针变量作为字符串是非常严重的错误。在下面的例子中，将使用未初始化的指针 j 来创建字符串 “abc”。

```
char *j;

j[0] = 'a'; /* wrong */
j[1] = 'b'; /* wrong */
j[2] = 'c'; /* wrong */
j[3] = '\0'; /* wrong */
```

这里，指针 j 没有初始化，编译器不知道它指向的地址，因此把字符 a、b、c 和\0 存入 j 所指向的内存将会对程序产生无法预期的影响，可能导致程序崩溃或行为异常。

67.4 String I/O

与输出字符串相比，读入字符串的麻烦在于读入的字符串可能比用来存储字符串的变量更长。

67.4.1 String Output

可以使用 printf 函数或 puts 函数来输出字符串，相应的转换说明是%s。

```
char str[] = "Hello, world.";
printf("%s", str);
puts(str);
```

printf 函数会逐个输出字符串中的字符直到遇到空字符为止。如果空字符丢失，printf 函数会越过字符串的末尾继续输出，直到最终在内存的某个地方找到空字符为止。

如果只是显示字符串的一部分，可以使用%.ps 转换说明，这里的 p 代表要显示的字符数量。

为了使字符串在指定域内显示，C 语言提供了%ms 转换说明来指明要在大小为 m 的域内显示字符串。

- 对于超过 m 个字符的字符串，printf 函数并不会不会截断字符串，而是会全部显示出整个字符串。
- 如果字符串长度小于 m，则会在域内右对齐输出。
- 如果要强制左对齐输出字符串，可以在 m 前加一个减号。

m 和 p 可以组合使用，%m.ps 会时字符串的前 p 个字符在大小为 m 的域内显示。

C 函数库提供的 printf 函数和 puts 函数也可以用来输出字符串，可以按如下的方式使用 puts 函数。

```
char str[] = "\n";
puts(str);
```

其中，puts 函数只有一个参数，也就是需要显示的字符串，参数中没有格式串。

使用 printf 函数不提供格式串而直接输出字符串变量是危险的。如果字符串变量中包含字符%，printf 函数会把% 作为转换说明的开始，那么就不会获得预期的结果。

使用 puts 函数输出字符串结束时总会添加一个额外的换行符，使光标移动至下一输出行的开始处，因此上述示例会输出两个空行。

67.4.2 String Input

为了一次性读入字符串，可以使用 scanf 函数或 gets 函数，也可以以每次一个字符的方式来读入字符串。

使用转换说明%s 来让 scanf 函数读入字符串。

```
char str[STR_LEN+1];
scanf("%s", str);
```

在调用 scanf 函数时，不需要在 str 前添加 & 运算符，编译器会自动把数组名转换为指针来处理。

scanf 函数在读入字符前会跳过前面的所有空白字符，然后把读入的字符存储到 str 数组中，直到再遇到空白字符为止，因此使用 scanf 函数读入字符串时永远不会包含空白字符。

通常情况下，scanf 函数无法读入一整行输入，换行符、空格符或制表符都会使 scanf 函数停止读入。

为了每次读入一整行输入，可以使用 gets 函数。

- gets 函数和 scanf 函数都会把读入的字符存储到数组中。
- gets 函数和 scanf 函数都会在读入的字符串末尾添加一个空字符。

另外，在其他方面 gets 函数则不同于 scanf 函数。

- gets 函数不会在开始读入字符串之前跳过空白字符(scanf 函数会跳过)；
- gets 函数会持续读入字符直到找到换行符为止(scanf 函数会在任意空白字符处停止)。
- gets 函数会忽略掉换行符，不会将其存储到数组中，并用空字符代替换行符。

下面的示例显示了 scanf 函数和 gets 函数之间的差异。

```
#include <stdio.h>

#define STR_LEN 10

main()
{
  char str[STR_LEN+1];

  printf("Enter a test string: ");
  scanf("%s", str);
  printf("%s\n", str);

  printf("Enter the test string again: ");
  gets(str);
  printf("%s", str);
}
$ gcc -c test.c
$ ./a.out
Enter a test string: How are you doing?
-> How
Enter the test string again:
-> are you doing?
```

在把字符读入数组时，scanf 函数和 gets 函数都无法检测何时填满数组，因此它们可能越过数组的边界存储字符，这将导致程序行为异常。

一般情况下，可以使用转换说明%ns 来代替%s 来使 scanf 函数更安全，这里的数字 n 指出可以存储的最大字符的数量。

相比 scanf 函数和 printf 函数，gets 函数和 puts 函数实现简单，通常运行速度也更快，但是它们天生就是不安全的，因此 fgets 函数和 fputs 函数等是更保险的选择。

67.4.3 Character I/O

scanf 函数和 gets 函数都有风险且不够灵活，因此实际开发中可能需要开发自定义的输入函数来用每次一个字符的方式读入字符串，从而可以提供比标准输入函数更大程度的可控性。

如果要开发自定义输入函数，下面是一些需要考虑的问题。

- 在开始存储字符串之前，函数应该跳过空白字符码？
- 决定什么字符会导致函数停止读入：换行符、任意空白字符还是其他某些字符，需要存储这类字符还是忽略掉。
- 如果无法存储过长的字符串，程序应该执行什么操作：忽略额外的字符，还是将它们留给下一次的输入操作。

假定自定义输入函数不会跳过空白字符，在第一个换行符处（不把换行符存储到字符串中）停止读取，并且忽略额外的字符，由此函数将具有如下的原型。

```
int read_line(char str[], int n);
```

其中，str 表示用来存储输入的字符数组，而且 n 是最大能读入字符的数量。如果输入行包含的字符多于 n，read_line 函数将忽略多余的字符。

read_line 函数的返回值是实际存储在 str 中的字符数量（从 0 到 n 之间的任意数），不可能总是需要 read_line 函数的返回值，但是有这个返回值也没问题。

在 C 语言标准库中，有大量函数的值并例行公事的丢掉了，尽管 read_line 函数经常被设计为过程，这里的实现仍然保留函数的返回值以供调用。

read_line 函数的主要构造是一个循环，只要 str 仍有空间，那么该循环就继续逐个读入字符并把它们存储起来，循环直到读入换行符时终止[2]。

下面是 read_line 函数的完整实现。

```
int read_line(char str[], int n)
{
  char ch;
  int i = 0;

  while((ch == getchar()) != '\n')
    if(i < n)
      str[i++] = ch;
  str[i] = '\0'; /* terminates string */
  return i; /* number of characters stored */
}
```

在函数回返之前，read_line 的实现在字符串的末尾放置一个空字符。

scanf 函数和 gets 函数等的标准做法都是自动在读入字符串的末尾放置一个空字符，在实现自定义函数的时候也必须考虑到这一点。

另外，如果在 read_line 函数的实现中 getchar 函数读入字符失败，可能的原因还包括读入到达了文件末尾，此时 getchar 函数返回的是 int 型的值 EOF。

下面是改进后的 read_line 函数，增加了用来检测 getchar 函数的返回值是否为 EOF 的判断条件。

```
int read_line(char str[], int n)
{
  int ch;
  int i = 0;

  while((ch = getchar()) != '\n' && ch != EOF)
    if(i < n)
      str[i++] = ch;
  str[i] = '\0';
```

[2]严格地说，如果 getchar 函数读入字符失败，也应该终止循环，但是在当前的实现中忽略这种复杂情况。

```
        return i;
}
```

67.4.4 String Selection

C 语言中的字符串是以字符数组的形式存储的，可以用数组下标来访问特定的字符。例如，为了对字符串 s 中的每个字符进行特定的处理，可以设定一个循环来对计数器 i 进行自增操作，并且通过表达式 s[i] 来选择字符。

如果需要设计一个函数来统计字符串中空格的数量，利用数组下标的实现如下：

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for(i = 0; s[i] != '\0'; i++)
    if(s[i] == ' ')
      count ++;
    return count;
}
```

在字符数组 s 的声明中包含 const 用以表明 count_spaces 函数不会修改数组。

- 如果 s 不是字符串，count_spaces 函数将需要第 2 个参数来指明数组的长度。
- 当 s 是字符串时，count_spaces 函数可以通过测试字符是否为空字符来定位 s 的末尾。

如果使用指针来跟踪字符串中的当前位置，在处理字符数组时会更方便。

下面使用指针来重新实现 count_space 函数，这样就是可以使用 s 来跟踪字符串中的位置。

```
int count_spaces(const char *s)
{
  int count = 0;

  for(; *s != '\0'; s ++)
    if(*s == ' ')
      count ++;
    return count;
}
```

这里使用 const 并不会阻止 count_spaces 函数对 s 的修改，它的作用是阻止函数改变 s 所指向的字符，这样参数 s 在传递给 count_spaces 函数时都是指向字符串的第一个字符。

另外，s 是传递给 count_spaces 函数的参数的副本，因此对 s 进行自增操作不会影响参数。

在实现 count_spaces 函数的过程中提出了一些关于如何编写字符串处理函数的问题。

1. 只要使用方便，可以随意使用数组或指针操作字符串中的字符，或者混合使用这两种方法。
   使用指针实现字符串处理函数时，可以不再需要局部变量，因此直接使用指针可以简化函数。
   从传统意义上来说，更倾向于使用指针来处理字符串。
2. 在字符串处理函数中，将形式参数声明为数组或指针没有区别。
   在编译时，编译器会自动把作为形式参数的数组转换为指针来处理。
3. 形式参数的形式(s[] 或 *s)不会对实际参数的应用产生影响。
   当调用字符串处理函数时，实际参数可以是数组名、指针变量或字符串字面量，字符串处理函数不会区分实际参数之间的差异。

第 68 章

String Library

大多数现代编程语言都提供了可以对字符串进行复制、比较、合并、选择子串等类似操作的运算符，但是 C 语言没有提供操作字符串的运算符。

在 C 语言中，字符串被当作字符数组来处理，因此对字符串的限制和对数组一样。特别地，它们都不能用 C 语言内置的运算符来进行复制和比较操作。

如果直接对字符串进行复制或比较会失败。例如，在下面的示例中利用 = 运算符把字符串复制到字符数组会产生错误。

```
char str1[12];
char str2[12];
str1 = "abc"; /* wrong*/
$ gcc -c test.c
warning: assignment makes integer from pointer without a cast [enabled by default]

char str1[12];
char str2[12];
str1 = "abc"; /* wrong*/
str2 = str1; /* wrong */
$ gcc -c test.c
error: incompatible types when assigning to type ‘char[12]’ from type ‘char *’
```

C 语言把对字符串直接进行复制或比较解释为一个指针与另一个指针之间的(非法的)赋值运算，只有使用 = 来初始化字符数组(字符串)是合法的。

```
char str1[12] = "abc"; /* legal */
char str2[12] = "cde"; /* legal */
```

在字符数组的声明中，“=”不是赋值运算符。

在 C 语言中，使用关系运算符或判等运算符来比较字符串是合法的，但是无法产生预期的结果。

```
char str1[12] = "abc";
char str2[12] = "cde";

if(str1 == str2) puts("same pointer");
else puts("different pointer");
```

上述示例把 str1 和 str2 作为指针来进行比较，并不是比较两个数组的内容。很容易看出，str1 和 str2 有不同的地址，所以测试表达式的值永为假。

68.1 strlib.h

为了能用概念上较简单的模型处理字符串，可以引入另一个字符串库，它可以通过 strlib.h 接口访问。

和 genlib.h 和 simpleio.h 一样，strlib.h 接口也是标谁 ANSI C 库的扩展。因此，要把 strlib.h 包含到程序中时，必须要用引号。

```
#include "strlib.h"
```

这里,strlib.h 接口形成了抽象层次结构的最高层,使得字符串操作相对容易。

- 调用 strlib.h 中的函数时,会自动为保存结果分配内存空间。
- 调用 string.h 中的函数时,客户必须明确地为结果分配内存空间。

strlib.h 接口的主要优势在于它能使字符串作为抽象类型来处理,这里可以认为抽象类型(abstract type)是按照它的行为而不是按照它的表示定义的类型。

抽象类型的行为是由能在这种类型的对象上实施的操作来定义的。特定抽象类型的合法操作被称为它的基本操作(primitive operation),被定义为与该类型相关的接口中的函数。这些操作的细节和数据的基本表示都被隐藏在该接口的实现中。不管客户何时要操作某个抽象类型的值,客户必须用该接口提供的函数。

对字符串来讲,现在已经可以实现的基本操作有:

- 在程序中指定字符串常量。
- 用 GetLine 从用户处读入一个字符串。
- 用 printf 在屏幕上显示一个字符串。
- 用 StringEqual 确定两个字符串是否完全相等。

当利用字符串时,你可能需要完成如下的一些操作:

- 确定字符串的长度。
- 在字符串中选择第一个字符,或更通用一些,选择第 i 个字符。
- 把两个字符串连接起来,形成一个较长的字符串。
- 将单个字符转换为由一个字符组成的字符串。
- 抽取字符串的一部分,形成一个较短的字符串。
- 比较两个字符串,确定按字母顺序哪一个出现在前面。
- 确定一个字符串是否包含某一特定的字符或一组特定的字符。

上面这些需求中每一个操作都是由 strlib.h 接口中的函数提供的,该接口为程序员提供了使用字符串所需要的工具,而不需要理解表示的细节问题,而隐藏实现细节正是数据抽象的初衷。

在初期,每个函数都是由先给出它的原型,然后描述它的操作的模式来引入的,接着必须继续提供包括每个参数类型的完整的原型。

在强调抽象的高级设计中,引入新函数时将采用称为隐含原型(implicit prototype)的方式描述,隐含原型使我们可以很容易看懂如何使用这个函数。

例如,RandomInteger 的隐含原型为

```
RandomInteger(low, high);
```

这种利用隐含类型介绍新的函数,然后按照参数的名字介绍函数的作用的描述风格大量应用于复杂的程序设计过程中。

我们不从每个函数的作用谈起,而是关注要完成的抽象操作,描述完成这些操作所需要的函数。隐含原型是一个函数调用的实例,包括参数的名字。函数都可以用隐含类型引入,然后按照参数的名字介绍函数的作用。

68.1.1 String Length

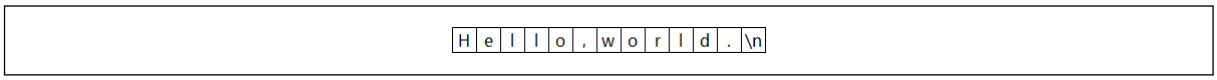
当写一个操作字符串的程序时,经常需要知道特定的字符串包含几个字符。字符串包含的字符总数(包括所有的字母、数字、空格、标点符号和特殊字符)被称为字符串的长度(length)。

使用 strlib.h 接口,调用 StringLength(s) 函数便可以得到字符串的长度。例如,要得到如下字符串的长度:

```
"Hello, world.\n"
```

这个串的长度是 14,单词 Hello 中有 5 个字符,world 中又有 5 个字符,还包括两个标点符号(一个逗号和一个句号),一个空格以及一个换行字符。

因此,函数调用



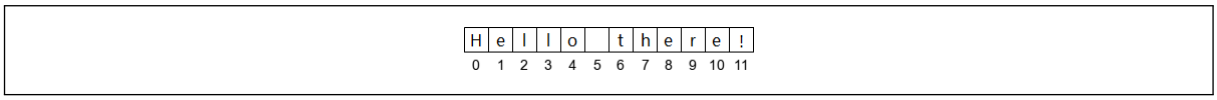

```
StringLength("Hello, world.\n")
```

将返回 14。下面的程序从用户那里读入一行文本,并报告它的长度。

```
main()
{
  string str;
  printf( "This program tests the StringLength function.\n" );
  printf( "Enter a string:" );
  str = GetLine();
  printf( "The length is %d.\n" , StringLength(str));
}
```

68.1.2 Selecting Character

C 语言中,字符串中的位置是从 0 开始编号的。例如,在字符串"Hello there!"中,每个字符的编号如下图所示:



写在字符串的每个字符下面的正数称为它在字符串中的下标(index)。

为了使程序员能按给定的下标在字符串中选择特定的字符,strlib.h 接口提供了一个函数 IthChar,它有两个参数,一个字符串和一个表示下标的整数,该函数返回一个字符。例如,如果变量 str 包含字符串 "Hello there!",调用 IthChar(str, 0) 将返回字符 'H'。同样,调用 IthChar(str, 5) 将返回 ' ',这是一个空格。

> C 语言中字符的编号是从 0 开始,而不是从 1 开始。如果忘记这条规则,就会假设 IthChar(str, 5) 返回的是字符串中的第 5 个字符,实际上 IthChar (Str, 5) 返回的是下标位置为 5 的字符,也就是字符串中的第 6 个字符。

给出了库函数 IthChar 之后,可以定义一个新的函数 LastChar(str),它返回 str 中的最后一个字符,如下所示:

```
char LastChar(string str)
{
  return (IthChar(str, StringLength(str) - 1));
}
```

68.1.3 String Concatenation

字符串的另一个有用的函数是 Concat 函数,它将两个字符串连接起来,当中不加任何字符。在程序设计中,这个操作被称为连接(concatenation)。例如,

```
Concat("Hello", "there")
```

的值是十个字符组成的字符串 "Hellothere"。

如果要在两个单词间放一个空格,必须执行另一个连接。Concat 函数每次只能取两个参数,因此为了连接三个或更多的字符串,必须多次调用 Concat。每次调用连接两个字符串。例如,如果变量

word1 包含 “Hello”, 变量 word2 包含 “there”, 为了产生 11 个字符的字符串 “Hello there”, 必须嵌套调用 Concat, 如下列表达式所示:

```
Concat(Concat(word1, " "), word2);
```

也可以用 Concat 定义一个函数 ConcatNCopies(n, str), 它返回一个由 n 个 str 拷贝连接起来的字符串。例如, 调用函数 ConcatNCopies(10, “*”), 则返回一个由十个星号组成的字符串。ConcatNCopies 的实现为:

```
string ConcatNCopies(int n, string str)
{
  string result;
  int i;

  result = “” ;
  for(i = 0; i < n; i ++){
    result = Concat(result, str);
  }
  return (result);
}
```

从某种意义上说, 该例子中的实现策略类似于 Factorial 函数的实现。在这两种情况中, 函数都用一个局部变量记录 for 循环的每个周期中的部分计算结果。在 ConcatNCopies 函数中, for 循环的每个周期连接 str 的值到前一个 result 的后面。因为每个周期加一个 str 的拷贝到 result 的后面, 那么 n 个周期后的 result 的最终值应该是由该字符串的 n 份拷贝组成。

对 Factorial 和 ConcatNCopies 中的每一个函数来说, 用于保存结果的变量的初始化是值得注意的。在 Factorial 函数中, 变量 product 被初始化为 1, 因此随着计算的进行, 把它乘以每个 i 的连续值正好记录了这个结果。

在 ConcatNCopies 的情况下, 对应的语句初始化变量 result, 使得它通过连接而增长。在第一个循环周期后, 变量 result 保存了字符串 str 的一份拷贝。

在第一个循环周期前, result 必须包含该字符串的 0 份拷贝, 即没有任何字符。没有任何字符的字符串称为空串(empty string), 在 C 语言中用一对双引号 (“”) 表示。

当需要通过连接将一系列的片段连接到一个已存在的字符串变量中, 构成一个新的字符串时, 应该把该字符串初始化为空串。

> 常见错误: Concat 函数总是取两个参数。如果需要连接两个以上的字符串, 必须嵌套调用 Concat。最里层的调用连接两个字符串, 下一个调用将另一个字符串加到结果中, 依此类推。

68.1.4 Converting characters

当使用 Concat 函数时, 经常会遇到要将一个字符加入到一个已有的字符串中的情况。Concat 函数看起来似乎是一个合适的工具, 但不是非常适合这种情况。Concert 函数要求它的参数都是字符串。在许多情况下, 我们所要连接的可能是一个字符串和一个字符。为了解决这个问题, strlib.h 库包含了一个函数 CharToString(ch)。这个函数有一个字符型的参数 ch, 返回一个仅由该字符组成的字符串。在将一个字符转换为字符串后, 就可以把它和其他字符串连接起来。

为了理解如何应用这个技术, 假设要写一个函数 ReverseString(str), 该函数返回一个按 str 相反次序排列的新字符串, 因此, ReverseString(“ABC”), 返回 (“CBA”)。

为了实现这个函数, 可以用 for 循环一个个地处理原来的字符串中的字符, 用连接形成一个新字符串。如果从左到右处理, 将原来串中的每个字符加在新串的前面, 那么新的串将以反序出现。

下面的 ReverseString 的实现说明了这个策略:

```
string ReverseString(string str)
{
  string result;
  int i;

  result = "" ;
  for(i = 0; i < StringLength(str); i ++){
    result = Concat(CharToString(IthChar(str, i)), result);
  }
  return (result);
}
```

注意,每个字符在连接到变量 result 的开始处以前,必须被转换成字符串。

68.1.5 Extracting String

连接将几个短的字符串形成长的字符串。我们还经常要做相反的工作:将一个字符串分解为较短的字符串。

一个字符串如果是某一较长的字符串的一部分,则被称为一个子串(substring),strlib.h 库提供了一个函数 SubString(s, p1, p2),它的作用是从 s 中抽取从位置 p1 到 p2 之间的字符,包括 p1 和 p2。因此,函数调用

```
SubString("Hello there!", 1, 3);
```

将返回字符串 "ell"。在 C 语言中,字符编号是从 0 开始,因此下标位置为 1 的字符是 'e'。

作为 SubString 的一个应用实例,函数 SecondHalf(s) 返回一个由 s 中后半部分字符组成的子串,如果字符串长度是奇数,则包括中间的字符:

```
string SecondHalf(string str)
{
  int len;
  len = StringLength(str);
  return (SubString(str, len / 2, len -1));
}
```

SubString 函数对一些特殊情况处理如下:

1. 如果 p1 是负数,它会被设为 0,使它指向字符串的第一个字符。
2. 如果 p2 大于 StringLength(s) - 1,它会被设置为 StringLength(s) - 1,使它指向最后一个字符。
3. 如果 p1 大于 p2,则 SubString 返回一个空串。

虽然在对字符串没有更多的经验前可能不太清楚选择这种设计的理由,但当调用 SubString 时,这些规则使我们不需要检查一些特殊情况,使字符串编程更容易。

68.1.6 Comparing Strings

strlib.h 提供了一个函数 StringEqual 比较两个字符串是否完全相同。另外在许多情况下,确定两个字符串按字母顺序的先后位置也是非常有用的,于是 strlib.h 提供了一个函数 StringCompare 完成这个功能。

StringCompare 有两个字符串参数 s1 和 s2,它返回一个整数,该整数的符号指出了两个字符串的关系:

1. 如果按字母顺序,s1 出现在 s2 的前面,则 StringCompare 返回一个负整数。
2. 如果按字母顺序,s1 出现在 s2 的后面,则 StringCompare 返回一个正整数。
3. 如果两个字符串完全相同,则 StringCompare 返回 0。

因此,如果要确定按字母顺序 s1 是否出现在 s2 的前面,可以用下列语句判断:

```
if(StringCompare(s1, s2) < 0) . . .
```

虽然 StringCompare 函数返回一个整数，但除了它有一个正确的符号之外，对整数的值作任何假设都是不合法的。在某些系统中，StringCompare 的返回值总是-1、0 或 1。在另一些系统中，这个值似乎是带有正确符号的任意整数。

计算机所用的“字母顺序”与某些领域所用的词典的次序是不同的。当用 StringCompare 比较两个字符串时，它用字符代码集赋给它的数值来计算。这个次序被称为词典顺序（lexico-graphic order），与某些领域传统的字母顺序是不一样的。

例如，在某个字母索引中，可能会发现 aardvark 在 Achilles 前面，因为传统的词典不区分大小写字母。如果用参数“aardvark”和“Achilles”调用 StringCompare 函数，该函数只是比较 ASCII 代码。在 ASCII 代码中，小写字母'a' 是出现在大写字母'A' 的后面。按词典顺序，“Achilles”出现在前面。因此，函数调用

```
StringCompare( “aardvark” ,  “Achilles” )
```

返回一个正整数。

当调用 StringCompare 时，它先比较两个字符串的首字符。如果这两个字符不同，StringCompare 考虑这两个字符在 ASCII 序列中的关系，然后返回一个整数指出结果。如果第一个字符相同，StringCompare 继续看第二个字符，继续这个过程，直到检测到两个不同的字符为止。如果 StringCompare 检测到两个字符串中的某一个已经结束，那么这个字符串被认为是在长的字符的前面，就如在传统的字母顺序中一样。例如，

```
String("abc", "abcdefg")
```

将返回一个负整数。

仅当两个字符串完全匹配，长度相同时，StringCompare 才返回 0。在 C 语言中，使用 StringCompare 或 StringEqual 的最大问题并不是判断应该用哪一个函数。这很简单。困难的是要记得使用它们。很容易犯的错误是用传统的关系运算取代这两个函数。如果要知道 s1 是否出现在 s2 的前面，甚至有经验的程序员也可能会写下如下语句，把这个条件表示为

```
if(s1 < s2) . . . /*该语句不会工作*/
```

这种形式的表达式没有起到预期的作用。更糟的是，编译器甚至不告诉你有这个错误，因为这样的表达式对编译器来说是有意义的，只不过不是我们所要的意义。事实上，当把关系运算符用于两个字符串时，返回的布尔值是完全与它们的值无关。

记住，用关系运算符比较两个字符串必定会产生一个错误。避免这个错误将节省许多调试的时间。

> 常见错误：当比较字符串的值时，记住要用 StringEqual 和 StringCompare，不要使用关系运算符。C 编译器检测不到这个错误，但程序会给出一个完全无法预计的结果。

68.1.7 Searching String

有些应用中会需要搜索一个字符串，看它是否包含某一特定的字符或子串。strlib.h 接口提供了两个函数 FindChar 和 FindString 来实现这个功能。FindChar 的原型是

```
int FindChar(char ch, string text, int start);
```

该函数搜索字符串 text，从 start 指定的下标位置开始，寻找字符 ch 的第一次出现。如果发现了该字符，FindChar 返回这个字符的下标位置。如果在 text 结束前没有发现这个字符，FindChar 返回值-1。下面的例子说明了这个操作（还是要记住，下标号从 0 开始）：

```
FindChar('l', ''Hello there'', 0)  (返回2)
FindChar('l', ''Hello there'', 3)  (返回3)
FindChar('l', ''Hello there'', 4)  (返回-1)
```

和字符串比较一样，搜索字符串的函数对大小写是区别对待的。因此，调用

```
FindChar('h', ''Hello there'', 0)  (返回7)
```

返回 7，因为小写字母 h 的第一次出现在下标位置 7。在下标位置 0 的大写字母 H 被忽略了。

可以用 FindChar 实现一个产生首字母缩写(acronym))的函数，它按次序取一系列单词的首字母形成一个新的单词。

例如，单词 scuba 是由 self contained underwater breathing apparatus 的首字母形成的首字母缩写词。函数 Acronym 取一个由若干个独立的单词组成的字符串，返回它们的首字母缩写词。因此，调用函数

```
Acronym("self contained underwater breathing apparatus")
```

将返回“scuba”。

只要单词用单个空格分开，并且没有其他字符出现，Acronym 的实现就非常简单，它取最前面的字母，然后进入循环，搜索空格。一旦发现了空格，就将下一个字符连到保存结果的字符串变量的最后。当字符串中找不到空格时，首字母缩写词就完成了。这个策略可以翻译成下列 C 语言的实现：

```
string Acronym(string str)
{
  string acronym;
  int pos;
  acronym = CharToString(IthChar(str, 0));
  pos = 0;
  while(TRUE){
    pos = FindChar( ‘ ’ , str, pos + 1);
    if(pos == -1) break;
    acronym = Concat(acronym, CharToString(IthChar(str, pos + 1)));
  }
  return (acronym);
}
```

函数 FindString(str, text, Start) 与 FindChar 相似，除了第一个参数是字符串以外。函数搜索字符串 text，从位置 start 开始寻找字符串 str。如果找到匹配的字符串，FindString 返回匹配开始处的下标位置。例如，

```
FindString( “there” , “Hello there” , 0)
```

将返回值 6。

如果没有找到匹配的字符串，和 FindChar 一样，FindString 返回-1。

例如，函数 ReplaceFirst(str, pattern, replacement) 搜索字符串 str，将字符串 pattern 的第一次出现用字符串 replacement 来取代，将整个新串作为函数的返回值返回。如果 pattern 串没有出现，则返回原来的串。

下面所示的程序 repfirst.c 包含了 ReplaceFirst 函数的实现，以及一个测试程序。

```
/*
 * File: repfirst.c
 * ----------------
 * This file implements and tests the function ReplaceFirst.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string ReplaceFirst(string str, string pattern, string replacement);
```

```
/* Main program */

main()
{
  string str, pattern, replacement;

  printf("This program edits a string by replacing the first\n");
  printf("instance of a pattern substring by a new string.\n");
  printf("Enter the string to be edited:\n");
  str = GetLine();
  printf("Enter the pattern string: ");
  pattern = GetLine();
  printf("Enter the replacement string: ");
  replacement = GetLine();
  str = ReplaceFirst(str, pattern, replacement);
  printf("%s\n", str);
}

/*
 * Function: ReplaceFirst
 * Usage: newstr = ReplaceFirst(str, pattern, replacement);
 * ------------------------------------------------------
 * This function searches through the string str and replaces the
 * first instance of the pattern with the specified replacement.
 * If the pattern string does not appear, str is returned unchanged.
 */

string ReplaceFirst(string str, string pattern, string replacement)
{
  string head, tail;
  int pos;

  pos = FindString(pattern, str, 0);
  if (pos == -1) return (str);
  head = SubString(str, 0, pos - 1);
  tail = SubString(str, pos + StringLength(pattern), StringLength(str) - 1);
  return (Concat(Concat(head, replacement), tail));
}
```

68.1.8 Case conversion

strlib.h 库中包含两个函数 ConvertToUpperCase(s) 和 ConvertToLowerCase(s) 用于将任何字母字符转换为指定的大小写。

例如，调用函数

```
ConvertToUpperCase("Hello, world.")
```

将返回字符串“HELLO, WORLD.”。

注意，字符串中的任何非字母的字符(如逗号、空格、句号)不受影响。

与 strlib.h 中接口中的其他函数一样，ConvertToUpperCase 和 ConvertToLowerCase 不改变参数中的任何字符，只是将完整的新串作为函数的结果返回。

因此，要改变存储在字符串变量 word 中的值，使得所有的字母都以小写字母出现，需要用一个赋值语句，如

```
word = ConvertToLowerCase(word);
```

如果只是调用

```
ConvertToLowerCase(word); /*该语句未改变word*/
```

而没有加赋值操作，则 word 中的字符不会改变。

下面用 IthChar 和 Concat 实现 ConvertToLowerCase 函数：

```
string ConvertToLowerCase(string str)
{
  string result;
  char ch;
  int i;

  result = " ";
  for(i = 0; i < StringLength(str); i ++){
    ch = IthChar(str, i);
    result = Concat(result, CharToString(tolower(ch)));
  }
  return (result);
}
```

68.1.9 Numeric conversion

strlib.h 接口导出两个函数，IntegerToString 和 RealToString，它们将一个数值转换为字符串表示。

函数 IntegerToString 将整数 n 转换为一个数字组成的字符串，如果 n 是负数，前面再加一个负号。例如调用 IntegerToString(123) 将返回字符串“123”，调用 IntegerToString(-4) 将返回字符串“-4”。

RealToString(d) 函数将浮点数转换为字符串，该字符串的格式类似于由 printf 用格式码%G 的显示结果。有时会产生一个以科学记数法表示的数字，例如调用 RealToString(3.14) 返回“3.14”，而调用 RealToString(0.00000000015) 将返回“1.5E-10”。

如果要将一个数字的文本表示作为字符序列来处理，那么函数 IntegerToString 和 RealToString 是很有用的。

例如，可以用 IntegerToString 写一个函数 ProtectedIntegerField(n, places)，它生成一个字符串，该字符串包括整数 n 的文本表示，前面有足够的星号，使得整个字符串的长度至少与 places 给定的值一样长。

ProtectedIntegerField 的实现如下所示，它用到了原先定义的 ConcatNCopies 函数：

```
string ProtectedIntegerField(int n, int places)
{
  string numstr, fill;
  numstr = IntegerToString(numstr);
  fill = ConcatNCopies(places –StringLength(numstr), “*” );
  return (Concat(fill, numstr));
}
```

这个函数在检查拼写的应用中非常有用，例如，如果在程序中出现 printf 调用：

```
printf("$%s.00\n", ProtectedIntegerField(123,8) );
```

将产生以下运行结果：

```
$*****123.00
```

这些星号使某些人难以修改这个值。

strlib.h 接口还导出了函数 StringToInteger 和 StringToReal，用于将表示数值的字符串转换为数值。例如，调用 StringToInteger(“42”) 返回整数 42。

同样，调用 StringToReal(“3.14159”) 返回浮点数 3.14159。如果函数的参数不是合法的数字串，将会报告一个错误。

这两个函数主要用于输入操作。作为一个实例，下面 addlist.c 的实现中用空行作为输入结束标记：

```
main()
{
  int total;
  string line;
  printf( “This program add a list of numbers.\n” );
  printf( “Single end of list with a blank line.\n” );
  total = 0;
  while(TRUE){
    printf( “?” );
    line = GetLine();
    if(StringEqual(line, “ ” )) break;
    total += StringToInteger(line);
  }
  printf( “The total is %d\n” , total);
}
```

由于 GetInteger 不能读一个空行作为数据，因此在 addlist.c 中没有将空行作为输入结束标记的一种选择。

在程序通过 strlib.h 引入 GetLine 读入输入值后，空行将作为一个空串出现。如果从用户读入的行不是空行，那么该行中的字符将转换成整数，并加到运行结果中。

68.2 string.h

C 语言的大多数字符串操作都是由 string.h 接口提供的函数来完成的。

通过对字符串操作的分层抽象，string.h 接口代表了某个层次的字符串操作的模型，从而可以从一个全新的角度看待字符串。

相比扩展的 strlib 库，标准 C 的字符串库的不同在于为字符串中的字符分配内存空间的方式。

- 在扩展的 strlib.h 接口中，字符串处理函数本身就会分配必要的内存空间，因此客户无需关注内存分配的细节。
- 在 string.h 接口中，字符串处理函数要求客户手动为需要保存结果的字符串分配内存。

作为 C 标准库的一部分，在 string.h 中定义的函数被强制要求可以在任何支持 C 语言的平台上运行，或者说 string.h 提供了一组用于操作字符串的标准工具。

```
#include <string.h>
```

使用 string.h 接口是 ANSI C 中字符串操作的标准方法，从 <string.h> 中导出的每个函数都至少需要一个字符串作为实际参数。

调用 string.h 接口导出的字符串处理函数时，客户通常需要声明一个保存结果的字符数组，并将其作为参数传递给库函数，然后再把新字符串值写入调用函数提供的内存空间中。

Table 68.1: string.h 导出的常用字符串操作函数

函数	说明
strcpy(dest, src)	将字符从 src 复制到 dest
strncpy(dest, src, n)	最多将 n 个字符从 src 复制到 dest
strcat(dest, src)	将字符串 src 追加到字符串 dest 后面
strncat(dest, src, n)	最多将 n 个字符从 src 追加到 dest 后面
strlen(s)	返回字符串 s 的长度
strcmp(s1, s2)	返回一个整数，表示字符串 s1 和 s2 比较的结果
strncmp(s1, s2, n)	类似于 strcmp，但是最多比较 n 个字符
strchr(s, ch)	返回一个指针，指向 s 中的第一个 ch 的实例(或者返回 NULL)

函数	说明
strrchr(s, ch)	返回一个指针，指向 s 中的最后一个 ch 的实例(或者返回 NULL)
strstr(s1, s2)	返回一个指针，指向 s1 中的第一个 s2 的实例(或者返回 NULL)

把字符串形式参数声明为 char * 类型，同时允许实际参数可以是字符数组、char * 类型变量或者字符串字面量，上述这些都适合作为字符串。另外，如果在调用函数中的字符串形式参数没有限定为 const 存储类型，那么对应的实际参数不应该是字符串字面量。

```
$ man string.h
NAME
     string.h – string operations
SYNOPSIS
     #include <string.h>
DESCRIPTION
     Some of the functionality described on this reference page extends the ISO C standard.
     Applications shall define the appropriate feature test macro to enable the visibility
         of these symbols in this header.

     The <string.h> header shall define the following:
     NULL  Null pointer constant.
     size_t As described in <stddef.h> .

     The following shall be declared as functions and may also be defined as macros.

          void  *memccpy(void *restrict, const void *restrict, int, size_t);
          void  *memchr(const void *, int, size_t);
          int    memcmp(const void *, const void *, size_t);
          void  *memcpy(void *restrict, const void *restrict, size_t);
          void  *memmove(void *, const void *, size_t);
          void  *memset(void *, int, size_t);
          char  *strcat(char *restrict, const char *restrict);
          char  *strchr(const char *, int);
          int    strcmp(const char *, const char *);
          int    strcoll(const char *, const char *);
          char  *strcpy(char *restrict, const char *restrict);
          size_t strcspn(const char *, const char *);
          char  *strdup(const char *);
          char  *strerror(int);
          int   *strerror_r(int, char *, size_t);
          size_t strlen(const char *);
          char  *strncat(char *restrict, const char *restrict, size_t);
          int    strncmp(const char *, const char *, size_t);
          char  *strncpy(char *restrict, const char *restrict, size_t);
          char  *strpbrk(const char *, const char *);
          char  *strrchr(const char *, int);
          size_t strspn(const char *, const char *);
          char  *strstr(const char *, const char *);
          char  *strtok(char *restrict, const char *restrict);
          char  *strtok_r(char *, const char *, char **);
          size_t strxfrm(char *restrict, const char *restrict, size_t);

     Inclusion of the <string.h> header may also make visible all symbols from <stddef.h>.
```

string.h 包含了宏定义、常量以及函数和类型的声明，涉及的内容除了字符串处理之外，还包括大量的内存处理函数，因此也可以说 string.h 这个命名是不恰当的。

string.h 中的常量和类型分别是 NULL 和 size_t，其中：

名称	说明
NULL	表示空指针常量的宏，即表示一个不指向任何有效内存单元地址的指针常量。
size_t	无符号整型（unsigned int），被用于 sizeof 运算符的返回值类型。

实际上，string.h 接口中的概念太抽象，使得某些常用的操作用 string.h 接口难以完成。例如，当使用 string.h 中的函数时，不能很容易地从函数返回字符串值，也不能直接为一个变量赋一个字符串值。

另外，string.h 提供的部分函数存在一些安全隐患（例如缓存溢出等），导致程序员宁愿使用一些更安全的函数而放弃一定的可移植性。同时，这些字符串函数只能处理 ASCII 字符集或兼容 ASCII 的字符集（如 ISO-8859-1），在处理存在多字节字符的字符集（如 UTF-8 时）会产生一个警告，指出对字符串“长度”的计算是以字节而不是以 Unicode 字符为单位。

非 ASCII 兼容字符集的字符串处理函数一般位于 wchar.h 中。

Table 68.2: ANSI C string.h 函数

名称	说明
void *memcpy(void *dest, const void *src, size_t n);	将 n 字节长的内容从一个内存地址复制到另一个地址；如果两个地址存在重叠，则最终行为未定义
void *memmove(void *dest, const void *src, size_t n);	将 n 字节长的内容从一个内存地址复制到另一个地址；与 memcpy 不同的是它可以正确作用于两个存在重叠的地址
void *memchr(const void *s, char c, size_t n);	在从 s 开始的 n 个字节内查找 c 第一次出现的地址并返回，若未找到则返回 NULL
int memcmp(const void *s1, const void *s2, size_t n);	对从两个内存地址开始的 n 个字符进行比较
void *memset(void *, int, size_t);	用某种字节内容覆写一段内存空间
char *strcat(char *dest, const char *src);	在字符串 dest 之后连接上 src
char *strncat(char *dest, const char *src, size_t n);	在字符串 dest 之后连接上 src，最多增加 n 个字符
char *strchr(const char *, int);	从字符串头开始查找某字符出现的位置
char *strrchr(const char *, int);	从字符串尾开始查找某字符出现的位置
int strcmp(const char *, const char *);	基于字典顺序比较两个字符串
int strncmp(const char *, const char *, size_t);	基于字典顺序比较两个字符串，最多比较 n 个字节
int strcoll(const char *, const char *);	基于当前区域设置的字符顺序比较两个字符串
char *strcpy(char *toHere, const char *fromHere);	将一个字符串从一个位置复制到另一个位置
char *strncpy(char *toHere, const char *fromHere, size_t);	将一个字符串从一个位置复制到另一个位置，最多复制 n 个字节
char *strerror(int);	返回错误码对应的解释字符串，参见 errno.h（非线程安全函数）
size_t strlen(const char *);	返回一个字符串的长度
size_t strspn(const char *s, const char *strCharSet);	从字符串 s 的起始处开始，寻找第一个不出现在 strCharSet 中的字符，返回其位置索引值
size_t strcspn(const char *s, const char *strCharSet);	从字符串 s 的起始处开始，寻找第一个出现在 strCharSet 中的字符，返回其位置索引值
char *strpbrk(const char *s, const char *strCharSet);	在字符串 s 中查找 strCharSet 中任意字符第一次出现的位置的指针值
char *strstr(const char *haystack, const char *needle);	在字符串 haystack 中查找字符串 needle 第一次出现的位置，heystack 的长度必须长于 needle
char *strtok(char *, const char *);	将一个字符串分隔成一系列字符串；此函数非线程安全，且不可重入

名称	说明
size_t strxfrm(char *dest, const char *src, size_t n);	根据当前 locale 转换一个字符串为 strcmp 使用的内部格式

Table 68.3: ISO C 扩展函数

名称	说明
void *memccpy(void *dest, const void *src, int c, size_t n);	在两块不重叠的内存地址间复制内容，直至复制了 n 字节或遇到内容为 c 的字节
void *mempcpy(void *dest, const void *src, size_t n);	memcpy 的变体，返回写入的最后一个字节的地址指针
errno_t strcat_s(char *s1, size_t s1max, const char *s2);	strcat 的变体，带边界检查
errno_t strcpy_s(char *s1, size_t s1max, const char *s2);	strcpy 的变体，带边界检查
char *strdup(const char *);	将字符串的内容复制到一段新分配的内存空间
int strerror_r(int, char *, size_t);	将 strerror() 的结果放入一段给定的内存缓冲，此函数是线程安全的
char *strerror_r(int, char *, size_t);	使用线程安全的方式返回 strerror() 的结果。在必要的时候才使用给定的内存缓冲 (与 POSIX 中的定义不一致)
size_t strlcat(char *dest, const char *src, size_t n);	strcat 的变体，带边界检查
size_t strlcpy(char *dest, const char *src, size_t n);	strcpy 的变体，带边界检查
char *strsignal(int sig);	与 strerror 类似，返回有符号数 sig 对应的错误解释字符串(非线程安全函数)
char *strtok_r(char *, const char *, char **);	strtok 的线程安全且可重入的版本

68.2.1 strcpy

strcpy(字符串复制)函数可以把源字符串 src 复制给目标字符串 dest，它为如何给调用函数返回一个新的字符串值提供了最好的方法。

下面是 strcpy 函数在 <string.h> 接口中的原型。

```
char *strcpy(char *dest, const char *src);
```

准确地说是 strcpy 函数把 src 指向的字符串复制到 dest 所指向的字符数组中，并返回目标数组的地址。为了和赋值运算符相一致，复制操作是从右向左进行的。

- strcpy 函数把 src 中的字符依次复制到 dest 中，直到(并且包括)遇到 src 中的第一个空字符为止。
- strcpy 函数返回 dest(即指向目的字符串的指针)。
- strcpy 函数不会改变 src 指向的字符串。

```
$ man strcpy
NAME
     strcpy, strncpy – copy a string

SYNOPSIS
     #include <string.h>

     char *strcpy(char *dest, const char *src);
     char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
     The strcpy() function copies the string pointed to by src, including the terminating
```

```
       null byte ('\0'), to the buffer pointed to by dest.
       The strings may not overlap, and the destination string dest must be large enough to
       receive the copy. Beware of buffer overruns!

       The strncpy() function is similar, except that at most n bytes of src are copied.
       Warning: If there is no null byte among the first n bytes of src, the string placed
       in dest will not be null-terminated.

       If the length of src is less than n, strncpy() writes additional null bytes to dest
       to ensure that a total of n bytes are written.

RETURN VALUE
       The strcpy() and strncpy() functions return a pointer to the destination string dest.
```

如果需要操作一个新字符串，同时又保留它的原值，strcpy 函数是很有用的。例如，假设字符串变量 line 包含一行从用户处读入的文本，程序要求对 line 中的字符做一些修改，直接使用数组选择来更改 line 中的字符会破坏 line 原来的内容，而且以后还可能会用到这些内容。为了避免改变 line，可以将其中的各个字符复制到另外一个数组后再进行操作。

保存数组中间副本的数组称为缓冲区（buffer），在 stcpy 函数执行过程中使用字符缓冲区来装入源字符串的数据副本。

当使用 ANSI 字符串库时，通常的做法是明确声明一个字符数组作为缓冲区，大小足够存放应用中可能遇到的最大字符串，引入常量 MaxLine 作为最长输入串的长度，从而得到缓冲区的声明。

```
char buffer[MaxLine+1];
```

声明一个比最大字符串的长度大 1 的缓冲区是为了给最大字符串末尾的空字符预留空间。

在使用字符串处理函数时，最不能犯的错误是使用下面的赋值语句将 line 中的字符复制到 buffer。

```
buffer = line; /* wrong */
```

和任何数组一样，字符数组在 C 语言中不是左值，不能出现在赋值符号的左边，只能用 strcpy 等函数来复制字符。

```
strcpy(buffer, line);
```

strcpy 函数调用会将 line 的字符全部复制到 buffer 中，直到遇到表示源字符串结束的空字符为止。strcpy 函数总是将空字符随其他字符一起复制到目标字符串中，从而保证了新字符串的正确终止。

可以使用数组或指针来实现 strcpy 函数，下面的示例是 strcpy 函数的数组实现。

```
void strcpy(char dest[], char src[])
{
  int i;

  for(i = 0; src[i] != '\0'; i++){
    dest[i] = src[i];
  }
  dest[i] = '\0';
}
```

虽然大多数情况下会主动忽略掉 strcpy 函数的返回值，但是某些情况下保留 strcpy 函数的返回值会比较有用。例如，在将 strcpy 函数的调用作为较大表达式的一部分时，就可以使用 strcpy 函数的返回值来达到和多重赋值同样的效果。

```
strcpy(str2, strcpy(str1, "abcd")); /* similar to int a=b=0; */
```

在 C 语言中通常将字符串作为指针来考虑，因此大多数字符串处理函数都是利用指针来实现的。

```
char *
strcpy(char *dest, const char *src)
```

```
{
  while(*dest++ = *src++)
    ;
}
```

在上述的 strcpy 的指针实现中，所有的工作在 while 语句的测试中全部完成，while 语句的循环体只是一个分号，分号本身构成的是 C 语言中合法但不产生任何作用的语句——空语句（null statament）。

该测试首先从源字符串 src 复制一个字符到目标缓冲区 dest，然后更新每个指针使它们指向下一个字符的地址。C 语言将所有非零值解释为 TRUE，因此只要结果不为零，strcpy 函数中的 while 循环就会一直进行下去，直到字符串末尾的空字符被复制时才会终止。

> 在调用 strcpy 函数时，要确保第一个参数指定了一个字符数组，其大小足以保存所有正在复制的字符串值，包括表示字符串结束的空字符。如果不能保证必须的内存空间，应该明确加入捕捉这个错误的代码。

strcpy 函数弥补了不能使用赋值运算符来复制字符串的不足。例如，如果要把字符串 “abcd” 存储到字符串 str1 中，直接使用赋值运算符会遇到下面的错误：

```
char str1[12];
str1 = "abcd";
$ gcc -c test.c
error: incompatible types when assigning to type ‘char[12]’ from type ‘char *’
```

str1 是字符数组名，而且不能出现在赋值运算符的左侧。

```
char str1[12];
str1 = strcpy(str1, "abcd");
$ gcc -c test.c
error: lvalue expected
```

调用 strcpy 函数成功时，返回值是指向目标字符串的指针，指针之间不能进行赋值操作。

在使用 strcpy 函数时，客户的责任是分配足够大的空间来保存目标字符串和表示字符串结束的空字符。如果 strcpy 函数试图写入的字符比客户分配的空间多，程序就会失败，而且原因难以捕捉。

在下面把字符串 “abcd” 复制给字符串 str1 和 str2 的示例中，执行到 strcpy(str2, str1) 时，strcpy 函数无法检查 str1 指向的字符串的大小是否真的适合 str2 指向的数组，因此就可能会出现失败的情况。

```
char str1[12];
char str2[12];
strcpy(str1, "abcd");

strcpy(str2, "abcd");              /* asignment 1 */
strcpy(str2, str1);                /* asignment 2 */
strcpy(str2, strcpy(str1, "abcd")); /* asignment 3 */
```

假设 str2 指向的字符串长度为 n，如果 str1 指向的字符串有不超过 n-1 个的字符，那么复制操作可以完成。但是，如果 str1 指向更长的字符串，而 strcpy 函数会一直复制到 str1 的第一个空字符为止，这样 strcpy 函数会越过 str2 指向的数组的边界继续复制。无论原来存放在 str2 数组后面的内存中的数据是什么，strcpy 函数都会将其覆盖，因此就无法预测执行结果。

写入的数据超出作为缓冲区的数组的大小是一种常见的程序设计错误，一般将这种错误称为缓冲区溢出（buffer overflow）。

如果目标字符串是一个未妥善初始化的指针变量，也会发生同样严重的问题。

```
string str; /* char * str; */
strcpy(str, "Hello, world.\n"); /* wrong */
```

这里，未初始化变量 str 使其指向某一确定数组，源字符串可能会被复制到内存中无法预知的区域。

作为程序员，如果无法确定分配给目标字符串的空间是否足够，就必须在使用 strcpy 复制字符串之前检查数据的长度，因此在使用 strcpy 将 line 的内容复制到 buffer 之前，应该通过测试语句确保不会发生缓冲区溢出的情况。

```
if(strlen(line) > MaxLine) Error("Input line too lone!");
strcpy(buffer, line);
```

1988 年，康奈尔大学发布的因特网蠕虫病毒就是利用计算机不会检查机器内部缓冲区的字符内存是否被输入数据用尽的漏洞，通过输入大量数据用尽已分配的内存，然后覆盖系统程序本身并执行有利于自身的命令。

68.2.2 strncpy

在实际开发中，为了避免在使用 strcpy 函数时可能会导致的缓冲区溢出问题，一般是使用执行速度较慢但更安全的 strncpy 函数来复制字符串。

strncpy 函数允许客户指定一个长度限制，其原型如下：

```
char *strncpy(char *dest, const char *src, size_t n);
```

strncpy 函数和 strcpy 函数类似，都是将 src 指定的字符串中的字符复制到 dest 指定的字符数组中，区别在于 strncpy 最多从 src 复制 n 个字符，遇到空字符会提前停止。

下面是 strncpy 函数的一个简单实现。

```
/** A simple implementation of strncpy() */
char *
strncpy(char *dest, const char *src, size_t n)
{
   size_t i;

   for (i = 0; i < n && src[i] != '\0'; i++)
      dest[i] = src[i];
   for ( ; i < n; i++)
      dest[i] = '\0';

   return dest;
}
```

strncpy 函数允许客户指定输入字符串的最大长度，从而可以防止写入的字符超过字符数组的大小。

```
char buffer[MaxLine+1];
strncpy(buffer, src, MaxLine);
```

调用 strncpy 函数可以保证至多只能有 MaxLine 个字符被复制，因此可以安全地将字符串从 line 复制到 buffer，并且不会覆盖 buffer 后面的数据。

strncpy 的缺陷在于以下几个方面的限制。

- strncpy(dest, src ,n) 函数调用返回时，只有在源字符串长度少于 n 个时才能使目标数组以空字符结束。

  如果 src 正好包含 n 个字符，调用就会将这 n 个字符复制到 dest 数组中，但无法在字符数组的末尾保存一个空字符。

  为了保证目标字符串的正确终止，必须为 dest 分配一个额外的元素，并显式地将 dest[n] 初始化为空字符。
- 复制源字符串后，strncpy(dest, src, n) 函数调用会在 dest 每个空元素中都写上空字符，直到填满 n 个位置。

如果 MaxLine 是 1000，即使 line 很短也会在调用 strncpy(buffer, line, MaxLine) 时给 buffer 的前 1000 个元素都赋上新值，而使用空字符填满目标数组的剩余元素会大大降低 strncpy 函数的效率。

68.2.3 strcat

ANSI 字符串库中包含用于字符串连接的函数，但是调用这些函数的结果和扩展库 strlib.h 中的 Concat 函数不同。

- Concat 函数返回一个全新的字符串且不改变任何参数。
- strcat(dest, src) 必须使用客户提供的字符串空间才能将 src 中的字符追加到 dest 的末尾。

strcat（字符串拼接）通常用于将较小的字符串组合成较大的字符串，它具有如下的原型：

```
char *strcat(char *dest, const char *src);
```

strcat 函数把字符串 src 的内容追加到字符串 dest 的末尾，并且返回字符串 dest（指向结果字符串的指针）。

```
NAME
     strcat, strncat - concatenate two strings

SYNOPSIS
     #include <string.h>

     char *strcat(char *dest, const char *src);
     char *strncat(char *dest, const char *src, size_t n);

DESCRIPTION
     The strcat() function appends the src string to the dest string, overwriting the
         terminating null byte ('\0') at the end of dest, and then adds a terminating null
          byte.
     The strings may not overlap, and the dest string must have enough space for the
         result.
     If dest is not large enough, program behavior is unpredictable; buffer overruns are a
          favorite avenue for attacking secure programs.

     The strncat() function is similar, except that
     * it will use at most n bytes from src; and
     * src does not need to be null-terminated if it contains n or more bytes.

     As with strcat(), the resulting string in dest is always null-terminated.

     If src contains n or more bytes, strncat() writes n+1 bytes to dest (n from src plus
         the terminating null byte). Therefore, the size of dest must be at least strlen(
         dest)+n+1.

RETURN VALUE
     The strcat() and strncat() functions return a pointer to the resulting string dest.
```

下面列举了一些 strcat 函数的用法。

```
strcpy(str1, "abcd"); /* str1 --- abcd */
strcat(str1, "efg"); /* str1 --- abcdefg */
strcpy(str2, "hijk"); /* str2 --- hijk */
strcat(str1, str2); /* str1 --- abcdefghijk*/
```

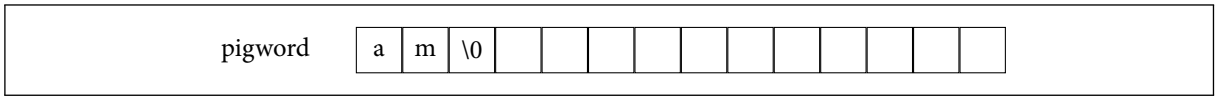
现在考虑使用 strcat 和 strcpy 重新实现 Pig Latin 的组成部分，假定变量 head 和 tail 分别包含字符串“src”和“am”，可以使用以下代码将这两个字符串重新组合成字符串“amsrcay”。

```
char pigword[MaxLine+1];
```

```
strcpy(pigword, tail);
strcat(pigword, head);
strcat(pigword, "ay");
```

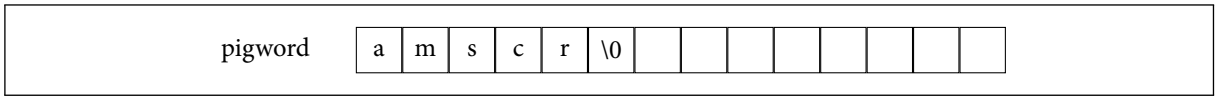
当声明 pigword 时，无法得知其中元素的初始内容，也不能假设它包含的是空字符串。通过调用 strcpy 函数可以确保 tail 中的字符被复制到了数组 pigword 的开头。

```
strcpy(pigword, tail);
```



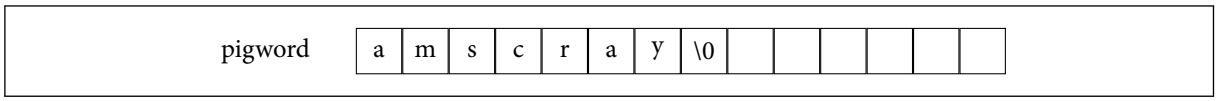
接下来，通过调用 strcat 函数找到 pigword 的末尾并把 head 的内容复制到 tail 的后面。

```
strcat(pigword, head);
```



为了完成 Pig Latin 的转换，继续调用 strcat 函数来把字符串“ay”追加到 pigword 后面。

```
strcat(pigword, "ay");
```



通常也会忽略 strcat 函数的返回值，下面的例子说明了可能使用返回值的方法。

```
strcpy(str1, "abcd");             /* str1 --- abcd */
strcat(str1, strcat(str1, "efg")); /* str1 --- abcdefgabcdefg */
```

如果 str1 指向的数组无法容纳 str2 指向的字符串中的字符，那么调用 strcat(str1, str2) 将会把 str2 中的字符和\0 添加到 str1 中已存储的字符串的末尾，因此结果将是不可预测的。

68.2.4 strncat

在使用 strcat 函数时，也可能会连接过多字符造成缓冲区溢出。

ANSI 字符串库提供的 strncat 函数有一定的改进，调用 strncat(dest, src, n) 最多能从源字符串 src 中复制 n 个字符拼接到目标字符串 dest 的末尾。

strncat 函数的原型如下：

```
char *strncat(char *dest, const char *src, size_t n);
```

为了确定 n 值，还需要检查目标字符串的长度，这样才能知道缓冲区的剩余空间。

下面的示例是 strncat 函数的一个简单实现。

```
/* A simple implementation of strncat() */

char*
strncat(char *dest, const char *src, size_t n)
{
   size_t dest_len = strlen(dest);
   size_t i;

   for (i = 0 ; i < n && src[i] != '\0' ; i++)
      dest[dest_len + i] = src[i];
```

```
    dest[dest_len + i] = '\0';

    return dest;
}
```

68.2.5 strcmp

strcmp(字符串比较)函数有下面的原型:

```
int strcmp(const char *s1, const char *s2);
```

strcmp 函数比较字符串 s1 和字符串 s2,然后根据 s1 是否小于、等于或大于 s2,并相应地返回小于、等于或大于 0 的值。

```
NAME
     strcmp, strncmp - compare two strings
SYNOPSIS
     #include <string.h>

     int strcmp(const char *s1, const char *s2);
     int strncmp(const char *s1, const char *s2, size_t n);
DESCRIPTION
     The strcmp() function compares the two strings s1 and s2.
     It returns an integer less than, equal to, or greater than zero if s1 is found,
         respectively, to be less than, to match, or be greater than s2.

     The strncmp() function is similar, except it compares the only first (at most) n
         bytes of s1 and s2.

RETURN VALUE
     The strcmp() and strncmp() functions return an integer less than, equal to, or
         greater than zero if s1 (or the first n bytes thereof) is found, respectively, to
          be less than, to match, or be greater than s2.
```

类似于字典中单词的编排方式,strcmp 函数利用字典顺序进行字符串比较。更精确地说,如果满足下列两个条件之一,那么 strcmp 函数就认为 s1 是小于 s2 的。

- s1 与 s2 的前 i 个字符一致,但是 s1 的第(i+1)个字符小于 s2 的第(i+2)个字符。例如,“abc”小于“bcd”,“abc”小于“abd”。
- s1 的所有字符与 s2 的字符一致,但是 s1 比 s2 短。例如,“abc”小于“abcd”。

通过选择适当的关系运算符(<、<=、>、>=)或判等运算符(==、!=),可以测试 str1 与 str2 之间任何可能的大小关系。

- 为了检查 str1 是否小于 str2,可以写成

  ```
  if(strcmp(str1, str2) < 0)
    ...
  ```

- 为了检查 str1 是否小于或等于 str2,可以写成

  ```
  if(strcmp(str1, str2) <= 0)
    ...
  ```

- 更特殊的,如果要测试两个字符串是否相同,只需调用 strcmp 函数并查看结果是否为 0。当程序能明确的将结果和 0 进行比较时,strcmp 不会引起混淆。

  ```
  if(strcmp(str1, str2))
    ...
  ```

  C 语言将整数 0 解释为 FALSE,而把其他非零整数解释为 TRUE,因此上述的 if 语句在 str1 与 str2 相同时不会执行。

当比较两个字符串中的字符时，strcmp 函数会查看表示字符的 ASCII 码，从而可以得出一些 strcmp 函数会遵循的规则。

- 所有的大写字母都小于所有的小写字母。
  在 ASCII 码中，65 ～ 90 的编码表示大写字母，97 ～ 122 的编码表示小写字母。
- 数字小于字母。
  在 ASCII 码中，48 ～ 57 的编码表示数字。
- 空格符小于所有打印字符。
  在 ASCII 码中，空格符的编码是 32。

具体来说，strcmp 函数的返回值可能是源于函数的传统编写方式。例如，在 K&R C 中 strcmp 的实现中，函数的返回值是字符串 s(源字符串)和字符串 t(测试字符串)中第一个“不匹配”字符的差。

```
int strcmp(char *s, char *t)
{
  int i;

  for(i = 0; s[i] == t[i]; i++)
    if(s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

- 如果 s 指向的字符串“小于”t 指向的字符串，那么结果为负数。
- 如果 s 指向的字符串“大于”t 指向的字符串，那么结果为正数。

现在已经不能保证 strcmp 函数在现代函数库中的实现与 K&R C 的实现的关系，因此对 strcmp 函数的返回值的假设没有特殊的意义。

为了找出给定的字符串在字符数组中的位置，下面 FindStringInArray 的实现中使用了 strcmp 函数。

```
int FindStringInArray(string key, string array[], int n)
{
  int i;

  for(i = 0; i < n && strcmp(key, array[i]); i++)
    ;
  if(i < n)
    return (i);
  return (-1);
}
```

for 循环控制行包括了所有的程序步骤，因此循环体只是一个分号。

只要 i 比 n 小，且 key 与当前数组项不符，for 循环就会持续进行，有两种情况下可以导致 for 循环退出。

- 数组中的元素都已经测试完毕；
- 循环在下标 i 处找到了分配的元素而终止。

68.2.6 strncmp

string.h 接口包含的另一个字符串比较函数是 strncmp。

调用 strncmp(s1, s2, n) 和 strcmp(s1, s2) 相似，只是 strncmp 最多只考虑两个字符串中的 n 个字符。只要字符串中的前 n 个字符都相同，strncmp 就返回 0，就算后面的字符不同也是如此。

68.2.7 strlen

strlen(求字符串长度)函数有下面的原型：

```
size_t strlen(const char *s);
```

定义在 C 语言函数库中的 size_t 类型是无符号整型(通常是 unsigned int 或 unsigned long int),因此可以认为 strlen 函数的返回值是整数。

```
NAME
     strlen - calculate the length of a string
SYNOPSIS
     #include <string.h>

     size_t strlen(const char *s);
DESCRIPTION
     The strlen() function calculates the length of the string s, excluding the
         terminating null byte ('\0').
RETURN VALUE
     The strlen() function returns the number of bytes in the string s.
```

strlen 函数返回字符串 s 的长度,或者更精确地说,strlen 函数返回 s 中第一个空字符前的字符的个数,但不包括第一个空字符。

当用数组作为函数的实际参数时,strlen 函数不会测量数组本身的长度,而是返回存储在数组中的字符串的字符个数(不包括空字符)。

在下面的示例程序中,将使用 C 语言的字符串库函数来实现一个显示一个月的每日提示列表的功能。

在程序运行时,用户需要输入一系列提示,每条提示都要有一个前缀来说明一个月中的哪一天。当用户用 0 代替有效的前缀输入时,程序将输入的所有信息按日期的顺序输出。

```
Enter day and reminder: 5 Prepare the travel
Enter day and reminder: 6 Buy ticket
...

Day Reminder
  5 Prepare the travel.
  6 Buy ticket
  8 Arrive at destination
 10 Go back home
...
```

在编写程序时,总体策略如下:

- 首先,读入一系列日期和提示的组合。
  1. 为了读入日期,可以使用 scanf 函数。
  2. 为了读入提示,可以使用 read_line 函数。
- 接着,按照顺序将信息进行存储(按日期排序)。
  输入的天和提示是分开输入的,因此需要把字符串存储在二维字符数组中,数组的每一行包含一条字符串。
  - 在读入某天以及对应的提示后,通过使用 strcmp 函数进行比较来查找数组从而确定这一条所在的位置。
  - 使用 strcpy 函数把该位置之后的所有字符串往后移动一个位置,从而空出存储天的位置。
  - 把输入的天存储到数组中之后,调用 strcat 函数来把提示附加到其后面。
- 最后,使用 for 循环将输入信息按日期顺序打印出来。

如果希望日期在两个字符的域中右对齐以便它们的个位可以对齐,这里可以选择用 scanf 函数把日期读入到整型变量中,然后调用 sprintf[1] 函数把天转换成字符串格式。

```
$ man sprintf
NAME
```

```
       printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf -
           formatted output conversion
   SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);
     Feature Test Macro Requirements for glibc:

     snprintf(), vsnprintf():
     _BSD_SOURCE || _XOPEN_SOURCE >= 500 || _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L; or
         cc -std=c99
   DESCRIPTION
       The functions in the printf() family produce output according to a format as
           described below.
       The functions printf() and vprintf() write output to stdout, the standard output
           stream; fprintf() and vfprintf() write output to the given output stream; sprintf
           (), snprintf(), vsprintf() and vsnprintf() write to the character string str.

       The functions snprintf() and vsnprintf() write at most size bytes (including the
           terminating null byte ('\0')) to str.

       The functions vprintf(), vfprintf(), vsprintf(), vsnprintf() are equivalent to the
           functions printf(), fprintf(), sprintf(), snprintf(), respectively, except that
           they are called with a va_list instead of a variable number of arguments. These
           functions do not call the va_end macro. Because they invoke the va_arg macro, the
            value of ap is undefined after the call.

       These eight functions write the output under the control of a format string that
           specifies how subsequent arguments (or arguments accessed via the variable-length
            argument facilities of stdarg(3)) are converted for output.

       C99 and POSIX.1-2001 specify that the results are undefined if a call to sprintf(),
           snprintf(), vsprintf(), or vsnprintf() would cause copying to take place between
           objects that overlap (e.g., if the target string array and one of the supplied
           input arguments refer to the same buffer).
```

sprintf 是个变参函数，主要功能是把格式化的数据写入某个字符串中，从而就可以将各种类型的数据构造成字符串。

sprintf 跟 printf 在用法上几乎一样，只是打印的目的地不同而已，前者打印到字符串中，后者则直接在命令行上输出。

sprintf 函数的原型如下：

```
int sprintf(char *buffer, const char *format[, argument]...);
```

除了前两个参数类型固定外，后面可以接任意多个参数，而且 sprintf 函数的精华就在于第二个参数——格式化字符串。

- buffer：char 型指针，指向将要写入的字符串的缓冲区。

- format:格式化字符串。
- $, argument$

  ...:可选参数,可以是任何类型的数据。
- 返回值:字符串长度(strlen)

printf 和 sprintf 都使用格式化字符串来指定串的格式,在格式串内部使用一些以“%”开头的格式说明符(format specifications)来占据一个位置,在后边的变参列表中提供相应的变量,最终函数就会用相应位置的变量来替代那个说明符,产生一个调用者想要的字符串。

例如,sprintf 函数在输出完毕时会自动添加一个空字符,那么调用下面的 sprintf 语句可以把 day 的值写入 day_str 中,得到的新的 day_str 就会是一个包含由空字符结尾的合法字符串。

```
sprintf(day_str, "%2d", day);
```

另外,为了确保用户输入的日期数字不超过两位,将使用下面的 scanf 函数来截断输入的数字,最多只读入两个数字。

```
scanf("%2d", &day);
```

scanf 函数的格式说明的% 和 d 之间的数 2 会指示 scanf 函数在读入两个数字后停止。

```
/* Prints a one-month reminder list */

#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50
#define MSG_LEN 60

int read_line(char str[], int n);

main()
{

}
```

68.2.8 strchr

string.h 接口也导出一些用于搜寻字符串的函数,其中函数 strchr(s, ch) 和 FindChar(ch, s, 0) 的功能相同。

- strchr 返回一个指向匹配字符的指针
- FindChar 返回匹配字符的下标。

如果没有找到匹配字符,strchr 返回 NULL。

下面是 strchr 的函数原型。

```
NAME
     strchr, strrchr, strchrnul - locate character in string

SYNOPSIS
     #include <string.h>

     char *strchr(const char *s, int c);
     char *strrchr(const char *s, int c);

     #define _GNU_SOURCE
```

```
       #include <string.h>

       char *strchrnul(const char *s, int c);

DESCRIPTION
       The strchr() function returns a pointer to the first occurrence of the character c in
           the string s.

       The strrchr() function returns a pointer to the last occurrence of the character c in
           the string s.

       The strchrnul() function is like strchr() except that if c is not found in s, then it
           returns a pointer to the null byte at the end of s, rather than NULL.

       Here "character" means "byte"; these functions do not work with wide or multibyte
           characters.

RETURN VALUE
       The strchr() and strrchr() functions return a pointer to the matched character or NULL
            if the character is not found.
       The terminating null byte is considered part of the string, so that if c is specified
           as '\0', these functions return a pointer to the terminator.

       The strchrnul() function returns a pointer to the matched character, or a pointer to
           the null byte at the end of s (i.e., s+strlen(s)) if the character is not found.
```

68.2.9 strrchr

函数 strrchr 和 strchr 的作用大致相同，只是 strrchr 是从字符串的末尾开始搜索，找到的是最后一个匹配的字符，而不是第一个。

```
strrchr(s, ch);
```

strrchr 从字符串 s 的末尾开始搜索，并试图找到最后一个匹配的字符 ch，否则返回 NULL。

68.2.10 strstr

string.h 接口中导出的 strstr 函数可以搜寻匹配的字符串。
下面是 strstr 的函数原型。

```
NAME
       strstr, strcasestr - locate a substring

SYNOPSIS
       #include <string.h>

       char *strstr(const char *haystack, const char *needle);

       #define _GNU_SOURCE

       #include <string.h>

       char *strcasestr(const char *haystack, const char *needle);

DESCRIPTION
       The strstr() function finds the first occurrence of the substring needle in the string
            haystack. The terminating null bytes ('\0') are not compared.
```

```
       The strcasestr() function is like strstr(), but ignores the case of both arguments.

RETURN VALUE
       These functions return a pointer to the beginning of the substring, or NULL if the
           substring is not found.
```

在 strstr 调用示例中，strstr 搜索字符串 s1 中出现的第一个字符串 s2，其作用和 FindString 类似。

```
strstr(s1, s2);
```

68.3 Library Efficiency

在函数的实现中，通常首先都是以清晰，而不是以效率作为目标。

通过提升函数库的抽象层次，可以对客户隐藏有关内存分配的复杂性，这样客户不必考虑分配固定大小的字符数组，也不必担心字符数组是否超出了内存空间的界限。在使用更高层次的字符串函数库时，所有的字符串都表示为指针，它指向从堆中通过动态分配所获得字符内存空间。

但是，当了解了字符串库的内部细节后，将会发现这里引入的许多实现的效率都是相当低的，低得无法适应一些重大的应用。然而，它们是简明的、可行的、容易理解的。当用这种形式的函数工作时，可以从概念上了解字符串工作的过程。

实际上，对现代编译器而言，用汇编语言代替 C 语言来编写库函数是很普遍的做法。如果 CPU 提供了相应的字符串指令，那么使用汇编语言编写字符串处理函数能够获得很高的效率。

68.3.1 String Tail

在 strlen 函数的实现中，需要搜索到字符串参数的末尾，并且使用一个变量来跟踪字符串的长度。

```
size_t strlen(const char *s)
{
  size_t n;

  for(n = 0; *s != '\0'; s ++)
    n ++;
  return n;
}
```

随着指针 s 从左到右扫描整个字符串，变量 n 始终跟踪当前已经扫描的字符数量。当 s 最终指向空字符时，n 所包含的值就是字符串的长度。

接下来，开始精简 strlen 函数的定义。首先，把 n 的初始化移到它的声明中。

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for(; *s != '\0'; s ++)
    n ++;
  return n;
}
```

由空字符的 ASCII 码是 0 容易看出，*s != '\0' 和 *s != 0 是一样的，二者都在 *s 不为 0 时结果为真，由此得到 strlen 函数的新版本。

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for(; *s != 0; s ++)
```

```
    n ++;
  return n;
}
```

根据自增操作符的性质，在同一个表达式中对 s 进行自增操作并且测试 *s 是可行的。

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for(; *s++;)
    n++;
  return n;
}
```

如果用 while 循环代替 for 循环，又可以得到 strlen 函数的另一个新版本。

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while(*s ++)
    n++;
  return n;
}
```

上述 strlen 函数的不同实现依次精简了最初的实现，但是对于运行速度的提升没有影响，因此对于某些编译器又提供了运行速度更快的实现版本。

```
size_t strlen(const char *s)
{
  const char *p = s;

  while(*s)
    s++;
  return s - p;
}
```

- 首先通过定位空字符位置的方式来计算字符串的长度，然后用空字符的地址减去字符串中第一个字符的地址，这样就不需要在 while 循环中对变量 n 进行自增，从而提高了运行速度。
- 在指针 p 的声明中使用 const 类型限定符来消除把 s 赋值给 p 会给 s 指向的字符串带来的风险。

其中，

```
while(*s)
  s++;
```

和

```
while(*s++)
  ;
```

都是“查找字符串结尾的空字符”的惯用法（programming idiom），其中的第一个版本最终使 s 指向了空字符。

相比而言，第二个版本更加简洁，但是最后使 s 正好指向了空字符后面的位置，因此在使用第二个版本来实现 strlen[2] 时就要多减一个字符。

```
size_t strlen(const char *str)
{
  const char *eos = str;

  while(*eos++)
```

```
    ;
  return (eos - str - 1);
}
```

另外,使用递归来实现 strlen 函数可以去掉中间变量,并使代码逻辑更明确。

```
size_t strlen(const char *str)
{
  if('\0' == *str)
    return 0;
  else
    return strlen(str + 1) + 1;
}
```

如果要代码更简洁,可以使用三元运算符?:来进一步精简代码。

```
size_t strlen(const char *str)
{
  return *str ? (strlen(++str)+1) : 0;
}
```

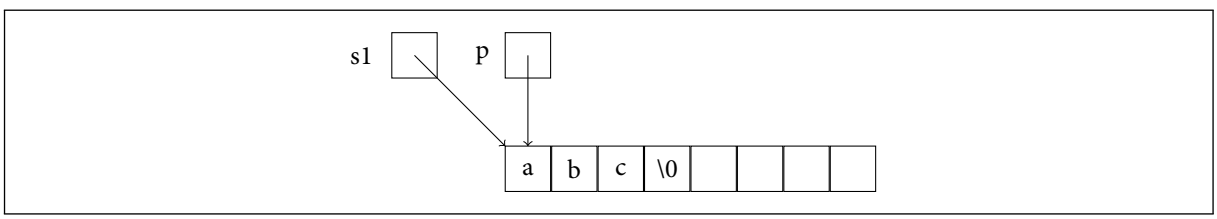
68.3.2 String Copy

为了实现 strcat 函数,可以使用如下的算法:

- 首先,查找字符串 s1 末尾空字符的位置,并且使指针 p 指向它;
- 接着,把字符串 s2 中的字符逐个复制到 p 所指向的位置。

```
char *strcat(char *s1, const char *s2)
{
  char *p;

  p = s1;
  while(*p != '\0')
    p++;
  while(*s2 != '\0')
  {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

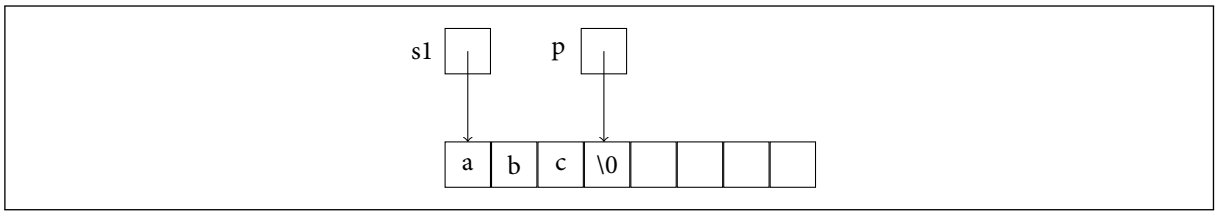
strcat 函数中的第一个 while 语句实现了第一步,把 p 指向字符串 s1 的第一个字符。

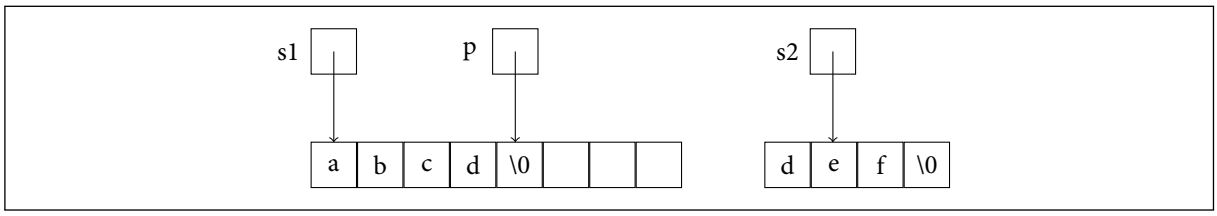


接着,p 开始自增直到指向字符串 s1 的空字符才停止。当循环终止时,p 必须指向空字符。

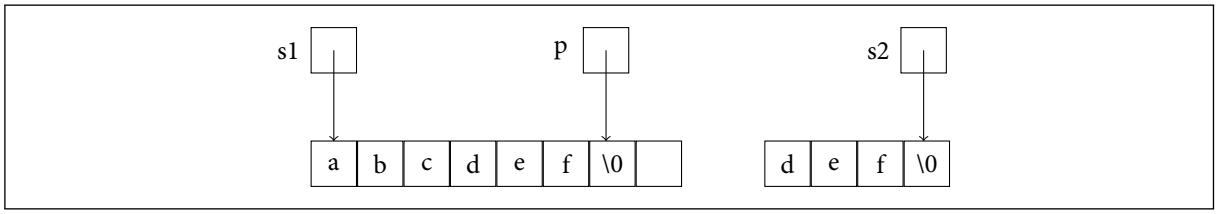
在第二个 while 语句中,将指针 s2 所指向的字符复制到接下来 p 所指向的位置,然后 p 和 s2 都进行自增。

假设 s2 最初指向的字符串是“def”,下面显示的是在第 2 个 while 循环中执行完第一次循环后,字符串 s1、s2 以及指针 p 的示意图。





当 s2 指向空字符时，循环终止。



当 p 指向的位置放置的字符为空字符之后，strcat 函数返回。

另外，通过类似于对 strlen 函数所采用的方法，可以简化 strcat 函数的实现。

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while(*p)
    p++;
  while(*p++ = *s2++)
    ;
  return s1;
}
```

上述改进的 strcat 函数的实现的核心是“字符串复制”的习惯用法。

```
while(*p++ = *s2++)
  ;
```

- 如果忽略了两个 ++ 运算符，那么圆括号中的表达式会简化为普通的赋值表达式（*p = *s2），这样只是把 s2 指向的字符复制到 p 所指向的地方。
- 添加 ++ 运算符后，p 和 s2 都进行了自增操作，通过重复执行自增操作就可以把 s2 指向的一系列字符复制到 p 所指向的位置。

在圆括号中的主要运算符是赋值运算符，所以 while 语句会测试赋值表达式的值，也就是测试复制的字符。

- 首先，如果赋值操作成功，则返回真。
- 然后，在赋值操作结束后，各自执行自增操作，因此返回值是当前指向的字符。

除空字符以外的所有字符的测试结果都为真，因此循环只有在复制空字符后才会终止，而且循环是在赋值之后终止，因此不需要单独增加一条语句来在新字符串的末尾添加空字符。

编译器可能会针对“字符串复制”的习惯用法给出警告“possibly incorrect assignment”，因为在通常在 while 语句的条件表达式中进行判断时使用的是 == 而不是 =，只是在这个特殊的用法是无效的。

为了消除编译器警告，可以使用下面的方式继续重写“字符串复制”的习惯用法。

```
while((*p++ = *s2++) != 0)
  ;
```

现在，while 语句测试的是条件，并不是直接使用赋值成功后返回 0 的事实。

68.3.3 String Invertation

在与字符串相关的实际程序开发中，通常不会用到较高层次的字符串库，都是使用 string.h 接口中的函数，或者直接和字符串的基本表示交互。

下面的字符串程序需要将以传统顺序表示的姓名转换为反向的顺序，即将 *First Middle Last* 转换为 `Last, First Middle`，从而有利于按字母顺序排序。

如果遵循 ANSI 字符串库对字符串操作的规则来设计转换函数 InvertName，必须选取两个参数来分别为原来的姓名和转换后的结果提供内存空间。

为了和 strcpy 保持一致，首先列出函数原型。

```
static void InvertName(char result[], char name[]);
```

使用 string.h 接口中的函数来实现 InvertName 会更简单。

```
/*
 * File: invert.c
 * --------------
 * This file implements a function InvertName(result, name)
 * that takes a name in standard order (first middle last) and
 * returns a new string in inverted order (last, first middle),
 * which makes it easier to alphabetize the names. The test
 * program reads in names and prints out their inverted form,
 * stopping when a blank line is entered.
 */

#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constant
 * --------
 * MaxName -- Maximum number of characters in a name
 */

#define MaxName 40

/* Private function prototypes */

static void InvertName(char result[], char name[]);

/* Main program */

main()
{
   char *standardName;
   char invertedName[MaxName+1];

   printf("This program converts a name in standard order\n");
   printf("into inverted order with the last name first.\n");
   printf("Indicate the end of input with a blank line.\n");
   while (TRUE) {
```

```
      printf("Name: ");
      standardName = GetLine();
      if (strlen(standardName) == 0) break;
      InvertName(invertedName, standardName);
      printf("%s\n", invertedName);
   }
}

/*
 * Function: InvertName
 * Usage: InvertName(result, name);
 * --------------------------------
 * This function inverts a name from its standard order
 *
 *   First Middle Last
 *
 * into inverted order, which is
 *
 *   Last, First Middle
 *
 * The client must supply an output array called result in which
 * the inverted name will be stored. That array must contain
 * at least MaxName character positions, plus one for a
 * terminating null character. If storing the inverted name
 * would exceed that limit, the function generates an error.
 * The output is always one character longer than the input
 * because of the comma, so it is possible to determine the
 * output length immediately.
 *
 * The last name is assumed to consist of all characters in the
 * name string following the last space character. If there are
 * no space characters in the word, the entire name is copied to
 * the destination array unchanged.
 */

static void InvertName(char result[], char name[])
{
   int len;
   char *sptr;

   len = strlen(name);
   sptr = strrchr(name, ' ');
   if (sptr != NULL) len++;
   if (len > MaxName) Error("Name too long");
   if (sptr == NULL) {
      strcpy(result, name);
   } else {
      strcpy(result, sptr + 1);
      strcat(result, ", ");
      strncat(result, name, sptr - name);
      result[len] = '\0';
   }
}
```

68.3.4 String Pass-through

为了在不同详细级别表示字符串的抽象层次，通常的做法是让一个接口重新命名比它层次更低的接口的函数，这样的函数称为转换函数(pass-through functions)。

提供一个传递函数而不让客户使用较低层次的对应的函数的目的是为了减少客户在概念上的复杂性。通过转换函数定义了完整的更高层次的接口后，就可以在更高的抽象层次上对数据进行操作。

首先，strlib.c 中的 StringLength 函数和 StringCompare 函数的实现分别与 string.h 接口导出的 strlen 函数和 strcmp 函数对应。

```
int StringLength(string s)
{
  return (strlen(s));
}
int StringCompare(string s1, string s2)
{
  return (strcmp(s1, s2));
}
```

其次，strlib 扩展库中的大多数函数的主要特征是它们动态地为新创建的字符串分配内存，因此引入了一个专用函数 CreateString(len) 来给长度为 len 的字符串分配内存。

考虑到字符串都以空字符结尾，因此需要为 '\0' 预留一个额外的字节。

```
static string CreateString(int len)
{
  return ((string) GetBlock(len+1));
}
```

CreateString 函数的引入满足了扩展库 strlib 中所有需要自己分配内存的函数的需求。例如，考虑下面的 CharToString 函数的实现，它为一个只含一个字符的字符串分配内存。

```
string CharToString(char ch)
{
  string result;

  result = CreateString(2);
  result[0] = 'ch';
  result[1] = '\0';
  return (result);
}
```

这样，CharToString 函数就可以动态地分配内存空间来存放只有一个字符的字符串，然后通过存储特定的字符和终止符来初始化内存单元。

为了在更高的抽象层次上定义字符串连接函数，一种方法是显式地实现所有关于复制的操作，即先为连接后的字符串分配空间，再依次从每个字符串复制字符。

```
string Concat(string s1, string s2)
{
  string s;
  int len1, len2, i;

  len1 = strlen(s1);
  len2 = strlen(s2);
  s = CreateString(len1 + len2);
  for(i = 0; i < len1; i++) s[i] = s1[i];
  for(i = 0; i < len2; i++) s[i + len1] = s2[i];
  s[len1 + len2] = '\0';
  return (s);
}
```

在上述的 Concat 函数的实现中,并没有利用较低层库中的函数,因此效率比较低。

通常,ANSI 字符串库中函数的设计应使得它们尽可能利用所针对的硬件平台的特性来有效的工作,这样产生的代码常常比采用显式字符串操作的程序效率更高。

下面是引入了 strcpy 作为 Concat 函数实现的代码。

```
string Concat(string s1, string s2)
{
  string s;
  int len1, len2;

  len1 = strlen(s1);
  len2 = strlen(s2);
  s = CreateString(len1 + len2);
  strcpy(s, s1);
  strcpy(s+len1, s2);
  return (s);
}
```

在上述的实现中,注意这一段意义模糊的代码。

```
strcpy(s + len1, s2);
```

要理解这行代码的意义,应该理解给指针加一个整数就会使指针向前推移指定数目个元素。在 Concat 函数的实现操作的是字符,所以目标地址只是从 s 后第 len1 个字符开始的数组地址,也就是 s2 的副本的位置。

调用 strcpy(s+len1,s2) 和调用 strcat(s,s2) 的效果是相同的,但是 strcpy 的形式更有效,它避免了搜索到字符串 s 的末尾的情况。

第 69 章

String Array

69.1 Ragged Array

思考下面关于在使用字符串时经常遇到的问题:

> 存储字符串数组的最佳方式是什么?

首先,最明显的解决方案是创建二维的字符数组,然后按照每行一个字符串的方式把字符串存储到数组中。

```
char planets[][8] = {
  "Mercury", "Venus", "Earth",
  "Mars", "Jupiter", "Saturn",
  "Uranus", "Neptune", "Pluto"
};
```

其中,C 语言允许省略二维数组中行的数量(可以从初始化中元素数量得出),但是要求必须说明列的数量,从而可以得到上述 planets 数组的可能形式。

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

如果字符串不足以填满数组的一整行,那么将用空字符来填补,因此导致了数组空间的“空字节”浪费。

现实中大部分字符串集都是长短字符串的混合,这样也暴露了字符串处理过程中低效性的根源。

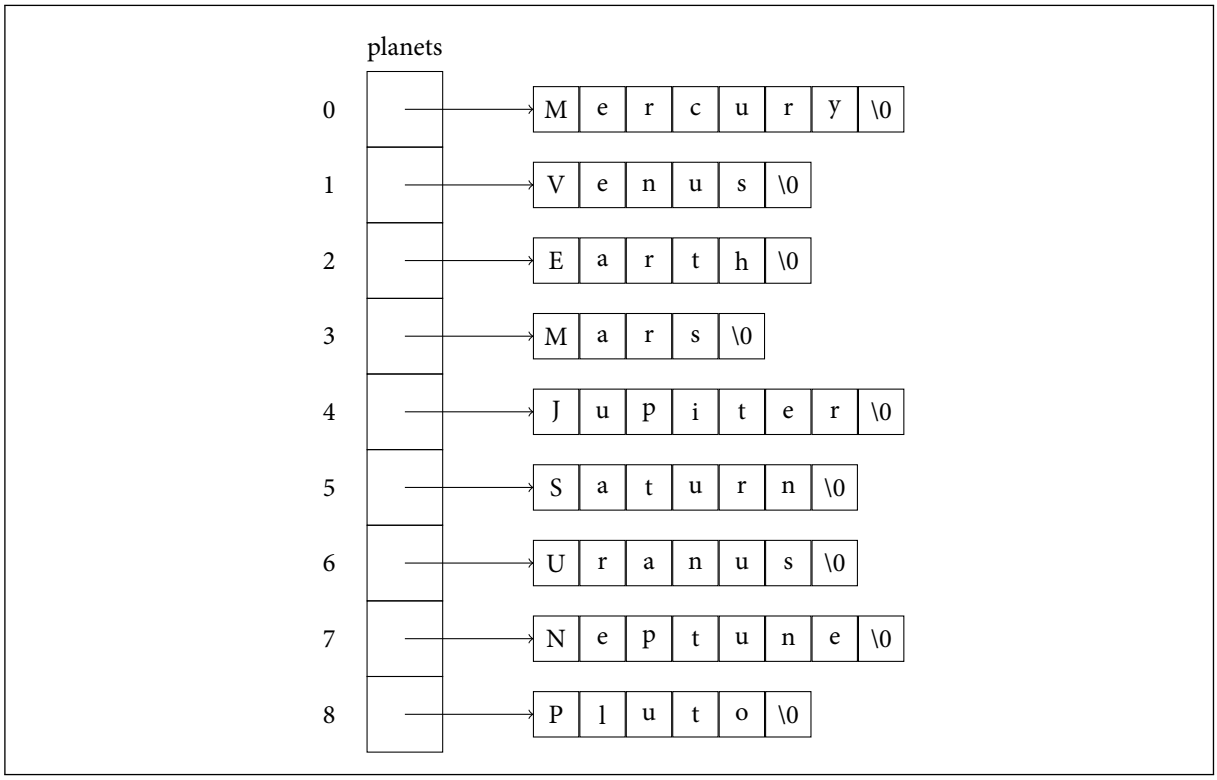
为了提高字符串处理速度,需要参差不齐的数组(ragged array,即数组的每一行有不同的长度)来存储字符串。

C 语言本身不提供数组长度可变的数组数据类型,不过 C 语言提供了模拟可变长度数组类型的工具——字符串数组,其中的元素都是指向字符串的指针。

下面是 planets 数组的另外一种写法,可以理解为指向字符串的指针的数组。

```
char *planets[] = {
  "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"
};
```

在字符串数组中，其中的每一个元素都是指向以空字符结尾的字符串的指针。



这样只需要为 planets 数组中的指针分配空间，但是字符串中不再有任何浪费的空间。

要访问字符串数组中某一个元素时，只需要给出 planets 数组的下标，这和访问二维数组元素的方式相同。例如，为了在 planets 数组中搜寻以字母 M 开头的字符串，可以使用下面的循环。

```
for(i = 0; i < 9; i++)
  if(planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

69.2 Command-line Argument

在执行 UNIX-like 系统中的程序时，经常需要提供一些信息——文件名或者改变程序行为的开关。考虑 UNIX 系统的 ls 命令，如果直接执行 ls 命令，将会显示当前目录下的文件名(对应的 DOS 命令是 dir)。

```
$ ls
Desktop Documents Downloads Music Pictures Public Templates Videos
```

如果将 ls 命令替换成 ls -l，将会显示详细的文件列表(包括文件大小、所有者、用户组、最后修改日期和时间等)。

```
$ ls -l
total 32
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Desktop
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Documents
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Downloads
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Music
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Pictures
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Public
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Templates
drwxr-xr-x 2 theqiong theqiong 4096 Mar 31 14:22 Videos
```

为了访问程序的命令行参数(command-line argument)[1],必须把 main 函数定义为包含两个参数的函数,这两个参数通常命名为 argc 和 argv[2]。

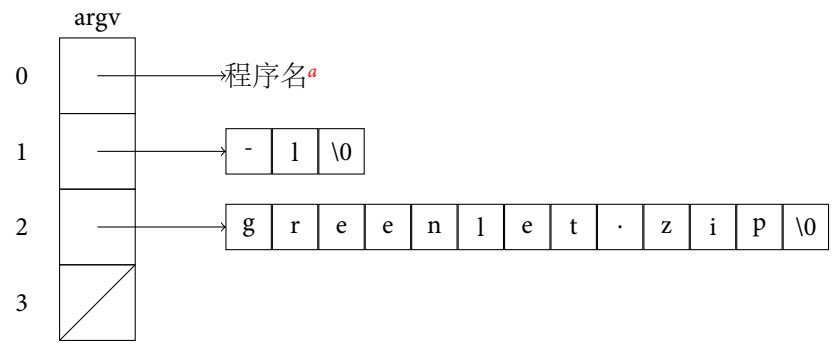
```
main(int argc, char *argv[])
{
  ...
}
```

- argc(参数计数)是命令行参数的数量(包含程序名本身)。
- argv(参数向量)是指向命令行参数的指针数组,这些命令行参数以字符串的形式存储。
  其中,argv[0] 指向程序名,而从 argv[1] 到 argv[argc-1] 则指向余下的命令行参数。
  另外,argc 有一个附加元素——argv[argc],该元素始终是一个空指针[3]。
  如果要进一步改变 ls 的行为,可以指定只显示一个文件的详细信息。

```
$ ls -l greenlet.zip
-rw-r--r--. 1 root root 74404 Jan 9 02:29 greenlet.zip
```

如果用户输入了上述的命令,那么 argc 将为 3。

- argv[0] 指向含有程序名的字符串(这里是“ls”)。
- argv[1] 指向程序的命令行参数字符串(这里是“-l”)。
- argv[2] 指向程序处理的对象(这里是“greenlet.zip”)。
- argv[3] 是空指针。



a在不同的操作系统上程序名可能会包括路径或其他信息。如果程序名不可用,那么 argv[0] 会指向空字符串。

针对 argv 是指针数组的事实,典型的访问命令行参数的方法是期望有命令行参数的程序将会设置循环来按顺序检查每一个参数。

设定访问命令行参数的循环的方法之一就是使用整型变量作为 argv 数组的索引。例如,下面的循环以每行一条的方式显示命令行参数。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i ++)
    printf("%s\n", argv[i]);
  return 0;
}
```

[1]程序并不总是在命令行中运行,因此 C 标准对应的术语是“程序参数”而不是“命令行参数”。例如,在窗口环境下,程序是通过点击鼠标来启动的,在这类环境下就是通过事件(event)来给程序传递信息的,传统意义上的命令行就不存在了。

[2]使用 argc 和 argv 仅仅是一种习惯,不是语言本身的要求。

[3]一般用宏 NULL 来代表空指针,空指针是一种不指向任何内容的特殊指针。

```
$ ./a.out a b c
a
b
c
```

另一种方法是构造一个指向 argv[i] 的指针，然后对指针重复进行自增操作来逐个访问数组剩下的元素。

因为数组的最后一个元素始终是空指针，所以循环可以在找到数组中第一个空指针时停止。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  char **p;

  for(p = &argv[1]; *p != NULL; p ++)
    printf("%s\n", *p);
  return 0;
}
```

这里 p 是指向字符指针的指针，必须小心使用。

- 把 p 设为 &argv[1] 的意义在于 argv[1] 是一个指向字符的指针，则 &argv[1] 就是指向指针的指针。
- *p 和 NULL 都是指针，因此测试 *p != NULl 表达式是没有问题的。
- 对 p 进行初始化并使其指向字符数组元素，因此对 p 进行自增操作将使 p 指向下一个元素。
- *p 是一个指向字符的指针，因此传递给 printf 的 *p 是要显示的字符的地址。

下面的示例程序完整地输出了与程序相关的命令行参数信息。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for(i = 0; i <= argc; i ++)
     printf("argv[%d] %s\n", i, argv[i]);

  return 0;
}
$ ./a.out a b c
argv[0] ./a.out
argv[1] a
argv[2] b
argv[3] c
argv[4] (null)
```

在声明数组型形式参数时，*a 的写法和 a[] 对数据类型没有影响，因此 argv 的声明可以用 **argv 来代替 *argv[]。

无论数据是否是数组的形式，都可以创建指向任何其他数据类型的指针数组，指针数组在动态存储分配联合(union)是特别有用的。

下面设计测试示例来说明访问命令行参数的方法，执行程序 planet 时在命令行中添加测试字符串，找出哪些字符串是行星的名字。

```
$ planet Jupiter venus Earth fred
```

在程序执行结果中会指出命令行中的每个字符串是否是行星的名字。如果是，将显示行星的编号(把最靠近太阳的行星编号为 1)。

注意，除非字符串的首字母大写并且其余字母小写，否则 planet 程序不会认为字符串是行星的名字。

```
/* Checks planet name */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = { "Mercury", "Venus", "Earth",
                      "Mars", "Jupiter", "Saturn",
                      "Uranus", "Neptune", "Pluto"};
  int i, j;

  for(i = 1; i < argc; i ++){
    for(j = 0; j < NUM_PLANETS; j ++)
      if(strcmp(argv[i], planets[j]) == 0){
        printf("%s is planet %d\n", argv[i], j+1);
        break;
      }
    if(j == NUM_PLANETS)
      printf("%s is not a planet.\n", argv[i]);
  }

  return 0;
}
```

planet 程序依次访问每个命令行参数，并把它与 planets 数组中的字符串进行比较，直到找到匹配的名字或者到了数组的末尾才停止。

在程序中会调用 strcmp 函数来进行字符串比较，这里传递给 strcmp 函数的参数是 argv[i]（指向命令行参数的指针）和 planets[j]（指向行星名的指针）。

Bibliography

[1] steedhorse. sprintf，你知道多少？，05 2005. URL http://blog.csdn.net/steedhorse/article/details/330206.

[2] tianmo2010. 实现 strlen() 函数，08 2011. URL http://blog.csdn.net/tianmohust/article/details/6667111.

[3] Wikipedia. C 风格字符串. URL http://zh.wikipedia.org/wiki/C%E9%A3%8E%E6%A0%BC%E5%AD%97%E7%AC%A6%E4%B8%B2.

Part XI

File

第 70 章

Introduction

如果需要在某程序执行结束后永久地保存信息,那么就必须将其存入支持永久存储的辅助存储设备上的文件中。

在计算中的文件可以分为文本文件和二进制文件,其中文本文件由字符数据组成,二进制文件则包含 ASCII 及扩展 ASCII 字符中编写的数据或程序指令。

- 广义的二进制文件即为文件,由文件在外部存储设备的存放方式为二进制而得名。
- 狭义的二进制文件即指除文本文件以外的文件。

其中,文本文件可以用来进行输入输出,并且通常情况下计算机会按顺序读写文本文件。

每一类文件都有自己所特有的文件结构,文件和目录代表了文件存放在磁盘等存储设备上的组织方法。

70.1 Overview

文件(或称文件、档案)是存储在某种长期储存设备或临时存储设备中的一段数据流,并且归属于计算机文件系统管理之下。

- "长期储存设备"一般指磁盘、光盘、磁带等。
- "短期存储设备"一般指计算机存储器,而且在内存中的一组字节通常不叫做文件,除非它们被存放在内存盘中。

需要注意的是,存储于长期存储设备的文件不一定是长期存储的,有些也可能是程序或系统运行中产生的临时数据,并于程序或系统退出后删除。

"文件"一词最早在 1952 年用于计算机数据方面,指的是在打孔卡上所储存的信息,因此以前文件的定义是"记录的串行",现在这种定义已经不常用了。

在目前的计算机中,文件是由软件创建的,而且符合特定的文件格式。

- 常用的文件是文本文件,是由一些字符的串行组成的。
- 二进制文件一般是指除了文本文件以外的文件。

文件系统是一种存储和组织计算机数据的方法,它使得对其访问和查找变得容易。虽然一个文件表现为一个单一的流,但它经常在磁盘不同的位置存储为多个数据碎片(或者是多个磁盘)。操作系统会将它们组织成文件系统,这样每个文件就可以放在特定的文件夹或目录中。

在计算机科学中,磁盘文件系统是一种在永久储存装置上管理数据的文件系统。绝大多数的磁盘文件系统都是为了针对如何让磁盘系统运作最佳化而设计的。

文件系统使用文件和树形目录的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念,用户使用文件系统来保存数据不必关心数据实际保存在硬盘(或者光盘)的地址为多少的数据块上,只需要记住这个文件的所属目录和文件名。在写入新数据之前,用户不必关心硬盘上的那个块地址没有被使用,硬盘上的存储空间管理(分配和释放)功能由文件系统自动完成,用户只需要记住数据被写入到了哪个文件中。

文件系统通常使用硬盘和光盘等存储设备,并维护文件在设备中的物理位置。但是,实际上文件系统也可能仅仅是一种访问数据的界面而已,实际的数据是通过网络协议(如 NFS、SMB、9P 等)提供的或者内存上,也可能根本没有对应的文件(如 proc 文件系统)。

严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型（Abstract data type）。

70.2 EOF

如果数据源是文件或流，那么文件结尾（End of File，缩写为 EOF）指的是操作系统无法从数据源读取更多数据的情形。

在 C 标准库中，类似 getchar 这样的数据读取函数返回一个与符号（宏）EOF 相等的值来指明文件结束的情况发生。

EOF 的真实值与不同的平台有关，但是通常是-1（比如在 glibc 中），并且不等于任何有效的字符代码。块读取函数返回读取的字节数，如果它小于要求读取的字节数，就会出现一个文件结束符。

从一个终端的输入从来不会真的“结束”（除非设备被断开），但把从终端输入的数据分区成多个“文件”却很有用，因此一个关键的串行被保留下来来指明输入结束。在 UNIX 和 AmigaDOS 中，将击键翻译为 EOF 的过程是由终端的驱动程序完成的，因此应用程序无需将终端和其它输入文件区分开来。

- Unix 平台的驱动程序在行首传送一个传输结束字符（Control-D，ASCII 编码为为 04）来指明文件结束。
- 在 AmigaDOS 中，驱动程序传送一个 Control-\来指明文件结束（而 Control-D 被用作中断字符）。

要向输入流中插入一个真正的 Control-D 字符，用户需要把一个“引用”命令字符放在它的前面（通常是 Control-V，表示下一个字符不作为控制字符，而是按照字面量使用）。

在微软的 DOS 和 Windows（以及 CP/M 和许多 DEC 操作系统）中，读取数据时终端不会产生 EOF。此时，应用程序知道数据源是一个终端（或者其它“字符设备”），并将一个已知的保留的字符或串行解释为文件结束的指明。

最普遍地说，它是 ASCII 码中的替换字符（Control-Z，代码 26）。一些 MS-DOS 程序，包括部分微软 MS-DOS 的 shell（COMMAND.COM）和操作系统功能程序（如 EDLIN），将文本文档中的 Control-Z 视为有意义数据的结尾，并且/或者在写入文本文档时将 Control-Z 添加到文档末尾。这是由于两个原因：

- 向后兼容 CP/M。CP/M 的第 1 版与第 2 版的文件系统以 128 字节“块”的倍数记录文件长度，所以当有意义数据在一个“块”的中间结束时，习惯上用 Control-Z 字符来标记它，此后至块结尾的字节为未利用。而 MS-DOS 文件系统总会记录文件确切的字节长度，所以在 MS-DOS 中文件不再必需以 Control-Z 字符来标记结尾。
- 应用程序在从终端和文本文档读取数据时得以使用相同的代码。

在 ANSI X3.27-1969 磁带标准中，文件结束是由带标记（tape mark）指明的，它由一个约 3.5 英寸的间隙和随后的一个字节组成，在九轨磁带中这个字节包含字符 13（十六进制），而在七轨磁带中包含字符 17（八进制）。

带结尾（end-of-tape）通常缩写为 EOT，它是由两个带标记指明的。这是在像 IBM 360 这样的机器上使用的标准。另外，指明快到磁带物理结尾的反射棒也被称为一个 EOT 标记。

70.3 Naming

文件名称是注明电脑上每一文件的特别字串。在不同的操作系统中，对文件名称在长度及可允许使用的字符上可能作出限制。

文件中的“通用资源标志符 - URI”最少是由四个部份组成的：

- 电脑 ID（IP 地址，网名或 LAN 电脑名称），例如 wikipedia.org、207.142.131.206 或\\MYCOMPUTER）
- 装置（磁盘、根挂载点、磁盘区等），例如 C: 或/
- 路径（目录树的位置：在第一个和最后一个路径分隔线之间的任何字符）
- 文件名称

要参照在远端主机中的文件,它的网络 ID 必须是提供的。如果它的 URI 没有路径部份,那文件部份便假设在当前工作的目录。

在很多的系统中(包括 DOS 及 UNIX)会以句点 (.) 方式来将文件名称分成两个部份,包括可含有一个或多个字符的扩展名。

- 文件的基本名称,即适当的文件名称和主要文件名称等。
- 文件的扩展名,通常用来指出与指定格式有关联的文件格式或 MIME 类型。

在同一个目录中,文件名称必须是要唯一的。但是,两个文件在不同的目录中,其名称是可以相同的。

在多数的操作系统中存在的编码问题导致不建议使用西欧或空白以外的字符作为文件名称。另外,在某些操作系统中,大小写不同的写法可以有不同的解释。

在某些的操作系统(例如 UNIX 及 Macintosh)中,可以容许一个文件可以多于一个名字,这称作硬连接。注意,硬链接与 Windows 快捷方式是不同的。

在大部份的操作系统中,文件系统里的某些字符因为含有特别的意思,因此在一个文件的名称中是不可以包括以下的字符:

- 任何控制字符 (0-31)
- / 斜线 (SLASH) (使用为路径分隔线,例如 UNIX 中的根目录符号)
- | 管道 (PIPE)
- \反斜线 (BACKSLASH) (使用为路径分隔线)
- ? 问号 (QUESTIONMARK) (在 Windows 操作系统中使用为一个万用字符)
- “” 双引号 (DOUBLE-QUOTATIONMARK) (这使用于标示含有空白字符的文件名称)
- * 星号 (STAR) (在 Windows 操作系统中使用为万用字符)
- : 冒号 (COLON) (这使用于决定哪一个挂载点 / Windows 操作系统中的磁盘)
- < 小于 (LESS THAN)
- > 大于 (GREATER THAN)
- . 句点 (可允许使用,但最后的句点会被诠释为扩展名的分隔)

另外,在 UNIX 系统中,以. 为首的被当作系统文件 (常作为软件配置的隐藏文件)

某些文件名称亦会保留,不能作为文件名称使用。例如,DOS 的装置文件:CON, PRN, AUX, CLOCK$, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9.

包含保留字的文件名称或文件结尾,例如 aux.c , file.aux 或 NUL.txt 等在内的文件名称也应该避免被使用。

在不同的操作系统中,包括扩展名在内的文件名称具有不同的最大长度。例如,在 ISO 9660 的文件系统中,最大的目录层次为 8 层。假设文件最大长度为 255 个字符,这可知道在这个文件系统中,URL 的最大长度是 2040 个字符。

70.4 Storage

文本文件因其结构简单而被广泛用于记录信息,它能够避免其它文件格式遇到的一些问题。此外,当文本文件中的部分信息出现错误时,往往能够比较容易的从错误中恢复出来,并继续处理其余的内容。

文本文件的一个缺点是,它的熵往往较低,也就是说,其实本可以用更小的存储空间记录这些信息。

ASCII 标准使得只含有 ASCII 字符的文本文件可以在 Unix、Macintosh、Windows、DOS 和其它操作系统之间自由交互,而其它格式的文件是很难做到这一点的。但是,在这些操作系统中,换行符并不相同,处理非 ASCII 字符的方式也不一致。

- LF:在 Unix 或 Unix 相容系统。

- CR+LF：MS-DOS、Windows、大部分非 Unix 的系统
- CR：Apple II 家族，Mac OS 至版本 9

C/C++ 利用转义序列 \n 来换行，后续发展起来的 Unicode 标准指定以下的字符为兼容标准的应用程序应识别的换行字符。

- LF：换行，U+000A
- VT：垂直定位，U+000B
- FF：换页符，U+000C
- CR：回车符，U+000D
- CR+LF：CR（U+000D）跟随 LF（U+000A）
- NEL：下一行，U+0085
- LS：分行，U+2028
- PS：分段，U+2029

文本文件在 MIME 标准中的类型为“text/plain”，它通常还附加编码的信息。

- 在 Mac OS X 出现前，当 Resource fork 指定某一个文件的类型为“TEXT”时，Mac OS 就认为这个文件是文本文件。
- 在 Windows 中，当一个文件的扩展名为“txt”时，系统就认为它是一个文本文件。

此外，处于特殊的目的，有些文本文件使用其它的扩展名。例如，计算机的源代码也是文本文件，它们的后缀是用来指明它的程序语言的。

70.5 Capacity

文件大小能衡量一个计算机文件的大小，通常情况下以带前缀的字节数表示。

文件实际所占磁盘空间取决于文件系统，文件系统的最大文件大小取决于保留存储尺寸信息的位数量及文件系统的总大小。例如，在 FAT32 文件系统中，单个文件的大小不能超过 4 GiB。

- 1 byte = 8 bits
- 1 KiB = 1,024 bytes
- 1 MiB = 1,048,576 bytes
- 1 GiB = 1,073,741,824 bytes
- 1 TiB = 1,099,511,627,776 bytes

Table 70.1: 文件大小换算表

名称	符号	二进制计量	十进制计量	字节数
KiloByte	KiB	2^{10}	10^{3}	1,024
MegaByte	MiB	2^{20}	10^{6}	1,048,576
GigaByte	GiB	2^{30}	10^{9}	1,073,741,824
TeraByte	TiB	2^{40}	10^{12}	1,099,511,627,776
PetaByte	PiB	2^{50}	10^{15}	1,125,899,906,842,624
ExaByte	EiB	2^{60}	10^{18}	1,152,921,504,606,846,976
ZettaByte	ZiB	2^{70}	10^{21}	1,180,591,620,717,411,303,424
YottaByte	YiB	2^{80}	10^{24}	1,208,925,819,614,629,174,706,176

70.6 Format

对于硬盘驱动器或任何计算机存储来说,有效的信息只有 0 和 1 两种。对于不同的信息有不同的存储格式,因此计算机必须设计有相应的方式进行信息-比特的转换。

文件格式(或文件类型)是指计算机为了存储信息而使用的对信息的特殊编码方式,用来识别内部储存的信息。每一类信息都可以一种或多种文件格式保存在计算机存储设备中,而且每一种文件格式通常会有一种或多种扩展名可以用来识别,但也可能没有扩展名。

有些文件格式被设计用于存储特殊的数据,例如:

- 图像文件中的 JPEG 文件格式仅用于存储静态的图像,而 GIF 既可以存储静态图像,也可以存储简单动画;
- Quicktime 格式则可以存储多种不同的媒体类型。
- Text 文件一般仅存储简单没有格式的 ASCII 或 Unicode 的文本;
- HTML 文件则可以存储带有格式的文本;
- PDF 和 DOC 格式则可以存储内容丰富的,图文并茂的文本。

每一种文件类型的识别代码(如 GIF 为 GGIF,BMP 为 BM 等)都是不一样的。当然,还有一些没有文件类型的识别代码(如 INI、INF)。

扩展名只是用以帮助应用程序识别文件格式,某些文件类型(如 DAT、TXT)没有固定的文件结构。例如,.txt 是包含极少格式信息的文字文件的扩展名。.txt 格式并没有明确的定义,它通常是指那些能够被系统终端或者简单的文本编辑器接受的格式。任何能读取文字的程序都能读取带有.txt 扩展名的文件,因此通常认为.txt 文件是通用的、跨平台的。

在英文文本文件中,ASCII 字符集是最为常见的格式,而且它在许多场合也是默认的格式。对于带重音符号的和其它的非 ASCII 字符,必须选择一种字符编码。在很多系统中,字符编码是由计算机的区域设置决定的。常见的字符编码包括支持许多欧洲语言的 ISO 8859-1。

由于许多编码只能表达有限的字符,通常它们只能用于表达几种语言。Unicode 制定了一种试图能够表达所有已知语言的标准,Unicode 字符集非常大,它囊括了大多数已知的字符集。Unicode 有多种字符编码,其中最常见的是 UTF-8,这种编码能够向后兼容 ASCII,相同内容的的 ASCII 文本文件和 UTF-8 文本文件完全一致。

微软的 MS-DOS 和 Windows 采用了相同的文本文件格式,它们都使用 CR 和 LF 两个字符作为换行符,这两个字符对应的 ASCII 码分别为 13 和 10。通常,最后一行文本并不以换行符(CR-LF 标志)结尾,包括记事本在内的很多文本编辑器也不在文件的最后添加换行符。

大多数 Windows 文本文件使用 ANSI、OEM 或者 Unicode 编码。Windows 所指的 ANSI 编码通常是 1 字节的 ISO-8859 编码,不过对于像中文、日文、朝鲜文这样的环境,需要使用 2 字节字符集。在过渡至 Unicode 前,Windows 一直用 ANSI 作为系统默认的编码。而 OEM 编码,也是通常所说的 MS-DOS 代码页,是 IBM 为早期 IBM 个人电脑的文本模式显示系统定义的。在全屏的 MS-DOS 程序中同时使用了图形的和按行绘制的字符。新版本的 Windows 可以使用 UTF-16LE 和 UTF-8 之类的 Unicode 编码。

同一个文件格式,用不同的程序处理可能产生截然不同的结果。例如用 Microsoft Word 查看 Word 文件的时候,可以看到文本的内容,而以无格式方式在音乐播放软件中播放,产生的则是噪声,因此一种文件格式对某些软件会产生有意义的结果,对另一些软件来看,就像是毫无用途的数字垃圾。

许多文件格式都有公开的、不同程度规范或者建议的格式。这些规范或者建议描述了数据如何编码,如何排列,或者是规定是否需要特定的计算机程序读取或处理。

需要注意的是,使用不公开的文件格式可能会带来额外的成本。要了解这类文件格式,或者需要通过对获得的文件进行逆向工程,或者通过向开发者付费来获得文件的格式。这里,第二种方式中往往还需要与开发者签订专有协议。

70.7 Access

用文本编辑器打开一个文本文件后,用户可以看到可读的纯文本内容。控制字符有时被编辑器当做文字指令,有时被当作类似纯文本那样可编辑的转义字符。尽管文本文件里面有纯文本信息,但是通过特殊方法,文件内的控制字符(尤其是文件结束字符)可以让纯文本不可见。

格式化文本(formatted text)与纯文本(plain text)相对,可以具有风格、排版等信息,如颜色、式样(黑体、斜体等)、字体尺寸、特性(如超链接)等。

格式化文本不等同于二进制文件,也不一定就不是 ASCII 文本。因为格式化文本不一定是二进制的,它也可以是一般的文本(例如 HTML、RTF 等文件),因此可以是 ASCII 文本文件。相反,纯文本(plain text)也可以非 ASCII 文件(如 UTF-8 编码的文件)。

作为文本文件的格式化文本是用标记语言来写的,不过 Microsoft Word 处理的格式化文本是二进制文件。

但是,从程序的角度来看,所有的文件都是数据流,文件系统为每一种文件格式规定了访问的方法(例如元数据等)。不同的操作系统都习惯性的采用各自的方式解决这个问题,每种方式都有各自的优缺点。

当然,对于现代的操作系统和应用程序,一般都需要这里所讲述的方法处理不同的文件。

- 扩展名(Filename Extension)
  扩展名是指文件名中,最后一个点(.)号后的字母串行。
  用扩展名识别文件格式的方式最先在数字设备公司的 CP/M 操作系统被采用,而后又被 DOS 和 Windows 操作系统采用。
  更改文件扩展名会导致系统误判文件格式。例如,将 filename.html 简单改名为 filename.txt 会使系统误将 HTML 文件识别为纯文本格式,因此 Windows Explorer 加入了限制向用户显示文件扩展名的功能。
  文件扩展名有着很大的缺陷或安全隐患,所以某些操作系统(例如 UNIX/Linux)已经不再遵循文件扩展名的规范,而是采用更精确的文件魔术数字(magic number)来确定文件类型。
- 特征签名(Signature)
  一种广泛应用在 UNIX 及其派生的操作系统上的方法是将一个特殊的数字存放在文件的特定位置里。
  最初这个数字一般是文件开始处的 2 个字节,现在一般是将任何可以独一无二字符串行都可以作为特征签名。例如 GIF 图形文件是将文件开始处的六个字节作为特征签名的,它可以是 GIF87a 或者 GIF89a。但也有些文件很难通过这种方式识别(比如 HTML 文件)。
  采用这种方式可以更好的防止对文件格式发生误判,并且特征签名可以给出关于文件格式的更详细的信息。这种方式的缺点是效率较低,特别是显示大量的文件时,由于每种特征签名具有不同的识别方式,将消耗系统大量的资源对文件格式进行判断。
  扩展名和元数据方式采用固定格式数据,可进行快速匹配,从而使应用程序可以利用特征签名来判断文件是否完整和有效。
- 元数据(Metadata)
  元数据与文件本身分开存放,因而可以将文件格式信息存放到磁盘特定的位置。
  不同的文件系统之间元数据可能需要转换,导致元数据的可移植性差。

苹果计算机的文件系统为每个文件的目录入口都存储了创建者和类型码。这些代码称作 OSType,例如一个苹果计算机创建的文件的创建者会是 AAPL 而类型也是 APPL。RISC 操作系统采用类似的系统,用一个 12 比特位的数字索引描述表。例如,十六进制的 FF5 代表 PoScript,文件类型就是 PostScript 文件。

NTFS、FAT12、FAT16 及 FAT32 文件系统可以保存额外的文件属性信息。一个文件可以有多种属性,它是由名字和与名字对应的值组成。例如,扩展属性“.type”用于判断文件的类型,可能是值包括“Plain Text”或“HTML document”。

ext2、ext3、ReiserFS、XFS、JFS 和 FFS 文件系统允许存储扩展的文件属性。它是由名字和与名字对应的值组成，名字应当是独一无二的。

最初在 RFC 1341 中说明的 MIME 最初是用于表示电子邮件的附件的类型，现在已经广泛地用于 Internet 有关的应用，并且正在被广泛地采用到其他的应用中。

MIME 用一个类型/子类型表示文件的类型。例如，text/html 代表文件是 HTML 文件，image/gif 表示 GIF 文件。

第 71 章

Text I/O

71.1 Comparision

程序使用变量存储各种信息(例如输入的数据、计算结果和运行过程中产生的任何中间值),但是这些信息只是瞬间存在的,一旦程序运行结束,变量的值就丢失了,因此能够永久存储信息是很重要的。

为了让信息存储在计算机上的时间比程序运行时间更长,一般的做法是收集信息并使其成为逻辑上结合更紧密的整体,并以文件的形式保存在永久存储介质中。

通常,文件存储在磁盘或其他存储介质中,但是其基本原理和操作模式是相同的。其中,包含文本的文件通常都是文本文件(text file),可以将其看作是存储在永久介质上的字符序列,并可以通过文件名来标识这些文件。

在计算机科学中,二进制文件一般指包含 ASCII 及扩展 ASCII 字符中编写的数据或程序指令的文件。

- 广义的二进制文件即为文件,由文件在外部存储设备的存放方式为二进制而得名。
- 狭义的二进制文件即指除文本文件以外的文件。

文件名及文件包含的字符之间的关系和变量名及其内容之间的关系是一样的。例如,文本文件可以被视作一种二维结构,即由单个字符组成的行序列。但从内部来看,文本文件是由一维的字符序列表示的,除了可见的打印字符外,文件还包含行结束符 '\n'。

文本文件和字符串在很多方面有相似之处,表现为如下两点:

- 文本文件与字符串都是由有序的字符集合组成的。
- 文本文件和字符串都有确定的结束点。其中,字符串的数据结束是以空字符表示的。

文本文件的结束是以一种特殊的文件结束标志来说明该标志之后再没有字符存在了。当从某一文件中读取字符时,必须检测文件结束标志,从而确定何时没有字符可读。

另一方面,字符串和文件有一些重要的区别,其中最重要的区别就是数据能否被永久保存。

- 字符串仅在程序运行的一段时间内临时存储在计算机内存中。
- 文件能被长期保存在存储设备中,直到被明确地删除或覆盖为止。

使用字符串和文件中的数据的方法也是不同的,具体表现在下面两个方面:

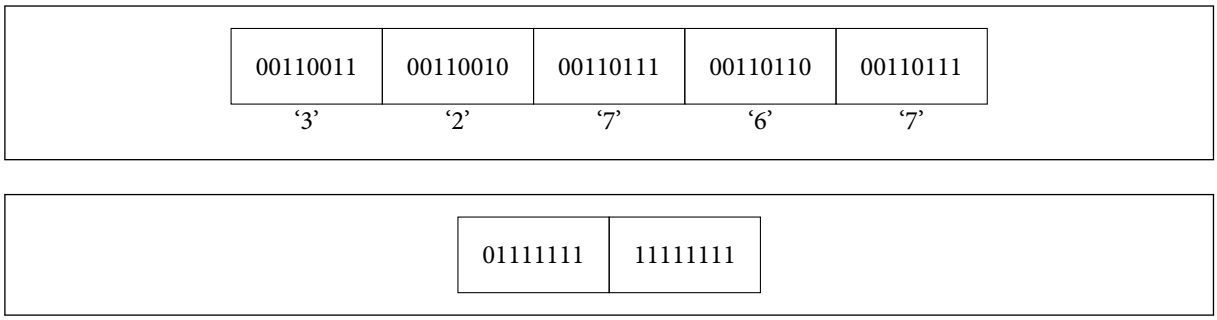
- 字符串是一个字符数组。指明适当的下标后就可以以任何次序选择字符。
- 程序对文件的读写通常是按顺序进行的。
  - 当程序读取一个已有的文件时,会从头到尾读取字符;
  - 当程序创建一个新文件时,会从第一个字符开始,按照字符顺序逐个写入字符。

stdio.h 支持两种类型的文件:文本文件和二进制文件。

在文本文件中,字节表示字符,这样就可以检查和编辑文本文件。但是,在二进制文件中的字节不一定表示字符,字节组还可以表示其他类型的数据(比如整数和浮点数)。在实际程序开发过程中,源文件一般都是文本文件,而可执行程序则很多都是二进制文件。

下面的示例说明文本文件和二进制文件之间的区别。如果以文本的形式保存数 32767,则可以把 3、2、7、6、7 作为字符存储起来,于是得到以 ASCII 字符集表示的 5 个字节如下:

如果以二进制形式存储数 32767,则只会占用两个字节,这样可以节省大量的空间。



无论使用文本文件还是二进制文件来存储数据，一个文件就是一个字节的序列。其中，文本文件按行进行划分，所以必须使用一些标志来标记每行的结尾，而且操作系统还可能使用特殊的字符来说明文本文件的结束。另一方面，二进制文件并不是按行进行划分的，而且二进制文件还可以合法的包含任何字符，因此也就不可能设置文件结束字符。

在 DOS 操作系统中，文本文件和二进制文件之间主要存在两方面的差异。

- 行的结尾。
  当文本文件中写入换行符时，次换行符会扩展成一对字符（\r\n 或 ^M），即换行符和跟随的回车符。与之对应的转换发生在输入过程中，但是把换行符写入二进制文件时，它就是单独一个字符（换行符）。
- 文件末尾。
  在文本文件中把字符 Ctrl+Z[1]（\x1a）设定为文件的结束标记。在二进制文件中字符 Ctrl+Z 没有特别的含义，处理方式和其他任何字符相同。

与此相对的，UNIX 操作系统对文本文件和二进制文件不进行区别，二者都会以相同的方式进行存储，每个 UNIX 文本文件在每行的结尾只有单独一个换行符，而且没有特殊字符来标记文件末尾。

当开发用来读写文件的程序时，需要考虑是文本文件还是二进制文件，特别是需要把文件的内容输出到屏幕上的情况。另一方面，文件复制程序就不能把要复制的文件设定为文本文件，否则就不能完全复制包含文件结束符的二进制文件。在无法确定文件是文本形式还是二进制形式时，安全的做法是把文件设定为二进制文件。

71.2 Stream

在 C 语言中，术语流（stream）意味着任意输入的源或任意输出的目的地。

- 小程序一般都是通过一个流（通常和键盘有关）获得全部的输入，并且通过另一个流（通常和屏幕相关）来写入全部的输出。
- 较大的规模的程序可能会需要额外的流，这些流常常表示为磁盘上的文件，但却可以和其他类型的设备相关联（例如打印机、网络端口、调制解调器和光盘驱动器等）。

下面将集中讨论磁盘上的文件，这类文件通用且容易理解，而且标准 I/O 库中的许多函数不仅可以处理表示成文件的流，也可以处理所有其他形式的流。

71.2.1 FILE Pointer

在 C 语言程序中流的访问是通过文件指针（file pointer）来实现的，文件指针拥有的类型为 FILE *。

使用文件指针表示的特定流具有标准化的名字，也就是标准 I/O 库中定义的标准文件（或标准流）。如果需要，可以声明一些额外的文件指针。

例如，除了标准流，在程序中可以定义其他文件流。

```
FILE *fp1, *fp2;
```

[1]不一定要在文本文件末尾有字符 Ctrl+Z，但是某些编辑器会把它放上。

操作系统通常会限制在任意某时刻打开的流的数量, 但是一个程序可以声明任意数量的 FILE * 类型的变量。

71.2.2 Standard File

标准 I/O 库定义的三个关于文件操作的特殊的标识符分别为 stdin、stdout、stderr, 它们作为 FILE * 常量可被所有程序使用, 称为标准文件(standard file)。

- 常量 stdin 指定标准输入文件, 它是用户的输入源。
- 常量 stdout 指出标准输出, 表示写给用户的输出数据所在的设备。
- 常量 stderr 代表标准错误文件, 用于报告任何用户应该看到的出错消息。

这三个标准流是备用的, 也就是说不能声明它们, 也无法打开或关闭它们。通常情况下, 标准文件都是指向计算机控制台, 这样就可以将 I/O 函数应用于控制台本身。

- 当从 stdin 读取数据时, 输入来源于键盘;
- 当向 stdout 或 stderr 写入数据时, 输出会出现在屏幕上。

不过, 某些系统为了方便起见可以改变上述联系, 所以标准输入会来源于某一文件, 或者标准输出会被分配给一些其他的文件或设备。

Table 71.1: 标准流

文件指针	流	默认的含义
stdin	标准输入	键盘
stdout	标准输出	屏幕
stderr	标准错误	屏幕

默认情况下, stdin 表示键盘, stdout 和 stderr 表示屏幕, 因此标准 I/O 库函数都是通过 stdin 获得输入, 并且用 stdout 进行输出的。

即使当前使用的系统不具备改变标准文件分配的能力, 标准文件的存在也是有益处的, 这表明了标准 I/O 库函数都可以直接用控制台操作, 这样使用 stdin、stdout 和 stderr 就可以直接读写字符、行或是格式数据。

71.2.3 Redirection

实际上, 某些操作系统允许通过所谓的重定向(redirection)机制来改变默认的标准流含义。例如, 在 UNIX 和 DOS 操作系统中可以强制程序从文件中而不是键盘获得输入。

下面的示例在命令行中的字符 < 后面跟上文件名来从文件中获得输入, 这被称为输入重定向(input redirection)。

```
$ test <in.dat
```

输入重定向的本质和是使 stdin 流表示为文件而非键盘, 但是重定向的绝妙之处在于程序不会意识到正在从文件中读取数据, 只是知道从 stdin 流中获得的任何数据都是从键盘上输入的。

输出重定向(output redirection)和输入重定向类似, 只是对 stdout 流进行重定向是通过在命令行中的字符 > 后面跟上文件名来实现的, 这样就可以把所有写入 stdout 流的数据保存到文件中。

```
$ test >out.dat
```

UNIX 和 DOS 系统还支持把输入重定向和输出重定向进行合并。

```
$ test <in.dat >out.dat
```

输出重定向的一个问题是会把本来需要写入 stdout 流的内容都写入文件中, 这样程序的错误报告也就只能在打开文件时才能看到, 而这些信息本来应该是出现在 stderr 流中的。

把错误信息写入 stderr 流而不是 stdout 流中，可以保证 stdout 发生重定向时，错误信息仍然会出现在屏幕上。

71.3 Operation

使用输入和输出重定向时，不需要打开、关闭文件或执行任何其他明确的文件操作。不过重定向在许多应用中受到限制，当程序依赖重定向时，它无法控制自己的文件，也无法知道这些文件的名字。

另外，重定向无法实现在同一时间输入/输出两个或多个文件。当重定向无法满足需要时，将终止使用标准 I/O 库提供的文件操作。

ANSI C 的标准 I/O 库提供的基本的文件操作是 C 语言的高级应用，其真正强大之处在于它允许客户以可移植的方式指定文件操作，而且这种方式是方便、有效的。

在 C 语言程序中使用文件时，需要完成以下工作：

1. 声明一个 FILE * 类型的变量。
2. 通过调用 fopen 函数将此变量和某实际文件相联系。
   这一操作称为打开文件(open)，打开一个文件要求指定文件名，并且指明该文件是用于输入(r)还是输出(w)。
3. 调用 stdio.h 中适当的函数完成必要的 I/O 操作。
   - 对于输入文件来说，这些函数从文件中将数据读取至程序中；
   - 对于输出文件来说，函数将程序中的数据转移到文件中去。
4. 通过调用 fclose 函数表明文件操作结束。
   这一操作称为关闭文件(close)，它断开了 FILE * 变量与实际文件之间的联系。

71.3.1 Declaration FILE *

标准 I/O 库中定义了一个称为 FILE 的类型，用于存储操作系统在追踪文件操作活动时所需的信息。

和使用任何指针一样，可以在 C 语言程序中声明一个 FILE * 变量。例如，下面的示例代码声明的变量 infile 就是一个指向 FILE 的指针类型变量。

```
FILE *infile;
```

不同的操作系统具有不同的文件系统结构，导致 FILE 类型的基本表示方法也不同。stdio.h 的主要目的是使我们无需考虑这些不同，这样从程序员就不需要知道任何具体的细节。

不论文件是否是为某一机器所定义的，所有操作文件时所需做的只是跟踪指向 FILE 结构的指针，相关细节的管理完全可以信任 stdio.h 的本地实现。

要为同时打开的多个文件分别声明 FILE * 变量。例如，如果程序要求从某一文件中读取信息并处理，然后将处理过的数据写入另一文件时，就需要声明两个 FILE * 变量。

```
FILE *infile, *outfile;
```

参数 FILE* 告诉函数如何对文件进行操作。例如，当调用函数从输入文件中读取数据时，应该把变量 infile 作为函数参数。同理，调用函数写入输出数据时需要指定变量 outfile。

71.3.2 Open File

当初次声明 FILE * 变量时，它和任何实际文件都没有关系。为了使 FILE * 变量和实际文件建立联系，必须调用函数 fopen，其格式如下：

```
file pointer ariable = fopen(file name,  mode);
```

使用流的方式打开文件要求调用 fopen 函数，其函数原型如下：

```
FILE *fopen(const char *filename, const char *mode);
```

其中，fopen 的第一个参数是一个用于指定文件名的字符串，文件名需遵循本地系统的文件命名规则，可能包含关于文件位置的信息（例如驱动器号和路径）。

在 fopen 函数调用中，一定要注意文件名中是否含有\字符，C 语言会把字符\看成是转义序列的开始标志。例如，下面的调用语句将始终无法执行，因为编译器会把\t 看成是转义字符。

```
fopen("c:\project\test.dat". "r"); // illegal
```

为了避免转义字符错误，可以用\\代替\：

```
fopen("c:\\project\\test.dat". "r"); // illegal
```

fopen 的第二个参数（“模式字符串”）也是一个字符串，用于确定数据传输方式。

模式字符串不仅依赖于需要对文件采取的操作，还取决于文件中的数据是文本形式还是二进制形式，通常采用以下几种方式之一：

- “r”
  以只读方式打开文件，fopen 返回的文件指针变量只能用于输入操作，且文件必须已经存在。
- “w”
  以写方式打开文件，fopen 返回的文件指针变量只能用于输出操作（文件不需要事先存在）。
  如果文件不存在，将会用指定的文件名创建一个新文件。如果已存在同名文件，那么此同名文件的内容会被删除。
- “a”
  以追加方式打开文件，该模式与“w”模式相似（文件不需要事先存在），返回的文件指针可以用于输出操作。与“w”模式不同的是，如果此文件已存在，向文件写入的新信息将添加在原数据的末端。
- “r+”
  打开文件用于读和写，从文件头开始。
- “w+”
  打开文件用于读和写，如果文件存在就覆盖先前内容。
- “a+”
  打开文件用于读和写，如果文件存在就追加。

例如，如果需要打开 test.txt 文件用于输入，并将其与变量 infile 相联系，那么可以使用下面的代码就可以读入数据。

```
infile = fopen("test.txt", "r");
```

fopen 函数返回一个文件指针，通常会把该指针存储到一个变量中，然后就可以在输入函数中将该指针变量作为实际参数来对文件进行操作。

如果无法打开文件，那么 fopen 操作将会失败。当需要的输入文件不存在或者发生其他错误（例如权限不足）时，fopen 调用将返回指针值 NULL，表示有错误发生。

永远不能假设可以打开文件，为了确保不会返回空指针，程序员有责任检查 fopen 函数的返回值并向用户报告所发现的错误，一种可行的办法是通过调用 Error 函数报告操作失败。

```
infile = fopen("test.txt", "r");
if(infile == NULL) Error("Can not open the file.");
```

实际上，无法找到输入文件很可能是在程序读入用户输入的文件名时发生的，这样可以无需让整个程序停止，而是让用户输入有效的文件名即可。

```
while(TRUE){
  printf("Input file name: ");
  filename = GetLine();
  infile = fopen(filename, "r");
```

```
    if(infile != NULL) break;
    printf("Can not open the file %s. Try again.\n", filename);
}
```

当使用 fopen 打开二进制文件时，需要在模式字符串中包含字母 b。

Table 71.2: fopen 打开二进制文件时的模式字符串

字符串	含义
“rb”	打开文件用于读
“wb”	打开文件用于写(文件不存在则创建)
“ab”	打开文件用于追加(文件不存在则创建)
“r+b”或“rb+”	打开文件用于读和写，从文件头开始
“w+b”或“wb+”	打开文件用于读和写(如果文件已存在就覆盖先前内容)
“a+b”或“ab+”	打开文件用于读和写(如果文件存在就追加)

标准 I/O 库对写数据和追加数据进行了区分。当向文件写数据时，通常会对先前的内容进行覆盖写。如果是以追加方式打开文件，实际上是在文件末尾进行添加，因而会保留文件的原始内容。

当对一个打开的文件执行既读又写的操作(模式字符串包含字符 +)时，如果未先调用一个文件定位函数，那么就不能把读转换为写，而且如果既没有调用 fflush 函数也没有调用文件定位函数，那么就不能把写转换为读。

71.3.3 I/O Operation

为了完成读出或写入实际数据的操作，应该根据应用选择一种策略。

最简单的方式就是使用 getc 和 putc 函数，逐个字符地读写文件。不过在很多情况下，逐行处理文件会更加方便。

为了达到此目的，stido.h 接口提供了 fgets 和 fputs 这两个函数，而且扩展接口 simpio.h 也提供了更简单的 ReadLine 函数，该函数能解决使用 ANSI 标准库函数时产生的很多问题。

在更高层次上，可以选择使用 fscanf 和 fprintf 函数来读写格式化数据，这样就可以将数字数据和字符串以及其他数据类型混合起来。

71.3.4 Close File

不管选择使用哪种 I/O 操作策略，在文件操作结束后都必须确保关闭所有打开的文件。

在 C 语言中，关闭文件指的是通过调用 fclose 函数并配合适当的文件指针来终止 FILE* 变量与文件的联系。例如，关闭和变量 infile、outfile 相关的文件可以使用下面的代码：

```
fclose(infile);
fclose(outfile);
```

fclose 函数可以使程序关闭不再使用的文件，而且 fclose 函数的实际参数必须是来自 fopen 函数或 freopen 函数的文件指针。

```
int fclose(FILE *stream);
```

如果 fclose 函数成功地关闭了文件指针，那么会返回零，否则会返回错误代码 EOF(在 stdio.h 中定义的宏)。实际上，退出程序时会自动关闭所有打开的文件，但是养成显式地关闭文件的习惯还是很有好处的。

- 可以容易地指出何时文件在使用以及何时文件关闭。
- 可以更容易地将程序组合到自己处理文件操作的较大程序中。

下面的示例说明了一个使用 fopen 函数和 fclose 函数的程序框架，该程序会打开文件 example.dat 并进行读数据操作，而且会检查打开是否成功，最后在程序终止前关闭文件。

```
#include <stdio.h>
#include <stdlib.h>

#define FILE_NAME "example.dat"
main()
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if(fp == NULL){
    printf("Can not open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }

  /* file operation */

  fclose(fp);
  return 0;
}
```

上述的 fopen 函数调用和 FILE * 变量声明可以组合到一起：

```
FILE *fp = fopen(FILE_NAME, "r");
```

另外，为了简化错误检查，也可以把检测文件打开是否成功的操作简化。

```
if((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

71.4 Add File

freopen 函数为已经打开的流附加一个不同的文件，最常应用于把文件和其中一个标准流相关联，这些标准流包括 stdin、stdout 和 stderr。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

例如，为了使程序开始往文件 foo 中写数据，可以使用下列形式的 freopen 函数调用：

```
if(freopen("foo", "w", stdout) == NULL){
  /* error, foo can not be opened. */
}
```

在关闭了任何先前与 stdout 相关联的文件之后（通过命名行重定向或者前一个 freopen 函数调用），freopen 函数将打开文件 foo，并且使该文件与 stdout 相关联。

freopen 函数通常返回的值是它的第三个实际参数（一个文件指针）。

- 如果无法打开新的文件，那么 freopen 函数会返回空指针。
- 如果无法关闭旧的文件，那么 freopen 函数会忽略掉错误。

当程序需要打开文件时，需要评估给程序提供文件名的方案的优劣，例如将文件名嵌入程序内部的做法无法提供更多的灵活性，而提示用户输入文件名是笨拙的做法，目前最好的解决方案是通过用户提供的命令行参数来获取文件名。

例如，当执行程度 demo 时，可以通过把文件名放入命令行的方式来给程序提供文件名。

```
$ demo name.dat time.dat
```

在 C 语言中，可以通过定义带有两个形式参数的 main 函数来访问命令行中的实际参数。

```
main(int argc, char *argv[])
{
  ...
}
```

其中，argc 是命令行实际参数的数量，，而 argv 是一个指针数组，数组中的指针都指向实际参数字符串。

- argv[0] 指向程序的名字。
- 从 argv[1] 到 argv[argc-1] 指向剩余的实际参数。
- argv[argc] 是空指针。

下面的示例程序可以检测文件是否存在，若存在则打开进行输入。在运行程序时，用户将给出要检测的文件的名字。

```
/* canopen.c */
/* Checks whether a file can be opened for reading */

#include <stdio.h>

main(int argc, char *argv[])
{
 FILE *fp;
 if (argc != 2) {
  printf("usage: canopen filename\n");
  return 2;
 }

 if ((fp = fopen(argv[1], "r")) == NULL) {
  printf("%s can't be opened\n", argv[1]);
  return 1;
 }

 printf("%s can be opened\n", argv[1]);
 fclose(fp);
 return 0;
}
```

如果在命令行中输入错误的实际参数数量，那么程序将输出信息“usage: canopen filename”来提醒用户 canopen 程序需要单独一个文件名。

另外，为了丢弃程序 canopen 的输出以及测试返回的状态值（如果可以打开文件，则值为 0，否则为 1），可以使用重定向。

71.5 Temporary File

现实世界中的程序经常需要产生临时文件，即只在程序运行时存在的文件。例如，C 语言编译器就会产生临时文件，编译时可能先把源代码翻译成一些存储在文件中的中间形式，在稍后把程序翻译成目标代码时编译器会读取这些文件，最后在完成了编译后就不需要保留程序的中间形式的文件来。

stdio.h 提供了两个函数来处理临时文件，即 tmpfile 函数和 tmpnam 函数。

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

tmpfile 函数产生临时文件，而且这些临时文件将保持到文件关闭或程序终止。tmpfile 函数的调用会返回文件指针，该指针可以用于访问临时文件。

```
FILE *tempptr;
tempptr = tmpfile(); /* create a temporary file */
```

如果产生临时文件失败，tmpfile 函数就会返回空指针。tmpfile 函数在方便使用的同时，也存在一些缺点：

1. 无法知道 tmpfile 函数产生的文件名；
2. 无法决定稍后是否要使文件成为永久性的。

如果上述关于 tmpfile 函数的限制导致 tmpfile 函数不适用，可以替换的解决方案就是用 fopen 函数产生临时文件。

为了产生具有新的可命名的临时文件，可以 tmpnam 函数为临时文件命名。

- 如果 tmpnam 函数的实际参数是空指针，那么 tmpnam 函数会把文件名存储到静态变量中，然后返回指向该静态变量的指针。

```
char *filename;
filename = tmpnam(NULL); /* creates a temporary file name */
```

- 如果为 tmpnam 函数提供了文件名，tmpnam 函数会把文件名复制到预定义的字符数组中。

```
char filename[L_tmpnam];
tmpnam(filename); /* creates a temporay file name */
```

L_tmpnam 在 stdio.h 中定义为一个宏，用来说明保存临时文件名的字符数组的长度。

在第二种情况中，tmpnam 函数也会返回指向临时文件名的指针。当传递指向 tmpnam 函数的指针时，一定要确信指针指向至少有 L_tmpnam 个字符的数组。

另外，还要当心不能过于频繁的调用 tmpnam 函数，在 stdio.h 中定义的宏 TMP_MAX 说明程序执行期间由 tmpnam 函数产生的临时文件名的最大数量。

71.6 File Buffer

从计算机体系结构上来看，从磁盘传出或传入信息是相对较慢的操作，这样的结果就是程序每次读写字符时无法直接访问磁盘文件来进行。

为了获得有效性能，引入了文件缓冲（buffering），这样写入流（output stream）的数据实际是存储在内存的缓冲区域内的。当缓冲区满了（或者关闭流）时，缓冲区会“清洗”（写入实际的输出设备）。

输入流（input stream）可以以类似的方式进行缓冲，缓冲区包含来自输入设备的数据，这样从缓冲区读数据就代替了从设备本身读数据，而从缓冲区读数据或则在缓冲区存储数据可以节约大量时间，因此在效率上可以取得更大的收益。

当然，还是需要把缓冲区的内容移动到磁盘，或者从磁盘移动到缓冲区，但是一个大的“块移动”比任何微小的字符移动要快得多。

71.6.1 fflush

标准 I/O 库会根据实际情况在后台自动完成缓冲操作，因此通常不用管理缓冲操作，仅有极少的情况下可能需要客户承担更主动的作用，C 语言标准库为此提供了 fflush 函数、setbuf 函数和 setvbuf 函数来手动管理缓冲。

```
int fflush(FILE *stream);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

当程序向文件中写输入时，数据通常是先保存到缓冲区而不直接保存到文件中。当缓冲区满了或者关闭文件时，缓冲区才会清空，而且调用 fflush 函数也可以手动清空文件的缓冲区。

- 为了清空和文件相关联的 fp 清空缓冲区，可以调用：

```
fflush(fp); /* flushes buffer for fp */
```

- 为了清空所有的缓冲区，可以调用：

```
fflush(NULL); /* flushes all buffers */
```

如果函数 fflush 调用成功，则返回零，否则返回 EOF。

71.6.2 setvbuf

setvbuf 函数允许改变缓冲流的方式，并且允许控制缓冲区的大小和位置。setvbuf 函数的第三个实际参数说明了期望缓冲区的类型：

- _IOFBF(满缓冲)
  当缓冲区为空时，从流读入数据，或者当缓冲区满时，向流写入数据。
- _IOLBF(行缓冲)
  每次从流读入一行数据或者向流写入一行数据。
- _IONBF(无缓冲)
  没有缓冲区，直接从流读入数据或者直接向流写入数据。

上述所有这三种宏都是在 stdio.h 中进行了定义。

setvbuf 函数的第二个实际参数(如果它不是空指针)指明了期望缓冲区的地址，缓冲区可以有静态存储期限、自动存储期限或是可以动态分配的。

通过使缓冲区自动化可以允许它的空间在块退出时可以被自动地重声明，其中动态分配缓冲区可以在不需要时释放缓冲区。

setvbuf 函数的最后一个实际参数是缓冲区内字节的数量，较大的缓冲区可以提供更好的性能，而较小的缓冲区可以节约空间，提高缓冲区利用率。

例如，下面的 setvbuf 函数的调用利用 buffer 数组中的 N 个字节作为缓冲区，而且设置 stream 的缓冲方式为满缓冲。

```
char buffer[N];
setvbuf(stream, buffer, _IOFBF, N);
```

注意，必须在打开 stream 后，而且执行任何其他在 stream 上的操作之前调用 setvbuf 函数。如果调用成功，setvbuf 函数返回零。如果要求的缓冲模式是无效的或者无法提供，那么 setvbuf 函数会返回非零值。

71.6.3 setbuf

setbuf 函数是一个较早期的函数，用于设置缓冲模式的默认值和缓冲区的大小。

```
void setbuf(FILE *stream, char *buf);
```

- 如果 buf 是空指针，那么 setbuf(stream, buf) 函数的调用就等价于

  ```
  (void) setvbuf(stream, NULL, _IONBF, 0);
  ```

- 如果 buf 是空指针，那么 setbuf(stream, buf) 函数的调用就等价于

  ```
  (void) setvbuf(stream, buf, _IOFBF, BUFSIZE);
  ```

这里，BUFSIZE 是在 stdio.h 中定义的宏，现在已经把 setbuf 看作是过时的，不建议在新程序中使用。

最后还是要提醒，在使用 setvbuf 函数或 setbuf 函数时，一定要确保在释放缓冲区之前已经关闭了流。

71.7 Remove File

remove 函数和 rename 函数允许程序执行基本的文件管理操作，而且不同于大多数文件操作函数，它们是对文件名而不是文件指针进行操作的。

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

如果调用成功,两个函数都返回零,否则返回非零值。

例如,如果需要删除文件 foo,可以执行下面的操作:

```
remove("foo");
```

如果程序使用 fopen 函数来产生临时文件,那么它可以使用 remove 函数在程序终止前删除临时文件,但要确信已经关闭了要移除的文件。

移除文件的效果就是当前打开的文件是由实现定义的。

71.8 Rename File

rename 函数改变文件的名字。

```
rename("foo", "bar"); /* renames "foo" to "bar". */
```

如果程序需要决定使文件变为永久的,那么 rename 函数是可以对用 fopen 函数产生的临时文件修改名字的。如果具有新的名字的文件已经存在了,那么结果由实现来定义。

如果打开了要修改名字的文件,那么一定要确保在调用 rename 函数之前该文件是关闭的。如果文件是打开的,则无法对文件进行改名。

第 72 章

Character I/O

如果对文件进行逐字符的处理，那么基本的操作是 getc 和 putc 函数，以及 ungetc 用于将一个字符写回到输入流中。

- 使用 getc(infile) 函数可以从 infile 指向的文件中读取下一个字符，并将其返回给调用函数。
- 使用 getchar 函数可以从标准输入文件中进行读取，因此与 getc(stdin) 作用相当。

在 stdio.h 中，getc 函数的原型如下：

```
int getc(FILE *infile);
```

getc 的函数原型指定返回一个整型值，虽然在概念上返回的是一个字符。这样设计 getc 的原因是因为返回一个字符会使得程序无法识别文件结束标记。

字符编码一共只有 256 个，且一个数据文件中可能包含其中的任意值，因此没有一个值（至少没有 char 类型的值）可以用作文件结束标记。

通过扩展 getc 定义，可以使得 getc 返回一个整型值来返回一个合法字符数据以外的值作为文件结束标记。通常在 stdio.h 中这个值被称为 EOF，尽管不能依赖 EOF 有内部值的事实，但是 EOF 还是有值-1。

> getc 返回一个整型值，而不是一个字符型的值。如果用字符型的变量存储 getc 的结果，程序就检测不到文件结束标记。

如果要写入一个单独的字符，可以用函数 putc(ch, outfile)，它将第一个参数写入指定的输出文件中。stdio.h 还包括函数 putchar(ch)，其定义与 putc(ch, stdout) 相同。

下面的示例将使用 getc 和 putc 函数来一个文件拷贝到另一个文件中。

```
/*
 * File: copyfile.c
 * ----------------
 * This program copies one file to another using character I/O.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void CopyFile(FILE *infile, FILE *outfile);
static FILE *OpenUserFile(string prompt, string mode);

/* Main program */

main()
{
   FILE *infile, *outfile;
```

```
   printf("This program copies one file to another.\n");
   infile = OpenUserFile("Old file: ", "r");
   outfile = OpenUserFile("New file: ", "w");
   CopyFile(infile, outfile);
   fclose(infile);
   fclose(outfile);
}

/*
 * Function: CopyFile
 * Usage: CopyFile(infile, outfile);
 * ---------------------------------
 * This function copies the contents of infile to outfile. The
 * client is responsible for opening these files before calling
 * CopyFile and for closing them afterward.
 */

static void CopyFile(FILE *infile, FILE *outfile)
{
   int ch;

   while ((ch = getc(infile)) != EOF) {
      putc(ch, outfile);
   }
}

/*
 * Function: OpenUserFile
 * Usage: fileptr = OpenUserFile(prompt, mode);
 * --------------------------------------------
 * This function prompts the user for a file name using the
 * prompt string supplied by the user and then attempts to
 * open that file with the specified mode. If the file is
 * opened successfully, OpenUserFile returns the appropriate
 * file pointer. If the open operation fails, the user is
 * informed of the failure and given an opportunity to enter
 * another file name.
 */

static FILE *OpenUserFile(string prompt, string mode)
{
   string filename;
   FILE *result;

   while (TRUE) {
      printf("%s", prompt);
      filename = GetLine();
      result = fopen(filename, mode);
      if (result != NULL) break;
      printf("Can't open the file \"%s\"\n", filename);
   }
   return (result);
}
```

其中，在 copyFile 中的 while 循环通过嵌入式赋值语句将读入字符和检测文件结束标记的操作结合起来。当程序判断 while 循环条件时，首先从下面的判断子表达式开始。

```
ch = getc(infile)
```

该表达式读入一个字符并将其赋给 ch，而且在执行循环体之前不能确保所赋的值不是 EOF，因此赋值语句外的括号是必不可少的，否则该表达式将把 ch 和 EOF 的比较结果赋给 ch，这是不正确的。

这一循环的作用与下面的长表达式是相同的，而上述的简写形式也是在绝大部分的 C 语言程序中的常用写法。

```
while(TRUE){
  ch = getc(infile);
  if(ch == EOF) break;
  putc(ch, outfile);
}
```

72.1 Update File

如果不是完全复制一个文件，也可以以类似的程序来实现在读的同时对字符进行转换。例如，下面的循环将数据从 infile 复制到 outfile，并将所有字符转化为大写形式。

```
while((ch = getc(infile)) != EOF){
  putc(topper(ch), outfile);
}
```

在现实中的很多情况下并不需要一个新文件，而只是需要修改现有文件，对现有文件进行修改的过程称为更新(update)文件。

更新文件的过程并不简单，因为对大多数系统而言，如果一个文件已经为进行输入而打开，这样锁定的文件就不允许再为输出打开。根据不同系统上的不同文件执行方式，导致调用哦个 fopen 会导致调用失败或损毁原来文件的内容。

更新一个文件最常用的办法是将新数据写入一个临时文件，在写好更新文件的所有内容后用这个临时文件替换原来的文件。

如果要实现一个将某个文件的字符全部转换为大写的程序，需要执行下面的步骤：

1. 打开原来的文件以便读入。
2. 打开一个临时文件以便写入，而且临时文件名不能和原来的文件重名。
3. 将输入文件复制到临时文件，并将所有小写字母替换为大写字母。
4. 关闭这两个文件。
5. 删除原来的文件。
6. 用原来的文件名字重命名临时文件。

在实现这一策略时，需要使用三个定义在 stdio.h 中的函数——tmpnam、remove 和 rename。

虽然我们可以为临时文件取任何名字，但是 stdio.h 接口中导出的函数 tmpnam 可以为临时文件自动生成文件名。

文件命名的习惯因机器不同而不同，调用函数 tmpnam(NULL) 会返回一个字符串，它的值适合作为当前机器上临时文件的名字，这样就可以按照如下代码创建并打开一个新的临时文件。

```
temp = tmpnam(NULL);
outfile = fopen(temp, "w");
```

当函数 tmpnam 以 NULL 作为参数调用时，返回一个指向标准 I/O 库实现专用的内存的指针，因此在结束第一个临时文件之前生成第二个临时文件是不安全的。如果必须这样做，可以用 CopyString 或 strcpy 将 tmpnam 的结果复制到另外的内存中。

删除一个文件只需调用函数 remove(name) 即可，此处的 name 是一个文件名。重命名一个文件同样非常简单，可以通过调用函数 rename(oldname, newname) 来完成。

和 ANSI 库中的很多其他函数一样，remove 和 rename 函数在调用成功后返回 0，调用失败后返回一个非零值。虽然有时会遇到将这些函数作为过程使用的情况，但检验返回值以确保调用成功是更安全的一种做法。

```
/*
 * File: ucfile.c
 * --------------
 * This program updates the contents of a file by converting all
 * letters to upper case.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void UpperCaseCopy(FILE *infile, FILE *outfile);

/* Main program */

main()
{
   string filename, temp;
   FILE *infile, *outfile;

   printf("This program converts a file to upper case.\n");
   while (TRUE) {
      printf("File name: ");
      filename = GetLine();
      infile = fopen(filename, "r");
      if (infile != NULL) break;
      printf("File %s not found -- try again.\n", filename);
   }
   temp = tmpnam(NULL);
   outfile = fopen(temp, "w");
   if (outfile == NULL) Error("Can't open temporary file");
   UpperCaseCopy(infile, outfile);
   fclose(infile);
   fclose(outfile);
   if (remove(filename) != 0 || rename(temp, filename) != 0) {
      Error("Unable to rename temporary file");
   }
}

/*
 * Function: UpperCaseCopy
 * Usage: UpperCaseCopy(infile, outfile);
 * --------------------------------------
 * This function copies the contents of infile to outfile,
 * converting alphabetic characters to upper case as it does so.
 * The client is responsible for opening and closing the files.
 */

static void UpperCaseCopy(FILE *infile, FILE *outfile)
{
   int ch;

   while ((ch = getc(infile)) != EOF) {
      putc(toupper(ch), outfile);
   }
```

```
}
```

72.2 Redo Read

从一个输入文件中读取数据时，经常会发生这样的问题——即直到读取了多余的字符后才发现早就应该停止读取了。例如，假设从一个文件读取字符以找出一个由十进制数组成的数字时，只要读到的是一个数字字符就需要继续读取下一个字符，于是得到读取字符直到非数字字符出现的循环如下：

```
while(isdigit(ch = getc(infile))){
  ...
}
```

如果当读到第一个非数字字符时才发现已经找到了一个十进制数，此时该非数字字符就成为循环结束的标志，但是它也很可能是下一次读文件时所需的值。通过调用 getc，已经将该字符读取到变量 ch 中，并且将它从输入流中取出了。

为了满足将读入的字符退回到原文件流的需求，C 语言提供了 ungetc(ch, infile) 函数来将字符 ch “推回”到原来的输入流中，作为下一次调用 getc 的返回值。

但是，C 语言库函数只能保证将一个字符推回到原输入文件，因此无法完成将多个字符全部都“推回”到输入文件的操作。不过实际上，能够推回一个字符就已经可以满足绝大多数情况的需要。

下面的示例程序将从一个文件复制到另一个文件，并且在此过程中删除所有的注释语句。

在 C 语言中，注释语句以字符序列/* 开始，以序列 */结束，因此删除注释语句的程序必须能在检测到开始标记/* 前复制字符，在检测到开始标记之后逐一读字符但不进行复制，直到检测到结束标记为止。

这个示例程序的需求难点在于注释标记是由两个字符组成，而每次从文件中只能复制一个字符，这样遇到/时需要有不同的处理策略。

- 如果/是注释标记的开始，此时就不能将其复制到输出文件中。
- 如果/是除号，此时就继续进行读入和复制。

但是，只有读入下一个字符后才能进行判断，如果下一个字符是“*”，那么要将两个字符都忽略掉，并标识出已经进入注释语句，只进行读入但不进行复制。如果下一个字符不是“*”，则需将其推回到输入流中，留待下一个循环周期中再复制它。

综合上述解释，下面使用 CopyRemovingComments 函数来实现这一策略。

```
static void CopyRemovingComments(FILE *infile, FILE *outfile)
{
  int ch, nch;
  bool commentFlag;

  commentFlag = FALSE;
  while((ch = getc(infile)) != EOF){
    if(commentFlag){
      if(ch == '*'){
        nch = getc(infile);
        if(nch == '/'){
          commentFlag = FALSE;
        }else{
          ungetc(nch, infile);
        }
      }else{
        if(ch == '/'){
          nch = getc(infile);
          if(nch == '*'){
```

```
            commentFlag = TRUE;
          }else{
            ungetc(nch, infile);
          }
        }
        if(!commentFlag) putc(ch, outfile);
      }
    }
  }
}
```

第 73 章

Line I/O

文件通常被划分为行，因此产生了每次读入整行数据的需求，为此 stdio.h 提供了执行行输入函数 fgets。

```
string fgets(char buffer[], int bufSize, FILE *infile);
// char *fgets(char *s, int size, FILE *stream);
```

fgets 函数的作用在于将下一行文件读入字符数组 buffer 中。通常，fgets 在读入第一个换行符后停止读入，但如果该行的长度超过了由函数参数 bufSize 限定的长度，则 fgets 函数将提前返回，因此 buffer 数组中表示终止的空字符前应是一个换行符，除非该行文件过长，超出 bufSize 的限制。

无论 fgets 读入的是一整行还是此行的一部分，通常情况下都会返回一个指针，指向函数中的第一个参数（即字符数组 buffer）。如果 fgets 在文件的末尾调用，则返回 NULL。

相应的行输出函数为 fputs，调用 fputs 函数可以将字符从一个字符串复制到输出文件，直到到达该字符串的末尾为止，其函数原型如下：

```
void fputs(string str, FILE *outfile);
// int fputs(const char *s, FILE *stream);
```

下面的示例将使用 fgets 和 fputs 来实现 CopyFile 函数。

```
static void CopyFile(FILE *infile, FILE *outfile)
{
  char buffer[MaxLine];

  while(fgets(buffer, MaxLine, infile) != NULL){
    fputs(buffer, outfile);
  }
}
```

当使用函数 fgets 时，必须为输入行提供一个缓冲区，因此在 CopyFile 的实现中，使用下面的声明在当前帧中显式分配字符数组的空间：

```
char buffer[MaxLine];
```

需要注意的是，该缓冲区的内存空间将在函数返回时释放。如果想要将行中的字符更长久地存储起来，需要将它们存储在函数调用结束后仍然存在的内存空间中。在某些情况下，这样的内存是一个全局字符数组，或是从调用函数传递过来的数组，但是通常来说，从堆中动态分配内存是一种更简便的办法。

如果对多次 fgets 调用使用同一个临时缓冲区，则需要为之分配新的内存空间，仅仅有赋值是不够的。例如，下面的示例从一个输入文件中读入两行数据，并将它们存储在字符串变量 line1 和 line2 中。

```
void ReadTwoLines(FILE *infile)
{
  string line1, line2;
  char buffer[MaxLine];

  fgets(buffer, MaxLine, infile);
  line1 = buffer;
  fgets(buffer, MaxLine, infile);
```

```
  line2 = buffer;
  ...
}
```

上述代码产生了不是我们想要的结果，line1 和 line2 的结果相同。出现这样的问题，原因在于变量 buffer 虽然在概念上是一个字符串，并且在内存中占有特殊的空间，但是下面的赋值语句仅仅将 buffer 的地址赋给了变量 line1。

```
line1 = buffer;
```

当程序继续执行，要重新利用这块空间读入第二行内容时，同一块空间就用来存放第二个字符串了，这样 line1 和 line2 最终都指向了相同的内存空间，因此二者的值是相同的字符串。

> 如果使用同一个字符缓冲区从同一个文件中读入多行，要记住在读入下一行数据之前，需要将数据从缓冲区中复制到其他存储空间。如果不进行这样的操作，前一行的内容将被覆盖。

为了解决这个问题，可以使用扩展库 strlib 中的 CopyString 函数来将每一个字符串写入各自的堆内存中。

```
void ReadTwoLines(FILE *infile)
{
  string line1, line2;
  char buffer[MaxLine];

  fgets(buffer, MaxLine, infile);
  line1 = CopyString(buffer);
  fgets(buffer, MaxLine, infile);
  line2 = CopyString(buffer);
  ...
```

另外，在扩展库 simpio 中定义的函数 ReadLine 也可以避免与 fgets 相关的其他问题。

- 难以确定缓冲区的大小。有些文件中的行比较长，因此很难选择一个适合所有文件的缓冲区的长度。
- 难以判断是否超出了缓冲区的界限。如果在调用函数 fgets 的过程中给定一个最大值，则意味着数据不会被写到所分配的缓冲区之外，但是同样需要知道 fgets 是否读入了一个完整的行。使用 fgets 时的唯一做法就是浏览缓冲区内的所有字符，检测是否包括换行符。
- fgets 存储换行符通常也会引起麻烦。在大多数的程序中，换行符只是一行结束的标记，而并不真正是数据的一部分。使用 fgets 意味着还需要采取其他的步骤将缓冲区中的换行符删除。

由扩展库引入的 ReadLine 函数和 GetLine 函数很相似，它们唯一的区别在于其输入数据来自作为参数传递的一个数据文件。与 fgets 相比，ReadLine 具有以下优点：

- ReadLine 在需要时自动分配堆内存，使得缓冲区不可能溢出。
- ReadLine 删除了标记每一行结束的换行符，所返回的数据只包含这一行中的字符。
- ReadLine 返回的每一个字符串都保存在各自的内存中，因此在存储一个字符串之前不必考虑是否需要进行复制。
- ReadLine 在遇到文件结束符时返回 NULL。

第 74 章

Format I/O

在标准 I/O 库提供了多种用于输入/输出操作的函数，可以根据不同需要进行选择，其中格式化 I/O 函数 scanf 和 printf 可以按照格式化的字段对来自标准输出和标准输入的数据进行转换。例如，格式化读/写函数可以在输入时把字符格式的数据转换为数字格式的数据，并且可以在输出时把数字格式的数据转换为字符格式的数据。

相比之下，标准 I/O 库也提供了读/写非格式化数据的函数 getc/putc、gets/puts 以及相关的函数，而且这些函数无法完成数据格式的转换。

- getc/putc：逐字符[1]地处理文件，每次读写一个字符。
- gets/puts：逐行地处理文件，每次读写一行字符。
- fread/fwrite：读写数据块。

对格式化输入/输出函数进行扩展又引入了 fprintf、fscanf 以及 sprintf 和 sscanf 函数。

- fscanf 和 fprintf 函数可以对文件进行格式化的 I/O 操作，因此 fprintf 函数也被称为 printf 函数的“文件版”。
- sscanf 和 sprintf 函数可以对字符缓冲区进行格式化 I/O 操作。

另外，stdio.h 中的 perror 函数、vfprintf 函数、vprinf 函数和 vsprintf 函数和 C 语言库中的其他内容关系紧密。

74.1 printf

函数 printf 具有三种不同的形式：

```
int printf(control string, ...);
int fprintf(output stream, constrol string, ...);
int sprintf(character array, control string, ...);
```

fprintf 函数和 printf 函数为输出流写入可变的数据项，并且使用格式串来控制输出的形式，二者都可以有可变数量的实际参数，返回值都是写入的字符数。若出错则返回一个负值。

- 函数 printf 通常将输出直接写入标准输出。
- 函数 fprintf 与 printf 基本相同，只是以一个 FILE 指针作为第一个参数，并将其输出写入该文件。
- 函数 sprintf 则以一个字符数组作为第一个参数，并将 printf 调用要显示的字符写入该数组，因此 sprintf 的调用函数应该确保数组空间足够大以容纳输出数据。

fprintf 函数和 printf 函数唯一的不同就是 printf 函数始终向标准输出流 stdout 中写入数据，而 fprintf 则是向其第一个实际参数指明的文件流中写入数据，因此 printf 函数的调用等价于 fprintf 函数把 stdout 作为第一个实际参数而进行的调用。

```
printf("Total: %d\n", total); /* writes to stdout */
fprintf(fp, "Total: %d\n", total); /* writes to fp */
```

除了输出的目的地不同之外，三种形式的 printf 函数的工作方式是相同的：它们将 control string 中的内容逐字符地复制到指定的目的地。

[1] putchar 等价于 putc(ch, stdout)，而 getchar 等价于 getc(stdin)。

- 如果该字符串中包含一个百分号(%),则该字符标志着格式码的开始,由函数 printf 中下一个参数中的字符串代替。
- 输出的格式由格式码末尾的字母决定,包括指明字段宽度、精度和对齐方式的修饰符等。

事实上,fprintf 函数并不只是把数据写入磁盘,它可以把数据写入任何输出流,而且 fprintf 函数最普遍的应用之一就是向 stderr 中写入出错信息。即使客户重定向 stdout,向 stderr 中写入的信息也会出现在屏幕上,标准错误流和磁盘文件是没有任何关系。

```
fprintf(stderr, "Error: data file can not be opened.\n");
```

在 stdio.h 中还有两个函数也可以向流写入格式化的输出——分别是 vfprintf 函数和 vprintf 函数,这两个函数都依赖于 stdarg.h 中定义的 va_list 类型。

74.1.1 Conversion Description

74.2 scanf

与 printf 函数对应,scanf 函数用来读入不同基本类型的值。

printf 和 scanf 有着很多不对称的地方,原因在于 C 语言的规则以及输出方向的不同转换等,导致了它们的用法也不相同。其中,scanf 和 printf 最重要的不对称性在于,printf 需要从其调用函数处获得多个值,而 scanf 则要将多个值返回给它的调用函数。

C 语言并不能很好地支持返回多个值的操作,因为这需要频繁地使用指针,由此也可能导致在 scanf 的调用过程中遗漏地址运算符(&),而且编译器无法识别这种错误。编译器可以接受不正确的 scanf 调用,但之后将导致整个程序莫名其妙地失败。

和 printf 一样,scanf 也有三种不同的形式。

```
scanf(control string, ...);
fscanf(input stream, control string, ...);
sscanf(character string, constrol string, ...);
```

- scanf 函数从标准输入中读取数据。
- fscanf 函数从由输入流参数指定的 FILE 指针处读入。
- sscanf 从指定的字符串中读入。

三种不同形式的 scanf 函数都是从某个源读入字符,并按照控制字符串中指明的方式进行转换,然后将数据值存入由调用函数通过额外的参数指定的内存中。

scanf 函数必须返回信息给调用函数,因此控制字符串后的参数必须使用应用调用,也就是说控制字符串后的每个参数都应该是一个指针。在大多数情况下,可以在变量名前用一个取址运算符(&)来将变量转化为它的地址,但必须注意字符数组的数组名本身已经是一个指针了,无需再对它进行取址运算。

scanf 函数的控制字符串由三类字符组成。

- 作为空格出现的字符。

  scanf 函数通常跳过空白字符去读下一个非空白的字符,这样的字符在 C 语言中被称为空白字符(white-space character),ctype.h 中的谓词函数 isspace 遇到这些字符时返回 TRUE。

  最常见的空白字符主要有空格字符、制表符和换行符等,在 scanf 函数的控制字符串中,空格的数量与输入中空格的数量相同。
- 百分号以及跟在其后的转换说明。
- 其他字符。

  这些字符必须与输入中的下一个字符相匹配,这也使得程序能够检查所需的标点符号(例如两个数字之间的逗号等)。

转换说明在结构上与 printf 相似,但是可用的选项是不同的。scanf 的转换说明由以下的几种选项组成,而且必须遵循下面的顺序:

scanf 函数在控制字符串后的每个参数都必须是一个指针。对于简单变量来说，前面的 & 表明取的是它的地址。如果该参数是一个字符数组，则该变量的名称已经自动解释为指针，因此无需再进行 & 运算。

- 由星号(*)表示的赋值屏蔽标志，该标志表明不将输入流中的值读入参数指定的地方。
- 一个可选的数字字段宽度，指出该字段可以读入的最大的字符个数。
- 大小的说明，h 表示 short 类型的整数值，l 表示 long 类型的整数或 double 类型的浮点数。
- 转换说明符。

scanf 函数的三种形式都会返回成功转换后的数据，而忽略由 * 号屏蔽掉的数据。如果在转换之前读到文件结束标志，则 scanf 函数会返回 EOF。

Table 74.1: scanf 函数的转换说明

代码	说明
%d	以十进制整数的形式读入下一个值，并将该整数值存入下一个指针参数指定的存储单元。变量的大小应与说明指定的大小相匹配。
%f、%e、%g	以浮点数的形式读入下一个值，并将该浮点数存入下一个指针参数指定的存储单元。如果指针类型的目标为 float，则转换说明应为%f。如果指针类型的目标为 double，则转换说明应为%lf。%e 和%g 与%f 是相同的，它们是为了与 printf 函数相对应而设置的。
%c	读入下一个字符，并将其存入下一个参数指定的地址，该参数必须是一个字符指针。与其他转换说明不同的是，%c 在转换时不会跳过空白字符。
%s	读入一个字符串，并将其存入下一个个参数指定的字符数组的连续单元中。调用程序必须确保该字符数组有足够的空间存储读入的值。遇到空白字符时停止读入。
%[...]	转换说明可由一对方括号 [] 及其括住的字符集组成。在这种情况下，读入一个字符串并将其存入下一个参数指定的地址中，直到遇到不包含在字符集中的字符时停止读入。
%[^...]	如果字符集是由 ^ 开始，则读入字符集以外的字符。例如，转换说明为%[0123456789]，则读入由数字组成的字符串。如果转换说明为%[^.!?]，则读入的字符不包括句号、叹号及问号。
%%	不进行任何转换，表示读入一个百分号。

在使用读入字符串的转换符（%s、%[...]、%[^...]）时，调用程序必须负责分配足够的空间来存放数据，因此在 scanf 中用来存放字符串的参数应该为调用程序声明的一个数组。例如，为了读入一个由空白字符界定的字符串，必须事先声明一个数组来存放字符串。

```
#define MaxWord 25
char word[MaxWord];
```

为字符数组分配空间后，就可以通过下面语句来读入字符串：

```
fscanf(infile, "%s", word);
```

要注意的是，在变量名的前面没有 & 字符，函数 fscanf 要求它的所有参数都为指向存储空间的指针，而数组名本身就是指向数组中第一个元素的指针。

上述例子的一个主要问题是存储空间很可能发生溢出。如果没有分配足够的内存空间，fscanf 将继续写入数据，占据后面的内存，破坏这些内存中原有的数据。为了防止缓冲区溢出，可以指定一个字段宽度，表示要读入的最大字符数。

如果 MaxWord 被定义为 25，那么下面的函数调用仅将最开始的 24 个字符存入数组，数组中仍有空间可以写入 '\0' 表示字符串结束，这样就可以保证不占用数组之外的其他空间。

```
fscanf(infile, "%24s", word);
```

其实, 在 fscanf 的控制字符串中可以避免使用常数 24, 虽然这种做法比较笨拙, 而且会使程序的可读性降低。如果要将常数与 MaxWord 的值联系起来, 应该通过调用 sprintf 函数生成 scanf 的控制字符串。

```
char controlString[MaxControlString];

sprintf(controlString, "%%%ds", MaxWord - 1);
fscanf(infile, controlString, word);
```

当函数 fscanf 被调用时, controlString 这个字符数组的内容为字符串 “%24s”。

在读入具有特定格式的输入时, 转换选项%[...] 和%[^...] 特别有用。例如, 为了读入具有下列格式的信息:

```
name:value
```

其中, name 为一个字符串, 而 value 为一个整数值, 可以按照下面的代码来调用 fscanf:

```
fscanf(infile, "%[^:]: \%d", name, &value);
```

上述做法的缺陷在于不能重复使用该语句以同一种形式读入多行数据, 原因在于当程序读入与 value 相对应的整型值后, 表示数据结束的换行符仍然留在输入流中, 并将被作为下一个字符被读入, 为此可以在控制字符串的最后增加一个换行符来解决这个问题。

```
fscanf(infile, "%[^:]: \%d\n", name, &value);
```

不过仍然无法满足需求, 包含一个换行字符不要求 fscanf 的输入流中有相匹配的换行符。相反地, 换行符会被看作是一个空白字符, 使 fscanf 跳过输入流中的任何空白字符。

- 如果假设换行符总是紧跟在数字后面, 那么一切正常。
- 如果希望程序能够检测出格式不正确的输入文件, 则需要使用其他的方案。

首先, 一种比较安全的方案是在调用 fscanf 时读入一个额外的字符并检测它, 确保它是一个换行符。

```
while(TRUE){
  nscan = fscanf(infile, "%[^:]: %d%c", name, &value, &termch);
  if(nscan == EOF) break;
  if(nscan != 3 || termch != '\n') Error("Bad input line.\n");
}
```

当使用该方案时, 程序同样也保证了 fscanf 正好读入了三个项目: 名称、值以及终止字符。

如果需要知道在出现错误时已读入多少文件内容, 最好用 fgets 读入一个完整行, 然后调用 sscanf 函数转换结果字符串中的字段。

```
while(fgets(line, MaxLine, infile) != NULL){
  nscan = sscanf(line, "%[^:]: %d\%c", name, &value, &termch);
  if(nscan != 3 || termch != '\n') Error("Bad input line.\n");
}
```

74.3 example

下面使用 scanf 函数从示例文件 elements.dat 中读入数据, 该文件为每一个化学元素列出下列信息:

- 元素名称: 通常不超过 15 个字符。
- 化学符号: 通常不超过 2 个字符。
- 原子序号: 表示原子核中质子数的一个整数。
- 原子量: 一个浮点数(用来表示该元素的各同位素的平均质量)。

下面是 elements.dat 的内容, 其格式与 CSV 文件相似, 以逗号将数据进行分隔。

```
Hydrogen, H, 1, 1.008
Helium, He, 2, 4.003
Lithium, Li, 3, 6.939
Beryllium, Be, 4, 9.012
Boron, B, 5, 10.811
Carbon, C, 6, 12.011
Nitrogen, N, 7, 14.007
Oxygen, O, 8, 15.999
Fluorine, F, 9, 18.998
Neon, Ne, 10, 20.183
```

为了按照原子序号、元素名称、化学符号和原子量的顺序并格式化输出结果，第一步是为 fscanf 设计一个控制字符串，以便正确地从文件中读入一行。

输入行由元素名称开始，元素名称是用逗号标志结束的字符串，因此最简单的方案是利用转换说明%15[^,] 来读入第一个逗号前出现的字符，而且将字段宽度设置为 15 也保证了 elementName 的缓冲区不溢出。

控制字符串在此转换符后应该跟一个逗号，与输入流中的内容相对应，此后为一个空格，用来跳过逗号后面的所有空格。

化学符号通过转换说明%2[^,] 来读入，情况与前一个转换说明基本相同，只是字段宽度较小。

最后两个字段分别为 int 和 double 类型的数值，因此为读入这些数值还应该在控制字符串中包括转换说明%d 和%lf，用来读入原子量的转换说明必须加入字母 l，它所对应的变脸是 double 类型的。

在行结束处，需要将原子量后面的字符也读入，并确保其为一个换行符。

综上所述，得到完整的 fscanf 语句如下：

```
nscan = fscanf(infile, "%15[^,], %2[^,], %d, %lf%c",
  elementName, elementSymbol,
  &atomicNumber, &atomicWeight, &termch);
```

其中，fscanf 中的前两个变量为字符数组，已经被视为地址，而变量 atomicNumber、atomicWeight 和 termch 变量都不是数组，因此需要在它们前面加上 & 来进行取址操作。

格式化 I/O 函数在生成输出表时也很有用，其中表中的元素名称和它对应的化学符号显示在一个字段中。另外，为了使原子量一栏能够精确地对齐，就需要确保名称和符号字段有固定的宽度，这里可以使用 strlen 函数计算每一个字符串，然后将其写入合适的空间，或者可以使用 sprintf 函数将两段合并为一个字符串，然后通过 printf 中的标准字段宽度生成正确的输出。

```
/*
 * File: elements.c
 * ----------------
 * This program copies the information from the elements.dat
 * file into a table formatted into fixed-width columns. The
 * data values in the file are read using fscanf.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * ElementsFile -- Name of the elements data file
 * MaxElementName -- Maximum length of element name
 * MaxSymbolName -- Maximum length of element symbol
 */

#define ElementsFile "elements.dat"
```

```
#define MaxElementName 15
#define MaxSymbolName 2

/* Main program */

main()
{
   FILE *infile;
   char elementName[MaxElementName+1];
   char elementSymbol[MaxSymbolName+1];
   char namebuf[MaxElementName+MaxSymbolName+4];
   int atomicNumber;
   double atomicWeight;
   char termch;
   int nscan;

   infile = fopen(ElementsFile, "r");
   if (infile == NULL) Error("Can't open %s", ElementsFile);
   printf("  Element (symbol) Atomic Weight\n");
   printf("-------------------------------------\n");
   while (TRUE) {
      nscan = fscanf(infile, "%15[^,], %2[^,], %d, %lf%c",
                     elementName, elementSymbol,
                     &atomicNumber, &atomicWeight, &termch);
      if (nscan == EOF) break;
      if (nscan != 5 || termch != '\n') {
         Error("Improper file format");
      }
      sprintf(namebuf, "%s (%s)", elementName, elementSymbol);
      printf("%3d. %-20s %8.3f\n", atomicNumber, namebuf, atomicWeight);
   }
}
```

scanf 的优势在于为读取固定格式的输入提供了一个简便的方案。通常只要确定一个正确的转换说明，就可以通过调用一个函数来读取很多数据项。特别是需要编写测试应用的代码并且读入一些测试数据时，scanf 不失为一个很好用的工具，这种情况下，测试数据及其格式都应该是已知的。

但是，scanf 函数往往对于无法控制来源的输入流束手无策。如果输入数据中可能存在错误，程序就必须测试这些输入数据以确保它们具有正确的格式。但是，使用 scanf 时通常不可能彻底检查输入数据。

在商业化的 C 语言程序中，错误检测是极其重要的一个环节，因此 scanf 并没有得到广泛的使用，对此 ANSI 库标准化委员会主席 P.J.Plauger 给出了下面的提示：

> 输入的转换说明并不像输出的转换说明那样完整。很多时候会发现不可能确定哪里出现了错误，并适当地恢复错误，因此多年的经验说明输入函数的用途实在有限。

Part XII

Error Handling

第 75 章

Introduction

真正商用软件都必须“非常健壮”——能够从错误中恢复正常而不至于崩溃。为了使程序非常健壮，需要开发者能够预见程序执行时可能遇到的错误，包括对每个错误进行检测，并在错误发生时提供合适的行为来处理。

程序中有很多种方法报告执行过程中出现的错误情况，例如可以使用定义在扩展库 genlib.h 中的 Error 函数向用户报告错误。当然，在实际的开发中使用的更为标准的方法。

- 调用 assert 函数；
- 查询 errno 变量。

在程序中响应一个错误的过程称为错误处理（error handling）。使一个程序输出一个错误信息，并停止执行并不是特别复杂的错误处理策略，尤其是在库函数中。

最好能提供一些调用程序可以用来采取正确行为的机制，这样当错误情况出现时，程序作出某些适当的反应。如果不检查这些情况，程序可能给出不正确的答案，或根本不可能产生答案。例如，如果 sqrt 不检查负数参数，那么为了估计一个不存在的结果，程序将进入无限循环。使一个程序报告错误并终止是比使它永远运行更有用的响应。

75.1 Exception Handling

错误的检测和处理并不是 C 语言的强项，而且 C 语言对运行时错误有多种形式表示，并没有提供一种统一的方式。

在 C 语言中，必须由开发者将检测错误的代码编写到程序中，因此很容易忽略一些可能发生的错误。实际上，当这些被忽略的错误中有某个发生时，程序仍然可以继续运行，但是结果未必正确。

C++ 语言对 C 语言的这一弱点进行了改进，提供了一种新的处理错误的方式——异常处理（exception handling）。

75.2 Maths Functions

math.h 中的库函数对错误的处理方式与其他库函数不同。

当发生错误时，math.h 中的大多数函数会将一个错误代码存储到一个名为 errno[1] 的特殊变量中。

此外，如果函数的返回值大于 double 类型的最大取值，math.h 中的函数会返回一个特殊的值，这个值由 HUGE_VAL[2] 宏定义。

HUGE_VAL 是 double 类型，但不一定是一个普通的数[3]。

- 定义域错误：函数的实参超出了函数的定义域。

  当定义域错误发生时，函数的返回值是由实现定义的，同时 EDOM（“定义域错误”）会被存储到 errno 中。

  在一些 math.h 的实现中，当定义域错误发生时，函数会返回值 NaN（“非数”），NaN 是在 IEEE 标准中定义的另一个特殊的值（与“无穷”类似）。

[1] errno 在 errno.h 中定义。
[2] HUGE_VAL 宏在 math.h 中定义。
[3] IEEE 浮点运算标准定义了一个值叫做“无穷”，这个值是 HUGE_VAL 的一个合理选择。

- 取值范围错误:函数的返回值超出了 double 类型的取值范围。
  如果返回值的绝对值过大(溢出),函数会根据结果的符号返回正的或负的 HUGE_VAL。
  此外,值 ERANGE(“取值范围错误”)会被存储到 errno 中,如果返回值的绝对值太小(下溢出),函数返回零,某些实现可能也会将 ERANGE 保存到 errno 中。

第 76 章

Assert

assert 函数在 assert.h 中声明，用于使程序可以监控自己的行为，并提早检测可能会发生的错误。

```
#include <assert.h>

void assert(scalar expression);
```

assert 函数实际上是一个宏，但是它是按照函数的使用方式设计的。

assert 函数只接受一个参数，这个参数必须是一种“断言”——某个我们认为在正常情况下一定为真的表达式。

断言可以用来检测两种问题：

- 如果程序正确执行就不应该发生的问题；
- 超出程序控制范围之外的问题。

每次执行 assert 函数时，都会检查其参数的值。如果参数的值不是 0，assert 函数会输出一条信息到 stderr（标准错误流），并调用 abort 函数来终止程序执行。

例如，假定在 test.c 中声明了一个长度为 N 的数组 a，但是程序中的语句可能会因为 i 不在 0 ～ N-1 而导致程序失败。

```
a[i] = 0;
```

在这种情况下，可以使用 assert 宏在给 a[i] 赋值前进行检查。

```
a[1]:
assert(0 <= i && i < N);
a[i] = 0;
```

如果 i 的值小于 0 或者大于等于 N，那么程序会在输出如下的消息后终止。

```
Assertion failed: 0 <= i && i < N, file TEST.C, line xxx
```

标准 C 并不要求显示的信息和上述格式相同，但是标准 C 要求在显示的信息中以文本格式指明传递给 assert 函数的参数、包含 assert 调用的文件名以及 assert 调用所在的行号。

assert 的不足在于其引入的额外检查增加了程序的运行时间，实时程序对于运行时间的增加可能是无法接受的，因此一般是在测试版本中使用 assert 调用。

为了在正式发布的程序中禁止 assert 调用，可以在包含 assert.h 之前定义宏 NDEBUG。

```
#define NDEBUG
#include <assert.h>
```

记住，NDEBUG 宏的值并不重要，这样当程序中有错误发生时就可以去掉 NDEBUG 宏的定义，并重新执行 assert 函数来对程序进行诊断。

注意，不要在 assert 调用中使用有副作用的表达式，或有副作用的函数调用。当禁用 assert 调用时，这些表达式或函数调用将不再会被计算。例如，在下面的示例中，定义了 NDEBUG 宏之后，assert 调用会被忽略而且 malloc 函数也不再被调用。

```
assert((p = malloc(n+1)) != NULL);
```

第 77 章

Error

77.1 errno

C 语言标准库中的一些函数通过向 errno.h 中声明的 errno 变量存储一个错误代码(正整数)来表示有错误发生。大部分使用 errno 变量的函数集中在 math.h,其他则在标准库的其余部分。

一般情况下,库函数通过给 errno 赋值来产生程序运行出错的信号。如果 errno 不为零则表示在函数调用过程中有错误发生,因此可以通过检查 errno 的值是否为零来判断函数执行是否成功。

例如,如果需要检查 sqrt 函数的调用是否成功,可以使用类似下面的示例代码:

```
errno = 0;
y = sqrt(x);
if(errno != 0){
  fprintf(stderr, "sqrt error, program terminated.\n");
  exit(EXIT_FAILURE);
}
```

C 语言库函数不会负责将 errno 清零,因此对于可能会改变 errno 变量值的函数,在调用前将 errno 置零非常重要。

当错误发生时,向 errno 中存储的值通常是定义在 errno.h 中的宏 EDOM 或 ERANGE,它们分别代表两种在数需函数调用时可能发生的错误。

- 定义域错误(EDOM)
  传递给函数的一个参数不属于函数的定义域。例如,用负数作为 sqrt 函数的参数就会导致一个定义域错误。
- 取值范围错误(ERANGE)
  函数的返回值太大以至于无法用 double 类型的值表示时会发生取值范围错误。例如,用 1000 作为 exp 函数的参数经常会导致一个取值范围错误,因为 e^{1000} 太大以至于在大多数计算机中都无法用 double 类型表示。

当函数调用发生错误时就可能会导致这两种错误,可以使用 errno 分别与 EDOM 和 ERANGE 进行比较来确定错误类型。

在不同的 C 语言实现中,在 errno.h 中往往还定义了其他关于错误条件的宏,这是 C 语言允许的。只要宏的名字以字母 E 开头并在紧随其后使用数字或大写字母组成,都是合法的。

77.2 perror

当库函数向 errno 存储了一个非零值时,为了显示描述这种错误的信息,一种可能的实现方式是调用 perror 函数(定义在 stdio.h 中)来完成。

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>
```

```
const char *sys_errlist[];
int sys_nerr;
int errno;
```

perror 函数会按照如下的顺序来输出错误提示信息。

1. 调用 perror 的参数;
2. 一个分号;
3. 一个空格;
4. 一条出错消息(消息的内容根据 errno 的值来决定);
5. 一个换行符。

perror 函数会输出到 stderr,而不是标准输出。例如,在下面使用 perror 函数的示例中将说明 perror 的使用。

```
errno = 0;
y = sqrt(x);
if(errno != 0){
  perror("sqrt error");
  exit(EXIT_FAILURE);
}
```

perror 函数在 sqrt 函数出错后输出的信息是由实现定义的。例如,下面是一种可能的出错信息。

```
sqrt error: Math argument
```

一般情况下,Math argument 是与 EDOM 错误相对应的信息,而 ERANGE 错误则通常会对应于另一条信息(例如 Result too large)。

77.3 strerror

strerror 函数定义在 string.h 中,当以错误代码调用 strerror 时,函数会返回一个指针,它指向一个描述这种错误的字符串。

```
#include <string.h>

  char *strerror(int errnum);

  /* XSI-compliant */
  int strerror_r(int errnum, char *buf, size_t buflen);

  /* GNU-specific */
  char *strerror_r(int errnum, char *buf, size_t buflen);
```

例如,调用

```
puts(strerror(EDOM));
```

可能会输出

```
Math argument
```

如果给 strerror 函数传递 errno 作为参数,那么函数 perror 输出的错误信息与 strerror 所返回的信息是相同的。

第 78 章

Signal

signal(信号)是 signal.h 提供的处理异常情况的工具。信号有两种类型:运行时错误(例如除 0)和程序以外导致的错误。

许多操作系统都允许用户中断或终止运行的程序,C 语言都将这些事件作为信号,因此当有错误或外部事件发生时,就称产生了一个信号。

大多数信号是异步的,它们可以在程序执行过程中的任意时刻发生,而不仅仅是在开发者所预计的特定时刻发生。

信号可能会在任何时刻发生,因此必须用一种明确的方式来统一处理它们。

signal.h 中定义了一系列的宏来表示不同的信号。

Table 78.1: Signal

宏	含义
SIGABRT	异常终止(可能由于调用 abort 函数导致)
SIGFPE	在数学运算中发生错误(可能是除 0 或溢出)
SIGILL	非法指令
SIGINT	中断
SIGSEGV	非法存储访问
SIGTERM	终止请求

信号的名字可以追溯到早期在 DEC PDP-11 计算机上运行的 C 语言编译器,而且 PDP-11 的硬件可以检测到一些错误,例如“Floating Point Exception”(SIGFPE)和“Segmentation Violation”(SIGSEGV)等。

大多数 C 语言的实现都至少支持上述信号宏其中的一部分,而且 C 语言标准并不要求所有的信号都自动发生。对于特定的硬件平台或操作系统,不是所有的信号都有意义。

78.1 signal

在 signal.h 中最重要的函数就是 signal 函数,它引入的信号处理函数可以用于处理给定的信号。

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

或

```
void (*signal(int sig, void (*func)(int)))(int);
```

在 signal 函数中,第一个参数是特定信号的代码,第二个参数是一个指向函数的指针,该函数就是当信号发生时用来处理信号的函数。

例如,下面的 signal 函数调用对 SIGINT 信号指定了一个处理函数 handler,这样在随后的程序执行过程中出现了 SIGINT 信号后,handler 函数就会自动被执行。

```
signal(SIGINT, handler);
```

每个信号处理函数都必须有一个 int 类型的参数，这样当一个特定的信号出现并调用相应的处理函数时，信号的代码会作为参数传递给处理函数。

知道是哪种信号导致了处理函数被调用是十分有用的，尤其是它允许用户对多个信号使用同一个处理函数。

信号处理函数几乎可以做它所有想做的事情，这其中可能包括忽略该信号、执行一些错误修复动作或者是终止程序等。但是，除非信号是由调用 abort 函数或 raise 函数触发的，否则信号处理函数不应该调用任何库函数，或试图使用一个静态存储期限的变量。

信号处理函数返回时，程序会从信号触发点恢复并继续执行。但是，有一些特殊情况。

- 如果信号是 SIGBRT，当处理函数返回时程序会终止(异常地)。
- 如果信号是 SIGFPE，处理函数返回的结果是未定义的，不能在后面使用。

signal 函数的返回值经常被忽略，但是如果需要，可以将其保存在变量中。特别是，如果需要恢复原来的处理函数，那么就需要保留 signal 函数的返回值。

具体而言，signal 函数的返回值是指向对于指定信号的前一个处理函数的指针。

```
void (*orig_handler) (int); /* function pointer */
orig_handler = signal(SIGINT, handler);
```

上述的调用将 handler 函数设置为 SIGINT 的处理函数，并将指向原始的处理函数的指针保存变量 orig_handler 中。如果要恢复原来的处理函数，需要使用下面的代码：

```
signal(SIGINT, orig_handler); /* restores original handler */
```

除了实现自定义的信号处理函数，还可以选择使用 signal.h 提供的预定义的信号处理函数。

signal.h 提供了两个预定义的信号处理函数，每个都是用宏来表示的。

- SIG_DFL

  函数 SIG_DFL 按“默认”的方式处理信号，可以使用下面的方式来引入 SIG_DFL。

  ```
  signal(SIGINT, SIGDFL); /* use default handler */
  ```

  调用 SIG_DFL 的结果是由实现定义的，不过大多数情况下会导致程序终止。

- SIG_IGN

  可以使用下面的方式来引入 SIG_IGN，并指明随后当信号 SIGINT 发生时，忽略该信号。

  ```
  signal(SIG\_INT, SIG\_IGN); /* ignore SIGINT signal */
  ```

除了 SIG_DFL 和 SIG_IGN，signal.h 可能还会提供其他的信号处理函数，但是都是以 SIG 开头并在随后使用大写字母表示。

当程序开始执行后，根据不同的实现，每个信号的处理函数都会被初始化为 SIG_DFL 或 SIG_IGN。

即使信号处理函数不支持调用库函数，这一例外在 C 语言也是允许的，只要第一个参数是当前正在被处理的信号，信号处理函数就可以合法地调用 signal 函数。

另外，signal.h 还定义了 SIG_ERR 宏来检测引入信号处理函数时是否发生错误。如果一个 signal 调用失败(即无法对指定的信号引入处理函数)，就会返回 SIG_ERR 并在 errno 中置入一个正值。

为了测试 signal 调用是否失败，可以使用下面的代码：

```
if(signal(SIGINT, handler) == SIG_ERR){
  /* error; can not install handler for SIGINT */
}
```

在整个信号处理机制中，如果信号是由处理这个信号的函数触发的，可能会导致无限递归。

为了避免无限递归，C 语言规定除了 SIGILL 以外，当一个信号的处理函数被调用时，该信号对应的处理函数要被重置为 SIG_DEL(默认处理函数)或以其他方式加以锁定。这些操作都是在后台执行的，用户无法控制这一过程。

信号处理完成之后，除非处理函数被重新引入，否则该信号不会被同一函数处理两次。当然，一种实现方法是在处理函数返回前调用 signal 函数。

在 signal.h 中，sig_atomic_t 类型是一个整数类型。按照 C 标准，它可以“作为一个基本元素”使用。换句话说，CPU 可以使用一条机器指令从内存中获取它的值或存储它的值，而不用两条或更多的机器指令。

大多数 CPU 都可以只用一条指令装载或存储一个整数，因此 sig_atomic_t 一般会被定义为 int。

通常，一个信号处理函数不应该访问有静态存储期限的变量，但是 C 语言标准允许一种例外的情况：信号处理函数可以向 sig_atomic_t 类型的变量存入一个值，只要该变量被声明为 volatile（类型限定符）。

考虑一下如果信号处理函数试图修改一个比 sig_atomic_t 类型大的变量的情况，如果程序在信号发生前从内存中获取了这个变量的一部分，然后在信号处理后获取余下的部分，那么程序可能以一个无用的值终止。但是，sig_atomic_t 类型的变量只需要用一条语句获取，而且 olatile 类型的变量每次使用时必须重新获取，因此就杜绝了上述问题的发生。

78.2 raise

通常信号都是自然产生的，但是程序也可以通过其他方式来自行触发信号，raise 函数就可以实现这一目的。

```
int raise(int sig);
```

raise 函数的参数指定了所描述信号的代码，例如可以使用 raise 函数来触发 SIGABRT 信号：

```
raise(SIGABRT); /* raises the SIGABRT signal */
```

raise 函数的返回值可以用来检测调用是否成功，0 代表成功，非 0 则代表失败。

下面的示例程序说明了如果使用信号。

```
/* Tests signals */

#include <signal.h>
#include <stdio.h>

void handler(int sig);
void raise_sig(void);

main()
{
  void (*orig_handler)(int);

  printf("Installing handler for signal %d\n", SIGILL);
  orig_handler = signal(SIGILL, handler);
  raise_sig();

  printf("Changing handler to SIG_IGN\n");
  signal(SIGILL, SIG_IGN);
  raise_sig();

  printf("Restoring original handler\n");
  signal(SIGILL, orig_handler);
  raise_sig();

  printf("Program terminates normally\n");
  return 0;
}
```

```
void handler(int sig)
{
  printf("Handler called for signal %d\n", sig);
}

void raise_sig(void)
{
  raise(SIGILL);
}
```

首先，为 SIGILL 信号引入一个惯用的处理函数(并保存好原先的处理函数)，然后调用 raise_sig 函数产生一个信号。

```
printf("Installing handler for signal %d\n", SIGILL);
orig_handler = signal(SIGILL, handler);
raise_sig();
```

其次，将 SIG_IGN 设置为 SIGILL 的处理函数并再次调用 raise_sig 函数。

```
printf("Changing handler to SIG_IGN\n");
signal(SIGILL, SIG_IGN);
raise_sig();
```

最后，将 SIGILL 信号原先的处理函数重新引入，并再次调用 raise_sig 函数。

```
printf("Restoring original handler\n");
signal(SIGILL, orig_handler);
raise_sig();
```

实际上，调用 raise 函数并不需要在单独的函数中，这里定义 raise_sig 函数只是为了强调它具有如下的特殊之处。

> 无论信号是从哪里发出的(无论是从 main 函数中还是从其他函数中)，raise 函数都会被最近引入的处理函数捕获。

在 C 语言标准中并没有定义所有的信号处理机制，因此不同的实现中结果可能不一致。

C 语言标准允许的另一个例外情况是，如果信号处理函数是由 raise 或 abort 调用的，那么就可以调用标准库函数(例如 printf 函数等)。

```
#include <signal.h>

int raise(int sig);

#include <stdlib.h>

void abort(void);
```

第 79 章

Jump

通常情况下，函数会返回到它被调用的位置，goto 语句也无法将程序流程跳转到其他位置，只能在同一函数内部跳转。

C 语言中通过 setjmp.h 可以使一个函数直接跳转到另一个函数，而不需要返回。

79.1 setjmp

在 setjmp.h 中最重要的内容就是 setjmp 宏和 ongjmp 函数，其中 setjmp 宏“标记”程序中的一个位置，然后就可以使用 longjmp 跳转到该位置。

```
int setjmp(jmp_buf env);
void longjmp(jum_buf env, int val);
```

C 语言的非局部跳转机制有许多潜在的用途，但是主要被用于错误处理。如果要为以后的跳转标记一个位置，可以调用 setjmp 宏，调用的参数是一个 jmp_buf 类型的变量(同样也定义在 setjmp.h 中)。

C 语言标准要求 jmp_buf 必须是一个数组类型，因此传递给 setjmp 的实际上是一个指针，从而可以使 setjmp 修改传递给它的参数。

setjmp 宏会将当前“环境”(包括一个指向 setjmp 宏自身被调用的位置的指针)保存到变量中，以便随后可以在调用 longjmp 函数时使用，然后返回 0。

按照标准 C，只有两种使用 setjmp 宏的方式是合法的。

- 作为表达式语句(可能会强制转换成 oid)。
- 作为 if、switch、while、do 或 for 语句中控制表达式的一部分。

  整个控制表达式必须符合下面的形式之一：

  ```
  setjmp(...)
  !setjmp(...)
  constexp op setjmp(...)
  setjmp(...) op constexp
  ```

  其中，constexp 是一个计算结果为整数的常量表达式，并且 op 是关系运算符或判等运算符。

某些编译器允许不符合这些规则的 setjmp 调用，但是不遵守这些规则的程序将是无法移植的。

79.2 longjmp

可以使用 longjmp 函数来返回到 setjmp 宏所标记的位置，调用的参数是在调用 setjmp 宏时使用的同一个 jmp_buf 类型的变量。

具体来说，要返回 setjmp 标记的位置时，longjmp 会首先根据 jmp_buf 变量的内容恢复当前环境，然后从 setjmp 宏调用中返回。这种情况下，setjmp 宏的返回值是 val(如果 val 的值为 0，那么 setjmp 宏会返回 1)，也就是调用 longjmp 函数时的第二个参数。

需要注意的是，一定要确保作为 longjmp 函数的参数已经被 setjmp 初始化了，否则调用 longjmp 会导致未定义的行为，很可能导致程序崩溃。

总而言之，setjmp 会在第一次调用时返回 0，随后 longjmp 函数将控制权重新转交给最初的 setjmp 宏调用，而 setjmp 在这次调用时会返回一个非零值。

下面的示例程序使用 setjmp 宏在 main 函数中标记一个位置，然后函数 f2 通过调用 longjmp 函数返回到这个位置。

```
/* Tests setjmp/longjmp */

#include <setjmp.h>
#include <stdio.h>

static jmp_buf env;

void f1(void);
void f2(void);

main()
{
  int ret;

  ret = setjmp(env);
  printf("setjmp returned %d\n", ret);
  if (ret != 0) {
    printf("Program terminates: longjmp called\n");
    return 0;
  }
  f1();
  printf("Program terminates normally\n");
  return 0;
}

void f1(void)
{
  printf("f1 begins\n");
  f2();
  printf("f1 returns\n");
}

void f2(void)
{
  printf("f2 begins\n");
  longjmp(env, 1);
  printf("f2 returns\n");
}
```

setjmp 宏的最初调用返回 0，因此 main 函数会调用 f1。接着，f1 调用 f2，f2 使用 longjmp 函数将控制权重新转交给 main 函数，而不是返回 f1.

当 longjmp 函数执行时，控制权重新回到 setjmp 宏调用，然后 setjmp 宏返回 1(也就是在 longjmp 函数调用时所指定的值)。

实际上，大部分变量的值保留了 longjmp 函数被调用时的值，只是 setjmp 宏调用中的自动变量的值是不确定的，除非该变量被声明为 volatile 或者在执行 setjmp 后没有被修改过。

在信号处理函数中调用 longjmp 函数是合法的，只要信号处理函数的调用不是由在信号处理函数执行过程中触发的信号来引起的。

Part XIII

Advanced

第 80 章

Declaration

80.1 Introduction

通过声明变量和函数，可以在检查程序潜在的错误以及把程序翻译成目标代码两方面为编译器提供至关重要的信息。

在 C 语言声明中出现的数据项包括存储类型、类型限定符、声明符以及初始化式等，从而在 C 语言编程中起到核心的作用。

声明为编译器提供有关标识符含义的信息。在大多数通用格式中，声明具有下列格式：

```
declaration-specifier declarator;
```

其中，声明说明符（declaration-specifier）描述声明的数据项的性质，而声明符（declarator）给出了数据项的名字，并且可以提供关于数据项性质的额外信息。

声明说明符可以划分为以下 3 大类：

1. 存储类型
   在声明中最多只能出现一种存储类型（auto、static、extern 或 register），并必须把它放置在声明的首要位置。
2. 类型限定符
   C 语言只提供了两种类型限定符：const 和 volatile。在声明中可以指定一个类型限定符、两者都有或者不指定类型限定符。
3. 类型说明符
   第一，类型说明符包括 void、char、short、int、long、float、double、signed 和 unsigned 等关键字，并且不限定关键字出现的顺序。
   第二，类型说明符也包括结构、联合和枚举的说明以及以 typedef 创建的类型名等。

类型限定符和类型说明符必须跟随在存储类型的后面，但是二者的顺序没有严格的限制，一般是将类型限定符放在类型说明符的前面。

声明符包括标识符（简单变量的名字）、后面跟随 [] 的标识符（数组名）、前置 * 的标识符（指针名）以及后面跟随圆括号的标识符（函数名）等。

声明符之间用逗号分隔，而且表示变量的声明符后面可以跟初始化式。

下面是一个有存储类型、类型说明符和声明符的变量声明。

```
static float x, y, *p;
```

下面是一个有类型限定符、类型说明符和初始化式的字符数组声明。

```
const char month[] = "January";
```

下面是一个有存储类型、类型限定符、类型说明符和声明符的长整型数组声明。

```
extern const unsigned long int a[10];
```

下面是一个有存储类型、类型说明符和声明符的函数声明。

```
extern int square(int);
```

80.2 Storage Type

存储类型可以用于变量、较小范围的函数和形式参数的说明。

C 语言程序中的每个变量都具有 3 个性质:

1. 存储期限

   变量的存储期限决定了为变量分配和释放内存的时间。

   - 具有自动存储期限的变量在所属块被执行时获得内存单元,并在块执行结束时释放内存单元,变量被销毁。
   - 具有静态存储期限的变量在程序运行期间占用同样的内存单元,从而可以允许在程序执行期间都保存变量值。

2. 作用域

   变量的作用域是指引用变量的程序范围,由编译器来决定作用域范围。

   - 在变量的块作用域中,变量从声明的地方一直到闭合块的末尾都是可见的。
   - 在变量的文件作用域中,变量从声明的地方一直到文件的末尾都是可见的。

   编译器用标识符的作用域来确定在文件定义处引入的标识符是否合法。

3. 链接

   变量的链接确定了源代码的不同部分可以共享变量的范围,链接的处理由链接器来决定。

   - 具有外部链接的变量可以被源代码中的若干(或全部)文件共享。
   - 具有内部链接的变量只能属于某一文件,并被该文件中的函数所共享。
   - 无链接的变量属于单独一个函数,而且不能被共享。

   在把源文件翻译成目标代码时,编译器会把具有外部链接的标识符存储到目标文件内的表中,这样可以让链接器访问到具有外部链接的标识符,而具有内部链接或无链接的标识符对链接器不可见。

变量的默认存储期限、作用域和链接都依赖于变量声明的位置。

- 在块内部(包括函数体)声明的变量具有自动存储期限、块作用域,并且无链接。
- 在程序的最外层,任意块外部声明的变量具有静态存储期限、文件作用域和外部链接。

下面的例子说明了变量 i 和变量 j 的默认性质。

```
int i; /* 静态存储期限、文件作用域、外部链接 */

void f(void)
{
  int j; /* 自动存储期限、块作用域、无链接 */
}
```

对许多变量而言,默认的存储期限、作用域和链接都是可以符合要求的。当这些性质无法满足要求时,可以通过指定明确的存储类型来改变变量的性质:auto、static、extern 和 register。

80.2.1 auto

auto 存储类型只对属于块的变量有效。

auto 类型的变量具有自动存储期限、块作用域,并且无链接。

对于在块内部声明的类型,auto 类型是默认的。

80.2.2 static

static 存储类型可以用于全部变量,而无需考虑变量声明所在的位置。但是,块外部声明的变量和块内部声明的变量会有不同的效果。

- 当 static 用于块外部时,说明变量具有内部链接。

- 当 static 用于块内部时，说明变量具有静态存储期限。

```
static int i; /* 静态存储期限、文件作用域、内部链接 */

void f(void)
{
  static int j; /* 静态存储期限、块作用域、无链接 */
}
```

在用于外部块声明时，static 隐藏了它所在声明文件内的变量，这样只有出现在同一文件中的函数可以使用该变量。

在下面的例子中，函数 f1 和函数 f2 都可以访问到变量 i，但是其他文件中的函数则无法访问该变量，从而可以对不同的文件实现信息隐藏。

```
static int i;

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

在用于块内声明的变量时，static 使变量在程序执行期间驻留在同一存储单元内。和每次程序离开闭合块就会销毁值的自动变量不同，static 可以使变量值一直保留到程序结束。

- static 使块内的变量只在程序执行前进行一次初始化，而自动变量则是在程序每次执行到块时才进行初始化。
- 函数进行递归调用时都会获得新的自动变量的集合，但是 static 使递归函数的全部调用都可以共享同一个变量。
- 函数不应该返回指向自动变量的指针，但可以返回指向静态变量的指针。

在函数内部声明静态变量，可以允许函数在“隐藏”区域内的调用之间共享信息。隐藏区域是程序其他部分无法访问到的地方，使用静态变量可以使程序更有效率。

```
char digit_to_hex_char(int digit)
{
  const char hex_chars[16] = "0123456789ABCDEF";
  return hex_chars[digit];
}
```

在函数 digit_to_hex_char 内声明 hex_chars 数组后，每次调用 digit_to_hex_char 函数时，都会把字符 012345678ABCDEF 复制给数组 hex_chars 来对其进行初始化。

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] = "0123456789ABCDEF";
  return hex_chars[digit];
}
```

在函数内部将数组声明为静态变量后，该变量只进行一次初始化，从而可以改善函数的速度。

80.2.3 extern

使用 extern 存储类型可以在源文件之间共享同一个变量。

下面的声明告诉编译器 i 是整型变量，但是 extern 关键字只是提示编译器需要从源代码树中找到变量的定义，可能在同一个文件的后面部分，也可能在另一个文件中。

如果在一个文件中把变量 i 定义在任意函数的外面,默认情况下它具有文件作用域和外部链接。

```
int i;
```

在其他文件中, 如果有函数 f 需要访问变量 i, 那么在 f 的函数体中可以在变量 i 的声明前增加 extern,从而可以使变量 i 具有块作用域和外部链接。如果除函数 f 以外的其他函数需要访问变量 i,那么它们将需要单独进行声明(或者,把变量 i 的声明移动到函数 f 外部,从而使其具有文件作用域)。

```
void f(void)
{
  extern int i;
}
```

变量在程序中的不同声明会建立不同的作用域(块作用域或文件作用域),但是只能有一次定义,编译器不会为外部变量分配存储单元。

按照 C 语言规则,初始化外部变量的声明可以用作变量的定义,因此下面的声明

```
extern int i = 0;
```

等价于声明

```
int i = 0;
```

这样的规则可以防止用不同方法对外部变量进行多次初始化声明。

外部变量始终具有静态存储期限,但是其作用域依赖于变量的声明位置。如果声明在块内部,那么变量具有块作用域,否则具有文件作用域。

```
extern int i; /* 静态存储期限、文件作用域 */

void f(void)
{
  extern int i; /*静态存储期限、块作用域*/
}
```

要确定外部变量的链接有一定难度。如果外部变量在文件中较早的位置(任何函数定义的外部)定义,那么它具有内部链接,否则(通常情况下)具有外部链接。

和变量的声明类似, 函数的声明 (和定义) 也可以包含存储类型, 但是只支持 extern 和 static 关键字。

- 使用 extern 关键字声明函数时,说明函数具有外部链接,也就是允许在其他源文件中调用函数。
- 使用 static 关键字声明函数时,说明函数具有内部链接,也就是说只能在定义函数的源文件内部调用函数。
- 如果未指明函数的存储类型,那么 C 语言会假设函数具有外部链接。

在下面的函数声明中,函数 f 具有外部链接,函数 g 具有内部链接,而函数 h 具有外部链接。

```
extern int f(int i);
static int g(int i);
int h(int i);
```

声明外部函数和声明自动变量一样,二者都没有使用的目的,但是使用 extern 是无害的。

通过 static 关键字可以限制函数的使用范围, 从而可以减少“名字空间”污染, 并使程序更容易维护。

- 首先, 使用 static 关键字声明静态函数, 可以对其他源文件隐藏函数, 从而对函数的修改不会影响到其他文件中的同名函数。
- 其次, 如果另一个源文件中的函数传递的函数指针受到影响, 可以通过检查函数定义所在的文件来快速定义错误。
- 另外, 可以在不同的源文件中定义同名的静态函数而不受影响, 从而可以有效地防范“名在空间”污染。

函数的形式参数具有和自动变量相同的存储期限、块作用域和链接，其中只有寄存器类型可以用于说明形式参数存储类型。

80.2.4 register

声明寄存器变量要求编译器把变量存储在寄存器[1]中。

指明变量是寄存器变量仅是一种要求（而非命令），编译器可以自行决定把寄存器变量存储在内存中还是寄存器中。

register 存储类型只对声明在块内地变量有效，而且寄存器变量和自动变量有一样地存储期限、作用域和链接。

但是，由于寄存器没有地址，寄存器变量也就是没有地址，所以对寄存器变量使用取地址运算符 & 是非法地。

一般地，寄存器变量用于需要频繁进行访问和/或更新的数据。例如，在 for 语句中的循环控制变量可以声明为寄存器变量。

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;
  for(i=0; i<n; i++)
    sum += a[i];
  return sum;
}
```

现代 C 语言编译器可以自动分析和决定将变量保留在寄存器中是否可以获得最大的效率。

综合分析 C 语言存储类型可知，自动类型没有任何效果，寄存器类型被废弃。

80.3 Type Qualifier

C 语言只提供了两种类型限定符：volatile 和 const。

- volatile 类型限定符只用在底层编程中，用于指示编译器哪些数据是“易变”的。
- const 类型限定符可以用来声明类似于变量的“只读”对象，其值可以访问但无法修改。

80.3.1 const

C 语言不允许使用 const 来定义常量表达式，const 关键字只是用于保证在对象的生命周期中保留常量，而在程序的整个执行周期内未必如此。

- 下面的声明产生了“只读”对象 n，且其值为 10。

```
const int n = 10;
```

- 下面的声明产生了“只读”数组 days_per_month。

```
const int days_per_month[] =
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,31};
```

实际上，作为文档格式，const 可以提示用户和编译器，对象的值不会改变。这样当为某种可能的应用类型编写程序时（特别是嵌入式系统），使用 const 关键字可以指示编译器将数据存储到 ROM（只读内存）中。

在 C 语言程序中，#define 和 const 都可以用来声明常量（“只读”对象），但是它们之间有明显的差异。

[1]寄存器是在 CPU 中的存储单元。在传统计算机架构中，存储在寄存器中的数据的访问速度和更新速度要高于存储在内存中的数据。

- 可以用 #define 指令产生数字常量、字符常量或字符串常量的名字，而 const 关键字可以用于产生任何类型的只读对象(包括常量数组、常量指针、常量结构和常量联合等)。
- 使用 const 关键字产生的对象遵循和变量相同的作用域规则，而用 #define 指令定义的常量不遵守这些规则。特别地，不能用 #define 指令产生具有块作用域的常量。
- 和宏的值不同，可以在调试器中显示“只读”对象的值。
- 宏可以用于常量表达式，但是 const 关键字则不可以。
  例如，数组边界必须是常量表达式，所以下面的声明是错误的。

```
const int n = 10;
int a[n]; /* wrong */
```

假设使用 const 关键字声明了变量。

```
void f(int n)
{
  const int m = n;
  ...
}
```

当调用函数 f 时，m 将会被初始化为函数 f 的实际参数的值，然后 m 将在 f 返回之前保留常量。接下来，当再次调用函数 f 时，m 可能会得到不同的值，这就是会出现问题的地方。

接下来，如果使用 m 来指定数组的长度，那么在函数 f 被调用前数组 a 的长度都是未知的，这显然违反了 C 语言要求编译器必须明确每个数组的长度的规定。

```
void f(int n)
{
  const int m = n;
  int a[m]; /* wrong */
}
```

在块外部声明的 const 型对象具有外部链接，并且可以在文件之间对其进行共享。

在下面的示例中，可能在其他文件对变量 n 进行了定义，这使得编译器无法确定数组 a 的长度。

```
extern const int n;
int a[n]; /* wrong */
```

针对这种情况，C 语言建议使用 #define 指令来定义数组长度。

```
#define N 10
int a[N];
```

C 语言中的 const 关键字主要是为了保护存储在数组中的常量数据，而 #define 指令主要用来定义数字或字符常量以及数组边界，从而可以用在 switch 语句或其他要求常量表达式的地方。

C++ 语言通过允许 const 型对象出现在常量表达式中改进了 const 关键字的使用。

- 首先，C++ 允许 const 型对象是整数；
- 第二，C++ 允许 const 型对象的初始化式是常量。

```
const int n = 10;
int a[n]; /* legal in C++, but not in C */
```

默认情况下，C++ 语言还指定 const 型对象具有内部链接，这样就使 const 型对象的定义可以放入头文件中。

80.3.2 volatile

80.4 Declarator

声明符是由标识符(声明的变量或函数的名字)以及可能在前面的符号 * 或者跟随在后面的 [] 或 () 共同组成的。

80.4.1 Array

以 [] 结尾的声明符表示数组。

```
int a[10];
```

如果数组是形式参数，或者数组有初始化式，以及数组的存储类型为 extern，那么中括号内可以为空。

```
extern int a[];
```

这里，数组 a 是在程序中某一个文件定义的，所以当前编译器可以不需要知道数组的长度。在多维数组中，只有第一维中括号可以为空，编译器可以根据元素数量自动计算出第一维的数字。

80.4.2 Pointer

用 * 开头的声明符表示指针。

```
int *p;
```

80.4.3 Function

用 () 结尾的声明符表示函数。

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

C 语言允许在函数声明中忽略形式参数的名字。

```
int abs();
void swap(int *, int *);
int find_largest(int [], int);
```

或者，C 语言也允许使用空括号来声明过程。

```
int abs();
void swap();
int find_largest();
```

这里，使用空括号和声明函数空参数(void)是不同的，其中后者说明函数没有实际参数。

> 来自于经典 C 的使用空括号声明函数的形式正在迅速消失，这种格式比标准 C 的函数原型形式差，因为空括号形式不允许编译器检查函数调用是否有正确的实际参数。

在函数声明中必须说明函数的返回类型，但是对函数的形式参数没有具体要求，而且实际程序中的声明符往往组合了符号 *、[] 和 ()。

例如，下面的语句声明了一个指向函数 pf 的指针，并且指明返回值类型为 void。

```
void (*pf)(int);
```

通过把 *、[] 和 () 组合在一起，可以创建更复杂声明符。

- 数组声明符的格式与数组下标方式相匹配。
- 指针声明符的格式与间接寻址运算符方式相匹配。
- 函数声明符的格式与函数调用的语法相匹配。

考虑下面的数组 file_cmd，该数组的元素都是指向函数的指针。

```
(*file_cmd[])(void)
```

在调用该函数的格式中，圆括号、中括号和 * 都在同样的位置上。

```
(*file_cmd[])();
```

80.4.4 Typedefs

首先,考虑下面的复杂声明。

```
int *(*x[10])(void);
```

C 语言提供了两条规则用来理解复杂声明。

- 始终从内往外解析声明符。
  换句话说,首先定位用来声明的标识符,并且从定位处的声明开始解释。
- 选择优先级始终先是 [] 和 () 后是 *。

在下面的声明中,ap 是指针数组,而 fp 是返回指针的函数。

```
int *ap[10];
float *fp(float);
```

针对示例中的复杂声明,首先定位的标识符是声明的 (x),分析 *x[10] 得到这是一个指针数组。接下来从左侧找到数组元素的指针,也就是指向数组元素指针的指针。最后,在右侧找到指针所指向的数据类型(不带实际参数的函数)以及确定返回值。

在 C 语言有不能声明的特性类型,分别如下:

- 函数不能返回数组。

```
int f(int)[]; /*function can't return array*/
```

- 函数不能返回函数。

```
int g(int)(int); /* function can't return function */
```

- C 语言不支持函数型的数组。

```
int a[10](int); /* c don't support functional array */
```

然而,可以通过使用指针获得相似的效果:函数可以返回指向数组第一个元素的指针,也可以返回指向函数的指针,而且数组的元素可以是指向函数的指针。

在 C 语言中可以利用类型定义简化复杂的声明。继续以下面的复杂声明为例来说明使用类型定义来分解的过程。

```
int *(*x[10])(void);
```

在分解的过程中,首先定义 x 为 F_ptr_array 类型,然后定义 F_ptr_array 为有 F_ptr 值的数组,接下来定义 F_ptr 为指向 F 类型的指针,而且 F 是不带实际参数的函数,F 函数返回指向整型指针。

```
typedef int *F(void);
typedef F *F_ptr;
typedef F_ptr F_ptr_array[10];
F_ptr_array x;
```

80.5 Initializer

C 语言允许在声明变量时对其进行初始化,初始化和赋值不一样,因此不能把声明中的符号 (=) 和赋值运算符相混淆。

- 简单变量的初始化式和变量类型一致。

```
iint i = 5;
```

- 如果类型不匹配,则会用和赋值运算相同的规则对初始化进行类型转换。

```
int j = 5.5;
```

- 指针变量的初始化式必须是具有和变量相同类型或 void* 类型的指针表达式。

```
int *p = &i;
```

- 数组、结构或联合的初始化式通常是一组封闭在大括号内的值。

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

下面是用于控制初始化式的额外规则。

- 具有静态存储期限的变量的初始化式必须是常量。

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

这里 FIRST、LAST 都是宏，编译器可以计算出初始化式的值。如果 FIRST、LAST 是变量，那么初始化式就是非法的。

- 具有自动存储期限的变量的初始化式不需要是常量。

```
int f(int n)
{
  int last = n - 1;
  ...
}
```

- 数组、结构或联合的初始化式必须只能包含常量表达式，不允许有变量或函数调用。

```
#define N 2
int powers[5] = {1, N, N*N, N*N*N, N*N*N*N};
```

这里 N 是常量，所以上述初始化式是合法的。如果 N 是变量，那么程序将无法进行编译。

- 自动类型的结构或联合的初始化式可以是另外一个结构或联合。

```
void g(struct complex c1)
{
  struct complex c2 = c1;
  ...
}
```

初始化式不需要一定是变量或形式参数名。例如，c2 的初始化式可以是 *p，这里的 p 具有 struct complex* 类型或 f(c1) 类型，其中 f 是返回 complex 结构类型的函数。

未初始化式并不总是有未定义的值，变量的初始化值依赖于变量的存储期限。

- 具有自动存储期限的变量没有默认的初始值。不能预测自动变量的初始值，而且每次变量变为有效时可以对值进行改变。
- 具有静态存储期限的变量默认情况下的值为 0。用 calloc 函数分配的内存是简单的给字节的位置 0，而静态变量则不同。

  静态变量的初始化是基于类型的正确初始化，即整数变量初始化为 0，浮点变量初始化为 0.0，而指针则初始化为空指针。

在进行 C 语言程序开发时，最好为静态类型的变量提供初始化式，而不是依赖事实上保证的 0。如果程序访问到没有明确初始化的变量，那么其他人可能不容易确定出是否变量设置为 0，或者很难确定出变量是否在程序中的某处进行了赋值初始化。

第 81 章

Pointer

C 语言引入指针后，可以利用指向变量的指针作为函数的实际参数以达到允许函数修改传递给它的变量的目的。另外，C 语言的库函数可以把指向函数的指针作为实际参数来提高程序效率。

根据指针与数组的关系，可以对数组元素进行指针的算术运算来处理数组。

在程序的运行过程中，内存的分配与释放一般是在编译过程中就确定了，指针的引入使得可以进行动态存储分配，从而使程序在执行期间可以获得需要的内存块。

- 在程序运行时动态分配字符串，可以比通常的固定长度的字符数组方式更灵活。
- 通过指针实现数组的动态存储分配时，可以避免数组空间的浪费。
- 不再需要存储单元时，可以动态地释放已分配的内存块。

动态分配的结构可以链接在一起形成表、树和其他更灵活的数据结构，从而使指针发挥更大的作用。

81.1 Memory Allocation

C 语言的数据结构通常是大小固定的。例如，数组拥有数量固定的元素，而且每个元素具有固定的大小。

数组的大小在声明时就已经确定了，但是也使得在没有修改程序并且再次编译程序的情况下无法改变数据结构的大小，所以说固定大小的数据结构缺乏灵活性。

如果数组的大小是固定的，那么无论声明数组的大小是多少，数组始终有可能被填满。为了解决这一问题，C 语言引入了动态存储分配（dynamic storage allocation）——即在程序运行期间分配内存单元，这样就可以根据实际需要设计可以调整大小的数据结构。

实际上，动态存储分配适用于所有类型的数据，但是更常用于字符串、数组和结构，尤其是动态分配的结构可以被链接成表、树或其他数据结构。

81.2 Dynamic Allocation

为了动态地分配存储空间，需要调用内存分配函数，它们都由 stdlib.h 导出。

- malloc 函数——分配内存块，但是不对内存块进行初始化。
- calloc 函数——分配内存块，并且对内存块进行清除。
- realloc 函数——调整先前分配的内存块。

一般情况下，很难估计数组的实际大小，较方便的做法是在程序运行时再来确定数组的实际大小。C 语言通过动态存储分配解决了这个问题，允许在程序执行期间为数组分配空间，然后通过指向数组的第一个元素的指针访问数组。

malloc 函数和 calloc 函数都可以用于为数组分配内存空间，而且使用动态存储分配产生的数组与普通数组的各项特性都是相同的。其中，相比 malloc 函数，calloc 函数更为常用，因为它会对分配的内存空间进行初始化。

在实现程序逻辑的编码过程中，可以使用 realloc 函数来对数组进行“扩容”或“缩减”。

81.2.1 malloc

使用 malloc 函数为数组分配存储空间的做法和用它为字符串分配空间非常相似。

数组元素的大小需要使用 sizeof 运算符来计算。例如，如果需要使用 malloc 函数为整型数组分配存储空间，需要在声明指针变量后确定数组元素的大小。

```
int *a;
a = malloc(n * sizeof(int));
```

这里，将指针 a 指向分配好的内存块后，就可以忽略 a 是指针的事实，并且把它当作数组的名字。例如，根据数组和指针的关系，可以使用下面的循环对 a 指向的数组进行初始化。

```
for(int i = 0; i < n; i++)
  a[i] = 0;
```

在为数组分配了内存空间后，可以使用指针的算术运算代替下标来访问数组元素。

当计算数组所需要的空间时始终要使用 sizeof 运算符，如果分配的内存空间数量不够会产生严重的后果。例如，使用下面的语句为 n 个整数的数组分配空间时，如果 int 型数据大于两个字节，那么 malloc 函数将无法分配足够大的内存块，这样往数组中存储数据时可能会导致程序崩溃或者行为异常。

```
a = malloc(n * 2);
```

81.2.2 calloc

在为数组分配内存时，C 语言提供的 calloc 函数会更好用。

calloc 函数在 stdlib.h 中的原型如下：

```
void *calloc(size_t nmemb, size_t size);
```

calloc 函数为 nmemb 个元素的数组分配内存空间，其中每个元素的长度都是 size 个字节。如果要求的空间无效，那么 calloc 返回空指针。

```
NAME
     malloc, free, calloc, realloc – allocate and free dynamic memory

SYNOPSIS
     #include <stdlib.h>

     void *malloc(size_t size);
     void free(void *ptr);
     void *calloc(size_t nmemb, size_t size);
     void *realloc(void *ptr, size_t size);

DESCRIPTION
     The malloc() function allocates size bytes and returns a pointer to the allocated
         memory. The memory is not initialized. If size is 0, then malloc() returns either
          NULL, or a unique pointer value that can later be successfully passed to free().

     The free() function frees the memory space pointed to by ptr, which must have been
         returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if
         free(ptr) has already been called before, undefined behavior occurs. If ptr is
         NULL, no operation is performed.

     The calloc() function allocates memory for an array of nmemb elements of size bytes
         each and returns a pointer to the allocated memory. The memory is set to zero. If
          nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value
          that can later be successfully passed to free().
```

```
       The realloc() function changes the size of the memory block pointed to by ptr to size
            bytes. The contents will be unchanged in the range from the start of the region
           up to the minimum of the old and new sizes. If the new size is larger than the
           old size, the added memory will not be initialized. If ptr is NULL, then the call
            is equivalent to malloc(size), for all values of size; if size is equal to zero,
            and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is
           NULL, it must have been returned by an earlier call to malloc(), calloc() or
           realloc(). If the area pointed to was moved, a free(ptr) is done.

RETURN VALUE
       The malloc() and calloc() functions return a pointer to the allocated memory that is
           suitably aligned for any kind of variable. On error, these functions return NULL.
            NULL may also be returned by a successful call to malloc() with a size of zero,
           or by a successful call to calloc() with nmemb or size equal to zero.

       The free() function returns no value.

       The realloc() function returns a pointer to the newly allocated memory, which is
           suitably aligned for any kind of variable and may be different from ptr, or NULL
           if the request fails. If size was equal to 0, either NULL or a pointer suitable
           to be passed to free() is returned. If realloc() fails the original block is left
            untouched; it is not freed or moved.
```

在分配了内存后，calloc 函数会通过对所有位设置 0 的方式进行初始化。例如，下列 calloc 函数的调用为 n 个整数的数组分配存储空间，并且保证全部初始为零。

```
a = calloc(n, sizeof(int));
```

calloc 函数可以用来为非空数组分配空间，并且调用以 1 作为第一个实际参数的 calloc 函数来为任何类型的数据项分配空间。

在下面的示例中，使用 calloc 函数为结构分配内存空间并且将结构的成员 x 和 y 都初始化为 0，并将指针 p 指向结构。

```
struct point {int x, y;} *p;
p = calloc(1, sizeof(struct point));
```

81.2.3 realloc

即使是动态分配的数组，数组的大小在程序运行时可能也需要调整，realloc 函数可以调整数组的大小使它更适合实际情况。

realloc 函数的原型如下：

```
void *realloc(void *ptr, size_t size);
```

当调用 realloc 函数时，指针 ptr 必须指向内存块，且该内存块一定是先前通过调用 malloc 函数，calloc 函数或 realloc 函数来获得的。

size 表示内存块的新尺寸，新尺寸可能会大于或小于原有的尺寸。

realloc 函数不要求 ptr 指向正在用作数组的内存，但是实际上通常是这样的。如果传递给 realloc 函数的指针不是来源于先前 malloc 函数、calloc 函数或 realloc 函数的调用，那么程序可能会行为异常。

在 C 语言标准中明确地包含了有关 realloc 函数行为的规则。

- 当扩展内存块时，realloc 函数不会对添加进内存块的字节进行初始化。
- 如果 realloc 函数不能按要求扩大内存块，那么它会返回空指针，并且在原有的内存块中的数据不会发生改变。
- 如果 realloc 函数调用时以空指针作为第一个实际参数，那么它的行为和 malloc 相同。

- 如果 realloc 函数调用时以 0 作为第二个实际参数，那么它会释放内存块。

在实际使用 ralloc 函数时，客户希望它适当有效。

- 在要求减小内存块大小时，realloc 函数应该“在适当位置”缩减内存块，而不需要移动存储在内存块中的数据。
- realloc 函数应该始终试图扩大内存块而不需要对其进行移动。
- 如果无法扩大内存块，realloc 函数应该在其他位置分配新的内存块，然后把旧块中的内容复制到新块中。

realloc 函数返回时，应该在第一时间内对指向内存块的所有指针进行更新，原因就是 realloc 函数可能移动了其他地方的内存块。

81.3 Free-up Storage

malloc 函数和其他内存分配函数所获得的内存块都来自一个称为堆(heap)的存储池，而且调用这些函数经常会耗尽堆，或者要求大的内存块也可能耗尽堆，这都会导致函数返回空指针。

如果程序通过调用内存分配函数分配了内存块，但是丢失了这些内存块的追踪路径，这样就会造成空间浪费。

```
int *p, *q;
p = malloc(...);
q = malloc(...);
p = q;
```

开始时，指针 p 和 q 指向的位置未知，在调用 malloc 分配了内存块之后，p 和 q 指向不同的内存块，后续的赋值操作使得两个指针都指向指针 q 原来指向的内存块。

对于使用内存分配的内存块，如果指针丢失，那么就再也不能使用该内存块了。对于程序而言，不能再访问的内存块被称为内存垃圾(memory garbage)，产生内存垃圾的程序有内存漏洞(memory leak)。

一些编程语言提供了内存垃圾收集器(garbage collector)来处理内存垃圾的定位和回收，但是 C 语言没有提供。

每个 C 语言都需要负责自己的垃圾回收，C 语言引入了 free 函数来释放不需要的内存。

81.3.1 free

free 函数的原型如下：

```
void free(void *ptr);
```

把指向不再需要内存块的指针传递给 free 函数就可以释放内存块，被释放的内存块会返回给堆，这样使得后续的 malloc 函数或其他内存分配函数的调用可以复用内存池中的存储单元。

```
int *p, *q;
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

free 函数的实际参数必须是指针，而且该指针必须是已经由内存分配函数返回的，调用不带其他实际参数(例如指向变量或数组元素的指针)的 free 函数可能导致不可预测的行为。

81.4 Advanced Pointer

81.4.1 NULL Pointer

81.4.2 Dangling Pointer

81.4.3 Dual Pointer

81.4.4 Function Pointer

81.5 Dynamic String

81.6 Struct Pointer

在大多数情况下,存储结构化数据的变量通常被定义为记录的指针,而不是记录本身。

更重要的是,当把记录的指针传递给某个过程时,过程可以改变记录的内容。如果直接传递记录变量,对应的记录将被复制,从而无法对其内容作出永久性的修改。

在 C 语言中,利用指针传递数组变量可以让函数对数组元素进行修改。同样地,传递记录的指针可以对记录中的字段进行操作,而且指向记录的指针通常比记录小且更容易操作。

在下面的示例中声明了指向 employeeT 记录的指针,首先定义 employeeT 类型。

```
typedef struct{
  string name;
  string title;
  string ssnum;
  double salary;
  int withholding;
}employeeT;
```

接下里,声明一个 employeeT 类型的指针变量。

```
employeeT *emptr;
```

这样就声明了一个指向 employeeT 类型对象的指针 emptr,而且上述声明仅仅为指针提供了空间。在使用指针 emptr 之前,仍然需要为记录中的字段提供完整的空间。

81.6.1 Definition

81.7 Dynamic Array

81.8 Listed List

81.8.1 Declare Node

81.8.2 Create Node

81.8.3 Select Node

81.8.4 Insert Node

81.8.5 Track Node

81.8.6 Delete Node

81.8.7 Sort Node

第 82 章

Blocks

Blocks are a nonstandard extension added by Apple Inc. to their implementations of the C, C++, and Objective-C programming languages that uses a lambda expression-like syntax to create closures within these languages. Blocks are supported for programs developed for Mac OS X 10.6+ and iOS 4.0+,[1] although third-party runtimes allow use on Mac OS X 10.5 and iOS 2.2+.

Apple designed blocks with the explicit goal of making it easier to write programs for the Grand Central Dispatch threading architecture, although it is independent of that architecture and can be used in much the same way as closures in other languages. Apple has implemented blocks both in their own branch of the GNU Compiler Collection and in the Clang LLVM compiler front end. Language runtime library support for blocks is also available as part of the LLVM project. The Khronos group uses blocks syntax to enqueue kernels from within kernels as of version 2.0 of OpenCL.

Like function definitions, blocks can take arguments, and declare their own variables internally. Unlike ordinary C function definitions, their value can capture state from their surrounding context. A block definition produces an opaque value which contains both a reference to the code within the block and a snapshot of the current state of local stack variables at the time of its definition. The block may be later invoked in the same manner as a function pointer. The block may be assigned to variables, passed to functions, and otherwise treated like a normal function pointer, although the application programmer (or the API) must mark the block with a special operator (`Block_copy`) if it's to be used outside the scope in which it was defined.

Given a block value, the code within the block can be executed at any later time by calling it, using the same syntax that would be used for calling a function.

A simple example capturing mutable state in the surrounding scope is an integer range iterator:[7]

Compile the code example with clang compiler commands below:

Blocks bear a superficial resemblance to GCC's extension of C to support lexically scoped nested functions. However, GCC's nested functions, unlike blocks, cannot be called after the containing scope has exited.

GCC-style nested functions also require dynamic creation of executable thunks when taking the address of the nested function. On most architectures (including X86), these thunks are created on the stack, which requires marking the stack executable. Executable stacks are generally considered to be a potential security hole. Blocks do not require the use of executable thunks, so they do not share this weakness.

```
#include <stdio.h>
#include <Block.h>
typedef int (^IntBlock)();

IntBlock MakeCounter(int start, int increment) {
  __block int i = start;

  return Block_copy( ^ {
    int ret = i;
    i += increment;
    return ret;
  });

}

int main(void) {
  IntBlock mycounter = MakeCounter(5, 2);
  printf("First call: %d\n", mycounter());
  printf("Second call: %d\n", mycounter());
  printf("Third call: %d\n", mycounter());

  /* because it was copied, it must also be released */
  Block_release(mycounter);

  return 0;
}
/* Output:
  First call: 5
  Second call: 7
  Third call: 9
*/
```

```
clang -fblocks blocks-test.c -lBlocksRuntime
```

块（blocks）是由 LLVM 提出的类似于 lambda 表达式的非标准 C 语言扩展，亦可以应用于 Objective-C 与 C++ 中。它的语法类似于这些函数中的闭包，即由大括号包括的语句块。

苹果设计块的一个目的是使设计基于 Grand Central Dispatch 线程结构的程序更容易，但块是独立于这一构架的，它也可以在其它程序中以与普通语句块十分相似的方式应用。苹果已经在苹果修改版的 GCC 编译器以及 Clang LLVM 编译器前端中实现了这一特性；同时，LLVM 计划，包括了支持块特性的运行时库。

与函数定义类似，块可以有参数，也可以在其内部声明私有变量。与普通的 C 函数定义不同，块可以使用其上文中定义的变量。一个块定义会产生一个不透明的值，该值同时包括了块内代码的引用和定义时栈内局部变量的快照(而非调用时)。块可以在定义后被调用，其行为与函数指针相同。块可以如同函数指针一般被赋值到变量中，作为函数的参数传递，但若块需要在其被定义的范围之外被使用时，程序员(或 API)需要将该块用特别的运算符(`Block_copy`)标记。

在定义块之后，块内的代码可以在任何时间被调用，语法与调用函数相同。

下面是一个简单的计数器的例子：

```
#include <stdio.h>
#include <Block.h>
typedef int (^IntBlock)();

IntBlock MakeCounter(int start, int increment) {
  __block int i = start;

  return Block_copy( ^ {
    int ret = i;
    i += increment;
    return ret;
  });

}

int main(void) {
  IntBlock mycounter = MakeCounter(5, 2);
  printf("First call: %d\n", mycounter());
  printf("Second call: %d\n", mycounter());
  printf("Third call: %d\n", mycounter());

  /* 由于是复制的块，因此需要释放 */
  Block_release(mycounter);

  return 0;
}
/* Output:
  First call: 5
  Second call: 7
  Third call: 9
*/
```

对该示例使用 clang 进行编译，命令如下：

```
clang -fblocks blocks-test.c -lBlocksRuntime
```

块在外表上与 GCC 的 C 扩展语句块内的嵌套函数相似。然而，嵌套函数与块不同，在退出当前语句块后就不能被调用了。块特性已经被提交到 C 标准委员会，作为 C1x 标准的一系列提案。

第 83 章

Include guard

In the C and C++ programming languages, an #include guard [1], sometimes called a macro guard, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive. The addition of #include guards to a header file is one way to make that file idempotent.

The following C code demonstrates a real problem that can arise if #include guards are missing:

```
/* File "grandfather.h" */
struct foo {
    int member;
};
```

```
/* File "father.h" */
#include "grandfather.h"
```

```
/* File "child.c" */
#include "grandfather.h"
#include "father.h"
```

Here, the file “child.c” has indirectly included two copies of the text in the header file ”grandfather.h”. This causes a compilation error, since the structure type foo is apparently defined twice. In C++, this would be a violation of the One Definition Rule.

There is the use of #include guard below:

```
/* File "grandfather.h" */
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
    int member;
};

#endif /* GRANDFATHER_H */
```

Here, the first inclusion of “grandfather.h” causes the macro `GRANDFATHER_H` to be defined. Then, when “child.c” includes “grandfather.h” the second time, the `#ifndef` test returns false, and the preprocessor skips down to the `#endif`, thus avoiding the second definition of struct foo. The program compiles correctly.

```
/* File "father.h" */
#include "grandfather.h"
```

```
/* File "child.c" */
#include "grandfather.h"
#include "father.h"
```

Different naming conventions for the guard macro may be used by different programmers. Other common forms of the above example include `GRANDFATHER_INCLUDED`, `CREATORSNAME_YYYYMMDD_HHMMSS` (with the appropriate time information substituted), and names generated from a UUID. (However, names starting with one or two underscores, such as `_GRANDFATHER_H` and `__GRANDFATHER_H`, are reserved to the implementation and must not be used by the user.) It is important to avoid duplicating the name in different header files, as including one will prevent the symbols in the other being defined.

In order for #include guards to work properly, each guard must test and conditionally set a different preprocessor macro. Therefore, a project using #include guards must work out a coherent naming scheme for its include guards, and make sure its scheme doesn't conflict with that of any third-party headers it uses, or with the names of any globally visible macros.

For this reason, most C and C++ implementations provide a non-standard #pragma once directive. This directive, inserted at the top of a header file, will ensure that the file is included only once. The Objective-C language (which is a superset of C) introduced an #import directive, which works exactly like #include, except that it includes each file only once, thus obviating the need for #include guards.

在 C 和 C++ 编程语言中，#include 防范[2]，有时被称作宏防范，用于处理 #include 指令时，可避免重复引入的问题。在标头档加入 #include 防范是一种让文件等幂的方法。

以下的 C 语言程式展示了缺少 #include 防范时会出现的问题：

```
/* File "grandfather.h" */
struct foo {
    int member;
};
```

```
/* File "father.h" */
#include "grandfather.h"
```

```
/* File "child.c" */
#include "grandfather.h"
#include "father.h"
```

此处 child.c 间接引入了两份 grandfather.h 标头档中的内容。明显可以看出，foo 结构被定义两次，因此会造成编译错误。

使用 #include 防范的做法如下：

```
/* File "grandfather.h" */
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

struct foo {
    int member;
};

#endif /* GRANDFATHER_H */
```

```
/* File "father.h" */
#include "grandfather.h"
```

```
/* File "child.c" */
#include "grandfather.h"
#include "father.h"
```

此处 grandfather.h 第一次被引入时会定义宏 `H_GRANDFATHER`。当 father.h 再次引入 grandfather.h 时，#ifndef 测试失败，编译器会直接跳到 #endif 的部分，也避免了第二次定义 foo 结构，程序也就能够正常编译。

为了让 #include 防范正确运作，每个防范都必须检验并且有条件地设定不同的前置处理宏。因此，使用了 #include 防范的方案必须制订一致性的命名方法，并确定这个方法不会和其他的标头档或任何可见的全域变量冲突。

为了解决这个问题，许多 C 和 C++ 程式开发工具提供非标准的指令 `#pragma once`。在标头档中加入这个指令，能够保证这个文件只会被引入一次。不过这个方法会被潜在性显著的困难阻挠，无论 #include 指令是否在不同的地方，但实际上起源于相同的开头。同样的，因为 `#pragma once` 不是一个标准的指令，它的语意在不同的程式开发工具中也许会有微妙的不同。

第 84 章

#pragma once

In the C and C++ programming languages, `#pragma once`[4] is a non-standard but widely supported preprocessor directive designed to cause the current source file to be included only once in a single compilation. Thus, `#pragma once` serves the same purpose as #include guards, but with several advantages, including: less code, avoidance of name clashes, and sometimes improved compile speed.

```
File "grandparent.h"

#pragma once

struct foo
{
    int member;
};

File "parent.h"

#include "grandparent.h"

File "child.c"

#include "grandparent.h"
#include "parent.h"
```

The primary advantage is that since the compiler itself is responsible for handling `#pragma once`, it is not necessary for the programmer to create new macro names such as `GRANDPARENT_H` in the Include guard article's example. This eliminates the risk of name clashes, meaning that no header file can fail to be included at least once.

Using `#pragma once` instead of include guards will for some compilers improve compilation speed since it is a higher-level mechanism; the compiler itself can compare filenames or inodes without having to invoke the C preprocessor to scan the header for `#ifndef` and `#endif`.

Common compilers such as GCC, Clang, and EDG-based compilers include specific optimizations to recognize and optimize the handling of include guards, and thus little or no speedup benefit is obtained from the use of `#pragma once`.

在 C 和 C++ 编程语言中，`#pragma once`[3] 是一个非标准但是被广泛支持的前置处理符号，会让所在的文件在一个单独的编译中只被包含一次。以此方式，`#pragma once` 提供类似 include 防范的目的，但是拥有较少的代码且能避免名称的碰撞。

参考 include 防范里其中一种状况的示例或其他的使用方法。如下：

```
file "grandfather.h"
#pragma once

struct foo {
    int member;
};

file "father.h"
#include "grandfather.h"

file "child.c"
#include "grandfather.h"
#include "father.h"
```

使用 `#pragma once` 代替 include 防范将加快编译速度，因为这是一种高级的机制；编译器会自动比对文件名称或 inode 而不需要在头文件去判断 `#ifndef` 和 `#endif`。

另一方面，部分编译器（例如 GCC、clang 等）也包含特别的代码来识别和有效率的管理 include 防范。因此使用 `#pragma once` 并不会得到明显的加速。

此外，因为编译器自己必须承担管理 `#pragma once`，它不必定义新的指令名称，例如在 include 防范文章示例的 `H_GRANDFATHER`。这能排除名称碰撞的风险，意思就是至少第一次包含头文件不会再有错误。

然而，这种高级的管理有好也有坏。设计者必须依赖编译器正确的管理 `#pragma once`。编译器如果犯错，例如没有辨认出在相同文件中的两个不同符号链接名称指针，此时编译会错误。编译器对于 `#pragma once` 可能包含相关的 Bug。2005 年，GCC 文件中将 `#pragma once` 列为“已淘汰”的特性。随着 gcc 3.4 的发布，gcc 解决了 `#pragma once` 中的一些问题（主要是跟符号链接和硬链接有关），并且去掉了 `#pragma once` 的“已淘汰”的标签。

Bibliography

[1] Wikipedia. Include guard, . URL http://en.wikipedia.org/wiki/Include_guard.

[2] Wikipedia. Include 防范, . URL http://zh.wikipedia.org/zh-cn/Include%E9%98%B2%E7%AF%84.

[3] Wikipedia. pragma once, . URL http://zh.wikipedia.org/wiki/Pragma_once.

[4] Wikipedia. pragma once, . URL http://en.wikipedia.org/wiki/Pragma_once.

Part XIV

Library

第 85 章

Overview

每一个函数的名称与特性会被写成一个计算机文件,这个文件就称为头文件(head file),但是实际的函数实现是被分存到函数库文件里。

头文件的命名和领域是很常见的,但是函数库的组织架构也会因为不同的编译器而有所不同。

C 标准函数库(C Standard library)是所有目前符合标准的头文件的集合以及常用的函数库实现程序(例如 I/O 输入输出和字符串控制等)。与 COBOL、Fortran 和 PL/I 等编程语言不同,在 C 语言的工作任务里不会包含嵌入的关键字,所以几乎所有的 C 语言程序都是由标准函数库的函数来创建的,大多数 C 标准函数库在设计上做得相当不错。。

标准函数库通常会随编译器一起提供,例如 GNU C 运行时库通常作为 GCC 的一个部分发布,它最初是 FSF 为其 GNU 操作系统所写,但目前最主要的应用是配合 Linux 内核。

C 语言的标准函数库总共划分为 15 个部分,每个部分用一个头描述。许多 C 编译器常会提供扩展的非 ANSI C 函数功能,这些随附在特定编译器上的标准函数库,对其他不同的编译器来说可能是不兼容的。

实际程序开发中,包含的头通常会多于 15 个,这些额外添加的头不属于标准库的范畴,所以不能假设其他的编译器也可以支持这些头。

扩展的头通常提供一些针对特定的硬件平台或操作系统的函数,因此它们可能提供对屏幕或键盘更多的支持函数。另外,用于支持图形或窗口界面的头也是很常见的。

有些少部分的编译器会为了商业优势和利益而把某些旧函数视同错误或提出警告。例如,字符串输入函数 gets()(以及 scanf() 读取字符串输入的使用上)是很多缓存溢出的原因,而且大多的程序设计指南会建议避免使用它。另一个较为奇特的函数是 strtok(),它原本是作为早期的词法分析用途,但是它非常容易出错(fragile),而且很难使用。

85.1 Header

绝大多数编译器将标准头以文件形式存储,其中包括函数原型、类型定义以及宏定义等。如果在程序中调用了标准头中的函数,或是使用了头中定义的类型或宏,那么就需要在文件开头将相应的头包含进来。

根据 C 语言标准,“标准头”不需要一定是文件,实际上还允许将标准头直接内置在编译器自身中,因此在一个程序中包含了多个头时,#include 指令的顺序无关紧要。

任何包含了标准头的源文件都必须遵守两条原则。

1. 该源文件不能以任何目的再使用在头文件中定义过的宏的名字。
   例如,如果包含了 <stdio.h>,就不能再定义 NULL 了。
2. 具有文件作用域的库名(或类型名)也不可以在源代码层次中重新定义。
   例如,在 <stdio.h> 中已经将 size_t 定义为类型名,那么就不允许在文件作用域内将 size_t 重定义为其他任何标识符。

另外,在 C 语言中还有一些其他的限制。

- 由一个下划线(_)和一个大写字母开头或由两个下划线(__)开头的标识符,属于标准库中保留的标识符,不允许以任何目的使用这种形式的标识符。

- 由一个下划线(_)开头的标识符被保留,用作文件作用域内的标识符和标记。除非仅声明在函数内部,否则不应该使用这类标识符。
- 在标注库中所有外部链接的标识符被保留,用于作为需要外部链接的标识符。特别地,所有标准库函数的名字都被保留。

这些规则并不是强制性的,但是对程序的所有源代码文件都起作用。如果不遵守这些规则可能会降低程序的可移植性。

85.2 Macros

C 语言标准允许在头中定义与库函数同名的宏,从而可以使用宏来替代小的函数。

为了起到保护作用,C 语言标准还要求有实际的函数存在,因此在头文件中声明一个函数并同时定义一个有相同名字的宏的情况并不少见。

在 <ctype.h> 中有大量的同名的函数和宏定义的例子。例如,对于用来检测一个字符是否可以打印的函数 isprint,通常的做法是在 <ctype.h> 中定义 isprint 作为一个函数:

```
int isprint(int c);
```

同时,也把 isprint 定义为一个宏:

```
#defne isprint(c) ((c) >= 0x20 && (c) <= 0x7e)
```

宏名会在预处理阶段被替换,因此在默认情况下,对 isprint 的调用会被作为宏调用。

在大多数情况下,使用宏来替代实际的函数可能会提高程序的运行速度。然而在某些情况下,可能需要的是一个真实的函数,从而可以缩小可执行代码的大小,或者获得一个指向这个函数的指针。

如果确实存在这种需求,可以使用 #undef 指令来删除宏定义,从而可以访问真实的函数。例如,下面的示例代码通过取消 isprint 的宏定义来使用 isprint 函数。

```
#include <ctype.h>
#undef isprint
```

当所提供的宏没有被定义时,#undef 指令不会起任何作用,因此即使 isprint 不是宏,上述示例中这样的做法也也不会带来任何负面影响。

此外,还可以通过给名字加圆括号来屏蔽个别宏调用。

```
(isprint) (c)
```

预处理器无法分辨出带圆括号的宏,除非宏名后跟着一个左圆括号,但是这种做法对编译器无效,它仍可以分离出 isprint 函数。

第 86 章

Introduction

86.1 assert.h

<assert.h> 仅包含 assert 宏，用于检查程序状态。如果检查失败，则程序会被终止。

86.2 ctype.h

<ctype.h> 包含用于字符分类及大小写转换的函数。

86.3 errno.h

<errno.h> 用于输出 errno（“error number”）。

errno 是一个左值（lvalue），可以在调用特定库函数后进行检测，并判断调用过程中是否有错误发生。

86.4 float.h

<float.h> 提供了用于描述浮点类型特性（包括值的范围及精度）的宏，但是没有类型和函数的定义。

```
NAME
     float.h - floating types

SYNOPSIS
     #include <float.h>

DESCRIPTION
     The characteristics of floating types are defined in terms of a model that describes
         a representation of floating-point numbers and values that provide information
         about an
     implementation's floating-point arithmetic.

     The following parameters are used to define the model for each floating-point type:

     s    Sign (±1).

     b    Base or radix of exponent representation (an integer >1).

     e    Exponent (an integer between a minimum e_min and a maximum e_max).

     p    Precision (the number of base-b digits in the significand).
```

```
f_k  Non-negative integers less than b (the significand digits).

A floating-point number x is defined by the following model:

In addition to normalized floating-point numbers (f_1>0 if x!=0), floating types may
    be able to contain other kinds of floating-point numbers, such as subnormal
    floating-point numbers ( x!=0, e= e_min, f_1=0) and unnormalized floating-point
    numbers ( x!=0, e> e_min, f_1=0), and values that are not floating-point numbers,
     such as infinities and NaNs. A NaN is an encoding signifying Not-a-Number. A
    quiet NaN propagates through almost every arithmetic operation without raising a
    floating-point exception; a signaling NaN generally raises a floating-point
    exception when occurring as an arithmetic operand.

The accuracy of the floating-point operations ( '+', '-', '*', '/' ) and of the
    library functions in <math.h> and <complex.h> that return floating-point results
    is implementation-defined. The implementation may state that the accuracy is
    unknown.

All integer values in the <float.h> header, except FLT_ROUNDS, shall be constant
    expressions suitable for use in #if preprocessing directives; all floating values
     shall be constant expressions. All except DECIMAL_DIG, FLT_EVAL_METHOD,
    FLT_RADIX, and FLT_ROUNDS have separate names for all three floating-point types.
     The floating-point model representation is provided for all values except
    FLT_EVAL_METHOD and FLT_ROUNDS.

The rounding mode for floating-point addition is characterized by the implementation-
    defined value of FLT_ROUNDS:

-1   Indeterminable.

 0   Toward zero.

 1   To nearest.

 2   Toward positive infinity.

 3   Toward negative infinity.
All other values for FLT_ROUNDS characterize implementation-defined rounding behavior
    .

The values of operations with floating operands and values subject to the usual
    arithmetic conversions and of floating constants are evaluated to a format whose
    range and precision may be greater than required by the type. The use of
    evaluation formats is characterized by the implementation-defined value of
    FLT_EVAL_METHOD:

-1   Indeterminable.

 0   Evaluate all operations and constants just to the range and precision of the
     type.

 1   Evaluate operations and constants of type float and double to the range and
     precision of the double type; evaluate long double operations and constants to
     the range and precision of the long double type.

 2   Evaluate all operations and constants to the range and precision of the long
     double type.
```

```
All other negative values for FLT_EVAL_METHOD characterize implementation-defined
    behavior.

The values given in the following list shall be defined as constant expressions with
    implementation-defined values that are greater or equal in magnitude (absolute
    value) to those shown, with the same sign.

 * Radix of exponent representation, b.

FLT_RADIX 2

 * Number of base-FLT_RADIX digits in the floating-point significand, p.

FLT_MANT_DIG

DBL_MANT_DIG

LDBL_MANT_DIG

 * Number of decimal digits, n, such that any floating-point number in the widest
     supported floating type with p_max radix b digits can be rounded to a floating-
     point number with n decimal digits and back again without change to the value.

DECIMAL_DIG 10

 * Number of decimal digits, q, such that any floating-point number with q decimal
     digits can be rounded into a floating-point number with p radix b digits and
     back again without change to the q decimal digits.

FLT_DIG 6

DBL_DIG 10

LDBL_DIG 10
 * Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a
     normalized floating-point number, e_min.

FLT_MIN_EXP

DBL_MIN_EXP

LDBL_MIN_EXP

 * Minimum negative integer such that 10 raised to that power is in the range of
     normalized floating-point numbers.

FLT_MIN_10_EXP -37

DBL_MIN_10_EXP -37

LDBL_MIN_10_EXP -37

 * Maximum integer such that FLT_RADIX raised to that power minus 1 is a
     representable finite floating-point number, e_max.

FLT_MAX_EXP
```

```
DBL_MAX_EXP

LDBL_MAX_EXP

 * Maximum integer such that 10 raised to that power is in the range of representable
      finite floating-point numbers.
FLT_MAX_10_EXP +37

DBL_MAX_10_EXP +37

LDBL_MAX_10_EXP +37

The values given in the following list shall be defined as constant expressions with
    implementation-defined values that are greater than or equal to those shown:

 * Maximum representable finite floating-point number.
FLT_MAX 1E+37

DBL_MAX 1E+37

LDBL_MAX 1E+37

The values given in the following list shall be defined as constant expressions with
    implementation-defined (positive) values that are less than or equal to those
    shown:

 * The difference between 1 and the least value greater than 1 that is representable
     in the given floating-point type, b**1-p.

FLT_EPSILON 1E-5

DBL_EPSILON 1E-9

LDBL_EPSILON 1E-9

 * Minimum normalized positive floating-point number, b**e_min.

FLT_MIN 1E-37

DBL_MIN 1E-37

LDBL_MIN 1E-37

The following sections are informative.
```

float.h 提供了两个对所有浮点型都适用的宏——FLT_ROUNDS 和 FLT_RADIX。

- FLT_ROUNDS 说明了浮点加法的舍入模式。
- FLT_RADIX 指定了指数基数的形式，最小值是 2(二进制)。

Table 86.1: float.h FLT_ROUNDS 宏(舍入模式)

取值	含义
-1	不确定
0	趋于零
1	趋于最近有效值
2	趋于正无穷

取值	含义
3	趋于负无穷

float.h 中的其他宏用于描述特定类型的特性，根据宏是针对 float、double 还是 long double 类型，每个宏都会以 FLT、DBL 或 LDBL 开头。

Table 86.2: float.h 有效数字宏

宏	取值	描述
FLT_MANT_DIG		有效数字的个数(基数 FLT_RADIX)
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	≥ 6	有效数字的个数(十进制)
DBL_DIG	≥ 10	
LDBL_DIG	≥ 10	

Table 86.3: float.h 指数宏

宏	取值	描述
FLT_MIN_EXP		FLT_RADIX 能表示的最小(负的次幂)
DBL_MIN_EXP		
LDBL_MIN_EXP		
FLT_MIN_10_EXP	≤ -37	10 能表示的最小(负的次幂)
DBL_MIN_10_EXP	≤ -37	
LDBL_MIN_10_EXP	≤ -37	
FLT_MAX_EXP		FLT_RADIX 能表示的最大次幂
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	$\geq +37$	10 能表示的最大次幂
DBL_MAX_10_EXP	$\geq +37$	
LDBL_MAX_10_EXP	$\geq +37$	

Table 86.4: float.h 最大值、最小值和差值宏

宏	取值	描述
FLT_MAX	$\geq 10^{+37}$	最大的浮点数
DBL_MAX	$\geq 10^{+37}$	
LDBL_MAX	$\geq 10^{+37}$	
FLT_MIN	$\leq 10^{-37}$	最小的规格化浮点数[1]
DBL_MIN	$\leq 10^{-37}$	
LDBL_MIN	$\leq 10^{-37}$	
FLT_EPSILON	$\leq 10^{-5}$	最小的正数 ε[2]，ε 满足：$1.0 + \varepsilon$ 不等于 1.0
DBL_EPSILON	$\leq 10^{-9}$	
LDBL_EPSILON	$\leq 10^{-9}$	

[1] 在 C 语言中，最小的规格化浮点数也就是最接近 0 的值(最小正数)。
[2] 在 C 语言中，ε 代表两个数之间的最小差值。

float.h 中定义的宏一般用于数值分析等应用中。

86.5 limits.h

<limits.h> 提供了用于描述整数类型和字符类型特性(包括它们的最大值和最小值)的宏,但是没有定义类型和函数。

```
NAME
     limits.h - implementation-defined constants

SYNOPSIS
     #include <limits.h>

DESCRIPTION
     Some of the functionality described on this reference page extends the ISO C standard
          . Applications shall define the appropriate feature test macro to enable the
          visibility of these symbols in this header.

     Many of the symbols listed here are not defined by the ISO/IEC 9899:1999 standard.
          Such symbols are not shown as CX shaded.

     The <limits.h> header shall define various symbolic names. Different categories of
          names are described below.

     The names represent various limits on resources that the implementation imposes on
          applications.

     Implementations may choose any appropriate value for each limit, provided it is not
          more restrictive than the Minimum Acceptable Values listed below. Symbolic
          constant names beginning with _POSIX may be found in <unistd.h> .

     Applications should not assume any particular value for a limit. To achieve maximum
          portability, an application should not require more resource than the Minimum
          Acceptable Value quantity. However, an application wishing to avail itself of the
           full amount of a resource available on an implementation may make use of the
          value given in <limits.h> on that particular implementation, by using the
          symbolic names listed below. It should be noted, however, that many of the listed
           limits are not invariant, and at runtime, the value of the limit may differ from
           those given in this header, for the following reasons:

      * The limit is pathname-dependent.

      * The limit differs between the compile and runtime machines.

     For these reasons, an application may use the fpathconf(), pathconf(), and sysconf()
          functions to determine the actual value of a limit at runtime.

     The items in the list ending in _MIN give the most negative values that the
          mathematical types are guaranteed to be capable of representing. Numbers of a
          more negative value may be supported on some implementations, as indicated by the
           <limits.h> header on the implementation, but applications requiring such numbers
           are not guaranteed to be portable to all implementations. For positive constants
           ending in _MIN, this indicates the minimum acceptable value.

  Runtime Invariant Values (Possibly Indeterminate)
     A definition of one of the symbolic names in the following list shall be omitted from
```

```
     <limits.h> on specific implementations where the corresponding value is equal to
     or greater than the stated minimum, but is unspecified.

This indetermination might depend on the amount of available memory space on a
    specific instance of a specific implementation. The actual value supported by a
    specific instance shall be provided by the sysconf() function.

{AIO_LISTIO_MAX}

     Maximum number of I/O operations in a single list I/O call supported by the
         implementation.
     Minimum Acceptable Value: {_POSIX_AIO_LISTIO_MAX}

{AIO_MAX}

     Maximum number of outstanding asynchronous I/O operations supported by the
         implementation.
     Minimum Acceptable Value: {_POSIX_AIO_MAX}

{AIO_PRIO_DELTA_MAX}

     The maximum amount by which a process can decrease its asynchronous I/O priority
          level from its own scheduling priority.
     Minimum Acceptable Value: 0

{ARG_MAX}
     Maximum length of argument to the exec functions including environment data.
     Minimum Acceptable Value: {_POSIX_ARG_MAX}

{ATEXIT_MAX}

     Maximum number of functions that may be registered with atexit().
     Minimum Acceptable Value: 32

{CHILD_MAX}
     Maximum number of simultaneous processes per real user ID.
     Minimum Acceptable Value: {_POSIX_CHILD_MAX}

{DELAYTIMER_MAX}

     Maximum number of timer expiration overruns.
     Minimum Acceptable Value: {_POSIX_DELAYTIMER_MAX}

{HOST_NAME_MAX}
     Maximum length of a host name (not including the terminating null) as returned
         from the gethostname() function.
     Minimum Acceptable Value: {_POSIX_HOST_NAME_MAX}

{IOV_MAX}

     Maximum number of iovec structures that one process has available for use with
         readv() or writev().
     Minimum Acceptable Value: {_XOPEN_IOV_MAX}

{LOGIN_NAME_MAX}
     Maximum length of a login name.
     Minimum Acceptable Value: {_POSIX_LOGIN_NAME_MAX}
```

```
{MQ_OPEN_MAX}

     The maximum number of open message queue descriptors a process may hold.
     Minimum Acceptable Value: {_POSIX_MQ_OPEN_MAX}

{MQ_PRIO_MAX}

     The maximum number of message priorities supported by the implementation.
     Minimum Acceptable Value: {_POSIX_MQ_PRIO_MAX}

{OPEN_MAX}
     Maximum number of files that one process can have open at any one time.
     Minimum Acceptable Value: {_POSIX_OPEN_MAX}

{PAGESIZE}
     Size in bytes of a page.
     Minimum Acceptable Value: 1

{PAGE_SIZE}

     Equivalent to {PAGESIZE}. If either {PAGESIZE} or {PAGE_SIZE} is defined, the
         other is defined with the same value.

{PTHREAD_DESTRUCTOR_ITERATIONS}

     Maximum number of attempts made to destroy a thread's thread-specific data
         values on thread exit.
     Minimum Acceptable Value: {_POSIX_THREAD_DESTRUCTOR_ITERATIONS}

{PTHREAD_KEYS_MAX}

     Maximum number of data keys that can be created by a process.
     Minimum Acceptable Value: {_POSIX_THREAD_KEYS_MAX}

{PTHREAD_STACK_MIN}

     Minimum size in bytes of thread stack storage.
     Minimum Acceptable Value: 0

{PTHREAD_THREADS_MAX}

     Maximum number of threads that can be created per process.
     Minimum Acceptable Value: {_POSIX_THREAD_THREADS_MAX}

{RE_DUP_MAX}
     The number of repeated occurrences of a BRE permitted by the regexec() and
         regcomp() functions when using the interval notation {\(m,n\};
     Minimum Acceptable Value: {_POSIX2_RE_DUP_MAX}

{RTSIG_MAX}

     Maximum number of realtime signals reserved for application use in this
         implementation.
     Minimum Acceptable Value: {_POSIX_RTSIG_MAX}

{SEM_NSEMS_MAX}

     Maximum number of semaphores that a process may have.
```

Minimum Acceptable Value: {_POSIX_SEM_NSEMS_MAX}

{SEM_VALUE_MAX}

The maximum value a semaphore may have.
Minimum Acceptable Value: {_POSIX_SEM_VALUE_MAX}

{SIGQUEUE_MAX}

Maximum number of queued signals that a process may send and have pending at the receiver(s) at any time.
Minimum Acceptable Value: {_POSIX_SIGQUEUE_MAX}

{SS_REPL_MAX}

The maximum number of replenishment operations that may be simultaneously pending for a particular sporadic server scheduler.
Minimum Acceptable Value: {_POSIX_SS_REPL_MAX}

{STREAM_MAX}
The number of streams that one process can have open at one time. If defined, it has the same value as {FOPEN_MAX} (see <stdio.h>).
Minimum Acceptable Value: {_POSIX_STREAM_MAX}

{SYMLOOP_MAX}
Maximum number of symbolic links that can be reliably traversed in the resolution of a pathname in the absence of a loop.
Minimum Acceptable Value: {_POSIX_SYMLOOP_MAX}

{TIMER_MAX}

Maximum number of timers per process supported by the implementation.
Minimum Acceptable Value: {_POSIX_TIMER_MAX}

{TRACE_EVENT_NAME_MAX}

Maximum length of the trace event name.
Minimum Acceptable Value: {_POSIX_TRACE_EVENT_NAME_MAX}

{TRACE_NAME_MAX}

Maximum length of the trace generation version string or of the trace stream name.
Minimum Acceptable Value: {_POSIX_TRACE_NAME_MAX}

{TRACE_SYS_MAX}

Maximum number of trace streams that may simultaneously exist in the system.
Minimum Acceptable Value: {_POSIX_TRACE_SYS_MAX}

{TRACE_USER_EVENT_MAX}

Maximum number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event POSIX_TRACE_UNNAMED_USER_EVENT.
Minimum Acceptable Value: {_POSIX_TRACE_USER_EVENT_MAX}

{TTY_NAME_MAX}

```
        Maximum length of terminal device name.
        Minimum Acceptable Value: {_POSIX_TTY_NAME_MAX}

    {TZNAME_MAX}
        Maximum number of bytes supported for the name of a timezone (not of the TZ
            variable).
        Minimum Acceptable Value: {_POSIX_TZNAME_MAX}

    Note: The length given by {TZNAME_MAX} does not include the quoting characters
        mentioned in Other Environment Variables .

Pathname Variable Values
    The values in the following list may be constants within an implementation or may
        vary from one pathname to another. For example, file systems or directories may
        have different characteristics.

    A definition of one of the values shall be omitted from the <limits.h> header on
        specific implementations where the corresponding value is equal to or greater
        than the stated minimum, but where the value can vary depending on the file to
        which it is applied. The actual value supported for a specific pathname shall be
        provided by the pathconf() function.

    {FILESIZEBITS}
        Minimum number of bits needed to represent, as a signed integer value, the
            maximum size of a regular file allowed in the specified directory.
        Minimum Acceptable Value: 32

    {LINK_MAX}
        Maximum number of links to a single file.
        Minimum Acceptable Value: {_POSIX_LINK_MAX}

    {MAX_CANON}
        Maximum number of bytes in a terminal canonical input line.
        Minimum Acceptable Value: {_POSIX_MAX_CANON}

    {MAX_INPUT}
        Minimum number of bytes for which space is available in a terminal input queue;
            therefore, the maximum number of bytes a conforming application may require
            to be typed as input before reading them.
        Minimum Acceptable Value: {_POSIX_MAX_INPUT}

    {NAME_MAX}
        Maximum number of bytes in a filename (not including terminating null).
        Minimum Acceptable Value: {_POSIX_NAME_MAX}
        Minimum Acceptable Value: {_XOPEN_NAME_MAX}

    {PATH_MAX}
        Maximum number of bytes in a pathname, including the terminating null character.
        Minimum Acceptable Value: {_POSIX_PATH_MAX}
        Minimum Acceptable Value: {_XOPEN_PATH_MAX}

    {PIPE_BUF}
        Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
        Minimum Acceptable Value: {_POSIX_PIPE_BUF}

    {POSIX_ALLOC_SIZE_MIN}

        Minimum number of bytes of storage actually allocated for any portion of a file.
```

```
        Minimum Acceptable Value: Not specified.

    {POSIX_REC_INCR_XFER_SIZE}

        Recommended increment for file transfer sizes between the {
            POSIX_REC_MIN_XFER_SIZE} and {POSIX_REC_MAX_XFER_SIZE} values.
        Minimum Acceptable Value: Not specified.

    {POSIX_REC_MAX_XFER_SIZE}

        Maximum recommended file transfer size.
        Minimum Acceptable Value: Not specified.

    {POSIX_REC_MIN_XFER_SIZE}

        Minimum recommended file transfer size.
        Minimum Acceptable Value: Not specified.

    {POSIX_REC_XFER_ALIGN}

        Recommended file transfer buffer alignment.
        Minimum Acceptable Value: Not specified.

    {SYMLINK_MAX}
        Maximum number of bytes in a symbolic link.
        Minimum Acceptable Value: {_POSIX_SYMLINK_MAX}


Runtime Increasable Values
    The magnitude limitations in the following list shall be fixed by specific
        implementations. An application should assume that the value supplied by <limits.
        h> in a specific implementation is the minimum that pertains whenever the
        application is run under that implementation. A specific instance of a specific
        implementation may increase the value relative to that supplied by <limits.h> for
         that implementation. The actual value supported by a specific instance shall be
        provided by the sysconf() function.

    {BC_BASE_MAX}
        Maximum obase values allowed by the bc utility.
        Minimum Acceptable Value: {_POSIX2_BC_BASE_MAX}

    {BC_DIM_MAX}
        Maximum number of elements permitted in an array by the bc utility.
        Minimum Acceptable Value: {_POSIX2_BC_DIM_MAX}

    {BC_SCALE_MAX}
        Maximum scale value allowed by the bc utility.
        Minimum Acceptable Value: {_POSIX2_BC_SCALE_MAX}

    {BC_STRING_MAX}
        Maximum length of a string constant accepted by the bc utility.
        Minimum Acceptable Value: {_POSIX2_BC_STRING_MAX}

    {CHARCLASS_NAME_MAX}
        Maximum number of bytes in a character class name.
        Minimum Acceptable Value: {_POSIX2_CHARCLASS_NAME_MAX}

    {COLL_WEIGHTS_MAX}
```

```
        Maximum number of weights that can be assigned to an entry of the LC_COLLATE
            order keyword in the locale definition file; see Locale .
        Minimum Acceptable Value: {_POSIX2_COLL_WEIGHTS_MAX}

    {EXPR_NEST_MAX}
        Maximum number of expressions that can be nested within parentheses by the expr
            utility.
        Minimum Acceptable Value: {_POSIX2_EXPR_NEST_MAX}

    {LINE_MAX}
        Unless otherwise noted, the maximum length, in bytes, of a utility's input line
            (either standard input or another file), when the utility is described as
            processing text files. The length includes room for the trailing <newline>.
        Minimum Acceptable Value: {_POSIX2_LINE_MAX}

    {NGROUPS_MAX}
        Maximum number of simultaneous supplementary group IDs per process.
        Minimum Acceptable Value: {_POSIX_NGROUPS_MAX}

    {RE_DUP_MAX}
        Maximum number of repeated occurrences of a regular expression permitted when
            using the interval notation \{m,n\};
        Minimum Acceptable Value: {_POSIX2_RE_DUP_MAX}


Maximum Values
    The symbolic constants in the following list shall be defined in <limits.h> with the
        values shown. These are symbolic names for the most restrictive value for certain
         features on an implementation supporting the Timers option. A conforming
        implementation shall provide values no larger than these values. A conforming
        application must not require a smaller value for correct operation.

    {_POSIX_CLOCKRES_MIN}

         The resolution of the CLOCK_REALTIME clock, in nanoseconds.
         Value: 20 000 000

    If the Monotonic Clock option is supported, the resolution of the CLOCK_MONOTONIC
        clock, in nanoseconds, is represented by {_POSIX_CLOCKRES_MIN}.


Minimum Values
    The symbolic constants in the following list shall be defined in <limits.h> with the
        values shown. These are symbolic names for the most restrictive value for certain
         features on an implementation conforming to this volume of IEEE Std 1003.1-2001.
         Related symbolic constants are defined elsewhere in this volume of IEEE Std
        1003.1-2001 which reflect the actual implementation and which need not be as
        restrictive. A conforming implementation shall provide values at least this large
        . A strictly conforming application must not require a larger value for correct
        operation.

    {_POSIX_AIO_LISTIO_MAX}

         The number of I/O operations that can be specified in a list I/O call.
         Value: 2

    {_POSIX_AIO_MAX}
```

```
     The number of outstanding asynchronous I/O operations.
     Value: 1

{_POSIX_ARG_MAX}
     Maximum length of argument to the exec functions including environment data.
     Value: 4 096

{_POSIX_CHILD_MAX}
     Maximum number of simultaneous processes per real user ID.
     Value: 25

{_POSIX_DELAYTIMER_MAX}

     The number of timer expiration overruns.
     Value: 32

{_POSIX_HOST_NAME_MAX}
     Maximum length of a host name (not including the terminating null) as returned
         from the gethostname() function.
     Value: 255

{_POSIX_LINK_MAX}
     Maximum number of links to a single file.
     Value: 8

{_POSIX_LOGIN_NAME_MAX}
     The size of the storage required for a login name, in bytes, including the
         terminating null.
     Value: 9

{_POSIX_MAX_CANON}
     Maximum number of bytes in a terminal canonical input queue.
     Value: 255

{_POSIX_MAX_INPUT}
     Maximum number of bytes allowed in a terminal input queue.
     Value: 255

{_POSIX_MQ_OPEN_MAX}

     The number of message queues that can be open for a single process.
     Value: 8

{_POSIX_MQ_PRIO_MAX}

     The maximum number of message priorities supported by the implementation.
     Value: 32

{_POSIX_NAME_MAX}
     Maximum number of bytes in a filename (not including terminating null).
     Value: 14

{_POSIX_NGROUPS_MAX}
     Maximum number of simultaneous supplementary group IDs per process.
     Value: 8

{_POSIX_OPEN_MAX}
     Maximum number of files that one process can have open at any one time.
```

```
     Value: 20

{_POSIX_PATH_MAX}
     Maximum number of bytes in a pathname.
     Value: 256

{_POSIX_PIPE_BUF}
     Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
     Value: 512

{_POSIX_RE_DUP_MAX}
     The number of repeated occurrences of a BRE permitted by the regexec() and
         regcomp() functions when using the interval notation {\(m,n\};
     Value: 255

{_POSIX_RTSIG_MAX}

     The number of realtime signal numbers reserved for application use.
     Value: 8

{_POSIX_SEM_NSEMS_MAX}

     The number of semaphores that a process may have.
     Value: 256

{_POSIX_SEM_VALUE_MAX}

     The maximum value a semaphore may have.
     Value: 32 767

{_POSIX_SIGQUEUE_MAX}

     The number of queued signals that a process may send and have pending at the
         receiver(s) at any time.
     Value: 32

{_POSIX_SSIZE_MAX}
     The value that can be stored in an object of type ssize_t.
     Value: 32 767

{_POSIX_STREAM_MAX}
     The number of streams that one process can have open at one time.
     Value: 8

{_POSIX_SS_REPL_MAX}

     The number of replenishment operations that may be simultaneously pending for a
         particular sporadic server scheduler.
     Value: 4

{_POSIX_SYMLINK_MAX}
     The number of bytes in a symbolic link.
     Value: 255

{_POSIX_SYMLOOP_MAX}
     The number of symbolic links that can be traversed in the resolution of a
         pathname in the absence of a loop.
     Value: 8
```

{_POSIX_THREAD_DESTRUCTOR_ITERATIONS}

The number of attempts made to destroy a thread's thread-specific data values on thread exit.
Value: 4

{_POSIX_THREAD_KEYS_MAX}

The number of data keys per process.
Value: 128

{_POSIX_THREAD_THREADS_MAX}

The number of threads per process.
Value: 64

{_POSIX_TIMER_MAX}

The per-process number of timers.
Value: 32

{_POSIX_TRACE_EVENT_NAME_MAX}

The length in bytes of a trace event name.
Value: 30

{_POSIX_TRACE_NAME_MAX}

The length in bytes of a trace generation version string or a trace stream name.
Value: 8

{_POSIX_TRACE_SYS_MAX}

The number of trace streams that may simultaneously exist in the system.
Value: 8

{_POSIX_TRACE_USER_EVENT_MAX}

The number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event POSIX_TRACE_UNNAMED_USER_EVENT.
Value: 32

{_POSIX_TTY_NAME_MAX}
The size of the storage required for a terminal device name, in bytes, including the terminating null.
Value: 9

{_POSIX_TZNAME_MAX}
Maximum number of bytes supported for the name of a timezone (not of the TZ variable).
Value: 6

Note:
The length given by {_POSIX_TZNAME_MAX} does not include the quoting characters mentioned in Other Environment Variables .

```
{_POSIX2_BC_BASE_MAX}
     Maximum obase values allowed by the bc utility.
     Value: 99

{_POSIX2_BC_DIM_MAX}
     Maximum number of elements permitted in an array by the bc utility.
     Value: 2 048

{_POSIX2_BC_SCALE_MAX}
     Maximum scale value allowed by the bc utility.
     Value: 99

{_POSIX2_BC_STRING_MAX}
     Maximum length of a string constant accepted by the bc utility.
     Value: 1 000

{_POSIX2_CHARCLASS_NAME_MAX}
     Maximum number of bytes in a character class name.
     Value: 14

{_POSIX2_COLL_WEIGHTS_MAX}
     Maximum number of weights that can be assigned to an entry of the LC_COLLATE
         order keyword in the locale definition file; see Locale .
     Value: 2

{_POSIX2_EXPR_NEST_MAX}
     Maximum number of expressions that can be nested within parentheses by the expr
         utility.
     Value: 32

{_POSIX2_LINE_MAX}
     Unless otherwise noted, the maximum length, in bytes, of a utility's input line
         (either standard input or another file), when the utility is described as
         processing text files. The length includes room for the trailing <newline>.
     Value: 2 048

{_POSIX2_RE_DUP_MAX
     Maximum number of repeated occurrences of a regular expression permitted when
         using the interval notation \{m,n\}; see Regular Expressions .
     Value: 255

{_XOPEN_IOV_MAX}

     Maximum number of iovec structures that one process has available for use with
         readv() or writev().
     Value: 16

{_XOPEN_NAME_MAX}

     Maximum number of bytes in a filename (not including the terminating null).
     Value: 255

{_XOPEN_PATH_MAX}

     Maximum number of bytes in a pathname.
     Value: 1024
```

```
Numerical Limits
   The values in the following lists shall be defined in <limits.h> and are constant
       expressions suitable for use in #if preprocessing directives. Moreover, except
       for {CHAR_BIT}, {DBL_DIG}, {DBL_MAX}, {FLT_DIG}, {FLT_MAX}, {LONG_BIT}, {WORD_BIT
       }, and {MB_LEN_MAX}, the symbolic names are defined as expressions of the correct
        type.

   If the value of an object of type char is treated as a signed integer when used in an
        expression, the value of {CHAR_MIN} is the same as that of {SCHAR_MIN} and the
       value of {CHAR_MAX} is the same as that of {SCHAR_MAX}. Otherwise, the value of {
       CHAR_MIN} is 0 and the value of {CHAR_MAX} is the same as that of {UCHAR_MAX}.

   {CHAR_BIT}
        Number of bits in a type char.
        Value: 8

   {CHAR_MAX}
        Maximum value of type char.
        Value: {UCHAR_MAX} or {SCHAR_MAX}

   {CHAR_MIN}
        Minimum value of type char.
        Value: {SCHAR_MIN} or 0

   {INT_MAX}
        Maximum value of an int.
        Minimum Acceptable Value: 2 147 483 647

   {LONG_BIT}

        Number of bits in a long.
        Minimum Acceptable Value: 32

   {LONG_MAX}
        Maximum value of a long.
        Minimum Acceptable Value: +2 147 483 647

   {MB_LEN_MAX}
        Maximum number of bytes in a character, for any supported locale.
        Minimum Acceptable Value: 1

   {SCHAR_MAX}
        Maximum value of type signed char.
        Value: +127

   {SHRT_MAX}
        Maximum value of type short.
        Minimum Acceptable Value: +32 767

   {SSIZE_MAX}
        Maximum value of an object of type ssize_t.
        Minimum Acceptable Value: {_POSIX_SSIZE_MAX}

   {UCHAR_MAX}
        Maximum value of type unsigned char.
        Value: 255
```

```
    {UINT_MAX}
         Maximum value of type unsigned.
         Minimum Acceptable Value: 4 294 967 295

    {ULONG_MAX}
         Maximum value of type unsigned long.
         Minimum Acceptable Value: 4 294 967 295

    {USHRT_MAX}
         Maximum value for a type unsigned short.
         Minimum Acceptable Value: 65 535

    {WORD_BIT}

         Number of bits in a word or type int.
         Minimum Acceptable Value: 16

    {INT_MIN}
         Minimum value of type int.
         Maximum Acceptable Value: -2 147 483 647

    {LONG_MIN}
         Minimum value of type long.
         Maximum Acceptable Value: -2 147 483 647

    {SCHAR_MIN}
         Minimum value of type signed char.
         Value: -128

    {SHRT_MIN}
         Minimum value of type short.
         Maximum Acceptable Value: -32 767

    {LLONG_MIN}
         Minimum value of type long long.
         Maximum Acceptable Value: -9223372036854775807

    {LLONG_MAX}
         Maximum value of type long long.
         Minimum Acceptable Value: +9223372036854775807

    {ULLONG_MAX}
         Maximum value of type unsigned long long.
         Minimum Acceptable Value: 18446744073709551615

  Other Invariant Values
    The following constants shall be defined on all implementations in <limits.h>:

    {CHARCLASS_NAME_MAX}

         Maximum number of bytes in a character class name.
         Minimum Acceptable Value: 14

    {NL_ARGMAX}

         Maximum value of digit in calls to the printf() and scanf() functions.
         Minimum Acceptable Value: 9
```

```
       {NL_LANGMAX}

            Maximum number of bytes in a LANG name.
            Minimum Acceptable Value: 14

       {NL_MSGMAX}

            Maximum message number.
            Minimum Acceptable Value: 32 767

       {NL_NMAX}

            Maximum number of bytes in an N-to-1 collation mapping.
            Minimum Acceptable Value: No guaranteed value across all conforming
                implementations.

       {NL_SETMAX}

            Maximum set number.
            Minimum Acceptable Value: 255

       {NL_TEXTMAX}

            Maximum number of bytes in a message string.
            Minimum Acceptable Value: {_POSIX2_LINE_MAX}

       {NZERO}

            Default process priority.
            Minimum Acceptable Value: 20

       The following sections are informative.

  APPLICATION USAGE
       None.

  RATIONALE
       A request was made to reduce the value of {_POSIX_LINK_MAX} from the value of 8
           specified for it in the POSIX.1-1990 standard to 2. The standard developers
           decided to deny this request for several reasons:

        * They wanted to avoid making any changes to the standard that could break
            conforming applications, and the requested change could have that effect.

        * The use of multiple hard links to a file cannot always be replaced with use of
            symbolic links. Symbolic links are semantically different from hard links in
            that they associate a pathname with another pathname rather than a pathname with
             a file. This has implications for access control, file permanence, and
            transparency.

        * The original standard developers had considered the issue of allowing for
            implementations that did not in general support hard links, and decided that
            this would reduce consensus on the standard.

       Systems that support historical versions of the development option of the ISO POSIX-2
            standard retain the name {_POSIX2_RE_DUP_MAX} as an alias for {_POSIX_RE_DUP_MAX
           }.
```

```
{PATH_MAX}
     IEEE PASC Interpretation 1003.1 #15 addressed the inconsistency in the standard
         with the definition of pathname and the description of {PATH_MAX}, allowing
         application writers to allocate either {PATH_MAX} or {PATH_MAX}+1 bytes. The
          inconsistency has been removed by correction to the {PATH_MAX} definition
         to include the null character. With this change, applications that
         previously allocated {PATH_MAX} bytes will continue to succeed.

{SYMLINK_MAX}
     This symbol refers to space for data that is stored in the file system, as
         opposed to {PATH_MAX} which is the length of a name that can be passed to a
         function. In some existing implementations, the filenames pointed to by
         symbolic links are stored in the inodes of the links, so it is important
         that {SYMLINK_MAX} not be constrained to be as large as {PATH_MAX}.
```

下面是 limits.h 中用于描述字符类型(char、signed char 和 unsigned char)的宏以及取值范围中的最大值或最小值。

Table 86.5: limits.h 字符型宏

宏	取值	描述
CHAR_BIT	≥ 8	每个字符包含位的个数
SCHAR_MIN	≤ -127	最小带符号字符
SCHAR_MAX	$\geq +127$	最大带符号字符
UCHAR_MAX	≥ 255	最大无符号字符
CHAR_MIN	[3]	最小字符
CHAR_MAX	[4]	最大字符
MB_LEN_MAX	≥ 1	多字节字符最多包含的字节数

其他在 limits.h 中定义的宏针对整数类型(short int、unsigned short int、int、unsigned int、long int 以及 unsigned long int)并定义了它们的最大值或最小值。

Table 86.6: float.h 整型宏

宏	取值	描述
SHRT_MIN	≤ -32767	最小短整型数
SHRT_MAX	$\geq +32767$	最大短整型数
USHRT_MAX	≥ 65535	最大无符号短整型数
INT_MIN	≤ -32767	最小整型数
INT_MAX	$\geq +32767$	最大整型数
UINT_MAX	≥ 65535	最大无符号整型数
LONG_MIN	≤ -2147483674	最小长整型数
LONG_MAX	$\geq +2147483674$	最大长整型数
ULONG_MAX	≥ 4292967295	最大无符号长整型数

limits.h 中定义的宏在查看编译器是否支持特定大小的整数时十分方便。例如,如果要判断 int 类型是否可以用来存储大数 100 000,可以使用下面的预处理命令。

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

[3]如果 char 类型被当作 signed char 类型,CHAR_MIN 与 SCHAR_MIN 相等,否则 CHAR_MIN 为 0。
[4]如果 char 类型被当作 signed char 或 unsigned char,CHAR_MAX 分别与 SCHAR_MAX 或 UCHAR_MAX 相等。

这样,如果 int 类型不适用,#error 指令会中止编译。

更进一步,可以使用 limits.h 中的宏来帮助程序选择正确的类型定义。

假设 Quantity 类型的变量必须存储 100000 以上的大数,那么如果 INT_MAX 大于 100000,就可以将 Quantity 定义为 int,否则需要定义为 long int。

```
#if INT_MAX >= 100000
typedef int Quantity
#else
typedef long int Quantity
#endif
```

86.6 locale.h

<locale.h> 提供了帮助程序适配某个国家或地区。这些与本地化相关的行为包括数显示的方式(包括用于小数点的字符)、货币的格式(包括货币符号)、字符集以及日期和时间的表示形式等。

86.7 math.h

<math.h> 提供了大量用于数学计算的函数,包括三角函数、双曲函数、指数函数、对数函数、幂函数、就近取整函数、绝对值运算函数和取余函数等。

<math.h> 中的大部分函数使用 double 类型的实际参数,并返回一个 double 类型的值。

```
NAME
     math.h - mathematical declarations

SYNOPSIS
     #include <math.h>

DESCRIPTION
     Some of the functionality described on this reference page extends the ISO C standard
         . Applications shall define the appropriate feature test macro to enable the
         visibility of these symbols in this header.

     The <math.h> header shall include definitions for at least the following types:

     float_t
          A real-floating type at least as wide as float.

     double_t
          A real-floating type at least as wide as double, and at least as wide as float_t
              .

     If FLT_EVAL_METHOD equals 0, float_t and double_t shall be float and double,
         respectively; if FLT_EVAL_METHOD equals 1, they shall both be double; if
         FLT_EVAL_METHOD equals 2, they shall both be long double; for other values of
         FLT_EVAL_METHOD, they are otherwise implementation-defined.

     The <math.h> header shall define the following macros, where real-floating indicates
         that the argument shall be an expression of real-floating type:


          int fpclassify(real-floating x);
          int isfinite(real-floating x);
```

```
     int isinf(real-floating x);
     int isnan(real-floating x);
     int isnormal(real-floating x);
     int signbit(real-floating x);
     int isgreater(real-floating x, real-floating y);
     int isgreaterequal(real-floating x, real-floating y);
     int isless(real-floating x, real-floating y);
     int islessequal(real-floating x, real-floating y);
     int islessgreater(real-floating x, real-floating y);
     int isunordered(real-floating x, real-floating y);

The <math.h> header shall provide for the following constants. The values are of type
     double and are accurate within the precision of the double type.

M_E  Value of e

M_LOG2E Value of log_2e

M_LOG10E Value of log_10e

M_LN2 Value of log_e2

M_LN10 Value of log_e10

M_PI Value of pi

M_PI_2 Value of pi/2

M_PI_4 Value of pi/4

M_1_PI Value of 1/pi

M_2_PI Value of 2/pi

M_2_SQRTPI Value of 2/ sqrt pi

M_SQRT2 Value of sqrt 2

M_SQRT1_2 Value of 1/sqrt 2


The header shall define the following symbolic constants:

MAXFLOAT
     Value of maximum non-infinite single-precision floating-point number.

HUGE_VAL
     A positive double expression, not necessarily representable as a float. Used as
         an error value returned by the mathematics library. HUGE_VAL evaluates to +
         infinity on systems supporting IEEE Std 754-1985.

HUGE_VALF
     A positive float constant expression. Used as an error value returned by the
         mathematics library. HUGE_VALF evaluates to +infinity on systems supporting
         IEEE Std 754-1985.

HUGE_VALL
     A positive long double constant expression. Used as an error value returned by
```

the mathematics library. HUGE_VALL evaluates to +infinity on systems supporting IEEE Std 754-1985.

INFINITY
A constant expression of type float representing positive or unsigned infinity, if available; else a positive constant of type float that overflows at translation time.

NAN A constant expression of type float representing a quiet NaN. This symbolic constant is only defined if the implementation supports quiet NaNs for the float type.

The following macros shall be defined for number classification. They represent the mutually-exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with FP_ and an uppercase letter, may also be specified by the implementation.

FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO

The following optional macros indicate whether the fma() family of functions are fast compared with direct code:

FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL

The FP_FAST_FMA macro shall be defined to indicate that the fma() function generally executes about as fast as, or faster than, a multiply and an add of double operands. The other macros have the equivalent meaning for the float and long double versions.

The following macros shall expand to integer constant expressions whose values are returned by ilogb(x) if x is zero or NaN, respectively. The value of FP_ILOGB0 shall be either {INT_MIN} or - {INT_MAX}. The value of FP_ILOGBNAN shall be either {INT_MAX} or {INT_MIN}.

FP_ILOGB0
FP_ILOGBNAN

The following macros shall expand to the integer constants 1 and 2, respectively;

MATH_ERRNO
MATH_ERREXCEPT

The following macro shall expand to an expression that has type int and the value MATH_ERRNO, MATH_ERREXCEPT, or the bitwise-inclusive OR of both:

math_errhandling

The value of math_errhandling is constant for the duration of the program. It is unspecified whether math_errhandling is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier

with the name math_errhandling, the behavior is undefined. If the expression (math_errhandling & MATH_ERREXCEPT) can be non-zero, the implementation shall define the macros FE_DIVBYZERO, FE_INVALID, and FE_OVERFLOW in <fenv.h>.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
double   acos(double);
float    acosf(float);
double   acosh(double);
float    acoshf(float);
long double acoshl(long double);
long double acosl(long double);
double   asin(double);
float    asinf(float);
double   asinh(double);
float    asinhf(float);
long double asinhl(long double);
long double asinl(long double);
double   atan(double);
double   atan2(double, double);
float    atan2f(float, float);
long double atan2l(long double, long double);
float    atanf(float);
double   atanh(double);
float    atanhf(float);
long double atanhl(long double);
long double atanl(long double);
double   cbrt(double);
float    cbrtf(float);
long double cbrtl(long double);
double   ceil(double);
float    ceilf(float);
long double ceill(long double);
double   copysign(double, double);
float    copysignf(float, float);
long double copysignl(long double, long double);
double   cos(double);
float    cosf(float);
double   cosh(double);
float    coshf(float);
long double coshl(long double);
long double cosl(long double);
double   erf(double);
double   erfc(double);
float    erfcf(float);
long double erfcl(long double);
float    erff(float);
long double erfl(long double);
double   exp(double);
double   exp2(double);
float    exp2f(float);
long double exp2l(long double);
float    expf(float);
long double expl(long double);
double   expm1(double);
float    expm1f(float);
```

```
long double expm1l(long double);
double   fabs(double);
float    fabsf(float);
long double fabsl(long double);
double   fdim(double, double);
float    fdimf(float, float);
long double fdiml(long double, long double);
double   floor(double);
float    floorf(float);
long double floorl(long double);
double   fma(double, double, double);
float    fmaf(float, float, float);
long double fmal(long double, long double, long double);
double   fmax(double, double);
float    fmaxf(float, float);
long double fmaxl(long double, long double);
double   fmin(double, double);
float    fminf(float, float);
long double fminl(long double, long double);
double   fmod(double, double);
float    fmodf(float, float);
long double fmodl(long double, long double);
double   frexp(double, int *);
float    frexpf(float value, int *);
long double frexpl(long double value, int *);
double   hypot(double, double);
float    hypotf(float, float);
long double hypotl(long double, long double);
int      ilogb(double);
int      ilogbf(float);
int      ilogbl(long double);

double   j0(double);
double   j1(double);
double   jn(int, double);

double   ldexp(double, int);
float    ldexpf(float, int);
long double ldexpl(long double, int);
double   lgamma(double);
float    lgammaf(float);
long double lgammal(long double);
long long llrint(double);
long long llrintf(float);
long long llrintl(long double);
long long llround(double);
long long llroundf(float);
long long llroundl(long double);
double   log(double);
double   log10(double);
float    log10f(float);
long double log10l(long double);
double   log1p(double);
float    log1pf(float);
long double log1pl(long double);
double   log2(double);
float    log2f(float);
long double log2l(long double);
```

```
double   logb(double);
float    logbf(float);
long double logbl(long double);
float    logf(float);
long double logl(long double);
long     lrint(double);
long     lrintf(float);
long     lrintl(long double);
long     lround(double);
long     lroundf(float);
long     lroundl(long double);
double   modf(double, double *);
float    modff(float, float *);
long double modfl(long double, long double *);
double   nan(const char *);
float    nanf(const char *);
long double nanl(const char *);
double   nearbyint(double);
float    nearbyintf(float);
long double nearbyintl(long double);
double   nextafter(double, double);
float    nextafterf(float, float);
long double nextafterl(long double, long double);
double   nexttoward(double, long double);
float    nexttowardf(float, long double);
long double nexttowardl(long double, long double);
double   pow(double, double);
float    powf(float, float);
long double powl(long double, long double);
double   remainder(double, double);
float    remainderf(float, float);
long double remainderl(long double, long double);
double   remquo(double, double, int *);
float    remquof(float, float, int *);
long double remquol(long double, long double, int *);
double   rint(double);
float    rintf(float);
long double rintl(long double);
double   round(double);
float    roundf(float);
long double roundl(long double);

double   scalb(double, double);

double   scalbln(double, long);
float    scalblnf(float, long);
long double scalblnl(long double, long);
double   scalbn(double, int);
float    scalbnf(float, int);
long double scalbnl(long double, int);
double   sin(double);
float    sinf(float);
double   sinh(double);
float    sinhf(float);
long double sinhl(long double);
long double sinl(long double);
double   sqrt(double);
float    sqrtf(float);
```

```
long double sqrtl(long double);
double   tan(double);
float    tanf(float);
double   tanh(double);
float    tanhf(float);
long double tanhl(long double);
long double tanl(long double);
double   tgamma(double);
float    tgammaf(float);
long double tgammal(long double);
double   trunc(double);
float    truncf(float);
long double truncl(long double);

double   y0(double);
double   y1(double);
double   yn(int, double);
```

The following external variable shall be defined:

```
extern int signgam;
```

The behavior of each of the functions defined in <math.h> is specified in the System Interfaces volume of IEEE Std 1003.1-2001 for all representable values of its input arguments, except where stated otherwise. Each function shall execute as if it were a single operation without generating any externally visible exceptional conditions.

The following sections are informative.

APPLICATION USAGE

The FP_CONTRACT pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another FP_CONTRACT pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another FP_CONTRACT pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state (on or off) for the pragma is implementation-defined.

RATIONALE

Before the ISO/IEC 9899:1999 standard, the math library was defined only for the floating type double. All the names formed by appending 'f' or 'l' to a name in <math.h> were reserved to allow for the definition of float and long double libraries; and the ISO/IEC 9899:1999 standard provides for all three versions of math functions.

The functions ecvt(), fcvt(), and gcvt() have been dropped from the ISO C standard since their capability is available through sprintf(). These are provided on XSI-conformant systems supporting the Legacy Option Group.

86.8 setjmp.h

<setjmp.h> 提供了 setjmp 函数和 longjmp 函数。

setjmp 函数会“标记”程序中的一个位置，随后可以用 longjmp 返回被标记的位置，这样就可以绕过正常的函数返回机制，从一个函数跳转到另一个(仍然在活动中的)函数。

setjmp 函数和 longjmp 函数主要用来处理程序执行过程中的重大问题。

86.9 signal.h

<signal.h> 提供了用于异常情况(信号)处理的函数，包括中断和运行时错误。

- signal 函数可以设置一个函数，使系统会在给定信号发生时自动调用该函数；
- raise 函数用来产生一个 ie 信号。

86.10 stdarg.h

<stdarg.h> 提供给函数可以处理任意个参数的工具，类似 scanf 和 printf。

86.11 stddef.h

<stddef.h> 提供了常用的类型和宏的定义，但没有声明任何函数。

<stddef.h> 定义的类型包括：

- ptrdiff_t
  ptrdiff_t 代表当进行指针相减运算时其结果的类型。
- size_t
  size_t 代表运算符 sizeof 的返回值类型。
- wchar_t
  wchar_t 代表一种足够大的、可以用于表示所有支持的地区的所有字符的类型。

上述所有这 3 种类型都是整数类型，其中 ptrdiff_t 必须是带符号的类型，而 size_t 则必须是无符号的类型。

<stddef.h> 中还定义了两个宏。

- NULL
  NULL 用来表示空指针。
- offsetof
  offsetof 宏需要两个参数：类型(结构类型)和指定成员(结构的一个成员)。
  offsetof 宏会计算结构的起点到指定成员间的字节数。

考虑下面的结构：

```
struct s{
  char a;
  int b[2];
  float c;
}
```

offsetof(struct s, a) 的值一定是 0，C 语言确保结构的第一个成员的地址与结构自身地址相同。

在上述示例中，无法确定的说出 b 和 c 的偏移量是多少。

- 一种可能是 offsetof(struct s, b) 是 1(因为 a 的长度是一个字节), offsetof(struct s, c) 是 5(假设整数是 16 位)。
- 某些编译器会在结构中留下一些空洞(无效字节), 从而会影响到 offsetof 产生的值。
  例如, 对于在 a 后面留下一个无效字节的编译器, b 和 c 的偏移量相应会是 2 和 6, 这就是 offsetof 宏的优点——对于任意编译器, 它都会返回正确的偏移量, 方便写出移植性更好的程序。

offsetof 有许多用途。例如, 假如需要将结构 s 的前两个成员写入文件, 但是忽略成员 c, 这里无法使用 fwrite 函数来写 sizeof(struct s) 个字节, 否则会将整个结构写入文件爱你, 但可以只写 offsetof(struct s, c) 个字节。

最后一点, 只有少数程序真的需要包含 <stddef.h>, 一些在 <stddef.h> 中定义的类型和宏在其他头文件中也会出现。例如, NULL 宏在 <locale.h>、<stdio.h>、<stdlib.h>、<string.h> 和 <time.h> 中也有定义。

86.12 stdio.h

<stdio.h> 提供了大量用于输入/输出的函数, 包括对顺序读写和随机读写的操作。

86.13 stdlib.h

<stdlib.h> 定义了大量无法划归于其他库的函数, 包括将字符串转换成数、产生伪随机值、执行内存管理任务、与操作系统通信、执行搜索与排序以及对多字节字符及字符串进行操作的函数。

86.14 string.h

<string.h> 提供了用于进行字符串操作的函数, 包括复制、拼接、比较及搜索等。

86.15 time.h

<time.h> 提供相应的函数来获取日期和时间、操纵时间和以多种方式显示时间等。

第 87 章

GNU C Library

GNU C Library(glibc)是 GNU 发布的 libc 库(即 C 运行库)。本质上,glibc 是程序运行时使用到的一些 API 集合,它们一般是已预先编译好并以二进制代码形式存在 Linux 类系统中。

glibc 是 Linux 系统中最底层的 API,几乎其它任何运行库都会依赖于 glibc。glibc 除了封装 Linux 操作系统所提供的系统服务外,它本身也提供了许多其它一些必要功能服务的实现。

glibc 包含了几乎所有的 UNIX 通行的标准,并且就像其他的 UNIX 系统一样,其内含的文件群分散于系统的树状目录结构中,像一个支架一样支撑起整个操作系统。