第十章　大型程序

实践出真知。

10.1　预处理指示

在 C 语言源程序中，凡是以 # 开头的都是给编译器的命令，告诉编译器如何编译源程序，例如 #include、#define 等。在真正开始编译之前，这些指示性语句[1]要被预先处理，例如做文件包含和符号替换等。完成这些预处理工作的程序叫预处理器。在早期，预处理器和编译器是分开的，源程序先经预处理器处理后，再送给编译器编译。现代的编译器都集成了预处理器，因此不再严格区分预处理和编译，然而人们仍然习惯性地说以 # 开头的指示为“预处理”指示。这里只讲在大型程序中常见的预处理指示及其经典用法。

10.1.1　文件包含和常量定义

文件包含

用于文件包含的 #include 指示有两种形式：

```
1  #include < 文件名>
2  #include " 文件名"
```

尖括号形式用于包含系统头文件，双引号形式用于包含自定义头文件。二者的细微差别在于：如果使用尖括号形式，那么编译器将直接在系统头文件所在的目录查找被包含的文件；如果使用双引号形式，那么编译器将先搜索当前目录，如果找不到，再搜索系统头文件所在的目录。

在 #include 指示中，也可以给出文件所在的路径，例如：

```
#include "C:\progs\include\Stack.h"
```

[1]这里所谓的“指示性语句”，是和“指令性语句”相对而言的。指令性语句产生目标代码，而指示性语句不产生目标代码。在 C 语言中，如果只说“语句”，一般是指以分号结尾的语句，然而以分号结尾的语句并非都是“指令性语句”，例如某些类型定义语句。

虽然双引号使文件名看起来很像字符串常量，但是编译器不会把反斜线看成转义符。不管怎样，因为反斜线经常被视为转义符，所以建议使用正斜线。为了使程序具有更好的适应性，建议使用相对路径，而不用绝对路径。双引号形式支持相对路径，而尖括号形式一般不支持。因此，上面的文件包含应该写成

```
#include "../include/Stack.h"
```

其中“../”表示上一级目录。如果要表示当前目录，则使用“./”。

虽然 #include 可以包含.c 文件，但是强烈建议只用来包含.h 文件。

常量定义

对于程序中使用到的常量，在大多数情况下，都应该用 #define 为之定义常量名。例如：

```
1  /* 常量名采用全大写，单词之间用下划线连接 */
2  #define PI 3.14
3  #define LENGTH_OF_MOBILE_NUMBER 11
4  #define LENGTH_OF_ID_CARD_NUMBER 18
5  #define MAX_NUM_OF_STUDENTS_IN_EACH_CLASS 60
```

有了常量定义，在程序中就可以使用常量名代替常量值。定义符号常量是一个好的编程习惯，在大型程序开发中尤为重要。其好处有以下三点：

1. 可读性好：当你看到常量名时，很容易想起它是什么常量。上面定义了四个常量，没有任何说明，你能猜出它们的含义吗？
2. 避免不一致性和打字错误：如果不定义圆周率 PI，那么直接写常量值很可能不一致甚至出错，例如出现 3.14、3.1416、3.14159、3.14195 等。
3. 便于修改：如果要把圆周率的精度提高到 3.1416，那么只要在宏定义处修改即可，其他地方无须任何改动。

10.1.2 条件编译

条件编译是指当满足某种条件时，才编译指定的代码，否则不编译。下面通过三个经典用法来学习条件编译。

保护头文件

在大型程序设计中，程序员往往需要自定义一些头文件。如果一个头文件被包含多次，那么将产生编译错误，这是一个常见问题。为了避免头文件的重复

包含问题，可以使用 #ifndef 和 #endif 指示来保护头文件的内容。例如，Stack.h 可以写成下面的形式：

Stack.h	解 释
`#ifndef STACK_H`	如果没有定义 STACK_H，
`#define STACK_H`	那么就定义 STACK_H。
`// Contents`	头文件的内容写在这里。
`#endif`	#if 的结束。

当首次 #include 这个文件时，编译器发现没有 STACK_H 的定义，于是就定义 STACK_H，然后把头文件的内容包含进来。当再次 #include 这个文件时，编译器发现已经有 STACK_H 的定义，于是就不再包含这个文件。关于其中的宏名 STACK_H，虽然可以任意取名，但是最好用头文件名和下划线的组合来命名。

其中 #ifndef 用来判断是否未定义某个符号。如果要判断是否已经定义了某个符号，那么可以用 #ifdef。我们也可以用 #if 指示和 defined 运算符代替 #ifdef 和 #ifndef：

```
#ifdef STACK_H      等价于  #if defined(STACK_H)
#ifndef STACK_H     等价于  #if !defined(STACK_H)
```

选择目标平台

如果一个程序要运行在多个不同的操作系统平台上，那么源程序往往需要面向不同的平台分别编译。对于与平台相关的代码，通常写成下面的形式：

```
#if defined(WINDOWS)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#else
#error "Error: No platform specified."
#endif
```

其中 #elif 是 #else 和 #if 的简写，#error 用来在预处理阶段输出错误信息。这样处理后，当面向 Windows 平台编译时，只要“#define WINDOWS”即可，无须修改其他代码。

C24 增加了 #elifdef 和 #elifndef，分别相当于 #elif defined 和 #elif !defined。还增加了 #warning，用来输出编译过程中的警告信息，用法与 #error 类似。

定义调试模式

在调试程序的过程中，一般需要加入一些 printf 语句，用于输出中间结果，以便定位程序错误。在程序调试好之后，需要把这些调试语句删除。当程序很大时，这是一件相当麻烦的事情。而且，有时还需要多次调试，当再次调试时，很可能又要把刚刚删除的语句再加进来。为了避免这个麻烦，可以自定义调试模式。首先定义一个 DEBUG 符号：

```
#define DEBUG 1
```

然后对于所有的调试代码，前后加上 #if 和 #endif：

```
#if DEBUG
printf("x=%d\n",x);  // 这是一条调试语句
printf("y=%d\n",y);  // 这也是一条调试语句
#endif
```

这样处理之后，当 DEBUG=1 时，程序处于调试模式，所有的调试语句将被编译并执行。如果要退出调试模式，只要把 DEBUG 的值改为 0 即可。当 DEBUG=0 时，调试语句不会被编译，所以不必删除。

10.2 多文件程序

10.2.1 划分头文件和源文件

大型程序通常由多个头文件和源文件组成，并通过 #include 把它们联系起来。这里以 8.2中的 SimpleStack 的实现及其在进制转换中的应用为例，讲述头文件和源文件的划分。

头文件的内容

在 8.2的程序中，定义了 Status 类型。这个类型在其他程序中也经常用到，为了使其他程序能共享此定义，不妨单独定义一个头文件 Status.h。

Status.h 文件

```
1  #ifndef STATUS_H
2  #define STATUS_H
3  typedef enum {OK=0, ERROR=1} Status;
4  // 为了区分不同的错误，也可以定义更多的错误代码
5  #endif
```

关于 SimpleStack 的定义和实现，应该划分成两个文件：SimpleStack.h 和 SimpleStack.c。一般说来，头文件的内容由文件包含、宏定义、类型定义、外部变量声明、函数接口和必要的注释所组成。其中的外部变量“声明”不是外部变量的“定义”，关于声明与定义的区别，详见 5.1.2中的全局变量部分。static 函数因为不向外部提供接口，所以不宜写在头文件中，而应该写在源文件中。最后强调一点，头文件必须有详细的注释。

SimpleStack.h 文件

```
 1  #ifndef SIMPLE_STACK_H
 2  #define SIMPLE_STACK_H
 3
 4  /* 本文件提供简单栈，除以下两处外，其他地方均不得修改。*/
 5  #define STACK_SIZE 100
 6  // 简单栈的容量。如需更大容量，请修改这里的定义。
 7
 8  typedef unsigned int Element;
 9  // 栈元素的数据类型。如需其他类型，请修改这里的定义。
10
11  #include "Status.h"
12  // 包含 Status 类型
13
14  typedef struct {
15      Element a[STACK_SIZE];
16      int n;
17  } SimpleStack;
18
19  void Init(SimpleStack* pStack);
20  // 将 pStack 指向的栈初始化。
21
22  bool IsEmpty(SimpleStack* pStack);
23  // 如果 *pStack 栈为空，返回 true，否则返回 false。
24
25  Status Push(SimpleStack* pStack,Element* pElem);
26  // 将 *pElem 压入 *pStack 栈。成功返回 OK，失败返回 ERROR。
27
28  Status Pop(SimpleStack* pStack,Element* pElem);
29  /* 将 *pStack 的栈顶数据元素弹出到 *pElem 中。
30   * 如果成功，返回 OK；如果失败，返回 ERROR。 */
31
32  #endif
```

当需要使用 SimpleStack 时，只要简单地包含 SimpleStack.h 即可。

源文件的内容

与头文件相对应的源文件一般只有具体的函数实现和必要的文件包含。此外，在源文件中也可以定义全局变量，然后在头文件中用 extern 关键字声明。因为全局变量对程序结构通常有较坏的影响，所以要慎用。下面的 SimpleStack.c 源文件是对 SimpleStack.h 头文件中函数接口的实现。

SimpleStack.c 文件

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

void Init(SimpleStack* pStack){
    pStack->n=0;
}

bool IsEmpty(SimpleStack* pStack){
    return pStack->n==0?true:false;
}

Status Push(SimpleStack* pStack,Element* pElem){
    这里是 Push 的具体实现。
}

Status Pop(SimpleStack* pStack,Element* pElem){
    这里是 Pop 的具体实现。
}
```

当把 SimpleStack 应用于进制转换时，应用程序的源文件 main.c 应该写成下面的样子：

main.c 文件

```
#include <stdio.h>
#include "Status.h"
#include "SimpleStack.h"

Status ConvertNumber(unsigned num, int base,char s[], int n);
/* 这里是函数注释，从略。*/

这里是 ConvertNumber 的具体实现。

int main(){
    这里是主函数的具体实现。
}
```

10.2.2　构建多文件程序

不管是构建单文件程序还是构建多文件程序，都需要经过编译和链接。

编译和连接

程序中的每个源文件都必须分别编译，通常一个源文件产生一个目标文件。目标文件在 Windows 系统中的扩展名为.obj，在 UNIX 系统中的扩展名为.o。头文件不需要单独编译。头文件被包含在源文件中，并与源文件一起编译。链接器把编译后产生的目标文件和库函数的代码链接在一起，生成可执行文件，即得可以运行的程序。

现代编译器都支持一步构建程序。如果在 Unix 系统上使用 gcc 编译器，那么可以使用下面的命令行来构建进制转换程序：

```
gcc -o convert main.c SimpleStack.c
```

这条命令先把两个源文件 (main.c 和 SimpleStack.c) 编译成目标文件，再链接成可执行文件。gcc 命令的-o 选项用来指定输出文件的名字，这里输出的是可执行文件，指定的文件名是 convert。

编写 makefile 文件

大型程序通常有很多源文件，在命令行上输入很多文件名是一件相当乏味的事情。为了便于构建大型程序，可以先建立 makefile 文件，再用 make 命令构建程序。对于上面的进制转换程序，我们为之建立这样的 makefile 文件：

makefile 文件

```
1  convert: main.o SimpleStack.o
2      gcc -o convert main.o SimpleStack.o
3  main.o: main.c SimpleStack.h Status.h
4      gcc -c main.c
5  SimpleStack.o: SimpleStack.c SimpleStack.h Status.h
6      gcc -c SimpleStack.c
```

在上面的 makefile 文件中，共有三组代码，每组称为一条编译规则。每条编译规则由两行组成：第一行描述依赖关系，冒号前面的是输出文件，冒号后面的是输入文件，输出文件依赖于输入文件；第二行是待执行的编译命令[2]，它根据输入文件生成输出文件。下面是对上述 makefile 中的每一条编译规则的解释：

[2]从源程序到可执行程序需要经过预处理、编译、汇编和链接四个步骤，现代编译器都把这些过程集成在一起，为了叙述方便，我们统称为编译。

第一条规则 第 1 行的 convert 是要输出的可执行文件的名字，这个文件依赖于 main.o 和 SimpleStack.o 文件。第 2 行的 gcc 是编译命令，-o 选项指定输出文件的名字。这条命令根据 main.o 和 SimpleStack.o 两个目标文件生成可执行文件 convert，可见这条 gcc 命令实际上是链接命令。

第二条规则 第 3 行显示 main.o 目标文件依赖于 main.c 源文件和 SimpleStack.h、Status.h 两个头文件。第 4 行对 main.c 进行编译，生成 main.o 文件，-c 选项表示只编译、不链接。

第三条规则 这条规则与第二条规则类似，不再解释。

使用 makefile 文件构建多文件程序有两个好处：一个好处是简化命令行的输入。否则每次构建程序时，都要在命令行输入很多源文件或目标文件的名字。另一个好处是避免重复编译，只有当所依赖的文件发生变化时，才有必要重新编译。例如，根据上述 makefile 文件中第 5 行所示的依赖关系，如果 SimpleStack.c、SimpleStack.h 和 Staus.h 都没有变化，那么第 6 行的编译命令就不会被执行。

使用 make 命令构建程序

在写好 makefile 文件之后，只要在命令行简单地使用

```
make
```

即可构建应用程序。这个 make 命令是 Unix 系统的标准命令[3]，它在当前目录搜索名为 Makefile 或 makefile 的文件，并根据文件中的编译规则构建程序。当使用 make 命令时，也可以指定输出文件的名字，例如：

```
make 输出文件
```

make 命令的输出文件可以是 makefile 文件中的任意一个输出文件，通常是可执行文件。如果不为 make 指定输出文件，那么默认的输出文件为 makefile 中第一条规则中的输出文件。因此，makefile 中的第一条规则的输出文件通常是最终的可执行文件，而其他规则的先后顺序则是任意的。

通过工程文件构建多文件程序

在集成开发环境中，当我们要写一个程序时，一般首先要建立工程文件。集成开发环境通过工程文件把组成程序的多个文件关联起来，形成一个完整的项目。这些文件包括头文件、源文件和资源文件等。在工程文件的帮助下，构建多文件程序十分方便，我们就不必编写繁琐的 makefile 文件了。

[3]作为类 Unix 系统，Linux 当然也有这个命令。

10.3　编写大型程序的基本原则

随着程序规模的增大，软件开发变得越来越困难，以致于在二十世纪六十年代出现“软件危机”。软件危机促成了软件工程学科的诞生，也促进了软件概念的发展。过去认为，软件 = 程序；现在认为，软件 = 程序 + 数据 + 文档。

不以规矩，无以成方圆。要开发大型程序，必须遵守一定的规矩。软件工程就是研究软件开发过程中应该遵守的规矩的学科。软件工程的内容超出了本书的范围，这里只介绍编写大型程序的两个最基本的原则。

10.3.1　遵守编程规范

遵守编程规范是编写大型程序的基本要求。本书已经在 4.7中介绍了命名规范和格式规范，不再重复。这里只通过一个案例让我们认识到遵守编程规范的重要性。2018 年 9 月 19 日，美国威斯康星州一位名叫 Anthony Tong 的程序员在 WTS Paradigm 软件公司办公大楼枪击了 4 名同事，因为他无法忍受他的同事不为函数接口写注释、不遵守驼峰命名法和强制更新版本等不规范做法。虽然这种过激行为必须受到法律的严惩，但是和不遵守编程规范的人一起写程序确实是一件非常痛苦的事情。

10.3.2　结构化设计

结构化设计是克服开发大型程序的困难的最有效和最基本的方法。结构化设计的概念最早是由荷兰计算机科学家狄克斯特拉于 1969 年提出的。什么是结构化设计？至今没有严格的公认的定义，因此不同的教材对结构化设计的解释也不尽相同。本书作者认为，软件的结构化设计就是基于分而治之的思想，按照高内聚和松耦合原则进行的软件结构设计，其目的是使软件具有良好的结构，从而使软件易于开发和维护。

软件的结构化通常表现为层次化、过程化和对象化等，可以统称为模块化。那么怎样理解模块呢？模块可大可小，它具有不同的粒度或层次。从语句粒度上看，顺序结构、选择结构和循环结构就是程序的三种基本模块。从过程粒度上看，一个函数就是一个模块。从对象粒度上看，一个对象就是一个模块。从文件粒度上看，一个文件就是一个模块。如果从更高的体系结构的层次上看，软件系统中的一层或一个子系统都是一个模块。

我们用内聚性衡量模块内部各元素彼此结合的紧密程度，用耦合性衡量模

块之间的相互耦合的紧密程度。本书在讲解函数设计时提到高内聚和松耦合的设计原则，这个原则不仅适用于函数层次上的模块化设计，也适用于所有层次上的模块化设计。软件结构化设计要求各个层次上的模块都要做到高内聚和松耦合。

10.4　编程实践：计算器

我们以计算器程序作为编程实践题目。希望通过这次编程实践，达到这样三个目的：第一、对本门课的知识和技能做全面的巩固和大幅度的提高；第二、掌握编写大型程序的基本技能；第三、为将来学习算法和数据结构打下坚实的基础。请读者务必高度重视这个编程实践！如果你能认认真真地按照要求一步一步地做下去，那么你的编程能力将会大大提高，你会发现本门课如此的简单，你一切都懂了！正所谓实践出真知。

请编写一个计算器程序，基本要求如下：用户从键盘输入一个表达式，程序以字符串形式接收、并计算表达式的值。构成表达式的合法字符包括阿拉伯数字、+（加）、−（减）、∗（乘）、/（除）、圆括号、正负号和小数点。

10.4.1　表达式分析

词法分析

以字符串形式出现的表达式由操作数、运算符和分隔符组成，例如

$$-((0.12+2.34)*4.5-5.6)*(+6.78-9)$$

其中 0.12 等是操作数，+、−、∗ 是运算符，括号是用来改变优先级的分隔符。我们称它们为单词。显然操作数是由多字符组成的单词，至于运算符，虽然题目中要求的 +、−、∗、/ 四个基本运算符都只有一个字符，但是由多字符组成的运算符并不罕见，例如 C 语言中的“`>=`”、“`++`”和“`sizeof`”等。分隔符也可以由多字符组成，例如 C 语言中表示注释的“`//`”和“`/*`”。因此一个扩展性好的设计应该把操作数、运算符和分隔符都按照多字符单词来处理。词法分析的任务就是把组成表达式的单词提取出来。

正负号的处理

我们既用 ± 表示正负号，又用 ± 表示加减运算符，这就要求程序必须能够区分。通过仔细观察，你会发现可以这样判定：

1. 如果 ± 作为第一个字符出现，那么它一定是正负号。
2. 如果 ± 紧跟在左括号的后面，那么它一定是正负号。
3. 除上述两种情况外，其他的 ± 都是加减运算符。

那么怎样处理正负号呢？因为正负号其实就是和零的加减运算，所以一个简单的办法是在正负号的前面加零，这样就把正负号和加减运算符统一起来了。例如：

$$-1.2*(+2.3) \implies 0-1.2*(0+2.3)$$

10.4.2　表达式转换

我们书写表达式时，一般把二元运算符放在两个操作数的中间，例如 $a+b$，表示将 a 和 b 做 $+$ 运算，我们称这种表达式为中缀表达式。虽然中缀表达式看起来十分自然，但是计算机处理起来并不方便。1929 年，波兰数学家 Lukasiewicz 提出一种把运算符放在两个操作数前面的表示法，例如 $+ab$，表示做 $+$ 运算，两个操作数是 a 和 b。我们称这种表达式为前缀表达式，又称波兰式。如果把运算符写在两个操作数的后面，例如 $ab+$，那么就得到一种后缀表达式，又称逆波兰式。由于计算机求后缀表达式的值要方便得多，这就涉及到中缀表达式到后缀表达式的转换问题。

中缀表达式转后缀表达式

在转换过程中，需要使用一个栈，用来临时存放运算符和分隔符。转换步骤如下：

1. 从中缀表达式中取一个单词。
2. 如果是操作数，把这个操作数输出到后缀表达式中。
3. 如果是运算符，把它和栈顶元素的优先级作比较。如果它的优先级大于栈顶元素的优先级，那么就直接进栈；否则，就把栈顶元素弹出并输出到后缀表达式，重复这个操作，直到栈顶元素的优先级小于该运算符的优先级为止，然后该运算符进栈。
4. 如果是左括号，直接进栈。
5. 如果是右括号，弹出栈顶元素并依次输出到后缀表达式，直到弹出左括号为止，左括号不输出。
6. 重复执行 1∼5 步，直到把中缀表达式的所有单词处理完毕为止。
7. 把栈中的运算符依次弹出并输出到后缀表达式。

运算符的优先级

运算符	#	(	+	−	*	/
优先级	0	1	2	2	3	3

如上表所示，加减同级、乘除同级、乘除的优先级固然高于加减。此外，为了使程序设计方便，可以规定左括号的优先级低于加减；还可以增设一个栈底标志 #，并规定其优先级为最低。

我们通常认为括号具有更高的优先级，那么这里的左括号的优先级为什么比加减还低呢？当左括号入栈后，应该先计算括号内的式子，因此栈内的其他运算符都不予考虑。这时栈顶的左括号把栈内其他运算符隔开，作为括号底部 (左部) 的标志，其作用类似于栈底标志 #，此时尚不知右括号在哪里。中缀表达式中左括号后的第一个运算符必须保证进栈，而运算符进栈的条件是其优先级高于栈顶元素的优先级，因此我们规定栈内左括号的优先级低于加减运算符的优先级。事实上，规定左括号的优先级和增设栈底标志都不是必须的，这样做只是为了使程序设计更方便。只有当你亲手实现这个程序时才能明白这样做的好处。

转换实例

中缀表达式是以字符串形式输入的，后缀表达式一般也是以字符串的形式输出的，词法分析提取的单词一般也是字符串形式。表 10.1显示了中缀表达式 $a+(b*c-d)/e$ 转后缀表达式的过程，其中 a、b 等字母表示操作数。然而在实际的计算器中，操作数都是数字串形式的常量。当把中缀形式的算术表达式转成后缀形式时，为了区分单词，应该加入自定义的分隔符。例如，把 $1.73+2.72*3.14$ 转后缀表达式时，如果不加分隔符，将输出“1.732.723.14 * +”；如果加入空格作分隔符，则输出“1.73␣2.72␣3.14␣*␣+”。

10.4.3　表达式求值

数字串转浮点数

表达式求值的最基本操作是把字符串形式的操作数转成浮点形式。对于单个数字字符，只要减去字符 ‘0’ 的 ASCII 值即可得到相应的数值。而对于数字字符串，则需要把各位上的数字乘以权值再求和。例如：

“123.45” $\Longrightarrow 1\times10^2+2\times10^1+3\times10^0+4\times10^{-1}+5\times10^{-2}=123.45$

表 10.1　$a+(b*c-d)/e$ 转后缀表达式

输入	栈 (左为底)	输出后缀表达式	说　明
初始	#		设置栈底标志 #，其优先级最低。
a	#	a	操作数直接输出。
$+$	# $+$	a	$+$ 的优先级高于 #，直接入栈。
$($	# $+$ $($	a	左括号直接入栈。
b	# $+$ $($	$a\ b$	操作数直接输出。
$*$	# $+$ $($ $*$	$a\ b$	$*$ 的优先级高于 (，直接入栈。
c	# $+$ $($ $*$	$a\ b\ c$	操作数直接输出。
$-$	# $+$ $($ $-$	$a\ b\ c\ *$	弹出并输出 $*$ 后，$-$ 入栈。
d	# $+$ $($ $-$	$a\ b\ c\ *\ d$	操作数直接输出。
$)$	# $+$	$a\ b\ c\ *\ d\ -$	弹出栈顶元素，直到弹出左括号。
$/$	# $+$ $/$	$a\ b\ c\ *\ d\ -$	$/$ 的优先级高于 $+$，直接入栈。
e	# $+$ $/$	$a\ b\ c\ *\ d\ -\ e$	操作数直接输出。
结束	#	$a\ b\ c\ *\ d\ -\ e\ /\ +$	弹出所有运算符，并依次输出。

当用程序实现时，从左到右扫描数字串，整数部分和小数部分的转换不便于统一，应该分别计算。设 s[] 是数字字符串形式的十进制数，只含有 0~9 和小数点，把 s[] 转成浮点数的算法如下：

```
ConvertToFloat(s[])
 1  i=0;
 2  value=0;
 3  while s[i]!='\0' and s[i]!='.', do {
 4      value=value*10+(s[i]-'0');
 5      i=i+1;
 6  }
 7  if s[i]=='\0', then {
 8      return value;
 9  }
10  /* 以上是整数部分，以下是小数部分 */
11  i=i+1;
12  weight=0.1;
13  while s[i]!='\0', do {
14      value=value+(s[i]-'0')*weight;
15      weight=weight/10;
16      i=i+1;
17  }
18  return value;
```

后缀表达式的计算

后缀表达式的求值算法如下：

1. 从后缀表达式读取一个单词。
2. 如果是操作数，把它转成浮点数，然后进栈。
3. 如果是运算符，识别出是哪种运算，然后弹出两个操作数，做相应的运算，再把计算结果压入栈。
4. 重复 1～3 步，直到把后缀表达式中的单词处理完毕。最后栈中只剩一个数，即为表达式的值。

设有中缀表达式“0.6＋(1.1＋1.9)＊3.1”，转成后缀表达式为：

0.6␣1.1␣1.9␣+␣3.1␣*␣+

表 10.2显示了对它的计算过程，栈中最后剩下的 9.9 即为表达式的值。

表 10.2 后缀表达式“0.6␣1.1␣1.9␣+␣3.1␣*␣+”的计算过程

输入单词	栈（左边是栈底）	说　明
0.6	0.6	把字符串 0.6 转成浮点数，然后压入栈。
1.1	0.6　1.1	把字符串 1.1 转成浮点数，然后压入栈。
1.9	0.6　1.1　1.9	把字符串 1.9 转成浮点数，然后压入栈。
＋	0.6　3.0	弹出 1.9 和 1.1，相加，再把结果 3.0 压入栈。
3.1	0.6　3.0　3.1	把字符串 3.1 转成浮点数，然后压入栈。
＊	0.6　9.3	弹出 3.1 和 3.0，相乘，再把结果 9.3 压入栈。
＋	9.9	弹出 9.3 和 0.6，相加，再把结果 9.9 压入栈。

中缀表达式的直接计算

前文表明，要计算一个中缀表达式的值，应该先转成后缀表达式，再计算后缀表达式的值。事实上，在把中缀表达式转后缀表达式的同时，也可以完成计算，而不必输出后缀表达式。具体做法与前文所述基本相同，只是把这两步合成一步而已。下面给出简要的描述。

同样要使用两个栈，设用于存储操作数的栈为 dataStack，用于存储运算符的栈为 operStack。从左向右扫描中缀表达式，如遇操作数，压入 dataStack；如遇运算符，压入 operStack。当从 operStack 弹出运算符时，就从 dataStack 弹出两个操作数，并做相应的运算，再把运算结果压入 dataStack。这样就可以直接计算中缀表达式，而不必输出后缀表达式。

10.4.4　计算器程序的优化和增强

请读者在亲手实现上述计算器程序之后再来阅读这部分内容。

现在你已经实现了计算器的基本功能。虽然你的计算器已经做好了，但是不能满足于此。本书作者在这里带着你一步一步地优化和增强你的计算器程序，如果你认真去做了，那么你的编程水平将得到极大的提高。

(1) 规范化

检查你的源程序，是否做到了规范化？源程序是按照文件包含、常量定义、类型定义、函数声明、函数实现、主函数这样的顺序书写的吗？常量名是全大写并用下划线连接单词吗？变量名是采用小驼峰表示法吗？自定义类型名是采用大驼峰表示法吗？函数名规范吗？这些名字能让其他程序员见名知义吗？在常量定义处、类型定义处、函数声明处和一些不易懂的代码处是否有清晰的注释？该对齐的地方对齐了吗？该缩进的地方缩进了吗？等等。格式问题多种多样，请仔细检查。代码规范是开发大型程序的基本要求。

(2) 调试模式

调试程序通常是一件很麻烦的事情，开发环境提供的单步执行和设置断点等调试手段往往不能满足调试需求。为了减轻调试负担，一般要自定义调试模式。请为你的程序增加调试模式，参见 10.1.2。

(3) 全面测试

墨菲定律说：“凡是可能出错的，一定会出错。[4]”请先把每个函数测试好，再综合测试。给你的程序以各种不同的输入，包括正确的输入和错误的输入，程序必须十分健壮，能够从错误中恢复正常而不至于崩溃。而且不要忘记，每次修改错误时，很可能又引入了新的错误。

(4) 支持带函数的表达式

请增强你的计算器，让它支持三角函数、反三角函数、平方根函数和乘方运算符。乘方运算符用“^”表示，例如 x^2 表示 x^2。这要求你的程序具有良好的可扩展性，有独立的词法分析，能够方便地增加函数和运算符。

(5) 批量计算

请用你的计算器做批量计算。也就是说，用户把待计算的多个表达式输入到一个文件中，然后你的计算器对这个文件中的表达式逐个计算，并把计算结果输出到另一个文件中。这实际上是在考察你对文件的操作。

[4]这句话实际上是菲纳格 (Finagle) 说的，又称菲纳格定律，原文是：“If anything can go wrong, it will.”这句话是对墨菲定律最好的表述。因为最早提出类似观点的是墨菲 (Edward A. Murphy)，所以人们仍然把这句话称为“墨菲定律”。

(6) 泛型栈

在计算表达式时，需要使用两个栈：一个是操作数栈，另一个是运算符栈。因为操作数栈存储的是浮点型元素，而运算符栈存储的是整型或字符型元素，它们具有不同的数据类型，所以我们需要实现两个栈，而这两个栈的实现代码几乎是一样的，只是栈元素的类型不一样而已。请实现一个可以存储任意类型数据的泛型栈，参见 8.5.2，并应用到你的计算器程序中。

(7) 面向对象

初学编程时的思维方式基本都是面向过程的，然而只有面向过程的思维方式不足以克服开发大型程序的困难。既然你的计算器程序使用了栈对象，请检查栈对象是否完整。如果你的栈是残缺不全的，那么就足以证明你的思维方式不是面向对象的。请把缺失的函数补上，即使你的计算器程序没有用到，也要写全，因为那是一个完整的对象。同时不要忘记对栈对象做全面的测试。

(8) 结构化

请用高内聚和松耦合的原则检查你的程序是否做到了结构化。对于程序中的每个函数，其设计是否合理？它是高内聚和松耦合的吗？函数内部的逻辑是否清晰易懂？在做好结构化之后，你就可以考虑把源程序分成多个文件了。虽然 C 语言不提供对象的封装功能，但是我们可以把做好的对象放在独立的文件里。这时你的程序大致由 3 个文件组成：Stack.h、Stack.c 和 main.c。如果你的程序真正做到了规范化和结构化，那么你就具备了开发大型程序的能力。

(9) 链表和链式栈

请实现一个链表，然后用这个链表实现栈，并把这个栈应用到你的计算器程序中。这样你的程序大致由这些文件组成：Status.h、LinkedList.h、LinkedList.c、LinkedStack.h、LinkedStack.c 和 main.c。此外，还有两个已经做好的备用文件：Stack.h 和 Stack.c，你随时可以用 Stack.h 替换 LinkedStack.h 而无须修改程序中的其他代码。这一步要求使用链式栈是在考察你对链式结构的操作。如果你做到了，那么你已经为数据结构和软件工程等课程的学习打下了坚实的基础。

编程实践题

1. 开发一个学生成绩管理系统。系统的需求请自己分析。
2. 为超市开发一个商品管理系统。系统的需求请自己分析。
3. 编写俄罗斯方块游戏程序。