第十章　程序开发技术

本章是这本书实质性内容的最后一章，这里要通过一些程序实例，综合性地总结本书中所讨论的各方面内容，包括 C 语言里的一些主要机制以及写程序的各种基本技术，讨论一些在编写较大程序时需要考虑的问题，同时介绍一些有用的编程技术。

10.1　分别编译和 C 程序的分块开发

随着学习的不断深入，我们在编程序时要处理的问题变得越来越复杂，写出的程序也越来越长了。应该看到，教科书里的问题都做了许多简化，任何实际问题通常都比书里的例子和习题中提出的问题更复杂得多。随着程序变得越来越大，许多新问题也随之出现了。

最明显情况的是源程序文件越来越大。大文件处理起来很麻烦，屏幕只能显示其中很小一部分，翻阅查找信息都更费精力，更费时间。大的源文件总是慢慢建立起来的，建立过程中经常需要编译调试其中的某个或者某些小部分，做起来就非常麻烦。

一个可能的方法是把不关心的部分“注释”起来。但这一方法很难系统化地采用，因为新的临时性注释可能与程序中原有的注释冲突，改起来常常需要做许多变动。大型源文件的另一个缺点是，哪怕你只改动了其中的一个字符，为了后面的程序调试，也需要把整个文件重新加工一次，这又会浪费许多时间。此外，这种做法也相当危险。由于许多东西都放在一个大文件里，如果开发或调试新加入部分时无意改动了已经做好的部分，那就又会带来无穷无尽的烦恼。另一方面，一个大的软件系统通常是几个、几十个甚至更多人共同工作的结果。许多人怎样用一个源文件工作？我们根本无法想象许多人一起改动一个大地程序文件时会发生什么事情。

总之，实际程序开发通常不可能采用一个文件写一个程序的方式（除非程序很小，就像我们开始学习时遇到的例子，实际上那些例子也牵涉到多个文件，因为每个程序都用到系统的头文件），因为存在许多无法解决的困难，为此人们考虑了采用多个源文件开发一个程序的方式。C 语言的设计本身就考虑了这个问题，并提供了一些支持机制。这里我们将以前一章的程序实例为例，介绍这方面的情况。

10.1.1　分块开发的问题和方法

采用多个源程序文件开发一个程序的过程通常称为*分块开发*。分块开发得到 C 语言程序加工过程的支持，因为 C 语言允许加工对象不是完整的程序，而是一个个的源程序文件。分块开发出的各个源文件可以分别进行预处理和编译，得到一组程序目标文件；最后可以用连接程序把这些目标文件和系统库函数、程序基本运行系统等连接起来，形成最后的可执行文件。第五章的图 5.5 描述了这种开发过程的基本情况。

预处理的对象通常是一组源程序文件，得到的是一组新的源文件，其中不再包含任何预处理命令（所有文件包含命令都用相应文件的内容取代，所有的宏都已替换掉，所有条件编译的效果都已体现在作为预处理结果的源文件中，应该清除的代码段均已除去）。编译程序处理的就是一组这样的源程序（人们通常将称它们为编译单位，translation units），将它们分别编译为一组目标代码模块。只有到了连接时才需要一个程序的全部信息（程序的所有目标代码模块）。连接系统将所有有关的目标代码模块装配到一起，解决这些模块里所有的定义和使用之间的关联问题，并将必要的运行支持模块和系统库函数代码也连接到程序里，最终

得到一个可执行程序。

分块开发中最重要工作就是程序结构的“物理”组织。在 C 语言里做分块开发，需要借助于 C 系统的预处理功能，以达到对源程序的适当物理划分，并设法保证组成同一个程序的不同部分之间的一致性，以便使编译之后的目标代码模块能组合成一个具有内在一致性的完整的可执行程序。

在一般性地讨论分块开发中的问题之前，让我们先看一个工作实例。

10.1.2 程序实例：学生成绩处理

这里还用读者已熟悉的例子：学生成绩文件的处理。这个例子也作为本章后面许多讨论的公用实例。我们对这个实例做一些扩充，假定成绩文件给出了每个学生一门课程的期中考试成绩，平时成绩和期末考试成绩。程序需要按照一定比例计算出课程的最终成绩，并能在此基础上做统计，画学生成绩分布直方图，做排序输出等。

在考虑分块开发问题之前，我们先给出一个在当前情况下的合理实现。首先为该工作起一个名字，称之为 stu，这个名字也将作为程序主文件的名字。我们假定文件中的成绩数据具有如下形式：

```
02001014 zhangshan 86 80 91
02001016 lisi 77 90 69
... ...
```

数据在文件里分行排列，每行给出一个学生的成绩数据。

为完成这一程序，我们先定义表示学生记录的结构类型作为程序里的基本数据表示。程序需要包含一批标准头文件，引进几个程序中使用的常量：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

enum {
   MAXNUM = 400,
   MIDDLE = 20, EXECISE = 30, FINAL = 50, /* 成绩比例 */
   HISTLEN = 60,        /* 最长行的长度（字符数）*/
   SEGLEN = 5,          /* 分段长度 */
   SEGNUM = 100/SEGLEN+1 /* 分段数，根据分段长度自动算出 */
};

/* 公用类型的定义 */
typedef struct {
   unsigned long num;
   char name[20];
   double mid, exe, final, score;
} StuRec;

/* 全局性数据对象的定义 */
StuRec students[MAXNUM];
```

这里用一个 StuRec 类型的全局数组存储程序里使用的公共数据。

下面给出程序里的一些主要部分，有些部分则给只出梗概。程序启动后进入交互状态，主函数 main 处理学生记录文件的打开关闭：

```
int main(void) {
   FILE *fp;
   char fn[128];

   do { /* 交互式获取文件名 */
       getnstr("Student record file name: ", 128, fn);
       if ((fp = fopen(fn, "r")) == NULL)
```

```
            printf("Can't open file: %s\n", fn);
        else {
            commander(fp, fn);
            fclose(fp);
        }
    } while (next("file"));

    return 0;
}
```

这里的 getnstr 是第 8 章单词练习程序里开发的读取一个字符串的函数，这里将读入的字符串用作文件名，设法打开该文件。commander 是程序里的一个主要函数。对一个打开的文件，commander 首先将文件里的学生成绩记录读入，而后转入交互命令的处理：

```
void commander (FILE* fp, char *fn) {
    int n, cmd;
    n = readSRecs(fp, MAXNUM, students);
    if (n <= 1) {
        printf("File %s: too few data items.\n", fn);
        return;
    }
    printf("File %s: %d student records read.\n", fn, n);

    do {
        cmd = getcmd("Cmds: 1,Statistics; 2,Histogram; "
                "3,Sort and store to file\n", 1, 3);
        switch (cmd) {
        case 1: statistics(n, students);
                break;
        case 2: histogram(n, students, SEGLEN);
                break;
        case 3: sortoutput(n, students);
                break;
        }
    } while (next("command"));
}
```

除成绩统计和直方图外，这里还增加了一个成绩排序输出的函数 sortoutput。

上面两个函数都用到前面背单词程序定义的函数 next。此外这里还需定义一个取得用户命令的函数 getcmd，它显示一个信息串，要求读入一定范围里的一个数。该函数的实现应该参考 getnstr 和前面几次定义的 getnumber，读入一个数之后应该丢掉整个行中其余的东西。下面是几个函数的原型：

```
int getcmd (char *prompt, int c1, int cn);
int next(char s[]);
void getnstr (char prompt[], int lim, char bf[]);
```

这几个简单函数的代码就不列在这里了。

完成统计和生成直方图的函数需要做少许修改，以处理目前所采用的结构数组：

```
void statistics(int n, StuRec tb[]) {
    int i;
    double s, sum, avr;

    if (n <= 1) {
        printf("Data too few. Statistics stop.\n");
        return;
    }

    for (sum = 0.0, i = 0; i < n; ++i) sum += tb[i].score;
    avr = sum/n;
    for (sum = 0.0, i = 0; i < n; ++i)
        sum += (tb[i].score - avr)*(tb[i].score - avr);
    s = sqrt(sum/(n-1));
```

```
    printf("Total students: %d\n", n);
    printf("Average score: %lf\n", avr);
    printf("Standard deviation: %lf\n\n", s);
}

void prtHH(int n) {
    int i;
    for (i = 0; i < n; ++i) putchar('H');
}

void histogram(int n, StuRec tb[], int high) {
    int i, mx;
    int segs[SEGNUM];

    for (i = 0; i < SEGNUM; ++i) segs[i] = 0; /* 初始化 */
    for (i = 0; i < n; ++i) /* 统计各分段人数 */
        segs[(int)tb[i].score / SEGLEN]++;

    for (mx = 0, i = 0; i < SEGNUM; ++i) /* 为规范化找最大个数 */
        if (segs[i] > mx) mx = segs[i];
    for (i = 0; i < SEGNUM; ++i) { /* 输出 */
        printf("<%3d: %4d|", (i+1)*SEGLEN, segs[i]);
        prtHH(segs[i]*high/mx);
        putchar('\n');
    }
    putchar('\n');
}
```

为提供排序输出功能，需要定义函数 sortoutput：

```
void sortoutput(int n, StuRec tb[]) {
    char fn[128];
    FILE *fp;
   do { /* 交互式获取文件名 */
        getnstr("File name for saving: ", 128, fn);
        if ((fp = fopen(fn, "w")) == NULL)
            printf("Can't open file: %s\n", fn);
        else {
            qsort(tb, n, sizeof(StuRec), scrcmp);
            printSRec(fp, n, tb);
            fclose(fp);
            break;
        }
   } while (next("Really want to save"));
}
```

其中调用了标准库函数 qsort，用到前一章定义的学生记录成绩比较函数：

```
int scrcmp(const void *vp1, const void *vp2) {
    StuRec *p1 = (StuRec*)vp1, *p2 = (StuRec*)vp2;
    return p1->score > p2->score ? 1 :
           p1->score == p2->score ? 0 : -1;
}
```

函数 sortoutput 还通过交互取得所需的输出文件名，这里还调用了一个输出函数:

```
int printSRec(FILE *fp, int limit, StuRec tb[]) {
    int i;
    for (i = 0; i < limit; ++i)
        fprintf(fp, "%-10lu%20s%6.1f%6.1f%6.1f%6.1f", tb[i].num,
            tb[i].name, tb[i].mid, tb[i].exe, tb[i].final, tb[i].score);
    return 0;
}
```

它以与输入函数 readSRecs 一致的形式将 tb 里的数据通过流 fp 输出到文件。注意，这

里用到第八章讨论的一些格式控制描述，以便保证表格对齐。

由于学生记录保存在一个结构数组里，而且每个人的成绩数据有三项，读入文件的函数也需要修改。我们可以采用下面几个函数完成一个文件中的学生记录输入：

```
static int check(double x) {
    return x >= 0.0 && x <= 100.0;
}

static int readrec(FILE* fp, StuRec *stp) {
   char s[256];
   if (fgets(s, 256, fp) == NULL) return EOF;
   if (sscanf(s, "%lu %s%lf%lf%lf", &stp->num, stp->name,
            &stp->mid, &stp->exe, &stp->final) == 5)
        if (check(stp->mid) && check(stp->exe) && check(stp->final)) {
            stp->score = (stp->mid*MIDDLE + stp->exe*EXECISE +
                          stp->final*FINAL) / 100;
            return 1;
        }
    return 0;
}

int readSRecs(FILE *fp, int limit, StuRec tb[]) {
    int i = 0, line = 1, n;
    double x;

    while (i < limit && (n = readrec(fp, &tb[i])) != EOF) {
        if (n == 0)
            printf("Data error, line %d\n", line);
        else ++i;

        ++line;
    }

    if (i == limit && !feof(fp)) { /* 还有数据 */
        printf("Too many data. Output is not correct.\n");
        return 0;
    }

    return i;
}
```

这里用行式输入函数将一行读入字符数组，而后用 sscanf 分析和赋值。如果一行的信息能够正确读入，readrec 就检查各个数是否满足值的限制，在未发现问题的情况下按照给定比例计算出学生的最终成绩。注意 readSRecs 最后的 if 语句，这里用标准函数 feof 检查输入流是否到达文件结束。在数组满但文件未读完的情况下生成错误信息。

完成了上面所有函数，并将它们适当排列或者加入适当的原型，这个程序就完成了。从逻辑上说，这个程序由一批函数定义组成，此外它还用 #include 包含了几个标准库头文件，定义了几个常量和一个全局性的结构数组。

10.1.3 分块重整

本节的目的是研究程序开发中的物理结构组织，主要想讨论如何将一个较大的程序划分为一组物理上独立的程序块（程序文件），而又能保证这些文件间的正确逻辑联系，保证这些文件的最终编译结果能连接成一个正确的程序。在抽象地讨论分块开发问题之前，作为实例，现在来考虑上面程序的物理划分。

上面源程序可以通过编译加工，最终得到可执行程序。这个编程工作已经完成了。现在再来考虑它的逻辑组织似乎有些马后炮的味道，因为程序已经做好，为什么还要去整理它，将它分块呢？实际上，有时人们确实需要这样做，因为程序常常由于新情况和需求而不断发

展变化。在不断的变化中，程序的规模通常会不断扩大，在这种扩大过程，重整程序的物理结构就成为了一项非常重要的工作。如果将程序的物理结构整理得比较好，就使它能更好地适应进一步的修改和扩充需求。当然，合理的划分首先需要分析情况，提出各种可能性。下面的讨论就是希望使读者能从这个分析中，看到在将程序划分为一些物理组成部分的过程中可能遇到的问题，以及应该如何做出选择。

上述程序反应了较简单的 C 程序的典型情况：程序包含了若干标准库（或其他库）头文件，定义了若干公用类型（这里的 StuRec），定义了一些函数，还有一些全局性的变量或常变量，一个主函数着控制整个程序的执行。

对于这个程序的一种可能划分方式是将它分为 4 个部分：主函数 main 和实现命令循环的 commander 独立出来，形成程序的高层控制部分。另外的三个部分是：完成输入输出的一组函数；完成对成绩记录的各种处理的函数；还有若干辅助性的功能函数，如 next、getnstr 等。现在考虑如何按上述功能划分将程序分为 4 个源程序文件，使它们可以分别编译，并保证最终能连接为一个有机的完整程序。

应该注意，这些部分并不是相互独立的：主函数所在的部分需要用到其他许多函数，许多部分都依赖于公用的结构 StuRec。我们可以在每个文件前面加上结构 StuRec 的定义、有关的函数原型定义等等，使它们可以独立地编译。但那样做有很大的危险：如果在此之后某个文件修改了，其他文件将得不到任何信息，如果所做修改破坏了程序的内在一致性，编译程序通常也不能帮我们检查出来。这绝不是我们希望看到的情况。

为了防止上述危险的出现，就必须贯彻前面提出的原则：*使同一程序对象的定义点和所有使用点都能参照同一个描述*。就目前问题而言，达到这一目标的一种方式是将所有类型定义、常量定义放到在一个文件中，把定义和使用出现在不同文件中的函数原型也都列在这个公用的信息文件里，而后让各个源程序文件都参看这个文件里的信息（#include 它）。按 C 语言习惯，为此目的创建的文件称为头文件，它们一般用 .h 作为文件名后缀，其作用就是为其他文件提供信息。我们将这个头文件命名为 stu.h，其内容如下：

```
/* file stu.h, 程序 stu 的公共头文件 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

/* 程序的公用常量定义 */
enum {
   MAXNUM = 400,
   MIDDLE = 20, EXECISE = 30, FINAL = 50, /* 成绩比例 */
   HISTLEN = 60,        /* 最长行的长度（字符数）*/
   SEGLEN = 5,          /* 分段长度 */
   SEGNUM = 100/SEGLEN+1 /* 分段数，根据分段长度自动算出 */
};

/* 公用类型定义 */
typedef struct {
   unsigned long num;
   char name[20];
   double mid, exe, final, score;
} StuRec;

/* 全局性数据对象的外部说明，此时不必给出数组长度 */
extern StuRec students[];

/* 在某一个文件里定义，在其他文件里使用的全部函数的原型 */
int readSRecs(FILE *fp, int limit, StuRec tb[]);
int printSRec(FILE *fp, int limit, StuRec tb[]);
```

```
void statistics(int n, StuRec tb[]);
void histogram(int n, StuRec tb[], int high);
void sortoutput(int n, StuRec tb[]);
int next(char s[]);
void getnstr (char prompt[], int lim, char bf[]);
int getcmd (char *prompt, int c1, int cn);
```

请注意，这里只列出了那些在程序的一个物理部分（一个源文件）里定义，同时又需要在程序的另一部分中使用的函数的原型。下面看各个源程序文件。首先是主程序文件，这里略去了所有不必要的细节，以节约篇幅：

```
/* file stu.c，程序 stu 的主源程序文件 */
#include "stu.h"

/* 全局性的数据对象通常定义在主文件里，或者按照归属原则定义在某个文件里 */
StuRec students[MAXNUM];

void commander (FILE* fp, char *fn) { ... ... }
int main(void) { ... ... }
```

主程序文件里定义了两个函数和一个外部变量，它通过包含命令引入了头文件 stu.h，因此就可以得到这里所用的所有函数的类型信息。

完成输入输出的文件也与此类似：

```
/* file stu_io.c，程序 stu 的输入输出源程序文件 */
#include "stu.h"

static int check(double x) { ... ... }
static int readrec(FILE* fp, StuRec *stp) { ... ... }
int readSRecs(FILE *fp, int limit, StuRec tb[]) { ... ... }
int printSRec(FILE *fp, int limit, StuRec tb[]) { ... ... }
```

由于函数 readrec 和 check 只在这个源文件里使用，故其原型不必写进公共信息文件。基于同样原因，我们将这两个函数定义为 static 函数，使函数的名字局部化（static 函数的作用域限制在其定义所在的编译单位里），防止它们与其他源文件里的全局名字冲突（万一开发其他部分的人也定义了同名的函数）。

程序的数据处理部分定义为下面源程序文件：

```
/* file stu_fun.c，程序 stu 的处理部分源程序文件 */
#include "stu.h"

void statistics(int n, StuRec tb[]) { ... ... }
static void prtHH(int n) { ... ... }
void histogram(int n, StuRec tb[], int high) { ... ... }
static int scrcmp(const void *vp1, const void *vp2) { ... ... }
void sortoutput(int n, StuRec tb[]) { ... ... }
```

程序中所使用的功能函数定义在另一个源文件里：

```
/* file utilities.c，程序 stu 使用的功能函数源程序文件 */
#include "stu.h"

int next(char s[]) { ... ... }
void getnstr (char prompt[], int lim, char bf[]) { ... ... }
int getcmd (char *prompt, int c1, int cn) { ... ... }
```

分块开发得到的程序结构参见图 10.1。可以看到，在上面整个组织结构里，头文件 stu.h 扮演了一个公共信息通道的角色，所有其他源程序文件都从这里得到所需的类型信息。由于这一信息通道的存在，我们就能保证相互独立的不同文件之间的一致性。如果某个函数的原

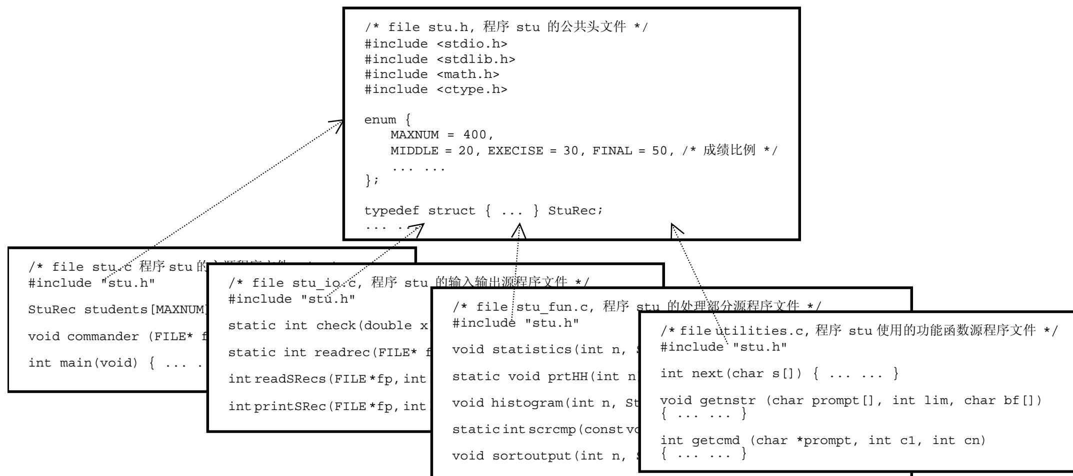

图 10.1　分块开发的程序及其关联结构

型修改了，编译程序就会发现头文件中的函数原型与函数定义不一致。如果修改了相应的函数原型，这一信息立刻就会反应到使用有关函数的源文件里去。

10.1.4 其他安排和考虑

应该看到，完全可能为同一个程序设计不同的物理组织结构，即使采用同样的几个源程序文件和头文件，各个文件的内容分配也可以不同，而同时又都能保证我们程序的语义和程序各部分之间的信息畅通。下面考虑另外一种可能方式。

前面设计中在头文件 stu.h 里包含了程序所需的所有标准头文件，这就使所有源程序文件间接地得到了标准库库功能的各方面信息。在一个大的程序里，可能用到标准库的许多功能，因此可能需要 #include 许多标准头文件。然而，并不是每个源程序文件都需要用到所有这些标准库，一个具体文件可能只用到其中很少一部分，甚至完全没有用到任何标准库功能。采用上面统一包含所有标准库头文件的方式，将使编译程序在编译每个源程序文件时，都需要处理一遍所有包含进来的标准头文件。对于更大的由许多源程序文件构成的系统，这样就可能造成很大的编译时间浪费。

为避免上述问题，我们可以换一种方式，让具体源程序文件包含自己所需的标准头文件。对于上面例子，去掉所有包含命令后的头文件是：

```
/* file stu.h 程序stu的公共头文件 */

enum {
   MAXNUM = 400,
   MIDDLE = 20, EXECISE = 30, FINAL = 50, /* 成绩比例 */
   HISTLEN = 60,        /* 最长行的长度（字符数）*/
   SEGLEN = 5,          /* 分段长度 */
   SEGNUM = 100/SEGLEN+1 /* 分段数，根据分段长度自动算出 */
};

typedef struct {
   unsigned long num;
   char name[20];
   double mid, exe, final, score;
} StuRec;

extern StuRec students[];

int readSRecs(FILE *fp, int limit, StuRec tb[]);
int printSRec(FILE *fp, int limit, StuRec tb[]);
void statistics(int n, StuRec tb[]);
```

```
void histogram(int n, StuRec tb[], int high);
void sortoutput(int n, StuRec tb[]);
int next(char s[]);
void getnstr (char prompt[], int lim, char bf[]);
int getcmd (char *prompt, int c1, int cn);
```

主程序文件现在变成了：

```
/* file stu.c 程序stu的主源程序文件 */ */
#include <stdio.h>
#include "stu.h"

StuRec students[MAXNUM];

void commander (FILE* fp, char *fn) { ... ... }
int main(void) { ... ... }
```

其他文件的情况与此类似，都需要在包含本程序的公共头文件 stu.h 的同时，包含自己所需的标准库头文件。这样做，在写每个具体源文件时可能多费一点事，但随后多次编译加工的效率却可能有所提高。不难看到，虽然各个文件的内容有所调整，这样一组文件仍然组成了一个有机整体，它们仍然满足我们前面提出的原则：同一程序对象的定义点和所有使用点都能参照同一个描述。

另外，如果在一个源文件里同时包含自己的头文件和系统标准库的头文件，人们一般采用的方式是将包含标准库头文件的命令写在前面。

10.1.5 一般性原则

从上面例子可以看到头文件如何成为程序各部分（各程序文件）之间保证信息一致性的桥梁，成为连接程序对象的定义和使用的纽带。定义好头文件是保证较大程序的开发工作能顺利进行的最重要环节。设想几个人共同开发一个大系统，他们之间当然需要有一些约定。如果一个人定义的东西被另一个人使用，就需要通过写出适当头文件的方式建立相互间的联系。因此，在一个程序开发工作中，最早成型的可能是一批头文件，它们形成了不同工作者之间的联系“标准”。如果自己的程序修改了，只要这一修改不影响与别的程序文件共用的头文件，那么这个修改就不会影响程序的其他部分，不会影响其他人的工作。如果自己的工作要求修改头文件，那么就需要与其他人联系，保证对头文件修改的一致性。即使是一个人，在采用分块方式写程序时也有类似情况。

另一方面，为实现一个源程序文件里应提供的功能，我们可能需要定义一些只在这一源文件内部使用的辅助函数。把这些函数定义为静态的，就不必担心这个函数的名字会与其他文件（其他人）定义的东西发生冲突。对于外部变量的情况也一样。从这里，读者也可以看到 C 语言中静态函数和静态机制在开发大程序中的作用。

这里应特别强调。在采用多文件方式开发程序时，许多外部对象（函数、外部变量等）的定义和使用将处在不同文件里。C 语言对许多情况并不强制性地要求做严格检查，如果我们没有提供足够的信息（主要是类型信息），编译程序就会做出一些默认假定，并按照这些假定继续工作下去。只要不发现矛盾，它就不认为程序里有什么错误。在下一个加工步骤中，连接程序只检查程序里需要的东西有没有，如果找到，就把它们连接起来。在连接时不再做任何类型一致性方面的检查。

可见，希望最终能够产生连接正确的程序，关键就在于保证编译过程的正确进行。而要保证编译正确，一个重要问题就是给编译程序提供有关外部程序对象的正确完全的信息。函数原型说明、变量的外部说明、类型定义、系统头文件包含等等，它们的作用都是为编译程序提供信息。写程序时必须认真写好这些东西，这些对于提高工作效率、减少程序中的隐含

错误都有极其重要的作用，也是本书中反复强调的问题。

按照惯例，开发 C 程序的人们通常把一个程序的源文件分成两类，一类是包含实际程序代码的基本程序文件，另一类是为上述基本程序文件提供必要信息的辅助性文件。人们通常约定，基本程序文件以 `.c` 为扩展名，将这种文件称为程序文件或者源代码文件；为其他文件提供信息的文件以 `.h` 为扩展名，称为头文件、`head` 文件，或简称为 `h` 文件。

C语言系统本身的实现也遵循这一方式。一个C语言系统总为我们提供了一组标准库头文件，还可能提供一些服务于特定系统（如DOS、Windows、UNIX等）的扩充头文件。这些头文件的作用就是为在C程序里使用标准库函数以及其他功能提供必要的信息。如果需要在程序里使用某些库函数，只要我们在源文件前面包含了必要的头文件，就能保证在编译过程能对源文件中有关函数调用正确进行处理[*]。

物理组织的合理原则

如果决定采用多个源文件的方式实现一个程序，随后应该怎样做？应该把程序中哪些东西放在头文件里，哪些放在程序文件里？关于这些，C 语言并没有做任何规定，人们在长期编程实践提出了一些合理方式。下面介绍的是作者认为比较合理的一套方式。

如果一个程序的源程序由多个源文件组成，其中一些是头文件，一些是程序文件。头文件和程序文件的内容安排应该遵循下面规则：

1. 头文件里只写不实际生成代码、不导致实际存储分配的描述。这里可以有：包含标准库头文件和其他头文件的预处理命令；各种公用宏定义（多个程序文件都使用的公共东西，尽量少用宏定义）；各种公共的类型定义；结构、联合、枚举的说明（仅仅是说明）；函数原型说明；变量的外部说明（关键字 `extern` 引导的说明，而不是变量定义，仅说明有某类型的外部变量在其他地方有定义）。进一步的建议是把与多个程序文件有关的结构等都定义为类型）。这里不写外部变量定义和函数定义。
2. 在各个程序文件里分别定义所有的外部变量和函数，并写出那些只在一个文件中局部使用的东西（如局部使用的类型，宏定义等等）。
3. 只用文件包含命令（`#include`）包含头文件，而不用它包含程序文件。
4. 通过头文件解决在一个程序文件里定义而在另一个程序文件里使用的信息传递问题。首先把与此有关的函数的完整原型、外部变量的完整外部说明写在某个头文件里。对于所有定义和使用这些函数或变量的程序文件，令它们都包含这同一个头文件。通过这种方式保证使用和定义之间的联系，保证编译程序能进行一致性检查。程序加工最后的连接步骤将保证对在其他文件里定义的函数的调用、对在其他文件里定义的外部变量的使用都能够实现。这里的原则仍然是，让定义和使用同一个程序对象（变量、函数、类型等等）的地方，都能看到同一个说明。

程序物理组织的具体考虑

面对一个具体程序开发问题时，我们应该怎样处理程序的物理结构组织问题呢？这里只能提出一些工作建议，最基本的原则当然是具体情况具体分析。大致工作步骤是：

1. 程序组织的第一步是估计程序的大小，根据这个估计考虑源文件应划分为几块。在程序开发过程中，这种初始划分也可能需要调整。例如发现一个文件膨胀得很大，就应当考虑是否应将它划分为两个或几个部分。这时又会出现如何划分和有关信息的组织管理问题，这时仍应该坚持这里提出的各项原则。
2. 考虑把程序所需的功能划分为若干部分，每部分中定义的东西互相之间应有较密切的逻

[*] 库函数的实际代码存在另外的库文件里。在对目标程序连接时，连接程序将从特定库文件里取出有关代码段，拼装到最终的可执行程序里，形成完整的程序。这是目前大部分C语言系统的通行实现方式。

辑联系，如提供类似功能，完成同类的工作，相互之间调用关系密切等。这样形成一种整体性，可以考虑放在一起建立一个程序文件。例如，输入输出有关的功能可考虑放在一起，如果输入和输出都很复杂，也可以考虑为它们各建立一个程序文件。主函数通常单独建立一个文件，其中也可以包含少数与之关系密切的其他函数定义。

3. 对于全局变量，应根据谁用谁管的归属原则，分别在不同的源文件里定义。程序中公共的全局变量（许多地方都要用的那些变量）一般在主程序文件里定义。
4. 建立一个或几个头文件。把所有公用的类型定义、公用的结构联合和枚举说明、公用的宏定义放在适当的头文件里，作为各个文件参考的依据。对标准头文件的使用，可以在具体的程序文件中用 `#include` 命令做文件包含。如果在许多地方都使用同一个标准头文件，或者某个头文件本身需要，也可以把对标准头文件的 `#include` 命令写在某头文件前面。有的头文件里还可能包含其他头文件，例如某个类型在一个头文件里定义，可能在另一个头文件里要使用，等等。
5. 对于所有在一个源程序文件里定义而在其他文件中使用的东西，都需要在某一个头文件里有说明（函数原型说明，或者变量的外部说明）。每个说明都应该建立起两个方向的联系：既与其定义所在的文件相联系，保证编译系统能进行定义和说明间的一致性检查；又与所有使用位置相联系，保证编译系统正确处理这些使用。建立联系的方式就是在程序文件里包含有关的头文件。
6. 将所有局部于一个源程序文件的东西都写在这个文件里。局部使用的外部变量和辅助函数应定义为静态的（在定义前加 `static` 关键字）。每个文件前部用 `#include` 命令包含必要的头文件（系统的或自己创建的），不用的东西尽量不包含进来。
7. 如果程序比较大，也完全可能需要为一组实现某些功能的源程序文件定义相应的头文件，而将它们所实现的功能通过另一个描述界面的头文件提供给其他使用这些功能的程序部分。

上面所说的只是一些一般性原则，在遇到具体问题时，必须要活学活用。

集成开发环境中的程序组织和开发

今天许多程序开发工作是在*集成程序开发环境*（IDE）中进行的，这类环境都对程序的分块开发提供了特别支持。集成开发环境里通常都提供了“项目”（Project）的概念和项目管理的一整套功能。一个项目就是一个（待）开发的程序，它通常由若干源程序文件和若干头文件实现。创建一个项目一般过程是：

1. 建立一个项目（文件），这个文件就代表了待开发的程序。
2. 一些 IDE 支持多个不同种类的程序开发，例如控制台应用程序（console application，即一般的以字符行形式与人交互的应用程序。本书的程序实例均属此类），具有图形用户界面的应用程序（依赖于具体开发所在的系统平台，如 Windows 系统下的开发，Unix 系统下的开发。本书并不打算讨论这类依赖于具体系统的开发）等等。这种情况下就需要选择适当的项目类型。对于目前的学习而言，总应该选择控制台应用项目。
3. 将所需的 C 源程序文件加入这一项目中（许多 IDE 提供了专门命令或者其他方式）。通常不必将头文件加入项目，IDE 可以根据项目中源文件里的包含命令确定哪些头文件与本项目有关。

在这样建立了项目之后，开发的工作对象就是这个项目了。随着开发工作的进展，我们还可以根据需要向项目里加入新的程序文件，或者调整删除项目里已有的程序文件。

举例说，在某个 IDE 环境中开发前面的学生成绩处理系统，我们可以创建一个项目，例如将它命名为 students。将准备创建的几个 C 源程序文件，包括 stu.c，stu_io.c，stu_fun.c 和 utilities.c 都加入这个项目里。而后就可以开始编辑这些文件了。一般不必将 stu.h 列进项

目里，但也应在同一个目录里创建这个文件，编辑其内容。

集成开发环境能帮人处理分块开发中的许多问题，主要是检查和利用源文件之间的依赖性。例如，某几个源程序文件可能对一个头文件有依赖关系（它们 `#include` 这个头文件），如果该头文件的内容修改了，所有依赖于它的源程序文件都必须重新编译后才能进行连接，这样才能保证结果程序的有效性。集成开发环境能帮助编程者检查处理这方面情况，在进行连接前重新编译必要的文件，保证所生成的可执行程序总能反映整个项目的最新修改情况，等等。这些功能都能减轻程序开发过程中人的负担。

IDE 通常提供了多个性质不同的程序加工命令，它们的功能有所不同。常见的有：

1. 编译（compile）：编译命令通常只针对 IDE 里的当前源程序文件，检查这个文件的语法结构，在没有语法错误的情况下将产生出与之对应的目标文件。在编辑源程序文件的过程中，常常需要不时通过编译检查代码中的语法错误。
2. 连接（link）：工作对象是整个项目，将属于这个项目的目标文件连为一体，形成一个完整的可执行程序文件。有些 IDE 在连接时会检查现存的目标文件是否都为最新的，在发现情况时给出提示信息。
3. 构造（make）：这是 IDE 中最重要功能，其工作目标就是创建出一个最新版本的可执行程序。构造命令的工作对象是整个项目，它将检查程序依赖关系，确定是否所有目标文件都已经存在，是否都为最新版本。发现缺少目标文件或有目标文件陈旧时调用编译程序生成有关目标文件，而后调用连接程序生成最新版本的可执行程序。如果现存的可执行程序已经是最新版本了，那么它就什么也不做。
4. 全部重做（rebuild all）：重新编译属于本项目的所有源程序文件，最后连接所得到的目标文件，生成新的可执行程序。

在使用具体的集成程序开发环境完成程序设计时，应设法了解它们的具体使用方式。这里的介绍反应了各种 IDE 的一般性情况，可供参考。

10.1.6 单一头文件结构和多个头文件结构

前面实例程序的物理组织方式可称为单一头文件程序结构。对于相对较简单的程序，人们常采用这种组织形式，将程序中所有需要交换的信息都存入一个头文件，并让每个程序代码文件都包含这个头文件，这种源文件组织形式如图 10.2 所示。在这种方式中，头文件起着一种公共信息中心的作用。可以把所有标准库包含命令写在头文件里，有时也把具体程序文件所需标准库功能的包含命令直接写在该程序文件最前面。

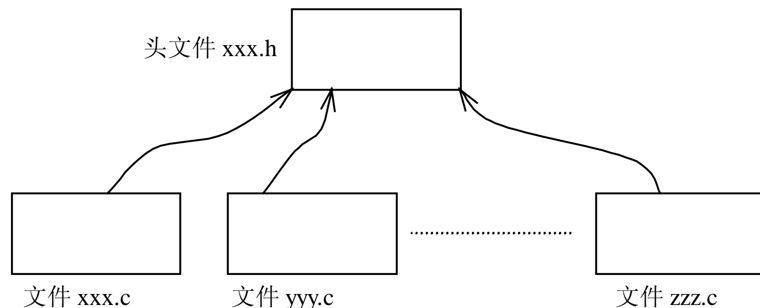

图 10.2 单一头文件的物理结构

如果程序更复杂些，单一头文件方式的缺陷就会变得明显起来。由于程序里只有一个信息中心，如果我们需要修改文件 yyy.c 中的某个函数 `f` 定义的头部，而文件 xxx.c 里使用了 `f`，那么 `f` 的原型一定已经写在文件 xxx.h 里，需要修改以便与 yyy.c 中的定义保持一致。由于所有源程序文件都依赖于头文件 xxx.h，因此它们都受到了影响，至少是都需要重新编译。重编译一个大系统的所有源程序可能需要几十分钟甚至几个小时，而原本这一修改只影响到 xxx.c 和 yyy.c 两个文件。这种情况不仅可能造成不同程序部分间并无必要的紧密联系，还带来了程序代码间出现更多相互干扰的可能性（例如，在头文件中定义的宏的作用将传播

到所有源程序文件中，有时这可能并不是我们所希望的）。

为了避免上述情况，对于更大的系统，人们通常采用多个头文件的物理结构，其形式如图 10.3 所示。程序中根据情况建立了若干个头文件，分别记录了一些需要在不同源程序文件之间交流的信息。各个程序文件根据需要，包含其中的一个或者几个头文件。这样形成一种分布式的信息记录，可以使程序文件间的影响局部化，有利于大程序的开发。

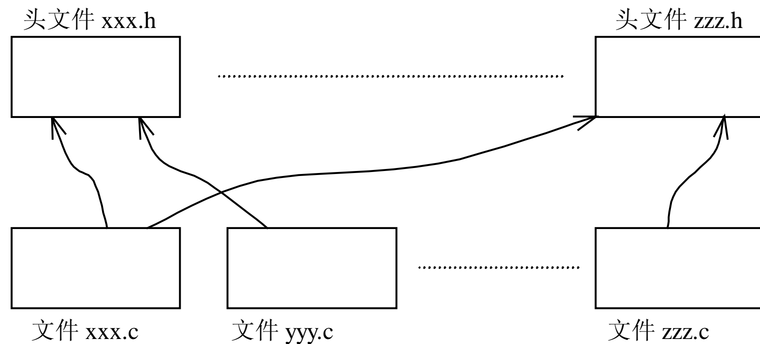

图 10.3 多个头文件的物理结构

举例来说，在前面学生成绩处理的例子里，源文件 utilities.c 里定义了一些常用的功能函数，这些函数只在源文件 stu.c 和 stu_io.c 里使用。进一步说，utilities.c 中的程序代码根本不依赖于其他程序模块的任何信息（并不依赖于 stu.h 里提供的任何信息），它们完全是独立的。一种合理方式是专门创建一个头文件 utilities.h，作为 utilities.c 与使用该文件提供的功能的程序文件之间的信息通道。下面是 utilities.h：

```
/* utilities.h, 功能函数头文件 */
#include <stdio.h>
#include <ctype.h>

int next(char s[]);
void getnstr (char prompt[], int lim, char bf[]);
int getcmd (char *prompt, int c1, int cn);
```

为保证 utilities.c 中的定义与头文件中的信息一致，这个文件应写成下面样子：

```
/* utilities.c, 功能函数源程序文件 */
#include "utilities.h"

int next(char s[]) { ... ... }
void getnstr (char prompt[], int lim, char bf[]) { ... ... }
int getcmd (char *prompt, int c1, int cn) { ... ... }
```

头文件 stu.h 修改为：

```
/* file stu.h 程序stu的公共头文件 */
enum {
   MAXNUM = 400,
   MIDDLE = 20, EXECISE = 30, FINAL = 50, /* 成绩比例 */
   HISTLEN = 60,        /* 最长行的长度（字符数）*/
   SEGLEN = 5,          /* 分段长度 */
   SEGNUM = 100/SEGLEN+1 /* 分段数，根据分段长度自动算出 */
};

typedef struct {
   unsigned long num;
   char name[20];
   double mid, exe, final, score;
} StuRec;

extern StuRec students[];

int readSRecs(FILE *fp, int limit, StuRec tb[]);
int printSRec(FILE *fp, int limit, StuRec tb[]);
void statistics(int n, StuRec tb[]);
```

```
void histogram(int n, StuRec tb[], int high);
void sortoutput(int n, StuRec tb[]);
```

而主程序文件 stu.c 修改为：

```
/* file stu.c 程序stu的主源程序文件 */ */
#include <stdio.h>
#include "stu.h"
#include "utilities.h"

StuRec students[MAXNUM];

void commander (FILE* fp, char *fn) { ... ... }
int main(void) { ... ... }
```

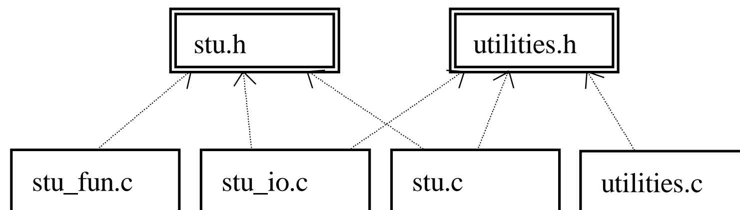

图 10.4 新物理组织方式下的程序文件依赖关系

这样修改后形成的程序结构如图 10.4 所示。此时如果我们修改文件 utilities.c 中函数定义并需要修改 utilities.h，只有文件 stu.c 受到影响。在这一图示中，我们没有画出各个程序文件对于标准库头文件的依赖关系。更细节的讨论可能还需要考虑这方面的关系。

一种极端方式是为每个提供某种服务的源文件建立一个相关的头文件，让需要使用该种服务的程序文件包含这个头文件。程序进一步复杂时，还可以考虑更复杂的结构，例如，有时几个头文件之间存在共享信息，这些信息可以通过一个公共的头文件提供。

对于我们的具体实例，可以为 stu_io.c 建立头文件 stu_io.h：

```
#include <stdio.h>

int readSRecs(FILE *fp, int limit, StuRec tb[]);
int printSRec(FILE *fp, int limit, StuRec tb[]);
```

为 stu_fun.c 建立头文件 stu_fun.h：

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

void statistics(int n, StuRec tb[]);
void histogram(int n, StuRec tb[], int high);
```

让这两个源程序文件分别包含自己的头文件和 stu.h，让主文件也包含这两个新的头文件。这样就形成了另一种结构，也能满足程序正确编译的需要。

应注意，这些讨论并不是说多头文件结构优于单一头文件结构，也不是说物理结构划分得越细越好。做一切事情都有个限度。对于较小的程序，引入许多头文件，采用复杂的物理组织方式不会带来任何实际利益。例如，在实际开发时，对于上面这样规模的小型程序，完全没必要为每个源程序文件建立一个头文件。上面的讨论只是想展示各种可能方式。

10.1.7 功能模块和程序库

在进一步探究程序开发方法时，我们迟早会想到程序的功能模块和库的问题。举例说，上面源文件 utilities.c 所提供的功能可能用到许多其他交互式程序里。采用在需要时将它们的源代码拷贝到其他程序里的方式不是好方法。这种做法不利于模块的版本更新（如果某些功能有了新实现，你怎样确定需要把它们拷贝到哪些文件里？），重复拷贝源代码也很容易

引进错误。实际上我们已经看到了正确的做法：

1. 为需要在一些程序里使用的功能建立一个源程序文件和一个与之对应的头文件；
2. 在源程序文件里实现所需的功能，包括：定义提供给外面使用的函数，为实现这些函数可能需要的辅助函数（应该定义为 static 函数），为实现有关功能可能需要定义的一些外部变量（根据需要考虑是否定义为 static 变量）；
3. 在对应头文件里给出有关类型定义和提供给外部使用的函数原型，或许还包括一些全局的变量说明（外部说明）。

这样，当某个程序里需要使用这种功能时，只需在相应文件前部包含这个头文件，并将上述源程序文件作为本项目的源文件，编译连接后就可以使用其功能了。

复数模块

为了使论述更具体些，我们以前一章设计的复数功能作为例子，展示有关的做法。

假定我们希望完成一个复数模块，使其他需要使用复数类型的运算的程序很容易利用这一模块提供的功能。根据上面的讨论，需要做的事情已经很清楚了：应该将复数模块所提供的功能在一个“界面”头文件里描述，用一个（或者几个）程序文件实现这些功能。这里所说的界面，也就是其他使用复数功能的程序能“看到”的所有信息。

由于复数功能的实现比较简单，下面考虑用一个头文件和一个程序文件，分别将它们命名为 complex.h 和 complex.c。为了使用方便，这里还考虑定义几个复数“常量”。这样，头文件 complex.h 可以是下面的样子（具体设计可以根据实际需要）：

```
/* 文件 complex.h */

/* 类型定义 */
typedef struct {
   double re, im;
} Complex;

/* 常量说明 */
extern const Complex Complex0;
extern const Complex Complex1;
extern const Complex ComplexI;

/* 构造函数 */
Complex mkComplex(double re, double im);
Complex d2Complex(double d);
Complex n2Complex(int n);

/* 运算函数 */
Complex addComplex(Complex x, Complex y);
Complex subComplex(Complex x, Complex y);
Complex tmsComplex(Complex x, Complex y);
Complex divComplex(Complex x, Complex y);
/* ... ... 其他函数原型 */

/* 比较函数 */
int eqComplex(Complex x, Complex y);

/* 输入输出函数 */
int readComplex(FILE *fp, Complex *xp);
void prtComplex(FILE *fp, Complex x);
```

在程序文件 complex.c 里定义所有的函数和上面所说的“常量”：

```
/* 文件 complex.c */

#include <stdio.h>
```

```
#include "compl.h"

/* 常量定义 */
const Complex Complex0 = {0, 0};
const Complex Complex1 = {1, 0};
const Complex ComplexI = {0, 1};

/* 构造函数 */
Complex mkComplex(double re, double im) { ...... }
Complex d2Complex(double d) { ...... }
Complex n2Complex(int n) { ...... }

/* 运算函数 */
Complex addComplex(Complex x, Complex y) { ...... }
Complex subComplex(Complex x, Complex y) { ...... }
Complex tmsComplex(Complex x, Complex y) { ...... }
Complex divComplex(Complex x, Complex y) { ...... }
/* 其他运算函数的定义 */

/* 比较函数 */
int eqComplex(Complex x, Complex y) { ...... }

/* 输入输出函数 */
int readComplex(FILE *fp, Complex *xp) { ...... }
void prtComplex(FILE *fp, Complex x) { ...... }
```

在这个模块里提供哪些函数是一个设计问题，可以根据需要考虑。这些函数的实现不难完成，前面也讨论过其中的一些问题。这里就不再重复了。

上面两个文件也清晰地贯彻了我们前面所说的头文件和程序文件的描述分配原则。如果某个程序里需要使用复数类型及其操作，那么就只需将 complex.c 加入其开发项目，需要使用复数功能的程序文件前都 `#include` 头文件 complex.h，就可以使用这里定义的功能了。因为我们的 complex.c 文件也是参照着头文件 complex.h 加工的，因此，这个头文件就成了维系 complex.c 与其未来使用者之间一致性的纽带。

C 程序模块的这种构造方式带来了非常有价值的性质：封装性。以上面的复数模块为例，如果某系统里使用了这个模块，程序里所有使用复数的地方都严格地只经过 complex.h 描述的界面使用复数功能，那么从系统的角度看，有关复数模块的具体实现现在已变得无关紧要了：只要它所提供地操作实现了所要求的功能，这个系统就能正常工作。即使实现模块的人们（由于某种原因）修改了模版的具体实现方式，只要这些修改没改变 complex.h 中各种函数的原型，系统的其他部分就完全不需要修改。

不难看出，这种封装与前面讨论的函数封装有许多类似之处。这里的模块头文件（例如 complex.h）扮演着与函数原型类似的作用，它描述了模块的使用界面。而模块功能的具体实现被封装在这一界面的后面。我们就像是在使用一种抽象的复数类型。

实践已经证明，这种封装性是所有复杂软件实现的基础。这样封装起的一个模块提供了一种抽象功能，上面的 complex 模块提供的似乎是一种抽象的复数，从使用者角度看，其具体实现是“不可见的”，或者说是不必关心也不需要关心的。这种方式产生了许多重要的效果。首先，一个这样的封装立刻把一个复杂的程序分解为两个界限清晰的世界：这种抽象功能的使用方和这种功能的定义方（与函数分解的情况类似，但表现在另一个层次上），这样就分解了程序的复杂性。这种分解形成了一种屏障，只要界面的两边维持原有界面约定不变，任何一边都可以独立地修改变化。

这种模块封装（包括像复数模块这样的数据类型封装）可用于形成大程序的另一层分解，是设计实现大型程序，处理大程序的复杂性的最重要技术。读者在今后的学习和时间中，将会更深刻地理解这些技术的重要性。

由于 C 语言是 1970 年代开发的语言，那时人们对于数据类型封装和模块封装的认识还很初步。因此，在 C 语言里完成这些构造，需要借助于一些编程约定和预处理功能（如上所述），语言本身对此的支持不够。后来的一些新语言都为大型程序构造的这方面需要专门提供了语言层的特征，例如 Ada 语言的模块概念，各种面向对象语言（如 C++和 Java 等）的类概念等等。读者在今后的学习和实践中将会进一步看到它们的重要性。

目标文件和库

仔细想想不难发现，一旦我们将 complex.c 做好，那些使用这个文件中所定义的功能的程序其实并不需要再去看 complex.c 的内容，只需要编译它并将编译结果连接到最终的可执行程序里。这样看来，有时提供 complex.c 的源文件并无必要。提供了源文件还造成做系统时不小心修改了这个文件，造成错误的可能性。

此外，还有一些情况下我们不希望将自己的源程序文件提供给别人。有时是为了某种安全性，因为拿到这一源文件的人就可以修改它，设法去访问修改其中一些不应该访问修改的东西。另外，有些源程序文件里包含着不希望别人了解的技术秘密。许多因素造成人们可能希望为其他人提供某种功能模块，但却不希望将源程序文件提供给别人。

上面的模块框架也使这种想法成为可能。假设我们不希望将 complex.c 提供给别人，那么就可以只提供 complex.h 头文件，再提供一个由 complex.c 编译后生成的目标文件。别人拿到这两个文件，就可以使用我们的复数模块所提供的功能了。为此他们只要：

1. 在编写程序时，让那些必要的源程序文件包含头文件 complex.h；
2. 在连接时将我们提供的实现复数功能的目标文件也连接到可执行程序里。

从这里可以看到标准库和其他程序库的影子。从某种意义上说，上面的 complex.h 和相应目标文件形成了一个“复数功能库”，甚至它的使用方式也与使用标准库类似。

这里还需要提一下，使用目标文件和使用库之间还有一点细微差异。通常，如果要求连接程序将一个目标文件连接到程序里，这个目标文件里的所有东西都会被放入最终的可执行程序。如果我们要求连接程序连接一个库文件，它会根据程序里的需要，将程序里实际需要的库代码找出来，只把这些代码连接到可执行程序里。

想想标准库的情况，就可以看到这种工作方式的必要性。许多 C 系统将标准库的所有代码放在一个库文件里，这个文件通常很大，因为标准库包含许多功能。我们当然不希望每个程序的可执行文件都包含所有标准库代码，因为其中往往只用到不多的标准库功能。

各种 C 语言系统通常都提供了创建库文件的功能，例如提供可将目标文件转换为库文件的工具，或提供直接编译产生库文件的工具。具体情况依赖于具体系统，这里不讨论了。

10.1.8 防止重复包含

在一个程序里定义使用的头文件可能有许多个，有的头文件里还要包含其他头文件，同一个程序文件也可能包含多个头文件，这样就有可能引起同一个头文件的重复包含问题。有时重复包含会导致编译过程不能正常完成，使后面的步骤无法进行。为此人们提出了防止重复包含的方法。人们最常用的方法很简单，是通过几个预处理命令实现的。

假设要定义一个头文件，它有可能被重复包含，这个头文件可以写成下面样子：

```
#if !defined(MY_HEAD_FILE1)
#define MY_HEAD_FILE1 1
... ... /* 这里写头文件原本要写的所有内容 */
#endif
```

这里的 MY_HEAD_FILE1 是为防止重复包含专门定义的宏名字，可以用自己选择的任何名

字。此后，如果这个文件被包含，预处理程序处理文件开始时，名字 MY_HEAD_FILE1 就有了定义。如果同一个编译单位的预处理中再次遇到这个文件（该文件又被包含），由于文件开始写出的条件不成立了，预处理程序就会把由#if 到对应#endif 之间的所有东西都丢掉，正好符合我们不希望多次包含的初衷。实际上，系统的所有头文件一般都采用了这套定义模式，以防我们写程序时由于重复包含而出现问题。

10.2 错误报告和处理

程序执行中常常遇到需要处理的错误情况，因此程序的设计和实现就需要考虑这方面的问题。本节讨论一些有关情况和可能的处理方法。

10.2.1 建立统一的错误报告机制

不难看出，在我们前面的实例里，实现错误报告等等的输出语句遍布整个程序，都是通过直接调用标准库函数 printf 实现的。也正因为此，许多源程序文件都需要包含标准库头文件<stdio.h>。这种做法有不少缺点：

1. 散布各处的错误报告串缺乏相互联系，缺乏统一模式，很难维护和管理；
2. 在程序里直接通过调用库函数报告错误，不利于修改错误信息报告方式，无论是传递错误信息的目标，或者错误报告的形式；
3. 程序错误报告使程序中许多部分都与标准库的具体功能建立了密切联系。

下面考虑通过一些方式缓解这些问题。

我们首先希望松弛程序中各个部分与具体标准库函数的联系，可采用的一种方法是定义一个自己的错误消息函数，程序里需要输出错误信息时都调用这个函数。这样，如果需要修改错误信息的定向或者表现形式等，那么就只需要修改这一函数。

首先总结一下程序中需要发出的各种消息。在程序里可以看到如下一些语句：

```
printf("Can't open file: %s\n", fn);
printf("Data too few. Statistics stop.\n");
printf("Data error, line %d\n", line);
```

等等。这些输出语句用一个格式串作为实际输出消息的框架，另有 0 个或几个参数。这些语句中利用 printf 的格式化功能生成实际的输出消息串。写好一个能满足这样多种需要的消息函数不是一件很简单的事情，下面讨论这个问题。

最简单的方式是简化输出形式，采用下面的简单消息函数：

```
void emessage(const char *estr) {
    fflush(stdout);
    fprintf(stderr, "Error: %s\n", estr);
}
```

由于对 stderr 流的输出采用非缓存方式，送入这个流的信息将立即显示在显式设备上（例如屏幕或者特定窗口中）。为防止标准输出流 stdout 缓冲的信息与这样输出的错误信息产生交错，上面函数里首先刷新 stdout，将该流中缓冲的信息实际输出到指定目标，而后再调用 fprintf 输出错误信息。

我们可以将函数 emessage 放入文件 utilities.c，而后将程序里原来用 printf 输出错误信息的语句都改为调用函数 emessage。这样做之后，程序中的错误信息就都统一到的一条通道中，比较容易统一地维护了。

10.2.2 定义变参数的错误报告函数

上面定义的简单错误报告函数有时不能令人满意，因为它只能输出简单的错误信息串，

我们不能通过它传递具体出错位置的其他有用信息。例如，原来错误消息语句：

```
printf("Can't open file: %s\n", fn);
```

不仅告诉用户出现了文件无法打开的情况，还将显示出导致文件打开操作失败的文件名，这种信息对于用户了解程序运行情况，确定下一步的工作方式等等都是非常有价值的。

上面的具体问题很容易解决，例如可以将前面的 emessage 函数修改如下：

```
void emessage(const char *estr, const char *fn) {
    fflush(stdout);
    fprintf(stderr, "Error: %s: fn\n", estr, fn);
}
```

但是这个新版本的使用太受限制了：它实际假定了除错误消息串之外需要输出的是另一个字符串，而且只有一个字符串。但是实际需要输出的信息可能不同，例如：

```
printf("Can't open file: %s\n", fn);
printf("Data too few. Statistics stop.\n");
printf("Data error, line %d\n", line);
```

这里需要输出的信息是多种多样的，不但可能具有不同类型，甚至参数的个数也可能不同。利用我们已经了解的函数定义机制，无法写出一个统一的函数完成这些工作。

总结一下我们的需要：希望定义函数 emessage，使之能完成各种错误信息的生成工作。为此它需要有类似 printf 的参数形式和功能：可以用一个"格式串"描述输出消息的框架，并将另一些实际表达式的值嵌入其中，形成最终的消息串；这种表达式的个数和类型都可以根据需要变化。此外，我们还希望用 emessage 将错误消息的输出目标和基本形式隐蔽起来，以便于统一管理和修改。这一目标将要求 emessage 具有与 printf 同样的原型，而怎样定义参数类型和个数可以变化的函数（变参数函数）是前面没有讨论的问题。

要定义变参数的函数，需要使用标准库<stdarg.h>中定义的功能，有关情况在第 11.7 节有详细讨论，请读者查阅。目前的 emessage 采用如下原型：

```
void emessage(const char *estr, ...);
```

也就是说，这个函数有一个字符串参数（作为格式描述串），而后可以有任意多的其他参数，这些参数的类型也没有规定（需要靠格式串里的描述去确定）。

在变参数函数里，需要借助于<stdarg.h>里定义的类型 valist 和其他相关功能，取得和使用由函数头部参数表里的 ... 代表的那些实际参数。对于函数 emessage，函数定义的基本框架是：

```
void emessage(const char *estr, ...) {
    valist vl; /* 如果需要其他定义，同样写在这里 */
    va_start(vl, estr);
    ... ... /* 从这里开始就可以利用va_arg宏顺序取得estr后面的各个实参了 */
    /* 实际输出信息的代码写在这里 */
    va_end(vl);
}
```

这里的 va_start 初始化 va_list 变量 vl，使我们可以在下面通过它去提取 estr 后面的各个实参。va_start 一般用宏实现，其第一个参数是需要初始化的 va_list 变量，第二个参数是函数的最后一个有名字的参数（可见这种函数至少应该有一个命名参数）。最后的 va_end 做一些清理工作，所有使用了 va_list 变量的函数结束前都必须调用 va_end，使程序恢复到可以正常执行的状态。

现在考虑实际信息的生成问题。我们完全可以自己处理字符串 estr，逐步输出其内容，并在适当的时候将其他参数的值插入其间。这样写出 emessage 的定义并没有本质性的困难，关心具体写法的读者可以自己考虑，可以参考《C 程序设计语言》中有关一个简化版本的 printf 实现的讨论，这里不仔细讨论了。我们想在这里顺便介绍另一组输出函数，借助于它们可以简化实现。这组函数的介绍参看第 11.8 节，它们的功能与普通 printf、fprintf 和 sprintf 一族函数完全一样，只是其参数表里最后的参数不是 ...，而是一

个 va_list。函数执行时将这个 va_list 参数代表的一组实参格式化输出。

借助于其中的函数 vfprintf，错误消息函数 emessage 的实现就非常简单了：

```
void emessage(const char *estr, ...) {
    valist vl;
    va_start(vl, estr);
    fflush(stdout);
    fprintf(stderr, "\nError: ");
    vfprintf(stderr, estr, vl);
    fputc('\n', stderr);
    va_end(vl);
}
```

对于上面消息实例，现在都可以直接改为：

```
emessage ("Can't open file: %s\n", fn);
emessage ("Data too few. Statistics stop.\n");
emessage ("Data error, line %d\n", line);
```

我们在这里借用了 printf 函数族的格式化功能，这使函数定义大大简化。如果需要，我们也可以自己定义格式串功能。

如果程序需要做的不是将错误信息直接送到流 stderr，那么可以利用 vsprintf，将格式化结果存入一个字符数组，而后就可以送到任何需要这种信息的地方。举例说，在图形用户界面的程序里，常需要将消息送到特定窗口去，那时就可以采用上述方式。

10.2.3 运行中错误的检查和处理

程序在执行中常常会遇到出现错误的情况，这种错误可能来自程序的外部（例如，由于人的输入错误），或者出自程序不同部分之间的信息交流（例如，某函数发现调用时的实参值不合法；或者某个需要用的全局变量的值不合要求）。现在需要考虑的是，写程序时应该怎样考虑和处理这方面的问题。

两种有问题的处理方法

我们用一个极其简单的例子讨论这个问题。假定程序里需要一个从标准输入流读入一个整数，并通过指针参数间接赋值的函数。下面定义提供了这一函数的基本功能：

```
void getnum(int *np) {
    scanf("%d", np);
}
```

有了这个定义之后，如果我们想从标准输入读一个数存入变量 n，而后按照这个数的值分配一块可见，其中能存放 n 个整数，就可以写：

```
int n, *p;
getnum(&n);
p = (double*)malloc(n * sizeof(int));
```

上述定义和程序中都没有考虑读入数据可能出错的情况。如果读入数的操作真的出了错，变量 n 可能没有得到合法值，随后的存储分配就完全没有保证了。

我们常看到一些书籍的程序实例里检查可能错误，在发现错误时输出了错误信息，而后却又若无其事地运行下去了。对于上面例子，相应做法就是将函数定义改为：

```
void getnum(int *np) {
    if (scanf("%d", np) != 1)
        printf("Can't read a number!\n");
}
```

将这一定义放在上面调用环境里，出错时程序确实输出了信息。但是另一方面，这时程序已经进入了错误状态（变量 n 的值不合法），继续下去可能产生很危险的结果。因此这种做法

是掩耳盗铃，已经发现了错误却不处理。实际程序里绝不能这样做。

我们常常还能看到另一种处理方式：在发现错误的情况下调用标准库函数 exit。exit 在标准头文件<stdlib.h>里说明（参看第 11 章），该函数的执行将导致**本程序**"正常结束"。所谓"正常结束"是指程序在结束之前完成所有必要的清理，包括刷新所有的输出流，将流缓冲区里的信息实际输出，关闭文件等等。按照这种方式，上述程序将被改写为：

```
void getnum(int *np) {
    if (scanf("%d", np) != 1) {
        printf("Can't read a number!\n");
        exit(1);
    }
}
```

这里还在结束程序前输出了一条信息。

对于简单的小程序，上面函数定义或许还可以考虑。然而这种函数根本不能用到任何真实的程序里。举例来说，如果某个文字处理系统里用了这种读数功能，在程序运行中，用户在需要提供数时不当心敲错了一个键，该系统就输出一条错误信息后立即退出执行，几个小时的工作结果再也找不到了。不会有人敢使用这种系统。

getnum 是一个底层函数，为程序上层功能的实现提供某种服务。这种底层函数在工作中有可能检查出某种错误状态，但是从原则上说，由于这种函数不掌握全局情况，没有关于自己被调用的环境信息，它们的设计实现里绝不能包含任何可能危害全局系统的事情。从这个角度说，上面的函数实现是一种典型错误，这种函数决不能用在实际系统里。

两种常见处理方案

关于底层功能的设计有一个原则：应该根据需要尽可能地检查错误。如果遇到的错误无法合理地在局部解决，就应该设法报告错误，要求调用函数的代码段去处理。就像企业里的一般职员，在发现不能局部解决的问题时应该向上级报告，因为其上级了解更大范围的情况，他们有可能从全局出发，找到更合理的处理方式。

标准库函数的实现始终贯彻了这种原则，值得我们在编程中效仿。标准库函数的错误处理主要采用了两种方式：第一种方式我们早已见识了，这种方式广泛出现在输入输出函数中，这就是通过返回特殊的函数值，向调用位置报告遇到本函数执行中遇到错误情况，说明所要求的工作没有正常完成。

可以看到，所有返回整数值的函数（例如 getchar，putchar，scanf，printf 等等）正常完成时都返回非负值（有时用这个值表示某种信息，例如 scanf 和 printf，或者就是实际结果，如 getchar），出错时返回预定义常量 EOF，这是一个负值。当然，一些函数在遇到文件结束时也返回 EOF 值，但标准库提供了函数 feof 和 ferror，利用它们可以检查究竟是遇到文件结束，还是出现了错误。

标准库的另一些函数返回某种指针值（例如 fopen，fgets，fputs 等等），它们都用返回空指针值 NULL 表示出现了问题。这时同样可以通过上述函数检查实际情况。

前面的许多程序实例中都包含了对程序运行中出现错误情况的处理，所定义的许多函数都采用返回值通告执行情况。采用这种方式的一个麻烦是"错误返回值"的选择，也就是说，应该选用什么返回值表示遇到了错误。选择错误返回值也是一个设计问题，有时也会出现无法找到合理的错误值的情况。

标准库里广泛采用的另一种方式是通过特定的"错误变量"传递信息，这种方式被用在标准数学函数中。标准库的数学函数大都具有类似 sin 的原型：

```
double sin(double);
```

因此设计者无法为这组函数选择一个"错误返回值"，因为每个 double 值都可能是某个数

学函数的返回值。另一方面，这些数学函数常常出现在更复杂的表达式内部，也使我们没有机会去检查每次调用的返回值。

标准库对于这种情况的处理办法是定义了一个表达式 errno（查看第 11.1 节）。如果程序执行中没有出现错误，这个表达式的值将是 0。如果出现了错误，其值就会非 0。C 语言标准并没有规定所有需要检查的错误，具体 C 语言系统可以自己确定。但标准规定至少数学函数应检查参数出错（例如对负数使用 log 和 sqrt）和返回值出错的情况（例如返回值溢出），并适当设置 errno 的值。程序了可以通过检查 errno 值的方式确定计算中是否出现过错误。标准库有几个预定义常量，可供程序里用于检查具体错误的情况。此外还有一个函数 perror，可以用于产生与当前错误有关的错误信息。我们可以在概念上将 errno 看作一个变量，其具体实现由 C 系统确定，第 11.1 节说明了 errno 的一般使用方法。

作为具体实例，实现复数除法的函数可以改写为：

```
Complex divComplex(Complex x, Complex y) {
    Complex c;
     double den = y.re * y.re + y.im * y.im;

    if (den == 0.0) {
        errno = ERANGE;
        c.re = 1; c.im = 0;
    }
    else {
        c.re = (x.re * y.re + x.im * y.im) / den;
        c.im = (x.im * y.re - x.re * y.im) / den;
    }
    return c;
}
```

其中的 ERANGE 是标准库定义的表示参数值错误的符号常量。

现在总结一下上述的两种基本错误处理模式，也简单说明它们的缺点：

1. 通过函数返回值报告工作情况，出现错误时返回特殊值，要求调用函数的地方处理错误。完全可能通过几个不同值报告不同的错误。这种方式的缺点是设计函数时需要选择合适的“错误返回值”，有时会遇到困难。此外，在每个函数调用处都检查错误，也会使程序中的正常控制流变得不那么清晰，干扰人的阅读和理解。
2. 在发现错误时将专门的错误变量设置为特定的值，供程序其他部分检查。显然这种错误变量只能是全局变量。这种方式的缺点是使用者可能忘记检查，因而没有察觉实际计算中已经出错了。

这些模式还有另外一些缺点。例如，通过返回值的方式只能直接在一层调用之间传递信息，如果函数需要将信息传递到几层调用外面的函数里处理，那么就很不方便了。

标准库为解决这些问题提供了另一些功能，关心这些方面的读者请查阅第 11 章中有关 <setjmp.h>和<signal.h>等头文件的介绍。由于有关功能牵涉到一些更深入的问题和技术，这里就不仔细讨论了。此外，由于程序中的错误处理是一个重要问题，人们在这方面开展了许多研究，提出了一些新的概念和机制，并将它们纳入某些新语言（例如 Ada、C++和 Java 都提供了称为“异常处理”的专门机制）。读者在今后的学习中可能遇到它们。

条件检查、断言和防御式程序设计

上面讨论了发现错误后如何处理的问题，随之而来的问题是在哪些时候应该去检查程序遇到的各种错误。这里同样没有“放之四海而皆准”的原则，但人们也根据实践总结出一些经验，值得我们编程时参考。

一般而言，每个函数对于自己的参数都有一些要求。除了基本的类型要求之外，还可能对参数值有进一步的要求。例如下面的简单求平均值函数：

```
double avrg(int n, double a[]) {
    int i;
    double s = 0.0;
    for (i = 0; i < n; ++i) s += a[i];
    return s/n;
}
```

在参数 n 的值为 0 时就会出现除 0 的错误。如果将这个函数放在某个大程序里，例如在其他地方出现下面函数调用：

```
x = avrg(m, b);
```

如果 m 的值当时是 0，后果就完全没有保证了。在定义这类函数时，一种可行方法是为错误情况选择一个“合理的”值。例如，可以在函数开始时增加对参数值的检查：

```
double avrg(int n, double a[]) {
    int i;
    double s = 0.0;
    if (n == 0) return 0.0;
    for (i = 0; i < n; ++i) s += a[i];
    return s/n;
}
```

但如果 n 是负值呢（它可能是前面某个地方计算出的值）？也可以考虑将检查改为：

```
if (n <= 0) return 0.0;
```

一段程序（例如一个函数）通常只能在送给它的数据满足一定要求时，才能完成人们所希望的某种有意义的工作；而当送给它的数据确实符合要求时，这段程序就应该保证做出所需要的结果。人们把一段程序执行前对于数据的要求称为它的前条件（precondition），把一段程序执行结束后能够保证的性质称为它的后条件（postcondition）。假设程序段 S 的前条件是 P，后条件是 Q，那么如果 S 在 P 为真的情况下开始执行，S 结束后就保证 Q 为真。这里的 P 和 Q 都应该是对一些变量等的情况的描述，例如上面函数的前条件就是要求参数 n 的值大于 0。一般而言，程序的前条件和后条件可以用逻辑表达式表示。从某种意义上说，一对适当的前后条件 P 和 Q 就描述了程序片段 S 的意义。有关程序段前后条件的认识是人们对程序和程序语言研究的重要成果，在程序理论和程序设计实践中都有非常重要的意义。

如果需要，我们可以为一些函数增加检查前条件或者后条件的语句，在发现这些条件并未满足的情况下报告错误，或者采取特别的处理方式。这类专门为检查错误而加入的描述常常被称为断言，有些语言为描述断言提供了特殊的描述形式。注意，程序里的断言并不是为了实现程序的正常控制流，而仅仅是为了检查程序片段被错误使用的情况。如果这种检查没有发现问题，它们就不会影响程序的实际效果。采用普通语句的形式写断言的优点是它可以保留到程序执行期间，在实际执行中起到一种安全卫士的作用。当然，这种作用还需要外层调用代码的配合。例如上面函数返回了 0，调用函数的代码段就应该注意到这种可能性，检查出现 0 的情况并给以适当的处理。

为了帮助编程者在程序调试中发现上述种类的错误，C 标准库提供了一种断言机制（是利用 C 语言的宏功能实现的），在<assert.h>里定义。这个头文件里提供了宏 assert，它在参数求值得到非 0 时不产生其他效果，在参数值为 0 时终止整个程序的运行，并输出一条错误信息，其中包括本源文件的名字和这个 assert 语句在文件里的行号。assert 通过调用另一个标准函数 abort 终止程序。

对于上面函数定义，下面定义中使用了 assert 宏：

```
double avrg(int n, double a[]) {
    int i;
    double s = 0.0;
    assert(n > 0);
    for (i = 0; i < n; ++i) s += a[i];
```

```
        return s/n;
    }
```

在参数出错时 assert 将报告错误，应该认为实际错误出现在调用函数的位置。

许多时候，在程序调试完成之后，我们希望去除程序里的所有断言。为此标准库提供了另一个宏名字 NDEBUG，定义这个宏名字就导致程序里的所有 assert 自动“失效”，不再对参数进行求值检查。这种机制使人可以很方便地打开或者关闭断言检查。

应该注意，assert 机制是为程序调试检查而提供的机制，让实际运行的程序依靠这种断言检查是危险的。如果检查中发现错误就可能导致程序终止，这未必是实际程序所希望的效果。这方面的问题前面已经讨论了。

程序中的检查应该做到什么样的程度？这个问题也没有唯一正确的答案。对此人们提出了一种称为防御性程序设计（defensive programming）的概念，其基本想法就是，程序片段（例如这里的函数）应当尽可能地做好自我保护，以保证其他地方出现的错误不会传入这里的代码，造成本函数的不可预知的行为，以至造成更大范围的破坏。

当然，防御性程序设计也有度的问题，一方面要考虑错误参数会不会造成实际损害，也要考虑函数的调用环境是否有产生错误的可能性。举例说，我们前面的打印直方图中的局部函数 prtHH：

```
    void prtHH(int n) {
        int i;
        for (i = 0; i < n; ++i) putchar('H');
    }
```

这个函数当然要求参数非负，否则就没有意义。但从一方面看，即使参数未负也不会产生任何不良后果。另外，这个函数只是在输出直方图的函数里局部使用。那里根本不会出现以负数调用的情况 prtHH。在这里增加检查，除了增加执行开销之外就没有什么收获。

函数 avrg 的情况就很不一样：它实现的是一种一般性的功能；很可能在一个程序里的许多地方使用。在写这个函数时，我们根本无法保证对它的调用都能满足某种先决条件。此时增加对参数的检查就是非常合理又很必要了。

另一种应考虑检查参数的情况是程序里各部分中对外提供服务的函数。由于这些函数被程序的其他地方调用，那些部分是分别开发的，可能由其他人完成。无法保证在调用我们的函数之前一定做了完全的检查。这样做也可以防止错误情况在程序的不同部分之间传播，给排除错误造成的极大困难。这种做法在大程序开发中也是非常有价值的。

最后还需要强调一下：防御性程序设计也不应该过分。如果每个函数都对参数做完全的检查，整个程序里就会出现大量重复检查，有可能大大降低程序的效率。另一方面，有些检查的代价很高，做之前也值得仔细考虑。例如，假定某函数要求参数数组中的元素互不相同，是否在函数工作之前先检查数组参数的实际情况，就很值得考虑了。

10.3 程序的初始化

前面完成的学生成绩统计程序已经是个有一点意思的程序了，它确实能在真实世界中完成一件有意义的工作。现在让我们重新分析所处理的问题，看看这个工作还有什么值得注意或者改进的地方。可以看到，整个程序的工作实际上依赖于几个量，其中总评成绩的计算依赖于所给定的成绩比例分配；直方图依赖于分数分段和最长直方图长度，两个数值确定了程序在处理直方图时的工作方式和显示形式。

应该看到，上述程序依赖于这些“参量”，适当改变参量后，这个程序仍能完成工作（学生成绩评定），只是工作方式有些变化。一个显见的事实是，改变这些参数中的一个或几个，

就能改变程序的具体行为方式。实际中人们也常常需要这种变化。例如，对不同课程的最终成绩判定，教师对期中、期末和练习所占比例常常有不同考虑。然而，在上面程序里，这些数值都是编程时确定的，限制了程序的使用范围，是一个需要考虑的问题。

10.3.1 程序行为和启动时初始化

显然，如果我们自己需要在上述参数的范围内修改程序行为，那么可以去修改源代码中的常量定义，而后重新编译，得到的可执行程序将具有所需的行为。这样看来，上面所说的情况似乎不是问题。然而在实际中这种做法往往行不通，下面讨论其中的问题。

如果把这个程序给别人用，一般而言，我们通常提供的是可执行程序而不是源程序。因为：第一，使用这个程序的人可能根本不了解程序设计，也不知道如何从源程序做出一个可执行的程序；第二，即使使用者知道怎样做，他们也未必有合适的工具（可能没有合用的编译系统等）去做出可执行程序；第三，源程序很容易被破坏，由于不当修改而变得不可用了，用户通常是不可能把它修理好的；第四，我们可能不希望将源程序提供给用户，因为源程序本身有价值，其中可能包含着我们的技术秘密等。

上面讨论说明，开发程序的人们常常面临这样一些问题：1）希望提供给用户的是可执行程序；2）希望所提供的程序能有一定灵活性，使用户可以根据需要调整软件的工作方式，而这种调整又不需要修改软件本身。与此同时，我们还使软件的使用尽可能简单而方便。使用方式极其复杂的软件不可能赢得用户，最终使用户抛弃我们。

下面以本章第一个分块开发结果（单一头文件结构的程序组织）为基础，讨论几种可能的处理方案和思想。这里的基本思想是将程序中的这些“参量”总结和提取出来，设法让用户可以自己去确定程序执行中所用的具体“参量”值。

我们用本章第一个分块开发结果作为实例，对它可能总结出下面一组量：

```
MAXNUM EXECISE，MIDDLE，FINAL，HISTLEN，SEGLEN
```

它们的值在一定程度上确定了程序的行为，我们要设法做的就是在允许修改它们的值的同时不需要修改程序。可以把这些量称为程序的*功能参量*，下面的基本想法是让用户在执行程序时有办法为这些参量提供实际值，从而能根据实际需要改变程序的行为。

由于这些量是在一批函数里使用的，又要在某个（某些）函数里设置，因此我们不能将它们定义在某个函数内部，只能将它们定义为一组全局变量：

```
int MAXLEN, EXECISE, MIDDLE, FINAL, HISTLEN, SEGLEN, SEGNUM;
```

SEGNUM 可以由其他参量算出，因此它不是独立的，但也可以将它放在这里统一处理。在一般情况下，我们一直用小写字母开头的标识符表示全局变量。这里采用大写字母拼写的标识符，是因为不想去全面修改已经开发好的程序。

要使用这些变量，程序中一些部分必须修改。需要去掉原有的枚举类型定义，在头文件 stu.h 里增加这些变量的 extern 声明，以便其他源文件能使用它们。还需要把全局数组 students 改为指针，动态分配学生记录的存储空间。修改后的头文件将是：

```
/* file stu.h 程序stu的公共头文件 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

extern int
   MAXNUM,
   MIDDLE, EXECISE, FINAL, /* 成绩比例 */
   HISTLEN, /* 最长行的长度（字符数）*/
   SEGLEN,  /* 分段长度 */
   SEGNUM;  /* 分段数，根据分段长度自动算出 */
```

```
typedef struct {
   unsigned long num;
   char name[20];
   double mid, exe, final, score;
} StuRec;

extern StuRec* students;

int readSRecs(FILE *fp, int limit, StuRec tb[]);
int printSRec(FILE *fp, int limit, StuRec tb[]);
void statistics(int n, StuRec tb[]);
void histogram(int n, StuRec tb[], int high);
void sortoutput(int n, StuRec tb[]);
int next(char s[]);
void getnstr (char prompt[], int lim, char bf[]);
int getcmd (char *prompt, int c1, int cn);
```

所有“功能参数”变量都必须在主文件 stu.c 里定义。

由于数组的大小需要用常量表达式确定，histogram 里的数组 segs 的定义也需要修改。通过动态分配是一种处理办法。为了简单，也可以利用做成绩直方图时最多每 1 分为一段的事实（由于 SEGLEN 是整数），这样就可以采用下面的数组定义：

```
int segs[101];
```

这个函数的其他部分都不需要修改。

为更好地支持程序的维护和修改（例如修改程序，考虑不同的初始化方式，如下所述），我们还应该把设置程序功能参量的工作定义为一个函数，设其原型为：

```
int init(int argc, char** argv);
```

在初始化成功时，令 init 返回非 0 值。将函数 main 修改为：

```
int main(int argc, char** argv) {
    FILE *fp;
    char fn[128];

    if (!init(argc, argv)) {
        emessage("Initiation fails. Stop program.\n");
        return 1;
    }

    ... ...
}
```

10.3.2 交互式初始化

许多读者可能马上想到第一种方式：让程序从标准输入获得信息，用户在程序启动后通过键盘输入指挥它。按这种设想，一个简单的初始化函数可能具有如下形式：

```
int init (int argc, char** argv) {
    int n;

    printf("Please give three ratio: middle execise final>");
    n = scanf("%d%d%d", &MIDDLE, &EXECISE, &FINAL);

    /* ... ... */

    students = (StuRec*)calloc(MAXNUM, sizeof(StuRec));
    return 1;
}
```

为保证程序的健壮性，我们还需要加入许多检测和处理，包括检查各个输入的正常完成，值位于允许范围。只有在正常完成所有参数的初始化的情况下，init 才返回 1，否则就返回 0。在初始化失败时程序根本无法继续工作，只能结束。当然，这时结束不会有任何损失，

因为什么工作都还没有做。为帮助用户了解正确输入方式，程序还可以先输出一段有关程序使用方式的说明信息，解释所需输入的每个参数的用途和取值范围等等。（在这里，我们实际上并没有使用 init 的参数，因此完全可以用 int init(void) 原型。采用这里的写法，只是为了下面讨论其他定义方式时不必修改 init 和 main 的原型。）

上述简单方式的主要缺点是程序的使用太麻烦。每执行一次程序，用户都需要逐一提供各个功能参量的值。为缓解这一问题，可以给这些功能参数赋以默认值，允许用户选择使用默认值或者设定新值。例如，一种简单做法是输入 0 表示采用默认值，或直接回车表示选取默认值，请读者考虑如何实现这些策略。这时需要把功能参量的定义改为：

```
int MAXNUM = 400,
     MIDDLE = 20, EXECISE = 20, FINAL = 50, /* 成绩比例 */
     HISTLEN = 60,      /* 最长行的长度（字符数）*/
     SEGLEN = 5,        /* 分段长度 */
     SEGNUM = 21;       /* 分段数，根据分段长度计算 */
```

实际程序很少采用交互式方式完成功能参量的初始化，因为这样做给用户带来的麻烦太多。在内部功能参量设计不变的情况下，下面介绍另外两种更常用的方式。

10.3.3 通过命令行参数

许多程序采用命令行参数为功能参量提供值。这时的 init 函数具有下面框架：

```
bool init (int argc, char** argv) {
    // 在这里分析命令行参数，并用得到的值设置功能参数
    ... ...
}
```

为完成这个 init，我们首先需要设计一种命令行形式，以便于对各个参数的分析。例如，可以为每个命令行参数选定一个字母，例如可以选用下面形式写出参数（这只不过是一种设计，读者完全可以提出其他形式的设计）：

```
N500 E30 M20 F50 H40 L5
```

其中用 N、E、M、F、S、H、L 分别代表 MAXNUM、EXECISE、MIDDLE、FINAL、HISTLEN 和 SEGLEN。我们还可以不要求特定顺序，可以缺少其中的一些项，对缺少的东西都采用默认值。这样，基本的 init 函数定义可以采用如下形式：

```
int init (int argc, char** argv) {
    while (*++argv != 0) {
        char *p = *argv;
        switch(*p) {
        case 'N':
            MAXNUM = atoi(++p); break;
        case 'E':
            EXECISE = atoi(++p); break;
        case 'M':
            MIDDLE = atoi(++p); break;
        case 'F':
            FINAL = atoi(++p); break;
        case 'H':
            HISTLEN = atoi(++p); break;
        case 'L':
            SEGLEN = atoi(++p); break;
        defualt:
            emessage("Unrecognized command-line argument: %c\n", *p);
            return 0;
        }
    }
    /* 做一些合法性检查 */
    if (EXECISE + MIDDLE + FINAL != 100) {
        /* 输出错误信息并返回0 */
    }
    /* ... ...  检查其他参数值和相互关系 */
    SEGNUM = 100/SEGLEN+1;
```

```
    students = (StuRec*)calloc(MAXNUM, sizeof(StuRec));
    return 1;
}
```

在分析各命令行参数时，需要从表示数值的字符序列表示求出数值。前面已经定义过这种函数。上面 init 函数里直接使用了 C 标准库里函数 atoi，它由表示整数的数字字符序列得到一个 int 值，这个函数在头文件<stdlib.h>里说明。代码中写 atoi(++p)是因为必须将指针 p 向前移一个字符，才能使它指向数字字符串的开始。

许多实际程序不仅需要通过命令行指定功能，还有一些其他参数，例如被处理的文件名等。为了区分这两类参数，需要更仔细地考虑命令行的格式（语法形式）。为方便处理，许多系统函数为功能参数选定了一个引导字符。例如，DOS 系统命令的功能参数都用 / 作为引导字符，而 UNIX 系统通常用 – 字符。这样，在分析一个命令行参数时，只需看一个字符，就可以断定其实际作用。随后的分析完全可以采用类似上面的模式。

10.3.4 采用初始化文件

采用命令行参数也有缺点。如果一个程序里需要确定的参量很多，每次启动程序提供命令行参数就容易出错。为此人们提出了另一种常用的系统初始化技术，那就是为我们的程序专门设计一个初始化文件，将设置功能参数实际数据存入这个文件里。

在程序启动后，函数 init 试着去打开和使用这个文件。如果文件打不开（可能是没有），程序就按照内部的默认方式工作。如果能打开文件，程序就用这个文件中提供的值设置有关的功能参数。这种文件的名字常常以.ini 或者.cfg 等为扩展名，称为软件系统的初始化文件或者配置文件。在各种真正复杂的软件里，我们总可以找到这类东西。

为能使用初始化文件为程序的功能参量提供初值，我们先要设计好这种文件，确定表示各种参数值的描述形式。而后实现一个完成功能参量初始化的函数，其功能类似于上面那个处理命令行的 init 函数，但实际数据是从指定文件里读入的。为上述程序提供这种初始化方式的工作留给读者作为练习。

经过上面各节的讨论和修改，这个程序已经越来越像一个可以用的软件了。上面讨论是一些能用于真实软件的思想和技术。

10.4 程序开发过程

如果一个程序不是极端简单，通常都不可能通过从第一个字符开始，以一行一行向下写的方式直线性地、一蹴而就地完成。即使是对不太大的程序采取这种方式，一下写出几百行程序代码，而后送给语言系统去编译加工，一般总是得到长长的一串错误信息。反复检查修改程序，好不容易完成了编译，但运行的情况又往往完全出乎我们的意料，不能实现所需的功能，自己对为什么产生这样的情况也完全没有预期，更无法立即想到看到修正的线索。随后我们只好投入非常痛苦而又漫长的检查错误、排除错误的工作。开发出的每个部分都可能包含错误，一大堆都包含错误的东西堆积在一起，常使人陷入一种束手无策的境况。

上述方式绝不是一种合理的工作方式，也不会是有效率的工作方式。本节将讨论有可能如何更有效地工作，有些内容是对本书中已经论述过的问题的总结。

人们在讨论程序设计时，常常提出*自上而下*或者*自下而上*的开发方法。不说是写程序，而是说开发程序，也是想说明这一过程的艰难，并不是平铺直叙一帆风顺的。这里的“上”指比较抽象的层面，“下”指更具体的层面，更接近计算机，接近基本程序设计语言的层面。一个程序只要不是极其简单，通常都不仅仅是在最基本的语言层描述的，而是通过语言提供的方式，由一组位于不同层面的描述组成。

举例来说，前面实例中的函数 main 就是基于一组标准库函数，以及几个自己定义的函数 getnstr、commander 和 next 描述的，而函数 commander 又是基于一组标准库函数和自己定义的函数 readSRec、getcmd、statistics、histogram、sortoutput 和 next 描述的。程序里的函数调用关系形成了一种分层，构成了程序的分层逻辑结构。

10.4.1 自上而下的开发

在开发一个程序的初期，我们往往需要从程序的整体功能出发，开发出程序的基本框架，将整个程序分解为一些基本部分。在 C 语言里就是分解为一些实现程序的各部分功能的函数。这样的分解在程序的后续开发阶段中还会不断进行下去。这种以分解程序功能为指向的工作过程就是自上而下的开发过程。

用前面的程序问题为例。在确定了程序要处理一系列学生成绩文件，并确定通过与用户的交互完成具体文件的确定之后，整个程序的基本工作框架也就确定了。由于对每个文件的处理又牵涉到一系列问题，因此我们考虑把这部分工作定义为函数 commander。这样基本上就有了程序的主函数框架。可以确定让主函数完成文件打开关闭工作（当然也可以将这部分工作放到 commander 里，那就形成了另一分解设计，在 main 和 commander 之间的关系和需要交换的信息会有所不同），由此形成的 main 函数见 10.1 节。

一旦我们写出了这个主函数，就应该考虑去编译它，看看这部分程序中是否有语法错误。在编译通过之后，我们还没办法运行，因为缺少 main 需要调用的一些部分。假设 next 和 getnstr 是现成的东西（已定义好并经过实际使用的检验），已经为它们建立了另一个源程序文件 utilities.c，并在程序头文件 stu.h 里放入了它们的原型。但 commander 完全是新的东西，没有它就无法进行程序连接（连接时将出现“commander 没有定义”一类的错误信息）。此时的正确策略不是赶快把 commander 写出来，因为如果那样做，同样问题又会出现在 commander 里，如此下去就只能把所有函数都定义好，然后在编译运行和调试了。

人们常用的解决办法是先为 commander 写一个简单实现，甚至空的实现。当然，这个实现应具有正确的函数头部，例如就用：

```
void commander (FILE* fp, char *fn) { }
```

因为 commander 不返回值，用一个空函数体，就能和 main 等等一起构成一个“完整的”程序，完成编译后就可以去调试运行了。由于分块开发的考虑，我们将这个函数也放在程序的主文件里。调试之前还需要准备一个数据文件。

目前情况下，我们已经可以检查 main 能否与用户正确交互，文件名正确时能打开它，没有相应文件时能“正确”报告错误等等。为了进一步检查，可以改用下面的实现：

```
void commander (FILE* fp, char *fn) {
    char line[128];
    printf("File name: %s", fn);
    fgets(line, 128, fp);
    printf("Commander: get file %s", line);
}
```

检查函数名是否正确传递，文件内容能否正常读入。在完成了这些检查之后，我们对 main 函数的功能有了更多信任，开发也可以进入下一个阶段了。

有时需要把待开发的一些部分放在另一源程序文件里。例如，假设现在已经写出了函数 commander 的实现，要测试这个函数，就需要函数 readSRec、statistics、histogram、sortoutput 的定义（假设 getcmd 已经定义）。我们计划将程序的输入输出功能实现在一个源程序文件里，将成绩处理功能实现在另一个文件里。在实现 commander 时显然已经确定了上述函数的原型（否则不可能写出 commander），而且应该已经将这些原型写入了程

序的头文件 stu.h。为了测试 commander，现在就应该创建文件 stu_io.c 和 stu_fun.c，并在其中写出上述函数的简单实现。例如，readSRec 可以定义为：

```
int readSRecs(FILE *fp, int limit, StuRec tb[]) {
    printf("readSrec called\n");
    return 0;
}
```

放在文件 stu_io.c 里，这里写 return 0 是为了保证函数确实返回了值（与函数头部一致，也因为 0 不会造成问题）。其他函数的情况与此类似。

虽然这些函数并没有做任何实际事情，但它们的存在却使我们可以去进一步调试程序，检查 commander 的各种功能是否都已正确实现。这些描述本身也是有价值的。一方面，它们为检查程序中各部分间的类型关系提供了信息。同时也构成了我们进一步开发的框架，后面的工作就是将函数功能的正确实现填充到这些框架的空位之中。如果一个函数比较复杂，我们甚至可以分步骤开发它。逐步将一个假的实现扩充为最终的真正实现。在完成了函数的一部分之后就进行检查（因为已经有了由上层开发形成的执行环境），看这部分是否满足需要，功能是否正确。人们将这类描述称为“假函数实现”或者存根（stub）。

上述描述大致给出了自上而下开发的工作方法，可能遇到的问题和解决办法。读者根据自己的经验，不难想象这一过程可能怎样继续下去。

10.4.2 自下而上的开发

程序开发未必是纯粹的自上而下分解过程，有时也需要按其他方式进行。举例来说，我们需要开发一个程序，在分析问题中发现程序里的许多地方都要用到复数运算。在这种情况下，开发一个正确完整的复数模块，将使程序其他部分的开发变得更加清晰方便。这时我们就应该在某个时候转到复数模块的开发工作中，先行将这一部分完成。而后就可以基于这个部分去设计和实现程序的其他部分了。

相对于程序的整体功能而言，像复数这样的模块属于底层的服务性模块。这种先行开发服务性的底层模块，而后再开发程序上层的工作方式，就是所谓的自下而上的开发，或者自底向上的开发方法。在这种方式下，我们逐步构造起一块块的底层功能，基于构造好的功能进一步开发上层模块，最终完成整个系统。

在前面经讨论过许多自下而上开发中的问题里，有许多程序实例是完成一个函数。在 C 语言程序里，许多开发工作的结果就是一批函数，或者包含若干类型、一组常量和变量的一集函数（如上面提到的复数模块）。在开发这种程序部件时，我们也需要采用系统化的方式，一边开发，一边调试并排除程序中的错误。只有通过这样的工作过程，我们才能逐步了解作为开发结果的函数或模块的功能和性质，为更上一层的集成做好准备。

在写好了一段完整的程序（例如一个函数等）后，就应该去编译它。编译程序可以检查出这段程序中的语法错误，还能检查出上下文关系方面的错误，例如所用的变量是否有定义，运算对象的类型是否与运算符的要求相符，变量的值能否转换等。每开发完一个部分（例如一个函数或者一个源程序文件），就应该尽可能地在这一部分的局部范围中检查和测试，纠正所发现的错误。这个工作的结果将得到一个潜在错误比较少的部分。进一步说，当两个相关的部分完成之后，我们就应该进一步去检查，看看它们的组合是否能完成预定的工作。这样一步步进行下去，在完成了程序的所有部分时，系统中剩下的错误就会少得多，进一步检查和排除整个系统中错误的工作也会容易得多。

只通过编译做程序静态检查远远不够，还应该去运行这种程序片段，在动态运行中检查它们的功能。正如前面所说，这样开发出一个或一组函数不是完整程序。为了能运行和测试，必须写出虚拟的主函数，由它去调用这些函数，以便运行和测试写出的程序片段。第 4 章最

后讨论了写驱动程序的问题。如果一个程序被划分为多个源文件，那么就可能需要写出包含虚拟主函数的源文件，用它编译的结果与需要调试的模块连接。为能这样去做分块开发，我们常常需要先做出一批有关的头文件。

通过这样的调试执行，修改更正发现的问题，也为最后的系统集成做好了准备。这种工作方式可以避免把调试工作都集中到系统开发的最后。通过预先的分别调试，就可能改正各独立模块内部的大部分问题，为最后集成和调试提供更好的基础。

10.4.3 实际开发过程

实际程序开发过程往往不是纯粹自上而下或者纯粹自下而上的，而是呈现出某种混合方式，自上而下的分解与自下而上的构造交替进行。

对于一个比较大的系统，在自上而下的功能设计和分解之后，所确定和仔细描述的各模块描述常常被交给不同的开发小组，由各个小组分别进行开发。各开发小组也可能对系统功能结构进行进一步分解，而后将任务分配给更小的小组或者个人。有关系统的各层次开发完成后，开发结果被逐层集成起来，通过上层结构联为一体，最终构成完整的系统。

实际上，在考虑如何做分解时，我们心里往往有一些基本的目标和模式。例如，在考虑数据输入部分的分解时，如何将其中的一些基本功能分解到能够直接调用标准库输入函数实现，或者可以归结到某种已经熟悉的处理模式，这些都是很有意义的目标。

一些常规的计算过程也是分解的启示。例如，将一组数据按照某种方式顺序排列（*排序*）常常有助于问题的解决；许多时候我们是在一组数据中查找满足某个条件的元素（称为*检索*或者*查找*）。许多工作有使用广泛的方法，或者有库函数可用。将程序分解到这个层次，就可以利用有关的功能，有时需要写一些短小的辅助函数（例如前面为 `qsort` 写的表示所需比较准则的函数）。

程序里的数据也有许多常用的组织方式。基本的构造手段是数组和结构，指针用于构造更复杂的结构。在后续的数据结构课程中，读者将会看到许多经典的数据组织方式。这些常用的和典型的方式也引导着程序功能的分解过程。一旦确定将程序里的数据用某种方式组织起来，随后的一些编程工作就是围绕着所确定的数据结构进行的。

另一方面，在程序开发的各个阶段，实现程序各部分基本功能的代码（函数等等）也需要逐步开发出来，这时往往需要采用自底向上的方式，一步步地实现这些部分。

完全采用自底向上底方式很容易看不见全局，这种情况可能导致所实现的部分不能很好地与程序的其他部分协同工作。在实现了程序中的一部分功能后，为了使之能够融入整个程序，也可能需要对其实现方式，使用界面（函数原型等等）做一些调整。有时我们会发现，对某些部分的少许调整可以使一段程序更容易使用，或能够用到更多的地方。

在程序开发中，这种为实际需要而做的功能调整或者形式调整也是常常出现的。这时，基本程序的清晰和易修改性质就非常重要了。本书自始至终强调这方面的问题，也就是为了读者及早养成良好的编程习惯，为了实际程序设计的需要。

还有一个问题也非常值得提出来。随着做出了越来越多的程序，我们会发现，其中许多程序的某些地方使用着类似的功能。对这些程序片段进行整理和推广，就可能做出一些可以用到许多不同程序里的功能函数或者模块。这种自下而上的积累又会成为我们今后分解程序的线索。重复使用已有的代码、设计等等的活动被称为*重用*，如何在程序开发的各个层次上支持各个层次重用，已经越来越受到人们的重视。C 语言开发的比较早，这方面的能力也比较贫乏。许多更新的语言将支持重用作为语言设计的一个重要目标。

总而言之，程序是一种非常复杂，而且正在变得更加复杂的人工制品。任何能有利于应对复杂性的技术和方法都可能在这里发挥作用。人们为解决程序的复杂性问题，从各个方面开展研究，提出了许多实践中证明有益理论、技术、方法，开发了许多工具系统。读者继续

在这个领域中学习，会逐渐接触到其中的一些东西。

10.5 进一步学习的建议

本书介绍了 C 语言的各种机制和用它们做程序设计的技术，在此期间还讨论了程序设计的许多一般性问题，就如何去思考与程序设计有关的问题，如何分析问题并做出决策，如何评价各种选择的优点与缺点，如何写好程序，以及好程序的各种评价准则等提出了许多观点和认识。这里想强调什么是好的正确的程序设计，以及怎样才能做出好程序来。书中程序实例也体现了作者对这些问题的认识。一些实例还反应了做程序的思考和工作过程。

程序设计作为一项智力劳动，已由千万研究者们和实践者们讨论、演练、探究了几十年。几十年积累下来的智力财富绝不是一本教材所能包容的。本书只想将其中最基本最典型的问题介绍给新接触这个领域的人们，希望为其未来发展铺一块基石，希望能帮助读者为今后的学习和工作打下较好的基础。读完本书，做好书中相当一部分练习，读者可能已在一些方面收获颇丰。但也应该认识到，这毕竟只是在计算机和程序设计领域的学习中走完的第一步，继续学习的路还很长很长。实际上，任何准备在这个领域里摸爬滚打的人都需要不断学习，需要不断地接受新鲜事物，需要更多的学习、思考和实践。

由于篇幅（一本书）和时间（目标是一个学期的课程）限制，本书不可能包含更多内容，这一点也反应在书名中，这里只能讨论程序设计和 C 语言两方面的最基本东西。有意在这一领域里继续学习的人们应该继续努力。这里想为读者介绍一些可以继续学习的书籍材料，并对有关方面的情况做一些评述，供关心这些方面的读者们参考。

算法和数据结构：

学过基本程序设计课程后，计算机科学技术教育的下一门课程是“数据结构”或“算法与数据结构”。计算机和相关领域学生都会接受该课程的系统教育，因此这方面情况不需要过多讨论。另一方面，作为业余读者和计算机爱好者，如果计划在计算机领域中继续努力，作为这个学习过程的下一步，也必须认真地去学习算法和数据结构的知识。

当程序处理的数据变得更复杂时，数据的组织问题会变得越来越重要（本书已有简单讨论）。这时，在考虑计算过程的实现之前，首先需要设计好数据的组织方式，将程序中所用的数据用某种易于操作的形式组织起来。数据结构讨论的就是这方面的问题。

目前市面上有关数据结构的书籍很多，内容大致分两个层次：第一，在抽象层次上讨论数据组织的问题和技术，介绍一些典型数据组织方式（数据结构），介绍一些与数据结构有关的典型算法的思想和细节，其中的一些算法与特定数据结构有关。第二，通过一种程序设计语言，讨论如何利用该语言的基本机制内实现各种有价值的数据结构和算法。如果进一步学习算法和数据结构的内容，可选一本采用 C 语言讨论的书籍。这方面的书不少，例如张乃孝等编著的《算法与数据结构——C 语言描述》，高等教育出版社 2002 年出版。

C++语言及面向对象的程序设计

C 语言是 1970 年代设计的语言，当时是为了作为一种替代汇编语言的工具。因此 C 语言中的程序设计是从比较低级的层次开始的，其中缺乏高级的程序组织机制和类型定义机制。C++是由 C 语言发展出来的一个语言，其中有一个与标准 C 语言基本兼容的子集，C++的标准库里也包含了 C 语言的标准库。C++语言的设计体现了人们对于程序设计的许多新认识，包括类型定义和数据抽象，面向对象的程序设计，通用型程序设计等等。

经过 20 多年的发展，C++已经成长为一种比较成熟的语言，1998 年 C++语言完成了标准化工作，成为一种有了清晰严格定义的标准化语言。在这些年里，人们用 C++写出了许多

非常重要的软件系统，包括许多系统软件和应用软件。通过实践，人们也开发出应用 C++语言的程序设计技术和一般性的所谓面向对象的程序设计技术。在学习了 C 语言之后，进一步学习 C++语言及其支持的程序设计技术是很有价值。

目前国内撰写或翻译的有关 C++语言及其程序设计的书籍很多，读者可以根据情况选择参考。其中必须提出的一本是 C++语言设计师 Bjarne Stroustrup 的《C++程序设计语言》。该书全面讨论了 C++语言的各个方面，以及 C++所希望支持的程序设计技术。《C++程序设计语言》用很大篇幅讨论面向对象程序设计的思想，以及采用面向对象的方式组织程序的技术和问题。书中有许多接近真实问题的实例。《C++程序设计语言》2000 年“特别版”的英文版已由高教出版社作为“教育部高教司推荐国外优秀信息科学与技术系列教学用书”影印出版，该书中译本由本书作者翻译，机械工业出版社 2002 年出版。

Andrew Koenig 和 Barbara E. Moo 从很早开始参加 C++语言开发，他们的《Accelerated C++》（中译本）是一本有关 C++语言的入门书。该书假定读者有一点程序设计经验，但不了解 C++。该书虽篇幅不大，内容却比较丰富，从 C++语言的基本机制讨论到面向对象的程序设计。该书完全以标准 C++为背景。除此之外，本书作者为以 C++语言作为第一门程序设计课程编写了一本《C++语言基本程序设计》，2003 年由科学出版社出版。它可作为 C++程序设计的入门读物。该书是本书的姐妹篇，有许多共性。但随着讨论的进展，读者可以逐渐看到在这两种语言中思考程序设计问题时越来越大的差异。C++语言有一个功能强大的标准库，利用这个库，许多在 C 语言里比较复杂的程序设计问题都会变得非常自然。

关于 C++语言的书籍还需做一点说明。市场上有些书籍不符合 C++标准，讨论的是多年前的 C++语言，也请读者在选择参考时注意。

程序设计的实践性问题：

程序设计中有许多实践性问题。这方面的情况本书中虽有所涉及，但由于篇幅等因素，不可能有更详尽深入的讨论。为了进一步学习程序设计和计算机科学技术，了解人们在程序设计实践中的考虑和经验也非常重要。

这里想推荐给读者的是一本篇幅不大，比较通俗而又很深刻的著作：Brian W. Kernignan 和 Rob Pike 的《程序设计实践》。该书作者在程序设计领域工作多年，也参与了许多实践性的培训活动，写过多部在世界上产生了重要影响的著作（包括《C 程序设计语言》等）。作者在书中讨论了实际程序设计中必然会遇到的许多问题，包括程序设计风格、算法和数据结构的选择、程序不同部分间界面的设计、程序测试和错误排除、程序执行效率等。如果读者希望进一步在程序领域工作，一定要读读这本书。《程序设计实践》的影印本已由机械工业出版社出版，其中译本由本书作者翻译，2000 年由机械工业出版社出版。

程序设计的理论和严格方法

程序设计的产出品是程序和软件。随着社会信息化的发展，各种计算机化的系统在社会生活中的地位和作用日益明显，由于计算机系统失误而造成的生命和财产损失也越来越引起社会的重视。我们需要有更可靠的计算机应用系统，这也呼唤着功能正确可靠的程序。从计算机诞生之初，如何开发正确的程序就一直是计算机专业人员特别关注的问题。

本书中讨论的是朴素直观的程序设计过程，从分析问题开始，通过分解逐步做出程序，经过调试并排除所发现的错误，最终得到一个“基本上”能完成所需工作的程序。这实际上也是目前绝大部分计算机软件开发过程的写照。但是，随着计算机系统越来越多地介入社会运转的各个关键领域，这种“基本上”能完成所需工作的系统已不能满足社会需要了。医疗设备控制系统的失误已导致一些病人的死伤；许多空难的最终调查说明祸根在飞行控制系统里的错误；多次航天发射的失败或航天器丢失也都是由于软件故障；核反应堆控制系统的故障可能对社会造成的危害更令人不寒而栗。为此，计算机工作者一直在努力研究能从根本上

扭转这种局面的理论、技术和方法，其中一类研究的目标是程序设计的严格方法。

如果读者想了解这方面的一些情况，可以阅读本书作者翻译的《从规范出发的程序设计》（机械工业出版社，2002），这是一本在欧洲不少重要大学里使用的教科书，常被用作第二门程序设计课程的教材。该书讨论了程序设计过程中的许多道理，讨论了一套严格的程序设计方法，可以认为它描述了一套"程序的数学演算"。举例说，本书介绍过的不变式概念在严格的程序设计方法中就有非常重要的地位。了解一点程序的理论，对于采用朴素方法做程序也会很有帮助，因为那里的观点和方法能帮助我们看清许多问题。

应该认识到，计算机领域中的许多东西都是相通的，程序和程序设计可以看作这里一切工作的基础，因为从本质上说计算机就是一种程序机器。可以说，任何人要想明天在计算机科学技术领域中很好地工作，无论是做实践性的还是理论性的工作，都需要有基本程序设计方面的扎实基础。对于认识计算机科学技术的整个领域及其所面对的基本的和深入的问题而言，这种基础是非常重要的东西，永远值得我们花时间去学习和思考。

练习

1. 重写前面提出的筛法程序，用一个 `int` 数组表示被处理的整数序列，但不是用每个数组元素表示一个整数，而是用数组元素 `int` 值的每个位表示一个整数。当一个整数没有被划掉时，它对应的位应该是 1（这实际上规定了初始数组中元素的值），而当它被划掉时对应的位就被改变成 0。这样可以大大地节约存储。利用位运算实现这个程序。
2. 本练习要求你实现"一台"简单的执行符号指令的计算机。假定"程序"文件里保存着一串（不超过 128 条）具有如下形式的"指令"程序：

```
rset 100
save 0
rset 204
...
```

假定这一计算机只有一个可保存一个整数的寄存器，以及 128 个可以保存整数的存储单元，这些单元分别编号为 0 到 127（单元地址）。该计算机包含下述指令：

指令形式	解释
`rset` *n*	将寄存器设置为值 *n*
`save` *m*	将寄存器里数据保存到单元 *m*
`load` *m*	将单元 *m* 的数据装入寄存器
`jmpz` *d*	如果寄存器里的数据为 0 就跳到指令 *d*
`plus` *m*	寄存器现有数据上加单元 *m* 的数据
`subs` *m*	寄存器现有数据上加单元 *m* 的数据
`mult` *m*	寄存器现有数据上加单元 *m* 的数据
`divd` *m*	寄存器现有数据上加单元 *m* 的数据
`read`	读入数据到寄存器
`rwrt`	将寄存器里的数据输出
`halt`	停止运行

假定程序里的指令从 0 开始排列编号。

请编写一个程序，它能从由命令行参数指定的文件读入具有上述形式的"程序"，并执行该"程序"。请用上述指令编写完成下面工作的程序文件：1）计算 1 到 n 的所有数之和，其中的 n 由读入"指令"得到。2）计算并输出前 n 个斐波那契数，其中的 n 由读入指令得到。3）计算数 n 的整数平方根，即平方不大于 n 的最大整数。请自己写出几个有趣的程序，并通过运行确定其正确性。

3. 设计并实现一台指令用数字形式编码的计算机。该机器的指令与前一计算机相同，但都用整数编码。每条指令的指令码及操作数一起放在一个整数（int）里，操作数占据整数的最低 12 位，指令码占据更高的 4 个二进制位。各指令对应的指令码如下：

符号指令	指令码	解释
rset *n*	0000	将寄存器设置为值 *n*
save *m*	0001	将寄存器里数据保存到单元 *m*
load *m*	0010	将单元 *m* 的数据装入寄存器
jmpz *d*	0011	如果寄存器里的数据为 0 就跳到指令 *d*
plus *m*	0100	寄存器现有数据上加单元 *m* 的数据
subs *m*	0101	寄存器现有数据上加单元 *m* 的数据
mult *m*	0110	寄存器现有数据上加单元 *m* 的数据
divd *m*	0111	寄存器现有数据上加单元 *m* 的数据
read	1000	读入数据到寄存器
rwrt	1001	将寄存器里的数据输出
halt	1010	停止运行

采用十六进制写法，下面整数表示的指令

```
0x00ff
```

表示将寄存器设置为整数 255。这里的指令码部分为 0，操作数部分为 255。下面指令：

```
0x4005
```

要求将单元 5 的内容加到寄存器现有内容上。请注意，按照现在的设计，指令 rset 只能在寄存器中设置正值。

假定文件里存放着这种指令书写的“程序”，你的计算机读入命令行参数所指定文件里的这种程序并执行它。请将你在前面写的一个或者几个程序翻译为这里的“机器指令”形式，实际运行并检查其执行效果。

4. 扩大前一题所开发的计算机的存储空间（请考虑一下，你的计算机可能提供多大的“内存”），合并上述程序里保存数据的存储区和保存“程序代码”的存储区，并根据需要修改其他方面的调整，使之能成为一个有机整体。现在你已经开发出了一台“存储程序计算机”，其基本结构和运行方式都与你所用的计算机类似。
5. 请写一个函数，它有一个输入文件指针参数和一个输出文件指针参数，从输入文件读入第 2 题的符号形式的“源代码程序”，输出对应的符合第 3 题所给形式的代码。写一个主函数，令你的函数处理一个或几个“源代码”文件，生成相应 “目标代码”文件。并将生成结果与你前面手工翻译的结果比较。你做出的就是一个“汇编程序”。
6. 请开发一个简单的“程序开发环境”。所提供的功能包括：
   - 调用某个编辑器编译第 3 题形式的“源程序文件”。你可以用<stdlib.h>里的 system 函数启动另一个程序执行。当那个程序结束时，控制将转回你的程序。
   - 调用你所编写的汇编函数，将刚刚编写好的源程序翻译为目标代码程序。
   - 调用你实现第 3 题的计算机的功能的函数（将那里开发的程序改造为一个函数）运行目标代码程序。
7. 扩充你在第 6 题所开发的程序设计环境，提供一种“单步执行”的调试功能。使你的计算机可以按照你的命令一次一条指令地执行程序，并可以随时检查寄存器和各个存储单元的内容。应允许在任何时候终止程序的运行。
8. 对于上面开发的程序做你所考虑的各种扩充。