10. 技能2——文件与工程

授课教师：游伟 副教授、孙亚辉 副教授

授课时间：周二14:00 – 15:30，周四10:00 – 11:30（教学三楼3304）

上机时间：周二18:00 – 21:00（理工配楼二层机房）

课程主页：https://www.youwei.site/course/programming

目录

引子：YOJ平台大数据量的输入输出

▾ 测试点 #3　✖ 2　得分：0　用时：14 ms　内存：2692 KiB

输入文件 (hokej4.in)

```
2000 500
27299 175
2753 191
29542 90
74587 135
83399 69
7596 19
68184 126
44060 40
75996 22
99422 69
<4605 bytes omitted>
```

YOJ-698：Hokej

输出文件 (hokej4.out)

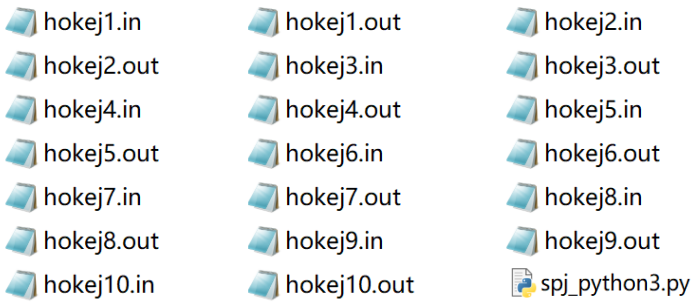
```
1071323636
13 29 34 53 162 196
108
31 162 126
37 13 180
46 53 73
83 34 298
100 196 342
126 29 130
14
<1237 bytes omitted>
```

选手输出

```
1071267415
10 126 162 239 360 475
31 162 348
32 162 414
33 162 479
34 162 398
35 162 321
36 162 236
37 162 389
38 162 200
39 16
<5509 bytes omitted>
```

YOJ的工作原理：

- 出题人提供测试点数据，每个测试点对应一组输入输出文件（*.in和*.out）
- 编译用户提交的源代码，生成可执行程序
- 对于每个测试点，测试平台会将输入文件作为标准输入（键盘输入）提供给目标程序
- 将目标程序在标准输出（屏幕输出）的结果与测试点的输出文件比较，看是否一致
- 对于部分有多种解都可行的题目，会采用特别的评判脚本，对输出的正确性进行评判



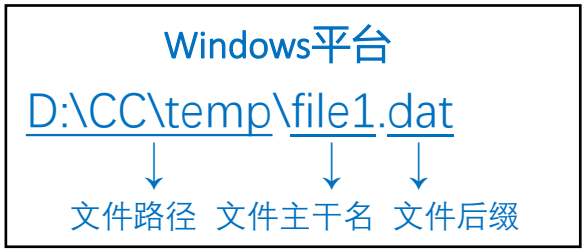
终端

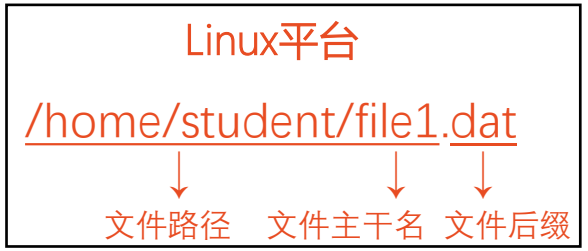
```
51246 151
19559 161
14037 93
56956 142
35963 199
25964 189
9540 81
96711 121
48179 30
45359 156
28063 30
30304 188
18785 55
17368 67
53851 110
38112 105
97239 94
56665 195
39291 166
21499 113
```

10.1 文件

- 文件（file）一般指存储在外部介质上数据的集合。操作系统是以文件为单位对数据进行管理的。
- 输入输出是数据传送过程，数据如流水一样从一处流向另一处，因此常将输入输出形象地称为流(stream)，即数据流。流表示了信息从源到目的端的流动。
  - 在输入操作时，数据从文件流向计算机内存
  - 在输出操作时，数据从计算机流向文件(如打印机、磁盘文件)
- C语言把文件看作一个字符(或字节)的序列，即由一个一个字符（或字节）的数据顺序组成。一个输入输出流就是一个字符流或字节流。
- 为了简化用户对输入输出设备的操作，使用户不必去区分各种输入输出设备之间的区别，操作系统把各种设备都统一作为文件来处理。例如，终端键盘是标准输入文件(stdin)，显示屏是标准输出文件(stdout)。

10.1.1 文件的基本概念





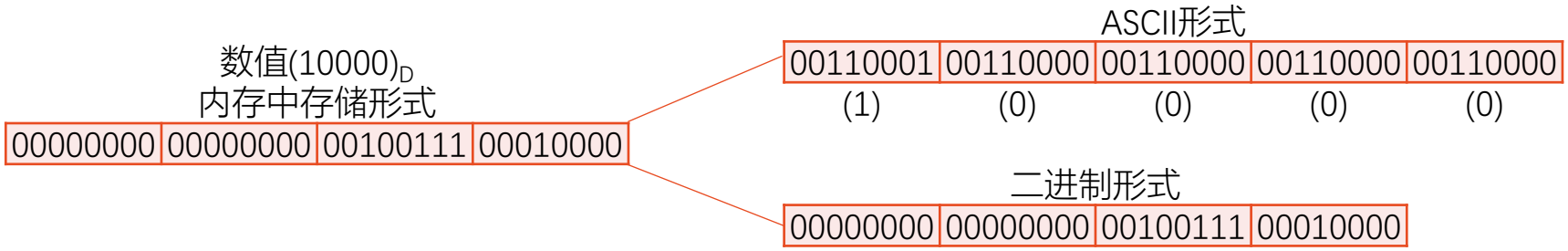
- 文件标识：文件路径+文件主干名+文件后缀
  - 文件路径表示文件在外部存储设备中的位置
  - 文件名主干的命名规则遵循标识符的命名规则
  - 文件后缀用来表示文件的性质
- 文件分类：在程序设计中，主要用到两类文件
  - 程序文件：包括源程序文件(后缀为.c/.cpp)、目标文件(后缀为.obj或.o)、静态链接库（后缀为.lib或.a）、动态链接库（后缀为.dll或.so）、可执行文件(后缀为.exe或无后缀)。这种文件的内容是程序代码。
  - 数据文件：文件的内容不是程序，而是供程序运行时读写的数据，如在程序运行过程中输出到磁盘(或其他外部设备)的数据，或在程序运行过程中供读入的数据。如一批学生的成绩数据、货物交易的数据等

思考：

- YOJ排行榜网页上的得分是用ASCII码形式存储还是二进制形式存储？
- YOJ后台是用那种形式存储得分？

10.1.1 文件的基本概念

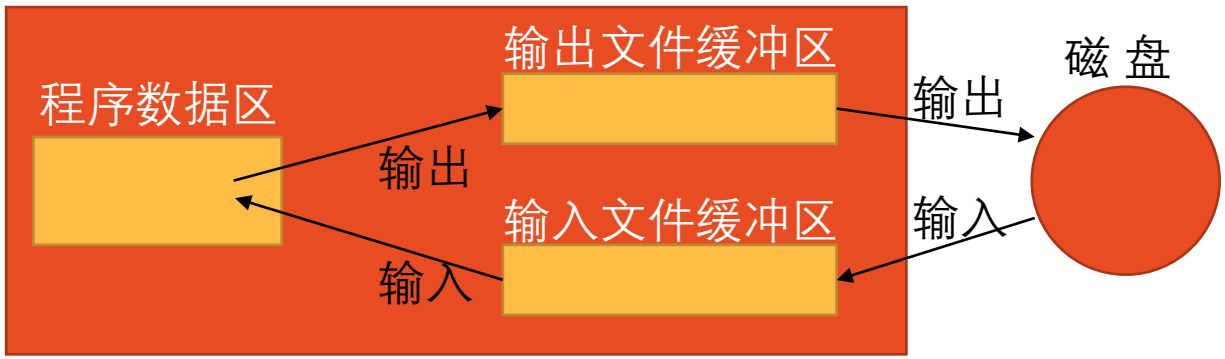
- 数据文件分类：根据组织形式不同，可分为文本文件和二进制文件
  - 数据在内存中是以二进制形式存储的，若不加转换地输出到外存，就是二进制文件，可以认为它就是存储在内存的数据的映像
  - 如果要求在外存上以ASCII码的形式存储，则需要在存储前进行转换，每个字节存放一个字符的ASCII码，形成文本文件
  - 字符一律以ASCII码形式存储，数值型数据既可以用ASCII码形式存储，也可以用二进制形式存储



  - 用ASCII码形式输出时字节与字符一一对应，一个字节代表一个字符，便于对字符进行逐个处理和输出，也便于人阅读理解。但一般占存储空间较多，而且要花费转换时间（数位合并和分离，myscanf和myprintf里%d的实现）
  - 用二进制形式输出数值，可以节省外存空间和转换时间，把内存中的存储单元中的内容原封不动地输出到磁盘(或其他外部介质)上。此时每一个字节并不一定代表一个字符，不易于人理解。

10.1.1 文件的基本概念

- 文件缓冲区
  - C标准采用缓冲文件系统处理来数据文件，系统自动在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区， 其大小由编译系统确定。
    - 在向文件输出数据时，它就作为输出缓冲区
    - 在从文件输入数据时，它就作为输入缓冲区
  - 文件缓冲区目的是节省存取时间、提高效率
    - 从内存向磁盘导出数据：必须先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘去
    - 从磁盘向内存导入数据：一次从磁盘文件将一批数据填充满内存缓冲区，然后再从缓冲区逐个地将数据送到程序数据区给程序变量



10.1.1 文件的基本概念

- 文件信息结构体
  - 每个被使用的文件都在内存中开辟一个相应的文件信息区，用来存放文件的有关信息（如文件的名字、文件状态及文件当前位置等）
  - 这些信息是保存在一个结构体变量中的，该结构体类型是由系统声明的，取名为FILE

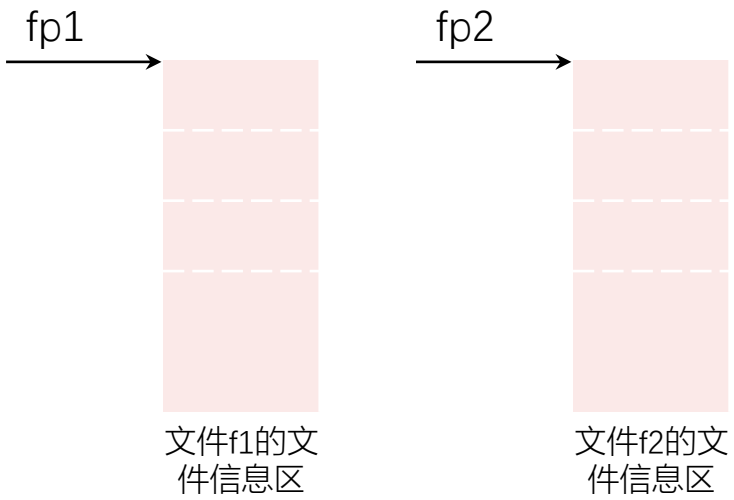
```
typedef struct
{     short level;              //缓冲区“满”或“空”的程度
      unsigned flags;           //文件状态标志
      char fd;                  //文件描述符
      unsigned char hold;       //如缓冲区无内容不读取字符
      short bsize;              //缓冲区的大小
      unsigned char*buffer;     //数据缓冲区的位置
      unsigned char*curp;       //文件位置标记指针当前的指向
      unsigned istemp;          //临时文件指示器
      short token;              //用于有效性检查
}FILE;
```

C编译环境提供的stdio.h头文件中有文件信息结构体声明

10.1.1 文件的基本概念

- 文件信息结构体指针

```
FILE *fp;
//定义一个指向FILE类型数据的指针变量
```



可以使fp指向某一个文件的文件信息区，通过该文件信息区中的信息就能够访问该文件。也就是说，**通过文件指针变量能够找到与它关联的文件**。若有n个文件，应设置n个指针变量，分别指向n个FILE类型变量，以实现对n个文件的访问。为方便起见，通常将这种指向文件信息区的指针变量简称为**指向文件的指针变量**。

注意

- 指向文件的指针变量并不是指向外部介质上的数据文件的开头，而是指向内存中的文件信息区的开头。

10.1.2 打开与关闭文件

- 在对文件读写之前，应该“打开”该文件；在使用结束之后，应该“关闭”该文件
- 打开文件：为文件建立相应的信息区(用来存放有关文件的信息)和文件缓冲区(用来暂时存放输入输出的数据)，指定一个指针变量指向该文件，建立起指针变量与文件之间的联系
- 关闭文件：撤销文件信息区和文件缓冲区，使文件指针变量不再指向该文件，显然就无法进行对文件的读写了

用fopen函数打开数据文件

fopen(文件名，使用文件方式)；

在打开一个文件时，通知编译系统以下3个信息：

① 准备访问的文件的名字

② 使用文件的方式（“读”还是“写”等）

③ 让哪一个指针变量指向被打开的文件

注意：

① 如果不能实现“打开”的任务，fopen函数将会带回一个空指针值NULL

② 有12种文件使用方式，其中有6种是在第一个字母后面加了字母b的，表示二进制方式打开。没加b，原样读写数据；有加b，写入时\n转换为\r\n；读取时\r\n转换为\n

③ 程序中可以使用3个标准的流文件——标准输入流（stdin）、标准输出流（stdout）和标准出错输出流（stderr）。程序开始运行时系统会自动打开这3个标准流文件

```
FILE*fp;                                    //定义一个指向文件的指针变量fp
fp=fopen("a1","r");                         //将fopen函数的返回值赋给指针变量fp
if (fp == NULL) { printf("cannot open this file\n"); exit(0); }     //检测是否打开失败
```

使用文件方式

使用方式	含义	若文件不存在	若文件存在
“r”（只读）	为了输入数据，打开一个已存在的文本文件	出错	成功
“w”（只写）	为了输出数据，打开一个文本文件	建立新文件	删除并重新建立
“a”（追加）	向文本文件尾添加数据	出错	文件读写位置标记移到文件末尾
“rb”（只读）	为了输入数据，打开一个二进制文件	出错	成功
“wb”（只写）	为了输出数据，打开一个二进制文件	建立新文件	删除并重新建立
“ab”（追加）	向二进制文件尾添加数据	出错	文件读写位置标记移到文件末尾
“r+”（读写）	为了读和写，打开一个文本文件	出错	成功
“w+”（读写）	为了读和写，建立一个新的文本文件	建立新文件	删除并重新建立
“a+”(读写）	为了读和写，打开一个文本文件	出错	文件读写位置标记移到文件末尾
“rb+”（读写）	为了读和写，打开一个二进制文件	出错	成功
“wb+”（读写）	为了读和写，建立一个新的二进制文件	建立新文件	删除并重新建立
“ab+”（读写）	为读写打开一个二进制文件	出错	文件读写位置标记移到文件末尾

用fclose函数关闭数据文件

fclose(文件指针)；

```
fclose(fp);                    //关闭文件指针fp对应的文件
```

在使用完一个文件后应该关闭它，以防止它再被误用。“关闭”就是撤销文件信息区和文件缓冲区，使文件指针变量不再指向该文件，也就是文件指针变量与文件“脱钩”，此后不能再通过该指针对原来与其相联系的文件进行读写操作，除非再次打开，使该指针变量重新指向该文件。

如果不关闭文件就结束程序运行将会丢失数据。因为向文件写数据时，先将数据输出到缓冲区，待缓冲区充满后才正式输出给文件。当数据未充满缓冲区时程序结束运行，有可能使缓冲区中的数据丢失。fclose函数关闭文件，先把缓冲区中的数据输出到磁盘文件，再撤销文件信息区。

fclose函数也带回一个值，当成功地执行了关闭操作，则返回值为0；否则返回EOF(-1)。

10.1.3 顺序读写数据文件

从文件中读写字符、字符串

函数名	调用形式	功能	返回值
fgetc	fgetc(fp)	从fp指向的文件读入一个字符	读成功，带回所读的字符，失败则返回文件结束标志EOF(即-1)
fputc	fputc(ch,fp)	把字符ch写到文件指针变量fp所指向的文件中	写成功，返回值就是输出的字符；写失败，则返回EOF（即-1）
fgets	fgets(str,n,fp)	从fp指向的文件读入一个长度为(n-1)的字符串，存放到字符数组str中，在最后加一个'\0'	读成功，返回地址str，失败则返回NULL。在读完n - 1个字符之前遇到换行符\n或文件结束符EOF，读入即结束。与gets不同，fgets将所遇到的换行符\n也作为一个字符读入。
fputs	fputs(str,fp)	把str所指向的字符串写到文件指针变量fp所指向的文件中	写成功，返回0；写失败，则返回EOF（即-1）

从文件中读写字符串

fprintf(文件指针, 格式字符串, 输出表列);

若执行成功，则返回写入字符的个数，否则返回一个负数。

fscanf(文件指针, 格式字符串, 输出表列);

若执行成功，则返回读入的参数个数；执行失败，则返回EOF。

注：fgetc/fputc, fgets/fputs, fscanf/fprintf这几个函数，均是按ASCII码方式进行输入输出。

10.1.3 顺序读写数据文件

- 以二进制方式从文件读写一组数据
  - 用fread函数从文件中读一个数据块，用fwrite函数向文件写一个数据块。在读写时是以二进制形式进行的。
  - 在向磁盘写数据时，直接将内存中一组数据原封不动、不加转换地复制到磁盘文件上；在读入时，将磁盘文件中若干字节的内容批量读入内存。

```
fread(buffer, size, count, fp);
```

```
fwrite(buffer, size, count, fp);
```

两个函数的返回值是实际读取/写入的count数目。返回值小于指定的count值，则代表错误发生。

buffer:	是一个地址。对fread，它是用来存放从文件读入的数据的存储区的地址。对fwrite，是要把此地址开始的存储区中的数据向文件输出（以上指的是起始地址）。
size:	要读写的字节数。
count:	要读写多少个数据项(每个数据项长度为size)。
fp:	FILE类型指针。

```
float f[10];
fread(f,4,10,fp);        //从fp所指向的文件读入10个4个字节的数据，存储到数组f中
fwrite(f, 4, 10, fp);    //向fp所指向的文件写入10个4个字节的数据，数据来源于数组f
```

10.1.3 顺序读写数据文件

- 示例：以二进制形式进行存储的文件和以ASCII码形式进行存储的文件之间相互转换

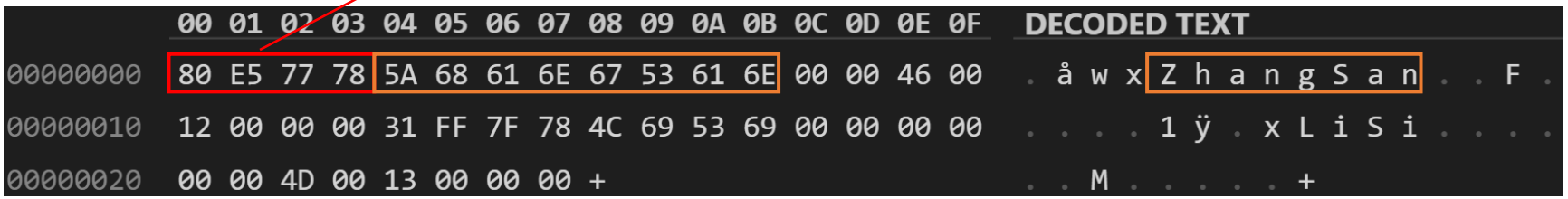
```
typedef struct {
    int sno;
    char name[10];
    char sex;
    int age;
} Student;
```

sno	name	sex	age
2021123456	ZhangSan	F	18
2021654321	LiSi	M	19

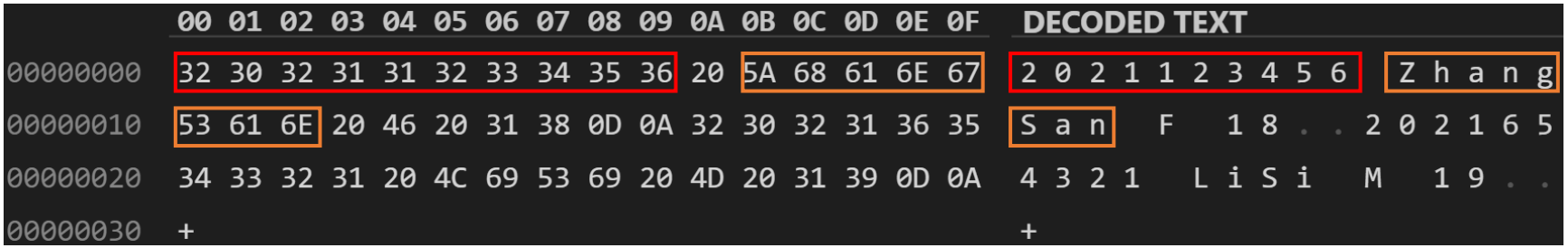
注：可在vscode中安装HexEditor插件查看二进制文件。

$(7877E580)_H = (2021123456)D$

student.bin



student.txt



10.1.3 顺序读写数据文件

- 示例：以二进制形式进行存储的文件和以ASCII码形式进行存储的文件之间相互转换

binary -> text

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. typedef struct { int sno; char name[10]; char sex; int age; } Student;    //学生数据类型

4. void main() {
5.   FILE *fpr, *fpw;
6.   Student stud;

7.   if ((fpr = fopen("student.bin", "rb")) == NULL) {       //以只读方式打开二进制文件
8.     printf("cannot open file\n"); exit(0);
9.   }
10.  if ((fpw = fopen("student.txt", "w+")) == NULL) {       //以读写方式创建文本文件
11.    printf("can create file\n"); exit(0);
12.  }

13.  while (fread(&stud, sizeof(Student), 1, fpr) == 1)      //读取二进制文件内容，以ASCII码形式写入文本文件
14.    fprintf(fpw, "%d %s %c %d\n", stud.sno, stud.name, stud.sex, stud.age);

15.  fclose(fpr);                                          //关闭读文件
16.  fclose(fpw);                                          //关闭写文件
17.}
```

10.1.3 顺序读写数据文件

- 示例：以二进制形式进行存储的文件和以ASCII码形式进行存储的文件之间相互转换

text-> binary

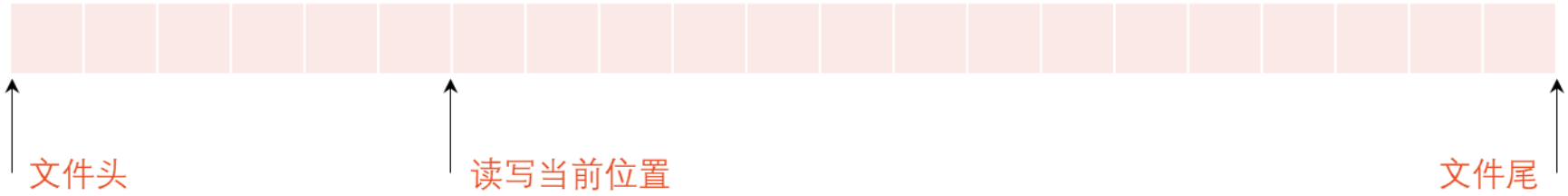
```
1. #include <stdio.h>
2. #include <stdlib.h>

3. typedef struct { int sno; char name[10]; char sex; int age; } Student;    //学生数据类型

4. void main() {
5.   FILE *fpr, *fpw;
6.   Student stud;

7.   if ((fpr = fopen("student.txt", "r")) == NULL) {       //以只读方式打开文本文件
8.     printf("can open file\n"); exit(0);
9.   }
10.  if ((fpw = fopen("student.bin", "wb+")) == NULL) {    //以读写方式创建二进制文件
11.    printf("can create file\n"); exit(0);
12.  }

13.  //读取文本文件内容，以二进制形式写入文件
14.  while (fscanf(fpr, "%d %s %c %d", &stud.sno, stud.name, &stud.sex, &stud.age) == 4)
15.    fwrite(&stud, sizeof(Student), 1, fpw);

16.  fclose(fpr);                                          //关闭读文件
17.  fclose(fpw);                                          //关闭写文件
18.}
```

10.1.4 随机读写数据文件

- 对文件进行顺序读写比较容易理解和操作，但有时效率不高
- 随机访问不是按数据在文件中的物理位置次序进行读写，而是可以对任何位置上的数据进行访问，比顺序访问效率高
- 随机读写时，需要先将文件位置标记定位在目标访问位置上，然后再进行读写。

文件位置标记

为了对读写进行控制，系统为每个文件设置了一个**文件位置标记**，用来指示“接下来要读写的下一个字符的位置”。一般情况下，在对字符文件进行顺序读写时，文件位置标记指向文件开头。对文件进行读/写操作，读/写完1个字符后，文件位置标记顺序向后移一个位置，在下一次执行读/写操作时，就将位置标记指向的字符进行读出或写入。依此类推，直到遇文件尾，此时文件位置标记在最后一个数据之后。



对流式文件既可以进行顺序读写，也可以进行随机读写。关键在于控制文件的位置标记。如果文件位置标记是按字节位置顺序移动的，就是顺序读写。如果能将文件位置标记按需要移动到任意位置，就可以实现随机读写。

所谓随机读写，是指读写完上一个字符（字节）后，不一定要读写其后续字符（字节），而可以读写文件中任意位置上所需要的字符（字节）。即对文件读写数据的顺序和数据在文件中的物理顺序一般是不一致的。可以在任何位置写入数据，在任何位置读取数据。

文件位置标记的定位

(1) 用rewind函数使文件位置标记指向文件开头 `rewind(文件指针);`

rewind函数的作用是使文件位置标记重新返回文件的开头，此函数没有返回值。

(2) 用fseek函数改变文件位置标记

`fseek(文件类型指针, 位移量, 起始点);`

起始点：0代表“文件开始位置”，1代表“当前位置”，2代表“文件末尾位置”

位移量：以“起始点”为基点，向前移动的字节数

fseek函数一般用于二进制文件。

```
fseek (fp,100L,0);          //将文件位置标记向前移到离文件开头100个字节处
fseek (fp,50L,1);           //将文件位置标记向前移到离当前位置50个字节处
fseek (fp,-10L,2);          //将文件位置标记从文件末尾处向后退10个字节
```

(3) 用ftell函数测定文件位置标记的当前位置

ftell函数的作用是得到流式文件中文件位置标记的当前位置，用相对于文件开头的位移量来表示。若调用函数时出错（如不存在fp指向的文件），返回值为-1L。

```
i=ftell(fp);                //变量i存放文件当前读写位置
if(i==-1L) printf("error\n");   //如果调用函数时出错，输出"error"
```

10.1.4 随机读写数据文件

- 示例：向磁盘文件中写入2个学生的数据，逆序读取学生信息。

```
1. #include<stdio.h>
2. #include <stdlib.h>

3. typedef struct { int sno; char name[10]; char sex; int age; } Student;                    //学生数据类型
4. Student studs[2] = {{2021123456, "ZhangSan", 'F', 18}, {2021654321, "LiSi", 'M', 19}}; //初始化结构体数组

5. void main() {
6.   FILE *fp;
7.   Student stud;

8.   if ((fp = fopen("student.bin", "wb+")) == NULL) {       //以读写方式建立二进制文件
9.     printf("cannot create file\n"); exit(0);
10.  }

11.  fwrite(studs, sizeof(Student), 2, fp);                  //将studs结构体的数据写入文件
12.  printf("cur offset: %d\n", ftell(fp));                  //输出文件读写指针当前的位置（相对于文件头的偏移量）

13.  fseek(fp, -sizeof(Student), 1);                        //移动文件位置标记至第一个数据块后
14.  fread(&stud, sizeof(Student), 1, fp);                  //读一个数据块到结构体变量
15.  printf("%10d %-10s %c %2d\n", stud.sno, stud.name, stud.sex, stud.age);    //在屏幕输出

16.  rewind(fp);                                            //将文件位置标记重新返回文件的开头
17.  fread(&stud, sizeof(Student), 1, fp);                  //读一个数据块到结构体变量
18.  printf("%10d %-10s %c %2d\n", stud.sno, stud.name, stud.sex, stud.age);    //在屏幕输出
19.}
```

10.1.5 文件读写的出错检测

1. ferror函数 `ferror(fp);`

在调用各种输入输出函数（如putc,getc,fread,fwrite等）时，如果出现错误，除了函数返回值有所反映外，还可以用ferror函数检查。

如果ferror返回值为0（假），表示未出错;

如果返回一个非零值，表示出错。

注意

对同一个文件每一次调用输入输出函数，都会产生一个新的ferror函数值，因此，应当在调用一个输入输出函数后立即检查ferror函数的值，否则信息会丢失。

在执行fopen函数时，ferror函数的初始值自动置为0。

2. clearerr函数

clearerr的作用是使文件出错标志和文件结束标志置为0。

假设在调用一个输入输出函数时出现错误，ferror函数值为一个非零值。应该立即调用clearerr(fp)，使ferror(fp)的值变成0，以便再进行下一次的检测。

只要出现文件读写出错标志，它就一直保留，直到对同一文件调用clearerr函数或rewind函数，或任何其他一个输入输出函数。

YOJ平台大数据量的输入输出如何解决?

```
1.   scanf("%d %d", &m, &n);
2.   for (i = 1; i <= n; i++)
3.     scanf("%d %d", &players[i].value, &players[i].endurance);
4.   /* solve the problem */
5.   printf("%lld\n%d\n", maxval, count);
6.   printf("%d %d %d %d %d %d\n", init[0], init[1], init[2], init[3], init[4], init[5]);
7.   for (i = 0; i < count; i++)
8.     printf("%d %d %d\n", substitutions[i].time, substitutions[i].down, substitutions[i].up);
```

```
1.   FILE *fpin, *fpout;
2.   fpin = fopen("hokej1.in", "r");
3.   fscanf(fpin, "%d %d", &m, &n);
4.   for (i = 1; i <= n; i++)
5.     fscanf(fpin, "%d %d", &players[i].value, &players[i].endurance);
6.   fclose(fpin);
7.   /* solve the problem */
8.   fpout = fopen("hokej1.out", "w+");
9.   fprintf(fpout, "%lld\n%d\n", maxval, count);
10.  fprintf(fpout, "%d %d %d %d %d %d\n", init[0], init[1], init[2], init[3], init[4], init[5]);
11.  for (i = 0; i < count; i++)
12.    fprintf(fpout, "%d %d %d\n", substitutions[i].time, substitutions[i].down, substitutions[i].up);
13.  fclose(fpout);
```

引子：mystdio工程项目

大作业mystdio工程项目的要求：

- myscanf函数写在myscanf.c文件，myprintf函数写在myprintf.c文件，main函数写在main.c文件；
- 在头文件mystdio.h中声明myscanf和myprintf函数， 在main.c文件里包含mystdio.h头文件；
- 将myscanf.c编译成myscanf.o，myprintf.c编译成myprintf.o，整合成一个动态链接库libmystdio.so；
- 将main.c编译成可执行程序mystdio，与libmystdio.so进行动态链接。
- 编写Makefile文件，完成自动化编译链接。

10.2 工程

- Linux命令
- 程序的生成和运行
- 自动化编译

10.2.1 Linux命令

- Linux命令的格式

command options arguments

命令 选项 参数

- 命令：命令的名字（能体现命令的功能）
- 选项：进一步的情况说明（不同选项对应不同的处理逻辑）
- 参数：待处理的对象（一般为文件路径）

例如：rm -rf /home/student/project

功能：无提示地强制递归删除/home/student/project目录下的所有文件和文件夹

选项：-rf recursively forcely

参数：待操作的文件路径

10.2.1 Linux命令

1：文件管理

- ls命令 – 显示指定工作目录下的内容及属性信息
- cp命令 – 复制文件或目录
- mkdir命令 – 创建目录
- mv命令 – 移动或改名文件
- pwd命令 – 显示当前路径

2：文档编辑

- cat命令 – 在终端设备上显示文件内容
- echo命令 – 输出字符串或提取Shell变量的值
- rm命令 – 移除文件或目录
- tail命令 – 查看文件尾部内容
- grep命令 – 强大的文本搜索工具

3：系统管理

- rpm命令 – RPM软件包管理器
- find命令 – 查找和搜索文件
- startx命令 – 初始化X-windows
- uname命令 – 显示系统信息
- resize2fs命令 – 调整文件系统大小

4：磁盘管理

- df命令 – 显示磁盘空间使用情况
- fdisk命令 – 磁盘分区
- lsblk命令 – 查看系统的磁盘
- hdparm命令 – 显示与设定硬盘参数
- vgextend命令 – 扩展卷组

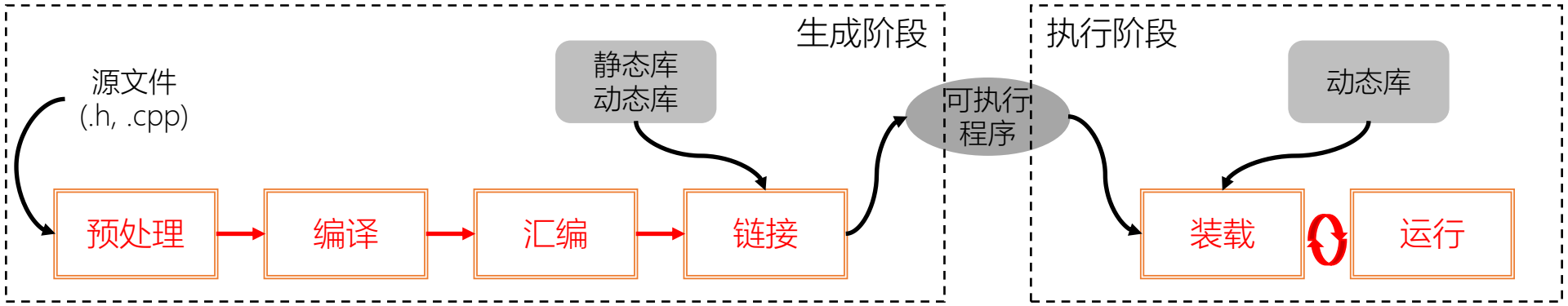
5：文件传输

- tftp命令 – 上传及下载文件
- curl命令 – 文件传输工具
- fsck命令 – 检查并修复Linux文件系统
- ftpwho命令 – 显示ftp会话信息
- lprm命令 – 删除打印队列中的打印任务

……

更多内容： https://www.runoob.com/linux/linux-command-manual.html

10.2.2 程序的生成和执行

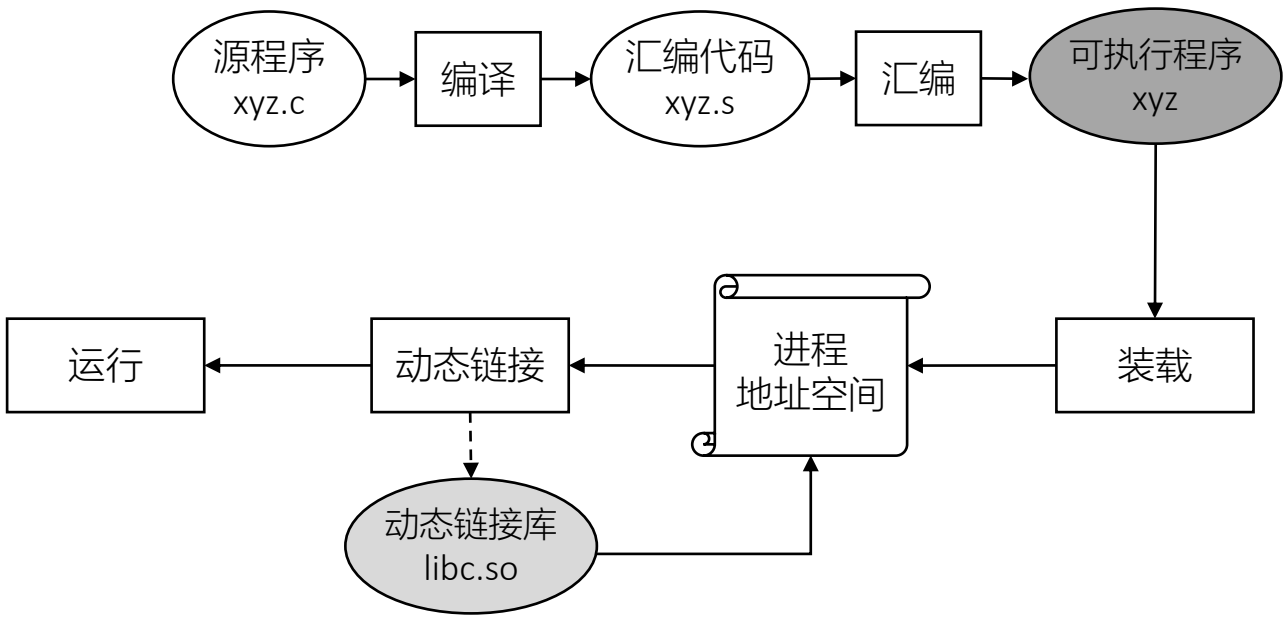


链接分为静态链接和动态链接：静态链接在生成阶段就把静态库加入到可执行程序中；动态链接在生成阶段只在可执行程序中加入一些描述信息。动态链接可细分为装载时链接和运行时链接：装载时链接在装载时将动态库导入内存空间，运行时链接在运行时将动态库导入内存空间。

- 预处理：展开宏定义、包含头文件、处理条件编译，生成预处理文件
- 编译：词法分析/语法分析/语义分析，生成汇编代码文件
- 汇编：将汇编代码逐句翻译为机器代码，生成目标文件
- 链接：将所有目标文件和库文件进行组合，生成一个可执行文件
- 装载：将可执行程序和所依赖的动态库装载进内存空间
- 运行：开始运行程序逻辑

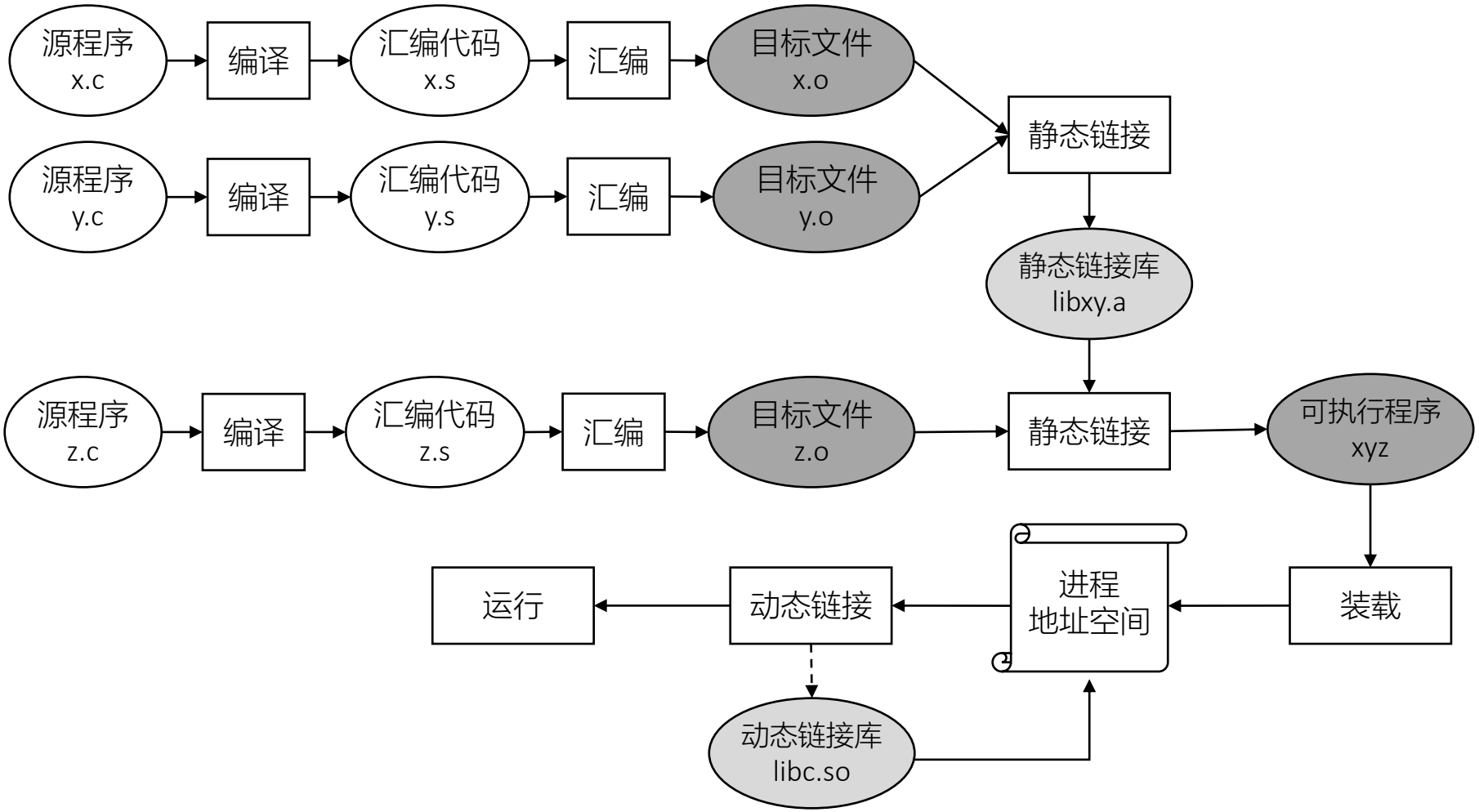
10.2.2 程序的生成和运行

- 单个文件的编译/链接/运行



10.2.2 程序的生成和运行

- 多个文件的编译/链接/运行



编译

- YOJ的默认编译选项：

  g++ program.cpp -o program -O2 -lm -mx32 -std=c++03

  - g++：使用的编译器是g++（一般用g++编译.cpp文件，gcc编译.c文件）
  - program.cpp：待编译的源代码文件名是program.cpp
  - -o program：生成可执行程序的文件名是program
  - -O2：优化强度（其他的优化强度有O0, O1, O3, Os, Ofast, Og等）
  - -lm：链接libm.so库（数学函数）
  - -mx32：在64位系统中，设置指针和long都是32位的
  - -std=c++03：使用c++03标准

- 更多编译选项参见：
  - 在终端运行man gcc
  - 访问https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

扩展：可以使用条件编译，
在调试时从文件读写输入输出，
提交时从标准输入输出读写数据

```c
#define DEBUG //comment this line when submit

int main()
{
#ifdef DEBUG
    FILE *fpin = fopen("test1.in", "r");
    FILE *fpout = fopen("test1.out", "w");
    fscanf(fpin, "%d", &n);
    fprintf(fpout, "%d", n);
    fclose(fpin);
    fclose(fpout);
#else
    scanf("%d", &n);
    printf("%d", n);
#endif
}
```

编译

- 条件编译
  - 一般情况下，源程序每行非注释代码都会被编译，
  - 有些情况下，希望只对其中一部分内容进行编译
  - 在程序中加上条件，让编译器只对满足条件的代码进行编译，将不满足条件的代码舍弃，这就是条件编译
- 示例

```c
void main() {
   gets(in_fmt);
   gets(out_fmt);
   gets(in_buf);

#ifdef TEST_STRING
   myscanf(in_fmt, str);
   myprintf(out_fmt, str);
#elif TEST_INTEGER
   myscanf(in_fmt, &d);
   myprintf(out_fmt, d);
#endif

   puts(out_buf);
}
```

gcc -DTEST_STRING -o main.o main.c

```c
void main() {
   gets(in_fmt);
   gets(out_fmt);
   gets(in_buf);

   myscanf(in_fmt, str);
   myprintf(out_fmt, str);

   puts(out_buf);
}
```

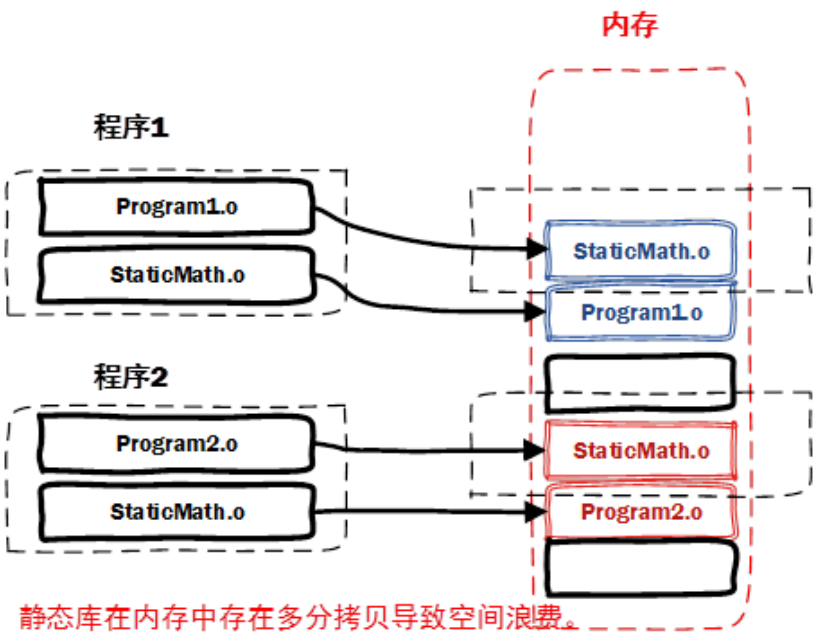
gcc -DTEST_INTEGER -o main.o main.c

```c
void main() {
   gets(in_fmt);
   gets(out_fmt);
   gets(in_buf);

   myscanf(in_fmt, &d);
   myprintf(out_fmt, d);

   puts(out_buf);
}
```

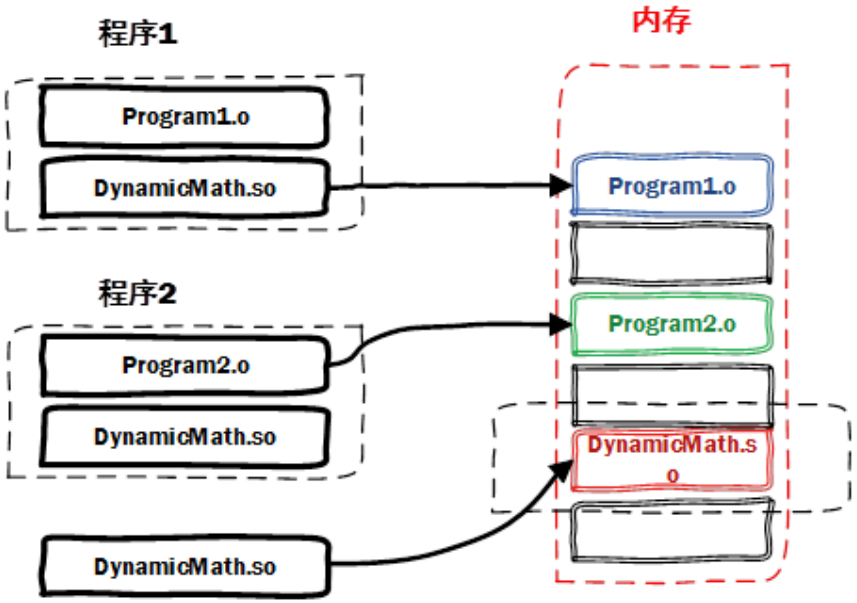
链接

- 静态链接
  - 在链接时，由链接器将库的内容加入到可执行程序中
  - 代码装载速度快，执行速度略比动态链接库快
  - 只需保证开发者的机器中有正确的库文件，无需考虑用户机器上的情况
  - 生成的可执行文件体积较大，包含相同的公共代码，造成浪费
- 动态链接
  - 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
  - 生成的可执行文件较静态链接生成的可执行文件小
  - 执行速度比静态链接稍慢
  - 要求用户机器上预装有所依赖的库

链接



静态库在内存中存在多分拷贝导致空间浪费。
假如，静态库占用**1M**内存，有**2000**个这样的程序，将占用近**2GB**的空间~~~~~



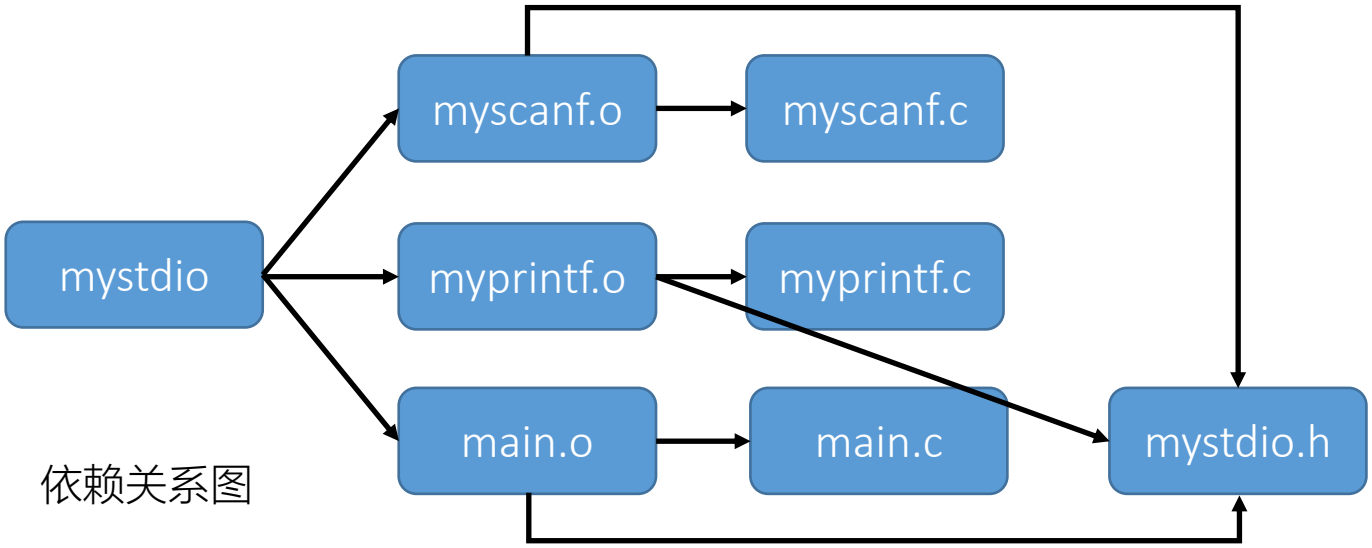
动态库在内存中只存在一份拷贝，避免了静态库浪费空间的问题。

静态链接　v.s.　动态链接

链接

- 方式1：将myscanf、myprintf编译成目标文件，进行静态链接

```
gcc -c myscanf.c                                  #编译生成myscanf.o
gcc -c myprintf.c                                 #编译生成myprintf.o
gcc -c main.c                                     #编译生成main.o
gcc -o mystdio main.o myscanf.o myprintf.o        #静态链接生成mystdio
```

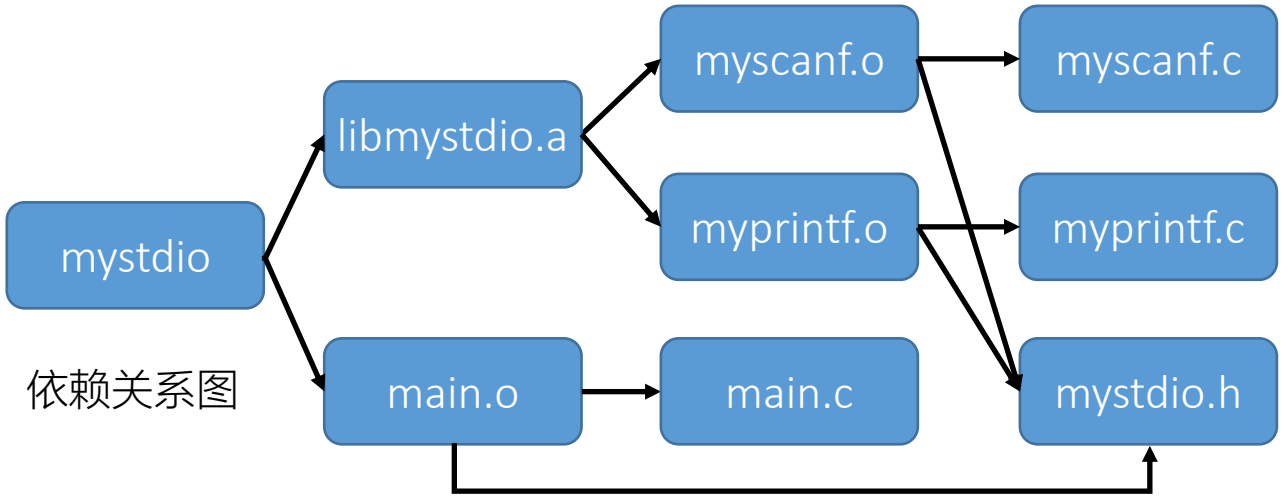


依赖关系图

链接

- 方式2：将myscanf，myprintf编译成静态链接库，进行静态链接

```
gcc -c myscanf.c                                    #编译生成myscanf.o
gcc -c myprintf.c                                   #编译生成myprintf.o
gcc -c main.c                                       #编译生成main.o
ar cr libmystdio.a myscanf.o myprintf.o             #生成静态链接库
gcc -o mystdio main.o -L. -lmystdio -Wl,-rpath=.  #静态链接生成mystdio
```

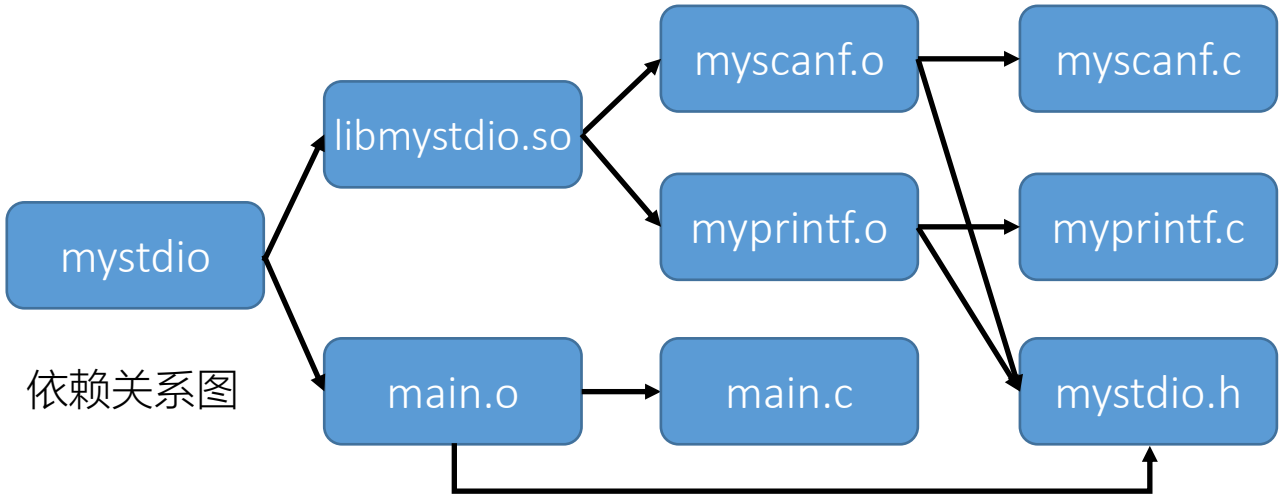


依赖关系图

链接

- 方式3：将myscanf，myprintf编译成动态链接库，进行动态链接

```
gcc -fPIC -c myscanf.c                          #编译生成myscanf.o
gcc -fPIC -c myprintf.c                         #编译生成myprintf.o
gcc -fPIC -c main.c                             #编译生成main.o
gcc -shared -o libmystdio.so myscanf.o myprintf.o    #生成动态链接库
gcc -o mystdio main.c -L. -lmystdio -Wl,-rpath=.     #动态链接生成
```



依赖关系图

扩展：在调试时，可以用通过输入输出重定向的方式从文件读写数据。

```
student@student-VirtualBox:~/tmp$ ./hojek <hojek1.in >hojek1.out
student@student-VirtualBox:~/tmp$ ls
hojek  hojek1.in  hojek1.out
```

运行

- 运行存放在/home/student目录下的可执行程序mytest
  - 方式1：/home/student/mytest（绝对路径运行）
  - 方式2：cd /home/student（进入/home/student目录）
    ./mytest（运行当前目录下的mytest）

```
PS C:\Users\youwe> cd "c:\Users\youwe\Documents\" ; if ($?) {
gcc -O2 mytest.c -o mytest } ; if ($?) { .\mytest }
```

- 输入输出重定向
  - 一般情况下，我们都是从键盘读取用户输入的数据给程序（标准输入），然后将程序运行产生的数据呈现到显示器上（标准输出）
  - 可以通过输入重定向，使得标准输入的数据从指定文件中读取；同样地，可以通过输出重定向，使得标准输出的数据写到指定文件中

执行mytest，输入重定向到input_file文件，输出重定向到output_file文件：

./mytest <input_file >output_file

更多内容： https://www.runoob.com/linux/linux-shell-io-redirections.html

运行

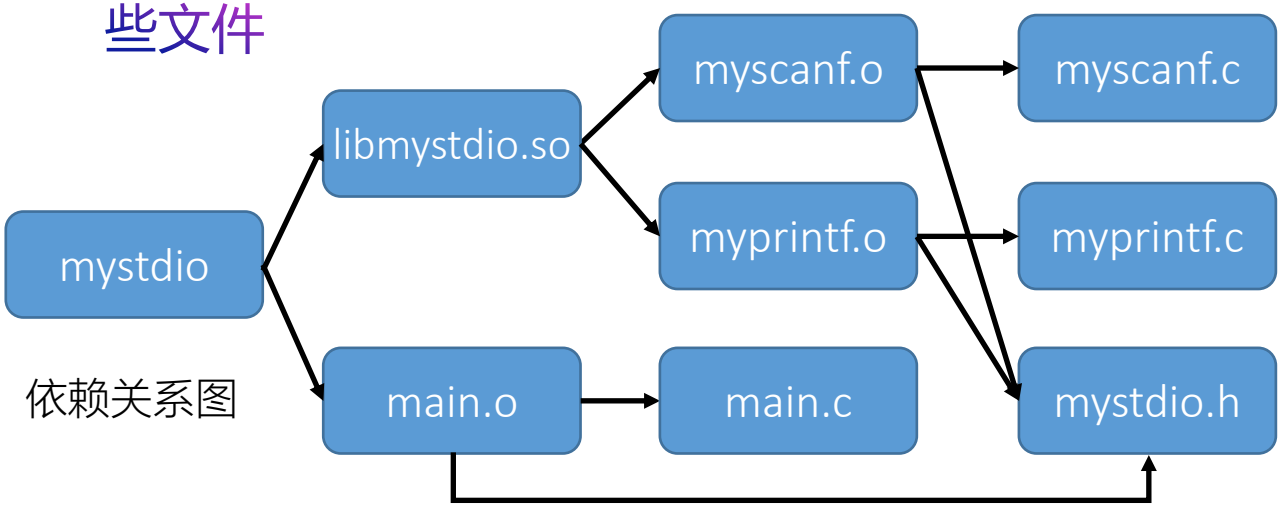
- 命令行参数
  - main函数的完整原型：int main(int argc, char* argv[], char* envp[])
  - argc是执行程序时的命令行参数个数，程序名本身也算一个命令行参数
  - argv是命令行中参数的具体值
  - envp是环境变量列表
- 示例：

运行：./cryptography -e -t plain.txt cipher.txt 128

```
1. const char *action, *type;   //动作：-e(encrypt)/-d(decrypt)；类型：-t(text)/-b(binary)
2. const char *input, *output; //输入输出文件名
3. unsigned char key;           //密钥

4. void main(int argc, char *argv[]) {
5.    if (argc != 6) { printf("Error\n"); exit(-1); }
6.    action = argv[1];
7.    type = argv[2];
8.    input = argv[3];
9.    output= argv[4];
10.   key = atoi(argv[5]);
11.   ……
12.}
```

10.2.3 自动化编译

- GNU make是一个命令工具，是一个用来控制软件构建过程的自动化管理工具。Make工具通过Makefile文件来完成并自动维护编译和链接工作
- Makefile用于自动编译和链接。一个工程有很多文件组成，每一个文件的改变都会导致工程的重新链接，但不是所有文件都要重新编译。Makefile中记录有文件的依赖信息，在make时决定需要重新编译哪些文件



依赖关系图

思考：

- main.c发生改变时，哪些文件需要重新编译链接？
- myscanf.c发生改变呢？
- mystdio.h发生改变呢？

10.2.3 自动化编译

- Makefile由一些列规则构成，每条规则形如

```
target: prerequisites
        command
```

  - target：本条规则生成的目标
  - prerequisites：生成target所需要依赖的其他目标
  - command：为了生成target所要执行的命令

- 在make命令执行时
  - 读取Makefile文件，分析所有规则
  - 为所有的目标创建依赖关系链
  - 根据依赖关系，决定哪些目标要重新生成
  - 执行生成命令

更多内容： https://seisman.github.io/how-to-write-makefile/Makefile.pdf

10.2.3 自动化编译

- 方式1：将myscanf、myprintf编译成目标文件，进行静态链接

  对应的Makefile

```
1. mystdio: main.o myscanf.o myprintf.o
2.         gcc -o mystdio main.o myscanf.o myprintf.o
3. main.o: main.c mystdio.h
4.         gcc -c main.c
5. myscanf.o: myscanf.c mystdio.h
6.         gcc -c myscanf.c
7. myprintf.o: myprintf.c mystdio.h
8.         gcc -c myprintf.c
9. clean:
10.        rm -rf mystdio *.o *.a *.so
```

10.2.3 自动化编译

- 方式2：将myscanf，myprintf编译成静态链接库，进行静态链接

  对应的Makefile

```
1. mystdio: main.o libmystdio.a
2.          gcc -o mystdio main.o -L. -lmystdio -Wl,-rpath=.
3. libmystdio.a: myscanf.o myprintf.o
4.          ar cr libmystdio.a myscanf.o myprintf.o
5. main.o: main.c mystdio.h
6.          gcc -c main.c
7. myscanf.o: myscanf.c mystdio.h
8.          gcc -c myscanf.c
9. myprintf.o: myprintf.c mystdio.h
10.         gcc -c myprintf.c
11.clean:
12.         rm -rf mystdio *.o *.a *.so
```

10.2.3 自动化编译

- 方式3：将myscanf，myprintf编译成动态链接库，进行动态链接

对应的Makefile

```
1. mystdio: main.o libmystdio.so
2.          gcc -o mystdio main.o -L. -lmystdio -Wl,-rpath=.
3. libmystdio.so: myscanf.o myprintf.o
4.          gcc -shared -o libmystdio.so myscanf.o myprintf.o
5. main.o: main.c mystdio.h
6.          gcc -fPIC -c main.c
7. myscanf.o: myscanf.c mystdio.h
8.          gcc -fPIC -c myscanf.c
9. myprintf.o: myprintf.c mystdio.h
10.         gcc -fPIC -c myprintf.c
11.clean:
12.         rm -rf mystdio *.o *.a *.so
```

本次mystdio工程使用这种方式进行