目录

C I/O

Program input

There are 3 main sources of input for programs:

1. from the command line
    - you get access to data on the command line by using *argc* and *argv[][]*
2. from standard input (also called *stdin*)
    - *stdin* can be the keyboard, a data file, or the output of another program
3. from an 'internally-defined' file
    - open a file, use *fscanf()*, and don't forget to close the file

1. command line

To read from the command line:

- include *argc* and *argv* in your parameter list for *main()*.
- use *sscanf()* to read the arguments (it stands for '**s**tring scanf()')
    - **the first argument of a *sscanf()* is a string**

Here is a program that counts from *1* to *num*, where *num* is provided by the user on the command line

切换行号显示

```
// countc.c
// reads an integer from the command line and counts
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   int num = 0;
   if (argc < 2 || sscanf(argv[1], "%d", &num) != 1) { // num is defined here
      fprintf(stderr,"Usage: %s integer\n", argv[0]);
      return EXIT_FAILURE;
   }
```

```
   for (int i=1; i<=num; i++) {
      printf("%d ", i);
   }
   printf("\n");
   return EXIT_SUCCESS;
}
```

Notice the program prints a 'Usage' message if an integer argument is missing (discussed in next session)

To execute the program:

```
prompt$ dcc -o countc countc.c

prompt$ ./count
Usage: ./countc integer

prompt$ ./countc !t#q
Usage: ./countc integer

prompt$ ./countc 10
1 2 3 4 5 6 7 8 9 10
```

2. standard input

To read from standard input (usually called simply *stdin*)

- a *scanf()* is used (instead of *sscanf()*)
  - **a *scanf()* misses the string argument of a *sscanf()***
- *so where does num comes from?*
  - ... the default 'channel' *stdin*

切换行号显示

```
//counts.c
// reads an integer from stdin and counts
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int num;
   if (scanf("%d", &num) != 1) {
      fprintf(stderr, "Usage: a number expected\n");
      return EXIT_FAILURE;
   }
   // the rest of the program is exactly the same as the command-line version
   for (int i=1; i<=num; i++) {
      printf("%d ",i);
   }
   printf("\n");
   return EXIT_SUCCESS;
}
```

Notice the *Usage* message this time is simpler than the command-line version above

- ... because we did not declare *argc* and *argv*, and so cannot use *argv[0]* this time!!

  - we could have if we wanted to of course

There are many ways to 'test' a program that reads *stdin*.

1. Using the keyboard

   ```
   prompt$ dcc -o counts counts.c
   prompt$ ./counts
   10
   1 2 3 4 5 6 7 8 9 10
   ```

   where the integer *10* was typed on the keyboard by the user, and the program generates the count from *1* to *10*.

2. Using a data file, *input.txt* say, which contains the integer *10* (followed by a newline).

   ```
   prompt$ more input.txt
   10

   prompt$ ./counts < input.txt
   1 2 3 4 5 6 7 8 9 10
   ```

3. Using a *pipe* command. A pipe command joins the *stdout* of a program to the *stdin* of another program. If we have a program called *write10.c*:

   切换行号显示

   ```
   // write10.c
   // just print the string 10
   #include <stdio.h>
   #include <stdlib.h>

   int main(void) {
      printf("10\n");
      return EXIT_SUCCESS;
   }
   ```

   then we can pipe its *stdout* to the *stdin* of our counting program

   ```
   prompt$ dcc -o write10 write10.c
   prompt$ dcc -o counts counts.c
   prompt$ ./write10 | ./counts
   1 2 3 4 5 6 7 8 9 10
   ```

   But you can actually generate a string much more easily in UNIX using *echo*

   ```
   prompt$ echo "10" | ./counts
   1 2 3 4 5 6 7 8 9 10
   ```

Some people prefer to use *getchar()* to read from *stdin*

切换行号显示

```
// echostdin.c

#include <stdio.h>
```

```
#include <stdlib.h>
int main(int argc, char* argv[]) {
  char c = getchar(); // get a char from stdin
  while (c != '\n') {
      printf("%c", c);
      c = getchar();
  }
  putchar('\n');
  return EXIT_SUCCESS;
}
```

```
prompt$ dcc echostdin.c
prompt$ echo bornfree | ./a.out
bornfree
prompt$ ./a.out
bornfree
bornfree
prompt%
```

where the first 'born free' the user typed in, and the second is the echo.

User prompting

You can still use a 'user prompt' when you use *stdin* but it messes up the output.

切换行号显示

```
// counts+.c
// reads an integer from stdin and counts
// prompts the user
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int num;
   printf("Please input a number: "); // this line added to counts.c
   if (scanf("%d", &num) != 1) {
      fprintf(stderr, "Usage: a number expected\n");
      return EXIT_FAILURE;
   }
   for (int i=1; i<=num; i++) {
      printf("%d ",i);
   }
   printf("\n");
   return EXIT_SUCCESS;
}
```

results in

```
prompt$ ./counts+
Please input a number: 10
1 2 3 4 5 6 7 8 9 10
```

where the program prints the user prompt, the user types in *10*, and the program then outputs the count to *10*.

- *that looks fine*

If you instead use a pipe as input, then you do not see what the input is

```
prompt$ echo "10" | ./counts+
Please input a number: 1 2 3 4 5 6 7 8 9 10
```

You see here that the *10* generated by the *echo* does not appear on the screen: you just see the output of the program

- *which is sort-of messed up*

> User prompts are not used often in UNIX because:
>
> 1. the *UNIX way* is to use command line arguments
> 2. it doesn't fit well into *stdin*/*stdout* framework (as we saw above)

3. a user file

A program can open and close, and read from, and write to, a file that is defined by the user

This is generally done when you have

- large volumes of stored data, or
- complex data (such as structs) or
- non-printable data

These don't happen often. Nevertheless, for the sake of completeness, here is a program that

- reads a number from a file *input.txt*
- writes the count from 1 to that number to the file *output.txt*
    - it is *user-friendly* : it tells the user that an output file has been created

切换行号显示

```
// files.c
// read a number 'num' from a file input.txt
// write a count from 1 to 'num' to the file OUT

#define IN  "input.txt"
#define OUT "output.txt"

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

#define NUMDIG 6 // size of numerical strings that are output

int main(void) {
   FILE *fpi, *fpo; // these are file pointers
   char s[NUMDIG];

   fpi = fopen(IN, "r");
   if (fpi == NULL) { // an important check
```

```
        fprintf(stderr, "Can't open %s\n", IN);
        return EXIT_FAILURE;
    }
    else {
        int num;
        if (fscanf(fpi, "%d", &num) != 1) { // an important check
            fprintf(stderr, "No number found in %s\n", IN);
            return EXIT_FAILURE;
        }
        else {
            fclose(fpi); // don't need the input file anymore
            fpo = fopen(OUT, "w");
            if (fpo == NULL) { // an important check
                fprintf(stderr, "Can't create %s!\n", OUT);
                return EXIT_FAILURE;
            }
            else { // got input and got an output file
                fprintf(fpo, "%s", "Counts\n");
                for (int i=1; i<=num; i++) {
                    sprintf(s, "%d", i);
                    fprintf(fpo, "%s\n", s);
                }
                fclose(fpo);
                printf("file %s created\n", OUT);
                return EXIT_SUCCESS;
            }
        }
    }
}
```

Notice: 布告

- all the error messages go to *stderr*
- the 'file is created' message goes to *stdout*
- read is done using *fscanf()*, and write using *fprintf()*
- as it is written the user <u>must know</u> that input and output files are used
  - ... could be re-written to prompt the user for the file names

If you create a data file *input.txt* that contains the string *13*, then you compile and execute the program

```
prompt$ dcc files.c
prompt$ ./a.out
file output.txt created
prompt$ more output.txt
Counts
1
2
3
4
5
6
7
8
9
10
11
12
13
```

If the input text file does not exist:

```
prompt$ ./a.out
Can't open input.txt
```

and it is for the user to figure out what that means 🙁

File I/O requires care in programming

- more housekeeping
- more difficult to maintain

You need to have a good reason to use files instead of using *stdin/stdout*

Program output

There are two standard output 'streams'

- *stdout*
- *stderr*

Both are normally defined to be the screen

The general form for a print statement is

切换行号显示

```
fprintf(stream, ...)
```

where 'stream' can be *stdout*, *stderr* or a user-defined file. Note

- the call *printf(...)* is the same as *fprintf(stdout, ...)*
- the 'stream' can be a user-defined file pointer
- a *fprintf(stderr, ...)* is usually reserved for serious errors
  - you may ask *is a 'Usage' message a 'serious error'?*
  - or ask *is incorrect input a 'serious error'?*
  - but it is clear that
    - a file that cannot be opened is a serious error
    - a string that cannot be read is a serious error

Note the 'systematic' naming:

- standard input is *scanf()*,
  - if you read from a string then use **s***scanf()*, where the first argument is the string
- standard output is *printf()*,
  - if you write to a file then use **f***printf()*, where the first argument is a stream

Like *stdin*, we can re-direct *stdout* to a file. For example:

```
dcc -o counts counts.c
```

```
./counts > output.txt
10
```

(where the integer 10 is input by the user) will result in the count from 1 to 10 going to the file *output.txt*

If you create a data file *input.txt* that contains the string *10*, then the following will generate the same output text file

```
./counts < input.txt > output.txt
```

As we saw before, you can let *echo* generate data and use that in a pipe. This also generates the same output text file.

```
echo "10" | ./counts > output.txt
```

Input/output: in summary

The vast majority of programs can be written just using these library I/O calls

- *scanf()* to read from *stdin*
- *sscanf()* to read from the command line
- *printf()* to write to *stdout*
- *fprintf()* to write to *stderr*

Testing can be controlled by shell scripts that execute programs with stdin coming from the script itself or data files

Lec01IO (2019-06-17 17:54:09由AlbertNymeyer编辑)