C Input and Output

EECS 2031

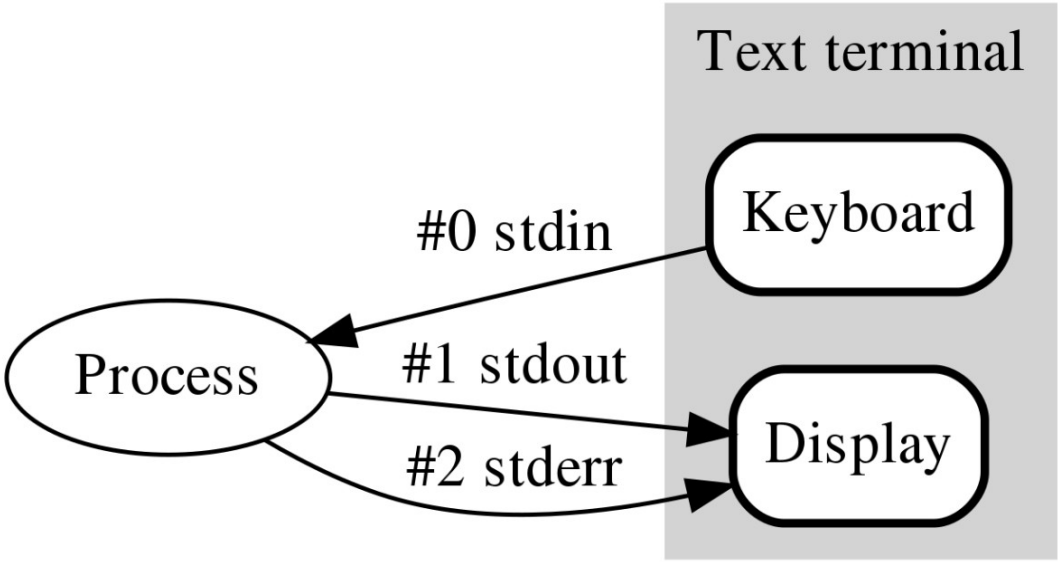
Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

stdin, stdout, stderr

- When your C program begins to execute, three input/output devices are opened automatically.
- stdin
  - The “standard input” device, usually your keyboard
- stdout
  - The “standard output” device, usually your monitor
- stderr
  - The “standard error” device, usually your monitor



Formatted Console Output

- In C formatted output is created using the printf( ) function.

- printf( ) outputs text to stdout

- The basic function call to printf( ) is of the form
  printf(format, arg1, arg2, …);

  where the format is a string containing
  - conversion specifications
  - literals to be printed

printf( ) conversions

Conversions specifications begin with **%** and **end with a conversion character**.

Between the % and the conversion character MAY be, in order

- A minus sign specifying left-justification
- The minimum field width
- A period separating the field width and precision
- The precision that specifies
  - Maximum characters for a string
  - Number of digits after the decimal for a floating point
  - Minimum number of digits for an integer
- An h for “short” or an l (letter ell) for long

Common printf( ) Conversions

- %d -- the int argument is printed as a decimal number
- %u -- the int argument is printed as an unsigned number
- %s -- prints characters from the string until ‘\0’ is seen or the number of characters in the (optional) precision have been printed (more on this later)
- %f -- the double argument is printed as a floating point number
- %x, %X -- the int argument is printed as a hexadecimal number (without the usual leading “0x”)
- %c - the int argument is printed as a single character

Output formatting

%(flags)(width)(.precision)specifier

- **%2d**: outputs a decimal (integer) number that fills at least 2 character spaces, padded with empty space. E.g.: 5 → " 5", 120 → "120"

- **%8.4f**: outputs a floating point number that fills at least 8 character spaces (including the decimal separator), with exactly 4 digits after the ".", padded with empty space.

Integer formatting

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7d", myInt);` `Value:     301`

flags	-: Left aligns the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+d", myInt);`
		`+301`
		
		`printf("%08d", myInt);`
		`00000301`
		
		`printf("%+08d", myInt);`
		`+0000301`

Floating-point formatting

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7.2f", myFloat);` `Value:   12.34`

.precision	Specifies the number of digits to print following the decimal point. If the precision is not specified, a default precision of 6 is used.	`printf("%.4f", myFloat);`
		`12.3400`
		
		`printf("%3.4e", myFloat);`
		`1.2340e+01`

flags	-: Left aligns the output given the specified width, padding the output with spaces.
	+: Prints a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+f", myFloat);`
		`+12.340000`
		`printf("%08.2f", myFloat);`
		`00012.34`

String formatting

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the string has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces.	`printf("%20s String", myString);` `Formatting String`
.precision	Specifies the maximum number of characters to print. If the string has more characters than the precision, the string will be truncated.	`printf("%.6s", myString);` `Format`
flags	-: Left aligns the output given the specified width, padding the output with spaces.	`printf("%-20s String", myString);` `Formatting String`

Formatted Output Example

- Use field widths to align output in columns

```
int i;
for (i = 1 ; i < 5; i++)
  printf("%2d %10.6f %20.15f\n", i,sqrt(i),sqrt(i));
```

```
12 1234567890 12345678901234567890
 1   1.000000    1.000000000000000
 2   1.414214    1.414213562373095
 3   1.732051    1.732050807568877
 4   2.000000    2.000000000000000
```

Unix input redirection

- By default, stdin is associated with the user's keyboard, but Unix allows us to redirect stdin to read data from a file when your program is executed. All scanf( ) statements in your program read from this file instead of the user's keyboard with no change to your code.
- Redirecting input from a file is useful for debugging -- you don't have to continually retype your input.
- Suppose your program's name is Project1 and you wish to get your input from a file named data1. To redirect stdin to read from data1, use this command at the Unix prompt

indigo 346 % Project1 < data1

Unix output redirection

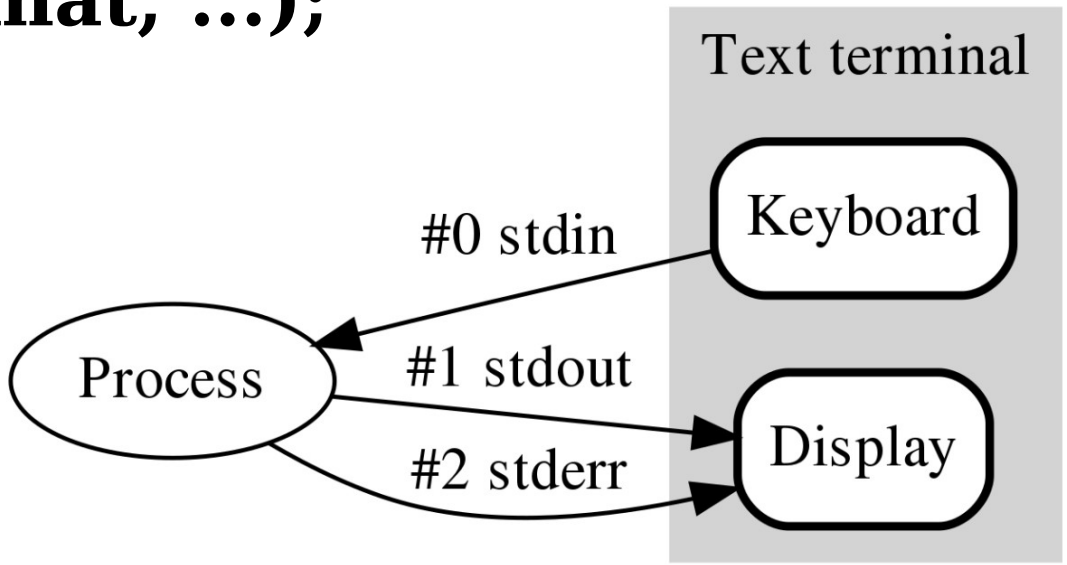
- By default, stdout is associated with the user's console, but Unix allows us to redirect stdout to output text to a file when your program is executed. All printf( ) statements in your program output to this file instead of the user's console, otherwise your program is unaffected.
- Suppose your program's name is Project1 and you wish to write your output to a file named logfile1. To redirect stdout to write to logfile1, use this command at the Unix prompt

indigo 346 % Project1 > logfile

- Can you redirect both input and output?

int fprintf(FILE *stream, const char *format, ...)

int dprintf(int fd, const char *format, ...);




```
fprintf(stdout, "a regular message on stdout\
n");
fprintf(stderr, “an error message on stderr\n");

dprintf(1, "a regular message on stdout\n");
```

```c
#include <stdio.h>
int main(void)
{
    printf("A regular message on stdout\n");
    /* Using streams with fprintf() */
    fprintf(stdout, "Also a regular message on stdout\n");
    fprintf(stderr, "An error message on stderr\n");

    dprintf(1, "A regular message, printed to "
           "fd 1\n");
    dprintf(2, "An error message, printed to "
            "fd 2\n");
    return 0;
}
```

```
$> ./output
A regular message on stdout
Also a regular message on stdout
An error message on stderr
A regular message, printed to fd 1
An error message, printed to fd 2
```

```c
#include <stdio.h>
int main(void)
{
    printf("A regular message on stdout\n");
    /* Using streams with fprintf() */
    fprintf(stdout, "Also a regular message on stdout\n");
    fprintf(stderr, "An error message on stderr\n");

    dprintf(1, "A regular message, printed to "
           "fd 1\n");
    dprintf(2, "An error message, printed to "
            "fd 2\n");
    return 0;
}
```

```
$> ./output 2> error.log
A regular message on stdout
Also a regular message on stdout
A regular message, printed to fd 1
```

Text File I/O

- Reading and writing from/to a text file is similar to getting input from `stdin` (with `scanf`) and writing to `stdout` (with `printf`).

- Reading data from a text file is accomplished with the function `fscanf( )`. This function works the same as `scanf( )`, but requires an additional parameter which is a "handle" to the file.

- Reading a line from a text file is accomplished using the `fgets( )` function. This function is similar to `gets( )` but requires a "handle" to a file and a maximum character count.

- Similarly, writing to a text file is accomplished with the function `fprintf()` which works the same as `printf( )`, but also requires a "handle" to the file to be read.

Opening and Closing

To read or write from a text file using `fscanf( ), fegets( )` or `fprintf( )`, the file must first be opened using **`fopen( )`**. The file should be closed using **`fclose( )`** when all I/O is complete.

`fopen( )` returns a handle to the file as the type FILE* (a pointer to a FILE struct) which is then used as the argument to `fscanf(), fgets( ), fprintf( )` and `fclose( )`.

The return value from `fopen( )` should be checked to insure that the file was in fact opened.

FILE

- FILE *fp; /* file pointer */
- FILE is a structure in C

```
struct _IO_FILE {
   char  *_IO_read_ptr;   /* Current read pointer */
   char  *_IO_read_end;   /* End of get area. */
   char  *_IO_read_base;  /* Start of putback and get area. */
   char  *_IO_write_base; /* Start of put area. */
   char  *_IO_write_ptr;  /* Current put pointer. */
   char  *_IO_write_end;  /* End of put area. */
   char  *_IO_buf_base;   /* Start of reserve area. */
   char  *_IO_buf_end;   /* End of reserve area. */
   int   _fileno;
   int   _blksize;
};

typedef struct _IO_FILE FILE;
```

fopen( )

- `fopen(char *name, char *mode)` requires two parameters
  - The name of the text file to be opened
  - The text file open “mode”
    - “r” - open the file for reading only
    - “w” - create the file for writing; if the file exists, discard the its contents
    - “a” - append; open or create the file for writing at the end
    - “r+” - open the file for reading and writing
    - “w+” - create the file for reading and writing; if the file exists, discard its contents
    - “a+” - open or create the file for reading or writing at the end

Modes

- fp = fopen(name, "r");
  - Returns NULL if file does not exist, or has no read permission.

fp = fopen(name, “w");

- If file does not exist, one will be created for writing.
- If file already exists, the content will be erased when the file is opened. So be careful!
- Returns NULL if file has no write permission.

Modes

- fp = fopen(name, “a"); /* append */
  - If file does not exist, one will be created for writing.
  - If file already exists, the content will be preserved.
  - Returns NULL if file has no write permission.
- fp = fopen(name, "rw");
  - File may be read first, but the old content will be erased as soon as something is written to the file.
- fp = fopen(name, "ra");
- fp = fopen(name, “aw"); /* same as “a” */

Using fopen( )

- Open the file named “bob.txt” for reading

  FILE * myFile = fopen(“bob.txt”, “r”);

- If fopen( ) fails, the special value NULL is returned. All calls to fopen should be checked

```
FILE *myFIle = fopen (“bob.txt”, “r”)
If (myFile == NULL)
{
    /* handle the error */
}
```

Closing Files

```
int fclose( FILE *fp )

fclose( ifp );
fclose( ofp );
```

- Most operating systems have some limit on the number of files that a program may have open simultaneously $\Rightarrow$ free the file pointers when they are no longer needed.
- **`fclose`** is called automatically for each open file when a program terminates normally.
- For output files: **`fclose`** flushes the buffer in which **`putc`** is collecting output.

fscanf.c

```c
#include <stdio.h>
#include <stdlib.h>  /* for “exit” */
int main ( )
{
   double x ;
   FILE *ifp ;

   /* try to open the file for reading, check if successful */
   /* if it wasn't opened exit gracefully */
   ifp = fopen("test_data.dat", "r") ;
   if (ifp == NULL) {
     printf ("Error opening test_data.dat\n");
     exit (-1);
   }
   fscanf(ifp, "%lf", &x) ;  /* read one double from the file */
   fclose(ifp);      /* close the file when finished  */

   /* check to see what you read */
   printf("x = %.2f\n", x) ;
   return 0;
}
```

Detecting end-of-file with fscanf

- When reading an unknown number of data elements from a file using `fscanf( )`, we need a way to determine when the file has no more data to read, i.e, we have reached the “end of file”.

- Fortunately, the return value from `fscanf( )` holds the key. `fscanf( )` returns an integer which is the number of data elements read from the file. If end-of-file is detected the integer return value is the special value **EOF**

EOF example code

```
1  /* code snippet that reads an undetermined number of
2  integer student ages from a file and prints them out
3  as an example of detecting EOF
4  */
5  FILE *inFile;
6  int age;
7
8  inFile = fopen( “myfile”, “r” );
9  if (inFile == NULL) {
10      printf ("Error opening myFile\n");
11      exit (-1);
12     }
13
14 while ( fscanf(infile, “%d”, &age ) != EOF )
15      printf( “%d\n”, age );
16
17 fclose( inFile );
```

fprintf.c

```c
/* fprintf.c */
#include <stdio.h>
#include <stdlib.h>  /* exit */
int main ( )
{
  double pi = 3.14159 ;
  FILE *ofp ;

  /* try to open the file for writing, check if successful */
  /* if it wasn't exit gracefully */
  ofp = fopen("test.out", “w") ;
  if (ofp == NULL) {
    printf ("Error opening test.out\n");
    exit (-1);
  }

  /* write to the file using printf formats */
  fprintf(ofp, “Hello World\n”);
  fprintf(ofp, “PI is defined as %6.5lf\n”, pi);

  fclose(ofp);  /* close the file when finished reading */

  return 0;
}
```

fprintf vs printf
fscanf vs scanf

- Function prototypes are identical except that `fprintf` and `fscanf` require `FILE*` parameter
- Format strings identical
- `fscanf`, `fprintf` are more general
- `printf` can be written using `fprintf`
  - fprintf(stdout, ...)
- Similarly, `scanf` can be written using `fscanf`
  - fscanf(stdin, ...)

Errors to stderr

- Errors should be output to `stderr` using `fprintf` rather to `stdout` using `printf`( )
- Do this
  - fprintf(stderr, “this is the error message\n”);

  instead of this
  - printf(“this is the error message\n”);
- For example

```
ofp = fopen("test.out", “w") ;
 if (ofp == NULL) {
   fprintf (stderr, "Error opening test.out\n");
   exit (-1);
 }
```