COP 4338 Class 9

Last Class:

- Advanced pointers

-Dynamic Allocation
-Linked lists
-Pointers to pointers
-Pointers to functions

Today's Class:

-Passing arguments
-I/O
-Files

Arrays of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

Q: What is the problem with this representation?

Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

Q: What is an alternative solution?

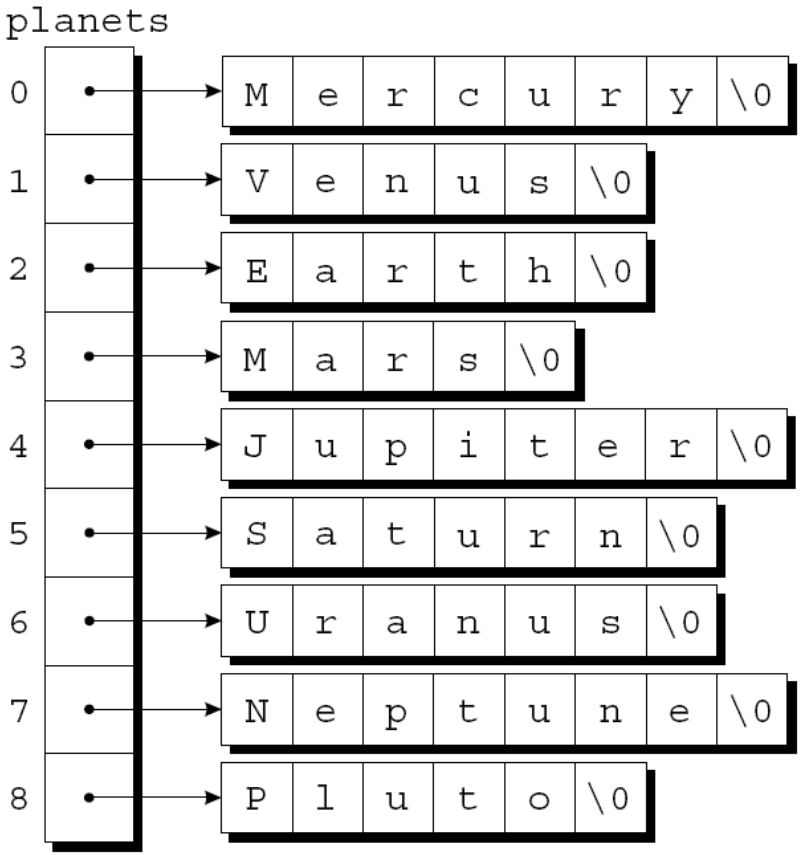
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- What we need is a ***ragged array,*** whose rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- Q: Create a loop that searches the `planets` array for strings beginning with the letter M:

Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- This may include a file name or a switch that modifies the program's behavior.
- Examples of the UNIX `ls` command:

```
ls
ls –l
ls -l remind.c
```

Please be familiar with the basic unix commands! Check moodle

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments

- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `argv[argc]` is always a ***null pointer***—a special pointer that points to nothing.
  - Remember: The macro `NULL` represents a null pointer.

Command-Line Arguments

- If the user enters the command line

  ```
  ls -l remind.c
  ```

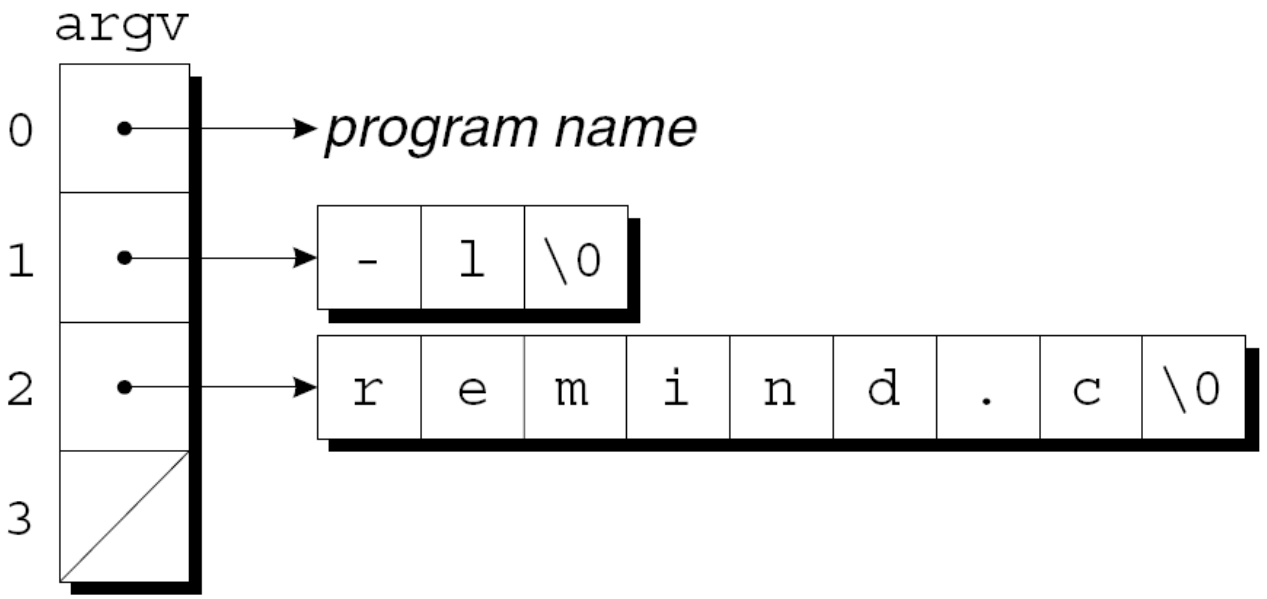
  Q: then `argc` will be , and `argv` will have the following appearance:

Command-Line Arguments

- If the user enters the command line

  `ls -l remind.c`

  then `argc` will be 3, and `argv` will have the following appearance:



Command-Line Arguments

- Since `argv` is an array of pointers, accessing command-line arguments is easy.
- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn.
- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Q: Explain why this works:

```
                        #include <stdio.h>
#include <string.h>
#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

Command-Line Arguments

What if we call the program:

```
planet Jupiter venus Earth fred
```

Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments.
- The program is designed to check a series of strings to see which ones are names of planets.
- The strings are put on the command line:

```
planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
  int i;
  printf("you entered %d command-line argument(s):\n", argc);
  for(i=0; i<argc; i++)
    printf("argv[%d]: %s\n", i, argv[i]);

   if(argc != 4) { printf("USAGE: %s <int> <long>
  <double>\n", argv[0]); return -1; }
  int arg1 = atoi(argv[1]);
  long arg2 = atol(argv[2]);
  double arg3 = atof(argv[3]);
  printf("arg1=%d, arg2=%ld, arg3=%lf\n", arg1, arg2, arg3);

  return 0;
}
```

Input/Output and Files

Introduction

- C's input/output library is one of the biggest and most important part of the standard library.
- The `<stdio.h>` header is the primary repository of input/output functions, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`.
- We will talk about these six functions.
- It also introduces many new functions, most of which deal with files.

Introduction

- Topics to be covered:
  - Streams, the `FILE` type, input and output redirection, and the difference between text files and binary files
  - Functions designed specifically for use with files, including functions that open and close files
  - Functions that perform “formatted” input/output
  - Functions that read and write unformatted data (characters, lines, and blocks)
  - Random access operations on files
  - Functions that write to a string or read from a string

Streams

- In C, the term ***stream*** means any source of input or any destination for output.
- Many small programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen).
- Larger programs may need additional streams.
- Streams often represent files stored on various media.
- However, they could just as easily be associated with devices such as network ports and printers.

File Pointers

- Accessing a stream is done through a ***file pointer,*** which has type `FILE *`.
- The `FILE` type is declared in `<stdio.h>`.
- Certain streams are represented by file pointers with standard names.
- Additional file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```

Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

File Pointer	*Stream*	*Default Meaning*
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.

Standard Streams and Redirection

- The I/O functions discussed in previous chapters obtain input from `stdin` and send output to `stdout`.
- Many operating systems allow these default meanings to be changed via a mechanism known as ***redirection.***

Standard Streams and Redirection

- A typical technique for forcing a program to obtain its input from a file instead of from the keyboard:

  ```
  demo <in.dat
  ```

  This technique is known as ***input redirection.***
- ***Output redirection*** is similar:

  ```
  demo >out.dat
  ```

  All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen.

Standard Streams and Redirection

- Input redirection and output redirection can be combined:

```
demo <in.dat >out.dat
```

- The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter:

```
demo < in.dat > out.dat
demo >out.dat <in.dat
```

What would be the problem with this?

Standard Streams and Redirection

- One problem with output redirection is that *everything* written to `stdout` is put into a file.
- Writing error messages to `stderr` instead of `stdout` guarantees that they will appear on the screen even when `stdout` has been redirected.

File Operations

- Simplicity is one of the attractions of input and output redirection.
- Unfortunately, redirection is too limited for many applications.
  - When a program relies on redirection, it has no control over its files; it doesn't even know their names.
  - Redirection doesn't help if the program needs to read from two files or write to two files at the same time.
- When redirection isn't enough, we'll use the file operations that `<stdio.h>` provides.

Text Files versus Binary Files

- What is the difference?

Text Files versus Binary Files

- `<stdio.h>` supports two kinds of files: text and binary.
- The bytes in a ***text file*** represent characters, allowing humans to examine or edit the file.
  - The source code for a C program is stored in a text file.
- In a ***binary file,*** bytes don't necessarily represent characters.
  - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
  - An executable C program is stored in a binary file.

Opening a file

```
FILE *fp;
fp = fopen (“abc.txt”,”r”);
```

fp is a file pointer

Use “w” for write mode, “a” for append mode

Opening a File

- Opening a file for use as a stream requires a call of the `fopen` function.
- Prototype for `fopen`:

```
FILE *fopen(const char * filename,
            const char * mode);
```

- `filename` is the name of the file to be opened.
  - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

```
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- When it can't open a file, `fopen` returns a null pointer.

Modes

- Factors that determine which mode string to pass to `fopen`:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

Modes

- Mode strings for text files:

String	*Meaning*
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Modes

- Mode strings for binary files:

String	*Meaning*
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```

Closing a File

- It's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against `NULL`:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) …
```

Obtaining File Names from the Command Line

- There are several ways to supply file names to a program.
  - Building file names into the program doesn't provide much flexibility.
  - Prompting the user to enter file names can be awkward.
  - Having the program obtain file names from the command line is often the best solution.
- An example that uses the command line to supply two file names to a program named `demo`:

```
demo names.dat dates.dat
```

Obtaining File Names from the Command Line

- How to do it?

Obtaining File Names from the Command Line

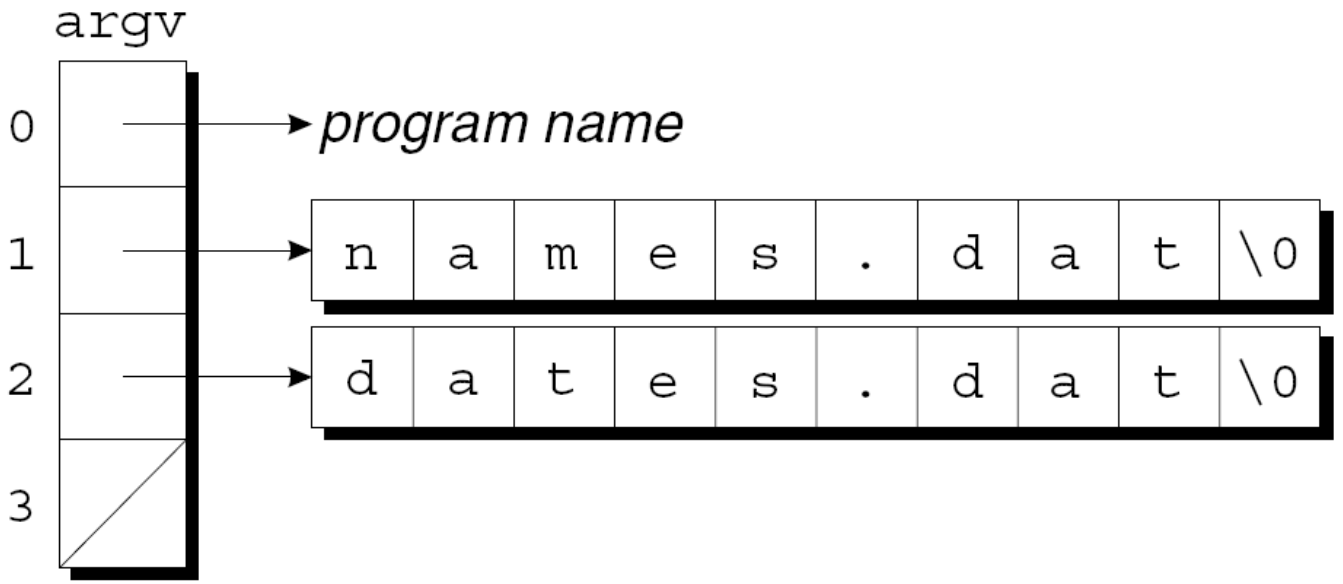
- We just saw how to access command-line arguments by defining `main` as a function with two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- `argc` is the number of command-line arguments.
- `argv` is an array of pointers to the argument strings.

Obtaining File Names from the Command Line

- `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer.
- In the `demo` example, `argc` is 3 and `argv` has the following appearance:



mystery.c

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

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

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

Program: Checking Whether a File Can Be Opened

- The mystery `.c` program determines if a file exists and can be opened for reading.
- The user will give the program a file name to check:

`mystery` *file*

- The program will then print either *file* `can be opened` or *file* `can't be opened`.
- If the user enters the wrong number of arguments on the command line, the program will print the message `usage: mystery filename`.

Miscellaneous File Operations

- The `remove` and `rename` functions allow a program to perform basic file management operations.
- Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*.
- Both functions return zero if they succeed and a nonzero value if they fail.

Miscellaneous File Operations

- `remove` deletes a file:

```
remove("foo");
  /* deletes the file named "foo" */
```

Formatted I/O

- The next group of library functions use format strings to control reading and writing.
- `printf` and related functions are able to convert data from numeric form to character form during output.
- `scanf` and related functions are able to convert data from character form to numeric form during input.

The …`printf` Functions

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.
- The prototypes for both functions end with the `...` symbol, which indicates a variable number of additional arguments:

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

- Both functions return the number of characters written; a negative return value indicates that an error occurred.
- Q: Why variable arguments?

The …printf Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

fprintfexample0.c

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen("test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fclose(fp);
}
```

The …printf Functions

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

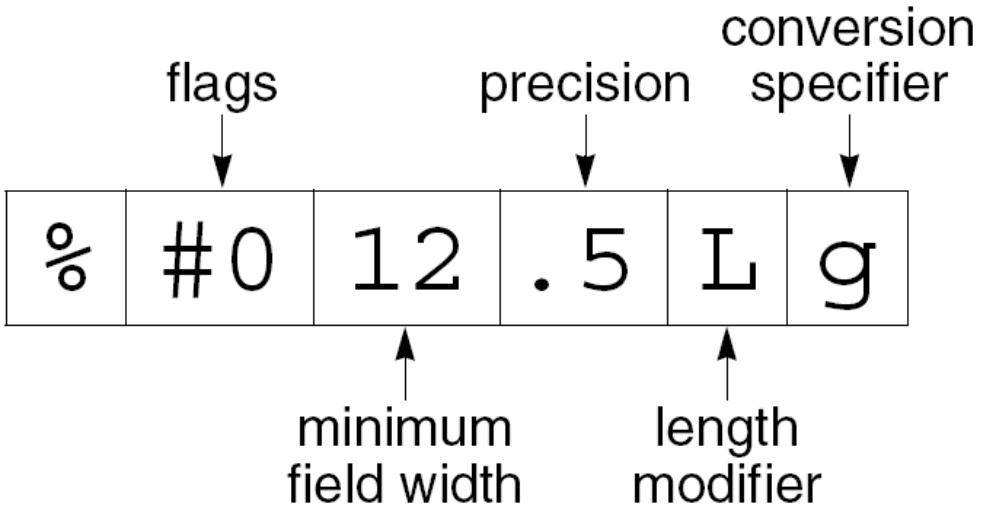
- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

…`printf` Conversion Specifications

- Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications.
  - Ordinary characters are printed as is.
  - Conversion specifications describe how the remaining arguments are to be converted to character form for display.

…`printf` Conversion Specifications

- A …`printf` conversion specification consists of the % character, followed by as many as five distinct items:



Check page 154 of K&R book

Examples of …`printf` Conversion Specifications

- Examples showing the effect of flags on the %d conversion:

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
%8d	•••••123	••••-123
%-8d	123•••••	-123••••
%+8d	••••+123	••••-123
% 8d	•••••123	••••-123
%08d	00000123	-0000123
%-+8d	+123••••	-123••••
%- 8d	•123••••	-123••••
%+08d	+0000123	-0000123
% 08d	•0000123	-0000123

The …scanf Functions

- `fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input.
- After the format string, any number of pointers—each pointing to an object—follow as additional arguments.
- Input items are converted (according to conversion specifications in the format string) and stored in these objects.

The …scanf Functions

- `scanf` always reads from `stdin`, whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);
  /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
  /* reads from fp */
```

- A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

The …scanf Functions

- Errors that cause the …scanf functions to return prematurely:
  - ***Input failure*** (no more input characters could be read)
  - ***Matching failure*** (the input characters didn't match the format string)

fscanfexample.c

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


int main()
{
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2014", fp); /* More on this later!*/

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );

   fclose(fp);

   return(0);
}
```

Character I/O

- The next group of library functions can read and write single characters.
- These functions work equally well with text streams and binary streams.
- The functions treat characters as values of type `int`, not `char`.
- One reason is that the input functions indicate an end-of-file (or error) condition by returning `EOF`, which is a negative integer constant.

Output Functions

- `putchar` writes one character to the `stdout` stream:

```
putchar(ch);    /* writes ch to stdout */
```

- `fputc` and `putc` write a character to an arbitrary stream:

```
fputc(ch, fp);  /* writes ch to fp */
putc(ch, fp);   /* writes ch to fp */
```

- `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function.

fputcexample.c

```
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char c;

  pFile = fopen ("alphabet.txt","w");
  if (pFile!=NULL) {

    for (c = 'A' ; c <= 'Z' ; c++)
      fputc ( c , pFile );

    fclose (pFile);
  }
  return 0;
}
```

Input Functions

- `getchar` reads a character from `stdin`:

```
ch = getchar();
```

- `fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);
ch = getc(fp);
```

- All three functions treat the character as an `unsigned char` value.
- As a result, they never return a negative value other than `EOF`.

Input Functions

- One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file.
- A typical `while` loop for that purpose:

```
while ((ch = getc(fp)) != EOF) {
  …
}
```

- Always store the return value in an `int` variable, not a `char` variable.
- Testing a `char` variable against `EOF` may give the wrong result.

Input Functions

```
#include <stdio.h>
int main( )
{   int c;
   printf( "Enter a value :");
   c = getchar( );
   printf( "\nYou entered: ");
   putchar( c );
   return 0;
}
```

fgetcexample.c

```
#include <stdio.h>
int main ()
{
  FILE * pFile;
  int c;
  int n = 0;
  pFile=fopen ("myfile.txt","r");
  if (pFile==NULL) printf ("Error opening file");
  else
  {
    do {
      c = getc (pFile);
      if (c == '$') n++;
    } while (c != EOF);
    fclose (pFile);
    printf ("File contains %d$.\n",n);
  }
  return 0;
}
```

mystery.c

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

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }
```

```
  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Program: Copying a File

- The `fcopy.c` program makes a copy of a file.
- The names of the original file and the new file will be specified on the command line when the program is executed.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

Program: Copying a File

- Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files.
- If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

Line I/O

- Library functions in the next group are able to read and write lines.
- These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

Output Functions

- The `puts` function writes a string of characters to `stdout`:

```
puts("Hi, there!");  /* writes to stdout */
```

- After it writes the characters in the string, `puts` always adds a new-line character.

Output Functions

- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);  /* writes to fp */
```

- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.
- Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

fputsexample.c

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "w+");

   fputs("This is c programming.", fp);
   fputs("This is a system programming language.", fp);

   fclose(fp);

   return(0);
}
```

Input Functions

- The `gets` function reads a line of input from `stdin`:

```
gets(str); /* reads a line from stdin */
```

- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).
- `fgets` is a more general version of `gets` that can read from any stream.
- `fgets` is also safer than `gets`, since it limits the number of characters that it will store.

Reminder

- "r" read: Open file for input operations. The file must exist.
- "w" write: Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
- "a" append: Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.
- "r+" read/update: Open a file for update (both for input and output). The file must exist.
- "w+" write/update: Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
- "a+" append/update: Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

Page 242 from K&R book

Example

```
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

Input Functions

- A call of `fgets` that reads a line into a character array named `str`:

```
fgets(str, sizeof(str), fp);
```

- `fgets` will read characters until it reaches the first new-line character or `sizeof(str)` – 1 characters have been read.
- If it reads the new-line character, `fgets` stores it along with the other characters.

Input Functions

- Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters.
- Otherwise, both return their first argument, which points to the array in which the input was stored.
- Both functions store a null character at the end of the string.

Yet another example

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

Block I/O

- The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step.
- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Block I/O

- `fwrite` is designed to copy an array from memory to a stream.
- Arguments in a call of `fwrite`:
  - Address of array
  - Size of each array element (in bytes)
  - Number of elements to write
  - File pointer
- A call of `fwrite` that writes the entire contents of the array `a`:

```
fwrite(a, sizeof(a[0]),
       sizeof(a) / sizeof(a[0]), fp);
```

Block I/O

- `fwrite` returns the number of elements actually written.
- This number will be less than the third argument if a write error occurs.

fwriteexample.c

```
/* fwrite example : write buffer */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char buffer[] = { 'x' , 'y' , 'z' };
  pFile = fopen ("myfile.bin", "wb");
  fwrite (buffer , sizeof(char), sizeof(buffer), pFile);
  fclose (pFile);
  return 0;
}
```

Block I/O

- `fread` will read the elements of an array from a stream.
- A call of `fread` that reads the contents of a file into the array `a`:

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

- `fread`'s return value indicates the actual number of elements read.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

Block I/O

- `fwrite` is convenient for a program that needs to store data in a file before terminating.
- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't need to be in array form.
- A call of `fwrite` that writes a structure variable `s` to a file:

```
fwrite(&s, sizeof(s), 1, fp);
```

freadexample.c

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

int main()
{
   FILE *fp;
   char c[] = "this is tutorialspoint";
   char buffer[20];

   /* Open file for both reading and writing */
   fp = fopen("file.txt", "w+");

   /* Write data to the file */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* Seek to the beginning of the file */
   fseek(fp, SEEK_SET, 0);

   /* Read and display data */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

File Positioning

- Every stream has an associated ***file position.***
- When a file is opened, the file position is set at the beginning of the file.
  - In "append" mode, the initial file position may be at the beginning or end, depending on the implementation.
- When a read or write operation is performed, the file position advances automatically, providing sequential access to data.

File Positioning

- Although sequential access is fine for many applications, some programs need the ability to jump around within a file.
- If a file contains a series of records, we might want to jump directly to a particular record.
- `<stdio.h>` provides five functions that allow a program to determine the current file position or to change it.

File Positioning

- The `fseek` function changes the file position associated with the first argument (a file pointer).
- The third argument is one of three macros:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

- The second argument, which has type `long int`, is a (possibly negative) byte count.

File Positioning

- If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file.
- If `fp` is a text stream, `ftell(fp)` isn't necessarily a byte count.
- As a result, it's best not to perform arithmetic on values returned by `ftell`.

ftell example

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int len;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror ("Error opening file");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);

   len = ftell(fp);
   fclose(fp);

   printf("The output is = %d bytes\n", len);

   return(0);
}
```

File Positioning

- The `rewind` function sets the file position at the beginning.
- The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`.
  - The difference? `rewind` doesn't return a value but does clear the error indicator for `fp`.

rewind example

```
#include <stdio.h>

int main()
{
   char str[] = "This is tutorialspoint.com";
   FILE *fp;
   int ch;

   /* First let's write some content in the file */
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);

   fp = fopen( "file.txt" , "r" );
```

```
while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);
   }
   rewind(fp);
   printf("\n");
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);

   }
   fclose(fp);

   return(0);
}
```

File Positioning

- Using `fseek` to move to the beginning of a file:
  ```
  fseek(fp, 0L, SEEK_SET);
  ```
- Using `fseek` to move to the end of a file:
  ```
  fseek(fp, 0L, SEEK_END);
  ```
- Using `fseek` to move back 10 bytes:
  ```
  fseek(fp, -10L, SEEK_CUR);
  ```
- If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

fseek example

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

int main()
{
   FILE *fp;
   char c[] = "this is tutorialspoint";
   char buffer[20];

   /* Open file for both reading and writing */
   fp = fopen("file.txt", "w+");

   /* Write data to the file */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* Seek to the beginning of the file */
   fseek(fp, SEEK_SET, 0);

   /* Read and display data */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

fseekexample.c

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is tutorialspoint.com", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Langauge", fp);
   fclose(fp);

   return(0);
}
```

File Positioning

- The call `fgetpos(fp, &file_pos)` stores the file position associated with `fp` in the `file_pos` variable.
- The call `fsetpos(fp, &file_pos)` sets the file position for `fp` to be the value stored in `file_pos`.
- If a call of `fgetpos` or `fsetpos` fails, it stores an error code in `errno`.
- Both functions return zero when they succeed and a nonzero value when they fail.

File Positioning

- An example that uses `fgetpos` and `fsetpos` to save a file position and return to it later:

```
fpos_t file_pos;
…
fgetpos(fp, &file_pos);
  /* saves current position */
…
fsetpos(fp, &file_pos);
  /* returns to old position */
```

fsetpos.c

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("World Hello!", fp);
   fclose(fp);

   return(0);
}
```

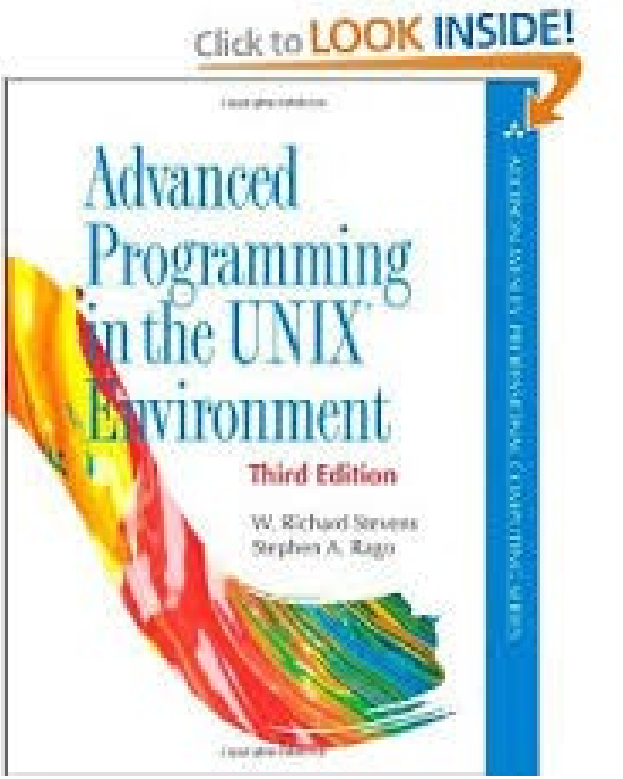
Fsetpos another

```
/* fsetpos example */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  fpos_t position;

  pFile = fopen ("myfile.txt","w");
  fgetpos (pFile, &position);
  fputs ("That is a sample",pFile);
  fsetpos (pFile, &position);
  fputs ("This",pFile);
  fclose (pFile);
  return 0;
}
```

fsetpos.c

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("World Hello!", fp);
   fclose(fp);

   return(0);
}
```

For Next Class



Read Chapters 1&2

COP 4338 Class 9

Last Class:

Passing arguments
-I/O
-Files

Today's Class:

-I/O
-Files

Reminder: First homework is out, worth 15% Due 5:00 PM Thursday 20th