Lab 10

Advanced C

CPSC 275
Introduction to Computer Systems

Today in Lab

- Command-line arguments
- Parsing command-line options
- File processing
- Structures

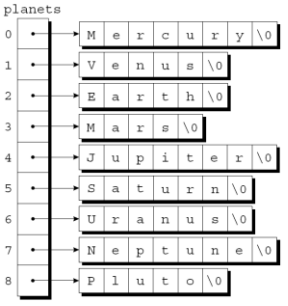
Arrays of Strings, Revisited

- Example:

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

- Each element is really a *pointer* to a string.

Arrays of Strings, cont'd



Arrays of Strings, cont'd

- To access one of the planet names, all we need do is subscript the `planets` array.
- To display the names of the planets:

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

Arrays of Strings, cont'd

- 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.
- Example:

```
$ ./repeat 10 computer
```

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

- `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,
- `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.
  - The macro `NULL` represents a null pointer.

Command-Line Arguments

- If the user enters the command line

```
$ ./repeat 10 computer
```

then `argc` will be 3 (why?), 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]);
```

Exercise 0

- Write a C program `repeat.c` which will take two command-line arguments: an integer *n* and a string and print the string *n* times.

Exercise 1

- Write a simple calculator program `calc.c`. The program should take three command-line arguments: an integer operand, a binary arithmetic operator (`+`, `–`, `*`, `/`), and an integer operand and display the result to the standard output. For example,

  ```
  $ calc 10 + 20
  ```

  should display `30`, the sum of the two numbers, `10` and `20`. Hint: Use the `atoi()` function to convert a string to an integer. See `man` page of `atoi` for details.

Parsing Command-Line Options

- Command-line arguments and options

  ```
  $ ls –l myfile
      opt   arg
  ```

- Some command-line options take values:

  ```
  $ tail –n 20 myfile
  ```

- The `getopt()` function is useful in parsing command-line options.

`getopt()`

```
#include <unistd.h>
#include <stdlib.h>
#include <getopt.h>

int getopt(int argc, char *argv[], char *optstring);
extern char *optarg;
```

- *argc* is the number of arguments.
- *argv* is an array of arguments.
- *optstring* is a string containing the option characters. If such a character is followed by a colon, the option requires a value.
- *optarg* is an *external* variable string containing the option values.
- If there are no more option characters, the function returns -1.

`getopt()`, cont'd

- Normally, getopt() is called in a loop.
- When getopt() returns -1, indicating no more options are present, the loop terminates.
- A switch statement is used to dispatch on the return value from getopt().

Example Using getopt()

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    int r, cval;
    while ((r = getopt(argc, argv, "ab:c:")) != -1)
       switch (r){
           case 'a':
               printf("Do option a!\n");
               break;
           case 'b':
               printf("Do option b with %s!\n", optarg);
               break;
           case 'c':
               cval = atoi(optarg);
               printf("Do option c with %d!\n", cval);
               break;
           default:
               printf("Error: Unknown option!\n");
           }
}
```

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.

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 * restrict filename,
            const char * restrict mode);
```

- `filename` is the name of the file to be opened.
- `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:

```
FILE *fp;
fp = fopen("infile.txt", "r");
  /* opens infile.txt for reading */
```

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

Modes 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)

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`.
- `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 "infile.txt"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(1);
  }
  …
  fclose(fp);
  return 0;
}
```

Reading From a Text File

`int fscanf(FILE *stream, char *format, …);`

reads input items from the file stream pointed by *stream*. scanf() is equivalent to fscanf() with **stdin** stream.

`int fgets(char *s, int n, FILE *stream);`

reads characters from the file stream pointed by *stream* and stores them in the array pointed by *s* up to *n*-1 characters and leaving *s* null terminated. Safer than fscanf().

Write To a Text File

`int fprintf(FILE *stream, char *format, …);`

writes output to the file stream pointed by *stream.* printf() is equivalent to fprintf() with **stdout** stream.

`int fputs(char *s, FILE *stream);`

writes the string pointed by *s* to the stream pointed to by *stream*.

LAB 10

Advanced Features of C

In today's lab, we explore some of the advanced features of C including:

- Command-line arguments
- Structures
- File processing
- Pointers to files

Exercise 0. Write a C program **repeat.c** which will take two command-line arguments: an integer *n* and a string and print the string *n* times.

Exercise 1. Write a simple calculator program **calc.c**. The program should take three command-line arguments: an integer operand, a binary arithmetic operator (+, -, *, /), and an integer operand and display the result to the standard output. For example,

```
$ calc 10 + 20
```

should display 30, the sum of the two numbers, 10 and 20. Hint: Use the `atoi`() function to convert a string to an integer. See man page of `atoi` for details.

Exercise 2. [From Lab 3] You are to make purchasing decisions of the paper goods based on price. Write a C program (**price.c**) that will take a list of products and stores and determine, on the basis of cost/square inch, which is the "best buy". But this time your program must display a sorted list of the products in the increasing of their cost/square inch.

Input comes from a text file called **price.in** which will consist of a positive integer *n* followed by *n* lines of data. Each line will contain the following information, with the elements separated by white space:

[brand code] [product code] [length in inches] [width in inches] [sheets per package] [store code] [price]

The brand name, product and store will each be represented by two-digits, as follows:

Code	*Brand*	*Code*	*Product*	*Code*	*Store*
01	Kleenex	01	Tissue	01	Shaw's
02	Charmin	02	TP	02	Walmart
03	Delsey			03	BJ's
04	Generic			04	Mom's

Output will consist of a sorted list of the products and the stores in the increasing of their cost/square inch.

Sample Input

```
3
02 02 2.25 2.75 500 02 0.89
01 01 8 5 200 01 1.39
02 02 2.25 2.75 480 04 1.00
```

Sample Output

```
Kleenex Tissue at Shaw's: $0.00017
Charmin TP at Walmart: $0.00029
Charmin TP at Mom's: $0.00034
```

NOTE: Assume that all input values are valid and there is no limit to the value of *n*.