CS 261
Fall 2016

Mike Lam, Professor

Getopt, Structs, and Typedefs

(aka more P1 topics)

Ad-hoc command-line parsing

```
int main (int argc, char **argv)
{
    // parse options
    for (int i = 0; i < argc; i++) {
        switch (argv[i][1]) {
            case ‘a’:    a_flag = true;    break;
            case ‘b’:    b_flag = true;    break;
            default:     report_err();     break;
        }
    }

    // get filename
    char *fn = argv[argc-1];
}
```

Valid comands:

```
./main file.txt
./main -a file.txt
./main -a -b file.txt
```

What if there's no filename at the end?
What if the filename is "aa.txt"?
How to handle parameters (e.g., “`-n 5`”)?
How to handle combined flags (e.g., “`-ab`”)?
What if there is no `argv[i][1]`?

Getopt

- There's a better way!
  - `getopt()` and `getopt_long()`
  - The latter enables longer options (e.g., "--help")
    - Useful (and mostly standard now), but we won't use it in this course
  - Basic idea: call `getopt()` repeatedly; it will return each of the flags individually even if they are grouped or out of order; returns -1 when done
  - Need to pass an `optstring` (list of valid flags as a string)
    - Use a colon to indicate a flag that takes a parameter (e.g., "`-n 4`")
- Static variables
  - `optarg`: pointer to parameter for flags that take them
  - `optind`: index of next flag
    - Use this to check for extra arguments at the end!

Getopt example

```
#include <getopt.h>

int main (int argc, char **argv)
{
    // parse options
    int opt;
    while ((opt = getopt(argc, argv, "ab")) != -1) {
        switch (opt) {
            case 'a':   a_flag = true;     break;
            case 'b':   b_flag = true;     break;
            default:    report_err();      break;         // invalid
        }
    }

    // check for and get filename
    if (optind != argc-1) {
        report_err();
        return 1;
    }
    char *fn = argv[optind];
}
```

Much more robust!

Exercise

- Write a program (args.c) that takes command-line parameters according to the following usage text:

```
Usage: ./args [options] <filename>
 Valid options:
  -a        Print an 'A'
  -b        Print a 'B'
  -c        Print a 'C'
  -n <i>    Print i copies of 'N'
```

Valid commands:

```
./args file.txt
./args -a file.txt
./args -a -c file.txt
./args -abc file.txt
./args -n 4 file.txt
./args -a -n4 file.txt
./args -a -n4 -c file.txt
```

Invalid commands:

```
./args
./args -a
./args -n file.txt
```

Typedefs

- A `typedef` is a way to create a new type name
  - Basically a synonym for another type
  - Usually postfixed with "`_t`"

```
typedef unsigned char byte_t;

byte_t b1, b2;
```

Structs

- A `struct` is a new kind of data type that contains a group of related sub-variables of any type (including structs!)
  - Variables must also be declared with `struct` keyword

```
struct vertex {
    double x;
    double y;
    bool visited;
};
```

```
int main()
{
    struct vertex p1;
    p1.x = 4.2;
    p1.y = 5.6;
    p1.visited = false;
}
```

```
double dist(struct vertex p1, struct vertex p2)
{
    return sqrt( (p1.x-p2.x)*(p1.x-p2.x) +
                 (p1.y-p2.y)*(p1.y-p2.y) );
}
```

Typedef structs

- We typically simplify the use of structs by creating a typedef name for them
  - For projects, we'll provide both structs and typedefs in headers

```
typedef struct vertex {
    double x;
    double y;
    bool visited;
} vertex_t;
```

```
int main()
{
    vertex_t p1;
    p1.x = 4.2;
    p1.y = 5.6;
    p1.visited = false;
}
```

```
double dist(vertex_t p1, vertex_t p2)
{
    return sqrt( (p1.x-p2.x)*(p1.x-p2.x) +
                 (p1.y-p2.y)*(p1.y-p2.y) );
}
```

Data alignment

- By default, the compiler is allowed to insert padding and/or re-arrange the members in memory to optimize the program
  - Often used to “align” fields on word-addressable boundaries
  - Use "`__attribute__((__packed__))`" to prevent this in GCC
  - You'll see this in the `elf.h` header file for P1
  - Caution: this is non-standard and potentially harmful

```
typedef struct {
    char a;
    char b;
    char c;
    int x;
} stuff_t;

sizeof(stuff_t) == 8
```

```
typedef struct __attribute__((__packed__)) {
    char a;
    char b;
    char c;
    int x;
} stuff_t;

sizeof(stuff_t) == 7
```

Example

- Write a program that reads three bytes from a file
- These bytes represent ASCII encodings of a person's first, middle, and last initials, respectively
- The program should print the initials as text characters
- With the optional “-u” switch, the program should print the initials as upper case even if not given that way
- With the optional “-p” switch, the program should print periods (“.”) after each letter

Exercises

- Extend `initials.c`
  - Add a new switch "`-h`" that prints help text and exits
  - Add a new switch "`-s`" that adds spaces between letters
  - Read and print multiple names from the file, one per line
  - Allow names to come from standard input if no filename is specified
- Small programs
  - Write a program that takes a single string parameter and reverses it
  - Write a program that reads a file and determines if each line is a palindrome
  - Write a program that takes as parameters a filename and a two-character hex value, appending the value to the end of the file
  - Write a program that reads a C source file and counts the number of lines that contain a C++ style comment (e.g., "`// text here`")
- Linux utility equivalents
  - Write an equivalent to "`hd`", which prints the contents of a file in hex followed by character equivalents
  - Write an equivalent to "`uniq`", which takes a list of words from standard in and reprints the list to standard out, omitting any immediately-following duplicates
  - Write an equivalent to "`sort`", which takes a list of numbers or words from standard in and sorts them, printing the sorted list to standard out (HINT: use the `qsort` function, and start with numbers!)