211: Computer Architecture
Fall 2019

Topic:

- C programming

typedef

typedef is used to name types (for clarity and ease-of-use)

- typedef <type> <name>;

Examples:

- typedef int Color;
- typedef struct flightType WeatherData;
- typedef struct ABGroup {

```
        int a;
        double b;
    } ABGroup;
```

Preprocessor

C compilation uses a preprocessor called cpp

The preprocessor manipulates the source code in various ways before the code is passed through the compiler

- Preprocessor is controlled by directives
- cpp is pretty rich in functionality

Our use of the preprocessor will be pretty limited

- #include <stdio.h>
- #include “myHeader.h”
- #ifndef MY_HEADER_H
  #define MY_HEADER_H
  …
  #endif

Standard C Library

Much useful functionality provided by Standard C Library

- A collection of functions and macros that must be implemented by any ANSI standard implementation
  - E.g., I/O, string handling, etc.
- Automatically linked with every executable
- Implementation depends on processor, operating system, etc., but interface is standard

Since they are not part of the language, compiler must be told about function interfaces

Standard header files are provided, which contain declarations of functions, variables, etc.

- E.g., stdio.h
- Typically in /usr/include

Command Line Arguments

When using a shell

$ hello 5

Entire command line will be given to your program as a sequence of strings

- White spaces are typically the separator characters
  - Shell dependent
- int main(int argc, char * argv []) {

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

  - argc: number of strings in command line
    - » In our example, argc = 2
  - argv: the strings themselves
    - » In our example, argv[0] = “hello\0” and arg[1] = “5\0”

Command Line Arguments

When using a shell

$ hello 5

Entire command line will be given to your program as a sequence of strings

- Integers are also passed in as strings
- Use atoi() to convert:
  - int x = atoi(s);
- Or strtoul() for better error checking:
  - char* endptr;
  - int x = strtoul(s, &endptr, 10);

System Calls

The operating system extends the functionality of the underlying hardware

- OS functionalities exported as a set of system calls
- In C, system calls are "wrapped" by C functions
  - System calls look like C function calls
- System calls are described in section 2 of online manual
  - E.g., man 2 open

In some instances, the C standard library adds functionality on top of system calls

- File I/O

File I/O

A file is a contiguous set of bytes

- Has a name
- Can create, remove, read, write, and append

Unix/Linux supports persistent files stored on disk

- Access using system calls: open(), read(), write(), close(), creat(), lseek()
- Provide random access
- Section 2 of online manual (man)

C supports extended interface to UNIX files

- fopen(), fscanf(), fprintf(), fgetc(), fputc(), fclose()
- View files as streams of bytes
- Section 3 of online manual (man)

fopen

The fopen (pronounced "eff-open") function associates a physical file with a stream.

```
FILE *fopen(char* name, char* mode);
```

First argument: `name`

- The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.

Second argument: `mode`

- How the file will be used:
  `"r"` -- read from the file
  `"w"` -- write, starting at the beginning of the file
  `"a"` -- write, starting at the end of the file (append)

fprintf and fscanf

Once a file is opened, it can be read or written using fscanf() and fprintf()

These are just like scanf() and printf() except with an additional argument specifying a file pointer

- fprintf(outfile, "The answer is %d\n", x);
- fscanf(infile, "%s %d/%d/%d %lf",
  &name, &bMonth, &bDay, &bYear, &gpa);

When started, each executing program has three standard streams open for input, output, and errors

- stdin, stdout, stderr

fclose

The fclose (pronounced "eff-close") function flushes and closes an open file.

```
int fclose(FILE* stream);
```

First argument: `stream`

- The file stream to close.

Classic Memory Bugs

Memory management is one of the biggest differences between C and Java

Here are some classic bugs that might afflict you

Bug - # 1

The classic `scanf` bug

```
scanf("%d", val);
```

Bug - # 2

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
  int *y = malloc(N*sizeof(int));
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      y[i] += A[i][j]*x[j];
  return y;
}
```

Bug - # 3

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Bug - # 4

Overwriting Memory

- Off-by-one error

```
int **p;
p = malloc(N*sizeof(int *));
for (i=0; i<=N; i++) {
  p[i] = malloc(M*sizeof(int));
}
```

Bug - # 5

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {
  while (*p && *p != val)
     p += sizeof(int);
  return p;
}
```

Bug - # 6

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
  int val;
  return &val;
}
```

Bug - # 7

Freeing Blocks Multiple Times

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

Bug - # 8

Referencing Freed Blocks

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
    y[i] = x[i]++;
```

Bug - # 9

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer

```
void foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Bug - # 10

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
  int val;
  struct list *next;
};

void foo() {
  struct list *head = malloc(sizeof(struct list));
  head->val = 0;
  head->next = NULL;
  <create and manipulate the rest of the list>
  ...
  free(head);
}
```