Welcome to the CS3410 C Primer

Please sit in the front rows so that you can see terminal output

If you can't read this, then you are too far away

C Primer

CS3410

Paul Upchurch & Jason Yosinski

Material

Introduction to writing C programs on a UNIX system.

Same material as CS2022, but condensed into three 2-hour sessions.

Knowledge of a modern high-level language is helpful (C++, Java). Otherwise, Google is your friend.

Schedule

January 28 Monday	Hello World, pointers, memory model, UNIX
February 7 Thursday	Arrays, structured data, debugging, I/O (file and network)
February 11 Monday	Preprocessor, serialization, threads, advanced topics (goto, exceptions, assembly), C for Java programmers

More info

See the course web page for CS2022.

Slides, homeworks and example code by Hussam Abu-Libdeh.

www.cs.cornell.edu/courses/CS2022/2011fa/

UNIX Access

All students have UNIX accounts in the CSUGLab.

1. Create your password at
http://www.csuglab.cornell.edu/userinfo/

2. ssh to csugXX.csuglab.cornell.edu

This info will be on the first homework.

Arrays and Strings

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 5

Arrays

- Contiguous memory
- Type is same as element-pointer
    - Accessing array elements is syntactic sugar for pointer arithmetic
- On the stack
    - Fixed-size (at compile time)
    - Compiler allocates
    - Compiler deallocates
- On the heap
    - Variable size (malloc)
    - Explicit allocation/deallocation

Declaring Arrays

```
void foo(int x) {
    int a[100];
    int b[] = {0, 1, 0, 2, 3, 1};
    int c[x]; // ERROR: Size must be const.

    a[0] = 10;
    a[5] = b[2];
    a[100] = 10; // BAD: Clobbering stack!!

    *(a + 1) = 20;   // same as a[1] = 20;
    *b = *(a + 5);   // same as b[0] = a[5];
}
```

Declaring Arrays

```
#include<stdlib.h>

void foo(int x) {
    int *a = malloc(x * sizeof(int));

    a[0] = 10;      // same as *a = 10;
    a[1] = a[0];    // same as *(a+1) = *a;

    free(a);
}
```

Relevant Library

`#include <string.h>`

- Set all elements to 0:

  memset(array, 0, **bytes**)
- Copy elements:

  memcpy(dst, src, **bytes**)
- Note:
  - bytes = number of elements * sizeof(int) for integer arrays.

Array Problems

- No array-bound checks. No warnings.
  - Can clobber stack or heap
  - **especially** with array-to-array copy when the destination array doesn't have enough space.
- `sizeof(array)` returns:
  - number of *bytes*, when exact size can be determined
  - size of pointer, when size cannot be determined at compile time and is treated as a pointer. Avoid this!

Characters

- Type for character: `char`
- 1-byte in size
- Enclosed in single-quotes
- `printf` format: `%c`
- ASCII character
    - Alpha: `'a'`
    - Digit: `'4'`
    - Special: `'\t'`
    - Null: `'\0'`
- Type for unicode charcter: `wchar_t`

Strings

- Just an array of characters
- String: `char *` or `char []`
- Terminated by *Null character* ('\0')
- Literals enclosed in double-quotes
- `"Hello"` is the same as
  `char str[] = {'H', 'e', 'l', 'l', 'o', '\0'}`

Strings

- `printf` format: `%s`
- `(str + 5)`:
    - type is `char *`;
    - value: substring starting at 6th character
- `*(str + 5)` or `str[5]`
    - the 6th character

`string.h` Library Functions

- Many functions for common string manipulation tasks.
    - . . . use them, they will make your life a lot easier
- Library functions expect null-terminated strings.
- When joining/copying/splitting strings, library inserts null-character where appropriate.

Getting Help on Library Functions

To quickly check the manual pages,

```
man func_name
```

on a Unix/Linux system. Example:

- `man strlen`

`string.h` Library Functions

- `strlen(s)`
  Length, **not** including '\0'
- `strncpy(dst, src, n)`
  Copies 'n' characters from src to dst (incl. '\0')
- `strncat(dst, src, n)`
  Copies characters from src to end of dst until dst has 'n' characters (incl. '\0')

`string.h` Library Functions

- `int strcmp(char *s1, char *s2)`
  Compares strings. Returns 0 when strings **are equal**. Positive when `s1` greater, negative when `s1` smaller. ASCII order.

Note: Cannot use == to check string equality since it compares pointers. Points to two different copies of the same string will be different.

string.h Library Functions

- char *strstr(char *haystack, char *needle)
  Search for a substring in a string
- char *strdup(char *str)
  Allocates space on the heap (with malloc) and copies the argument into the allocated space.
  Caller MUST free the returned string when done.
- char *strtok_r(char *str, char *delim, char **sav)
  Used to break apart a string into pieces. See man-page for details.

Arrays of Strings

- `char **` or `char *a[]`
- e.g. command-line arguments
- `a[0]` is a string (type `char *`)

Enum, Typedef, Structures and Unions

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 6

More Primitive Types

- Different sized integers
  - `int`: machine-dependent
  - `char`: 8-bits
  - `int8_t`: 8-bits signed
  - `int16_t`: 16-bits signed
  - `int32_t`: 32-bits signed
  - `int64_t`: 64-bits signed
  - `uint8_t, uint32_t, ...`: unsigned
- Floating point numbers
  - `float`: 32-bits
  - `double`: 64-bits

Complex Types

- Enumerations
  (user-defined `weekday`: `sunday, monday, ...`)
- Structures (user-defined combinations of other types)
- Union (same data, multiple interpretations)
- Function types (and function pointers)
- Arrays and Pointers of the above

Enumerations

```
enum days {mon, tue, wed, thu, fri, sat, sun};
// Same as:
// #define mon 0
// #define tue 1
// ...
// #define sun 6
```

```
enum days {mon=3, tue=8, wed, thu, fri, sat, sun};
// Same as:
// #define mon 3
// #define tue 8
// ...
// #define sun 13
```

Enumerations

```
enum days day;
// Same as:    int day;

for(day = mon; day <= sun; day++) {
    if (day == sun) {
        printf("Sun\n");
    } else {
        printf("day = %d\n", day);
    }
}
```

Enumerations

- Basically integers
- Can use in expressions like ints
- Makes code easier to read
- Cannot get string equiv.
- caution: `day++` will always add 1 even if enum values aren't contiguous.

Structures

```
struct mystruct {
    char name[32];
    int age;
    char *addr;
};
```

Structures

```
void foo(void) {
    struct mystruct person;             // uninitialized

    struct mystruct person2 = {         // initialization
        .name = {’f’,’o’,’o’,’\0’},
        .age = 22,
        .addr = NULL
    };

                                        // struct pointer
    struct mystruct *pptr =
     (struct mystruct *)malloc(sizeof(struct mystruct));

    ...
```

Structures

```
struct mystruct {
    char name[32];
    int age;
    char *addr;
};
```

```
...

person.age = 10;                  // direct access
person.addr = (char *)malloc(64);

pptr->age = 24;                   // indirect access
strncpy(pptr->name,"foo",32);  // through pointer
pptr->addr = NULL;

...
```

Structures

- Container for related data
- Chunks of memory; syntactic sugar for easy access.
- May have empty gaps between members

  (see `#pragma pack`)
- Hit: you'll need to use for linked-list assignment

Unions

```
union myunion {
    int x;
    struct {
        char b1;
        char b2;
        char b3;
        char b4;
    } b;
};

union myunion num;

num.x = 1000;
num.b.b1 = 5;
```

Unions

- Same memory space interpreted as multiple types
- Useful for plugins, sclicing network packets etc.

Function Pointers

```
int min(int a, int b);
int max(int a, int b);

int foo(int do_min) {
    int (*func)(int,int);       // declaring func. ptr

    if (do_min)
        func = min;
    else
        func = max;

    return func(10,20);         // indirect call
}
```

Function Poninters

- Points to a function
- Has a ∗-type of the function it points to

Renaming Types

- Complex types inconvenient to write over and over
  - `(enum day *)malloc(sizeof(enum day)`
  - `(struct foo *)malloc(sizeof(struct foo)`
  - `(union bar *)malloc(sizeof(union bar)`
  - `(int (*)(int,int))((void *)min)`

Renaming Types

```
typedef long old type newtype
typedef enum day day_t;
typedef struct foo foo_t;
typedef int (fptr_t)(int,int);
```

Debugging

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 7

Before we begin...

- A quick note on arrays
  - We said that there are similarities between arrays and pointers
  - You can use pointers as if they they are arrays (i.e. `ptr[1]`)
  - But they **are not exactly the same**

Before we begin...

- `ptr1 = ptr2;` **makes sense**
  - Here we are assigning the value of variable `ptr2` to the variable `ptr1`
  - The values just *happen to be* memory addresses
- `array1 = array2;` **does not make sense**
  - `array1` and `array2` are the base addresses of the array, but they are not full-fledged pointers (we can not have them point to different memory locations)
  - C does not automatically copy the values of one array to another (what if they are different in size?)
  - So expressions like `array1 = array2;` and `char str[100] = argv[1];` will give you compilation errors

Print Debugging

- Manually insert debugging statements
- Debugging statements print to screen
  - Caution: `stdout` is buffered. `printf` output may not appear before program crashes.
  - Solution: `stderr` is unbuffered.

printf debugging

```
fprintf(stderr, "%d %p", i, p);
```

- %d – `int`
- %s – `char *`
- %p – any pointer
- see man page for others `$ man 3 printf`

debug.c: Trace Information

```
#include <stdio.h>

int main(int argc, char **argv) {
  fprintf(stderr, "%s:%d:%s\t%s\n", __FILE__,
      __LINE__, __FUNCTION__, argv[0]);

  fprintf(stderr, "%s:%d:%s\t%s\n", __FILE__,
      __LINE__, __FUNCTION__, argv[1]);

  fprintf(stderr, "%s:%d:%s\t%s\n", __FILE__,
      __LINE__, __FUNCTION__, argv[2]);
}
```

```
trace.c:5:main ./trace
trace.c:8:main hello
trace.c:11:main world
```

GDB: GNU Debugger

- Using `printf` is fine to get a quick idea about what might be wrong
- Using trace printing can give more info
- But, no substitute for debugging!
- Debugging allows us to:
  - step into the code
  - see the execution path of our program
  - examine the values of all variables
  - set up breakpoints for careful examination
  - get a better idea of what is going wrong
- GDB is a command-line debugger for many languages including C
  - Not only debugger for C however!

GDB: Commands

- b <function> – Breakpoint on entering function
- r <args> – Run program
- list – print C code
- n – execute one statement
- s – execute one step (step into function calls)
- c – Continue running program
- p <variable> – print the value of a variable
- bt – Backtrace the stack
- fr <num> – Make stackframe <num> current frame for printing variables
- q – Quit
- help – More GDB help

GDB: GNU Debugger

```
[saikat@submit cs113]$ gcc -g -o cmd cmd.c
[saikat@submit cs113]$ ./cmd foo
Segmentation fault
[saikat@submit cs113]$ gdb ./cmd
...
(gdb) b main
Breakpoint 1 at 0x80483a4:  file cmd.c, line 3.
(gdb) r foo
...
Breakpoint 1, main (argc=1209306428, argv=0x4802f4c6) at
cmd.c:3
3 int main(int argc, char **argv) {
(gdb) n
main (argc=2, argv=0xbfb646e4) at cmd.c:6
6 n = atoi(argv[1]);
(gdb) p argc
$1 = 2
```

GDB: GNU Debugger

```
(gdb) p argv[0]
$2 = 0xbfb65c84 "/home/netid/cs113/cmd"
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x48045eae in ____strtol_l_internal () from /lib/libc.so.6
(gdb) bt
#0 0x48045eae in ____strtol_l_internal () from
/lib/libc.so.6
#1 0x48045c57 in __strtol_internal () from /lib/libc.so.6
#2 0x48043511 in atoi () from /lib/libc.so.6
#3 0x080483eb in main (argc=2, argv=0xbfb646e4) at cmd.c:7
(gdb) fr 3
#3 0x080483eb in main (argc=2, argv=0xbfb646e4) at cmd.c:7
7 m = atoi(argv[2]);
(gdb) p argv[2]
$3 = 0x0
```

Things to try

- Crash a program by dereferencing a NULL pointer.
- Crash a program by running out of stack space.
- Crash a program by clobbering the stack (e.g. the return address).
- Crash a program by calling `abort()`.

... debug each of these cases using GDB

File and Network I/O

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

Cornell University
(based on slides by Saikat Guha)

Fall 2011, Lecture 8

Input and Output

- Keyboard I/O
- Disk I/O
- Network I/O

Streams

- In many programming languages, input/output are done in streams
- Data exists on the stream, you consume part of it and move on
- Examples:
  - `stdout`: standard output stream
  - `stderr`: standard error output stream
  - `stdin`: standard input stream
  - files
  - network sockets (network connections)

Output to Terminal

- Write a line to stdout
  - `puts("hello world");`
- Write a formatted line to stdout
  - `printf("Borat says: Hi %s!\n", i);`
- Can write to streams other than stdout

  -
    ```
    fputs("an error message", stderr);
    fprintf(stderr, "Error on value %d\n", i);
    ```

Input from User (Keyboard)

Reading till end of line

```
char buf[128];
fgets(buf, 128, stdin);
```

Reading formatted input

```
int i, j;
char buf[128];
scanf("%d %d %s", &i, &j, buf);
```

File I/O

The C standard library way

- Use `fopen`/`fclose`
- Deals with *streams*

The POSIX way

- Use `open`/`close`
- Deals with *file descriptors*

We will only discuss POSIX I/O here. To read more about C streams, check out the `man` pages and/or http://www.cs.cf.ac.uk/Dave/C/CE.html

File I/O

Opening and closing files

```
int fd;                      // File Descriptor
fd = open("/path/to/file", O_RDWR | O_CREAT);
close(fd);
```

Reading and Writing

```
char buf[4096]; int len;
len = read(fd, buf, 4096)
len = write(fd, buf, 4096);
```

WARNING: Size passed is only a **suggestion**. May read/write fewer than requested number of bytes. Return value is number of bytes actually read/written. MUST retry if not fully read/written.

File I/O

- `lseek(fd, numbytes, SEEK_CUR);`
  Seek numbytes from from current location.
- `sync();`
  Ensure bytes hit the disk. Not needed for the most part.
- `FILE *ffd = fdopen(fd, "r");`
  Construct a stream from file-descriptor.
- `fprintf(ffd, "format", args);`
  Write formatted text output to file.
- `fscanf(ffd, "format", args);`
  Read formatted input from the file.
- `fclose(ffd);`
  Close a stream.

Network I/O

Opening and closing network sockets

```
int sock;                           // File Descriptor
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
close(sock);
```

Internet Addresses

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(0x7F000001);
addr.sin_port = htons(8080);
```

Fill the address info manually or get the info automatically with `getaddrinfo()`.
See `man getaddrinfo`

Network I/O

Server

```
srv = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
err = bind(srv, (struct sockaddr *)&addr, sizeof(addr));
if (err) ...
err = listen(srv, 5);
if (err) ...
cli = accept(srv, NULL, 0);
```

Client

```
cli = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
err = connect(cli, (struct sockaddr *)&addr, sizeof(addr));
if (err) ...
```

Read/Write data just as you would with file-descriptors.