

C Boot Camp

CS 110

Topics Covered

- Integer types in C
- Structs
- Pointers
- Memory management
- Procedural programming
- Arrays
- `typedef`
- `printf`
- Bitwise operations
- unions
- Function pointers
- `char *`'s
- C conventions
- Tips & Tricks

Integer types

- The size of the type `size_t` on a machine is known as the word size
- Word size is the natural unit of data
- Word size is given in bytes, converted to bits to describe machine – 64-bit, 32-bit, etc.
- C specifies ranges, not exact sizes for types
 - But sometimes this leads to problems when the underlying sizes of types change

Integer types

- Can hardcode number of bits in the type:
 - `int8_t`, `int16_t`, `int32_t`, etc. for signed ints
 - `uint8_t`, `uint16_t`, `uint32_t`, etc. for unsigned ints
 - Requires you to `#include <stdint.h>`
- Assn 1 has you interpret binary data from an old machine where integers were smaller
- Make sure you always use the correct types
 - Using the wrong types will yield confusing errors

Structs

- Ways to group data together
- Like data in classes, but without functions
- No public/private either – everything is publicly visible

Structs

```
struct inode {
    uint16_t    i_mode;
    uint8_t     i_nlink;        /* directory entries */
    uint8_t     i_uid;         /* owner */
    uint8_t     i_gid;         /* group of owner*/
    uint8_t     i_size0;       /* most significant of size */
    uint16_t    i_size1;       /* least sig */
    uint16_t    i_addr[8];     /* device addresses constituting
file */
    uint16_t    i_atime[2];    /* access time */
    uint16_t    i_mtime[2];    /* modify time */
}; /* <-- MUST HAVE THIS SEMICOLON !!! */
```

Structs

```
struct inode n;  
/* Now all of n's fields are filled with whatever was on  
   the stack before! All junk! */  
n.i_mode = 0;  
n.i_nlink = 0;  
...  
/* You have to explicitly clear the fields yourself */
```

Pointers

- The ability to refer to memory locations as variables
- A fundamental difference between C and Java
- The source of a lot of security issues
- One of the most useful features of C

Pointers

```
int x = 5; /*x's value is 5*/
int *y = &x; /*y's value is the address of x*/
*y++; /*Dereference y, then update x to 6*/
x++; /*Directly increment x to update to 7*/

/* Pointer arithmetic */
y++; /* y's new value is the address of y(!)
On pointers, "++" looks at the type of the pointer
(in this case it's int) and adds the size of that
type on the current architecture to the memory
address.
*/
```

Pointers with Structs

- **Syntax:** `struct inode *p;`
- **Accessing fields:** `p->i_mode`, `p->i_uid`
- `(*p).i_mode`, `(*p).i_uid` **also possible**
 - **This is really bad syntax! Do not do this!**
- Fields can be used just like simple variables:
`-p->i_size1++ /* For example */`

Memory Management

- Allocate memory with `malloc`
- Takes 1 parameter : number of bytes needed
- Might return `NULL` - must check this!
- Return value is of type `void *`, should cast
- Returned memory is not zeroed out
- Use `memset` to zero out if necessary

Memory Management

- Can also use `calloc` to allocate memory
- Takes 2 parameters:
 - Number of contiguous chunks of memory needed
 - Size of each chunk
- `calloc` zeroes out memory for you
- Can still return `NULL`, still returns `void *`
- Does the math for you, unlike `malloc`

Memory Management

- Free memory with `free`
- Takes 1 parameter – pointer to memory
- Argument must be a return value from `malloc` or `calloc`
- Do not free a pointer twice!
- Do not call `free` on an address halfway in a block of memory returned by `malloc` or `calloc`

Memory Management Advice

- Do not use `malloc` unless absolutely necessary
- If you need the callee to update some struct for the caller, use stack memory if possible
- Functions should clean up their own `malloc`'d memory or do 1 of the 2 cases below:
- Try to limit `malloc` to 2 cases:
 - Complex return value (e.g. a struct) for a function
 - Storing data in a long-lived structure

Memory Management Advice

- Functions should not free pointers passed in
- Unless they are destructor functions
- Destructor functions are called to clean up structs and other memory
- More on destructor functions later

Memory Management Advice

- Wrap `malloc/calloc` in a function that checks for `NULL` return value
- Use wrapped function – saves clutter
- Watch out for functions like `strdup` that call `malloc` for you – you have to free those too!
- Use `valgrind` to check your code for memory leaks (highly recommended)
 - See class website for `valgrind` instructions

Procedural Programming

- Also known as imperative programming
- Programming by side effect
- Focus of each line is a function call
 - Not an object whose method you're invoking
- Return types are simple – some examples:
 - int (e.g. `read`: 0 for success, -1 on failure)
 - pointer (e.g. `malloc`: `void *` for success, NULL on failure)

Procedural Programming

- Consider the following Java code:

```
public class Widget{
    private int x;
    public Widget(){
        x = 0;
    }
    public void incrementAndPrint(){
        x++;
        System.out.println("New value of x is: "+x);
    }
}
class Main{
    public static void main(String[] args){
        Widget w = new Widget();
        w.incrementAndPrint();
    }
}
```

Procedural Programming

- And now the procedural equivalent:

```
struct widget{
    int x;
};
void init_widget(struct widget *w){
    w->x = 0;
}
void increment_and_print(struct widget *w){
    w->x++;
    printf("New value of x is: %d\n", w->x);
}
int main(){
    struct widget w1;
    init_widget(&w1);
    increment_and_print(&w1);
    ...
```

Procedural Programming

Common pattern:

1. Make a struct and initialize its fields
2. Manipulate using functions that act on pointers to struct
 - “Init” function may be used as a faux-constructor
1. Function returns either a simple type (`int`, `char`) or nothing (updates struct instead)
2. Use a destructor function for cleanup

Arrays : Stack

- Contiguous blocks of memory

```
int buf[12]; /* array for 12 ints */  
buf[0] = 0;  
buf[0]++; /* buf[0] = 1 now */
```

- `buf` is also an `int *` - passing in `buf` is the same as passing in `&(buf[0])`.
- `buf+i` is the same as `&(buf[i])`.
- **No out of bounds protection! Be careful!**

Arrays : Heap

- Pointers can also be used as arrays
- ```
int *buf = (int *) malloc(12 *
sizeof(int));

/* hiding NULL check */

buf[0] = 0;

buf[0]++; /* buf[0] == 1 now */
```
- Corollaries from previous slide:
  - `buf[0]` is the same as `*(buf)`
  - `buf[i]` is the same as `*(buf + i)`

# typedef

- Way to refer to types with a shorthand syntax
- Eliminates cumbersome types

```
struct
 from_another_programmer_with_long_name{
 ...
}
```

- **Now just do this (on one line):**
  - `typedef struct  
 from_another_programmer_with_long_name  
 mystruct;`

# typedef

- Now can just do `mystruct foo;` to create an instance of that struct
- Syntax is :
  - `typedef <current form> <short form>;`
- Can also just be to describe data types with more useful annotations:
  - `typedef int score;`
  - `score mine = 5;`

# typedef

- Compiler sees through all levels of typedefs – only sees the underlying types
- typedefs are for programmer convenience
- `typedef struct{ ... } <name>;` is an alternative struct declaration syntax
  - The current form is the anonymous struct, and name given is defined to be that struct
  - Example: Can now do `inode i;` instead of `struct inode i;`

# printf : print formatted

- Console printing for C programs
- `printf("Hello world");` prints "Hello world", but no newline
- Newlines are appended manually:
  - `printf("Hello world\n");`
- To print variable values, use format strings:
  - `printf("The value of x is : %d", x); /* If x is an int */`

# printf

- Format flags indicate where values are substituted and how they are interpreted
  - `%d` `int`
  - `%s` `char *`
  - `%x` hexadecimal (useful for printing pointers)
  - `%f` `float`
  - `%u` `unsigned int`
- Use `%%` to print a literal `%` symbol

# printf

- Pass the values to be substituted in as extra parameters
- Must be in the order of the corresponding format flags
- **Make sure the types match!**
  - gcc will warn you if you're off
- Do not use %n – source of security bugs

# Bitwise Operations

- No `bool` type in ANSI (the old) C
  - Added in C99, `#include <stdbool.h>`
- Often need to track multiple boolean values
- Can combine into one metadata `int`
- Check if a particular bit is set to retrieve a particular value with `&` operator
- Set a particular bit as necessary with `|` operator

# Bitwise Operations

- **Example:** `i_mode` field in `struct inode`
- **Checking if a file is executable:**

```
#define IEXEC 0100
struct inode i;
struct inode *p;
/* Hiding initialization */
if(i.i_mode & IEXEC)
 printf("i executable");
if(p->i_mode & IEXEC)
 printf("p executable");
```

# Bitwise Operations

- **Example: setting a bit**

```
struct inode *p;
```

```
/* Hiding initialization of p */
```

```
p->i_mode = p->i_mode | IEXEC;
```

```
/* p now marked as executable */
```

- **Shorthand:** `p->i_mode |= IEXEC;`

# Bitwise Operations

- **Example: clearing a bit**

```
struct inode *p;
/* Hiding initialization of p */
p->i_mode = p->i_mode & ~IEXEC;
/* p now marked as non-executable */
```

- **Shorthand:**

```
- p->i_mode &= ~IEXEC;
```

- **~ operator takes the bitwise inversion of an integer's bit representation**

# Bitwise Operations

- Left (<<) and right (>>) bit shifts
- Take the bit representation and shift the bits over by the specified number:

```
x=4; x = x << 1; printf("%d\n", x);
/* previous line prints 8 */
x=4; x = x >> 1; printf("%d\n", x);
/* previous line prints 2 */
```

- Bits that “fall off” are dropped, and “empty bits” are set to zero
- Shorthand forms: >>= and <<=

# Bitwise Operations

- **Example: Bit shifting**

```
uint8_t x = 4; //00000100 in bits
x = x << 1; //00001000 in bits
printf("%d\n", x); //Prints 8
x = 128; //10000000 in bits
x = x << 1; //00000000 in bits
printf("%d\n", x);
```

# Bitwise Operations

```
struct inode {
 /* Hidden here */
 uint8_t i_size0; /* most significant of size */
 uint16_t i_size1; /* least sig */
 /* Hidden here */
};

int
inode_getsize(struct inode *inp) {
 return ((inp->i_size0 << 16) | inp->i_size1);
}
```

- `inode_getsize` should make sense now
- Shift the 8 most significant bits over by 16
- Bitwise OR in the 16 low bits to find the size

# Function Pointers

- Literally pointers to functions
- Pros:
  - Allow for run-time selection of function to use
  - Used by the standard C function `qsort`
  - Allow for callbacks and listeners in C
- Cons:
  - Slightly awkward syntax

# Function Pointers : Example

```
int max(int a, int b) {
 return a >= b ? a : b;
}
```

...

```
int (*func)(int, int);
/* <Return type> (*<name of pointer>)
 (<argument types>); */
func = NULL;
func = max;
int m = func(2,3); /* after this m = 3 */
```

# char \*

- Strings in C are represented as `char *`'s – arrays of characters
- `char *`'s don't carry their length with them (unlike C++/Java strings) so it is passed as another argument:
  - `void foo(char *buf, int len) { ... }`
- `char *`'s are `'\0'` (null) terminated
  - Extra character not counted in length

# char \*

- char \*'s can also be fixed-length
  - Then not necessarily null-terminated
- In Assn 1, 1 disk block is represented as
  - `char buf[512];`
- Not necessarily null terminated
- Do not do `char *buf[512];`
- Creates an array of 512 pointers!

# C Conventions

- Return 0 (or positive number) on success
- Return -1 on failure
- E.g. `diskimg_readsector`
  - Takes disk block number, reads block into memory
  - Returns 512 if successful
  - Returns -1 on failure

# C Conventions

- Take pointer to buffer
- On success: fill buffer with valid info, return success code
- On failure: return `-1`
- E.g. `diskimg_readsector`
  - `int diskimg_readsector(int fd, int sectorNum, void *buf) { ... }`
  - On success, `buf` is populated with data from disk
  - Otherwise untouched – cannot rely on its validity

# C Conventions

- Module name is included in function names
- Examples:
  - `file_getblock()` // in `file.c` and `file.h`
  - `diskimg_readsector()` // in `diskimg.h` and `diskimg.c`
  - `inode_iget()` // in `inode.h` and `inode.c`
- Most important that this applies to the functions exposed
  - Exposed functions should be prototyped in the `.h` file

# Tips & Tricks

- When switching from `corn` to `myth` (or vice versa), make clean and then recompile
- Make sure your code works on all machines!
- Use `fprintf(stderr, ...)` instead of `printf(...)` for debugging
  - Won't confuse grading scripts
  - Make sure you remove before submission!

# Tips & Tricks

- **Printf guide:**

<http://www.cplusplus.com/reference/cstdio/printf/>