Homework #1
- Sept 14

Reading
- Finish Chapter 1
- Start Chapter 2

Last time ……
Who can remind us what we covered last time?

- Memory Layout
  - Like a giant array
  - Addressed with integers
- Bitwise manipulations
Now some C....

```c
#include <stdlib.h>
#include <stdio.h>

int main (void) {
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```

What's in a header?

Preprocessor definition:
- `#define EXIT_SUCCESS 0`

Function prototypes
- Declare the argument/return types
- Example:
  ```c
  int factorial(int x);
  ```
  Note: ends in semi-colon.

External variable declarations (later)
Type declarations (also later)

A first approximation:
```c
/* maybe this is the prototype from stdio?*/
void printf(string x);

int main (void) {
    printf("Hello World\n");
    return 0; /* Replace #define'd symbol */
}
```
Close, but

Remember: C doesn’t have a “string” type
- char * represents a sequence of characters

printf actually
- Returns int (number of characters printed)
- Takes a variable number of arguments:
  ```c
  int printf(char * format, ...);
  ```

A variable number of arguments?

First arg: format specifier string
- Contains % directives (%d, %s, …)
Later args: values to replace % directives
Example:
```c
int x = 3;
char * s = “some string”;
printf(“x is %d and s is %s in”, x, s);
```

Basically right:

```c
int printf(char * format, ...);

int main (void) {
  printf(“Hello World\n”);
  return 0;
}

(Actually, lots of other stuff defined in those headers)
```

Two types of #include

```c
#include <stdio.h>
- Use angle brackets for system files
#include “myheader.h”
- Use quotes for header files in the local directory (i.e. that you wrote/were provided)
```
Basic syntax: just like Java

This could be C or Java:

```c
int i;
int counter = 0;
for ( i = 0; i < 10 ; i ++ ) {
    if (x[i] < y[i]) {
        counter++;
    }
}
return counter;
```

Caveat:

What about

```c
for (int i = 0; i < 10 ; i ++ ) {
...
}
```

Error: ‘for’ loop initial declaration used outside C99 mode

Solution: compile with -std=c99

Editing/compiling code

TA should have covered login (ssh) to linux.cs

Editing:
- Recommendation: programming editor
  - Emacs: my personal choice
  - Vim: people who love it, love it. I hate it

Compilation:
```bash
gcc -o outputName inputFile.c
```

Note: Your homework must compile/run with gcc on linux
- Use other compiler at home? Fine...
- Check on cs machines to avoid issues

In Java...

```java
int is a 4 byte signed integer
short is 2 byte signed integer
byte is a 1 byte signed integer
char is a 2 byte unsigned unicode char
float is an IEEE single precision number etc..
```
In C...

how many bytes is int?
short?
char?
(byte doesn’t exist)

In C...

how many bytes is int: sizeof(int)
short: sizeof(short)
char: sizeof(char)
Types can be “unsigned”
sizeof(expr) works too
int x;
sizeof(x); /* same as sizeof(int) */

Pointers

For any type T, T* is a “pointer to a T”
- int * = pointer to an int
- int ** = pointer to a pointer to an int

Specifies where in memory something is
Two operators:
- Address-of (&): “give me a pointer to”
- Dereference (*): “give me what is pointed to by”

Pointer example

int x = 307;
int y = 4115;
int * px = &x;
int * py = &y;
int ** ppx = &px;

What are the types
and values of *px, *py, *ppx,
and **ppx?
Strings: really just pointers

No string type
char * : pointer to a sequence of chars
“String”s end with ‘\0’ (null-terminator)
  • Don’t confuse the char ‘\0’ with a NULL pointer
String literals are read only
  • “Hello” has type const char *

“Hello”+3 = “lo” ?

Structs

Combine data together
  • size = sum of sizes of members
Predecessor of classes (C has no classes/objs)

Example:
```c
struct point {
    int x;
    int y;
};
```
```c
struct point p1;
struct point * p1ptr = &p1;
p1.x = 3;
p1.y = 4;
z = p1ptr->x + p1ptr->y;
```

More useful example
Linked List:
```c
struct llnode {
    int data;
    struct llnode * next;
};
```

Why must next be a pointer?
How do we allocate new nodes?
**Sidenote: typedef**

Always writing "struct llnode" = annoying
typedef lets you give a type a new name:
typedef struct llnode node;
Newly defined type is interchangeable with type it is defined as.

**Introducing malloc**

malloc allocates memory (from the heap)
- one argument: number of bytes to allocate
- returns: a pointer to the newly allocated memory
- returns NULL on error
- prototype in stdlib.h
Bad (very and horribly bad):
int * array = malloc(64);
Better:
int * array = malloc(16 * sizeof(int));
Best:
int * array = malloc(16 * sizeof(*array));

**Malloc for linked list**

```c
struct llnode {
    int data;
    struct llnode * next;
};
typedef struct llnode node;

node * n1 = malloc(sizeof(*n1));
n1->next = NULL;
n1->data = 42;
node * n2 = malloc(sizeof(*n2));
n2->next = n1;
n2->data = 17;
```

**You should be wondering..**

What type does malloc return?
- I've just assigned its return value to an int * and to a node *.....
For that matter, what type is NULL?
**Answer: void ***

- Pointer to any type
- Cannot be dereferenced
- Cannot be used for arithmetic
- Can assign any type of pointer to a `void *`
- Can assign a `void *` to any type of pointer

**More on void**

```c
void * malloc(size_t size);
#define NULL ((void*) 0)
Which of the following lines are legal?
void * v = malloc(sizeof(int));
int * p= v;
*p = 4;
```

**Answers:**

Which of the following lines are legal?

```c
void * v = malloc(sizeof(int));
int * p= v;
*p = 4;  // (Can't dereference void *)
*p = 4;  // Can't do arithmetic with void *
p = v+1;  // (Can't do arithmetic with void *)
v = &p;
```

**Example**

```c
int x = 42;
int f(int * p, int * q) {
    *p = 42;
    p[1] = *q;
    return p[2];
}
int main (void) {
    int * a = malloc (4 * sizeof(*a));
    a[0] = 0;  a[1] = 1;
    int ** b = & a;
    int c = f(a,42);
    **b = &c;
    return 0;
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0xF000</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0xEFEC</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0xEF8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEF4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEF0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFEC</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0x8000</td>
<td>42</td>
</tr>
</tbody>
</table>
int x = 42;

int f(int * p, int * q) { *p = 42; p[1] = *q; return p[2]; }

int main (void) {
    int * a = malloc (4 * sizeof (*a));
    a[0] = 0; a[1] = 1;
    int ** b = & a;
    int c = f(a, &x);
    **b = c;
    return 0;
}
int x = 42;
int f(int * p, int * q) {
    *p = 42;
    p[1] = *q;
    return p[2];
}

int main (void) {
    int * a = malloc (4 * sizeof(*a));
    a[0] = 0; a[1] = 1;
    int ** b = & a;
    int c = f(a,4x);
    **b = c;
    return 0;
}

CPS 104

Example

<table>
<thead>
<tr>
<th>Var</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0xF000</td>
<td>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td>0x8000</td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Var</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0xF000</td>
<td>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td>0x8000</td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
</tbody>
</table>
Example

```c
int x = 42;

int f(int * p, int * q) {
    *p = 42;
    return p[2];
}

int main (void) {
    int * a = malloc (4 * sizeof(*a));
    a[0] = 0; a[1] = 1;
    int ** b = & a;
    int c = f(a, &x);
    return 0;
}
```

CPS 104

Example

```c
int x = 42;

int f(int * p, int * q) {
    *p = 42;
    p[1] = *q;
    return p[2];
}

int main (void) {
    int * a = malloc (4 * sizeof(*a));
    a[0] = 0; a[1] = 1;
    int ** b = & a;
    int c = f(a, x);
    return 0;
}
```

CPS 104

Segmentation fault

Most common crash
Access to invalid memory address
- Dereferencing NULL
- Going past bounds of array (sometimes)
- Heap corruption (usually delayed)
- Writing read only data
Examples

```c
int * p = NULL;
*p = 4;
int * p; /* un-initialized */
*p = 4; /* probably will crash */
char * s = "Hello";
s[0] = 'x';
```

Free

C has no garbage collector
Must explicitly free dynamically allocated memory
```c
void free(void * ptr)
```
- ptr must have been allocated by malloc
- ptr cannot have been freed already
- after, any use of ptr incorrect

Bad things to do with free

Double freeing
```c
void * p = malloc(...);
.... free(p);
.... free(p);
```
ptr was not returned by malloc
```c
int * x = malloc(...);
int * p = x+1;
free(p);
```

More bad things

Use after free
```c
int * p = malloc(...);
x = p;
free(p);
s[0]= 4;
```
How the heap works

Not freeing the right pointer

Realloc

void * realloc(void * ptr, int size)

- changes the amount of space allocated to ptr
- ptr must be dynamically allocated with malloc (or realloc)
- returns NULL on failure
- realloc(NULL, x) is like malloc(x)
- realloc(x,0) is like free(x)

Example:

```c
int * p = malloc(4 * sizeof(*p));
int * p2;
....
p2 = realloc(p, 16*sizeof(*p2));
if (p2 == NULL) {
    /* p still valid, not enough space for 16*/
}
```
Input

char * fgets(char * s, int size, FILE * stream)
- reads until newline or EOF
- limit of size characters
- stdin is the stream for standard input
- returns s on success, NULL on error or when called at EOF

NEVER use char * gets(char * s)

C compilation

What's in a .o file?

Global variables
- “Mostly” assembled code
- Jump/call targets unresolved
- Global variable addresses unknown
  - Ld (x) => $r9

Need to fix these to make a binary

The linker’s job

“Glue” together object/library code
Resolve symbols => addresses
- call fact => call 0x40013770
Static

Duplicate symbols not permitted
static linkage
  • symbol only visible within file
  • Examples:
    static int x;
    static void foo() […]
Think “private”

Extern

Symbol definition is in another file
Tells compiler type info, but not to define it
Examples:
  • extern int x;
  • extern void foo();
    – Can be left implicit in function prototypes

Preprocessor

Macro definitions:
  • #define NULL ((void *) 0)
  • #define SQUARE(x) x*x
Conditional compilation
  #ifdef DEBUG
  #define DBGPRINT(x) printf(“%s\n”,x);
  #else
  #define DBGPRINT(x)
  #endif

Macro Caveats

Macro expansion: simple text replacement
What could go wrong with
  #define SQUARE(x) x*x
Macro Caveat 1

SQUARE(x+1) => x+1 * x+1
Always parenthesize macro argument use
- #define SQUARE(x) (x) * (x)

Macro Caveat 2

#define SQUARE(x) (x) *(x)
SQUARE(x++) = (x++) * (x++)
- Probably not what the programmer means
Never use side-effecting code as macro argument

Macro Mojo

- #stringify: text of macro argument
  - #define FOO(x) if(x) {printf( #x " is %d", x);}  
  FOO(ab + cd) =>
  if(ab + cd) {printf("ab + cd is %d", ab + cd);

- #token pasting: glue together
  - #define BAR(n) abc_##n  
  BAR(17) => abc_17

- Note: adjacent string literals: concatenated

Macro Mojo: why?

#define CHECK_NULL(ptr)  
if(ptr) {
  fprintf(stderr,  
    "Oops! " #ptr " is null at 
    __FILE__ " : %d",  
    __LINE__);  
  abort();  
}
More Macros

Newline ends a macro
Multiline? Use \n
#define X(abc) printf("Something %d\a", \n    abc);
Note: nothing (not even a space or a comment) can come between the \ and the newline!

Unions

Multiple variables that share space
union foo {
    int x;
    float f;
};
sizeof(union foo) = MAX(sizeof(int), sizeof(float))
Changing one changes the others
Why is this useful?

Enum

Enumerated type: many distinct constants
enum color_t {
    RED, BLUE, GREEN, PURPLE,
    ORANGE, YELLOW
};
Can specify specific values:
enum color_t {
    RED, BLUE=5, GREEN, PURPLE=7,...};
Advantages over simply #defining?

Man pages

Unix built-in help system
Usage:
• man malloc
• man -S3 printf (great for format specifiers!)
C library: section 3
• Unix system calls: section 2
• Commands: section 1
• man man for other sections (or help)
But if I don’t know what I need... how do I find it?

Keyword search:
  man -k regexp
Examples:
  • man -k “dynamic memory”
  • man -k “locate.*string”
Also useful: See Also section at end of a page

Function Pointers

Code for a function is in memory...
...so we can have a pointer to it
Function’s name is a pointer to it
(just like an array)
Can also have variables:
  int (*myFun) (int x) = factorial;
  ....
  myFun(3); /*Some people prefer (*myFun)(3)*/

Function Pointer Usage 1

Paramaterize a function over another function:
void doToList(lnode * head, void (*f)(int x)) {
  if(head == NULL) return;
  f(head->data);
  doToList(head->next, f);
}

Function Pointer Usage 2

Can be nicer than switch/case when all actions use/
result in the same type
typedef int (*eval) (int, int);
eval actions[] = {
  add, subtract, multiple, divide
};
  ....
  answer = actions[choice](a,b);
**Function Pointers in structs**

Consider:

```c
struct eval_node{
    int (*eval)(struct eval_node * me);
    int n;
    struct eval_node * left, * right;
};
typedef struct eval_node evn;
struct eval_node* newPlusEval(evn * left, evn * right)  {
    evn * ans = malloc (sizeof (*ans));
    ans->left = left;
    ans->right = right;
    ans->eval = addChildren;
    return ans;
}
```

That looks like crude OO

**dynamic dispatch = function pointers**

**`newPlusEval`: basically like constructor**

**Could add field/methods**

```c
c struct better_eval_node{
    struct eval_node parent;
    int anotherField;
    void (*foo)(struct better_eval_node * me);
};

• Could we do this?
  struct better_eval_node xyz;
  evn * foo = (evn *) &xyz;
```

**And thus began C++**

1979-1983: Bjarne Stroustrup worked on "C with classes"

*Cfront* translated C++ to C

More features of C++ came about later

**Wrap up**

**Today: Quick overview of C**

- Most important: pointers/malloc/free (again)
- Tried to hit most differences vs Java

**Next time: Debugging**

- Debugging = scientific method!
- `gdb`