Admin

Homework #1
  • Due Feb 6

Reading
  • Finish Chapter 1
  • Start Chapter 2
Last time …

Who can remind us what we covered last time?
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....

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

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

© Alvin R. Lebeck / Andrew Hilton
C compilation
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:

/* 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:
    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:
  int x = 3;
  char * s = “some string”;
  printf(“x is %d and s is %s \n”, 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`

`#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.cs
  – Use other compiler at home? Fine...
  – Check on linux.cs to avoid issues
In 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”
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?
Java: final int foo = 3;
C: const int foo = 3;

More complicated with pointers:

• const int * p; *p is const [think: (const int) *]
  – Same: int const * p;
• int *const p; p is const
• const int * const p; p & *p both const
• int * const ** const * p; *p & ***p both const
Arrays: really just pointers

int foo[6];
int * p = foo;
foo is just a pointer to element 0
*p and foo[0] are equivalent
  • So foo[1] is equivalent to…?
Pointer arithmetic

Pointer = memory address = number
Can do arithmetic with them:

```c
int foo[6];
int * p = foo;
int * p2 = p+1;
foo[x] equivalent to *(foo+x)
```

Note: the compiler makes it `foo+x*sizeof(int)`

Stupid C trivia: `foo[x]=*(foo+x)=*(x+foo)=x[foo]`
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” ?

```c
char * s = "Hello\n"
char * s2 = s + 3;
printf("%s", s2);
```
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;
};

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):

```c
int * array = malloc(64);
```

Better:

```c
int * array = malloc(16 * sizeof(int));
```

Best:

```c
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*

void * malloc(size_t size);
#define NULL ((void*) 0)

Which of the following lines are legal?

    void * v = malloc(sizeof(int));
    int * p = v;
    *v = 4;
    *p = 4;
    p = v+1;
    v = &p;
Answers:

Which of the following lines are legal?

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

```
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;
}
```

<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>0xEFFC</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFF4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFF0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x800C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8008</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8004</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0x4000</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;
}
Example

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;
}
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) ;
    **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>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFF4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFF0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xEFEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x800C</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x8008</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0x8004</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x8000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0x4000</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;
}
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) ;
  **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>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td>&lt;RA&gt;</td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1038</td>
<td></td>
</tr>
<tr>
<td>0x800C</td>
<td>3</td>
</tr>
<tr>
<td>0x8008</td>
<td>2</td>
</tr>
<tr>
<td>0x8004</td>
<td>1</td>
</tr>
<tr>
<td>0x8000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>x</td>
<td>0x4000</td>
</tr>
</tbody>
</table>
Example

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;
}

<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></td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td>&lt;RA&gt;</td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x800C</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x8008</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0x8004</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x8000</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0x4000</td>
<td>42</td>
</tr>
</tbody>
</table>
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) ;
    **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>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td>&lt;RA&gt;</td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck / Andrew Hilton
Example

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;
}
Example

```
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;
}
```

<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></td>
</tr>
<tr>
<td>p</td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td>q</td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td>&lt;RA&gt;</td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
<tr>
<td>x</td>
<td>0x4000</td>
<td>42</td>
</tr>
</tbody>
</table>
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) ;
    **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>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td></td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td></td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x800C</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x8008</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0x8004</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>0x8000</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0x4000</td>
<td>42</td>
</tr>
</tbody>
</table>
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) ;
    **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>0x8000</td>
</tr>
<tr>
<td>b</td>
<td>0xEFFC</td>
<td>0xF000</td>
</tr>
<tr>
<td>c</td>
<td>0xEFF8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0xEFF4</td>
<td>0x8000</td>
</tr>
<tr>
<td></td>
<td>0xEFF0</td>
<td>0x4000</td>
</tr>
<tr>
<td></td>
<td>0xEFEC</td>
<td>0x1038</td>
</tr>
<tr>
<td>x</td>
<td>0x4000</td>
<td>42</td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck / Andrew Hilton

CPS 104 46
Most common crash

Access to invalid memory address

- Dereferencing NULL
- Going past bounds of array (sometimes)
- Heap corruption (usually delayed)
- Writing read only data

Segmentation fault

© Alvin R. Lebeck / Andrew Hilton
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 `free()`ed already
- after, any use of `ptr` incorrect
Bad things to do with free

Double freeing

    void * p = malloc(...);
    .... free(p);
    .... free(p);

ptr was not returned by malloc

    int * x = malloc(...);
    int * p = x+1;
    free(p);
More bad things

Use after free

```c
int * p = malloc(....);

x = p;

free(p);

x[0]= 4;
```
How the heap works

x = malloc(490)

free(p2)
Not freeing the right pointer

\[ y = p1 + 100 \]
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)
Realloc

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
  – call factorial

• 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

```c
#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\n”, 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 \n        __FILE__ \: %d\n",
        __LINE__);
    abort();
}
More Macros

Newline ends a macro
Multiline? Use \
#define X(abc) printf("Something %d\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) = \text{MAX}(\text{sizeof(int)}, \text{sizeof(float)})

Changing one changes the others

Why is this useful?
Enum

Enumerated type: many distinct constants

```c
enum color_t {
    RED, BLUE, GREEN, PURPLE,
    ORANGE, YELLOW
};
```

Can specify specific values:

```c
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:

```c
void doToList(llnode * head, void (*f)(int x)) {
    if(head == NULL) return;
    f(head->data);
    doToList(head->next, f);
}
```
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;
    eval = addChildren;
    return ans;
}
```
That looks like crude OO

dynamic dispatch = function pointers
newPlusEval: basically like constructor
Could add field/methods

```
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