Data

variables

- **in C:** type variable_name = initializer;
  - aligned by default
  - e.g.: int x = 0; char v; double z;
- **in ASM:** variable_name: type initializer
  - e.g.
    ```
    .align 8
    x:
    .quad 0
    ```
  - *Make sure you are in .data segment!*
  - *Alignment directives are **not*** sticky
    - e.g.
      ```
      .align 8
      .byte 1
      z:
      .quad 0      # the label z is NOT on an 8 byte boundary!
      ```

Arrays

- **in C:** type variable_name[constant_size] = { };
  - e.g. int x[4] = { 0, 1, 2, 3 }; int z[5];
- **in ASM:**
  - e.g.:

```
.align 2
x:
.word 0, 1, 2, 3

.align 2
z:
.skip 20
```

Strings

- *Strings are arrays*
- **in C**: char s[] = “Hello world!”;
  - Notice how C allows for *constant initialized arrays* to be declared without a size. The size is *inferred*.
  - Personally I like this syntax: char *s = “Hello world!”
- **in ASM**: .string “Hello world!”
  - some tool chains like .asciz “Hello world!”

Pointers

- Pointers are integral to how computers work.
- Pointers are the source of a large fraction of errors.
- Modern languages work hard to remove pointers.
- You have to learn to love pointers.

Pointers, a processors view

- When a processor accesses memory it *always* uses a pointer.

  - e.g: mov *%rdi, %rax
    mov 0(%rdi), %rax
    # %rdi is the pointer and 0 is a value being added to it during the load

  - even when you write code such as:
    mov x, %rax
    # the processor really sees this as:
    mov 154243(%rip), %rax

Pointers in C

- Pointers in C are declared by prefacing with a *

  -
    ```
    int  *x;   // declare 8 bytes to hold the pointer
    x = NULL;
    ++x; // x now equals NULL + sizeof(int)  or 4
    ```
- If you have a pointer you usually want it to point to something. The **address-of** something is found with the & operator

  -
    ```
    int y, *x;
    x = &y;
    ```
- If you want to set the thing the pointer points to you do so with the **dereference** operator

  -
    ```
    int y, *x;
    x = &y;
    *x = 1;
    y = y * (*x);
    ```

SUPER BOWL XLVIII
CHAMPIONS
12
HAWKS
12
14

Pointers in Assembly

- Pointers in assembly are just bits

    -
      ```
      .align 8
      x_ptr: .quad 0
      ```
- Pointers are set like writing any other bits

    -
      ```
      .data
      .align 4
      y: .word 0
      .align 8
      x: .quad 0

      .code
      leaq y(%rip), %rax  # Load the address of y into rax
      mov %rax, x(%rip)  # store that address into the pointer
      ```
- Pointers are dereferenced like any other address usage

    -
      ```
      mov x(%rip), %rax  #Load the pointer to rax
      movl $1, 0(%rax)    # store a 32bit value to the memory location pointed to by x
      ```

What’s going on with %rip?

- Data lives *somewhere* in the 64 bit address space
- x64 (and basically every processor I’ve ever written code for) does not support a 64 bit constant encoded in the instruction. i.e., not this:
    - mov ($0x123456789abcdef), %rax
- Thus, runtime environments have the concept of a “gp” or “global pointer” region
    - on x64 this is now, rather funkily folded into the instruction pointer!

Array bounds

- Arrays in C are 0 indexed, i.e.:

  -
    ```
    int   x[5];
    x[0] = 7;


    z = x[5]; // NON SENSICAL or CRASH!
    x[-1] = 6; // You no longer work here…
    ```

More on boundlessness

- C makes **no assurances to you** about the relative order of data items. I.e.:

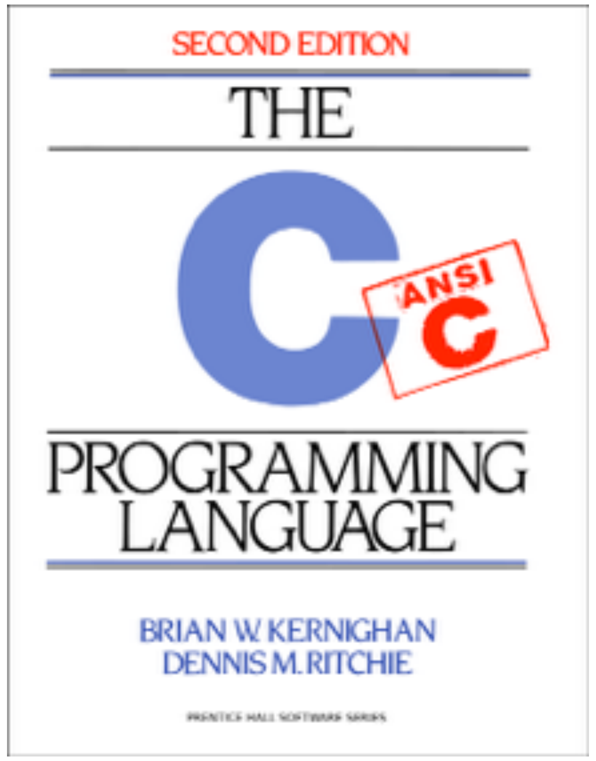
```
int x[1];
int y;

void foo() {
    x[0] = 0; // good
    x[1] = 1; // bounds error, does
              // **not** necessarily set y to 1
```

Same goes for the stack

-
  ```
  void foo() {
      int x;
      int y;
       …
  ```

- The y is not assured to be before or after x in memory. In fact, neither x nor y are assured to even be allocated memory locations!

C


SECOND EDITION

THE C PROGRAMMING LANGUAGE

ANSI C

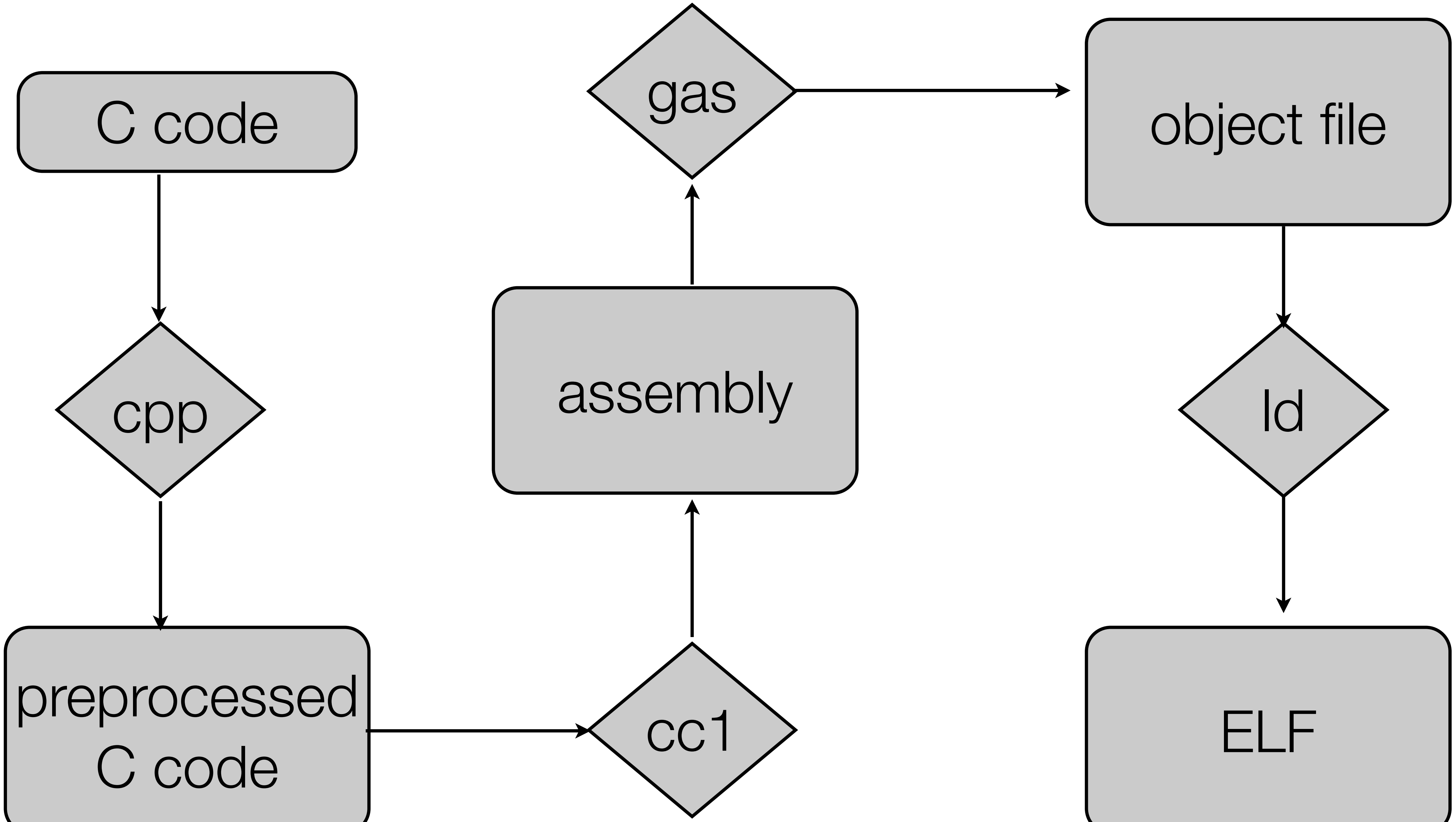
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

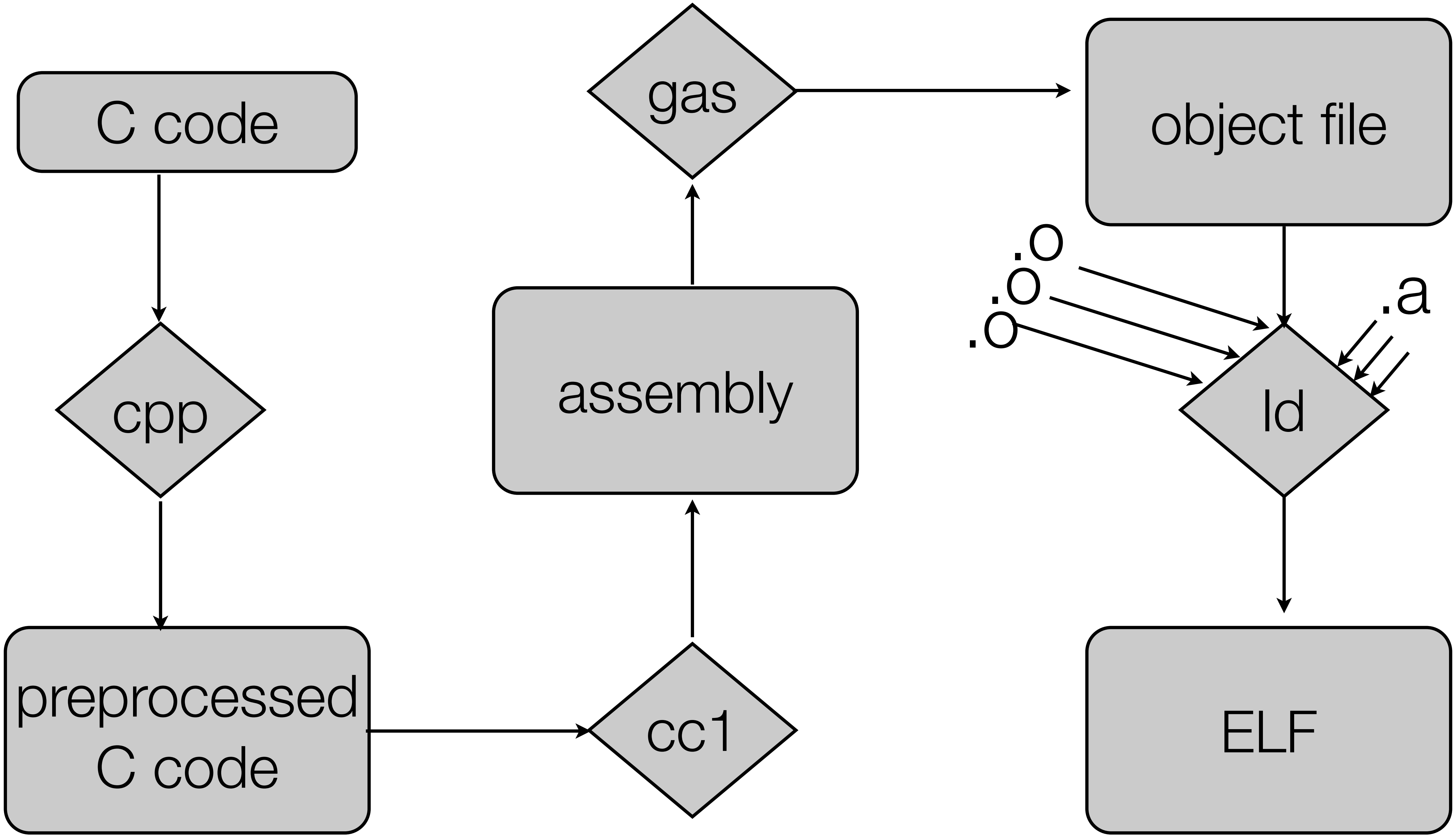
Why do we teach you C?

- C closely matches how computers really work
  - No hidden “runtime”
    - Can “trust” the compiler (sort of, more on this later)
  - Helps you understand how systems work
- Still an extremely important language
  - The Linux, Mac OS X, FreeBSD, and Windows kernels are all written in C
  - VMware, Xen, KVM, and other hypervisors are written in C
  - X windows is written in C
  - The list goes on and on, but pretty much all major systems products are written in C or C++ (which in the 90’s was C w/sugar, but is no longer)
    - With one major exception: the UI on Mac OS X uses Objective-C -- but major portions of the guts of the UI are still C (requiring some very badly written programs).
- Being proficient in C makes you more marketable
  - The world has 10M developers, but only 1M C/ASM developers. Guess who gets paid more? And, I would argue, has the more interesting job.

From C to ELF


C code
cpp
preprocessed C code
cc1
assembly
gas
object file
ld
ELF

From C to ELF


C code
cpp
preprocessed C code
cc1
assembly
gas
object file
.o
.o
.o
.a
ld
ELF

C topics

- The preprocessor
- Hello world!
- functions
- constructs
- The type system
- libc,libm
- assembly translation

cpp: The C preprocessor (invoke direct or “gcc -E”)

- cpp is an amazing thing (although Bjarne Stroustrup despised it)
- cpp is a *preprocessor* that is ran over your code before the C compiler actually gets it.
  - cpp is essentially a text processor that inputs a text file and outputs another text file.
- Responsible for numerous things in the C language, but of most importance:
- #include <file.h>
  - /* insert the file “file.h” into this file before compilation */
- #define MY_FUNKY_CONSTANT (-42)
  - /* translate all sightings of MY_FUNKY_CONSTANT into (-42) */
- #if MY_FUNKY_CONSTANT == (-42)
  /* insert my random code into the file */
  #else
  /* insert some other random code into the file */
  #endif
- __LINE__, __FILE__, __FUNCTION__ /* predefined macros */

#define

- #define is quite useful and has both obvious and extremely obscure syntax
- #define A (1) /* define A to be (1) */
- #define MAX(a,b) ((a) < (b) ? (b) : (a)) /* return the max of a or b */
- #define DBG(x) printf(“%s = %d\n”, #x, x) /* see below */
  DBG(my_int) is translated into: printf(“%s = %d\n”, “my_int”, my_int)
- #define CAT(x,y) x##y /* take symbols x and y and return the symbol xy */

- Also useful is:
  - #undef SYM /* *undefine* the macro SYM */

#define - advice

- Strive to **NEVER** have random numbers in your code.

-
  ```
  /* write-only code */
  if (input_field & 0x7f == 0x20)
      return 0x20;
  ```

-
  ```
  /* code with staying power */
  #define ASCII_LOWER_MASK 0x7f
  #define SPACE 0x20
  if (input_field & ASCII_LOWER_MASK == SPACE)
      return SPACE
  ```

#define - Words of caution

- **Caution:** there is no type system in macros.
- **Caution 2:** macro replacement is not aware of operator precedence
  - Always surround weak operators with ( ) your macros
    - /* BAD */
      #define ARRAY_SIZE 1000+500
      return ARRAY_SIZE * 5; /* I want the space for 5 arrays but that is not what I am going to get */

    - /* GOOD */
      #define ARRAY_SIZE (1000+500)

  - Always assume inputs to macros include weak operators
    - /* BAD */
      #define CALC_ARRAY_SIZE(a) a*5
    - /* GOOD */
      #define CALC_ARRAY_SIZE(a) ((a)*5)

#if, #ifdef - conditional compilation

- Conditionally, based on *statically knowable data* include blocks of code

-
  ```
  #if SYM == 1
  /* include this code */
  #elif SYM == 2
  /* include this code */
  #else
  /* include this code */
  #endif
  ```

-
  ```
  #ifdef SYM
  /* if the symbol SYM is defined in the preprocessor, include this code */
  #endif
  ```
- Often you will see code like this in a header file:

  -
    ```
    #ifndef _stdio_h
    #define _stdio_h
    /* rest of the header file
    #endif
    ```

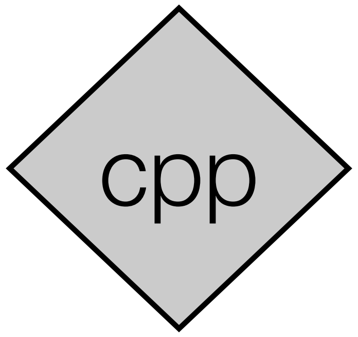
#if - **A Word of Caution**

- Code with conditional compilation in it can be difficult to maintain over the lifecycle of a project. For example, the following code will compile:

  -
    ```
    #define SYM
    #if !defined(SYM)
    int some_garbage that wont compile;
    #else
    int some_perfectly_legit_code;
    #endif
    ```
  - This sort of code can “live” in a code base for a very long time before it is discovered that the non-compiled branch of the code is broken. This is known as “bit rot”
  - **Advice:** Whenever possible don’t use conditional compilation. A single branch comparison is pretty fast these days and you won’t save much from avoid it. But there are plenty of legitimate uses for conditional compilation, so when you have to use it, do so.

Example

```
#define ARRAY_LENGTH (128)
#define LINEAR

int my_array[ARRAY_LENGTH];

int initialize_array() {
    int i;
    for (i = 0; i < ARRAY_LENGTH; i++)
#ifdef LINEAR
        my_array[i] = i;
#else
        my_array[i] = 0;
#endif
    }
```



```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "test.c"


int my_array[(128)];

int initialize_array() {
    int i;
    for (i = 0; i < (128); i++)
        my_array[i] = i;
    }
```

Hello world!

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf(“Hello world!\n”);
   return 0;
   }
```

Hello world!

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf(“Hello world!\n”);
   return 0;
   }
```

Contains the *function prototype* for printf. A function prototype is just a description of the function, not the implementation.

Hello world!

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf(“Hello world!\n”);
   return 0;
   }
```

Programs that link with libc/crt0 have a “main” function that is invoked after libc has been initialized.

main(...) returns an integer, however, most compilers will accept void main(...) as well, but this is just for old sloppy code. The value returned from main is passed back to the shell. As a matter of convention “0” typically means “no error”.

Hello world!

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf("Hello world!\n");
   return 0;
   }
```

main takes two arguments, a count of the number of arguments in the argv array and the argv array, which is a pointer to an array of pointers to strings.

Some compilers also support main(int argc, char *argv[], char *arge[]) where the 3rd argument is a list of environment variables. (Not standard as far as I know).

Hello world!

```
#include <stdio.h>

int main(int argc, char *ar
   printf("Hello world!\n");
   return 0;
   }
```

Call the function printf, which is contain in libc to print the string "Hello world!" followed by a newline to the console.

Mildly important: \n means insert a "newline" in the output. Sadly, there is not 1 definition of how a newline is printed to a screen. On MS-DOS/Windows systems it is the ASCII character 13 followed by 10. On Unix / Mac OS X systems it is just the ASCII character 10.

By standard, "\n" appends an ASCII 10 to the string in your program, and then code in libc either prints out 13/10 or 10 depending on the platform.

Hello world!

ANSI C or C99

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf("Hello world!\n");
   return 0;
   }
```

Old-school K&R C

```
#include <stdio.h>

int main(argc, argv)
    int argc;
    char *argv[];
    {
    printf("Hello world!\n");
    }
```

Do not use this! But be aware it exists! Lots of old code out there!

.h versus .c files

- .h is short for “header file”.
    - usually .c files include header files, although header files often include other header files (more on this later).
    - A header file is just like any other C file.
    - Typically a header file is used to specify the *interface* to a piece of code contained in a .c file.
        - There are exceptions to this:
            - Header files that specify basic types
            - Header files that specify interfaces to other source files (e.g. .s)
            - Header files that specify only static inline functions
- Header files are included included into other files in two ways:
    - #include <file.h> /* used to indicate system header search path */
    - #include “file.h” /* used to indicate an exact path. e.g. “include/file.h” */

header files - best practices

- Group and alphabetize your includes. Why? Because it will minimize annoying source-control repository conflicts. e.g.

    -
      ```
      #include <stdio.h>
      #include <stdlib.h>
      #include “my_project/include/a.h”
      #include “my_project/include/b.h”
      ```
- Compilers almost always have a “-I” flag to add a path to the standard include path. It has some use but can be abused easily. In general, I recommend
    - Use the -I flag as a “major” switch (switching target for example, like the Linux kernel).
    - Use the full path (relative to the base of the project) in other situations.

Header files - best practices 2

- ALWAYS make sure your header file can be included more than once without causing trouble. Do this in one of two ways:
  - Old school:
    ```
    #ifndef _my_header_h
    #define _my_header_h
    /* your normal stuff here */
    #endif
    ```
  - New way (and while not standard is almost universally accepted)
    ```
    #pragma once
    /* your normal stuff here */
    ```
  - *There are exceptions to this "include once" rule. Namely for cpp generated code! Use this very very sparingly.*

header files - best practices 3

- Make your header files self-contained.
  - BAD:
    uint64_t my_function(my_funky_type x);
  - GOOD:
    #include <inttypes.h>
    #include “my_funky_header.h” // where my_funky_type is defined
    uint64_t my_function(my_funky_type x);
    - Advantages are self-contained header files are easier to read and they cannot have their meaning changed by improper usage!
- Expose as little as possible from your .c code to the outside world with a .h file and nothing more.
  - Declare as much as you can static and don’t add it to the header file
  - Use opaque types (more on this later).

Function declarations versus implementation

- In C you declare how to call a function and what it returns. This is done *separately* from implementing it.
  - The *declaration* is often, but not always, included in the header file for a module.
- Example:
  - int my_function(char *s); /* this is a declaration */
  - int my_function(char *s) { /* this is the implementation */
        return atoi(s);
    }
- **static** functions do not need a declaration. Once the implementation is "seen", the compiler automatically infers the declaration **for all code after the implementation**.
  - This is the one reason you may want a declaration for a static function. So you can "use" it before the compiler has seen the implementation. e.g.
    - static my_private_function(int x);

Functions

- return_type function_name(argtype_one arg1, argtype_two arg2 /* more? */) {
```
    private_variable_to_invocation_instance private_var1;
    /* code */
    return whatever_I_mean_to_return;
}
```
- For example:
- int add2(int x, int y) {
```
  int z;
  z = x + y;
  return z;
}
```

Function, 2

- Arguments are either passed by value or passed by reference.
  - passed by value: void bar(int x); /* a **COPY** of x is passed to bar */
    passed by reference: void bar(int *x); /* a **POINTER** to x is passed to bar */
  - **Some arguments are written as if they are passed by value, when in fact, they are passed by reference -> namely, arrays are always passed by reference.**
    - void foo(char a[10]); /* looks like it is passed by value, but it is not */
    - void foo(char *a); /* it is basically the same as this modulo some type differences */
- Obviously, how arguments are used will change whether they are passed by value or reference:
  - int times2(int x) { return x * 2; }
  - int times2(int *x) { return (*x) * 2; }
- **Structures are passed by value** unless you explicitly state otherwise.
  - Be aware that these can be large...

Function, 3

- In C99 it is possible (and in some places preferred) to declare your variables wherever you like within a function. e.g.

-
  ```
  /* old school */
  void my_function() {
       int i;
       for (i = 0; i < 10; i++) {
  ....
  ```

-
  ```
  /* new style */
  void my_function() {
      for(int i = 0; i < 10; i++) {
  ....
  ```

- I have no substantive comments on which is better here. Note that in the old and new styles you can declare variables at the top of any scope-opening block. E.g., even in old-school C this is legit:
  - `void my_function() { int x; for(x = 0; x < 10; x++) { int y; y = 2 * x; ....`

Functions 4

- Functions can optionally return a value.
- **Best practice:** make the return value meaningful or don't use it at all.

-
  ```
  /* BAD */
  int fire_me() {
      /* some code */
      return 0;
  }
  ```

-
  ```
  /* GOOD */
  void hire_me() {
      /* some code */
  }
  ```
- Functions that always return the same value are (a) lame and (b) lead to sloppy thinking in the future. Should this return value be stored? Should it be checked? Think of the children (the developers that come after you to the code base).

Code blocks

- block of code: { }
- e.g.:
- if (x) {
      printf(“x not equal to 0!\n”);
      ++x;
  }

- Dark arts: the “,” “operator” can sometimes make compound expressions. e.g, x = (y = 0, z = 1, x + y) /* x= the last expression in the ( ) */
  - You almost NEVER see code like this, and with good reason, it is inscrutable. You should not write code like unless you absolutely have to. The one case I know of is in some cpp generated code it can be helpful....

if statements

- if (val) statement   e.g.
- if (x)
      printf(“x is not equal to 0!\n”);
- if (x) {
      printf(“x is not equal to 0!\n”);
  } else
      printf(“x is equal to zero!\n”);
- You often see this in old school code, and it is bad:
  - if (x = foo())   /* bad form. Call foo() assign to x, then compare to 0 */
    - This is so bad, most compilers now can warn you about this!
  - if ((x = foo()) != 0) /* a little better form */
  - x = foo(); if (x) /* much better */
  - x = foo(); if (x != 0) /* best, it removes the cast from int to bool */
- **Recommendation:** avoid the implicit cast to bool if you can.
  - if (x != 0)   /* GOOD */
  - if (x)   /* eh */

More on if and the ? short-hand

- **Warning:** Conditional operators in C short-circuit. E.g.:
  - if (x && use_x(x)) { ... } /* is a very common programing technique. */
    - if x == NULL then use_x will **NOT** be evaluated.
  - if (x || foo()) { ... } /* foo() will **NOT** be invoked */

- C includes a useful, but easily abused short hand for an if statement, the ?

  -
    ```
    if (x < y)
      m = x;
    else
      m = y;
    ```
  - m = (x < y ? x : y); /* personally I think this is bad form */
  - **Recommendation:**
    - Use sparingly. But can be very useful when calling a function. e.g.:
      printf(“resulting string: %s\n”, (s == NULL ? “null” : s));

looping functions

- for (initialization; comparison; increment_expression)
    - for(i = 0; i < n; ++i) { }

    -
      ```
      i = 0;
      loop_head:
      /* loop body */
      ++i;
      if (i < n) goto loop_head
      ```
- what does for(;;) do?

while

- while(cond) { }
- Repeat the code after the while so long as the condition is true. e.g.:

-
  ```
  while (i < n) {
     printf("i = %d\n", i);
     ++i;
  }
  ```

-
  ```
  while (i < n)
     do_my_stuff(i++);
  ```
- Some people write code like but they should be fired:
  - while (i++ < n) printf("i = %d\n");
  - QUIZ: what is printed out if i = 0 before the loop and n = 4 ?
- **break;** is a very useful construct to exit a loop from the middle:

  -
    ```
    while (1)
       if (have_data())
          break;
       else
          sleep(1);
    ```
  - You can break out of for and do/while loops as well...

do/while

- do {
  /* my code */
  } while(cond);

- Do the code in the loop body at least once, and evaluate the condition on whether or not to do it some more.

- **Side note:** You often see code like this in cpp macros:
  - #define checkError(x) do { if ((x) != 0) { printf(“Error!!!\n” } } while(0)
  - This is good form, and is so you must write code like this:
    - checkError(x); /* that is *with* the trailing ; */
  - It also prevents code like this: if (checkError(x)) { … }

goto

- C functions can include labels and you can goto them.
-
```
int foo() {
   /* some code */
   if (error)
       goto error_out;
   /* more code */
error_out:
    return -1;
}
```

- Contrary to what you may have been led to believe, goto is not an inherently evil act. BUT, it is best practices to only ever *forward* goto.
  - A forward goto is often seen in systems code for just the purpose illustrated above. As a way to "error out" of a function in a controlled way.
  - If you are backward goto'ing, you should seriously think about why you are writing code like that. Think of the children... use a loop construct instead.

switch

-
  ```
  switch(integral_type) {
      case constant1:
          /* code */
          break;
      case constant2:
           ...
      default:
           /* if nothing above satisfies */
  }
  ```
- e.g.:
  ```
  switch(c) {
      case ‘c‘: printf(“user hit c!\n”); break;
      case  ‘a‘: printf(“user hit a!\n”); break;
      default: printf(“unrecognized input: %c\n”, c);
  }
  ```

switch 2....

- If you forget the **break** then the statements fall through. Which can be very useful sometimes....

```
switch (c) {
    case '\t':
    case '\n':
        printf("character is a tab or a newline!\n");
    }
```

- **Recommendation**: It is often the case that you want to switch on a value that has a finite number of constant possibilities. You could declare these constants as separate #defines (which is way better than just random numbers in your code, which will fry your brain and get you fired), but consider an **enum**. More on enum's later, but the advantage is with an enum the compiler can warn you about a **missing case.** e.g.:

```
    enum retValues { retVal1 = 1, retVal2 };
    void foo(enum retValues x) {
        switch(x) {
            case retVal1: printf("retValue1!!\n"); break;
        }
     }
```

  - The compiler will warn you "case retVal2 not handled"

return

- int times2(int x) { return 2 * x; }
- In a lot of code you will see “return (2*x);” but this is just cruft. Nevertheless, you will want / need to adhere to whatever the local style is.
- It is permissible to have multiple returns:

  -
    ```
    char toupper(char x) {
       if (x >= ‘a‘ && x <= ‘z‘)
          return x - ‘a‘ + ‘A‘;
       return x;
    }
    ```
  - BUT! It can be difficult to debug code like this. In the short example above, sure, but in larger functions, it can be subtle where it returns. So **use sparingly.**

Naming and conventions

- This is not part of C programing, but a useful topic nonetheless.
- There are 4 naming conventions in use in the world:
    - random
    - camelCaps
    - bsd_style
    - pszHungarianNotation
- You likely will not get to choose which convention you will use at first. Most organizations have a "style guide" that developers must adhere to.
- Personally, I prefer self-documenting code, and as a consequence, I like long names:
    - uint64_t vtop(uint64 a); /* bad, what does this function do again? */
    - uint64_t virtual_to_physical(uint64 in_address);
- People can become irrationally passionate about these things...

Comments

- Old-school comments:
  ```
  /* start comment
      it can go on for many lines
       it ends here.... */
  ```
- New-style one line comments (borrowed from C++)
  ```
      // everything from here to the end of a line is a comment
  ```
- These days almost all compilers support //

- Personally I am not a fan of comments (see previous slide on descriptive function names). Just write clear code with good names. Comments tend to be bad because they go “stale”. Code changes, but no one edits the comments.
  - But, comments that describe the non-obvious or the rational for something are useful.

Functions, summary and best practices

- **Large functions are not a sin, code duplication is.**
  - Even near-duplication is bad. Whenever two or more blocks of code look similar, consider a function to support them. Handle the differences with arguments and if statements.
- **If a function returns a value it should be stored or checked.**
- **Functions should be private (static) unless they are called elsewhere**
- **Use descriptive naming conventions**
- **Use consistent formatting style**
  - And to make your life easy, don’t invent your own. Just write how emacs (or whatever) wants you to write.

C type system

- Yes, C has a type system. It is pretty weak.
- Core types:
  - char, short, int, long, float, double
  - modifiers: signed, unsigned, const, *, ()
    - The * or “is a pointer” can apply to itself: e.g. int **i; // pointer to a pointer to an int
    - The () or function pointer declaration is tricky, as we’ll see in a second
    - non-standard but long held extensions: long long, long double
- Types can be grouped into arrays
  - e.g.:
    - char s[128];
    - int matrix[10][10];
  - Arrays are 0 based and row major:
    - char s[128]; s[0] = 0; s[128] = SILENT DATA CORRUPTION :-)
  - In standard C array bounds are static. e.g. char s[128]; gcc permits a *non-standard extension* of variable length declaration: char s[length];

Pointers to functions

- Suppose you have a function like:

```
int   foobar(char *s) { ... }
```

- You can declare a pointer to a function for functions of that type like this:

```
int (*function_ptr)(char *s);
```

- And set it like this:

```
function_ptr = foobar;
```

- And use it like this:

```
function_ptr(“hi!”);
```

Pointer arithmetic

- Arithmetic on pointers occurs on the integral of size of the type they point to.
  - e.g. int *x = 0; ++x; // now x = 4!

- This is generally what you want, but not always.
  - Specifically in systems code you often see code like this:
    ```
    unsigned char *b_ptr = (unsigned char *) i_ptr;
    b_ptr -= 0x80000000;
    i_ptr = (int *) b_ptr;
    ```

  - This sort of code also often appears when you are reading or writing specific file formats.

New types can be created

- A new type can be created:
  - typedef old_type_name new_type_name;
    - e.g.: typedef unsigned long long uint64_t;

- New types are very useful for:
  - size-specific datatypes (uint64_t, etc)
  - function-pointer types (simplifies a particularly ugly declaration that we'll see shortly).
  - short hand (although this can be taken to far)

- **Advice:** A lot of runtime environments (Windows, ...) like to declare pointer-to typedefs. For example, suppose there is a type "my_type", in these environments there is also generally a type "Pmy_type" (or ptr_my_type or whatever the naming convention in use is). This lets you write code like:
  - Pmy_type pointer_to_my_type;
  - personally I prefer: my_type *my_type_ptr;

sizeof

- Types have a size. sizeof returns the size of that type in **bytes**.

- sizeof(int) = 4
- sizeof(double) = 8
- sizeof(int *) = 8 // on 64 bit systems
- sizeof(int *) = 4 // on 32 bit systems

- sizeof also works on de-referenced pointers (which will not be dereferenced to get the size. e.g.:

  ```
  int  *x = NULL;
  printf("sizeof something %d\n", sizeof(*x)); // this will not crash! it just prints
  // "sizeof something 4" is printed
  ```

- sizeof is computable at **compilation time.** It is not a runtime computation. The compiler converts it (ultimately) to a number.

static

- C has a keyword “**static**” which is used for two purposes:
  - Purpose 1: Making things private to a module.
    - By default, variables declares outside of a function, and functions themselves become global symbols in the output object file. By prefixing their declaration with “static” they become private. eg.
      - static int x = 0;
      - static int foobar(int z) { }
      - static inline foo(...) { }
  - Purpose 2: Making local variables to a function persistent across function invocation.

```
void foobar() { static int x = 0; ++x; printf(“%d “, x); }
// output: 0 1 2 3 4 5 6 ...
```

struct’s

- struct is among the most useful things in the C type system. A struct creates a new type which is a collection of types:

```
struct new_type_name {
    int    x;
    int    y;
 };
```

- You then declare a struct like:

```
struct new_type_name   my_struct;
```

- And use it like:

```
my_struct.x = 1;
my_struct.y = my_struct.x + 1;
```

Pointers to struct's

- Suppose you have:

```
    struct new_type_name {
        int    x;
        int    y;
     };
```

- And then you declare:

```
    struct new_type_name actual_storage, *ptr_to_actual_storage;
```

- And then go:

```
   ptr_to_actual_storage = &actual_storage;
```

- You then access x and y not with '.' but with '->'

```
   ptr_to_actual_storage->x = 1;
   ptr_to_actual_storage->y = ptr_to_actual_storage->x + 1;
```

- This syntax is to make it clear what is being de-referenced when a struct contains a pointer. e.g. struct f {int *i} *p; does *p.i dereference p or i? To avoid this, the -> syntax was invented.

More on struct’s

- **Warning:** Unless you say otherwise, the compiler does not have to lay a struct out in memory the way you wrote it in your code. E.g.:

```
struct my_struct {
    char s1;
    char s2;
    int    i1;
    };
```

- **Question:** how big is sizeof(struct my_struct) ???

More on struct’s

- **Warning:** Unless you say otherwise, the compiler does not have to lay a struct out in memory the way you wrote it in your code. E.g.:

```
struct my_struct {
    char s1;
    char s2;
    int    i1;
    };
```

- **Question:** how big is sizeof(struct my_struct) ???
  - **Answer: it varies.** Depending on how you set the compilation flags it can vary from 6 to 8 bytes (or more if you work at it).

struct’s and alignment/packing

```
struct my_struct {
    char s1;
    char s2;
    int  i1;
    };
```

- default layout: [s1, s2, padding, padding, i1, i1, i1, i1]
- -fpack-struct   [s1, s2, i1, i1, i1, i1]

- Generally speaking you do not need to think about this for struct’s used purely to communicate between different parts of your own application. But when a struct needs to be passed to the operating system, or another application or you need to write a struct to match a file-format or something you see over the network, then packing & alignment matter.
  - -fpack-struct is generally frowned upon, use #pragma’s instead:

    ```
    #pragma pack(push, 1)
    struct my_struct { ... };
    #pragma pack(pop)
    ```

  - You can neglect the push/pop but then *everything* after the pack(1) directive becomes packed. Not cool...

struct and typedef

- A very common programming form you see in code is:

```
typedef struct _my_struct {
     int x;
     int y;
     } my_struct;
```

  - which simultaneously declares a struct of type _my_struct and a typedef to that struct of name my_struct. This lets you write code like:

```
my_struct z;   z.x = a;   // observe, no struct in front of it!
```

- Note that the typedef is **NOT VALID** until the end of the statement. So you need to use the struct name inside the struct declaration:

```
typedef struct _list {
    struct _list *next;
    } list;
```

union's

- A union is declared like a struct, but means something quite different!
```
union my_union {
    int i;
    char c[4];
};
```

- sizeof(struct my_union) = 4

- Memory is [i/c[0], i/c[1], i/c[2], i/c[3]]   that is, each element of the union *shares the same memory.*

- The size of the union is the size of the maximum element in the union.

union’s are very useful.

- e.g.:

```
union int_type { unsigned int x; unsigned char  c[4]; };

switch_endian(unsigned int y) {
    union int_type z1, z2;
    z.x = y;
    z2.c[0] = z1.c[3];
    z2.c[1] = z1.c[2];
    z2.c[2] = z1.c[1];
    z2.c[3] = z1.c[0];
    return z2.x;
}
```

- Also useful for when two concepts are mutually exclusive in reality and you want to save space:

```
struct cpu_data {
   enum cpu_type the_type;
   union cpu_data_raw {
        struct amd {
            ....
        } amd;
        struct intel {
            ....
```

struct's and bitfields

- In C it possible to say "this thing only has N bits"   e.g:

```
    struct rflags {
       uint64_t cf:1;
       uint64_t _dummy1:1;
       uint64_t  pf:1;
       uint64_t _dummy0:1;
       uint64_t  af:1;
       // etc
   };
```

- **Warning:** What does the "uint64_t" mean here? The answer is it depends! On MSVC the default is the type of the bitfield has meaning. **In C99/gcc the type of the bitfield *regardless of what you put there* is int.**   This is a very subtle distinction that can lead to bizarre bugs. GCC includes a flag "-mms-bitfields" to make it behave like MSVC in this case.

opaque types

Important: this is more important than it looks, and is what separates a “level 1” from a “level 2” SDE (among other things)

- Opaque types are very useful. They allow you to *force* people to use the interface to a module.
- There is not explicit support for opaque types in C, it is all in how you use the tools provided.

BAD

```
// header.h
struct my_data_type {
   int whatever;
};

void do_something(
     struct my_data_type *t);
```

GOOD

```
// header.h
struct my_data_type;

struct my_data *allocate_my_type();
void do_something(
     struct my_data_type *t);

// header.c
#include “header.h”
struct my_data_type {
    int whatever;
};

void do_something(
     struct my_data_type *t) { }
```

Caution: The C compiler believes you

```
// header1.h

struct my_type { int x; };


// file2.c
#include “header1.h”

struct my_type *allocate() {
   struct my_type *p;

p = malloc(sizeof(*p) * 10);
}
```

```
// header2.h

struct my_type { double x; };

// file2.c
#include “header2.h”

void silently_corrupt_data() {
   struct my_type *p;
   p = allocate();
   for (int i; i < 10; i++) {
      p[0] = 0;
}
```

Caution: C++ has a bool type

- C used to have no such thing as bool type
    - Although it is quite common to make one:
      typedef int bool;
      #define true 1
      #define false 0
    - And some compilers like to implement one for you...

- C++ has a bool type.
    - BUT THIS NEED NOT ALWAYS BE IMPLEMENTED IN THE SAME WAY.
        - Some compilers use 1 byte
        - Some compilers use 4

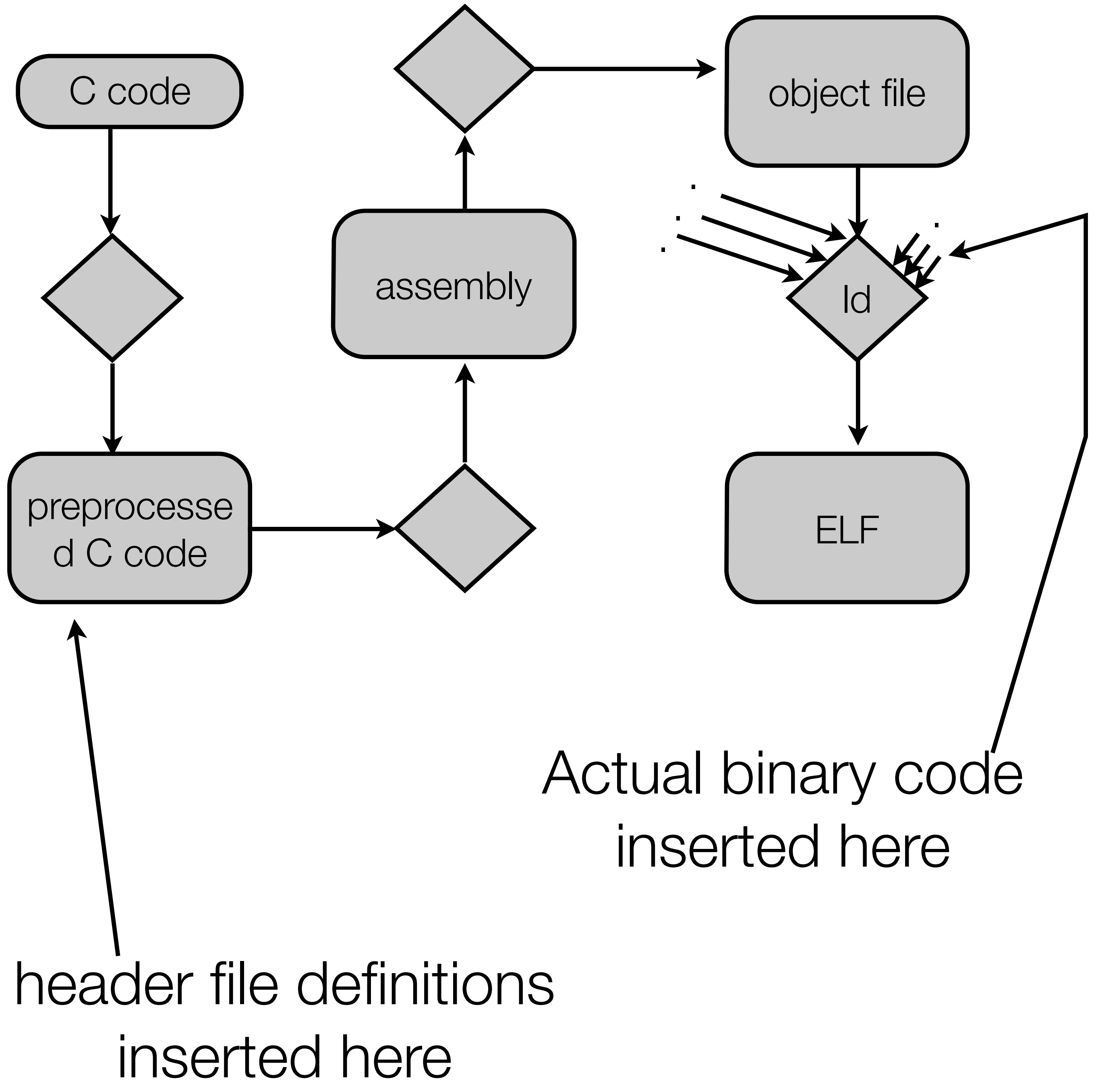
- C++ code can link against C code...

- And you guessed it... this story doesn’t end well... . there is a gcc flag -mone-byte-bool to “make things compatible” if you need it.

C types - summary

- C has a fairly straightforward type system.
- No inheritance, etc..
    - Although some folks try really hard to fake it with unions, void *’s, etc.
- You should exploit the type system for all it is worth.
    - Strong typing reduces subtle programming errors.
    - Strong typing makes code more maintainable
    - Avoid void * unless you really have to use it.
- Use opaque types to enforce the use of module interfaces
- Some very difficult to find bugs lay in the darker shadows of the type system
    - type’s for union fields...
    - linking against C++ code...
    - structure alignment...
    - duplicate definitions of things...

“Standard” libraries


C code
assembly
object file
ld
preprocessed C code
ELF
Actual binary code inserted here
header file definitions inserted here

- For all practical purposes, all *user mode* programs use a library “libc” (pronounced “lib c”).
- The phrase “link against” means to include the library during the linking phase.
- But keep in mind libc != C it is just a blob of bits with functions and variables you can call
  - When you write a device driver or an OS kernel, you would not want to link against libc.
- libc is “almost standard” but there are variations in the header file layout and the darker corners of the semantics between systems and versions.
- On MSFT platforms the library is named MSVCRT for reasons only Redmond knows...

libc

- libc provides numerous useful functions for manipulating data in your own program. It also provides nice wrappers for communicating with the operating system (on POSIX/Unix and to a limited extent Windows). And it provides it’s own “higher level” interface to the operating system I/O.
- Most useful for you to know:
  - memory allocation (malloc/free)
  - string manipulation (strcpy, strcmp)
  - file I/O (fopen, fwrite, fread, fprintf, printf...
  - conversion (atoi, atof, ...)
  - direct POSIX system calls (open, write, read, ...)
  - time functions (gettimeofday, ctime, ...)
  - assert (not exactly part of libc, but part of the header files)

libc - continued

- To properly use these functions you sometimes have to call them in the correct order (e.g. fopen before fwrite) and **you must include the correct header file.**
    - If you do not, and you don't have -Wall on your compiler flags, your program will compile, but all your arguments will be passed as int's (which is the "C way")
- Which header file you ask?
    - It depends... you need to look it up.
        - libc has *evolved* over time and it varies from system to system.
        - It is largely "standard enough" that variations can be dealt with by crafting the program to a handful of variations
        - Ever wonder what ./configure (autoconf) was all about? It is partly in response to this...
    - On unix systems, do "man function" to find out. Try it: "man malloc"

libm, so close, yet so far...

- For historical reasons (back before sophisticated linking and back when processors had *optional* floating point units!), the math functions of “libc” are contained in a different library “libm”
- libm contains your basic math operators:
  - cos
  - sin
  - tan
  - etc...

- *As an aside*: a lot of OS kernels disallow floating point. They do this because floating point is expensive to context switch.

Global variables

- file1.c
  ```
  int x; // globally visible to the linker variable x
  ```
- file2.c
  ```
  extern int x;  // tell the C compiler that somewhere at the link stage a symbol
                 // of name x and type int will exist.
  ```

- Typically the “extern” lives in a header file. It is legit to have code like this:

  ```
  extern int x;  // somewhere in the future x will be resolved....
  int x = 1;   // and here in this module is where it will be.  Oh and initialize it to 1
  ```

- NOT ok:
  - extern int x = 1;

malloc & free

- In C you can dynamically allocate regions of memory. You do this by calling “malloc”
    - void *malloc(size_t size);
- Notes:
    - C has no idea a region of memory is a certain type of thing (int’s, strings, etc). It is up to you the developer to know that
    - Unlike Java, objects **must be explicitly freed** otherwise your program will have memory leaks.
        - free(void *)
        - While this appears painful, it isn’t so bad, and enforces good programmer discipline
        - Handling forgotten (and duplicate) free’s isn’t so hard. You:
            - Roll your own protected versions of malloc/free to look for leaks
            - Use electric fence or other tools like it
            - Use good discipline, specifically *pools* of memory and implement a policy where by all objects within a pool have to be free’d at the time pool is no longer in use.

malloc & free

- **Advice:** Make it clear who “owns” a chunk of memory in your code. That module should be responsible for the conceptual task of allocating it, and freeing it.
  - This is separate from the mechanical task of allocating it. For example, a module may provide an interface through an opaque type, and to do so would provide functions like: allocate_object() and free_object(object *). It is not the module that owns the object, it is the module that is calling allocate/free
  - At a minimum wrap your malloc / free calls like so:
    ```
    extern int allocs;
    #define malloc(x) (++allocs, malloc(x))
    #define free(x) (--allocs, free(x))
    ```
  - Then at the end of your program print out allocs. If != 0 you have a problem.

C has very little idea about multi-dimensional arrays

- int matrix[4][20];   // ok, but not dynamic...
- int *matrix[20]; // not what you think!
- int matrix[][20]; // NOT ok
- int matrix[20][]; // NOT ok
- int matrix[][]; // Definitely NOT ok
- int **matrix; // hmm, a good start...

- To allocate a dynamically sized array you need to think about it not as one large block of memory. Instead, think about it as an array of pointers, each of which points to a block of memory.

```
int **m, i, j;
m = malloc(sizeof(int *) * ROWS);
for (i = 0; i < ROWS; i++)
    m[i] = malloc(sizeof(int) * COLUMNS);
for (i = 0; i < ROWS; i++)
    for(j = 0; j < COLUMNS; j++)
         m[i][j] = 0;
```

- Of course, this makes free’ing it a bit more work.

C summary

- C is a language that is very close to the hardware
- Systems software is most often written in it because there’s nothing hidden
- The “c language” as most people think about it is actually made up of three parts: the preprocessor, the language it’s, and the “standard libraries”
- C has a standard but it is only loosely followed
    - Programs are often written to compile/execute on only one platform, or have explicit cross-platform support
- While the product of a C compiler is largely predictable, and hence easy for software developers to reason about, it is not always so and there are subtle issues
    - More on this near the very end of the class when we talk about threads...