Outline

• Previously:
  • Computer is a machine that does what we tell it to do

• Next:
  • How do we tell computers what to do?
    • First a quick intro to C programming
    • Goal: to learn C, not teach you to be an expert in C
  • How do we represent data?
  • What is memory?
What is C?

- The language of UNIX
- Procedural language (no classes)
- Low-level access to memory
- Easy to map to machine language
- Not much run-time stuff needed
- Surprisingly cross-platform

**Why teach it now?**
To expand from basic programming to operating systems and embedded development.

Also, as a case study to understand computer architecture in general.
Hey, do you want to build a system that will become the gold standard of OS design for this century? We can call it UNIX.

Okay, but only if we also invent a language to write it in, and only if that language becomes the default for all systems programming basically forever. We’ll call it C!

Ken Thompson
Dennis Ritchie

AT&T Bell Labs, 1969-1972
Cool, it worked!

Told ya.
What were they thinking?

- Main design considerations:
  - Compiler size: needed to run on PDP-11 with 24KB of memory (Algol60 was too big to fit)
  - Code size: needed to implement the whole OS and applications with little memory
  - Performance
  - Portability

- Little (if any consideration):
  - Security, robustness, maintainability
  - Legacy Code
# C vs. other languages

<table>
<thead>
<tr>
<th>Most modern languages</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develop applications</td>
<td>Develop system code (and applications) (the two used to be the same thing)</td>
</tr>
<tr>
<td>Computer is an abstract logic engine</td>
<td>Near-direct control of the hardware</td>
</tr>
<tr>
<td>Prevent unintended behavior, reduce impact of simple mistakes</td>
<td>Never doubts the programmer, subtle bugs can have crazy effects</td>
</tr>
<tr>
<td>Runs on magic! (e.g. garbage collection)</td>
<td>Nothing happens without developer intent</td>
</tr>
<tr>
<td>Smart, integrated toolchain (press button, receive EXE)</td>
<td>Compiles to native machine code</td>
</tr>
</tbody>
</table>

```
$ make
g++ -o thing.o thing.c
g++ -o thing thing.o
```
Why C?

• Why C for humanity?
  • It’s a “portable assembly language”
  • Useful in OS and embedded systems and for highly optimized code

• Why C for this class?
  • Need to understand how computers work
  • Need a high-level language that can be traced all the way down to machine code
  • Need a language with system-level concepts like pointers and memory management
  • Java hides too much to do this
Task: Export a list of coordinates in memory to disk

Most languages

• Develop file format
• Build routine to serialize data out to disk
• Build routine to read & parse data in
• Benchmark if performance is a concern

C

• Read/write memory to disk directly
## Example C superpowers

### Task: Blink an LED

Atmel ATTINY4 microcontroller:
Entire computer (CPU, RAM, & storage)!
1024 bytes storage, 32 bytes RAM.

<table>
<thead>
<tr>
<th>Language</th>
<th>Size of executable</th>
<th>Size of runtime (ignoring libraries)</th>
<th>Total size</th>
<th>RAM used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>410 B</td>
<td>13 MB</td>
<td>13 MB</td>
<td>14 MB</td>
</tr>
<tr>
<td>Python</td>
<td>60 B (source code)</td>
<td>2.9 MB</td>
<td>2.9 MB</td>
<td>5.4 MB</td>
</tr>
<tr>
<td>Desktop C</td>
<td>8376 B</td>
<td>None</td>
<td>8376 B</td>
<td>352 kB</td>
</tr>
<tr>
<td>Embedded C (Arduino)</td>
<td>838 B</td>
<td>None</td>
<td>838 B</td>
<td>~16 B</td>
</tr>
</tbody>
</table>

```python
led = 0
while (true):
    led = NOT led
set_led(led)
delay for 1 sec
```

Max: 1024 B
Max: 32 B
What about C++?

- Originally called “C with Classes” (because that’s all it is)
- All C programs are C++ programs, as C++ is an extension to C
- Adds stuff you might recognize from Java (only uglier):
  - Classes (incl. abstract classes & virtual functions)
  - Operator overloading
  - Inheritance (incl. multiple inheritance)
  - Exceptions

Bjarne Stroustrup developed C++ in 1979 at Bell Labs

OUT OF SCOPE
C and Java: A comparison

### C

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

int main(int argc, const char* argv[]) {
    int i;

    printf("Hello, world.\n");

    for (i=0; i<3; i++) {
        printf("%d\n", i);
    }

    return EXIT_SUCCESS;
}
```

```bash
$ g++ -o thing thing.c && ./thing
Hello, world.
0
1
2
```

### Java

```java
class Thing {
    static public void main (String[] args) {
        int i;

        System.out.printf("Hello, world.\n");

        for (i=0; i<3; i++) {
            System.out.printf("%d\n", i);
        }
    }
}
```

```bash
$ javac Thing.java && java Thing
Hello, world.
0
1
2
```
Common Platform for This Course

- Different platforms have different conventions for end of line, end of file, tabs, compiler output, ...
- Solution (for this class): compile and run all programs consistently on one platform
- Our common platform:

Duke Linux Machines!

Don’t you gimme no “it worked on my box” nonsense!
See homework 0 or recitation #1 for the exciting answer!
High Level Language Program

Compiler

Assembly Language Program

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)

• Every computer architecture has its own **assembly language**

• Assembly languages tend to be pretty low-level, yet some actual humans still write code in assembly

• But most code is written in HLLs and **compiled**
  • Compiler is a program that automatically converts HLL to assembly
• **Assembler** program automatically converts assembly code into the binary **machine language** (zeros and ones) that the computer actually executes.
Machine Language → Inputs to Digital System

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signals for Finite State Machine

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

lw $15, 0($2)

lw $16, 4($2)

sw $16, 0($2)

sw $15, 4($2)

Transistors (switches) turning on and off
How does a Java program execute?

- Compile Java Source to Java Byte codes
- Java Virtual Machine (JVM) interprets/translations Byte codes
- JVM is a program executing on the hardware

- Java has lots of features that make it easier to program without making mistakes → training wheels are nice

- JVM handles memory for you
  - What do you do when you remove an entry from a hash table, binary tree, etc.?
The C Programming Language

- No virtual machine
  - No dynamic type checking, array bounds, garbage collection, etc.
  - Compile source file directly to machine

- Closer to hardware
  - Easier to make mistakes
  - Can often result in faster code → training wheels slow you down

- Generally used for ‘systems programming’
  - Operating systems, embedded systems, database implementation
  - C++ is object-oriented version of C (C is a strict subset of C++)
Creating a C source file

- We are not using a development environment (IDE)
- You will create programs starting with an empty file!
- Files should use .c file extension (e.g., hello.c)
- On a Linux machine, edit files with nedit (or emacs or ...)
The nedit window

- nedit is a simple point & click editor
  - with ctrl-c, ctrl-x, ctrl-v, etc. short cuts
- Feel free to use any text editor (gvim, emacs, etc.)
Hello World

- Canonical beginner program
  - Prints out “Hello …”
- nedit provides syntax highlighting

```c
#include <stdio.h>

int main()
{
    printf("Hello CompSci250!\n");
}
```
Compiling and Running the Program

- Use the g++ (or gcc) compiler to turn .c file into executable file
  - g++ -g -o <outputname> <source file name>
  - g++ -g -o hello hello.c (you must be in same directory as hello.c)
  - If no -o option, then default output name is a.out (e.g., g++ hello.c)
  - The -g option turns on debug info, so tools can tell you what’s up when it breaks

- To run, type the program name on the command line
  - ./ before “hello” means look in current directory for hello program
Key Language Issues (for C)

- Variable types: int, float, char, etc.
- Operators: +, -, *, ==, >, etc.
- Expressions
- Control flow: if/else, while, for, etc.
- Functions
- Arrays
  - Java: Strings → C: character arrays
  - Java: Objects → C: structures
  - Java: References → C: pointers
  - Java: Automatic memory mgmt → C: DIY mem mgmt

Black: C same as Java
Blue: C very similar to Java
Red: C different from Java
Variables, operators, expressions – just like Java

• Variables types
  • Data types: int, float, double, char, void
  • signed and unsigned int
  • char, short, int, long, long long can all be integer types
    • These specify how many bits to represent an integer

• Operators
  • Mathematical: + − * / %
  • Logical: ! && || == != < > <= >=
  • Bitwise: & | ~ ^ << >>
    (we’ll get to what these do later)

• Expressions: var1 = var2 + var3;
C Allows Type Conversion with Casts

- Use type casting to convert between types
  - variable1 = (new type) variable2;
- Be careful with order of operations – cast often takes precedence
- Example

```c
main() {
    float x;
    int i;
    x = 3.6;
    i = (int) x;  // i is the integer cast of x
    printf("x=%f, i=%d", x, i)
}
```

result: x=3.600000, i=3

SAME as Java!
Control Flow – just like Java

• **Conditionals**

```java
if (a < b) { ... } else {...}
switch (a) {
    case 0: s0; break;
    case 1: s1; break;
    case 2: s2; break;
    default: break;
}
```

• **Loops**

```java
for (i = 0; i < max; i++) { ... }  
while (i < max) {...}
```
Variable Scope: Global Variables

- Global variables are accessible from any function
  - Declared outside `main()`

```c
#include <stdio.h>
int X = 0;
float Y = 0.0;
void setX() { X = 78; }
int main()
{
    X = 23;
    Y = 0.31234;
    setX();
    // what is the value of X here?
}
```

- What if we had “`int X = 23;`” in `main()`?
Functions – mostly like Java

• C has functions, just like Java
  • But these are not methods! (not attached to objects)
• Must be defined *or at least declared* before use

```c
int div2(int x, int y); /* declaration here */
int main() {
    int a;
    a = div2(10, 2);
}
int div2(int x, int y) { /* implementation here */
    return (x/y);
}
```

• *Or you can just put functions at top of file (before use)*
Arrays – same as Java

Same as Java (for now...)

char buf[256];
int grid[256][512]; /* two dimensional array */
float scores[4096];
double speed[100];

for (i = 0; i< 25; i++)
    buf[i] = 'A'+i;        /* what does this do? */
Memory Layout and Bounds Checking

- There is **NO bounds checking** in C
  - i.e., it’s legal (but not advisable) to refer to
    - `days_in_month[216]` or
    - `days_in_month[-35]`
  - who knows what is stored there?

Storage for array `int days_in_month[12];`

Storage for other stuff

Storage for some more stuff

(each location shown here is an `int`)
Strings – not quite like Java

• Strings
  • char str1[256] = “hi”;
  • 0 is value of NULL character ‘\0’, identifies end of string

• What is C code to compute string length?
  ```
  int len=0;
  while (str1[len] != 0){
      len++;
  }
  ```

• Length does not include the NULL character
• C has built-in string operations
  • #include <string.h>  // includes string operations
  • strlen(str1);
Structures

- Structures are sort of like Java objects
  - They have member variables
  - But they do NOT have methods!

- Structure definition with `struct` keyword
  ```c
  struct student_record {
    int id;
    float grade;
  } rec1, rec2;
  ```

- Declare a variable of the structure type with `struct` keyword
  ```c
  struct student_record onerec;
  ```

- Access the structure member fields with dot (`.`), e.g. `structvar.member`
  ```c
  onerec.id = 12;
  onerec.grade = 79.3;
  ```
#include <stdio.h>

struct student_record {
    int id;
    float grade;
};

struct student_record myroster[100]; /* declare array of structs */

int main()
{
    myroster[23].id = 99;
    myroster[23].grade = 88.5;
}
Console I/O in C

- I/O is provided by standard library functions
  - available on all platforms
- To use, your program must have
  ```
  #include <stdio.h>
  ```
- ...and it doesn’t hurt to also have
  ```
  #include <stdlib.h>
  ```
- These are preprocessor statements; the .h files define function types, parameters, and constants from the standard library

"Standard IO"
Not "studio"

"Standard library"
Back to our first program

- `#include <stdio.h>` defines input/output functions in C standard library (just like you have libraries in Java)
- `printf(args)` writes to terminal
Input/Output (I/O)

• Read/Write to/from the terminal
  • Standard input, standard output (defaults are terminal)

• Character I/O
  • `putchar()`, `getchar()`

• Formatted I/O
  • `printf()`, `scanf()`
Character I/O

```c
#include <stdio.h>  /* include the standard I/O function defs */
int main() {
    char c;
    /* read chars until end of file */
    while ((c = getchar()) != EOF ) {
        if (c == 'e')
            c = '-';
        putchar(c);
    }
    return 0;
}

• EOF is End Of File (type Ctrl+D)
```
#include <stdio.h>
int main() {
    int a = 23;
    float f = 0.31234;
    char str1[] = "satisfied?";
    /* some code here… */
    printf("The variable values are %d, %f, %s\n", a, f, str1);
    scanf("%d %f", &a, &f); /* we’ll come back to the & later */
    scanf("%s", str1);
    printf("The variable values are now %d, %f, %s\n", a, f, str1);
}

- printf("format string", v1,v2,...);
  - \n  is newline character

- scanf("format string",...);
  - Returns number of matching items or EOF if at end-of-file
#include <stdio.h>
int main()
{
    int x = 0;
    while (scanf("%d", &x) != EOF) {
        printf("The value is %d\n", x);
    }
}

• This reads integers from the terminal until the user types ^d (ctrl-d)
  • Can use a.out < file.in

• WARNING THIS IS NOT CLEAN CODE!!!
  • If the user makes a typo and enters a non-integer it can loop indefinitely!!!

• How to stop a program that is in an infinite loop on Linux?
• Type ^c (ctrl-c). It kills the currently executing program.
Example: Reading Input in a Loop (better)

```c
#include <stdio.h>
int main()
{
    int x = 0;
    while(scanf("%d", &x) == 1) {
        printf("The value is %d\n", x);
    }
}
```

- Now it reads integers from the terminal until there’s an EOF or a non-integer is given.
- Type “man scanf” on a linux machine and you can read a lot about scanf.
  - You can also find these “manual pages” on the web, such as at die.net.
sscanf vs. atoi

- You can parse in-memory strings with `sscanf` (string scanf):

  ```c
  char mystring[] = "29";
  int r;
  int n = sscanf(mystring,"%d",&r);
  // returns number of successful conversions (0 or 1)
  ```

- You *could* use the `atoi` function to convert a string to an integer, but then you can’t detect errors.

  ```c
  char mystring[] = "29";
  int r = atoi(mystring);
  ```

  - The `atoi` function just returns 0 for non-integers, so `atoi("0") == atoi("hurfurf")` 😞

  `atoi` stands for a-to-i, as in array-to-integer, because strings are character arrays.
Header Files, Separate Compilation, Libraries

- C pre-processor provides useful features
  - `#include` filename just inserts that file (like `#include <stdio.h>`)
  - `#define MYFOO 8`, replaces MYFOO with 8 in entire program
    - Good for constants
    - `#define MAX_STUDENTS 100` (functionally equivalent to `const int`)

- Separate Compilation
  - Many source files (e.g., `main.c`, `students.c`, `instructors.c`, `deans.c`)
  - `g++ -o prog main.c students.c instructors.c deans.c`
  - Produces one executable program from multiple source files

- Libraries: Collection of common functions (some provided, you can build your own)
  - We’ve already seen `stdio.h` for I/O
  - `libc` has I/O, strings, etc.
  - `libm` has math functions (`pow`, `exp`, etc.)
  - `g++ -o prog file.c -lm` (says use math library)
Command Line Arguments

- Parameters to main (`int argc, char *argv[]`)
  - `argc` = number of arguments (0 to argc-1)
  - `argv` is array of strings
  - `argv[0]` = program name
- Example: `myProgram dan 250`
  - `argc=3`

```c
int main(int argc, char *argv[]) {
    int i;
    printf("%d arguments\n", argc);
    for (i=0; i< argc; i++) {
        printf("argument %d: %s\n", i, argv[i]);
    }
}
```
1) Java is object-oriented, while C is not

2) Memory management
   - Java: the virtual machine worries about where the variables “live” and how to allocate memory for them
   - C: the programmer does all of this
Memory is a real thing!

- Most languages – protected variables
- C – flat memory space

Figure from Rudra Dutta, NCSU, 2007
Let’s look at memory addresses!

- You can find the address of ANY variable with:

  `&`

  The address-of operator

```c
int v = 5;
printf("%d\n", v);
printf("%p\n", &v);
```

```
$ g++ x4.c && ./a.out
5
0xffffd32228c
```
Testing our memory map

```c
int x=5;
char msg[] = "Hello";

int main(int argc, const char* argv[]) {
    int v;
    float pi = 3.14159;

    printf("&x:   %p\n", &x);
    printf("&msg: %p\n", &msg);
    printf("&argc: %p\n", &argc);
    printf("&argv: %p\n", &argv);
    printf("&v:    %p\n", &v);
    printf("&pi:   %p\n", &pi);
}
```

```
$ g++ x.c && ./a.out
&x: 0x601020
&msg: 0x601024
&argc: 0x7fff85b78c2c
&argv: 0x7fff85b78c20
&v: 0x7fff85b78c38
&pi: 0x7fff85b78c3c
```
What’s a pointer?

- It’s a memory address you treat as a variable
- You declare pointers with:

```c
int v = 5;
int* p = &v;
printf("%d\n", v);
printf("%p\n", p);
```

The dereference operator

```
$ g++ x4.c && ./a.out
5
0x7fffe0e60b7c
```
What’s a pointer?

- You can **look up** what’s stored *at* a pointer!
- You **dereference** pointers with:

```c
int v = 5;
int* p = &v;
printf("%d\n", v);
printf("%p\n", p);
printf("%d\n", *p);
```

Prepend to any pointer variable or expression

```bash
$ g++ x4.c && ./a.out
5
0x7fffe0e60b7c
5
```
What is an array?

- The shocking truth: You’ve been using pointers all along!
- Every array **IS** a pointer to a block of memory
- **Pointer arithmetic:** If you add an integer N to a pointer P, you get the address of N **things** later from pointer P
  - “Thing” depends on the datatype of the P
- Can **dereference** such pointers to get what’s there
  - Interpreted according to the datatype of P
  - E.g. *(nums-1) is a number related to how we represent the letter ‘o’.

```
int x = 9;
char msg[] = “hello”;
short nums[] = {6,7,8};
```
Array lookups ARE pointer references!

int x[] = {15,16,17,18,19,20};

<table>
<thead>
<tr>
<th>Array lookup</th>
<th>Pointer reference</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>int*</td>
</tr>
<tr>
<td>x[0]</td>
<td>*x</td>
<td>int</td>
</tr>
<tr>
<td>x[5]</td>
<td>*(x+5)</td>
<td>int</td>
</tr>
<tr>
<td>x[n]</td>
<td>*(x+n)</td>
<td>int</td>
</tr>
<tr>
<td>&amp;x[0]</td>
<td>x</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;x[5]</td>
<td>x+5</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;x[n]</td>
<td>x+n</td>
<td>int*</td>
</tr>
</tbody>
</table>

(In case you don’t believe me)

```c
int n=2;
printf("%p %p", x   , x   );
printf("%d %d", x[0] , *x   );
printf("%d %d", x[5] ,*(x+5));
printf("%d %d", x[n] ,*(x+n));
printf("%p %p",&x[0],   x   );
printf("%p %p",&x[5],   x+5 );
printf("%p %p",&x[n],   x+n );
```

$ g++ x5.c && ./a.out
0x7fffa2d0b9d0 0x7fffa2d0b9d0
15 15
20 20
17 17
0x7fffa2d0b9d0 0x7fffa2d0b9d0
0x7fffa2d0b9e4 0x7fffa2d0b9e4
0x7fffa2d0b9d8 0x7fffa2d0b9d8

• This is why arrays don’t know their own length: they’re just blocks of memory with a pointer!

Definition of array brackets: **A[i] ⇔ *(A+i)**

Creepy-side effect: A[5] ⇒ *(A+5) ⇒ *(5+A) ⇒ 5[A], so 5[A] is legal & equivalent! (Don’t do this, it’s gross.)
Using pointers

- Start with an address of something that exists
- Manipulate according to known rules
- Don’t go out of bounds (don’t screw up)

```c
void underscorify(char* s) {
    char* p = s;
    while (*p != 0) {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}
```

```c
int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify(msg);
    puts(msg);
}
```

```
$ g++ x3.c && ./a.out
Here are words
Here_are_words
```
void underscorify(char* s) {
    char* p = s;
    while (*p != 0) {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}

// how a developer might code it
void underscorify2(char* s) {
    char* p;
    for (p = s; *p ; p++) {
        if (*p == ' ') {
            *p = '_';
        }
    }
}

// how a kernel hacker might code it
void underscorify3(char* s) {
    for ( ; *s ; s++) {
        if (*s == ' ') *s = '_';
    }
}
Pointers: powerful, but deadly

• What happens if we run this?

```c
#include <stdio.h>

int main(int argc, const char* argv[]) {
    int* p;

    printf(" p: %p\n", p);
    printf("*p: %d\n", *p);
}
```

$ g++ x2.c && ./a.out
p: (nil)
Segmentation fault (core dumped)
Pointers: powerful, but deadly

- Okay, I can fix this! I’ll initialize \( p \)!

```c
#include <stdio.h>

int main(int argc, const char* argv[]) {
    int* p = 100000;

    printf(" p:  %p\n", p);
    printf("*p:  %d\n", *p);
}
```

```
$ g++ x2.c
x2.c: In function ‘main’:  
x2.c:4:9: warning: initialization makes pointer from integer without a cast [enabled by default]
$ ./a.out
  p:  0x186a0
Segmentation fault (core dumped)
```
A more likely pointer bug...

```c
void underscorify_bad(char* s) {
    char* p = s;
    while (*p != '0') {
        if (*p == 0) {
            *p = '_';
        }
        p++;
    }
}

int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify_bad(msg);
    puts(msg);
}
```
void underscorify_bad2(char* s) {
    char* p = s;
    while (*p != '0') {
        if (*p == ' ') {
            *p = '_';
        }
        p++;
    }
}

int main() {
    char msg[] = "Here are words";
    puts(msg);
    underscorify_bad2(msg);
    puts(msg);
}

Worked but crashed on exit
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Worked totally!!
Effects of pointer mistakes

- Access an array out of bounds or some other invalid pointer location?
  - No visible effect
  - Totally weird behavior
  - Silent corruption & bad results
  - Program crash with OS error
• Memory is linear, all the variables live at an address
  • Variable declarations reserve a range of memory space
• You can get the address of any variable with the **address-of operator** &
  
  int x;   printf("%p\n", &x);
• You can **declare a pointer** with the **dereference operator** * appended to a type:
  
  int* p = &x;
• You can find the data at a memory address with the **dereference operator** * prepended to a pointer expression:
  
  printf("%d\n", *p);
• Arrays in C are just pointers to a chunk of memory
• Don’t screw up
Pass by Value vs. Pass by Reference

```c
void swap (int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 3;
    int b = 4;
    swap(a, b);
    printf("a = %d, b= %d\n", a, b);
}
```

```c
void swap (int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 3;
    int b = 4;
    swap(&a, &b);
    printf("a = %d, b= %d\n", a, b);
}
```
C Memory Allocation

• How do you allocate an object in Java?
• What do you do when you are finished with object?

• JVM provides garbage collection
  • Counts references to objects, when refs== 0 can reuse

• C does not have garbage collection
  • Must explicitly manage memory
C Memory Allocation

- **void* malloc(nbytes)**
  - Obtain storage for your data (like `new` in Java)
  - Often use `sizeof(type)` built-in returns bytes needed for `type`
  - `int* my_ptr = (int*) malloc(64);` // 64 bytes = 16 ints
  - `int* my_ptr = (int*) malloc(64*sizeof(int));` // 64 ints

- **free(ptr)**
  - Return the storage when you are finished (no Java equivalent)
  - `ptr` must be a value previously returned from `malloc`
C Memory Allocation

- **void** `calloc(num, sz)`
  - Like `malloc`, but reserves `num*sz` bytes, and initializes the memory to zeroes

- **void** `realloc(ptr, sz)`
  - Grows or shrinks allocated memory
    - `ptr` must be dynamically allocated
    - Growing memory doesn’t initialize new bytes
    - Memory shrinks in place
    - Memory may NOT grow in place
      - If not enough space, will move to new location and copy contents
      - Old memory is freed
      - Update all pointers!!!
  - **Usage:** `ptr = realloc(ptr, new_size);`
Memory management examples

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

int main() {
    // kind of silly, but let's malloc a single int
    int* one_integer = (int*) malloc(sizeof(int));
    *one_integer = 5;

    // allocating 10 integers worth of space.
    int* many_integers = (int*) malloc(10 * sizeof(int));
    many_integers[2] = 99;

    // using calloc over malloc will pre-initialize all values to 0
    float* many_floats = (float*) calloc(10, sizeof(float));
    many_floats[4] = 1.21;

    // double the allocation of this array
    many_floats = (float*) realloc(many_floats, 20*sizeof(float));
    many_floats[15] = 6.626070040e-34;

    free(one_integer);
    free(many_integers);
    free(many_floats);
}
```
Pointers to Structs

```c
struct student_rec {
    int id;
    float grade;
};

struct student_rec* my_ptr = malloc(sizeof(struct student_rec));
// ptr to a student_rec struct

To access members of this struct via the pointer:

    (*my_ptr).id = 3;  // not my_ptr.id
    my_ptr->id = 3;    // not my_ptr.id
    my_ptr->grade = 2.3;  // not my_ptr.grade
```
Example: Linked List

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

struct entry {
    int id;
    struct entry* next;
};

int main() {
    struct entry *head, *ptr;
    head=(struct entry*)malloc(sizeof(struct entry));
    head->id = 66;
    //head->next = NULL;

    ptr = (struct entry*)malloc(sizeof(struct entry));
    ptr->id = 23;
    ptr->next = NULL;

    head->next = ptr;

    printf("head id: %d, next id: %d\n", head->id, head->next->id);

    ptr = head;
    head = ptr->next;

    printf("head id: %d, next id: %d\n", head->id, ptr->id);

    free(head);
    free(ptr);
}
```
Source Level Debugging

- Symbolic debugging lets you single step through program, and modify/examine variables while program executes
- On the Linux platform: **gdb**
- Source-level debuggers built into most IDEs
Gdb

- To start:
  $ gdb ./myprog

- To run:
  (gdb) run arguments
## gdb commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list &lt;line&gt;</code></td>
<td>list (show) 10 lines of code at specified location in program</td>
</tr>
<tr>
<td><code>list &lt;function&gt;</code></td>
<td>List from first line to last line</td>
</tr>
<tr>
<td><code>list &lt;line&gt;,&lt;line&gt;</code></td>
<td>continue execution</td>
</tr>
<tr>
<td><code>run</code></td>
<td>start running the program</td>
</tr>
<tr>
<td><code>continue</code></td>
<td>continue execution</td>
</tr>
<tr>
<td><code>step</code></td>
<td>single step execution, including into functions that are called</td>
</tr>
<tr>
<td><code>next</code></td>
<td>single step over function calls</td>
</tr>
<tr>
<td><code>print &lt;var&gt;</code></td>
<td>show variable value</td>
</tr>
<tr>
<td><code>printf &quot;fmt&quot;, &lt;var&gt;</code></td>
<td>show variable each time execution stops</td>
</tr>
<tr>
<td><code>display &lt;var&gt;</code></td>
<td>show variable each time execution stops</td>
</tr>
<tr>
<td><code>undisplay &lt;var&gt;</code></td>
<td>show variable each time execution stops</td>
</tr>
</tbody>
</table>
# gdb commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>break &lt;line&gt;</code></td>
<td>set breakpoints (including conditional breakpoints)</td>
</tr>
<tr>
<td><code>break &lt;function&gt;</code></td>
<td>list, and delete, breakpoints</td>
</tr>
<tr>
<td><code>break &lt;line&gt; if &lt;cond&gt;</code></td>
<td>set variable to a value</td>
</tr>
<tr>
<td><code>info breakpoints</code></td>
<td>show the call stack &amp; args arguments and local variables</td>
</tr>
<tr>
<td><code>delete breakpoint &lt;n&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>set &lt;var&gt; &lt;expr&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>bt</code></td>
<td></td>
</tr>
</tbody>
</table>
gdb quick reference card

- GDB Quick Reference.pdf – print it!
- Also available annotated by me with most important commands for a beginner:
  GDB Quick Reference - annotated.pdf
Valgrind: detect memory errors

• Can run apps with a **process monitor** to *try to* detect illegal memory activity and memory leaks
C Resources

- MIT Open Course

- Courseware from Dr. Bletsch’s NCSU course on C (linked from course page)

- Video snippets by Prof. Drew Hilton (Duke ECE/CS)
  - Doesn’t work with Firefox (use Safari or Chrome)
Outline

• Previously:
  • Computer is machine that does what we tell it to do

• Next:
  • How do we tell computers what to do?
    • First a quick intro to C programming
  • How do we represent data?
  • What is memory, and what are these so-called addresses?