ECE/CS 250
Computer Architecture

Fall 2023

C Programming

John Board
Duke University

Slides are derived from work by
Daniel J. Sorin (Duke), Andrew Hilton (Duke), Alvy Lebeck (Duke), Benjamin Lee (Duke), Tyler Bletsch (Duke), John Board (Duke), and Amir Roth (Penn)

Also contains material adapted from CSC230: C and Software Tools developed by the NC State Computer Science Faculty

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?

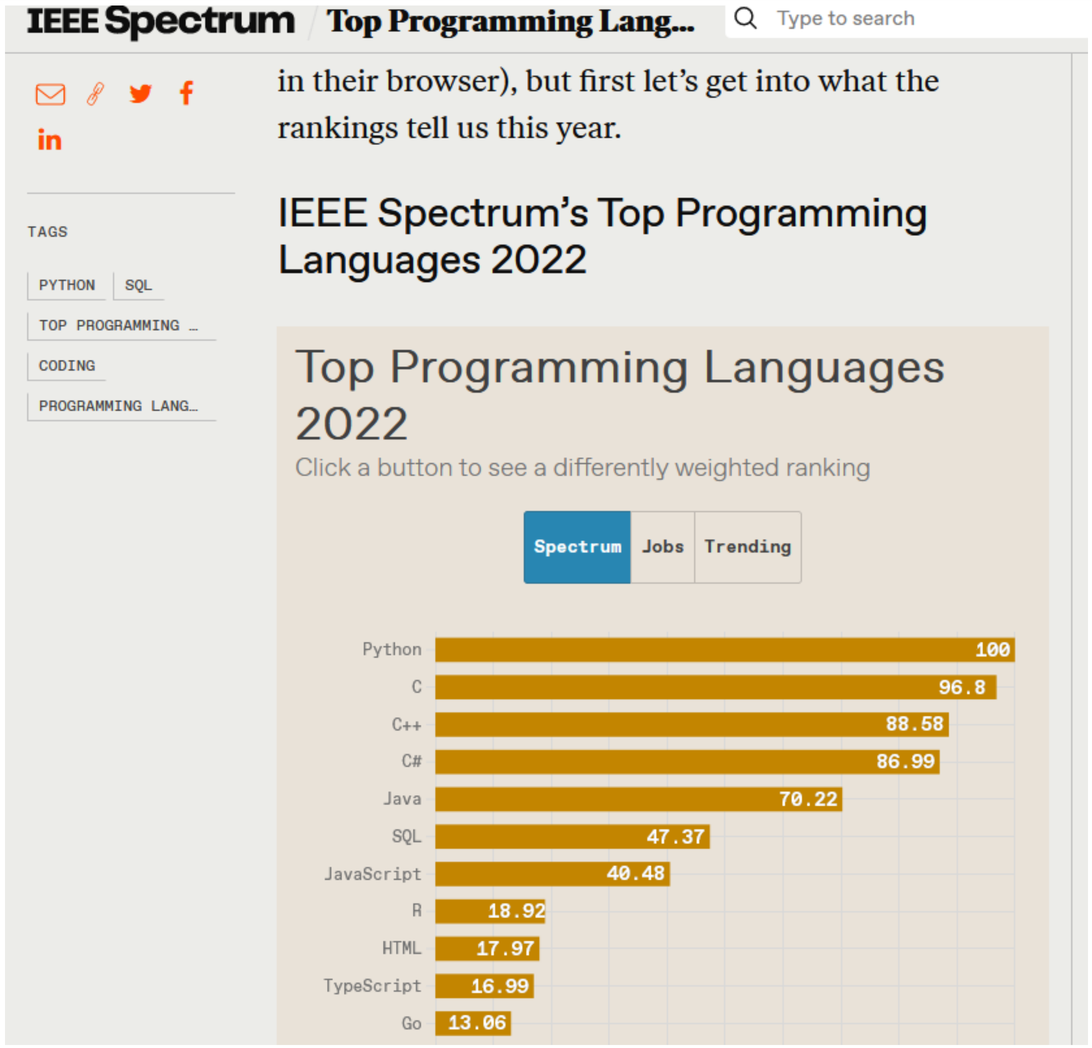
- The language of UNIX – designed to be a language operating systems could be written in, with direct access to low-level details of memory systems
- Procedural language (no classes)
- Low-level access to memory
- Easy to map to machine language
- Not much run-time stuff needed
- Surprisingly cross-platform (runs well on many different architectures/microarchitectures)

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.

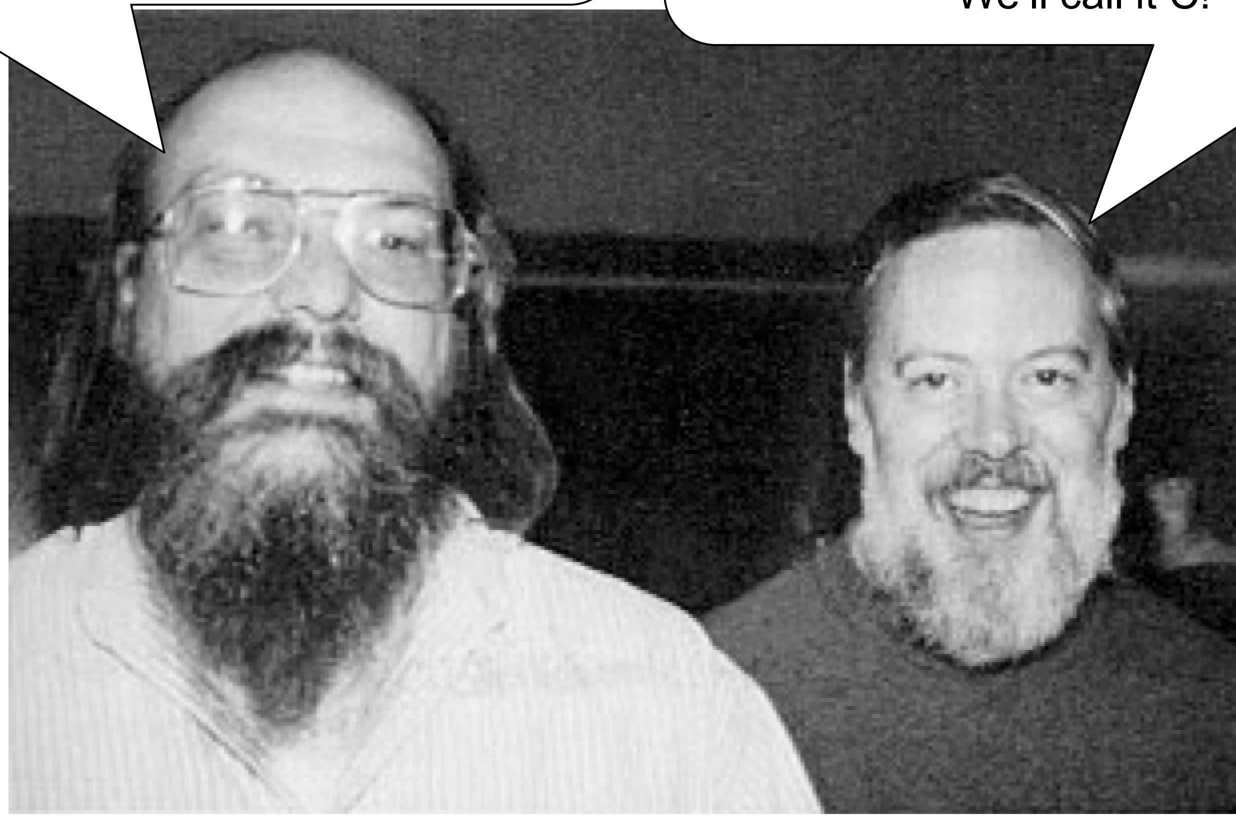
C and its offspring are still really important



The Origin of C

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

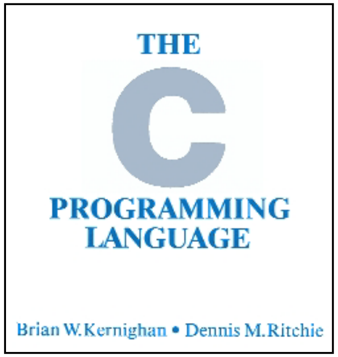
AT&T Bell Labs, 1969-1972


Told ya.
Cool, it worked!

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



Most modern languages	C
Develop applications	Develop system code (and applications) (the two used to be the same thing)
Computer is an abstract logic engine	Near-direct control of the hardware
Prevent unintended behavior, reduce impact of simple mistakes	Never doubts the programmer, subtle bugs can have crazy effects
Runs on magic! (e.g. garbage collection)	Nothing happens without developer intent
May run via VM or interpreter	Compiles to native machine code
Smart, integrated toolchain (press button, receive EXE)	Discrete, UNIX-style toolchain make → g++ (compilation) → g++ (linking) (even more discrete steps behind this)


```
$ 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 HPC (High Performance Computing), 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

Example C superpowers

Task: Blink an LED

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

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

Language	Size of executable	Size of runtime (ignoring libraries)	Total size	RAM used
Java				
Python				
Desktop C				
Embedded C (Arduino)				

Max: 1024 B

Max: 32 B

What about C++?

Bjarne Stroustrup developed C++ in 1979 at Bell Labs

- 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

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;
}
```

```
$ 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);
    }

  }
}
```

```
$ javac Thing.java && java Thing
Hello, world.
0
1
2
```

Common Platform for This Course

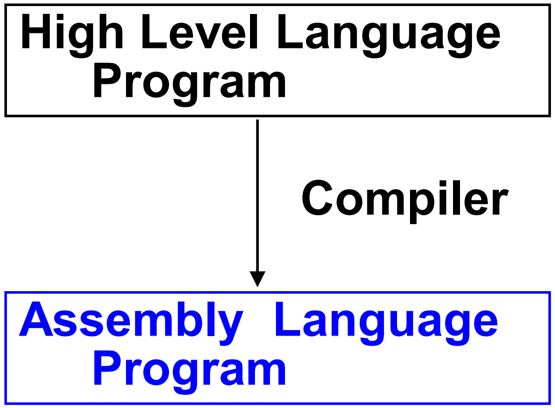
- Different platforms have different conventions for end of line, end of file, tabs, compiler output, …
- Solution (for this class): compile and run all programs consistently on one platform
- Our common platform:

Duke Linux Machines!

How to access Duke Linux machines?

See homework 0 or recitation #1 for the exciting answer!

HLL → Assembly Language




```
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 (250 and 350 students among them!)
- But most code is written in HLLs and compiled
  - Compiler is a program that automatically converts HLL to assembly

Assembly Language → Machine Language

High Level Language Program

↓ Compiler

Assembly Language Program

↓ Assembler

Machine 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)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

- Assembler program automatically converts assembly code into the binary machine language (zeros and ones) that the computer actually executes
- Easy 1:1 mapping to/from assembly statements and binary

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)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Transistors (switches) turning on and off

How does a Java program execute?

- Compile Java Source to Java Byte codes
- Java Virtual Machine (JVM) interprets/translates 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

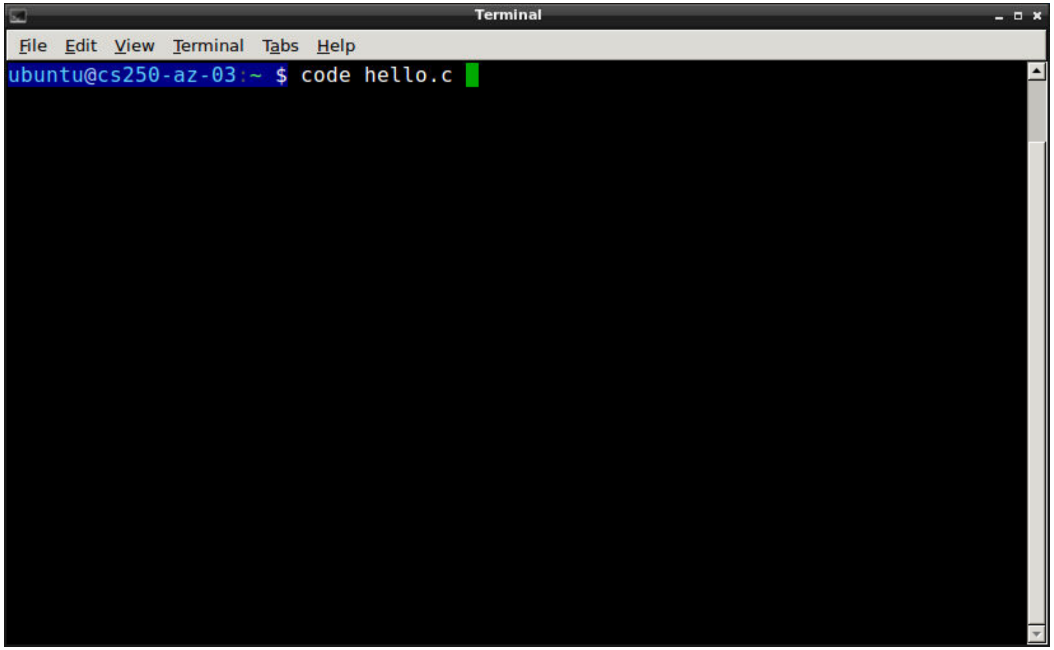
- No virtual machine
  - No dynamic type checking, array bounds, garbage collection, etc.
  - Compile source file directly to machine code

- Closer to hardware
  - Easier to make mistakes – it will let you (encourage you?!?) to shoot yourself in the foot
    - Because in writing an operating system rather than a general application you often do things that would be really strange in a "normal" program, especially with memory!
  - 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++)

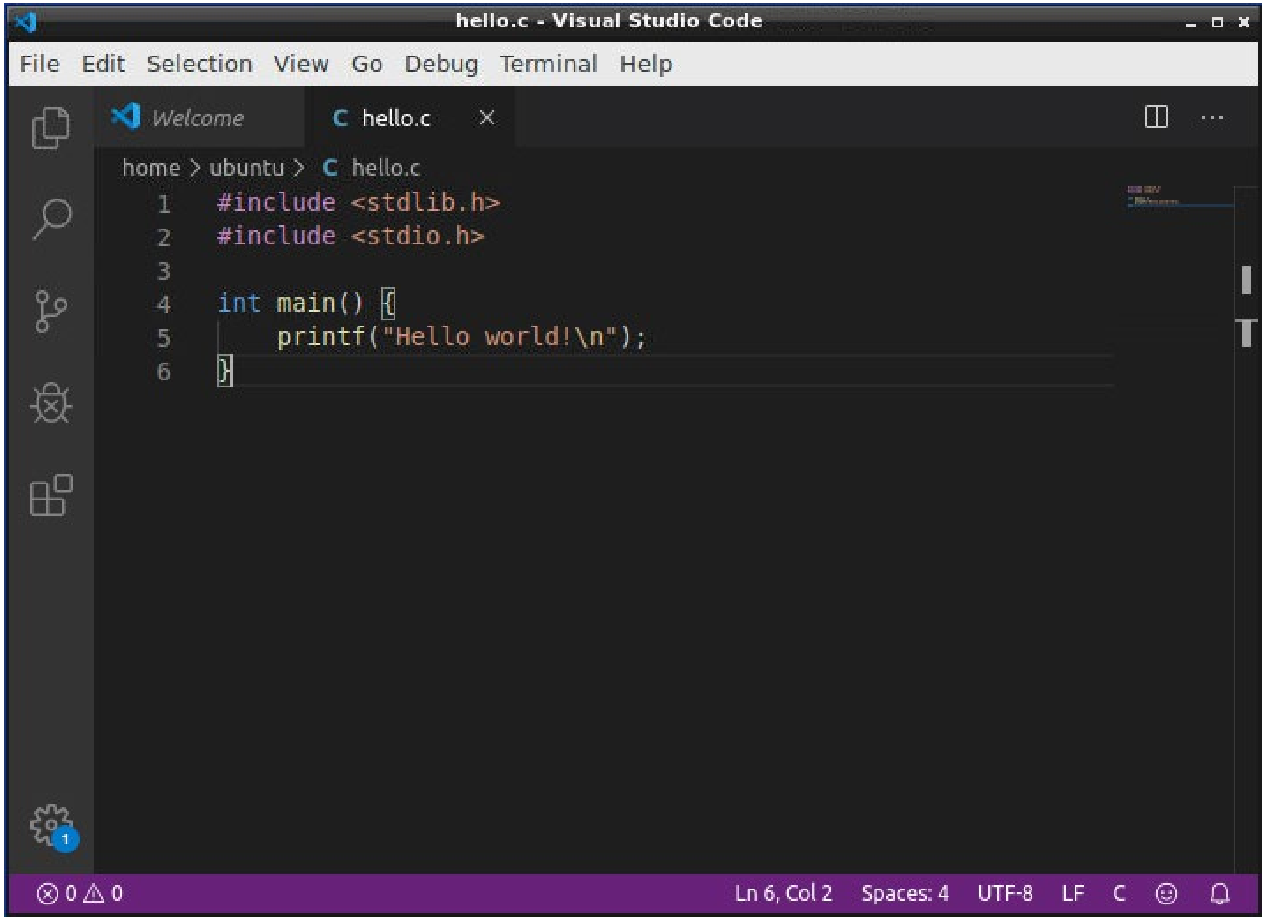
Creating a C source file

- 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 your chosen editor, e.g. Visual Studio Code (executable from command line as **`code <file>`**)



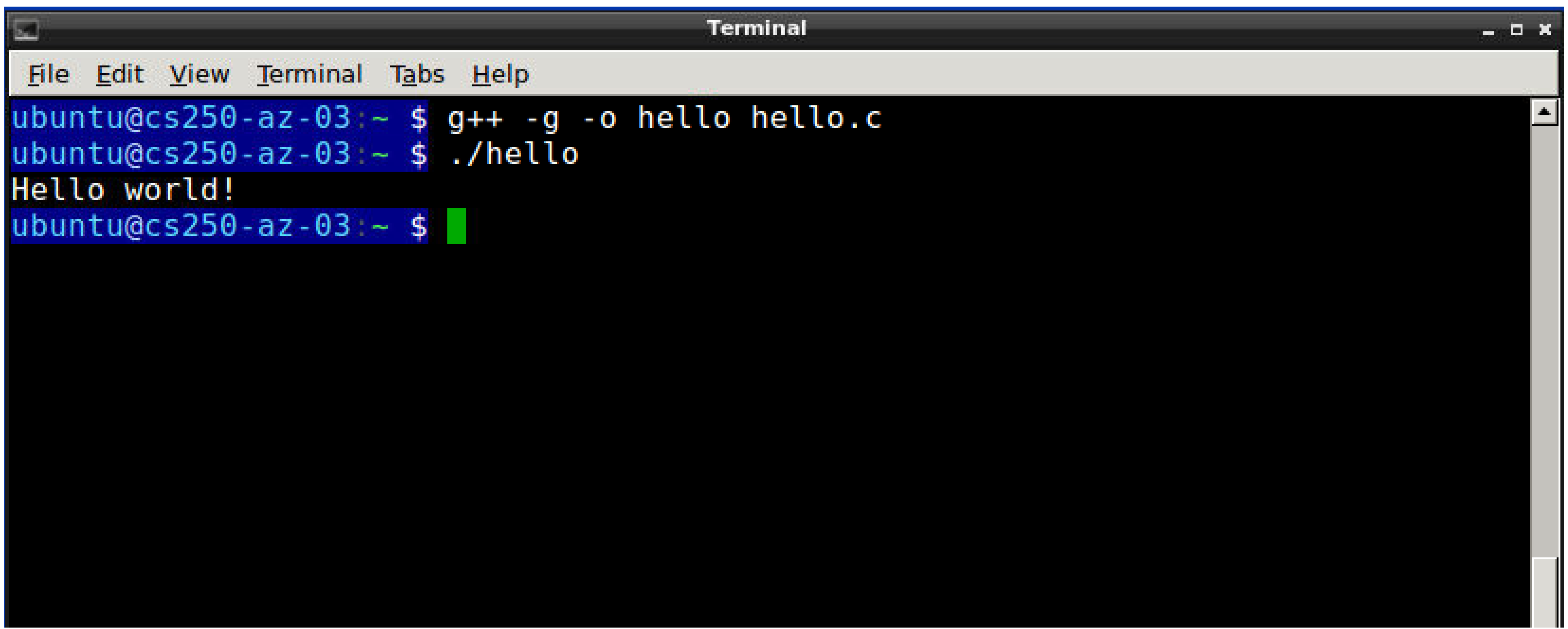
The vscode window

- Visual Studio Code is a fancy editor, but we'll use it like a simple editor
- Feel free to use any text editor (vim, emacs, etc.)



Compiling and Running the Program

- Use the g++ compiler to turn .c file into executable file
  - g++ -g   -o <outputfile>   <sourcefile>
  - 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



Breaking the third wall: gcc -S

```
[login01.egr.duke.edu]44: cat hello.c
#include <stdlib.h>
#include <stdio.h>

int main(){
printf("Hello World!\n");
}

[login01.egr.duke.edu]45: gcc -S hello.c
[login01.egr.duke.edu]46: cat hello.s
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
        .section        .note.GNU-stack,"",@progbits
[login01.egr.duke.edu]47:
```

- EEP! Intel assembly language! The compiler will happily show us the assembly code corresponding to our C (or any other language) code!

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

SAME
as 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

SAME
as Java!

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

```
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`

Control Flow – just like Java

SAME
as Java!

- Conditionals

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

- Loops

```
for (i = 0; i < max; i++) { ... }
while (i < max) {…}
```

Variable Scope: Global Variables

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

Similar to Java!

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

78

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

Makes a local X – separate from global X (this hides the global X within main)

Which X?
Global X = 78
Main's local X = 23

Is this a good idea? In most cases no, but it is legal and there are cases where it makes sense!

Functions – mostly like Java

Similar to 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

```
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…)

Similar to Java!

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

for (int i = 0; i< 25; i++)
   buf[i] = 'A'+i;        /* what does this do?  */
```

These are statically declared – i.e. of known max size before the program even runs, dynamically declared arrays will follow and are much more fun!

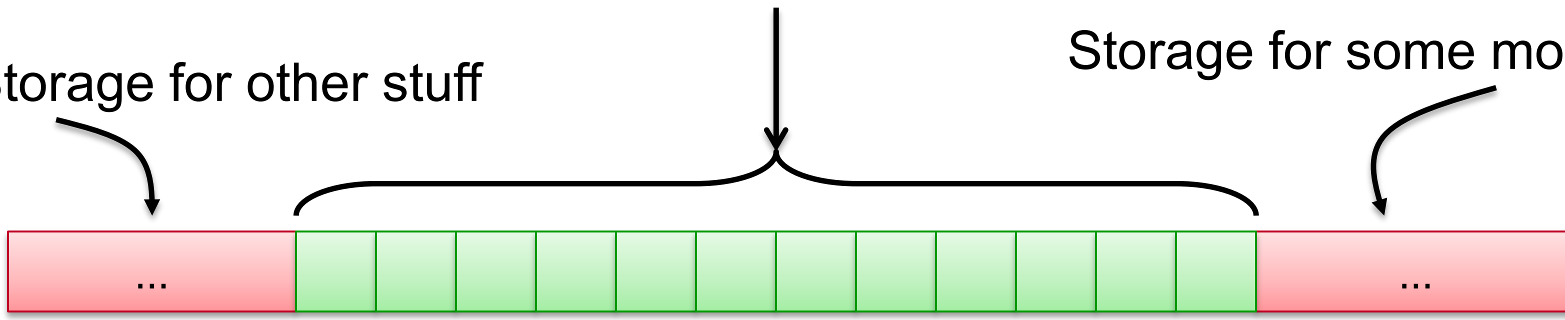
Memory Layout and Bounds Checking

DIFFERENT from Java!

Storage for array `int days_in_month[12];`

Storage for other stuff

Storage for some more stuff



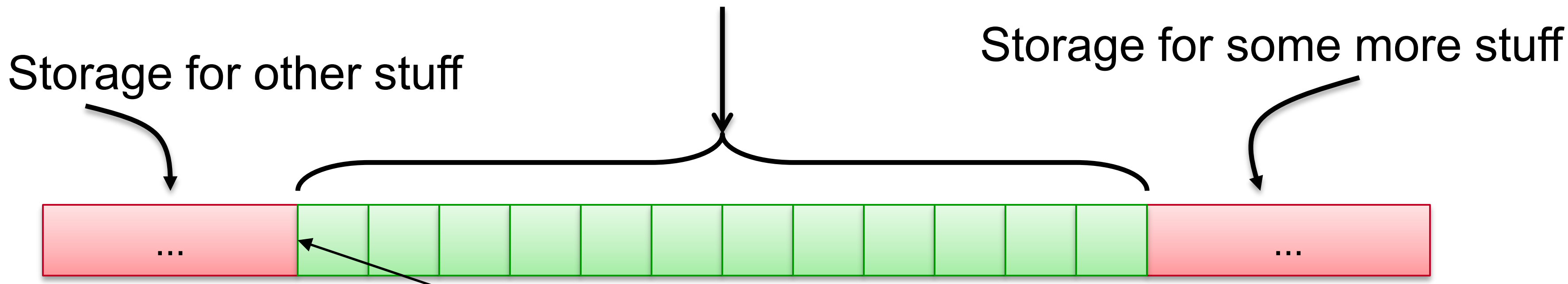
(each location shown here is an `int`)

- There is NO bounds checking in C
  - i.e., it's legal (but not advisable) to refer to `days_in_month[219]` or `days_in_month[-35]` !
  - who knows what is stored there?

Memory Layout and Bounds Checking

DIFFERENT from Java!

Storage for array **`int days_in_month[12];`**



(each location shown here is an **`int`**)

- Very specifically, "`days_in_month`" is here – the address of the beginning of `days_in_month[0]`

Strings – not quite like Java

DIFFERENT from Java!

- Strings
  - `char str1[256] = "hi";`
  - `str1[0] = 'h', str1[1] = 'i',str1[2] = 0;`
  - `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 itself
- C has built-in string operations
  - `#include <string.h>  // includes string operations`
  - `strlen(str1);`

Structures

DIFFERENT from Java!

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

- Structure definition with `struct` keyword

```
struct student_record {
     int id;
     float grade;
} rec1, rec2;
```

- Declare a variable of the structure type with `struct` keyword

```
struct student_record onerec;
```

- Access the structure member fields with dot ('.'), e.g. `structvar.member`

```
onerec.id = 12;
onerec.grade = 79.3;
```

Array of Structures

Similar to Java!

```
#include <stdio.h>
struct student_record {
        int id;
        float grade;
};

struct student_record myroster[200];  /* declare array of structs */
int main()
{
        myroster[23].id = 99;
        myroster[23].grade = 88.5;
}
```

Remember these run from `myroster[0]` to `myroster[199]`! There is no `myroster[200]` (well there is, but it is just the next unrelated thing in memory and almost certainly an error!

Console I/O in C

DIFFERENT from Java!

- I/O is provided by standard library functions
  - available on all platforms
- To use, your program must have

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

"Standard IO"

Not "studio"!!

- …and it doesn't hurt to also have

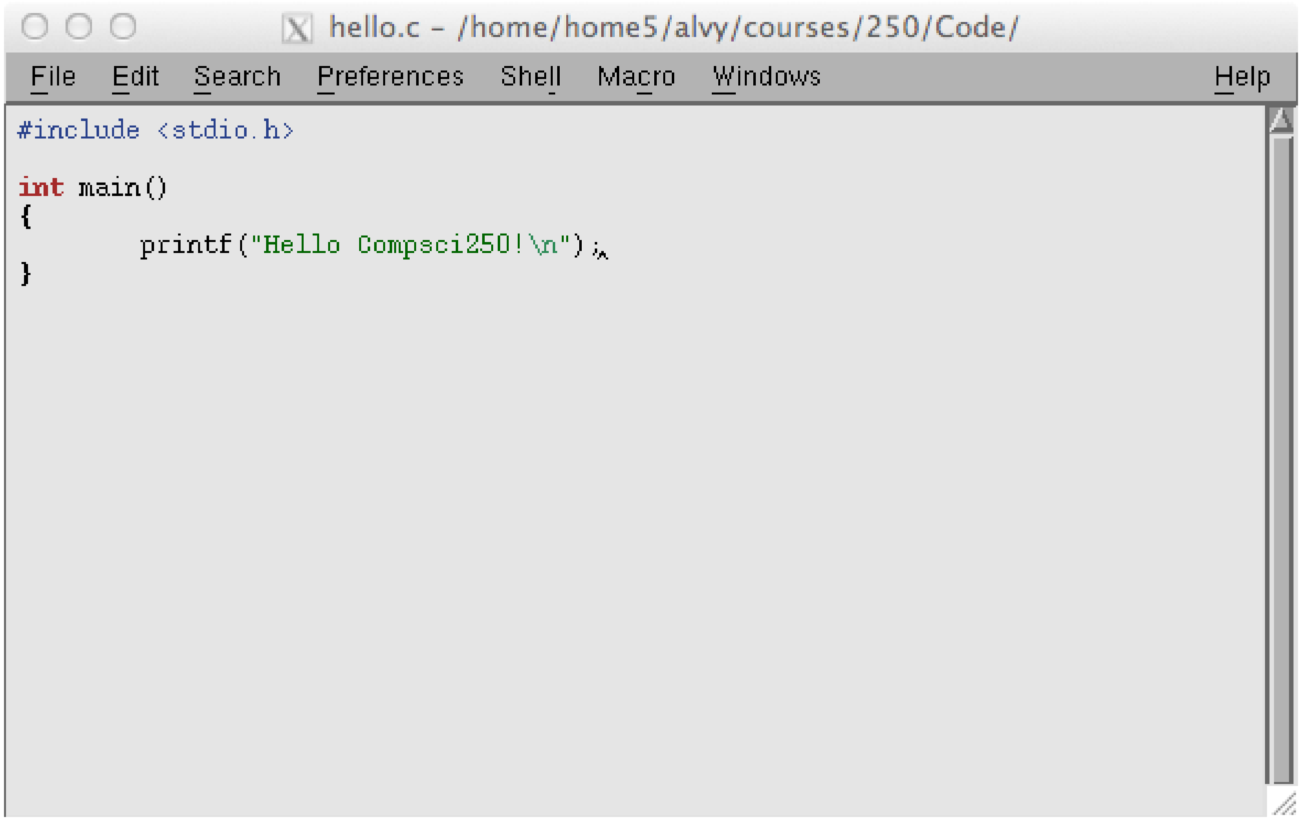
```
#include <stdlib.h>
```

"Standard library"

- *These "#" are preprocessor statements; the .h files define function types, parameters, and constants from the standard library. With* `#include`*, the contents of these files (which you can look at!) are simply included before compilation begins, but we don't clutter each file with all the gory details.*
- *There are other preprocessor statements we may or may not need!*

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)

DIFFERENT from Java!

- 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

DIFFERENT from Java!

```
#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 = ‘-’;                              FIXFIXFIX
        putchar(c);
    }
    return 0;
}
```

- EOF is End Of File (type Ctrl+D), 0x04 in ASCIIZ

Formatted I/O

```
#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() = print formatted
scanf() = scan (read) formatted

DIFFERENT from Java!

- `printf("format string", v1,v2,…);`
  - `\n` is newline character
- `scanf("format string",…);`
  - Returns number of matching items or EOF if at end-of-file

About printf and scanf

```
printf("Hello %s, you are %d years old.\n", name, age);
```

- Format specifiers:
  - `%d` Decimal integer (char/short/int/long/long long) — Doing scanf? Use & before var.
  - `%x` Hexadecimal integer (char/short/int/long/long long) — Doing scanf? Use & before var.
  - `%f` Float (float or double) — Doing scanf? Use & before var.
  - `%c` Character (char) — Doing scanf? Use & before var.
  - `%s` String (char[] or char*) — Doing scanf? DON'T use & before variable.
- Modifying them:

`%3d`	Minimum 3-characters, space padded right aligned	`_52`
`%-3d`	Same, but left aligned	`52_`
`%03d`	Same, but pad with zeroes instead of spaces	`052`
`%.2f`	Float, two digits after decimal	`2.52`
`%5.2f`	Float, two digits after decimal, space padded to 5 chars	`_2.52`

Read the **`printf`** & **`scanf`** documentation for more exciting info!!!

Example: Reading Input in a Loop

DIFFERENT from Java!

```
#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 `./prog < file.in` to redirect in from a file instead
- 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)

DIFFERENT from Java!

```
#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):

DIFFERENT from Java!

```
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.

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

  - The atoi function just returns 0 for non-integers, so atoi("0")==atoi("hurfdurf") ☹

atoi stands for a-to-i, as in array-to-integer, because strings are character arrays.

Header Files, Separate Compilation, Libraries

DIFFERENT from Java!

- 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

Similar to Java!

- 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`
  - `argv[0] = “./myProgram”, argv[1]=“dan”, argv[2]=“250”`

```
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]);
  }
}
```

Command-line arguments vs stdin

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Syntax: ./is <adjective>\n");
        return 0;
    }
    printf("Name: ");
    char name[64];
    scanf("%s",name);
    printf("%s is %s.\n", name, argv[1]);
    return 0;
}
```

Command-line arguments
- Typed after program name in shell
- Come in via argv[]
- Strings – can be parsed with sscanf

Stdin
- Typed into the running program
- Can be read with scanf

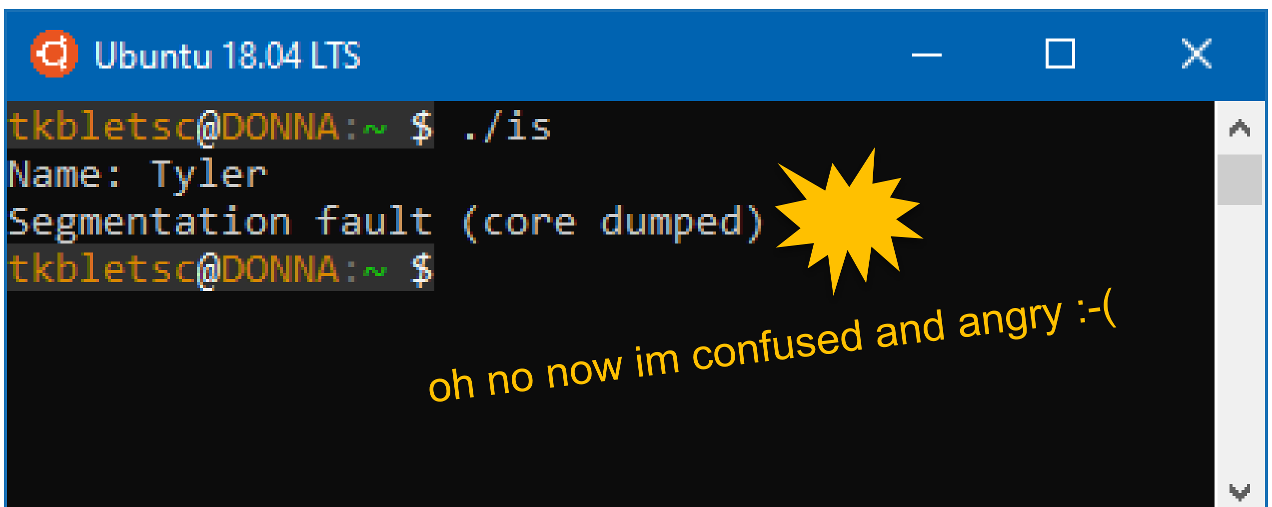
Ubuntu 18.04 LTS

```
tkbletsc@DONNA:~ $ ./is
Syntax: ./is <adjective>
tkbletsc@DONNA:~ $ ./is cool
Name: Tyler
Tyler is cool.
tkbletsc@DONNA:~ $
```

Also: DO ERROR CHECKING!

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    /*if (argc != 2) {
        printf("Syntax: ./is <adjective>\n");
        return 0;
    }*/
    printf("Name: ");
    char name[64];
    scanf("%s",name);
    printf("%s is %s.\n", name, argv[1]);
    return 0;
}
```

What if this were removed?



The Big Differences Between C and Java

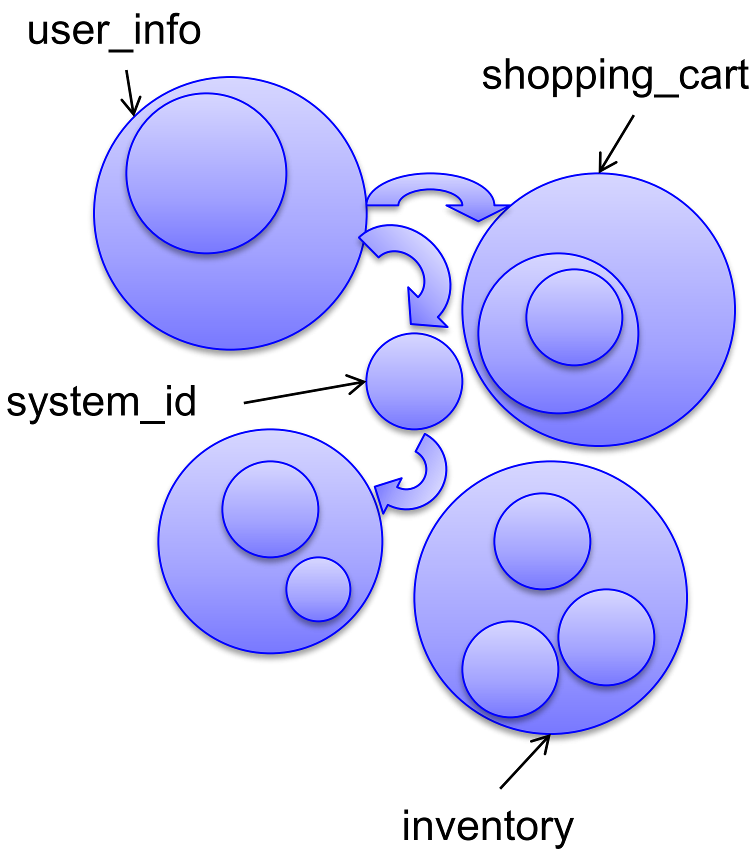
1) Java is object-oriented, while C is not

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



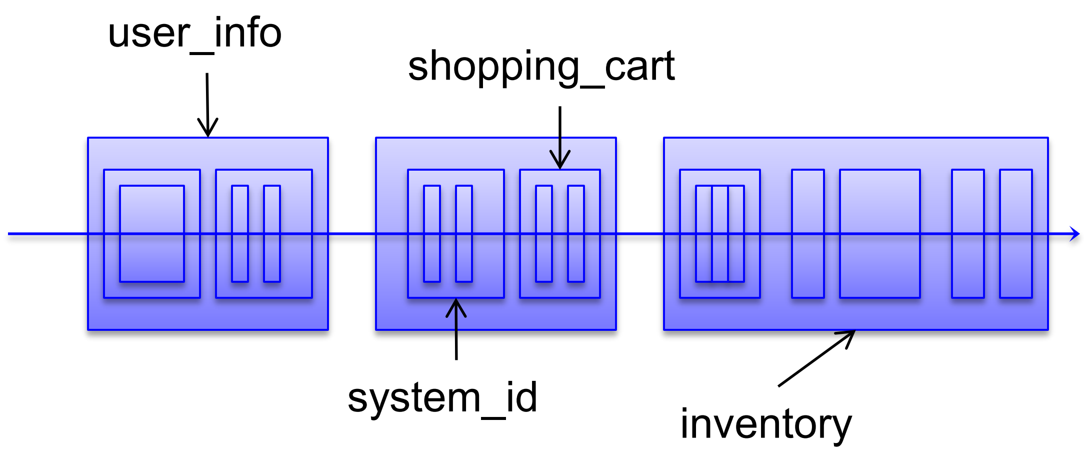


Figure from Rudra Dutta, NCSU, 2007

Let’s look at memory addresses!

- You can find the address of ANY variable with:

DIFFERENT from Java!

&

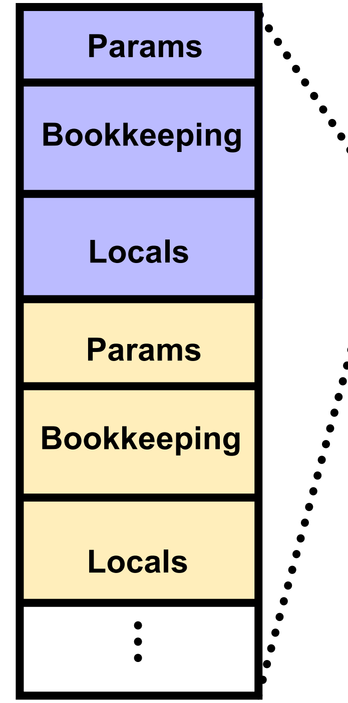
The address-of operator

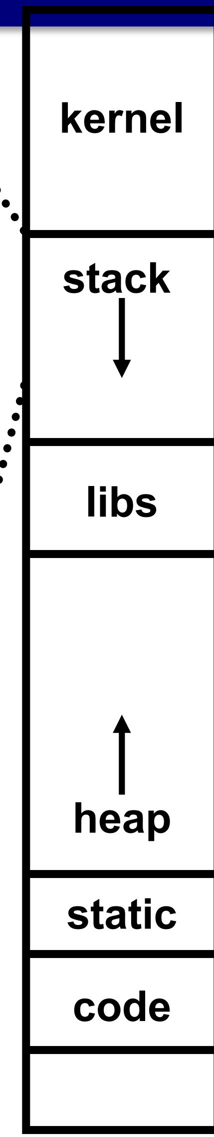
```
int v = 5;
printf(“%d\n”,v);
printf(“%p\n”,&v);
```

```
$ g++ x.c && ./a.out
5
0x7fffd232228c
```

Testing where variables live

```
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);
}
```



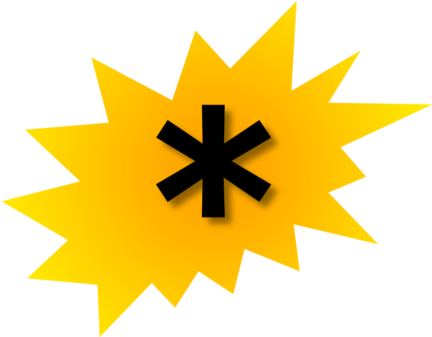



```
$ g++ x.c && ./a.out
&x:     0x601020
&msg:   0x601024
&argc:  0x7fff85b78c2c
&argv:  0x7fff85b78c20
&v:     0x7fff85b78c38
&pi:    0x7fff85b78c3c
```

What's a pointer?

- It's a <u>memory address</u> you treat as a <u>variable</u>
- You declare pointers with:

DIFFERENT from Java!



The *dereference* operator

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

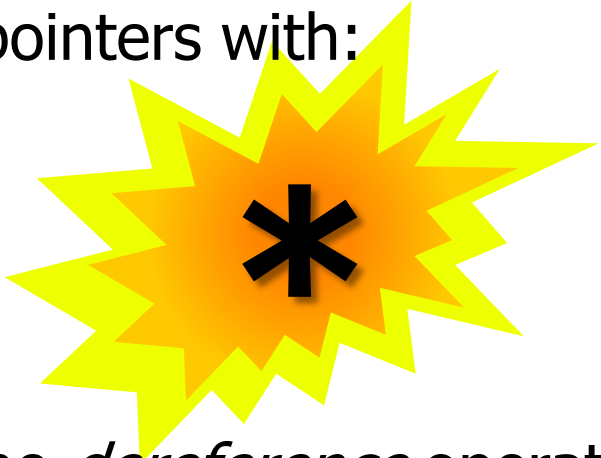
Append to any data type

```
$ g++ x.c && ./a.out
5
0x7fffe0e60b7c
```

What's a pointer?

DIFFERENT from Java!

- You can <u>look up</u> what's stored *at* a pointer!
- You **dereference** pointers with:



The *dereference* operator

```
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

```
$ g++ x.c && ./a.out
5
0x7fffe0e60b7c
5
```

Different types use different amounts of memory

- If I have an n-bit integer:
  - And it's **unsigned**, then I can represent $\{0 \;..\; 2^n - 1\}$
  - And it's **signed**, then I can represent $\{-(2^{n-1}) \;..\; 2^{n-1} - 1\}$
- Result:

Don't worry about this "signed" vs "unsigned" stuff yet. We'll cover this later, and you'll see this slide again.

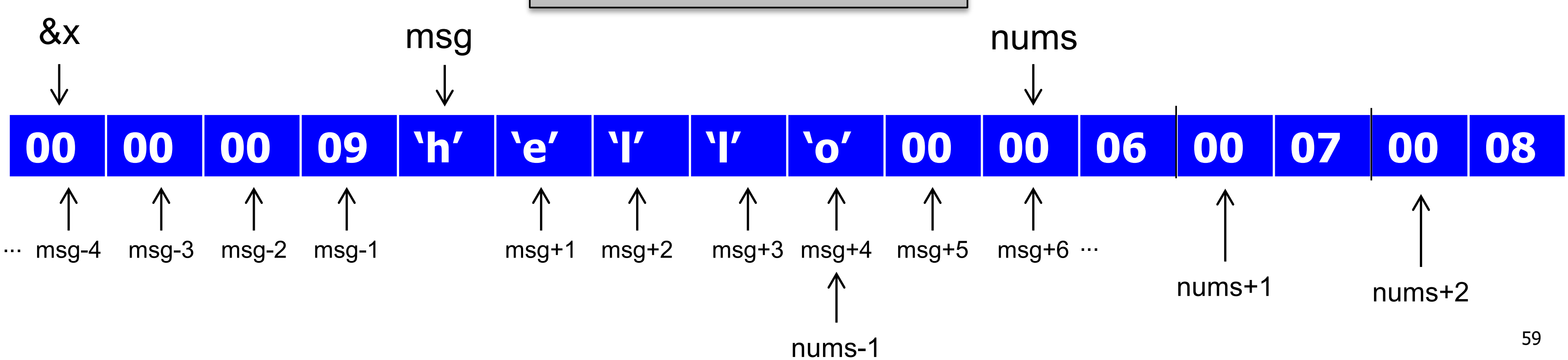
Size in bits	Size in bytes	Datatype	Unsigned range	Signed range
8	1	`char`	0 .. 255	-128 .. 127
16	2	`short`	0 .. 65,535	-32,768 .. 32,767
32	4	`int`	0 .. 4,294,967,295	-2,147,483,648 .. 2,147,483,647
64	8	`long long`	0 .. 18,446,744,073,709,600,000	-9,223,372,036,854,780,000 .. 9,223,372,036,854,780,000

- A **`float`** is 32 bits (4 bytes); a **`double`** is 64 bits (8 bytes)
- Size of a pointer? Depends on the platform!
  - Our **x86** platform for C: pointers are 64 bits (8 bytes)
  - The **MIPS** platform we'll learn soon: pointers will be 32 bits (4 bytes)

What is an array?

- The shocking truth:
  You've been using pointers all along!
- Every array <u>*IS*</u> 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 <u>*things*</u> 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};
```

Array lookup	Pointer reference	Type
x	x	int*
x[0]	*x	int
x[5]	*(x+5)	int
x[n]	*(x+n)	int
&x[0]	x	int*
&x[5]	x+5	int*
&x[n]	x+n	int*

(In case you don't believe me)

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

```
$ g++ x.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

DIFFERENT from Java!

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

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

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

```
$ g++ x.c && ./a.out
Here are words
Here_are_words
```

Shortening that function

```
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?

```
#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**!

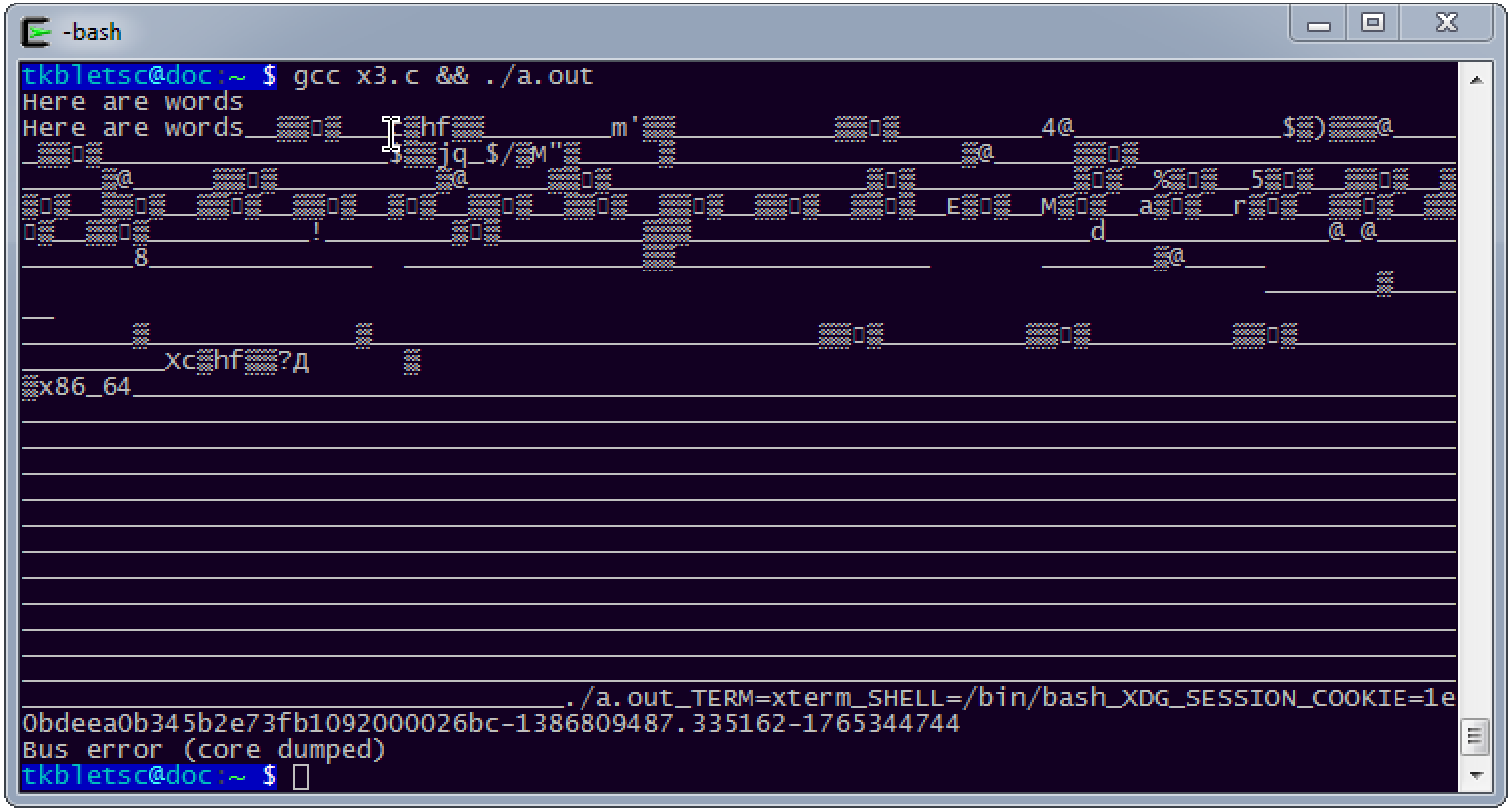
```
#include <stdio.h>

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…

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

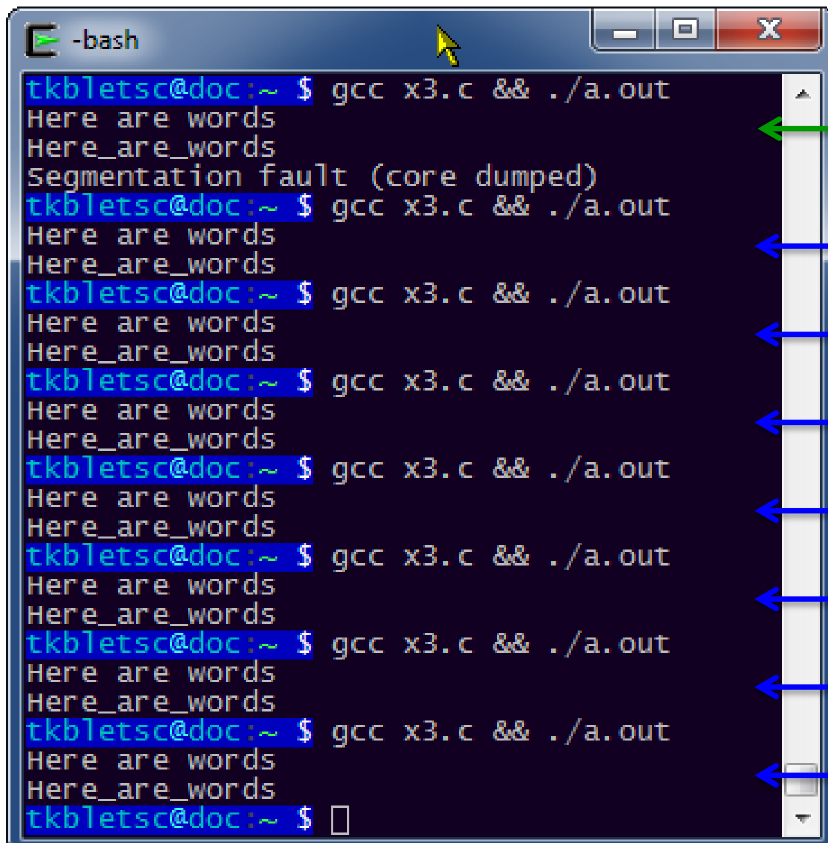
```
int main() {
  char msg[] = "Here are words";
  puts(msg);
  underscorify_bad(msg);
  puts(msg);
}
```



Almost fixed…

```c
void underscorify_bad2(char* s) {
  char* p = s;
  while (*p != '0') {
    if (*p == ' ') {
      *p = '_';
    }
    p++;
  }
}
```

```c
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!!

Effects of pointer mistakes

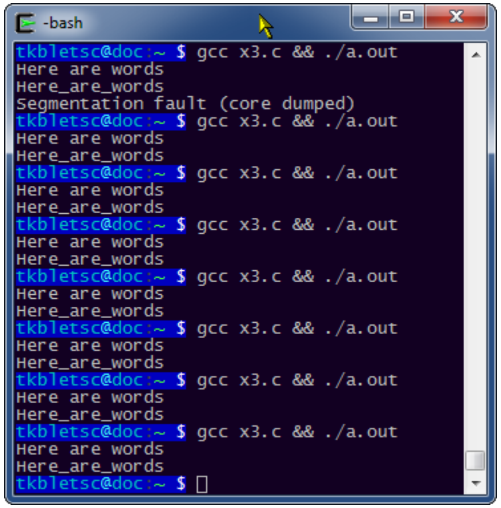
Access an array out of bounds
or some other invalid pointer location?

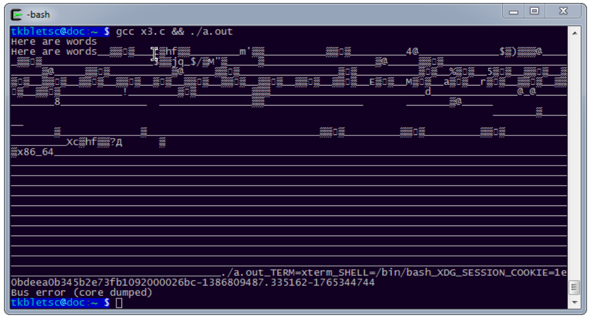
No visible effect

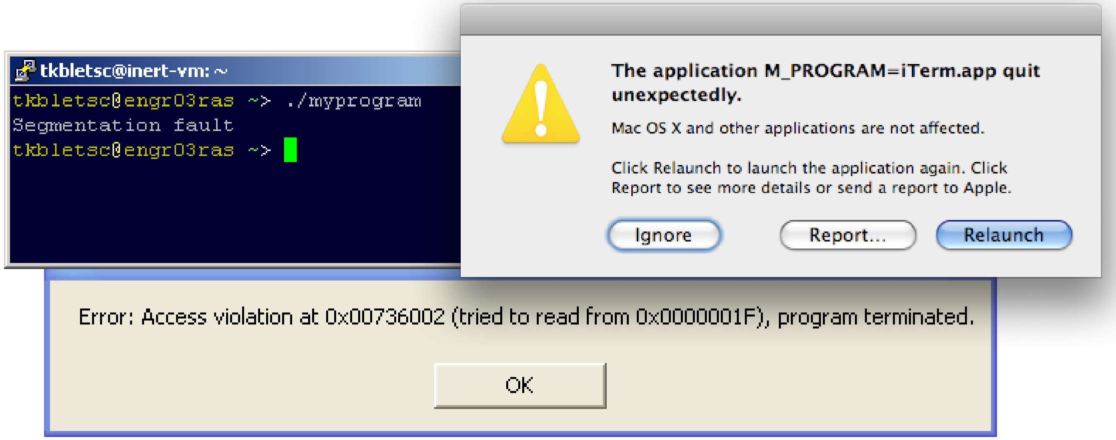
Silent corruption & bad results

Totally weird behavior

Program crash with OS error







Pointer summary

- **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;
```

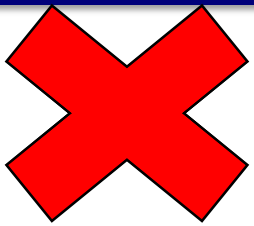
- You can find the data at a memory address with the **dereference operator *** prepended to a pointer expression:

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

- Arrays in C are just pointers to a chunk of memory
- Pointer math is done in *units of the underlying type*

  (An array of `int`s walks 4 bytes at a time)
- Don't screw up

Pass by Value vs. Pass by Reference

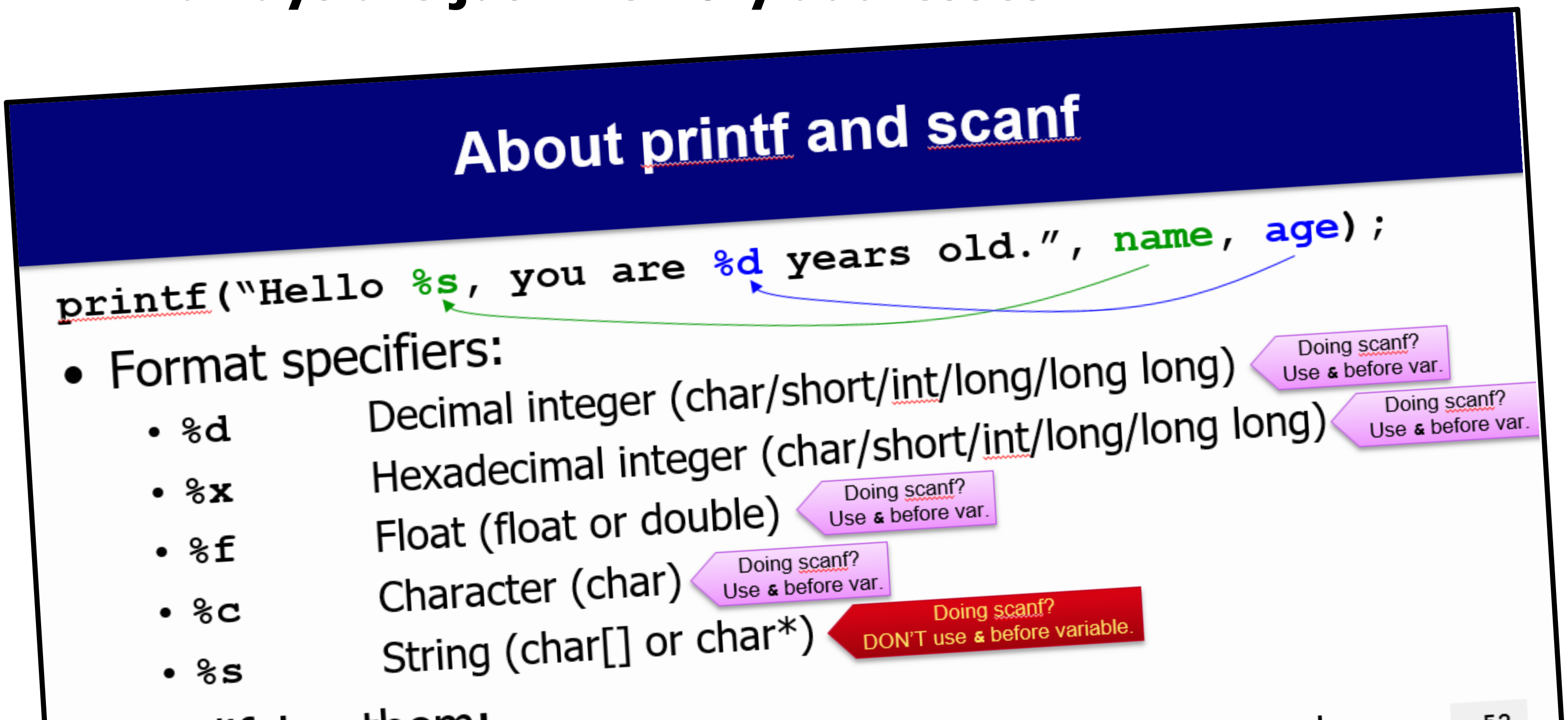


```
void swap (int x, int y){
  int temp = x;
  x = y;
  y = temp;
}
int main() {
  int a = 3;
  int b = 4;
  swap(a, b);
  printf(“a = %d, b= %d\n”, a, b);
}
```

```
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);
}
```

About "About printf and scanf"

- Remember this slide?
  - In **`scanf`**, why do **`%d`**, **`%x`**, **`%f`**, **`%c`** use a **`&`** before the variable?
    - Need to pass a pointer so scanf can mess with the content of them!
  - Why doesn't **`%s`** use a **`&`** before the variable?
    - Because **strings are arrays**, and **arrays are just memory addresses**!



C Memory Allocation: introducing the heap

DIFFERENT from Java!

- So far, we have **local** variables and **global** variables
  - Locals are short-lived (die when function returns).
  - Globals are long-lived but fixed-size (defined at compile time).
- What if we want memory that is allocated at runtime and long-lived?
  - You had this in Java: **objects**!
- C doesn't have objects, but you can allocate memory for stuff!
  - This is called **heap memory**.
  - Most memory used by programs is in heap memory!
  - Think: Tabs in your web browser.
    - Make a tab? Allocate
    - Close a tab? Deallocate

C Memory Allocation

DIFFERENT from Java!

- How do you allocate an object in Java?
  - The **`new`** keyword
- What do you do when you are finished with object?
  - Nothing, you just stop using it
  - How? JVM provides garbage collection
    - Counts references to objects, when refs== 0 can reuse
- How do you allocate heap memory in C?
  - The **`malloc`**, **`calloc`**, and **`realloc`** functions
- What do you do when you're finished with the memory?
  - You free it manually with the **`free`** function
  - C doesn't have garbage collection! Must explicitly manage memory.
  - The power is yours!

C Memory Allocation

DIFFERENT from Java!

- **`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

DIFFERENT from Java!

- **`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 an existing heap allocation
    - 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

```
#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

```
struct student_rec {
       int id;
       float grade;
};
struct student_rec* my_ptr = malloc(sizeof(struct student_rec));
// my_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
```

Linked lists: C vs Java

```c
struct Node {
  int id;
  struct Node* next;
};
struct Node* new_node(int id) {
  struct Node* newguy =
    (struct Node*) malloc(sizeof(struct Node));
  newguy->id = id;
  newguy->next = NULL;
  return newguy;
}

struct Node* prepend_to_list(struct Node* head, int id) {
  struct Node* newguy = new_node(id);
  newguy->next = head;
  return newguy;
}

void insert_after(struct Node* target, int id) {
  struct Node* newguy = new_node(id);
  newguy->next = target->next;
  target->next = newguy;
}

void print_list(struct Node* head) {
  for (struct Node* p = head; p != NULL; p = p->next) {
    printf("%d ", p->id);
  }
  printf("\n");
}
```

Initializing right after we malloc avoids trouble later!

```java
public class LinkedList {
  public static class Node {
    public int id;
    protected Node next;

    Node(int id) {


      this.id = id;
      this.next = null;

    }
  }
  public static Node prepend_to_list(Node head, int id) {
    Node newguy = new Node(id);
    newguy.next = head;
    return newguy;
  }

  public static void insert_after(Node target, int id) {
    Node newguy = new Node(id);
    newguy.next = target.next;
    target.next = newguy;
  }

  public static void print_list(Node head) {
    for (Node p = head; p != null; p = p.next) {
      System.out.printf("%d ", p.id);
    }
    System.out.printf("\n");
  }


}
```

Java does these steps implicitly!

Note: full runnable versions of these programs are available from the course site: `LinkedList.c` and `LinkedList.java`

Linked lists: Freeing the list in C

- When done, need to walk the list and free each node
- May be tempted to write the following:

```
void free_list_naive(struct Node* head) {
  while (head) {
    free(head);
    head = head->next;
  }
}
```

Free the block, okay

This arrow means dereference: we're using the memory we just freed!

- This is a use-after-free bug! It may **crash**!
- **You cannot rely on a freed piece of memory!**
- Solution: rescue out the next pointer into a local first:

```
void free_list(struct Node* head) {
  while (head) {
    struct Node* nextguy = head->next;
    free(head);
    head = nextguy;
  }
}
```

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

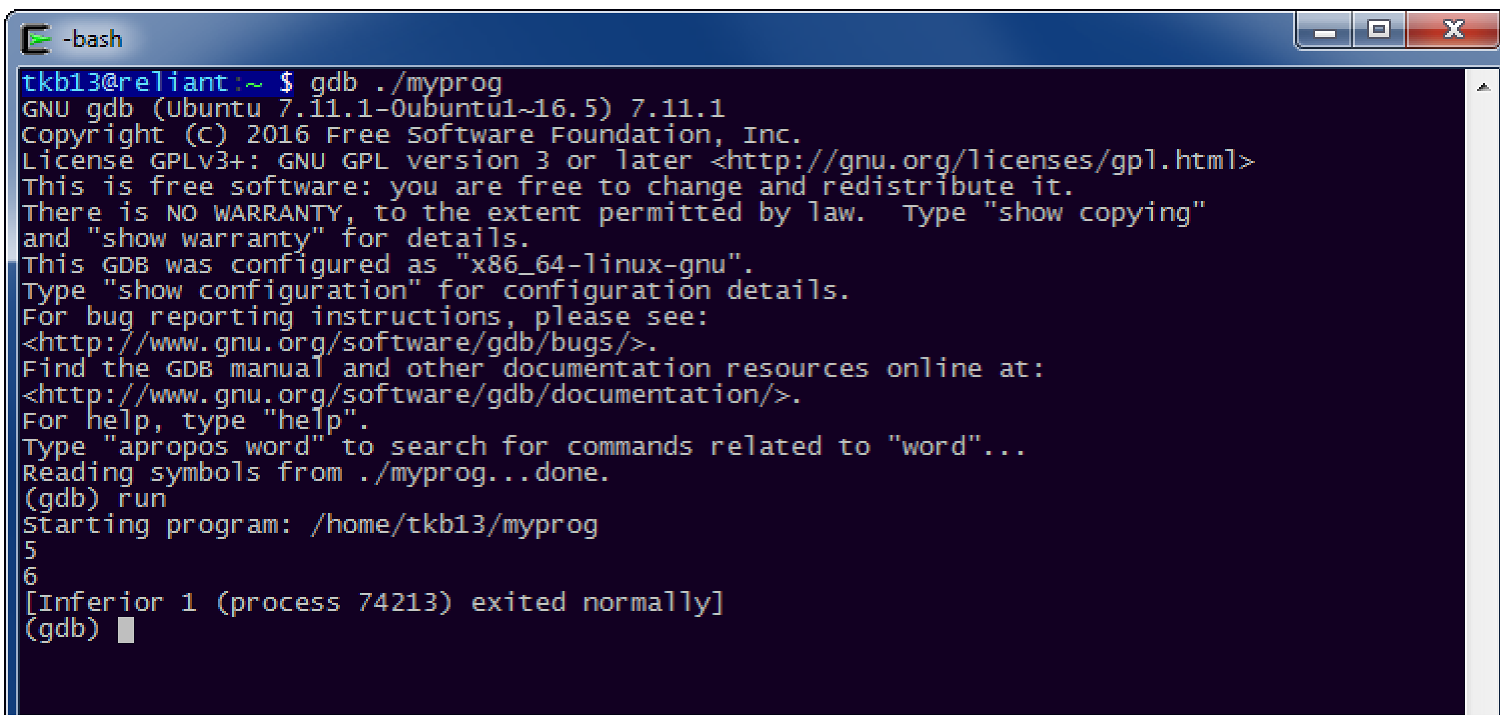
Gdb

- To start:

  $ **gdb ./myprog**

- To run:

  (gdb) **run** *arguments*



gdb commands

list <line> **list <function>** **list <line>,<line>**	list (show) 10 lines of code at specified location in program List from first line to last line
run	start running the program
continue **step** **next**	continue execution single step execution, including into functions that are called single step over function calls
print <var> **printf "fmt", <var>** **display <var>** **undisplay <var>**	show variable value show variable each time execution stops

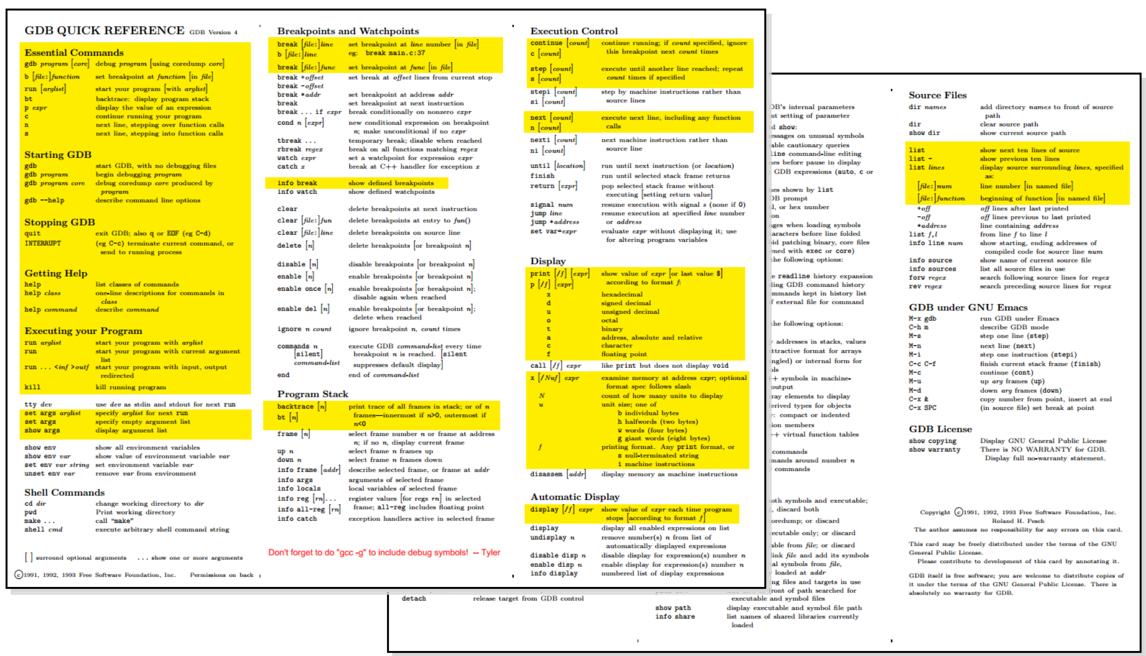
gdb commands

`break <line>`
`break <function>`
`break <line> if <cond>`
	set breakpoints (including conditional breakpoints)

`info breakpoints` `delete breakpoint <n>`	list, and delete, breakpoints
`set <var> <expr>`	set variable to a value
`backtrace full` `bt`	show the call stack & args arguments and local variables

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

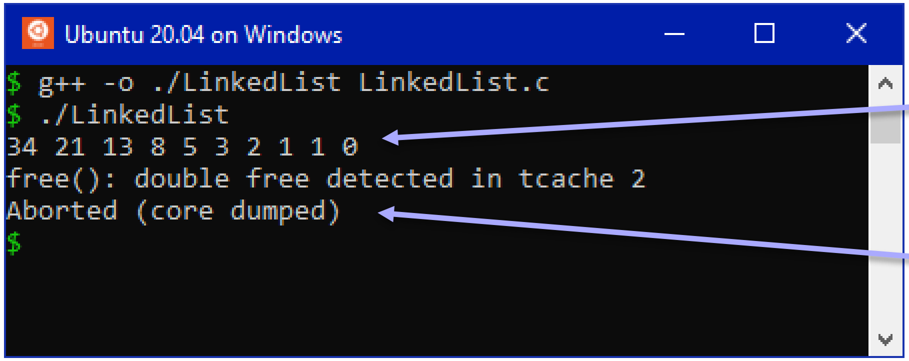
```
tkb13@login-teer-15:~ $ cat memleak.c
#include <stdlib.h>
void f() {
        char* p = (char*) malloc(20);
}

int main() {
        char* c = (char*) malloc(10);
        f();
}

tkb13@login-teer-15:~ $ valgrind --leak-check=yes ./memleak
==17572== Memcheck, a memory error detector
==17572== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==17572== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==17572== Command: ./memleak
==17572==
==17572==
==17572== HEAP SUMMARY:
==17572==     in use at exit: 30 bytes in 2 blocks
==17572==   total heap usage: 2 allocs, 0 frees, 30 bytes allocated
==17572==
==17572== 10 bytes in 1 blocks are definitely lost in loss record 1 of 2
==17572==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==17572==    by 0x4005CD: main (memleak.c:7)
==17572==
==17572== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==17572==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==17572==    by 0x4005B5: f() (memleak.c:3)
==17572==    by 0x4005D6: main (memleak.c:8)
==17572==
==17572== LEAK SUMMARY:
==17572==    definitely lost: 30 bytes in 2 blocks
==17572==    indirectly lost: 0 bytes in 0 blocks
==17572==      possibly lost: 0 bytes in 0 blocks
==17572==    still reachable: 0 bytes in 0 blocks
==17572==         suppressed: 0 bytes in 0 blocks
==17572==
==17572== For counts of detected and suppressed errors, rerun with: -v
==17572== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
tkb13@login-teer-15:~ $
```

Debugging our bad free code

- Remember this broken code?

```c
void free_list_naive(struct Node* head) {
  while (head) {
    free(head);
    head = head->next;
  }
}
```

- Let’s test it! First, we compile and run:



Program uses a linked list to show the first ten Fibonacci numbers backwards. Correct output…

But then it crashed ☹ Why??????????????

- Dang, time to debug…

Debugging our bad free code

Ubuntu 20.04 on Windows

```
$ gdb ./LinkedList
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./LinkedList...
(No debugging symbols found in ./LinkedList)
(gdb) run
Starting program: /mnt/c/Users/tkbletsc/Dropbox/Duke/ECE250/Slides/resources/linkedlist/LinkedList
34 21 13 8 5 3 2 1 1 0
free(): double free detected in tcache 2

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff5d5859 in __GI_abort () at abort.c:79
#2  0x00007ffff6403ee in __libc_message (action=action@entry=do_abort, fmt=fmt@entry=0x7ffff76a285 "%s\n")
    at ../sysdeps/posix/libc_fatal.c:155
#3  0x00007ffff64847c in malloc_printerr (
    str=str@entry=0x7ffff76c5d0 "free(): double free detected in tcache 2") at malloc.c:5347
#4  0x00007ffff64a0ed in _int_free (av=0x7ffff79bb80 <main_arena>, p=0x8005000, have_lock=0)
    at malloc.c:4201
#5  0x00000000080012ca in free_list_naive(Node*) ()
#6  0x0000000008001397 in main ()
(gdb)
```

Launch gdb with the program as the argument

Use `run` to actually execute it

Hmm, where did it crash exactly? `bt` will show us the stack backtrace

Well that's the function, but where's the dang line number?????? Oh noooooooooooooo

- We forgot to compile with **-g** so there's no debug symbols!

Debugging our bad free code

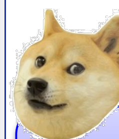
U… 20.04 on Windows

```
$ g++ -g -o ./LinkedList LinkedList.c
$ gdb ./LinkedList
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./LinkedList...
(gdb) run
Starting program: /mnt/c/Users/tkbletsc/Dropbox/Duke/ECE250/Slides/resources/linkedlist/LinkedList
34 21 13 8 5 3 2 1 1 0
free(): double free detected in tcache 2

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff5d5859 in __GI_abort () at abort.c:79
#2  0x00007ffff6403ee in __libc_message (action=action@entry=do_abort, fmt=fmt@entry=0x7fffff76a285 "%s\n")
    at ../sysdeps/posix/libc_fatal.c:155
#3  0x00007ffff64847c in malloc_printerr (
    str=str@entry=0x7fffff76c5d0 "free(): double free detected in tcache 2") at malloc.c:5347
#4  0x00007ffff64a0ed in _int_free (av=0x7fffff79bb80 <main_arena>, p=0x8005000, have_lock=0)
    at malloc.c:4201
#5  0x00000000080012ca in free_list_naive (head=0x8005010) at LinkedList.c:42
#6  0x0000000008001397 in main (argc=1, argv=0x7ffffffedbd8) at LinkedList.c:67
(gdb)
```

Recompile with **-g**!
Then gdb again.

Use `run` again

Use `bt` for stack backtrace again



Wow! Such line numbers!
Much arguments!

Debugging our bad free code

Ubuntu 20.04 on Windows

```
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff5d5859 in __GI_abort () at abort.c:79
#2  0x00007ffff6403ee in __libc_message (action=action@entry=do_abort, fmt=fmt@entry=0x7ffff76a285 "%s\n")
    at ../sysdeps/posix/libc_fatal.c:155
#3  0x00007ffff64847c in malloc_printerr (
    str=str@entry=0x7ffff76c5d0 "free(): double free detected in tcache 2") at malloc.c:5347
#4  0x00007ffff64a0ed in _int_free (av=0x7ffff79bb80 <main_arena>, p=0x8005000, have_lock=0)
    at malloc.c:4201
#5  0x00000000080012ca in free_list_naive (head=0x8005010) at LinkedList.c:42
#6  0x0000000008001397 in main (argc=1, argv=0x7ffffffedbd8) at LinkedList.c:67
(gdb) print head
No symbol "head" in current context.
(gdb) frame 5
#5  0x00000000080012ca in free_list_naive (head=0x8005010) at LinkedList.c:42
42              free(head);
(gdb) print head
$1 = (Node *) 0x8005010
(gdb) list
37          printf("\n");
38      }
39
40      void free_list_naive(struct Node* head) {
41          while (head) {
42              free(head);
43              head = head->next;
44          }
45      }
46
(gdb)
```

Use `frame` to set what stack frame we're "in" for printing purposes.

We can `print` variables and `list` code, and much more!

- But suppose this isn't clear enough? It doesn't actually say we used after free…

Debugging our bad free code

- Valgrind is a great tool for memory issues and crashes

Ubuntu 20.04 on Windows

```
$ valgrind ./LinkedList
==1930== Memcheck, a memory error detector
==1930== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1930== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1930== Command: ./LinkedList
==1930==
==1930== error calling PR_SET_PTRACER, vgdb might block
34 21 13 8 5 3 2 1 1 0
==1930== Invalid read of size 8
==1930==    at 0x1092CE: free_list_naive(Node*) (LinkedList.c:43)
==1930==    by 0x109396: main (LinkedList.c:67)
==1930==  Address 0x4a4b318 is 8 bytes inside a block of size 16 free'd
==1930==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1930==    by 0x1092C9: free_list_naive(Node*) (LinkedList.c:42)
==1930==    by 0x109396: main (LinkedList.c:67)
==1930==  Block was alloc'd at
==1930==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1930==    by 0x1091C1: new_node(int) (LinkedList.c:15)
==1930==    by 0x1091FD: prepend_to_list(Node*, int) (LinkedList.c:22)
==1930==    by 0x10935D: main (LinkedList.c:60)
==1930==
==1930==
==1930== HEAP SUMMARY:
==1930==     in use at exit: 0 bytes in 0 blocks
==1930==   total heap usage: 11 allocs, 11 frees, 4,256 bytes allocated
==1930==
==1930== All heap blocks were freed -- no leaks are possible
==1930==
==1930== For lists of detected and suppressed errors, rerun with: -s
==1930== ERROR SUMMARY: 10 errors from 1 contexts (suppressed: 0 from 0)
$
```

Run `valgrind` followed by the full command to debug

Normal program output

On line 43, we tried to read a piece of data 8 bytes in size illegally.

The address in question is analyzed; it's inside a block that was recently freed. Here's exactly *where* it was freed (line 42).

By the way, here's where this block was originally allocated, too, in case that helps.

All output lines that start like this are from valgrind; the number is the Process ID (pid) of the running program

- Wow, that tells the whole story! Thanks, valgrind!
  - Read the *whole story* that valgrind tells you, it's helping you!

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?