

# ECE/CS 250

# Computer Architecture

Fall 2021

## C Programming

Tyler Bletsch  
Duke University

Slides are derived from work by  
Daniel J. Sorin (Duke), Andrew Hilton (Duke), Alvy Lebeck (Duke),  
Benjamin Lee (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
- 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.

# 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



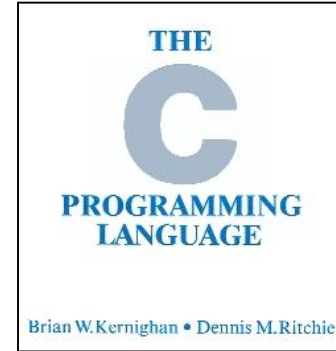
Cool, it worked!

Told ya.

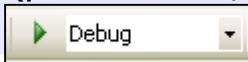
# What were they thinking?

- 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



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
```

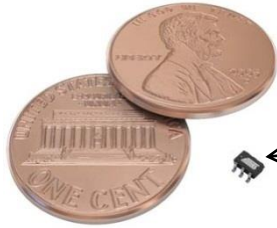
# Why C?

- Why C for humanity?
  - It's a "portable assembly language"
  - Useful in OS and embedded systems and 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
```

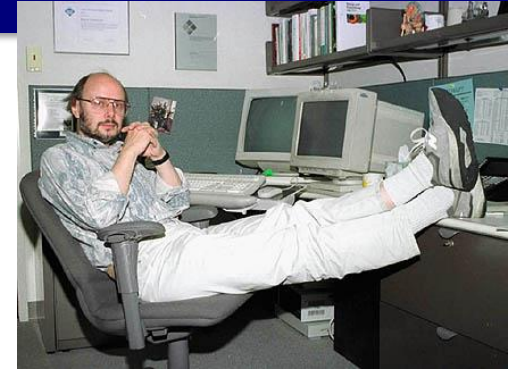
Language	Size of executable	Size of runtime (ignoring libraries)	Total size	RAM used
Java				
Python				
Desktop C				
Embedded C (Arduino)				

Max: 1024 B

Max: 32 B

# What about C++?

- 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

```
#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

```
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!

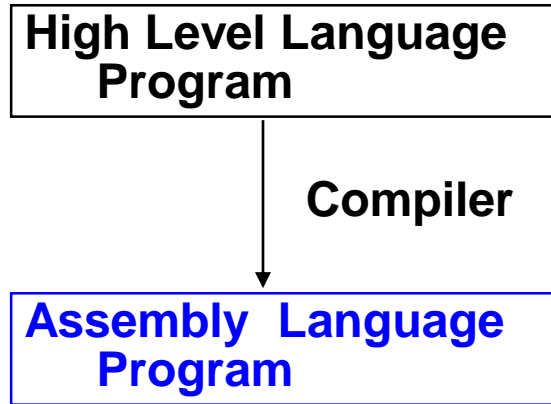


Don't you gimme no  
"it worked on my box"  
nonsense!

# How to access Duke Linux machines?

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

# HLL → Assembly Language

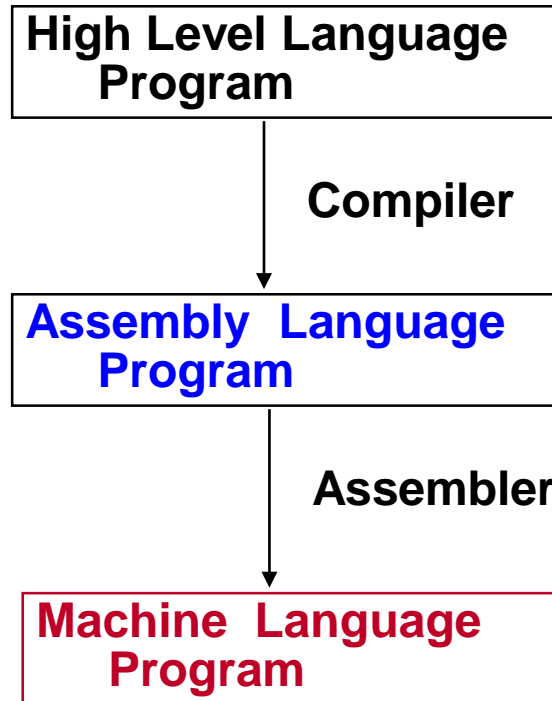


```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
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
- But most code is written in HLLs and **compiled**
  - **Compiler** is a program that automatically converts HLL to assembly

# Assembly Language → Machine 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)
```

```
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

# 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?

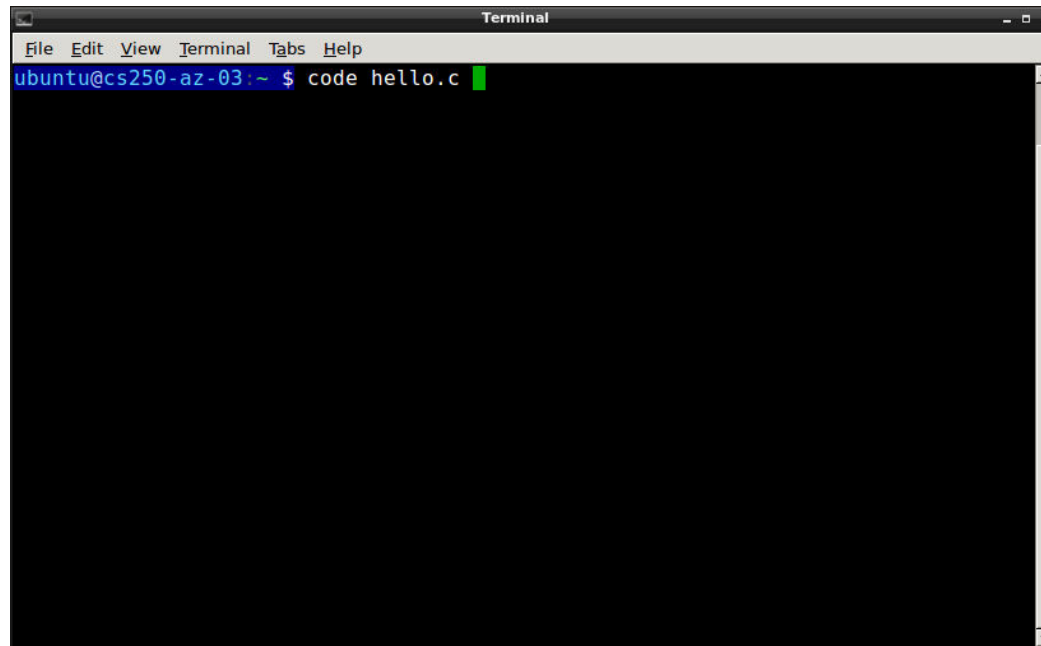
- Compile Java Source to Java Byte codes
- Java Virtual Machine (JVM) interprets/translates Byte codes
- JVM is a program executing on the hardware
  
- Java has lots of features that make it easier to program without making mistakes → training wheels are nice
  
- JVM handles memory for you
  - What 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
  - 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

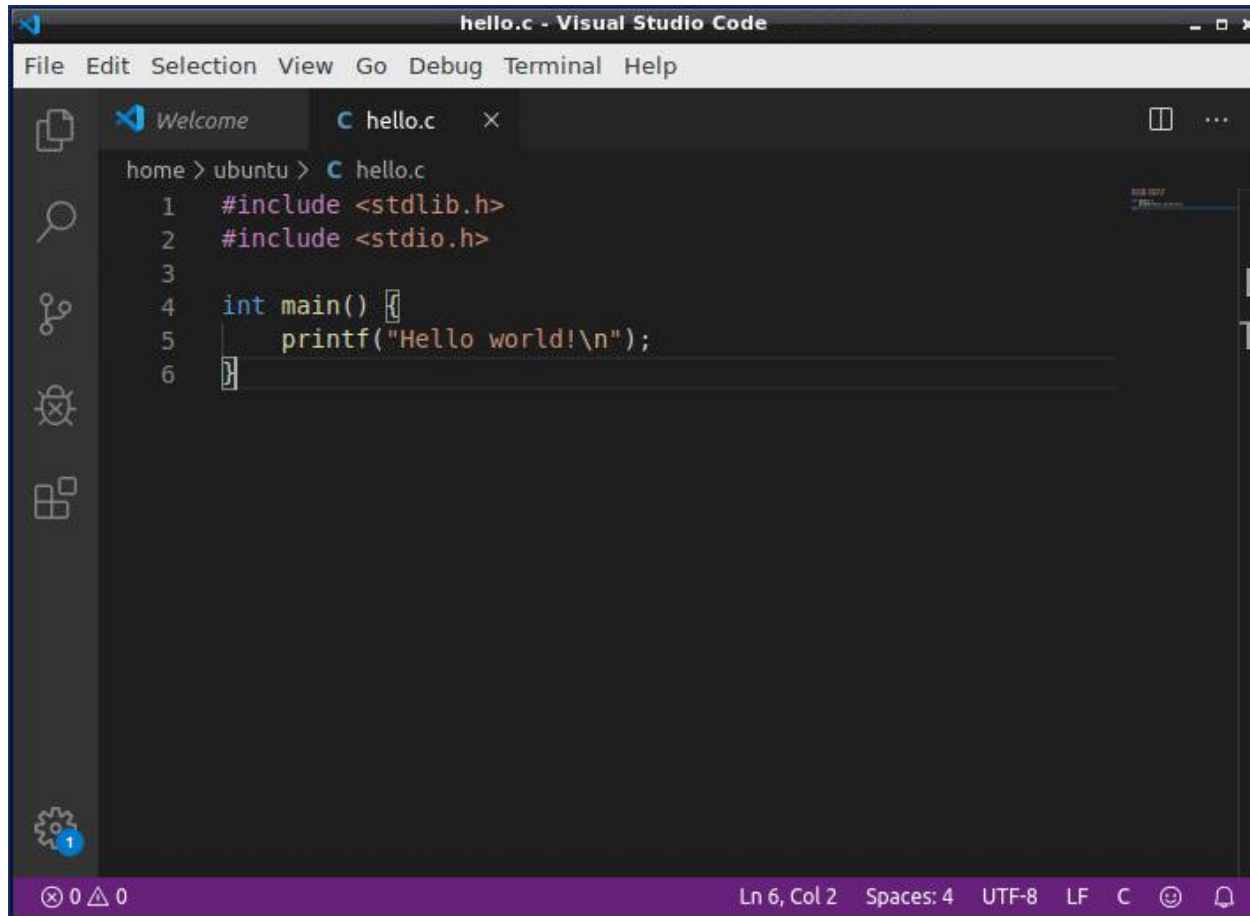
- We are not using a development environment (IDE)
- You will create programs starting with an empty file!
- Files should use .c file extension (e.g., hello.c)
- On a Linux machine, edit files with your chosen editor, e.g. Visual Studio Code (executable from command line as `code <file>`)



```
Terminal
File Edit View Terminal Tabs Help
ubuntu@cs250-az-03:~ $ code hello.c
```

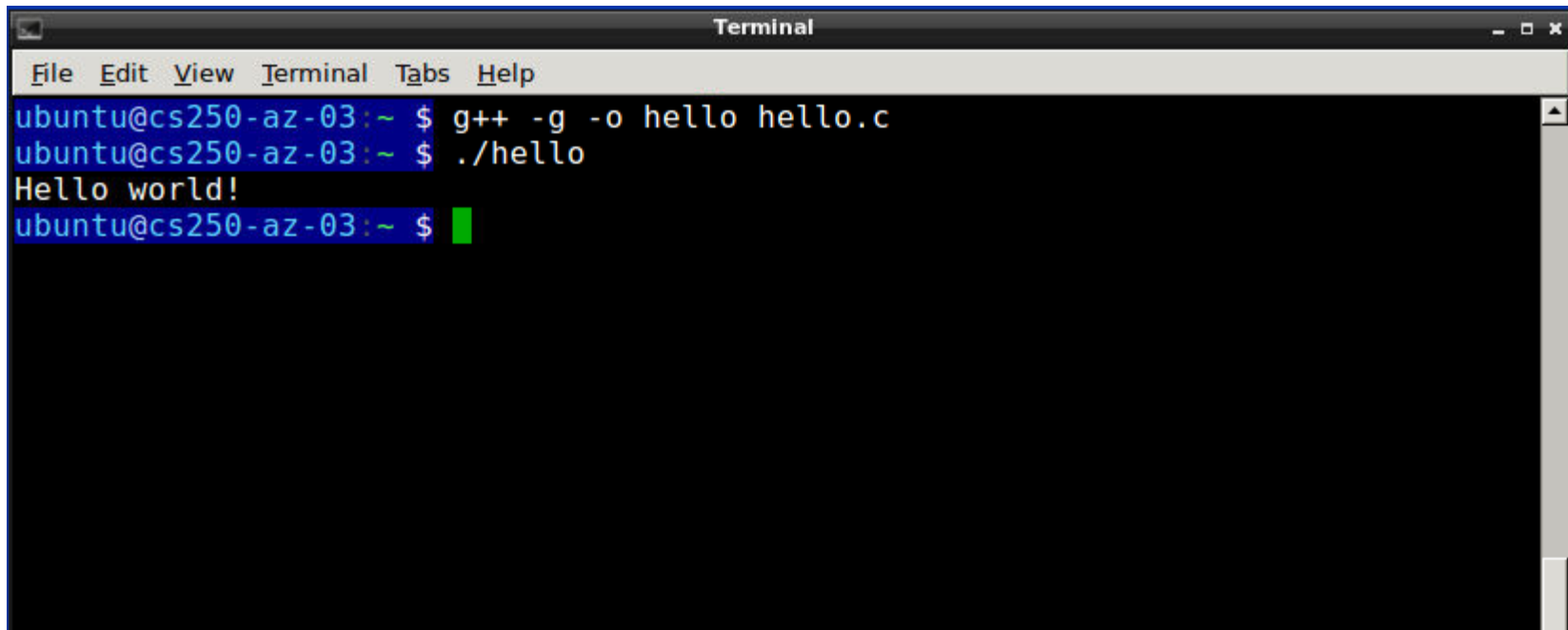
# The vscode window

- 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



```
Terminal
File Edit View Terminal Tabs Help
ubuntu@cs250-az-03:~ $ g++ -g -o hello hello.c
ubuntu@cs250-az-03:~ $ ./hello
Hello world!
ubuntu@cs250-az-03:~ $
```

# 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

```
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



- **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()`

```
#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

# 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

```
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

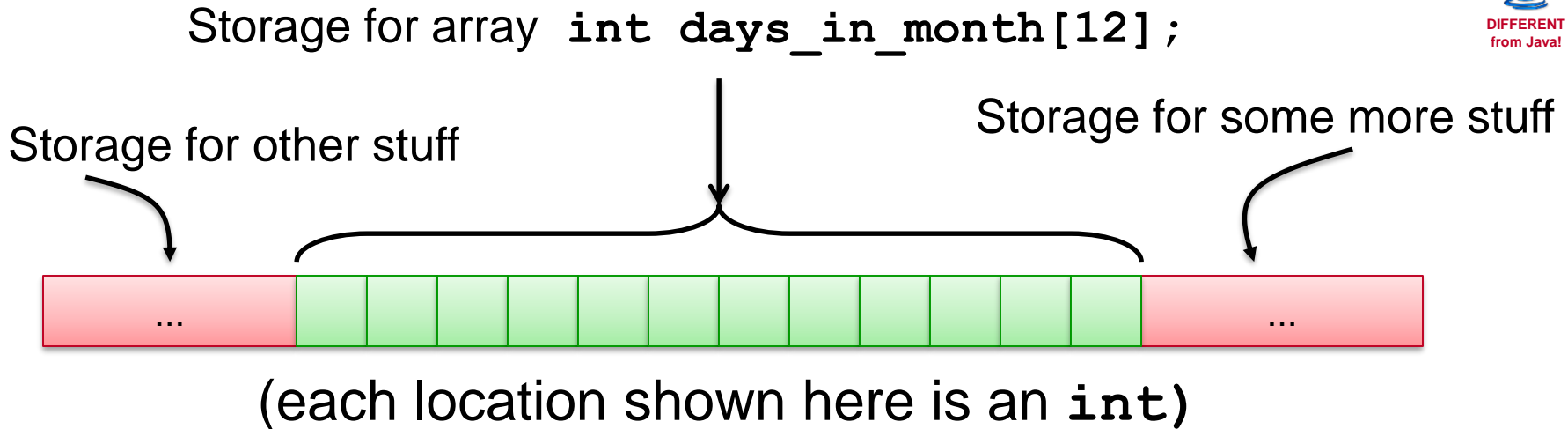
## Same as Java (for now...)



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

for (int 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[219]` or `days_in_month[-35]` !
  - who knows what is stored there?

# Strings – not quite like 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

- 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

```
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



```
#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>
```

"Standard IO"

Not "studio"!!

- ...and it doesn't hurt to also have

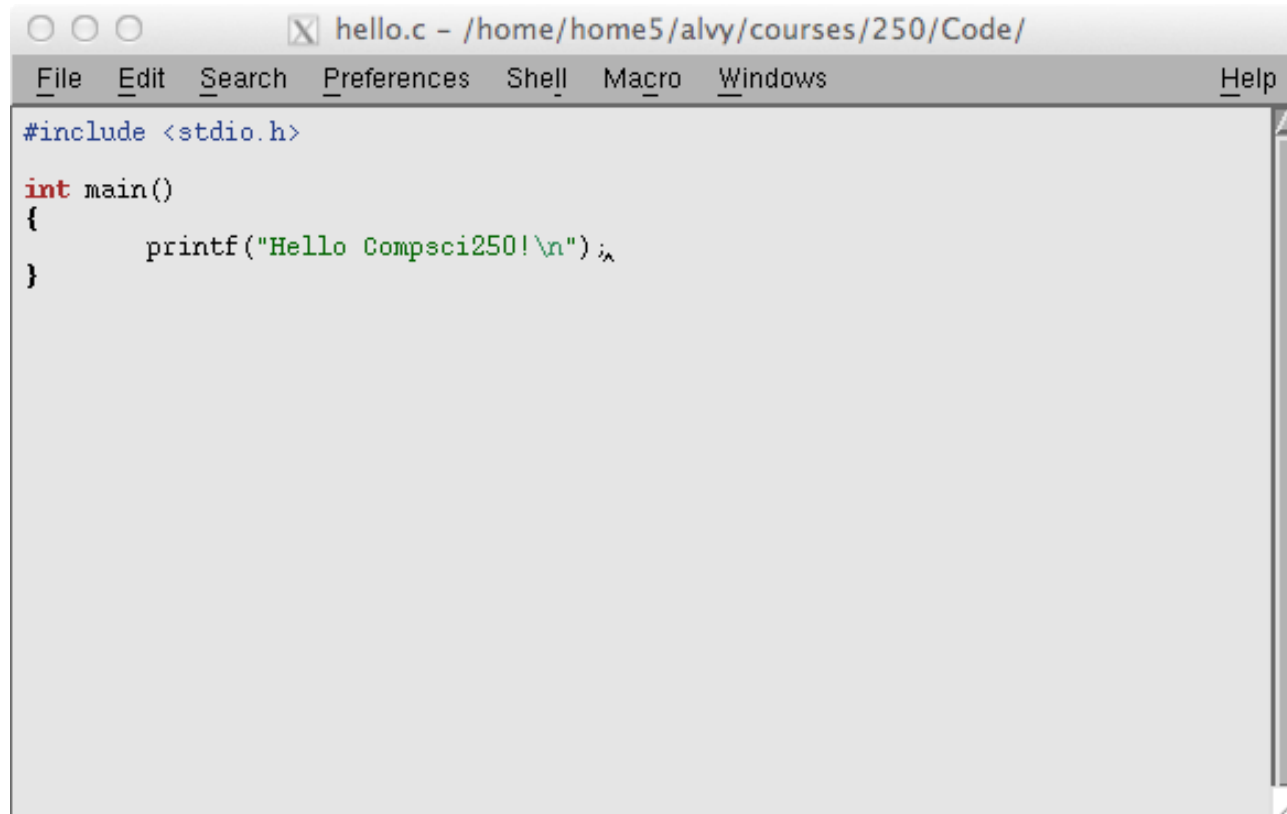
```
#include <stdlib.h>
```

"Standard library"

- *These are **preprocessor** statements; the .h files define function types, parameters, and constants from the standard library*

# 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

A screenshot of a code editor window titled "hello.c - /home/home5/alvy/courses/250/Code/". The window has a menu bar with "File", "Edit", "Search", "Preferences", "Shell", "Macro", "Windows", and "Help". The code content is as follows:

```
#include <stdio.h>

int main()
{
    printf("Hello Compsci250!\n");
}
```

# 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

```
#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)



# 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);
}
```

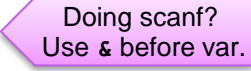
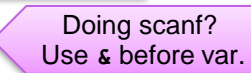
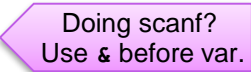
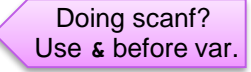
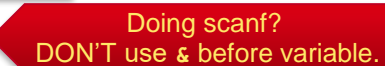
printf() = **print** formatted  
scanf() = **scan** (read) formatted

- 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.", name, age);
```

- Format specifiers:

- **%d** Decimal integer (char/short/int/long/long long) 
- **%x** Hexadecimal integer (char/short/int/long/long long) 
- **%f** Float (float or double) 
- **%c** Character (char) 
- **%s** String (char[] or char\*) 

- Modifying them:

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

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

# Example: Reading Input in a Loop



```
#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)



```
#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](http://die.net).



# sscanf vs. atoi

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



```
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



- 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`
  - `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

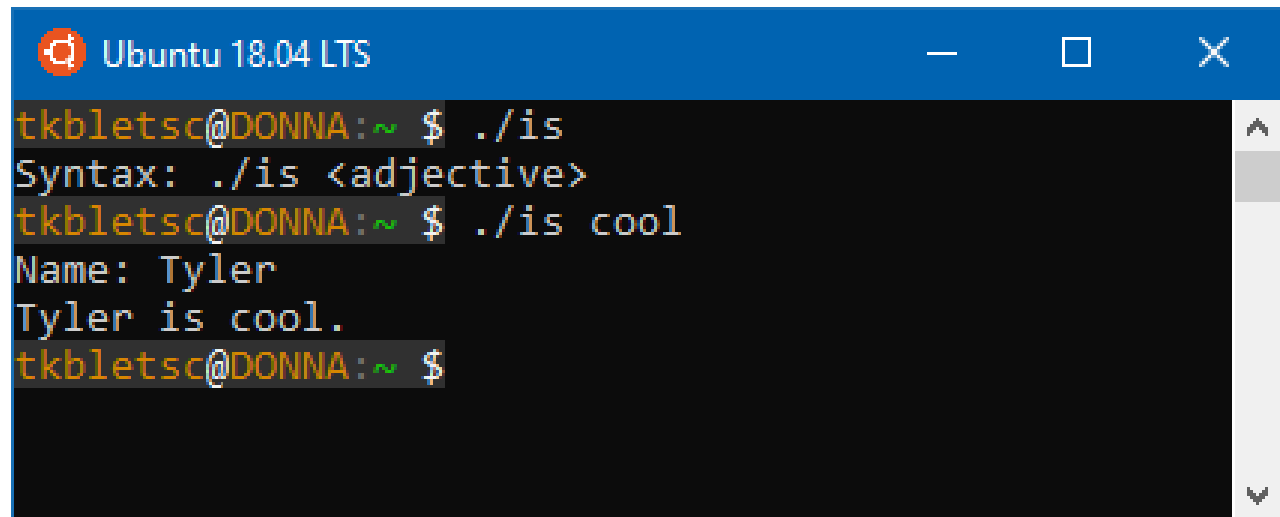
```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Syntax: ./is <adjective>\n");
        return 0;
    }
    printf("Name: ");
    char name[64];
    scanf("%s", name);
    printf("%s is %s.\n", name, argv[1]);
    return 0;
}
```

## 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

A terminal window titled "Ubuntu 18.04 LTS" showing the execution of the program. The user enters './is' and receives the syntax message. Then the user enters './is cool' and the program prints 'Name: Tyler' and 'Tyler is cool.' before returning to the prompt.

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

# Also: DO ERROR CHECKING!

```
#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?

```
Ubuntu 18.04 LTS
tkblets@DONNA:~$ ./is
Name: Tyler
Segmentation fault (core dumped)
tkblets@DONNA:~$
```

oh no now im confused and angry :-)

# The Big Differences Between C and Java

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

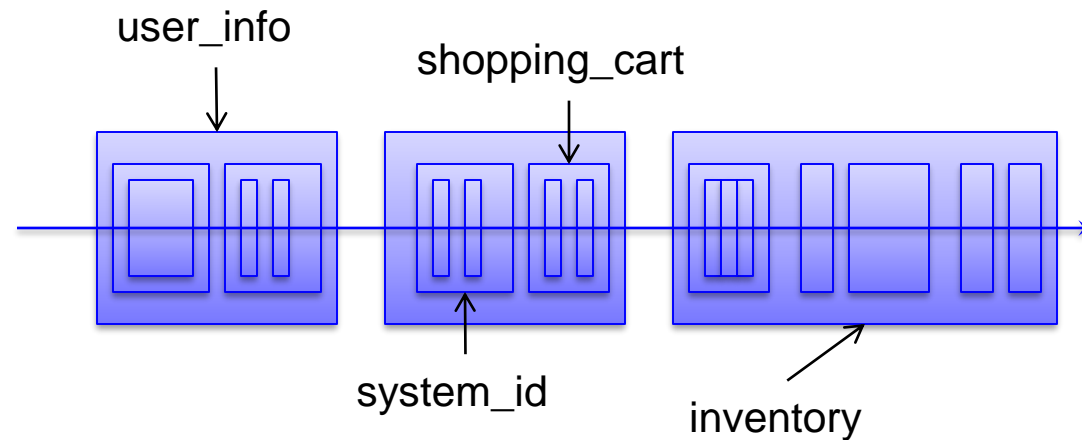
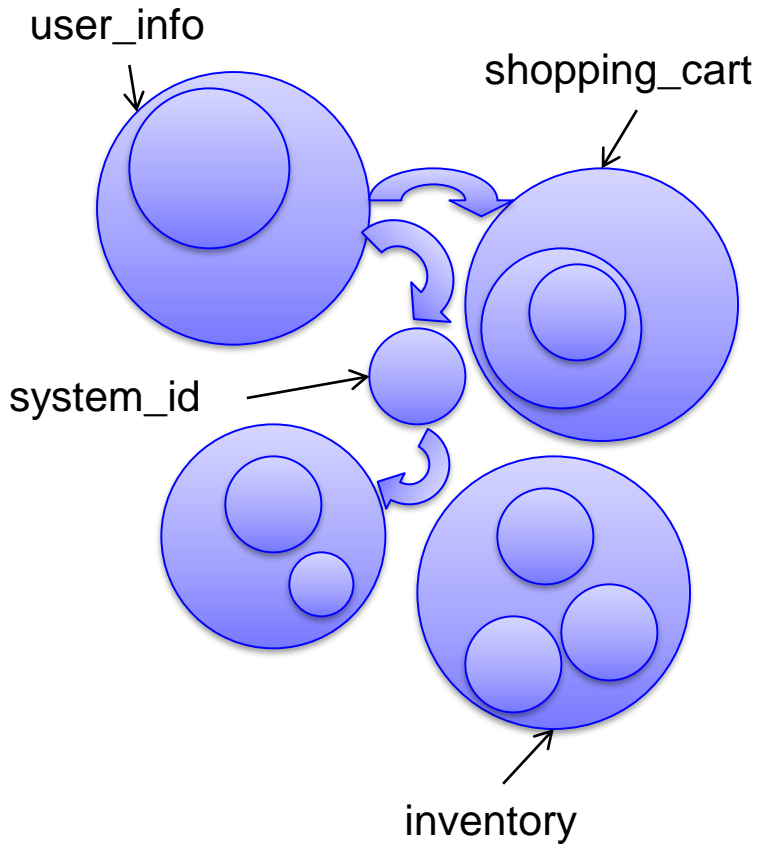
2) Memory management

- 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



# Let's look at memory addresses!

- You can find the address of ANY variable with:



The address-of operator

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

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



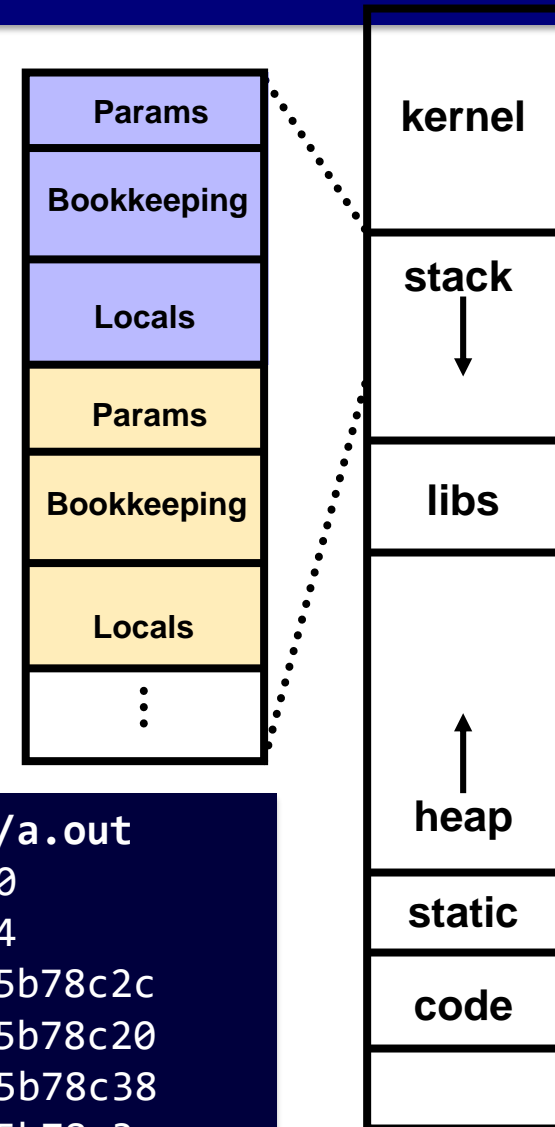


# Testing where variables live

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

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

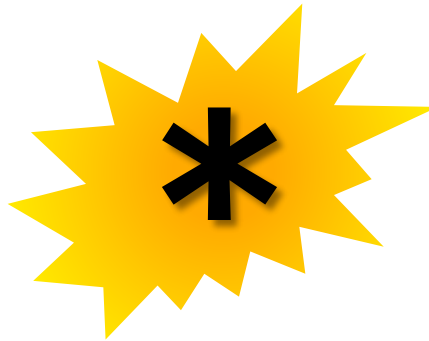
    printf("&x:      %p\n",&x);
    printf("&msg:    %p\n",&msg);
    printf("&argc:   %p\n",&argc);
    printf("&argv:   %p\n",&argv);
    printf("&v:      %p\n",&v);
    printf("&pi:    %p\n",&pi);
}
```



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

# What's a pointer?

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



The *dereference* operator

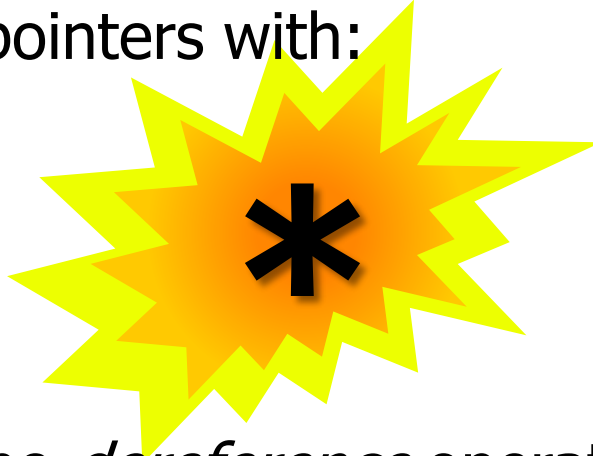
```
int v = 5;  
int* p = &v;  
printf(“%d\n”, v);  
printf(“%p\n”, p);
```

Append to any data type

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

# What's a pointer?

- You can look up 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  
0x7ffffe0e60b7c  
5
```

# Different types use different amounts of memory

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

- Result:

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

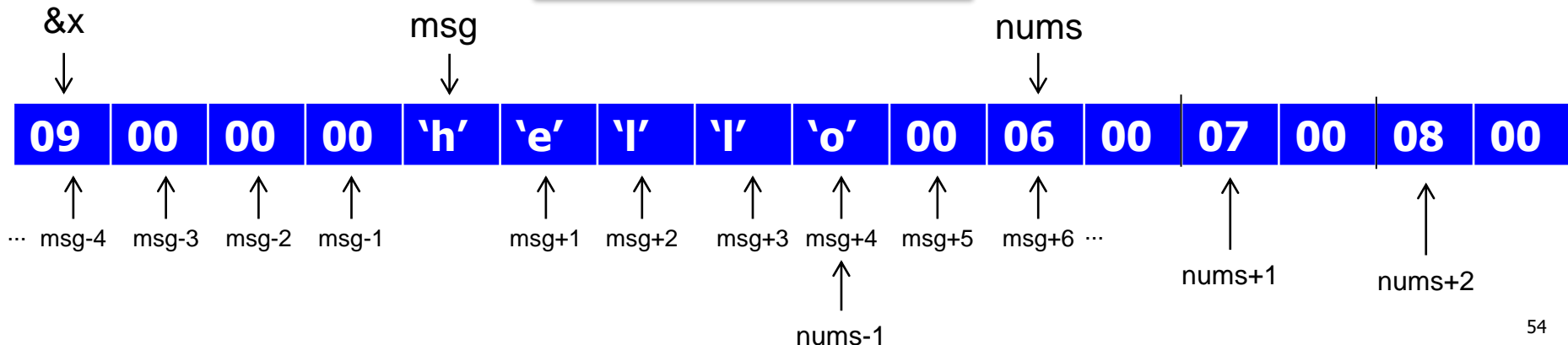
Size in bits	Size in bytes	Datatype	Unsigned range	Signed range
8	1	char	0 .. 255	-128 .. 127
16	2	short	0 .. 65,535	-32,768 .. 32,767
32	4	int	0 .. 4,294,967,295	-2,147,483,648 .. 2,147,483,647
64	8	long long	0 .. 18,446,744,073,709,600,000	-9,223,372,036,854,780,000 .. 9,223,372,036,854,780,000

- 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 *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};
```

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



- Start with an address of something that exists
- Manipulate according to known rules
- 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>
```

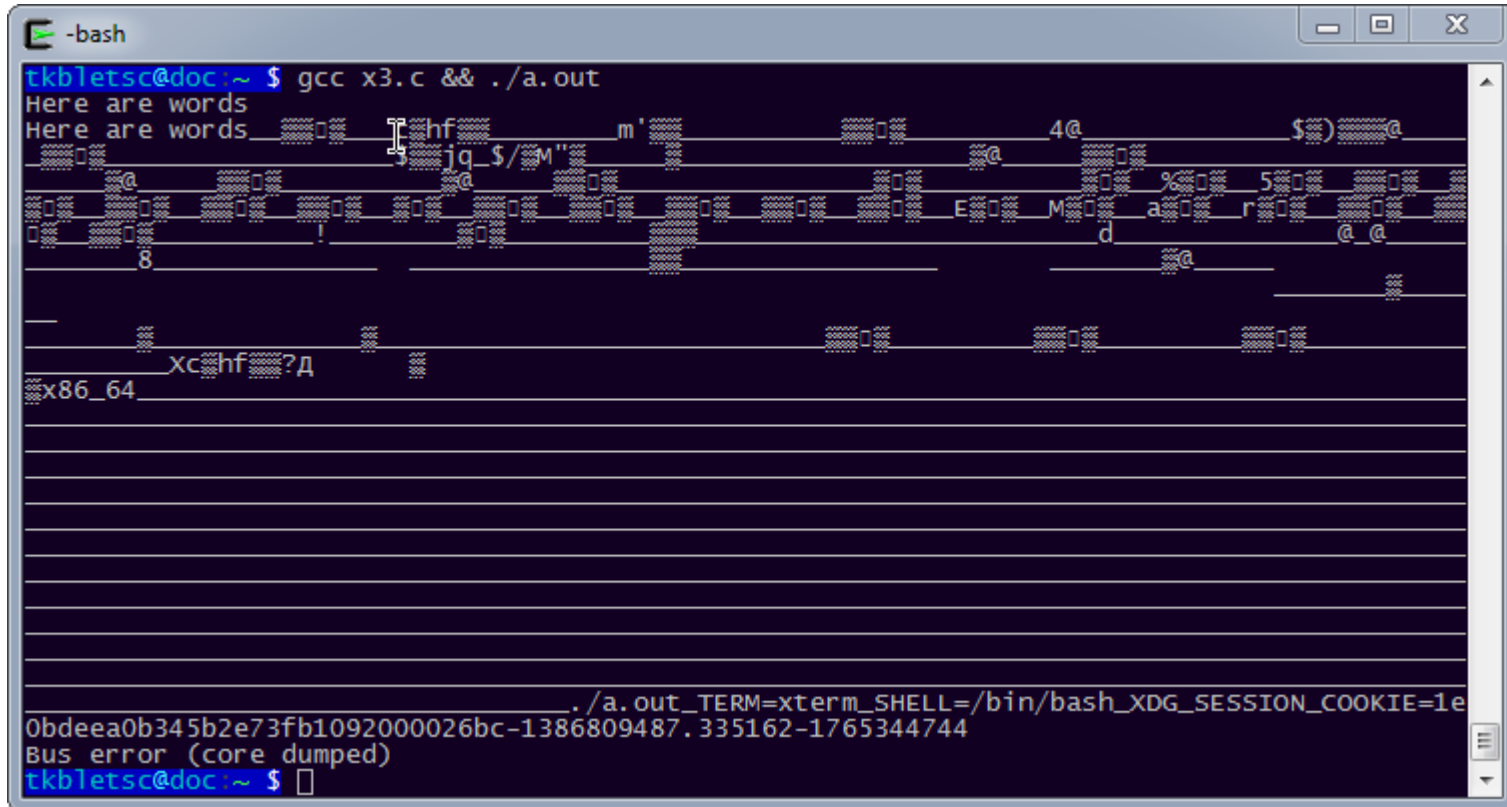
```
int main(int argc, const char* argv[]) {  
    int* p = 100000;  
  
    printf(" p: %p\n",p);  
    printf("*p: %d\n",*p);  
}
```

```
$ 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);  
}
```

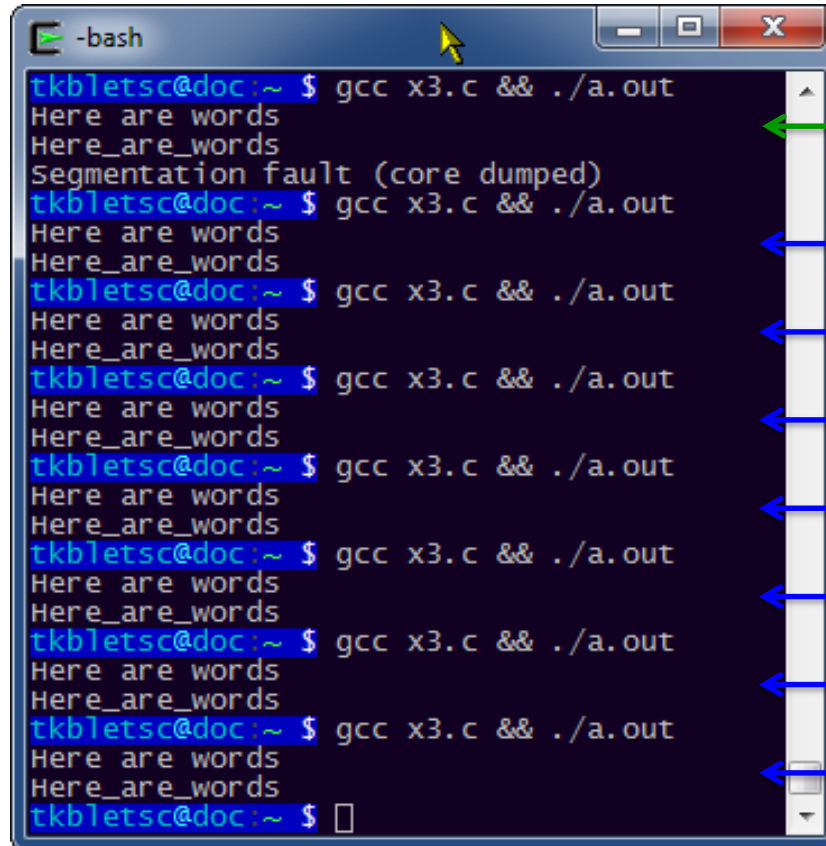


```
-bash  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here are words  
x86_64  
./a.out_TERM=xterm_SHELL=/bin/bash_XDG_SESSION_COOKIE=1e  
0bdeea0b345b2e73fb1092000026bc-1386809487.335162-1765344744  
Bus error (core dumped)  
tkblets@doc:~$
```

# Almost fixed...

```
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);  
}
```



```
-bash  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
Segmentation fault (core dumped)  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$ gcc x3.c && ./a.out  
Here are words  
Here_are_words  
tkblets@doc:~$
```

Worked but crashed on exit

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!



# Pointer summary

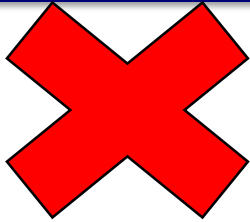
- **Memory is linear**, all the variables live at an address
  - Variable declarations reserve a range of memory space
- You can get the address of any variable with the **address-of operator &**

```
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 `ints` 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);
}
```

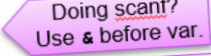
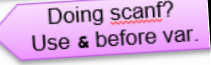
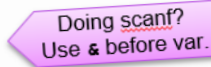
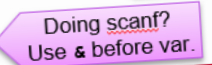

# About “About printf and scanf”

- 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!**

## About printf and scanf

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

### • Format specifiers:

- **%d** Decimal integer (char/short/int/long/long long) 
- **%x** Hexadecimal integer (char/short/int/long/long long) 
- **%f** Float (float or double) 
- **%c** Character (char) 
- **%s** String (char[] or char\*) 



# C Memory Allocation: introducing the heap



- 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



- 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



- **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  $\text{num} * \text{sz}$  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
```

# Example: Linked List

```
#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);
}
```

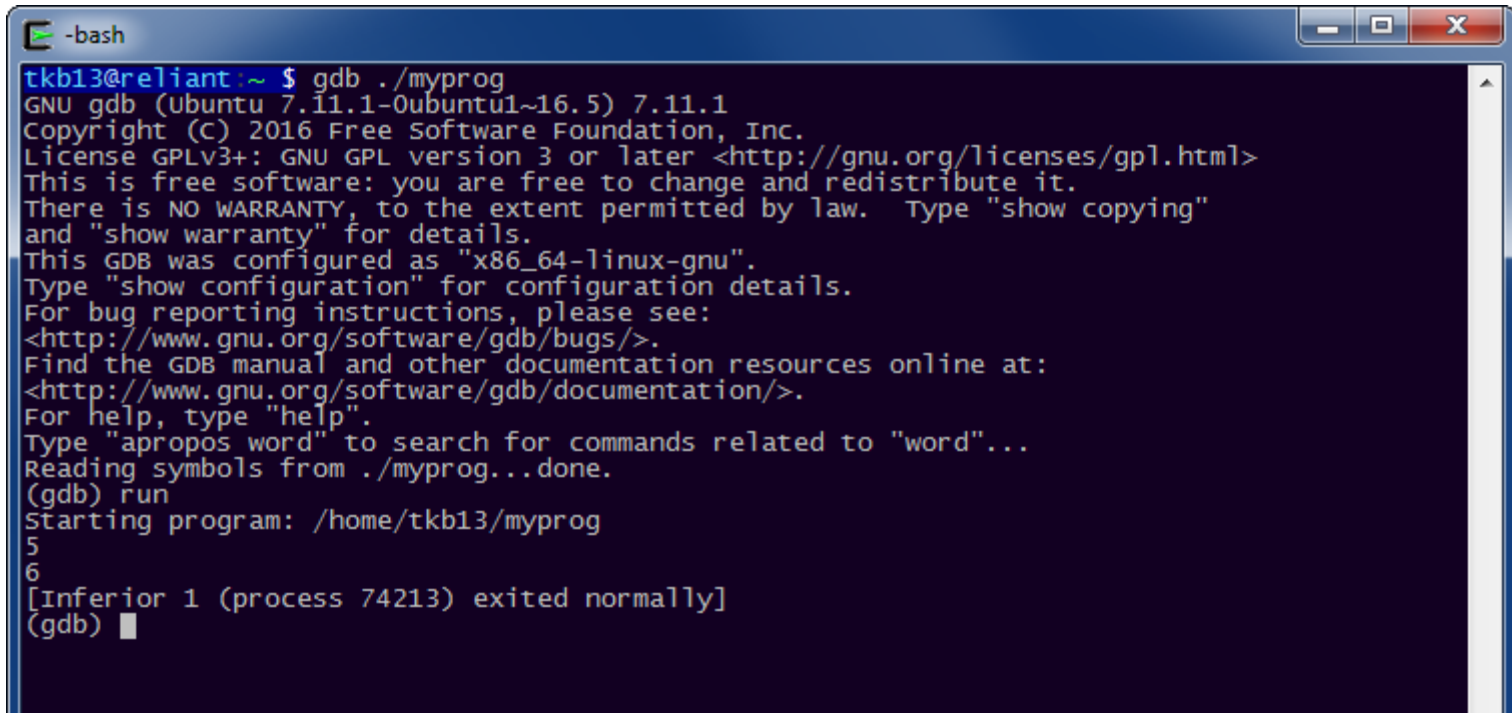
# Source Level Debugging

- 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***



```
-bash
tkb13@reliant:~$ gdb ./myprog
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 ./myprog...done.
(gdb) run
Starting program: /home/tkb13/myprog
5
6
[Inferior 1 (process 74213) exited normally]
(gdb) █
```

# **gdb commands**

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

# **gdb commands**

<code>break &lt;line&gt;</code> <code>break &lt;function&gt;</code> <code>break &lt;line&gt; if &lt;cond&gt;</code>	set breakpoints (including conditional breakpoints)
<code>info breakpoints</code> <code>delete breakpoint &lt;n&gt;</code>	list, and delete, breakpoints
<code>set &lt;var&gt; &lt;expr&gt;</code>	set variable to a value
<code>backtrace full</code> <code>bt</code>	show the call stack & args arguments and local variables



# 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:~  
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:~ $
```

# 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?