ECE 250 / CS 250
Computer Architecture

C Programming

Daniel Sorin

Some slides based on those from Alvin Lebeck, Benjamin Lee, Andrew Hilton, Amir Roth, Gershon Kedem

Outline

- Previously:
  - Computer is a machine that does what we tell it to do
- Now:
  - How do we tell computers what to do? Software!
    - » From high-level language to what machine actually runs
    - » A brief intro to C (and how it differs from Java)
- Next
  - How do we get from C code to bits (1s and 0s)?
    - » How do we represent instructions in bits?
    - » How do we represent data in bits?

Software: We Use High Level Languages

High Level Language Program

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

- There are many high level languages (HLLs)
  - Java, C, C++, C#, CUDA, Fortran, Basic, Matlab, Python, etc.
- HLLs tend to be English-like languages that are "easy" for programmers to understand
- In this class, C is our running example for HLL code. Why?
  - C has pointers (will explain much more later)
  - C has explicit memory allocation/deallocation
  - Java hides these issues (don't get me started on Matlab)

Software: We Use High Level Languages

A Point Worth Re-emphasizing

We're not learning C in order to become expert C programmers.

We're learning C because you can't understand how a computer works until you understand how software runs on the computer …

…. and with C we can see that mapping from the HLL to what the computer hardware actually does.

(And we can't do that with Java or python, unfortunately.)

- In this class, we'll focus on C as our running example for HLL code. Why?
  - C has pointers (will explain much more later)
  - C has explicit memory allocation/deallocation
  - Java hides these issues (don't get me started on Matlab)

HLL → Assembly Language

High Level Language Program

Compiler

Assembly Language Program

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

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

- Every comp. architecture has its own assembly language
- Assembly languages tend to be pretty low-level, yet some actual humans (not just you) still write code in assembly
- But most code is written in HLLs and compiled
  - Compiler is program that automatically transforms HLL to assembly

Assembly Language → Machine Language

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

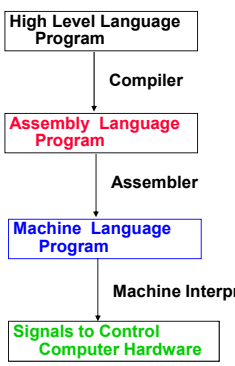
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
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

Machine Language → Hardware Control Signals




```
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

You Know (at Least) One HLL: Java

Example Java code

```
...
System.out.println("Please Enter In Your First Name: ");
String firstName = bufRead.readLine();
System.out.println("Please Enter In The Year You Were Born: ");
String bornYear = bufRead.readLine();
System.out.println("Please Enter In The Current Year: ");
String thisYear = bufRead.readLine();
int bYear = Integer.parseInt(bornYear);
int tYear = Integer.parseInt(thisYear);
int age = tYear - bYear ;
System.out.println("Hello " + firstName + ". You are " + age + " years
old");
```


Q: How Does a Java Program Run? (A: slowly)

- Compile Java source code to Java bytecode
  - Bytecode is like machine code, except portable across architectures
- Java Virtual Machine (JVM) interprets/translates bytecode
  - 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
  - Checks array bounds, does garbage collection, etc.
- Key feature: JVM handles memory for you
  - What do you do when you remove entry from hash table, binary tree, etc.?
- JVM's features are nice but "hidden" from you
  - To design computer, want to see everything

Outline

- Previously:
  - Computer is a machine that does what we tell it to do
- Now:
  - How do we tell computers what to do? Software!
    - » From high-level language to what machine actually runs
    - » A brief intro to C (and how it differs from Java)
- Next
  - How do we represent data?
  - What is memory?

But First, An Important Reminder

- You already know how to program
  1. Think about problem to solve
  2. Plan out how code should work
  3. Write code
  4. Debug code
- None of that changes in C or any other language!
  - Don't psych yourself out because the language is a bit different

The C Programming Language

- C: Like Java, but with much less hidden from you
- No virtual machine
  - No dynamic type checking, array bounds, garbage collection, etc.
  - Compile source file (e.g., hello.c) directly to executable program
- "Closer" to hardware
  - Easier to make mistakes
  - Can often result in faster code → training wheels slow you down
- Often used for 'systems programming'
  - Operating systems, embedded systems, databases, etc.
  - C++ is object-oriented version of C (C is strict subset of C++)

Learning How to Program in C

- You need to learn some C
- I'll present some slides next, but nobody has ever learned programming by looking at slides or a book
  - You learn programming by programming!
- Goals of these slides:
  - Give you big picture of how C differs from Java
    - » Recall (again!): you already know how to program
  - Give you some important pointers (forgive the awful pun!) to get you started

Skills You'll Need to Code in C

- You'll need to learn some skills
  - Using a Unix machine (you'll connect remotely to virtual one)
  - Using a text editor to write C programs
  - Compiling and executing C programs
- You'll learn these skills in Recitation #1
  - Use this opportunity to make HW#1 much, much easier
- Some other optional, useful resources
  - Kernighan & Richie book *The C Programming Language*
  - MIT open course *Practical Programming in C* (linked off webpage)
  - Prof. Drew Hilton's video tutorials (linked off webpage)

Key Language Issues: C vs Java

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

Key
Black: C same as Java
Blue: C very similar to Java
Red: C different from Java

Variables, Operators, and Expressions

Same as Java!

- Variables types (not exhaustive list)
  - Data types: int, float, double, char, void
  - Signed and unsigned int
- Operators
  - Mathematical +, -, *, /, %,
  - Logical !, &&, ||, ==, !=, <, >, <=, >=
  - Bitwise &, |, ~, ^ , <<, >> (we'll get to what these do later)
- Expressions: var1 = var2 + var3;

Control Flow

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) {…}
```


Comments

Similar to Java

- Java: everything from // to end of line
- C: everything between /* and */
  - Can go past line breaks

```
main() {   /* hi class! */
   int a;
/* This is where the cool code goes */
   a = 7; /* hmm, not terribly cool … */
}
```

NOTE: I will often use // as shorthand, even though not correct for C

Functions

Similar to Java

- C has functions, just like Java
  - But these are not methods! (i.e., they're not attached to objects)
- Must be declared before use (but can define later)

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

- Or can put functions at top of file (doesn't always work)

Arrays

Same as Java for now… some fun diffs later

```
char buf[256];
int grid[256][512];  /* two dimensional array */
float scores[4196];
double speed[100];

for (i = 0; i<= 25; i++){
   grid[i] = i*i-3;
}
```


Strings – not quite like Java

- String is array of chars, terminated with NULL char

```
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> // library with string ops
strlen(str1);
```


The Two Big Differences Between C and Java

1) Java is object-oriented, while C is not
   - This is not a big deal for purposes of ECE/CS 250 → not why we need to teach you C in this class
2) C makes memory visible
   - This is why we teach you C in ECE/CS 250

   All variables live in memory (much more on this later!)
- In Java: virtual machine worries about where variables "live" and how to allocate memory for them
- In C: programmer does all of this

Everything else is approximately the same!
Yes, there are differences, but they're minor

Diff #1: C Is Not Object-Oriented

- C is not object-oriented
- C **structure** is sort of like Java object
  - Structure has member variables but NO methods
- Structure definition with `struct` keyword

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

- Declare variable of structure type with `struct` keyword

```
struct student_record onerec;
```


Can Have Array of Structs

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

struct student_record myroster[100]; /* array of structs */
int main()
{
      myroster[23].id = 99;
      myroster[23].grade = 88.5;
}
```

Diff #2: C Makes Memory Visible

- Everything in program (instructions & variables) lives in memory
- Each program has memory: giant array of bytes that it uses to hold what it needs
  - 1 byte = 8 bits
  - 1 bit = something that can be either 0 or 1
- **True for all programming languages**, but Java hides this from programmer
  - C makes memory visible

A View of Memory

- Giant array of bytes, each with own unique address
  - Number of bytes = 2^{32} (on 32-bit machine) or 2^{64} on 64-bit

Address	Byte of Data at Address
2^{64}-1	01000101
2^{64}-2	11001100
1000	01011100
3	00000000
2	00001010
1	11111111
0	01110101

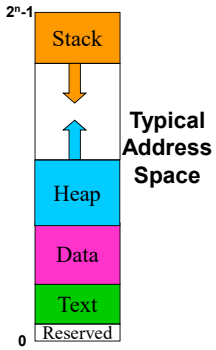
How Much Memory Do Variables Use?

- Everything in memory takes up some number of bytes
- Examples
  - char = 1 byte (1B)
  - int = 4B
  - float = 4B
  - double = 8B
  - MIPS instruction = 4B
  - x86 instruction: depends on which instruction
  - array: depends on array type and number of entries
  - struct: depends on size of struct

Once again, all of this is also true in Java

Memory Layout

- Memory is array of bytes, but there are conventions as to what goes where in this array
- Text: instructions (the program to execute)
- Data: **global** variables (declared outside any function = visible to all)
- Stack: **local** variables and other per-function state; starts at top & grows down
- Heap: **dynamically allocated** variables; grows up
- What if stack and heap overlap????

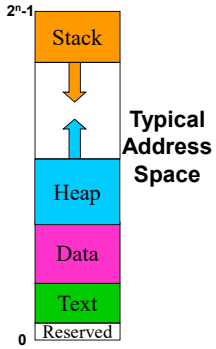


Memory Layout: Example

```
int anumber = 3;

int factorial (int x) {
  if (x == 0) {
    return 1;
  }
  else {
    return x * factorial (x - 1);
  }
}

int main (void) {
  int z = factorial (anumber);
  printf("%d\n", z); // print to screen
  return 0;
}
```



How Does C Make Memory Visible?

- In C, programmer can know memory address of variable
  - For now, trust me that this will be useful/terrifying
- Variable that holds address of another variable is called **pointer**
- Size of pointer is therefore same size as address
  - 32 bits = 4 bytes (4B) on 32-bit machine
  - 64 bits = 8B on 64-bit machine
- Java has references that are sort of related to pointers, but they're not the same

Pointers Are Basic Variable Types

- A pointer is a variable type, like an int or char
- But many flavors of pointers based on what's being pointed to
- Examples:
  - `int* x_ptr;  // x_ptr is pointer to int`
  - `char* y_ptr;  // y_ptr is pointer to char`
  - Etc.
- Pointers are like any other variables
  - Types must match
    - » Can't assign char* to int*
    - » Can't assign int to int*
  - Pointers live in memory (i.e., have addresses)

Where Do Pointers Come From?

- C provides "&" operator to get address of variable

```
char x='a';
char* x_ptr;
x_ptr = &x;
// x_ptr equals addr of x
```

- Assume x is at address 100
- Assume x_ptr is at address 253410 (takes 8B)

Address	Data	
2^{64}-1		
2^{64}-2		
253417	00000000	} x_ptr
	00000000	
253411	00000000	
253410	01100100 = 100_{10}	
100	01100001 = 'a'	}x
2		
1		
0		

Same Thing, But With int* Instead of char*

- C provides "&" operator to get address of variable

```
int x=3;
int* x_ptr;
x_ptr = &x;
// x_ptr equals addr of x
```

- Assume x is at address 100
- Assume x_ptr is at address 253410

Address	Data	
2^{64}-1		
2^{64}-2		
253417	00000000	} x_ptr
	00000000	
253411	00000000	
253410	01100100 = 100_{10}	
103	00000000	} x
102	00000000	
101	00000000	
100	00000011 = 3_{10}	
2		
1		
0		

Dereferencing Pointers

- Programmer can access variable through pointer to it
  - Called **dereferencing**
  - Uses "*" operator in front of pointer name (but not in declaration)

```
int x;      // declares x is an int
int* x_ptr; // declares x_ptr is an int* (ptr to int)
```

- Two equivalent ways to set x equal to 3

```
x = 3;
*x_ptr = 3;  // dereferencing x_ptr
```

- Once again, trust me for now that this will be useful

Small Syntax Issue: Pointers to Structs

```
struct student_rec {
      int id;
      float grade;
};
student_rec rec1;    // rec1 is student_rec struct
student_rec* my_ptr; // ptr to student_rec struct
my_ptr = &rec1;
```

To access members of this struct via the pointer my_ptr:

```
      my_ptr->id = 3;        // not my_ptr.id
      my_ptr->grade = 2.3;   // not my_ptr.grade
```


Memory Allocation of Vars: Static vs. Dynamic

- Variables are either declared statically or dynamically
  - **IMPORTANT: pay attention here**
- Static declaration
  - When you know exactly what you need before running program

```
int x;            // need only one int
float y;          // need only one float
char myname[100]; // need only 100 chars
```

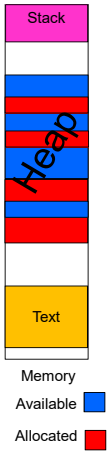
  - Static vars generally live on stack (or, in C, can be global)
- Dynamic declaration
  - When you do NOT know exactly what you need before running
    - » e.g., adding a new element to linked list
  - In Java, you used "new()" for this purpose
  - In C, you will use "malloc()" for this purpose

Dynamic Allocation of Memory for Variables

- Important to know when dynamic allocation is needed
- Consider list of structs (e.g., struct record)
- Consider list with fixed number of entries
  - `struct record[100];  // no need for dynamic allocation`
- Consider list with number of entries that depends on input to program
  - Can't use array, so what data structure do you use? Linked list!
  - Can dynamically allocate each entry in linked list (if each entry gets added in response to a new input)
  - Or can dynamically allocate all entries at once (if you find out at a given time exactly how many entries there will be)
  - Note: C compiler permits array with variable number of entries, but I disallow this (e.g., `struct record [numElements];`)

Using C's Heap Manager

- C has heap manager to manage heap for you
  - Recall: heap is dynamically allocated memory
  - Keeps track of used and free memory in heap
- You allocate/deallocate memory on heap
  - Use malloc() to allocate, e.g., new element for linked list
  - Use free() to deallocate previously allocated memory
  - Details on malloc() and free() on next slide
- Input to malloc(): how many bytes do you want?
  - Neat trick: C has sizeof() function to help you
  - e.g., sizeof(int) or sizeof(struct my_student), etc.
- malloc() returns pointer to first byte of allocation
  - Returns pointer of type void* → yuck
  - Can cast it to another type (more on next slides)



Let's Use malloc()

- Dynamically allocate space for 300 ints (=1200 bytes)

```
int* x_ptr; // x_ptr is a var of type int*
x_ptr =(int*)malloc(300*sizeof(int));
```

- C heap manager decides to allocate memory from 738124 to 739323
- x_ptr points to first byte in this allocation
- Could access first int (first 4 bytes) with *x_ptr

Address	Data
2^{64}-1	
739323	
738125	
738124	
329	
323	
322	
1	
0	

738124

1200 bytes (not to scale!)

x_ptr=64bit=8B long

C Memory Deallocation

- In Java, programmer never deallocates memory
  - Call new() to allocate
  - Let Java's garbage collector (GC) reclaim unused allocations
- In C, programmer must deallocate memory to let heap manager know it can be reallocated
- `free(ptr)`
  - ptr must be a value previously returned from malloc()

Pointers ←→ Arrays

- Pointers and arrays are VERY related in C
- Array → pointer:
  When you declare array, you're also declaring pointer to beginning of array

```
int numbers[100]; // numbers is var of type int*
int* num_ptr = numbers; // this is ok, types match
*numbers = 2;  // same as: numbers[0]=2
```

- Pointer → array:
  When you malloc() and get pointer, it's also an array

```
int* num_ptr = (int*)malloc(100*sizeof(int));
num_ptr[0] = 2;  // same as: *num_ptr = 2
num_ptr[1] = 33; // same as: *(num_ptr+1)=33
```


Pointer Arithmetic

- We can perform integer arithmetic on pointers
  - Yes, I know this looks like type mismatch, but it's OK

```
int x[20];  // recall that x is var of type int*
int* x_ptr = x; // same as: x_ptr = &x[0]
x[0]=7;
*(x+1)=42;
x_ptr = x_ptr + 1;   // yup, that's legal
// so now what address is x_ptr pointing to?
// what is x_ptr[0]?  how about x[0]?  Next slide …
```

- This is not something you can do with Java references
  - My conclusion: C is more fun/dangerous than Java

Arrays, Pointers, and Address Calculation

- x is a pointer, what is x+33?
- It's a pointer, but where?
  - What does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
  - One reason why we tell compiler what type of pointer we have, even though all pointers are really the same thing (and same size)

```
int* x=malloc(100*sizeof(int));
```

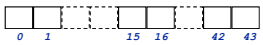

```
x[33] is the same as *(x+33)
If x is 160, then x+1 is
164, x+2 is 168
```

```
double* d=malloc(200*sizeof(double));
```


```
*(d+33) is the same as d[33]
if d is 176, then d+1 is
184, d+2 is 192
```


More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only
- what's at `*(begin+44)`?
- what does `begin++` mean?
- how are pointers compared using < and using == ?
- what is value of `end - begin`?

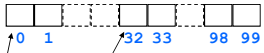



```
char* a = (char*)malloc(44);
char* begin = a;
char* end = a + 44;

while (begin < end)
{
   *begin = 'z';
   begin++;
}
```


More Pointers & Arrays

```
int* a = (int*)malloc(100*sizeof(int));
```




```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
```


Array Example

```
#include <stdio.h>

main()
{
  int* a = (int*)malloc (100*sizeof(int));
  int* p = a;
  int k;

  for (k = 0; k < 100; k++)
    {
      *p = k;
      p++;
    }
  printf("entry 3 = %d\n", a[3])
}
```


Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

swap	
x	42
y	100
temp	???
RA	c0

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

swap	
x	42
y	100
temp	**42**
RA	c0

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

swap	
x	**100**
y	100
temp	42
RA	c0

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
c0  swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

swap	
x	100
y	**42**
temp	42
RA	c0

- What does this print? Why?

Let's do a little Java…

```
public class Example {
  public static void swap (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  public static void main (String[] args) {
    int a = 42;
    int b = 100;
    swap (a, b);
    System.out.println("a =" + a + " b = " + b);
  }
}
```

Stack

main	
a	42
b	100

- What does this print? Why?

Let's do some different Java…

```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```

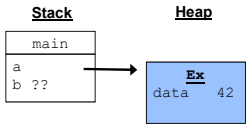
Stack

main	
a	??
b	??

- What does this print? Why?

Let's do some different Java…

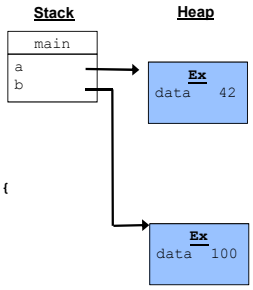
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

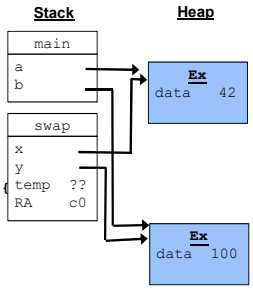
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

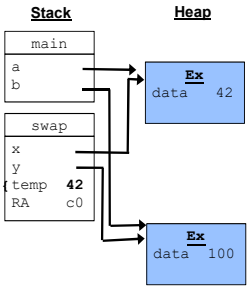
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

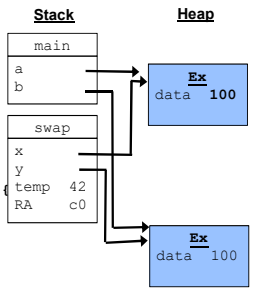
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

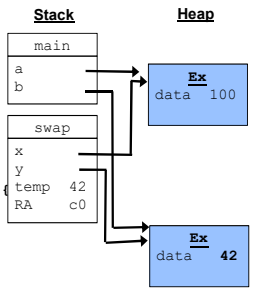
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

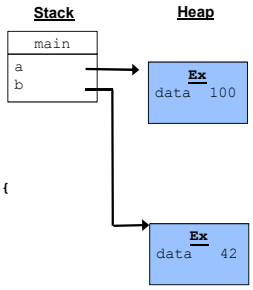
```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Let's do some different Java…

```
public class Ex {
  int data;
  public Ex (int d) { data = d; }
  public static void swap (Ex x, Ex y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
  }
  public static void main (String[] args) {
    Ex a = new Ex (42);
    Ex b = new Ex (100);
    swap (a, b);
    System.out.println("a =" + a.data +
                       " b = " + b.data);
  }
}
```



- What does this print? Why?

Pass by Value vs. Pass by Reference

```
void swap (int x, int y){
  int temp = x;
  x =
  y
}
main()
  int a
  int
  sw
  p
%d\n",  , b);
}
```

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


Working With Strings, Pointers, & Arrays

- Common mistake: pointer doesn't allocate space

```
char* name;  // space for char* on stack
name[0] = 'D';  // where is name[0]?
```

  - name is just pointer, no space for chars

- Here's what you wanted to do

```
char name[100];  // space for 100 chars on stack
name[0] = 'D';
```


Working With Strings, Pointers, & Arrays

- Common mistake in copying strings

```
char text[100];  // assume text holds "MIPS is fun\n"
char* otherText;
otherText=text; // copies pointer but not string
```

  - otherText is just pointer, doesn't have space for chars

- Here's what you wanted to do

```
char text[100] = "MIPS is fun\n";
char otherText[100];
strcpy(otherText, text); // from string.h
```

  - Instead of strcpy, could write loop to copy chars from name to otherName

Some Other Useful Stuff for C Programmers

- Some C tricks
  - Casting
  - Global variables
- Input/output (I/O)
  - Reading/writing files
  - Reading arguments passed on command line
- Including other code
  - Libraries
  - Other code you've written in other files

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 integer cast of x
        printf("x=%f, i=%d", x, i)
}
result: x=3.600000, i=3
```

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();
        // what is the value of X here?
}
```

- What if we had "`int X = 23;`" in main()?

Back to First Program from Recitation #1

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

Input/Output (I/O)

- Read/Write to/from the terminal
  - Standard input, standard output (defaults are terminal)
- Character I/O
  - `putchar(), getchar()`
- Formatted I/O
  - `printf(), scanf()`

Character I/O

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

- EOF is End Of File (type ^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 look at & 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

Example: Reading Input in a Loop

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

- 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.
- Type "man scanf" on a linux machine and you can read a lot about scanf
  - man = online manual

Header Files, Separate Compilation, Libraries

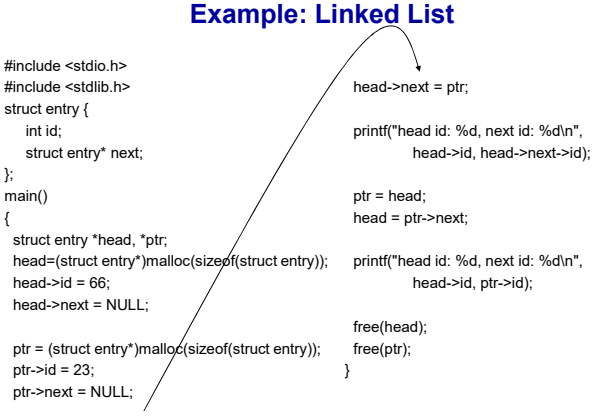
- C pre-processor provides useful features
  - #include filename just inserts that file (like #include <stdio.h>)
  - #define MYFOO 8, replaces MYFOO with 8 in entire program
    - » Good for constants
    - » #define MAX_STUDENTS 100 (functionally equivalent to const int)
- Separate Compilation
  - Many source files (e.g., main.c, students.c, instructors.c, deans.c)
  - gcc –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.)
  - » gcc –o prog file.c –lm (says use math library)
  - » You can read more about this elsewhere

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"

```
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]);
}
```


Example: Linked List



```
#include <stdio.h>
#include <stdlib.h>
struct entry {
    int id;
    struct entry* next;
};
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);
}
```


Summary

- C and Java are similar in many ways
  - Data types
  - Expressions
  - Control flow
- Two very important differences
  - No objects!
  - Explicit memory management
- Up next:
  - So what exactly are those chars, ints, floats?
  - And what exactly is an address?

Outline

- Previously:
  - Computer is machine that does what we tell it to do
- Next:
  - How do we tell computers what to do?
  - How do we represent variables with bits?