Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

Check your frequency:
- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA
   vote status light will indicate success
Today

- Functions & the stack
- Stack data structure, applied to memory
- Behavior of function calls
- Storage of function data, at assembly level
C Pointers Introduction

What is a pointer?
C Pointers Introduction

What is a pointer?

A pointer is like a mailing address, it tells you *where something is located*.

Every object (including simple data types) reside in the *memory* of the machine.

A pointer is an “address” telling you where that variable is located in memory.
Recall: Arrays

```c
int january_temps[31]; // Daily high temps
```

Array variable name means, to the compiler, the **beginning of the memory chunk**. (address)
Recall: Addressing Mode: Memory

movl (%rcx), %rax

- Use the address in register `%rcx` to access memory, store result in register `%rax`

### CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>42</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Index into memory using the address in `%rcx`.

2. Copy value at that address to `%rax`.

```
0x0: 0x8: 0x10: 0x18:
0xA0: ... 0xA60 0xA68 42
0xA70 0xA78 ... 0xFFFF000000:
```
Recall: Program Counter

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

X86_64 refers to the PC as %rip.
Pointers in C

• Like any other variable, must be declared:

• Example:
  – int *mypi;
  – int *mypi;
Recall: Dereferencing a Pointer

• To follow the pointer, we **dereference** it.

• **Dereferencing re-uses the * symbol.**

• If `iptr` is declared as an integer pointer, `*iptr` will follow the address it stores to find an integer in memory.
What will this do?

```c
int main(void) {
    int *ptr;
    printf("%d", *ptr);
}
```

A. Print 0  
B. Print a garbage value  
C. Segmentation fault  
D. Something else
Why Pointers?

• Using pointers seems like a lot of work, and if used incorrectly, things can go wrong.
• Pointers also add a level of “indirection” to retrieve / store a value

• Two main benefits:
  1. “Pass by pointer” function parameters
     • By passing a pointer into a function, the function can dereference it so that the changes persist to the caller.
  2. Dynamic memory allocation
     • A program can allocate memory on demand, as it needs it during execution
Allocating (Heap) Memory

• The standard C library (#include `<stdlib.h>`) includes functions for allocating memory:

```c
void* malloc(size_t size)
    – Allocate size bytes on the heap and return a pointer to the beginning of the memory block. (size_t is an unsigned int of 8 bytes on x86_64)

void free(void *ptr)
    – Release the malloc() ed block of memory starting at ptr back to the system.
```
What do you expect to happen to the 100-byte chunk if we do this?

// What happens to these 100 bytes?

```c
int *ptr = malloc(100);
```

```c
ptr = malloc(2000);
```

A. The 100-byte chunk will be lost.
B. The 100-byte chunk will be automatically freed (garbage collected) by the OS.
C. The 100-byte chunk will be automatically freed (garbage collected) by C.
D. The 100-byte chunk will be the first 100 bytes of the 2000-byte chunk.
E. The 100-byte chunk will be added to the 2000-byte chunk (2100 bytes total).
“Memory Leak”

• Memory that is allocated, and not freed, for which there is no longer a pointer.

• In many languages (Java, Python, ...), this memory will be cleaned up for you.
  – “Garbage collector” finds unreachable memory blocks, frees them.
  – (This can be a time consuming feature)
  – C doesn’t do this for you!
Why doesn’t C do garbage collection?

A. It’s impossible in C.

B. It requires a lot of resources.

C. It might not be safe to do so. (break programs)

D. It hadn’t been invented at the time C was developed.

E. Some other reason.
Memory Bookkeeping

- To free a chunk, you MUST call `free` with the same pointer that `malloc` gave you. (or a copy)

- The standard C library keeps track of the chunks that have been allocated to your program.
  - This is called “metadata” – data about your data.

- Wait, where does it store that information?
  - It’s not like it can use `malloc` to get memory...
```c
int *iptr = malloc(8);
```

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
int *iptr = malloc(8);
int *iptr = malloc(8);

C Library:
“Let me record this allocation’s info here.”
Size of allocation
Maybe other info
```c
int *iptr = malloc(8);
```

There could be another chunk after yours.
```c
int *iptr = malloc(8);
```

Stay within the memory chunks you allocate.

If you corrupt the metadata, you will get weird behavior.
Valgrind is your new best friend! 😊

```c
int *iptr = malloc(8);
```
Pointers as Arrays

“Why did you allocate 8 bytes for an int pointer?

```c
int *iptr = malloc(8);
```
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

The C compiler knows how big an integer is.

As an alternative way of dereferencing, you can use []’s like an array.

The C compiler will jump ahead the right number of bytes, based on the type.
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

```
<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>iptr[0]</td>
</tr>
<tr>
<td>iptr[1]</td>
</tr>
<tr>
<td>iptr[2]</td>
</tr>
<tr>
<td>iptr[3]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
iptr[2] = 7;
```

1. Start from the base of `iptr`. 
Pointers as Arrays

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));

iptr[2] = 7;
```

1. Start from the base of `iptr`.

2. Skip forward by the size of two ints.
Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Pointers as Arrays

• This is one of the most common ways you’ll use pointers:
  – You need to dynamically allocate space for a collection of things (ints, structs, whatever).
  – You don’t know how many at compile time.

```c
float *student_gpas = NULL;
student_gpas = malloc(n_students * sizeof(int));
...
student_gpas[0] = ...
student_gpas[1] = ...
```
Pointer Arithmetic

• Addition and subtraction work on pointers.

• C automatically increments by the size of the type that’s pointed to.
### Pointer Arithmetic

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1st integer</td>
</tr>
<tr>
<td>2nd integer</td>
</tr>
<tr>
<td>3rd integer</td>
</tr>
<tr>
<td>4th integer</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
int *iptr = NULL;

iptr = malloc(4 * sizeof(int));

int *iptr2 = iptr + 3;
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));

int *iptr2 = iptr + 3;

Skip ahead by 3 times the size of iptr’s type (integer, size: 4 bytes).
Why Pointers?

• Using pointers seems like a lot of work, and if used incorrectly, things can go wrong.
• Pointers also add a level of “indirection” to retrieve / store a value

• Two main benefits:
  1. “Pass by pointer” function parameters
     • By passing a pointer into a function, the function can dereference it so that the changes persist to the caller.
  2. Dynamic memory allocation
     • A program can allocate memory on demand, as it needs it during execution
Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main(void) {
    int x, y; // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```
Arguments are passed by value
- The function gets a separate copy of the passed variable

```c
int func(int a, int b) {
    a = a + 5;
    return a - b;
}

int main(void) {
    int x, y; // declare two integers
    x = 4;
    y = 7;
    y = func(x, y);
    printf("%d, %d", x, y);
}
```

It doesn’t matter what `func` does with `a` and `b`. The value of `x` in `main` doesn’t change.
Pass by Pointer

• Want a function to modify a value on the caller’s stack? Pass a pointer!

• The called function can modify the memory location it points to.
  – passing the address of an argument to function:
  – pointer parameter holds the address of its argument
  – dereference parameter to modify argument’s value

• You’ve already used functions like this:
  – readline library functions and scanf
  – pass address of (&) argument to these functions
Function Arguments

• Arguments can be pointers!
  – The function gets the address of the passed variable!

```c
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4;
    func(&x);
    printf("%d", x);
}
```
Argument can be pointers!
  - The function gets the address of the passed variable!

```c
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4;
    func(&x);
    printf("%d", x);
}
```
• Arguments can be pointers!
  – The function gets the address of the passed variable!

```c
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4; // x is at mem. address 0x1A
    func(&x);
    printf("%d", x);
}
```
• Arguments can be pointers!
  – The function gets the address of the passed variable!

```c
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4; // x is at mem. address 0x1A
    func(&x);
    printf("%d", x);
}
```
Pointer Arguments

• Arguments can be pointers!
  – The function gets the address of the passed variable!

```c
void func(int *a) {
    *a = *a + 5;
}

int main(void) {
    int x = 4;
    func(&x);
    printf("%d", x);
}
```

Prints: 9
Haven’t we seen this somewhere before?

Stack:
```
main:
0x1A
x = 9
```
Readfile Library

• In lab 1/lab 4 with read_int, read_float.
  – This is why you needed an & when you pass the address of a variable.
  – e.g.,
    ```c
    int value;
    status_code = read_int(&value);
    ```

• You’re asking read_int to modify a parameter, so you give it a pointer (the address of) that parameter.
  – read_int will dereference it and set it.
int main(void){
    int x, y;
    x = 10; y = 20;
    //assume x at mem. address 0xF1
    foo(&x, y);
    ...
}

void foo(int *b, int c){
    c = 99
    *b = 8;  // Stack drawn here
}

Pass by Pointer - Example

Stack

main:

foo:

x 10

y 20
int main(void){
    int x, y;
    x = 10; y = 20;
    // assume x at mem. address 0xF1
    foo(&x, y);
    ...
}

void foo(int *b, int c){
    c = 99
    *b = 8; // Stack drawn here
}

dereference parameter b to set argument x’s value

Pass by Pointer - Example
Passing Arrays

- An array argument’s value is its base address
- Array parameter “points to” its array argument
Passing Arrays

• An array argument’s value is its base address
• Array parameter “points to” its array argument

```c
int main(void){
    int array[10];
    foo(array, 10);
}
void foo(int arr[], int n){
    arr[2] = 6;
}
```
Passing Arrays

- An array argument’s value is its base address
- Array parameter “points to” its array argument

```c
int main(void) {
    int array[10];
    foo(array, 10);
}

void foo(int arr[], int n) {
    arr[2] = 6;
}
```
Passing Arrays

• An array argument’s value is its base address
• Array parameter “points to” its array argument

```c
int main(void){
    int array[10];
    foo(array, 10);
}
void foo( _______ , int n){
    arr[2] = 6;
}
```
Passing Arrays

• An array argument’s value is its base address
• Array parameter “points to” its array argument

```c
int main(void){
    int array[10];
    foo(array, 10);
}
void foo(int *arr, int n){
    arr[2] = 6;
}
```

pass a pointer instead!
Can you return an array?

- Suppose you wanted to write a function that copies an array (of 5 integers).
  - Given: array to copy

```c
int copy_array(int array[]) {
    int result[5];
    result[0] = array[0];
    ...
    result[4] = array[4];
    return result;
}
```

As written above, this would be a terrible way of implementing this. (Don’t worry, compiler won’t let you do this anyway.)
Consider the memory...

copy_array(int array[]) {
    int result[5];
    result[0] = array[0];
    ... result[4] = array[4];
    return result;
}

(In main):
copy = copy_array(…)
Consider the memory…

copy_array(int array[]) {
    int result[5];
    result[0] = array[0];
    ...
    result[4] = array[4];
    return result;
}

(In main):
copy = copy_array(…)

<table>
<thead>
<tr>
<th>copy_array:</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>copy:</td>
</tr>
</tbody>
</table>
Consider the memory…

```c
void copy_array(int array[]) {
    int result[5];
    result[0] = array[0];
    ...
    result[4] = array[4];
    return result;
}
```

(In main):

```c
void copy_array(int array[])
{
    int result[5];
    result[0] = array[0];
    ...
    result[4] = array[4];
    return result;
}
```

Left with a pointer to nowhere.

When we return from copy_array, its stack frame is gone!
Using the Heap

```c
int *copy_array(int num, int array[]) {
    int *result = malloc(num * sizeof(int));

    result[0] = array[0];
    ...
    return result;
}
```

`malloc` memory is on the heap. Doesn't matter what happens on the stack (function calls, returns, etc.)
Pointers to Pointers

- Why stop at just one pointer?

```c
int **double_iptr;
```

- "A pointer to a pointer to an int."
  - Dereference once: pointer to an int
  - Dereference twice: int

- Commonly used to:
  - Allow a function to modify a pointer (data structures)
  - Dynamically create an array of pointers.
  - (Program command line arguments use this.)
Overview

- Stack data structure, applied to memory
- Behavior of function calls
- Storage of function data, at assembly level
"A" Stack

• A stack is a basic data structure
  – Last in, first out behavior (LIFO)
  – Two operations
    • Push (add item to top of stack)
    • Pop (remove item from top of stack)
“The” Stack

• Apply stack data structure to memory
  – Store local (automatic) variables
  – Maintain state for functions (e.g., where to return)

• Organized into units called *frames*
  – One frame represents all of the information for one function.
  – Sometimes called *activation records*
Memory Model

- Starts at the highest memory addresses, grows into lower addresses.
Stack Frames

- As functions get called, new frames added to stack.

- Example: Lab 4
  - main calls get_values()
  - get_values calls read_float()
  - read_float calls I/O library
Stack Frames

- As functions return, frames removed from stack.

- Example: Lab 4
  - I/O library returns to read_float
  - read_float returns to get_values
  - get_values returns to main

All of this stack growing/shrinking happens automatically (from the programmer’s perspective).
What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.
What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.
What needs to be stored in a stack frame?
- Alternatively: What *must* a function know / access?

Local variables

Stack Frame Contents
Local Variables

If the programmer says:

```
int x = 0;
```

Where should x be stored?

(Recall basic stack data structure)

Which memory address is that?
How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.
How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.
Program Characteristics

• Compile time (static)
  – Information that is known by analyzing your program
  – Independent of the machine and inputs

• Run time (dynamic)
  – Information that isn’t known until program is running
  – Depends on machine characteristics and user input
The Compiler Can…

• Perform type checking.

• Determine how much space you need on the stack to store local variables.

• Insert assembly instructions for you to set up the stack for function calls.
  – Create stack frames on function call
  – Restore stack to previous state on function return
Local Variables

Compiler can allocate N bytes on the stack by subtracting N from the “stack pointer”: rsp
The Compiler Can’t…Predict User Input

```c
int main(void) {
    int decision = [read user input];
    if (decision > 5) {
        funcA();
    }
    else {
        funcB();
    }
}
```

can the compiler predict which func goes here apriori?

main

0xFFFFFFFF
The Compiler Can’t…Predict User Input

```c
int main(void) {
    int decision = [read user input];
    if (decision > 5) {
        funcA();
    }
    else {
        funcB();
    }
}
```
The Compiler Can’t…

Predict user input.
Can’t assume a function will always be at a certain address on the stack.

Alternative: create stack frames relative to the current (dynamic) state of the stack.
Stack Frame Location

Where in memory is the current stack frame?

- current top of the stack
- current bottom of the stack

function 2
function 1
main
0xFFFFFFFF
Recall: x86_64 Register Conventions

• Working memory for currently executing program
  – Address of next instruction to execute ( %rip )
  
    – Location of runtime stack (%rbp, %rsp )

  – Temporary data ( %rax - %r15 )

  – Status of recent ALU tests ( CF, ZF, SF, OF )

%rip
%rbp
%rsp
%rbx
%rbx
%rcx
%rcx
%rdx
%rdx
%rsi
%rsi
%rdi
%rdi
%r14
%r14
%r15
%r15
%8
%8
%9
%9
%10
%10
%11
%11
%12
%12
%13
%13

General purpose registers

Current stack top
Current stack frame
Program Counter (PC)
Condition codes (flags)
Stack Frame Location

Where in memory is the current stack frame?

- rsp: stack pointer
- rbp: frame pointer (base pointer)

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler ensures that this invariant holds.

This is why all local variables we’ve seen in assembly are relative to rbp or rsp!

**Invariant:**
The current function’s stack frame is always between the addresses stored in rsp and rbp.
How would we implement pushing x to the top of the stack in x86_64?

A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp
How would we implement pushing x to the top of the stack in x86_64?

A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp
Local Variables

- Generally, we can make space on the stack for N bytes by:
  - subtracting N from \( \text{rsp} \)

![Diagram showing how to make space on the stack for N bytes by subtracting N from rbp.]

Current Stack Frame

New variable of N bytes

Current Stack Frame

N bytes
Local Variables

- When we’re done, free the space by adding $N$ back to $rsp$:
  $$rsp + N$$
Stack Frame Contents

What needs to be stored in a stack frame? **What must a function know?**

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address
- Saved registers
- Spilled temporaries
What needs to be stored in a stack frame?
– Alternatively: What must a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address
- Saved registers
- Spilled temporaries
Stack Frame Relationships

• If function 1 calls function 2:
  – function 1 is the **caller**
  – function 2 is the **callee**

• With respect to main:
  – main is the **caller**
  – function 1 is the **callee**
Where should we store the following stuff?

- Previous stack frame base address
- Function arguments
- Return value
- Return address

A. In registers
B. On the heap
C. In the caller’s stack frame
D. In the callee’s stack frame
E. Somewhere else
Calling Convention

• You could store this stuff wherever you want!
  – The hardware does NOT care.
  – What matters: everyone agrees on where to find the necessary data.

• **Calling convention**: agreed upon system for exchanging data between caller and callee

• When possible, keep values in registers (why?)
  – Accessing registers is faster than memory (stack)
The function’s **return value**: In register %rax

The caller’s %rbp value (caller’s **saved frame pointer**)
- Placed on the stack in the callee’s stack frame

The **return address** (saved PC value to resume execution on return)
- Placed on the stack in the caller’s stack frame

**Arguments** passed to a function:
- First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
- Any additional arguments stored on the caller’s stack frame (shared with callee)
Top of the Stack

- Space for local & temporary vars, & saved register values
- saved %rbp (Caller’s stack frame)
  - Return address (saved program counter)
  - parameter 7 (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
  - ...
  - parameter n

Callee’s Frame or Active Frame (current frame in execution)

Caller’s Frame

Earlier Stack Frames

Bottom of Stack

- Stack Pointer %rsp
- Frame or Base Pointer %rbp

Both caller & callee can access these:
- push %rip (PC)
- push input arguments to callee

Lower memory address

Call

Return

Higher memory address
x86_64 Calling Convention

• The function’s **return value**: In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  – Placed on the stack in the callee’s stack frame

• The **return address** (saved PC value to resume execution on return)
  – Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  – First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  – Any additional arguments stored on the caller’s stack frame (shared with callee)
Return Value

• If the callee function produces a result, the caller can find it in %rax

• We saw this when we wrote our function in the weekly lab last friday
  – Copy the result to %rax before we finishing up