CS 31: Intro to Systems C Programming
L14: Arrays, Structs, Strings, and Pointers

Vasanta Chaganti & Kevin Webb
Swarthmore College
October 31, 2023
Reading Quiz
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
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:

- 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)
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)
Top of the Stack

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

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

- Space for local & temporary vars, & saved register values
- saved %rbp (Caller’s stack frame)

Caller’s Frame

- Return address (saved program counter)
- parameter 7 (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
- parameter n

Earlier Stack Frames

Bottom of Stack

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

- lower memory address
  - call
- higher memory address
  - return
Dynamic Stack Accounting

• Dedicate CPU registers for stack bookkeeping
  – %rsp (stack pointer): Top of current stack frame
  – %rbp (frame pointer): Base of current stack frame

• Compiler maintains these pointers
  – Does the compiler know the exact address they point to?
  – Compiler doesn’t know or care! (job of the OS to figure that out)

• To the compiler: every variable access is relative to %rsp and %rbp!
Compiler: updates to rsp/rbp on function call/return

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call..  

Immediately upon calling a new function:

1. push current `%rbp`
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant: The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:
1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

**Invariant:**
The current function’s stack frame is always between the addresses stored in rsp and rbp

```
caller's %rbp value
```

```
caller stack frame
...
```
Compiler: Upon a new Function Call..

Immediately upon calling a new function:
1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant: The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp

Invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Returning from a function call..

Returning from a function:

1. Set $\%rsp = \%rbp$ (callee stack frame no longer exists)
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp (callee stack frame no longer exists)
2. pop %rbp

Invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller’s rbp off the stack and set it to the value of rbp
   - decrement rsp

invariant: The current function’s stack frame is always between the addresses stored in rsp and rbp

X86_64 has another convenience instruction for this: leaveq
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller’s rbp off the stack and set it to the value of rbp
   - decrement rsp

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp

Back to where we started
x86 Calling Conventions: Function Call

Initial state

push %rbp (store caller’s base pointer)

mov %rsp, %rbp (establish callee’s frame pointer)

sub $SIZE, %rsp (allocate space for callee’s locals)
x86 Calling Conventions: Function Return

x86_64 provides a convenience instruction that does all of this: `leaveq`

-we want to restore the caller’s frame

mov %rbp, %rsp (restore caller’s stack pointer)

pop %rbp (restore caller’s frame pointer)
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)
Instructions in Memory

0x0

Operating system
Text
Data
Heap

Stack

0xFFF00000

funcA:
...
callq funcB
...
funcB:
push %rbp
mov %rsp, %rbp
...

Function A

Function B
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

add $5, %rcx

text Memory Region

callq funcB
add %rax, %rcx

funcB:
push %rbp
mov %rsp, %rbp
mov $10, %rax
leaveq
retq

Program Counter
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

\[
\text{add } $5, \%rcx
\]

Update PC to next instruction.

Execute the \texttt{addl}.

\begin{verbatim}
funcA:  
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...  
callq funcB  
  add %rax, %rcx  
  ...
funcB:  
  push %rbp  
  mov %rsp, %rbp  
  ...  
  mov $10, %rax  
  leaveq  
  retq
\end{verbatim}
Functions and the Stack

Executing instruction:
callq funcB

PC points to next instruction

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
Functions and the Stack

1. push %rip

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...
  callq funcB
  add %rax, %rcx
  ...

funcB:
  push %rbp
  mov %rsp, %rbp
  ...
  mov $10, %rax
  leaveq
  retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...  
callq funcB
  add %rax, %rcx
  ...  
funcB:
  push %rbp
  mov %rsp, %rbp
  ...  
mov $10, %rax
  leaveq
  retq
```

Stack Memory Region

Function B

Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

...
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Functions and the Stack

Text Memory Region

6. (resume funcA)

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
```

```
callq funcB
add %rax, %rcx
...
```

```
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Recap: PC upon a Function Call

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
```

```
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Functions and the Stack

Program Counter (%rip)

Stack Memory Region

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.

Stored PC in funcA
(Address of instruction: add %rax, %rcx)

Function A
...
### x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>push</strong></td>
<td>Create space on the stack and place the source there.</td>
<td><code>sub $8, %rsp&lt;br&gt;mov src, (%rsp)</code></td>
</tr>
<tr>
<td><strong>pop</strong></td>
<td>Remove the top item off the stack and store it at the destination.</td>
<td><code>mov (%rsp), dst&lt;br&gt;add $8, %rsp</code></td>
</tr>
<tr>
<td><strong>callq</strong></td>
<td>1. Push return address on stack 2. Jump to start of function</td>
<td><code>push %rip&lt;br&gt;jmp target</code></td>
</tr>
<tr>
<td><strong>leaveq</strong></td>
<td>Prepare the stack for return (restoring caller’s stack frame)</td>
<td><code>mov %rbp, %rsp&lt;br&gt;pop %rbp</code></td>
</tr>
<tr>
<td><strong>retq</strong></td>
<td>Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))</td>
<td><code>pop %rip</code></td>
</tr>
</tbody>
</table>
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)
Function Arguments

- Most functions don’t receive more than 6 arguments, so x86_64 can simply use registers most of the time.

- If we do have more than 6 arguments though (e.g., perhaps a printf with lots of placeholders), we can’t fit them all in registers.

- In that case, we need to store the extra arguments on the stack. By convention, they go in the caller’s stack frame.
If we need to place arguments in the caller’s stack frame, should they go above or below the return address?

A. Above
B. Below
C. It doesn’t matter
D. Somewhere else
If we need to place arguments in the caller’s stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn’t matter

D. Somewhere else
# x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>push</strong></td>
<td>Create space on the stack and place the source there.</td>
<td><code>sub $8, %rsp</code>&lt;br&gt;<code>mov src, (%rsp)</code></td>
</tr>
<tr>
<td><strong>pop</strong></td>
<td>Remove the top item off the stack and store it at the destination.</td>
<td><code>mov (%rsp), dst</code>&lt;br&gt;<code>add $8, %rsp</code></td>
</tr>
<tr>
<td><strong>callq</strong></td>
<td>1. Push return address on stack&lt;br&gt;2. Jump to start of function</td>
<td><code>push %rip</code>&lt;br&gt;<code>jmp target</code></td>
</tr>
<tr>
<td><strong>leaveq</strong></td>
<td>Prepare the stack for return (restoring caller’s stack frame)</td>
<td><code>mov %rbp, %rsp</code>&lt;br&gt;<code>pop %rbp</code></td>
</tr>
<tr>
<td><strong>retq</strong></td>
<td>Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))</td>
<td><code>pop %rip</code></td>
</tr>
</tbody>
</table>
Arguments

• Extra arguments to the callee are stored just underneath the return address.

• Does it matter what order we store the arguments in?

• Not really, as long as we’re consistent (follow conventions).

This is why arguments can be found at positive offsets relative to %rbp.
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

Bottom of Stack

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

Caller’s Frame

Earlier Stack Frames

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

higher memory address

return
Stack Frame Contents

• 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
Saving Registers

• Registers are a relatively scarce resource, but they’re fast to access. Memory is plentiful, but slower to access.

• Should the caller save its registers to free them up for the callee to use?
• Should the callee save the registers in case the caller was using them?
• Who needs more registers for temporary calculations, the caller or callee?

• Clearly the answers depend on what the functions do...
Splitting the difference…

• We can’t know the answers to those questions in advance…

• Divide registers into two groups:

  Caller-saved: `%rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
    Caller must save them prior to calling callee
    callee free to trash these,
    Caller will restore if needed

  Callee-saved: `%rbx, %r12, %r13, %r14, %r15
    Callee must save them first, and restore
    them before returning
    Caller can assume these will be preserved
Running Out of Registers

• Some computations require more than 16 general-purpose registers to store temporary values.

• *Register spilling*: The compiler will move some temporary values to memory, if necessary.
  – Values pushed onto stack, popped off later
  – No explicit variable declared by user
  – This is getting to the limits of CS 31!
    • – take CS 75 (compilers) for more details.
Up next…

- Connecting Arrays, Structs, and Pointers with assembly
Today

• Accessing *things* via an offset
  – Arrays, Structs, Unions
  – Connect accessing them in C with what we know about assembly

• How complex structures are stored in memory
  – Multi-dimensional arrays & Structs
Announcements

- Midterm, Lab4 grades are posted
- HW 3 is out – due on Thursday
So far: Primitive Data Types

• We’ve been using ints, floats, chars, pointers

• Simple to place these in memory:
  – They have an unambiguous size
  – They fit inside a register*
  – The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)
Composite Data Types

• Combination of one or more existing types into a new type. (e.g., an array of multiple ints, or a struct)

• Example: a queue
  – Might need a value (int) plus a link to the next item (pointer)

```c
struct queue_node{
  int value;
  struct queue_node *next;
}
```
Recall: Arrays in Memory

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

```c
int val[5];
0th bucket 1st bucket 2nd bucket 3rd bucket 4th bucket
```
We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

This is why we start counting from zero!
Skipping forward with an offset of zero ([0]) gives us the first bucket...
Which expression would compute the address of iptr[3]?

A. 0x0824 + 3 * 4
B. 0x0824 + 4 * 4
C. 0x0824 + 0xC
D. More than one (which?)
E. None of these
Which expression would compute the address of iptr[3]?

A. 0x0824 + 3 * 4
B. 0x0824 + 4 * 4
C. 0x0824 + 0xC
D. More than one (which?)
E. None of these

Heap

<table>
<thead>
<tr>
<th>Address</th>
<th>iptr[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>

What if this isn’t known at compile time?
Recall Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with () around the register’s name.

```
mov (%rcx), %rax
```
  - Use the address in register %rcx to access memory, store result in register %rax
Recall Addressing Mode: Displacement

• Like memory mode, but with a constant offset
  – Offset is often negative, relative to %rbp

`mov -24(%rbp), %rax`
  – Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.
Addressing Mode: Indexed

- Instead of only using one register to store the base address of a memory address, we can use a base address register and an offset register value.

```assembly
mov (%rax, %rcx), %rdx
```
- Take the base address in %rax, add the value in %rcx to produce a final address, index into memory and store the result in %rdx.
Addressing Mode: Indexed

Instead of only using one register to store the base address of a memory address, we can use a base address register and an offset register value.

```plaintext
mov (%rax, %rcx), %rdx

- Take the base address: %rax,
- add the value in %rcx: %rax + %rcx
- index into memory and store the result in %rdx.
```
Addressing Mode: Indexed

The offset (%rcx) can also be scaled by a constant.

\[
\text{mov } (\%rax, \%rcx, 4), \%rdx
\]

- Take the base address: %rax
- Multiply the offset by the scale: %rcx * 4
- Add the scaled offset to the base: %rax + %rcx * 4
- Now, index into memory at (%rax + %rcx * 4) and store the result in %rdx.

One register to keep track of base address.
One register to keep track of offset from base address.
Scale Constant
This mode has been on your assembly reference sheet all along!

**Memory (Indexed)**
Access memory at the address stored in a register (base) plus a constant, C, plus a scale * a register (index):
C(%base, %index, scale)

Examples:
(%rax, %rcx)
0x8(%rbp, %rax, 8)
Let’s try an example

Suppose:

```c
int iptr = malloc(4*sizeof(int));
// iptr is stored in register %rax.
int i=2; is stored at %rbp-8
```

C code says:

```c
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?
Let’s try an example

Suppose:

```c
int iptr = malloc(4*sizeof(int));
//iptr is stored in register %rax.
int i=2; is stored at %rbp-8
```

C code says:

```c
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?

```assembly
mov -8(%rbp), %rcx
```
Let’s try an example

Suppose:

```c
int iptr = malloc(4*sizeof(int));
//iptr is stored in register %rax.
int i=2; is stored at %rbp-8
```

C code says:

```c
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```
Let’s try an example

Suppose:

```plaintext
int iptr; is stored in register %rax.
int i=2; is stored at %rbp-8
iptr[i] = 9; //iptr[2] = 9;
```

In assembly:

```plaintext
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

![Heap diagram showing addresses and values]

- `rax: Array base address`
- `Registers:
  - rax: 0x0824
  - rcx
  - rdx: 9`
- `Heap:
  - 0x0824: iptr[0]
  - 0x0828: iptr[1]
  - 0x082C: iptr[2]
  - 0x0830: iptr[3]`
Let’s try an example

Suppose:

```c
int iptr; is stored in register %rax.
int i=2; is stored at %rbp-8
iptr[i] = 9; //iptr[2] = 9;
```

In assembly:

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

Registers:

- `rax`: Array base address
- `rcx`: index
- `rdx`: offset

Heap:

- `0x0824`: iptr[0]
- `0x0828`: iptr[1]
- `0x082C`: iptr[2]
- `0x0830`: iptr[3]
What happens when we increment i? What changes do we make in assembly?

Suppose:

```c
int iptr; is stored in register %rax.
int i=3; is stored at %rbp-8
iptr[i] = 10; // iptr[3] = 10;
```

In assembly:

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

= add (rcx * 4)
= add (2 * 4)
= add 8

rax: Array base address

Registers:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>0x0824</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td>9</td>
</tr>
</tbody>
</table>

Heap

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824:</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828:</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C:</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830:</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>
Suppose:

\[ \text{int iptr;} \]

is stored in register \%rax.

\[ \text{int i = 3; is stored at \%rbp - 8} \]

\[ \text{iptr[i] = 10; // iptr[3] = 10;} \]

In assembly:

\[ \text{mov -8(\%rbp), \%rcx} \]

\[ \text{mov \%rdx, (\%rax, \%rcx, 4)} \]

= add (\%rcx * 4)
= add (2 * 4)
= add 8

What happens when we increment \( i \)?

What changes do we make in assembly?

From here, if the program increments \( i \) (e.g., in a loop) and accesses the array at the new (incremented) position of \( i \):

Compiler can simply increment register \%rcx and access the next element of the array with the same \texttt{mov} command!
Structs

• Multiple values (fields) stored together
  – Defines a new type in C's type system

• Laid out contiguously by field (with a caveat we'll see later)
  – In order of field declaration.
Structs

Laid out contiguously by field (with a caveat we'll see later)
– In order of field declaration.

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
```
Structs

Struct fields accessible as a base + displacement
– Compiler knows (constant) displacement of each field

```c
struct student{
    int age;
    float gpa;
    int id;
};
```

```c
struct student s;
```
Structs

Struct fields accessible as a **base + displacement**
– Compiler knows (constant) displacement of each field

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
```

![Diagram showing memory allocation and field offsets]

- Given the starting address of a struct...
- The id field is always at an offset of 8 forward from the start.
Structs

Struct fields accessible as a base + displacement
In assembly: mov reg_value, 8(reg_base)

Where:
• reg_value is a register holding the value to store (say, 12)
• reg_base is a register holding the base address of the struct

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
s.id = 12;
```

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s.age</td>
<td>0x1234</td>
</tr>
<tr>
<td>s.gpa</td>
<td>0x1238</td>
</tr>
<tr>
<td>s.id</td>
<td>0x123c</td>
</tr>
</tbody>
</table>

Given the starting address of a struct...
The id field is always at an offset of 8 forward from the start.
Structs

• Laid out contiguously by field
  – In order of field declaration.
  – May require some padding, for alignment.
Data Alignment:

- Where (which address) can a field be located?

- **char (1 byte)**: can be allocated at any address:
  
  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, ...

- **short (2 bytes)**: must be aligned on 2-byte addresses:
  
  0x1230, 0x1232, 0x1234, 0x1236, 0x1238, ...

- **int (4 bytes)**: must be aligned on 4-byte addresses:
  
  0x1230, 0x1234, 0x1238, 0x123c, 0x1240, ...
Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.
Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware

- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)

- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary
Structs

• Laid out contiguously by field
  – In order of field declaration.
  – May require some padding, for alignment.

```c
struct student {
    int age;
    float gpa;
    int id;
};

struct student s;
```
struct student{
    char name[11];
    short age;
    int id;
};
How much space do we need to store one of these structures? Why?

```c
struct student{
    char name[11];
    short age;
    int id;
};
```

A. 17 bytes  
B. 18 bytes  
C. 20 bytes  
D. 22 bytes  
E. 24 bytes
struct student{
    char name[11];
    short age;
    int id;
};

size of data: 17 bytes
size of struct: 20 bytes!

Use sizeof() when allocating structs with malloc()!
Alternative Layout

```c
struct student{
    char name[11];
    short age;
    int id;
};
```

Same fields, declared in a different order.
Alternative Layout

struct student{
  char name[11];
  short age;
  int id;
};

size of data: 17 bytes
size of struct: 17 bytes

In general, this isn’t a big deal on a day-to-day basis. Don’t go out and rearrange all your struct declarations.
Aside: Network Headers

• In networks, we attach metadata to packets
  – Things like destination address, port #, etc.

• Common for these to be a specific size/format
  – e.g., the first 20 bytes must be laid out like ...

• Naïvely declaring a struct might introduce padding, violate format.
Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)
Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.

- Rearranging helps, but often padding after the struct can’t be eliminated.

```c
struct T1 {
    char c1;
    char c2;
    int x;
};
```

```c
struct T2 {
    int x;
    char c1;
    char c2;
};
```

![Diagram showing padding comparison between T1 and T2](image)
"External" Padding

Array of Structs: Field values in each bucket must be properly aligned:

```c
struct T2 arr[3];
```

Buckets must be on a 8-byte aligned address
struct student {
    int id;
    short age;
    char name[11];
};
struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, “Alice”);

Struct is declared on the stack. (NOT a pointer)
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));

How do we get to the id and age?
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));

How do we get to the id and age?

Option 1: Works but ugly

(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, "Alice");

Option 2: Use struct pointer dereference!

s->id = 406432;
s->age = 20;
strcpy(s->name, "Alice");
Memory alignment applies elsewhere too!

```c
int x;         vs.    double y;
char ch[5];
short s;
double y;
```

In nearly all cases, you shouldn't stress about this. The compiler will figure out where to put things.

Exceptions: networking, OS
Unions

• Declared like a struct, but only contains one field, rather than all of them.

• Struct: field 1 and field 2 and field 3 ...
• Union: field 1 or field 2 or field 3 ...

Intuition: you know you only need to store one of N things, don’t waste space.
Unions

```c
struct my_struct {
    char ch[2];
    int i;
    short s;
}

union my_union {
    char ch[2];
    int i;
    short s;
}
```

my_struct in memory

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = 'a';

u.i = 5;
```

Same memory used for all fields!

Reading `i` or `s` here would be bad
Unions

• You probably won’t use these often.

• Use when you need mutually exclusive types.

• Can save memory.

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Two-dimensional Arrays

• Why stop at an array of ints?  
  How about an array of arrays of ints?

int twodims[3][4];

• “Give me three sets of four integers.”

• How should these be organized in memory?
Two-dimensional Arrays

```c
int twodims[3][4];
for(i=0; i<3; i++) {
    for(j=0; j<4; j++) {
        twodims[i][j] = i+j;
    }
}
```
Two-dimensional Arrays

```c
int twodims[3][4];
for(i=0; i<3; i++) {
    for(j=0; j<4; j++) {
        twodims[i][j] = i+j;
    }
}
```
Memory Layout

- Matrix: 3 rows, 4 columns

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

Row Major Order:
- all Row 0 buckets, followed by
- all Row 1 buckets, followed by
- all Row 2 buckets, ...
Memory Layout

- Matrix: 3 rows, 4 columns

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{twodim[1][3]}: \quad \text{base addr} + \text{row offset (# rows * rows * sizeof(int))} + \text{col offset}
\]

\[
\begin{align*}
\text{twodim} & \quad + 1 \times \text{ROWSIZE} \times 4 + 3 \times 4 \\
0xf260 & \quad + 16 + 12 \\
= & \quad 0xf27c
\end{align*}
\]
If we declared `int matrix[5][3];`, and the base of matrix is 0x3420, what is the address of `matrix[3][2]`?

A. 0x3438  
B. 0x3440  
C. 0x3444  
D. 0x344C  
E. None of these
If we declared `int matrix[5][3]`;, and the base of matrix is 0x3420, what is the address of `matrix[3][2]`?

A. 0x3438
B. 0x3440
C. 0x3444
D. 0x344C
E. None of these

```
+ row offset (# rows * row_size * sizeof(data_type))
+ col offset
```
Dynamic Two-dimensional Array

• Given the *row-major order* layout, a "two-dimensional array" is still just a contiguous block of memory:

• The malloc function returns... a pointer to a contiguous block of memory!

```
  0xf260  0  twodim[0][0]
  0xf264  1  twodim[0][1]
  0xf268  2  twodim[0][2]
  0xf26c  3  twodim[0][3]
  0xf270  1  twodim[1][0]
  0xf274  2  twodim[1][1]
  0xf278  3  twodim[1][2]
  0xf27c  4  twodim[1][3]
  0xf280  2  twodim[2][0]
  0xf284  3  twodim[2][1]
  0xf288  4  twodim[2][2]
  0xf28c  5  twodim[2][3]
```
Dynamic Two-dimensional Array

- For this example, with three rows and four columns:

```
0   1   2   3
1   2   3   4
2   3   4   5
```

```c
int *matrix = malloc(3 * 4 * sizeof(int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more than one index (i.e., row and column).

Can't access: `matrix[i][j]`
For this example, with three rows and four columns:

```c
int *matrix = malloc(3 * 4 * sizeof(int));

// Compute the offset manually
index = i * ROWSIZE + j;
matrix[index] = ...
Two-dimensional array alternative

• (Dynamically) Allocate an array of pointers. For each pointer, (dynamically) allocate an array.

• How do we get an array of pointers?
Two-dimensional array alternative

• If we want a dynamic array of ints:
  – declare int *array = malloc(N * sizeof(int))

• So... if we want an array of int pointers:
  – declare int **array = malloc(...)
Two-dimensional array alternative

• If we want a dynamic array of ints:
  – declare int *array = malloc(N * sizeof(int))

• So... if we want an array of int pointers:
  – declare int **array = malloc(N * sizeof(int *))
  – The type of array[0], array[1], etc. is: int *
  – For each one of those, we can malloc an array of ints:
    • array[0] = malloc(M * sizeof(int))
Two-dimensional array alternative

```c
int **two_d_array;

two_d_array = malloc(sizeof(int *) * N);
for (i=0; i < N; i++) {
    two_d_array[i] = malloc(sizeof(int) * M);
}
```
Two-dimensional arrays

- We'll use BOTH methods in future labs.
Strings

- Strings are *character arrays*

- Layout is the same as:
  - char name[10];

- Often accessed as (char *)
String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```c
char name[10];
strcpy(name, "CS 31");
```
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

char name[10];
strcpy(name, “CS 31”);

• Null terminator (\0) ends string.
  – We don’t know/care what comes after
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

• Seems simple on the surface.
  – That null terminator is tricky, strings error-prone.
  – Strings used everywhere!

• You will implement use these functions in a future lab.
Up next…

• New topic: Storage and the Memory Hierarchy