Lecture 12
Structs and Procedures

Comp_Sci 213

Introduction to Computer Systems

Wednesday, Nov. 4, 2026

Kate Smith – Fall 2026

Slides adapted from:
Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Notes

- HW 2 (due nov 19)
- Attack lab out (nov 17)
  - Watch video! It is helpful

Topics

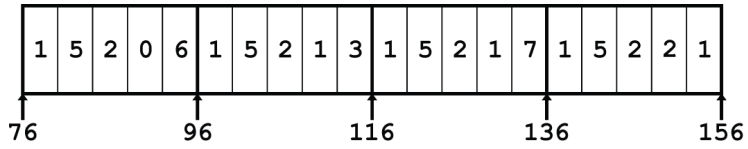
- Struct Layout
- Struct Padding and Alignment
- Procedures

Topics

- **Struct Layout**
- Struct Padding and Alignment
- Procedures

Recap: Nested versus Multi-Level Arrays

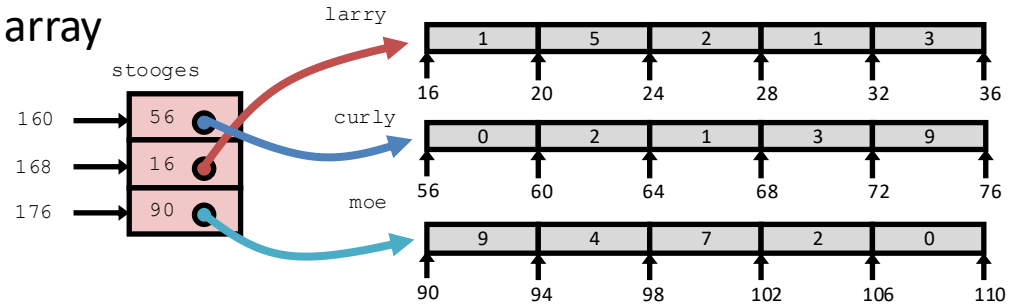
Nested array



`Mem[ord+(20*index)+(4*digit)]`

- Strengths
  - Fast element access
    - Single memory access
  - Efficient memory usage
    - Stored in contiguous memory
- Limitations
  - Requires fixed size rows
  - Large memory usage
    - All rows need to be allocated

Multi-level array

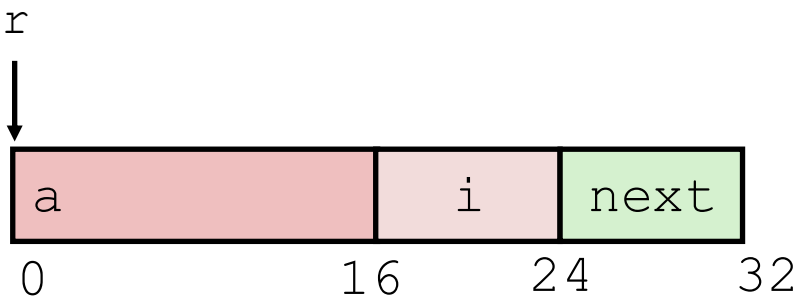


`Mem[Mem[stooges+(8*index)]+(4*digit)]`

- Strengths
  - Rows may be of different size
  - Rows could even be different types
    - First array would store `void*`
- Limitations
  - Slow element access
    - Two memory references
  - Memory fragmentation
    - Many small chunks allocated

Structure representation in C

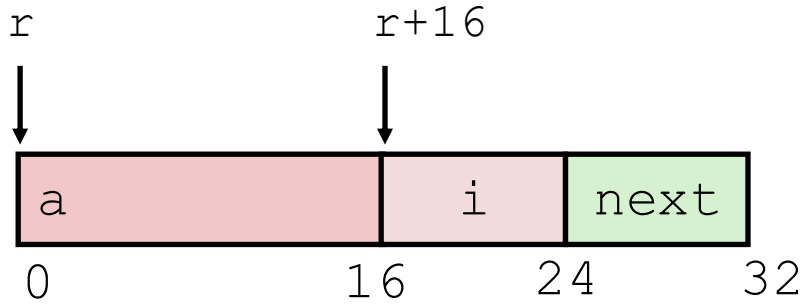
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as chunk of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering could yield a more compact representation
  - (We'll see how that could happen in a bit)
- Compiler determines overall size + positions of fields
  - Looking at memory, no way to tell it's a struct (like arrays, its just bytes)
  - It's all in how the code treats that region of memory!

Structure access

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Accessing Structure Member
  - Pointer **`r`** indicates first byte of structure
  - Access member with offsets
  - Offset of each structure member determined at compile time
    - Another use for Displacement in memory addressing!

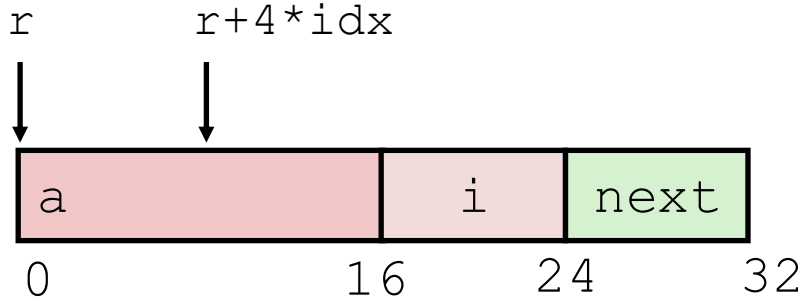
```
size_t get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi
  movq 16(%rdi), %rax
  ret
```

`r` is a pointer to a struct.
Dereference the ponter, then get the `i` field of the struct.

Array within a struct

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Same as before; just need to also index in the array
  - Pointer **`r`** indicates first byte of structure
    - Offset of each structure member determined at compile time
    - Offset into array determined based on index and type
  - Compute as **`*(structAddr + offset + K*index);`**
    - Uses full addressing mode!

```
int get_a (struct rec *r,
           size_t idx)
{
  return r->a[idx];
}
```

```
  # r in %rdi
  # idx in %rsi
  movl 0(%rdi,%rsi,4), %eax
  ret
```

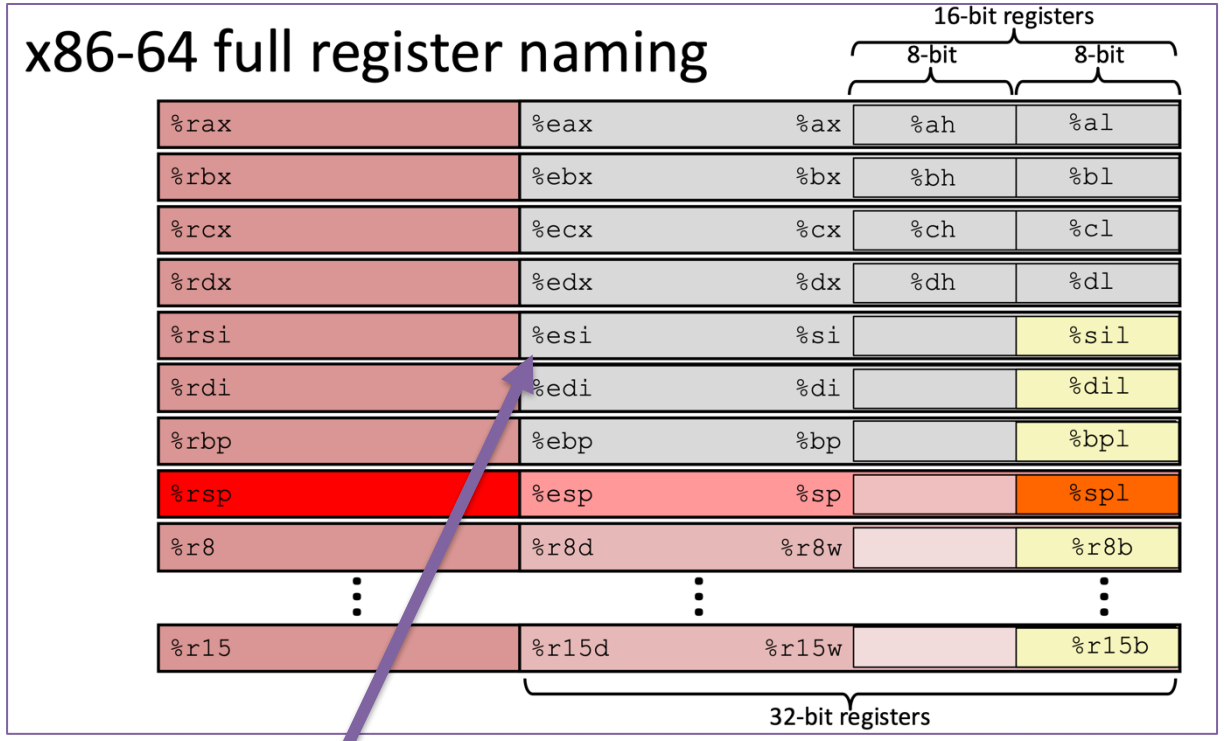
Structure Access Practice 1

Function Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```




```
movl  %esi , 4(%rdi)
ret
```

Structure Access Practice 1

Function Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

What happened??

- The struct layout – the compiler puts fields in declaration order
  - `int`: 4 byte size (the array has 2 `int`s)
  - `struct rec *n` : 8 byte size, this is a pointer to another `struct rec` (this is a way to link records together as in a linked list where `n` points to the next node)
- Total size of the struct is 24 bytes
- `%rdi` holds the structs address

```
movl  %esi , 4(%rdi)
ret
```

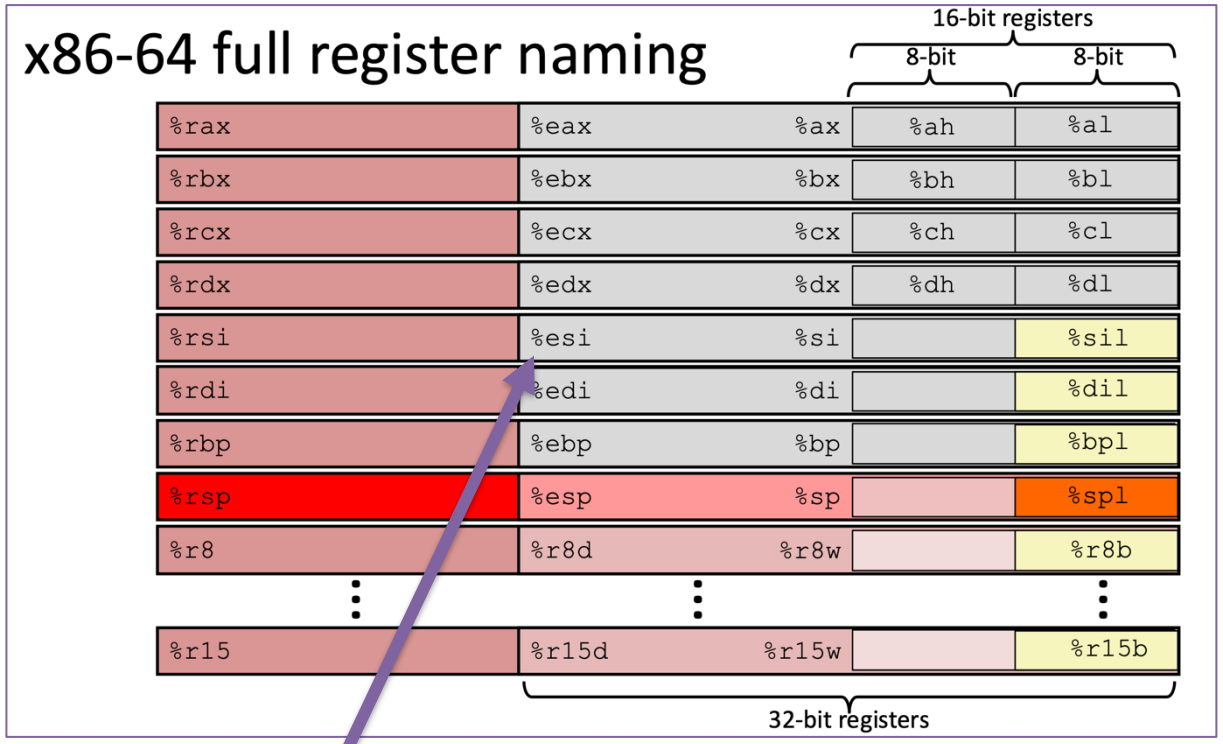
Structure Access Practice 2

Function Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_a1(struct rec *r,
       int val)
{
  r->a[1] = val;
}
```




```
movl   %esi , 12(%rdi)
ret
```

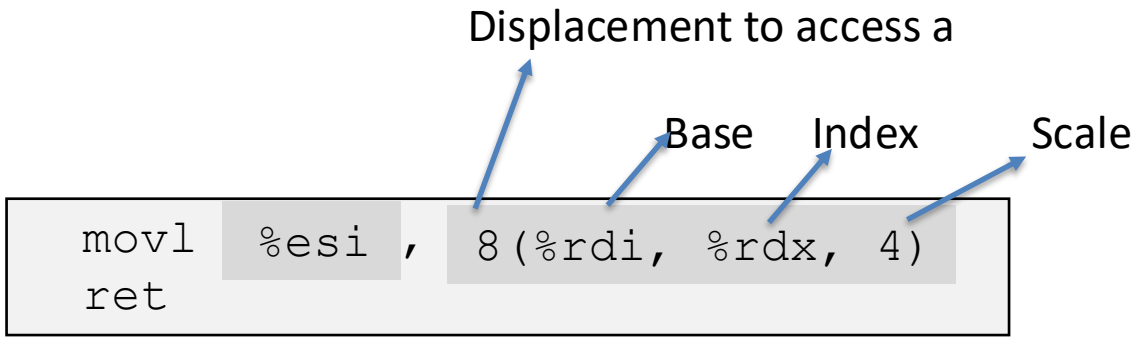
Structure Access Practice 3

Function Arguments:

1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_a(struct rec *r,
      int val,
      int index)
{
  r->a[index] = val;
}
```

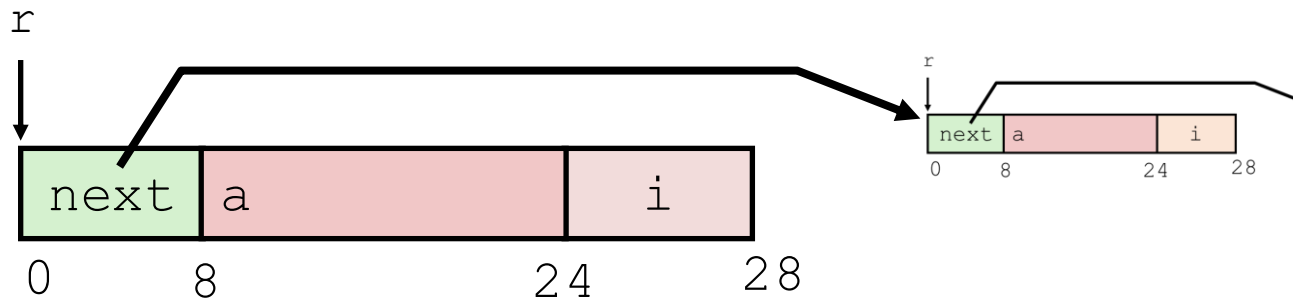


Following Linked List

By convention, null `next` pointer indicates end of list

```
void set_val
  (struct rec *r, int val)
{
  while (r) { // until we hit a NULL pointer
    int i = r->i; // read this node's index
    r->a[i] = val; // write value into a[i]
    r = r->next; // move to next node
  }
}
```

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```




```
.L11:                                   # loop:
  movslq  24(%rdi), %rax                #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)          #   M[r+8+4*i] = val
  movq    (%rdi), %rdi                  #   r = M[r]
  testq   %rdi, %rdi                    #   Test r
  jne     .L11                          #   if !=0 goto loop
```

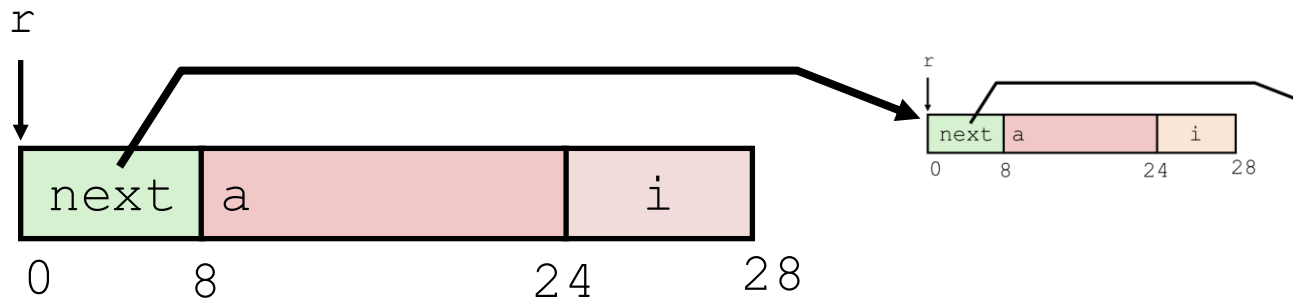
Register	Value
%rdi	r
%rsi	val

Following Linked List

By convention, null `next` pointer indicates end of list

```
void set_val
  (struct rec *r, int val)
{
  while (r) { // until we hit a NULL pointer
    int i = r->i; // read this node's index
    r->a[i] = val; // write value into a[i]
    r = r->next; // move to next node
  }
}
```

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```



Write `val` into `r->a[i]`

```
.L11:                                  # loop:
  movslq  24(%rdi), %rax               #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)         #   M[r+8+4*i] = val
  movq    (%rdi), %rdi                 #   r = M[r]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                         #   if !=0 goto loop
```

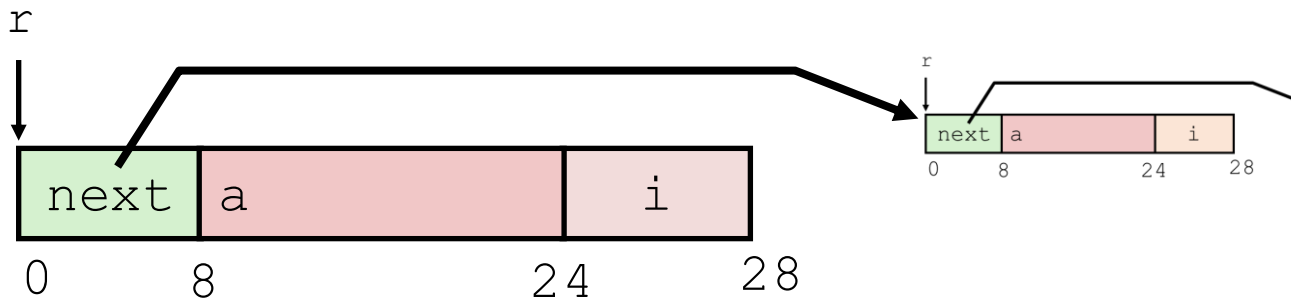
Register	Value
%rdi	r
%rsi	val

Following Linked List

By convention, null `next` pointer indicates end of list

```
void set_val
  (struct rec *r, int val)
{
  while (r) { // until we hit a NULL pointer
    int i = r->i; // read this node's index
    r->a[i] = val; // write value into a[i]
    r = r->next; // move to next node
  }
}
```

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```



Move to next node in list

```
.L11:                                   # loop:
  movslq  24(%rdi), %rax                #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)          #   M[r+8+4*i] = val
  movq    (%rdi), %rdi                  #   r = M[r]
  testq   %rdi, %rdi                    #   Test r
  jne     .L11                          #   if !=0 goto loop
```

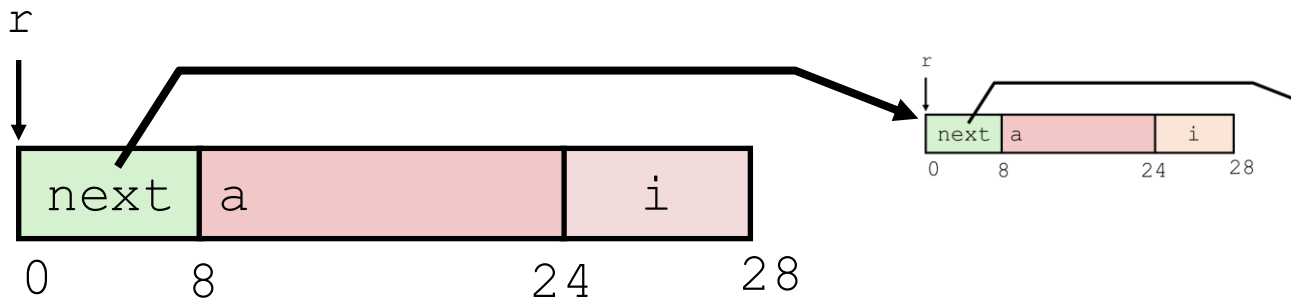
Register	Value
%rdi	r
%rsi	val

Following Linked List

By convention, null `next` pointer indicates end of list

```
void set_val
  (struct rec *r, int val)
{
  while (r) { // until we hit a NULL pointer
    int i = r->i; // read this node's index
    r->a[i] = val; // write value into a[i]
    r = r->next; // move to next node
  }
}
```

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```



`NULL` check

```
.L11:                                   # loop:
  movslq  24(%rdi), %rax                #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)          #   M[r+8+4*i] = val
  movq    (%rdi), %rdi                  #   r = M[r]
  testq   %rdi, %rdi                    #   Test r
  jne     .L11                          #   if !=0 goto loop
```

Register	Value
%rdi	r
%rsi	val

Topics

- Struct Layout
- **Struct Padding and Alignment**
- Procedures

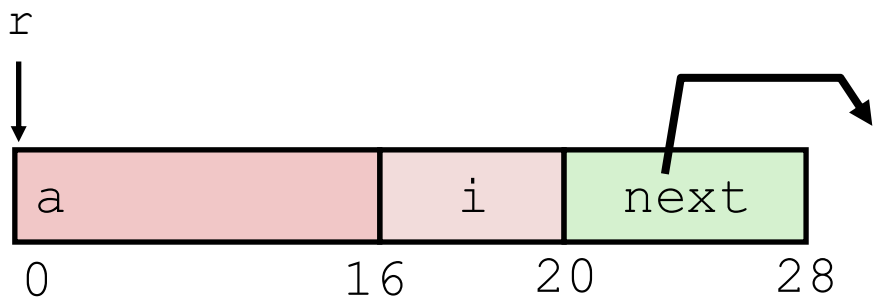
Alignment

- Aligned data
  - Primitive data type requires K bytes
  - Address must typically be a multiple of K (e.g., 1,2,4 or 8)
    - An address that is a multiple of K is called “K-byte aligned”
- Required on some machines; “recommended” on x86-64
  - But not doing it will really slow down your program
  - And some libraries might still crash
- For example, pointers need 8-byte alignment
  - Multiple of 8 is fine, non-multiple of 8 is bad

Problem: reordering can lead to different layouts

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```



Register	Value
%rdi	r
%rsi	val

```
.L11:                                  # loop:
  movslq  16(%rdi), %rax               #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)          #   M[r+4*i] = val
  movq    ??(%rdi), %rdi               #   r = M[r+??]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                         #   if !=0 goto loop
```

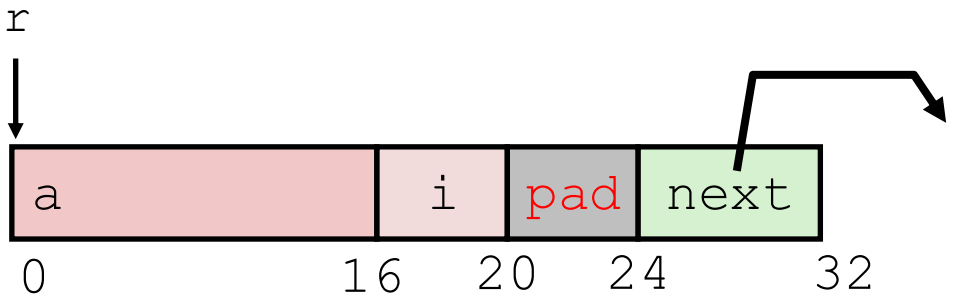
Alignment is wrong!!

Can't load 8 bytes from address 20 efficiently

Padding is added to struct to preserve *alignment*

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	**r**
%rsi	**val**

```
.L11:                                  # loop:
  movslq  16(%rdi), %rax               #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)          #   M[r+4*i] = val
  movq    24(%rdi), %rdi               #   r = M[r+24]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                         #   if !=0 goto loop
```

The why and how of alignment

- Motivation for aligning data
  - Inefficient to load or store values that span quad word boundaries
  - Hardware is really good at loading, e.g., 8 bytes at address 16, or 24, or 32
    - If you want 8 bytes at address 12, may need two memory reads. Oops…
    - Some unaligned accesses may even crash your code
- Secondary motivations
  - Having one value spanning 2 cache lines = two cache accesses per access
  - Virtual memory very tricky when a data spans 2 pages
  - See upcoming lectures on caches and virtual memory
- The compiler manages alignment
  - Inserts gaps in structure to ensure correct alignment of fields
    - Tradeoff: waste a little memory to improve performance
  - All variables need alignment, so the stack pointer is aligned too!

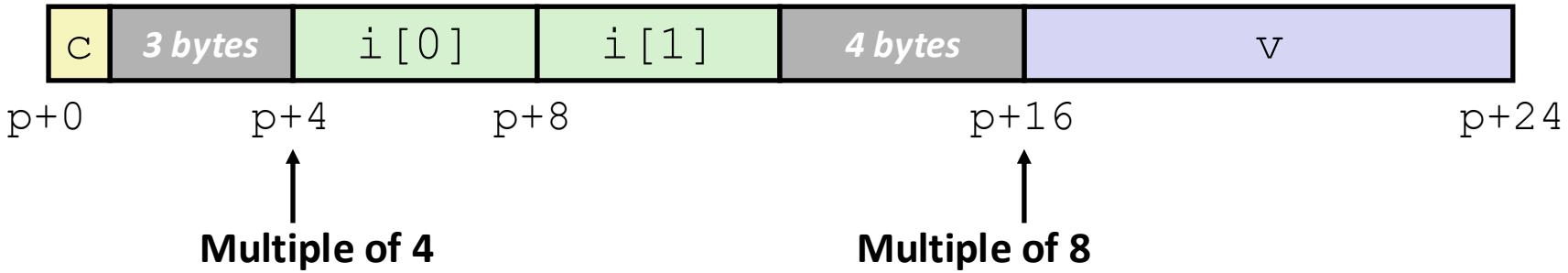
Specific Cases of Alignment (x86-64, Linux)

- Rule: always align to multiple of your size
- 1 byte: **`char`**
  - 1-byte aligned (no restrictions on address)
- 2 bytes: **`short`**
  - 2-byte aligned (lowest 1 bit of address must be **`0`**)
- 4 bytes: **`int`**, **`float`**
  - 4-byte aligned (lowest 2 bits of address must be **`00`**)
- 8 bytes: **`long`**, **`long long`**, **`double`**, **`char*`** (any pointer)
  - 8-byte aligned (lowest 3 bits of address must be **`000`**)
- 16 bytes: **`long double`**
  - 16-byte aligned (lowest 3 bits of address must be **`0000`**)
  - Max possible alignment requirement on x86-64
    - This is where the “stack moves by 16s” rule comes from

Satisfying Alignment within Structures

- Within structure
  - Must satisfy each element's alignment requirement
- Insert padding in the middle of the struct to guarantee this
- Examples:
  - `i[0]` aligned to 4-byte (multiple of element size, not array size)
  - `v` aligned to 8-byte

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

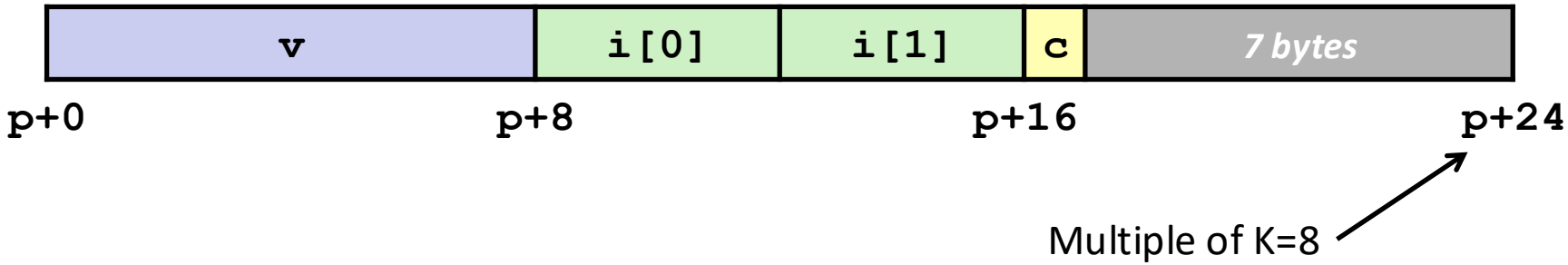


Meeting Overall Alignment Requirement

- Entire struct must be a multiple of its largest element

- For largest alignment requirement K
- Overall structure must be multiple of K
  - Trailing padding

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



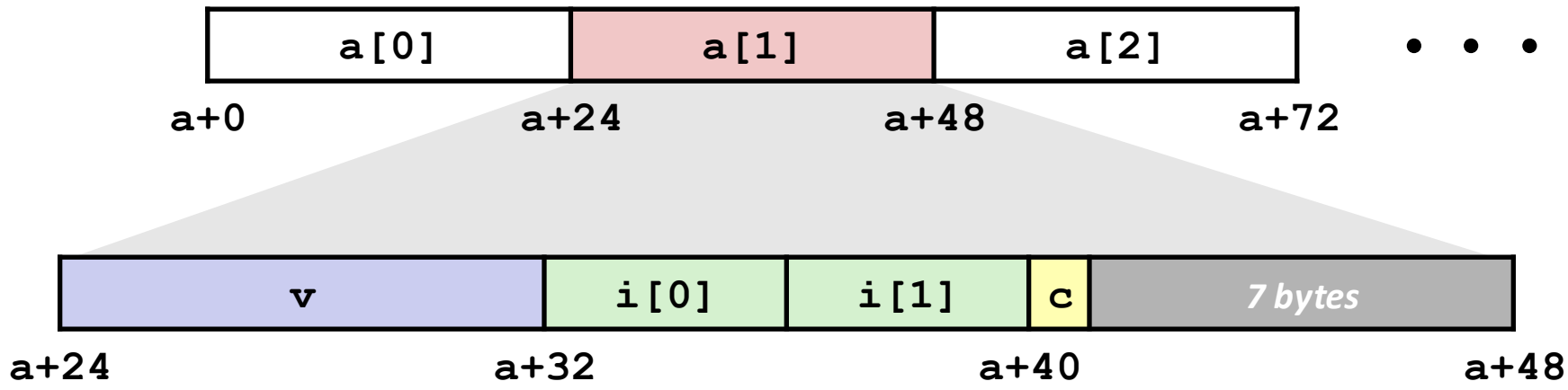
Arrays of Structures

- Arrays are the reason for the overall length requirement
  - Each struct must start at a multiple of its largest member. This means the member is aligned

- The compiler adds trailing padding even without array declaration

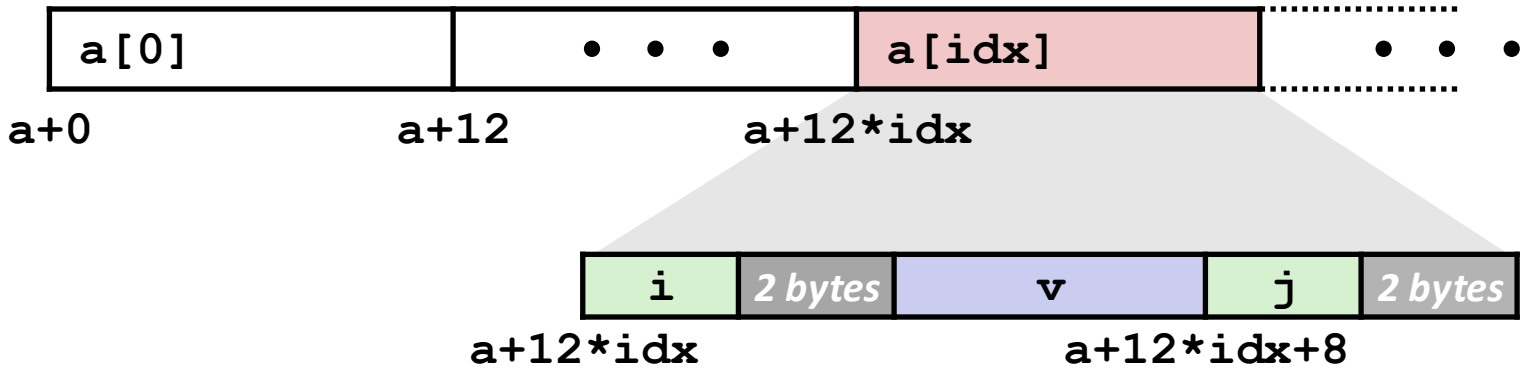
doubles are 8 bytes!

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```



- `sizeof(S3)=12`, including padding

- Compute array offset `12*idx`
- Element `j` is at offset 8 within structure

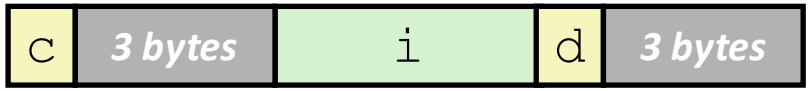
```
short get_j(int idx)
{
  return a[idx].j;
}
```

- Assembly contains displacement `a+8`
  - Compile-time constant resolved during linking

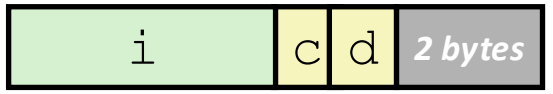
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Saving Space

-
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```




```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



- Effect: saved 4 bytes of memory
- C compilers cannot do this automatically!
  - They have to preserve field ordering
  - Programmers must do it manually
  - Other languages aren't bound to preserve ordering. Rust may reorder for you

Practice

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
} rec;
```

Practice

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
} rec;
```

2 bytes for **a**

(2 bytes for padding)
4 bytes for **b**

(no padding needed, 8-aligned)
24 bytes for **c**

(no padding needed, aligned)
1 byte for **d**

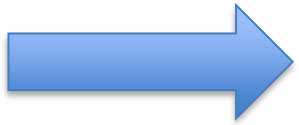
(7 bytes padding after struct)
= 40 bytes total

(Could have been 32 bytes if reordered)

Practice

- Reorder!

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
} rec;
```



```
typedef struct {
  char* c[3]; // offset 0-23 (8-aligned)
  int b;  // offset 24-27 (4-aligned)
  short a; // offset 28-29 (2-aligned)
  char d; // offset 30
} rec;    // 1 byte of padding -> 32 bytes
```

*****ALIGN REQUIREMENT! Each field has to sit at an offset that's a multiple of it's own size***

Structs in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

- Assume struct starts at address 0x1040
- What is the layout of fields within the struct?

Structs in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

- Assume struct starts at address 0x1040
- What is the layout of fields within the struct?
  - Two bytes for `short a`, then two bytes of padding
  - 4 bytes for `int b`
    3*8 bytes for `char* c[3]` (different shades of green for each element)
  - 1 byte for `char d`, then seven bytes of padding

Structs in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

- Assume struct starts at address 0x1040
- What is the layout of fields within the struct?
- What is the value of struct.b?
  - 0x93CDA0EF (little endian)

Topics

- Struct Layout
- Struct Padding and Alignment
- Procedures
  - C Memory Layout (review)
  - X86-64 Calling Convention

Topics

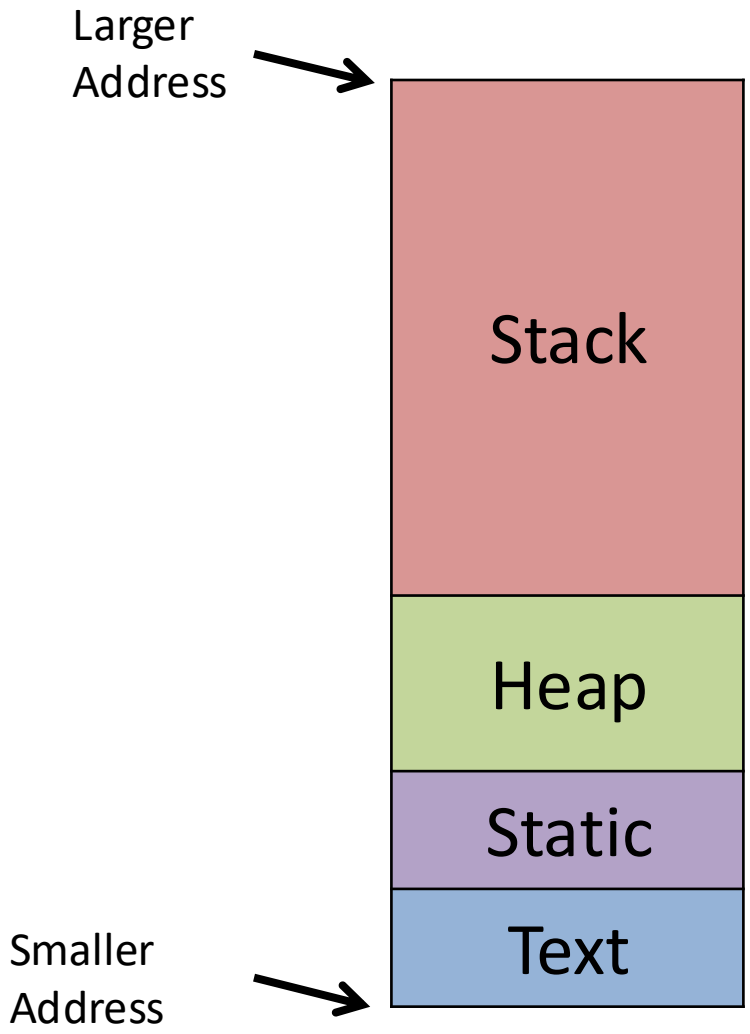
- Struct Layout
- Struct Padding and Alignment
- Procedures
  - **C Memory Layout (review)**
  - X86-64 Calling Convention

Virtual Memory

- ***Virtual memory*** is an abstraction that provides each process with the illusion that it has exclusive use of main memory
  - Each process has the same uniform view of memory known as its ***virtual address space***
- The virtual address space consists of well-defined areas:
  - **Stack** – memory used for local variables within functions and information needed to go to and return from functions. This region expands and contracts during program execution (grows downward).
  - **Heap** – memory used for dynamic memory allocation during program runtime. This region expands and contracts during program execution (grows upward).
  - **Static (Data)** – all data allocated statically outside of functions (global and static variables)
  - **Text** – where code lives, and some variables that are declared with the `const` keyword. This segment cannot be written to by the program.

C memory layout

- Stack Section
  - Local variables (that aren't in registers)
  - Function arguments
- Heap Section
  - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
  - Global variables
  - Static function variables
  - String literals
- Text Section (a.k.a Code Section)
  - Program code

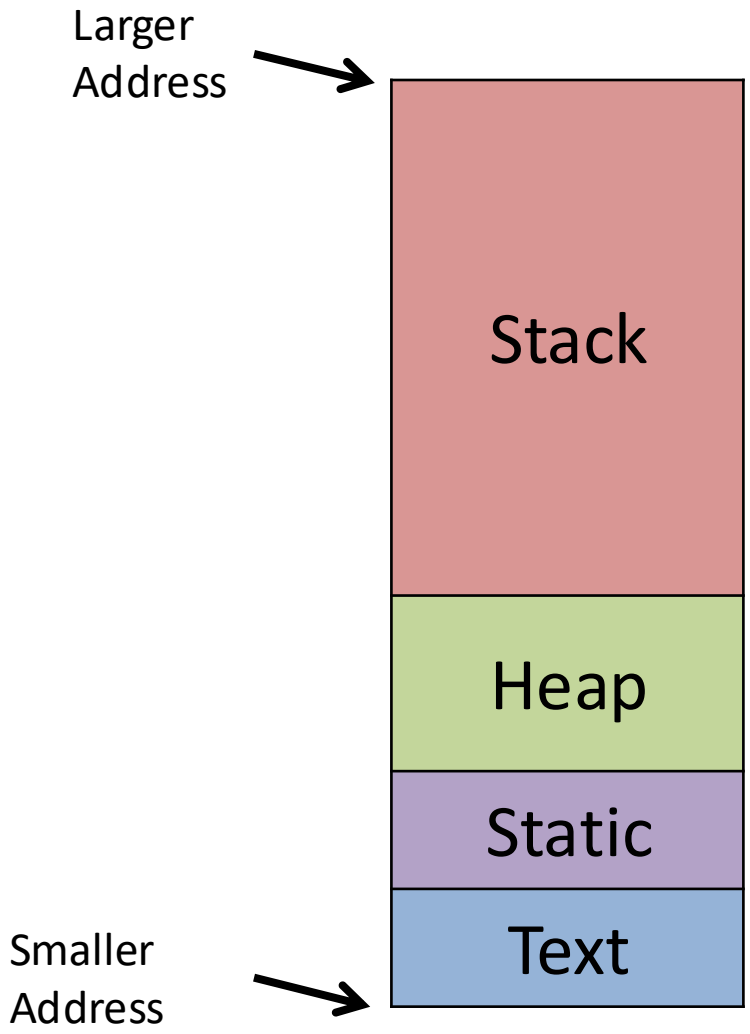


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

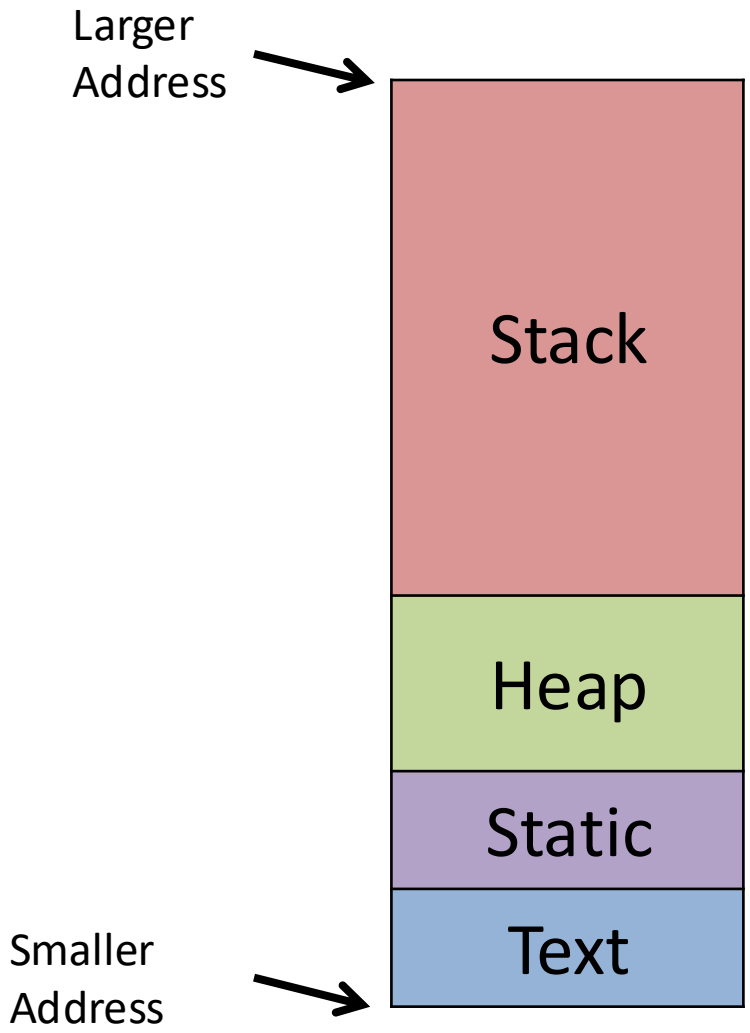


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

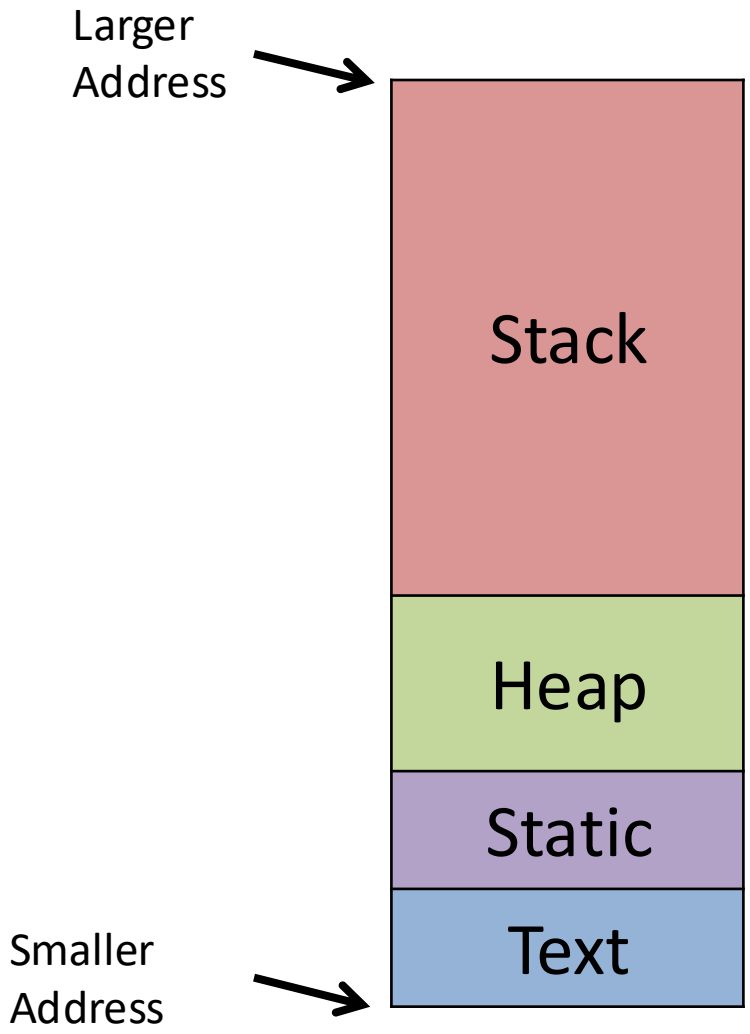


C memory layout

```
char glob_str[80] = {0};
void func short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

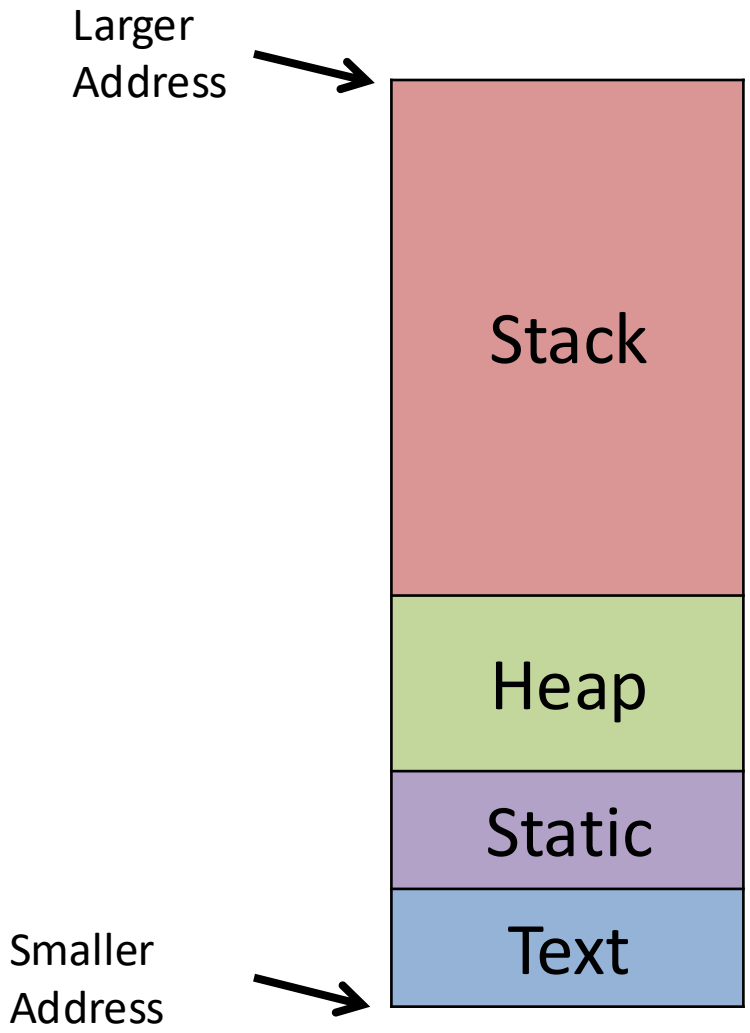


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

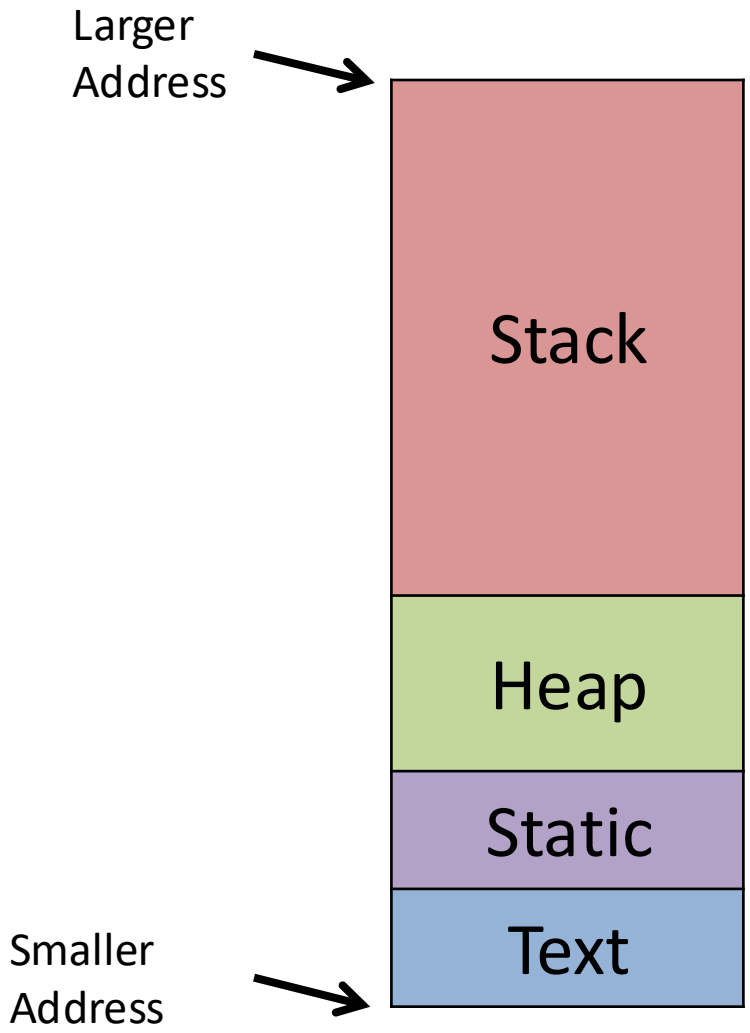


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

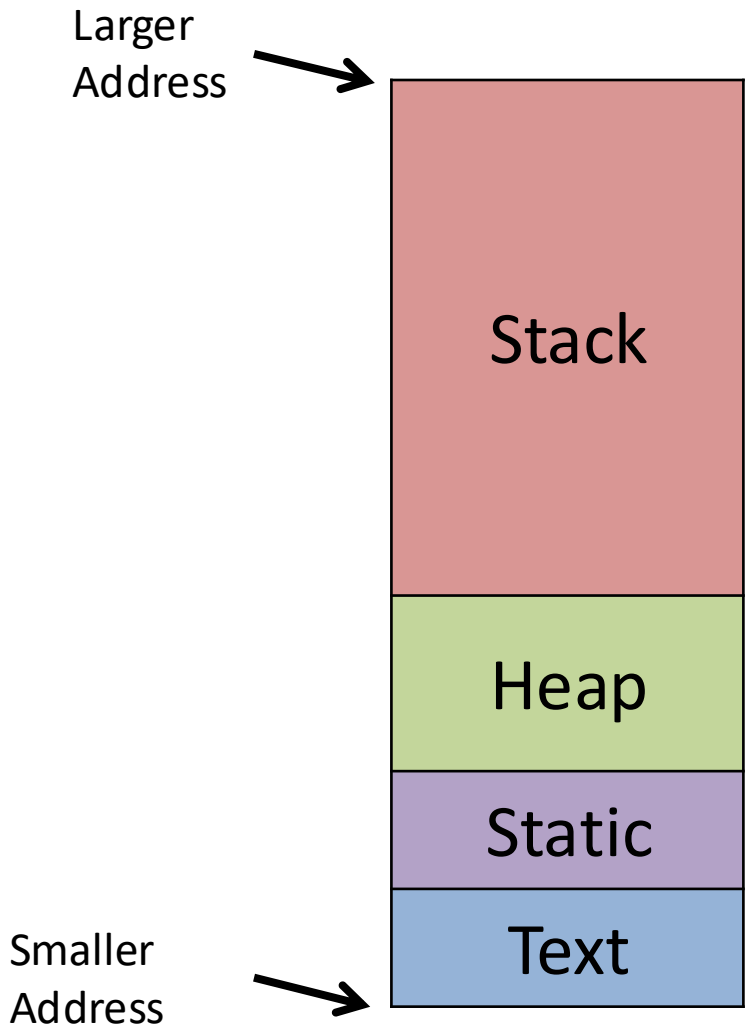


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

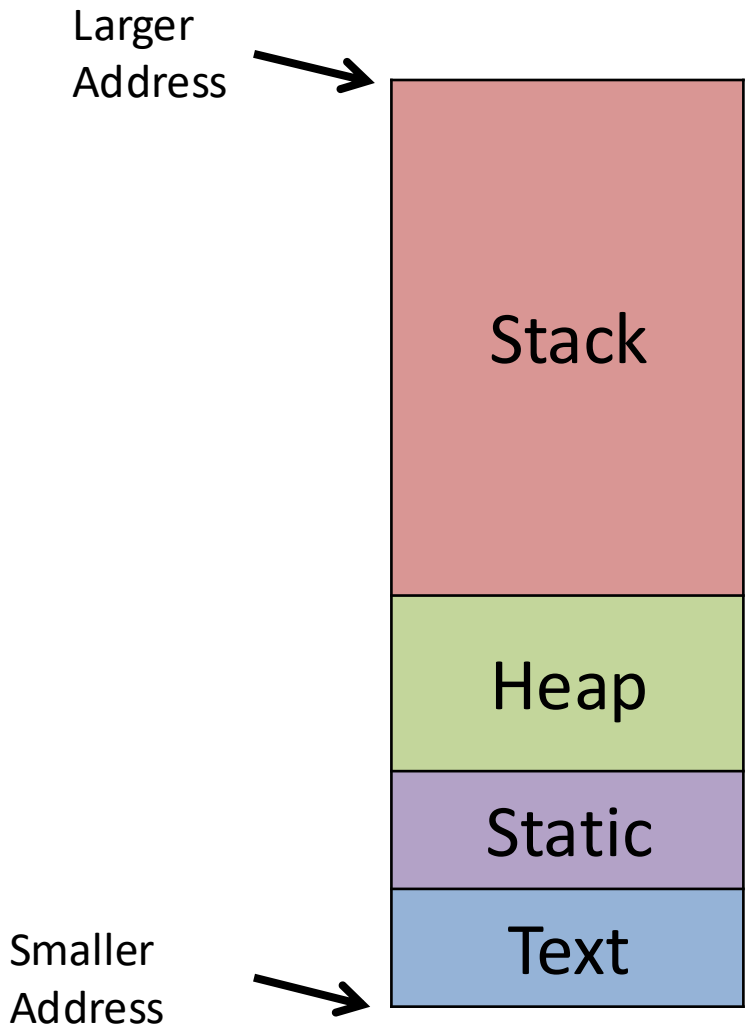


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

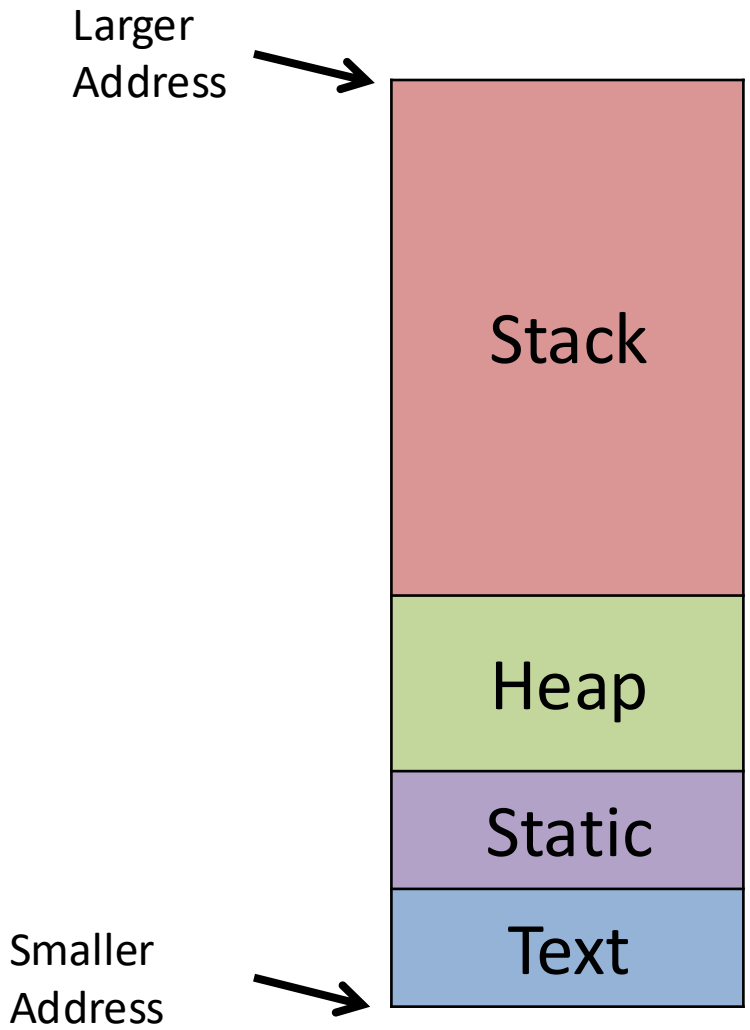


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

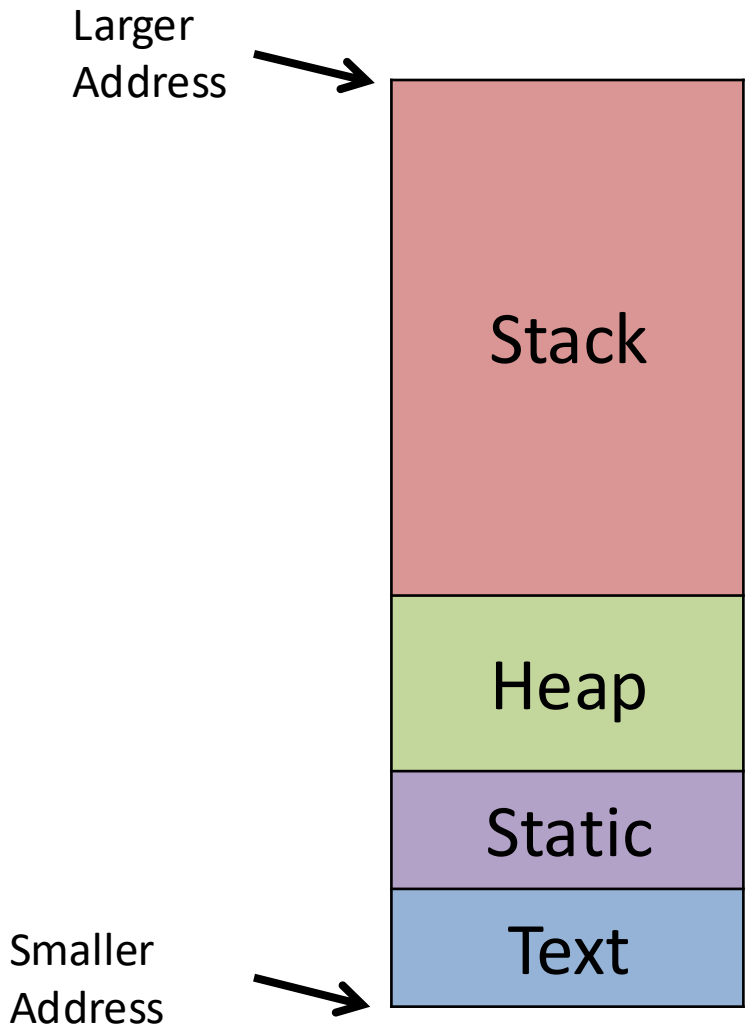


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```

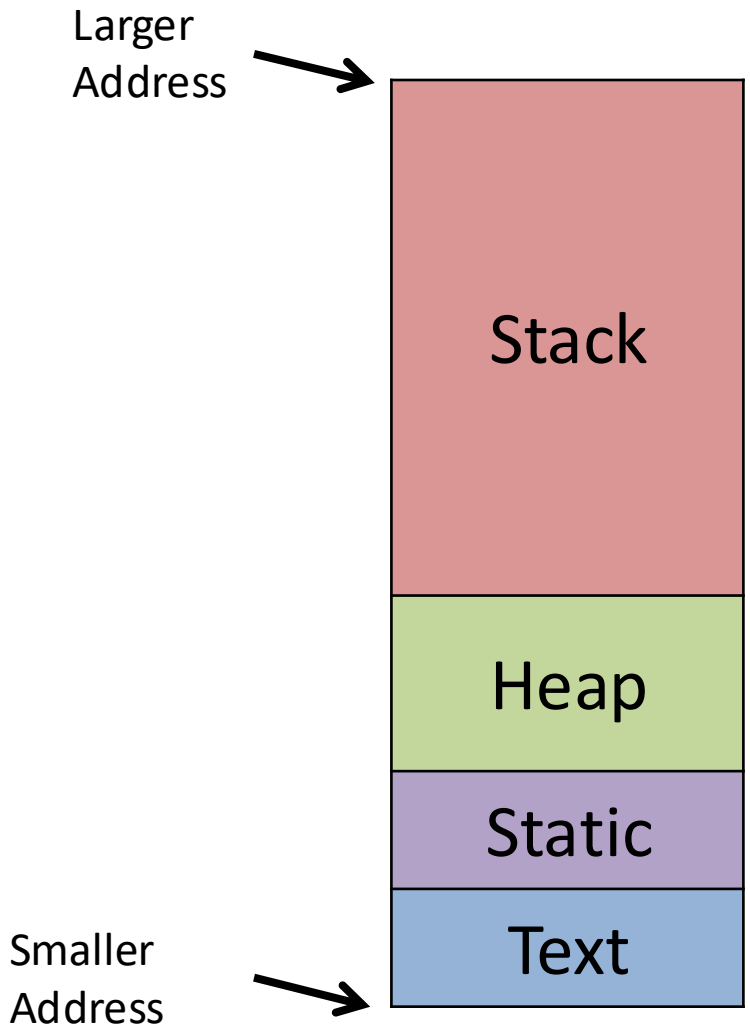


C memory layout

```
char glob_str[80] = {0};
void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

    printf("Hello CS213\n");
}
```



C memory layout

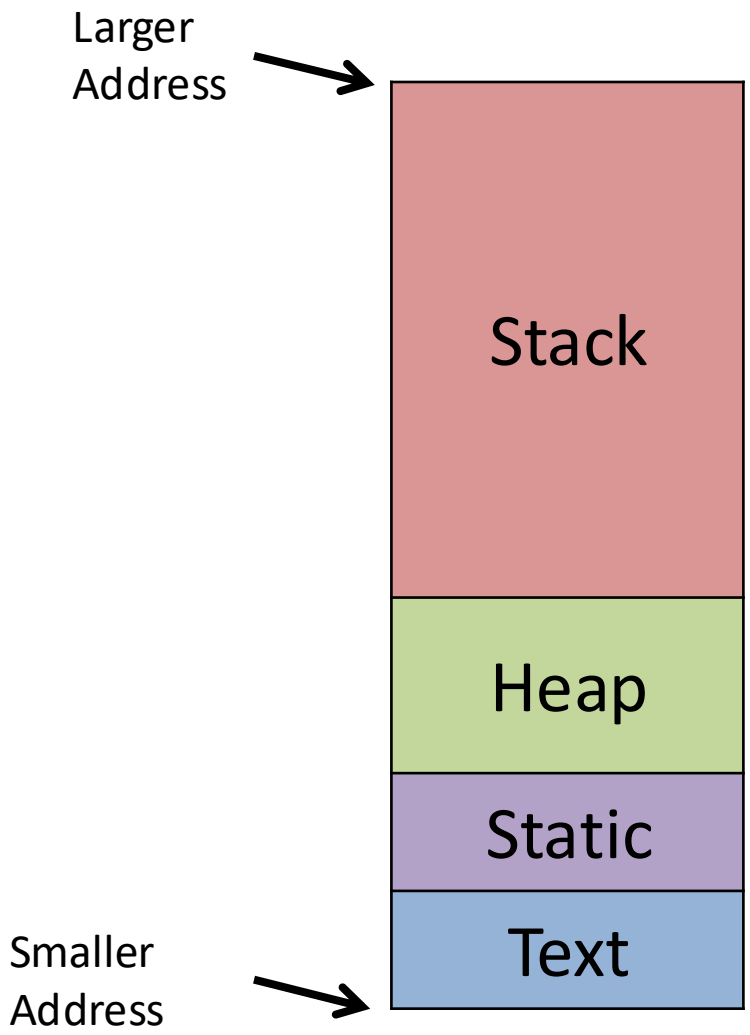
```
char glob_str[80] = {0};

void func(short b, int* f) {
    static int c = 3;

    char* d = "Test";
    int* e = malloc(sizeof(int));

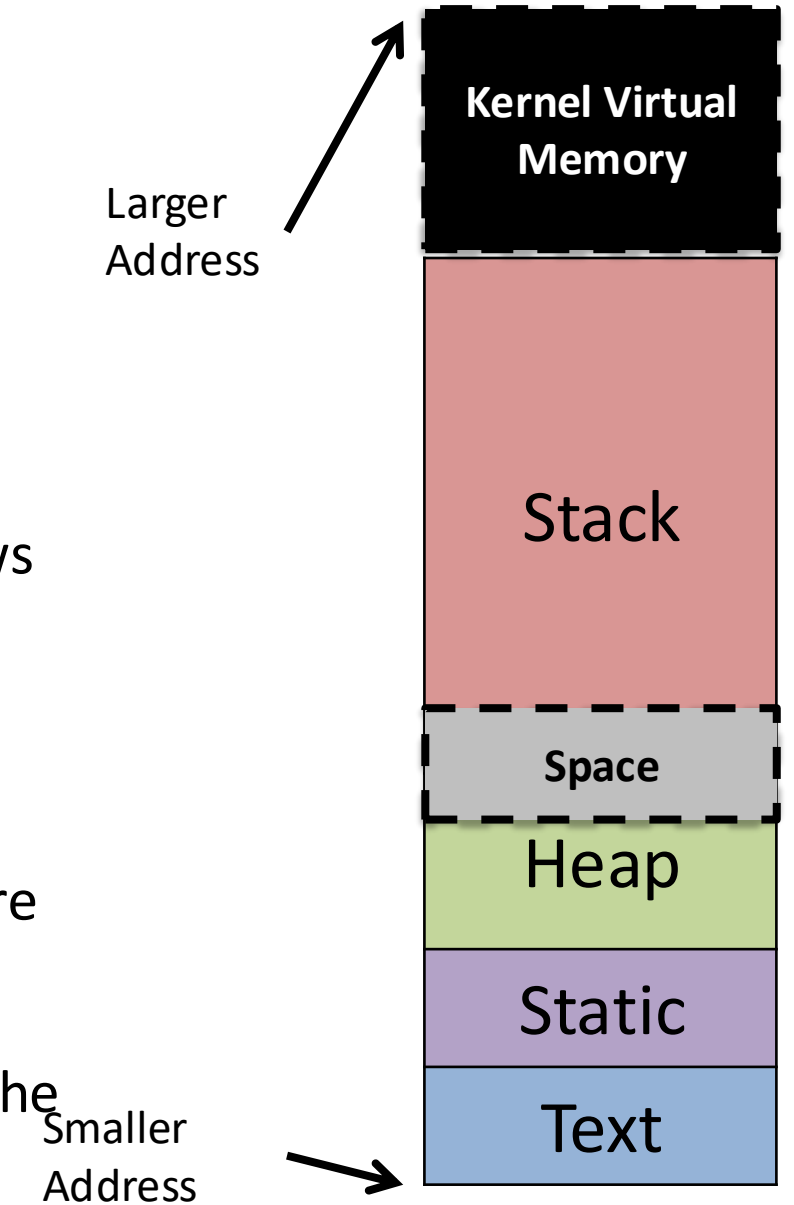
    printf("Hello CS213\n");
}
```

Assembly instructions generated by this function go in the Text section



Important notes!

- Illustration to the right gives relative locations for memory allocation of a C program – not exact, and not to scale
  - Stack and heap grow dynamically (down and up respectively)
  - There is actually a region of space in between that allows expansion / contraction (and holds other stuff too)
- ***Kernel virtual memory*** is found at the top region of the address space (invisible to user code)
  - ***Kernel*** – the portion of the OS that is always resident in memory. It is privileged code that runs with full hardware access, managing the CPU, memory, devices, and processes.
  - When an application program requires some action by the OS (e.g. read/write to file), a special ***system call*** instruction is performed that transfers control to kernel. Kernel performs action and returns to the program.



Interacting with data sections in assembly

- Stack
  - Stack pointer is saved in **`%rsp`** (remember, this was a special register saved for a special purpose!) and can be moved as needed
- Heap
  - C library (malloc) handles this above the machine level
  - i.e. from the machine point of view, there is no heap
- Static
  - Arbitrary pointers to memory can be created and used
    - With memory addressing instructions
    - Might references global values
- Text
  - Assembly code is placed here automatically
  - Labels are just addresses within the Text section

Question

Which sections are absolutely required, and which aren't?

- Text
- Static
- Heap
- Stack

Question

Which sections are absolutely required, and which aren't?

- Text: necessary! Holds the code!
- Static: only necessary if using global variables or strings
- Heap: only necessary if you heap-allocate (using malloc or automatically while using other languages)
- Stack: necessary if you use variables or call functions
  - Probably always needed unless you write in assembly….

Pushing and Popping Stack Data

- A stack is a data structure where values can be added (***push***) or deleted (***pop***), but only through the "Last in, first out" discipline
  - Holds variables, function calls…
- Popped value will be a value that was most recently pushed and is still on the stack
- Can be implemented as an array where we always insert and remove elements from one end of the array (***top*** of the stack)
  - Stack top has lower address (stack grows downward)
  - Stack bottom has higher address
- The stack pointer `%rsp` hold the address of the top stack element

If we wanted to push `%rbp` to the stack:

```
subq $8, %rsp   (decrement stack pointer by 8 for quad word)
movq %rbp, (%rsp)
```

Instruction	Effect	Description
pushq *S*	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← *S*	Push quad word
popq *D*	*D* ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Figure 3.8 Push and pop instructions.

Push and pop instructions take a single operand

Illustration of Stack Operation

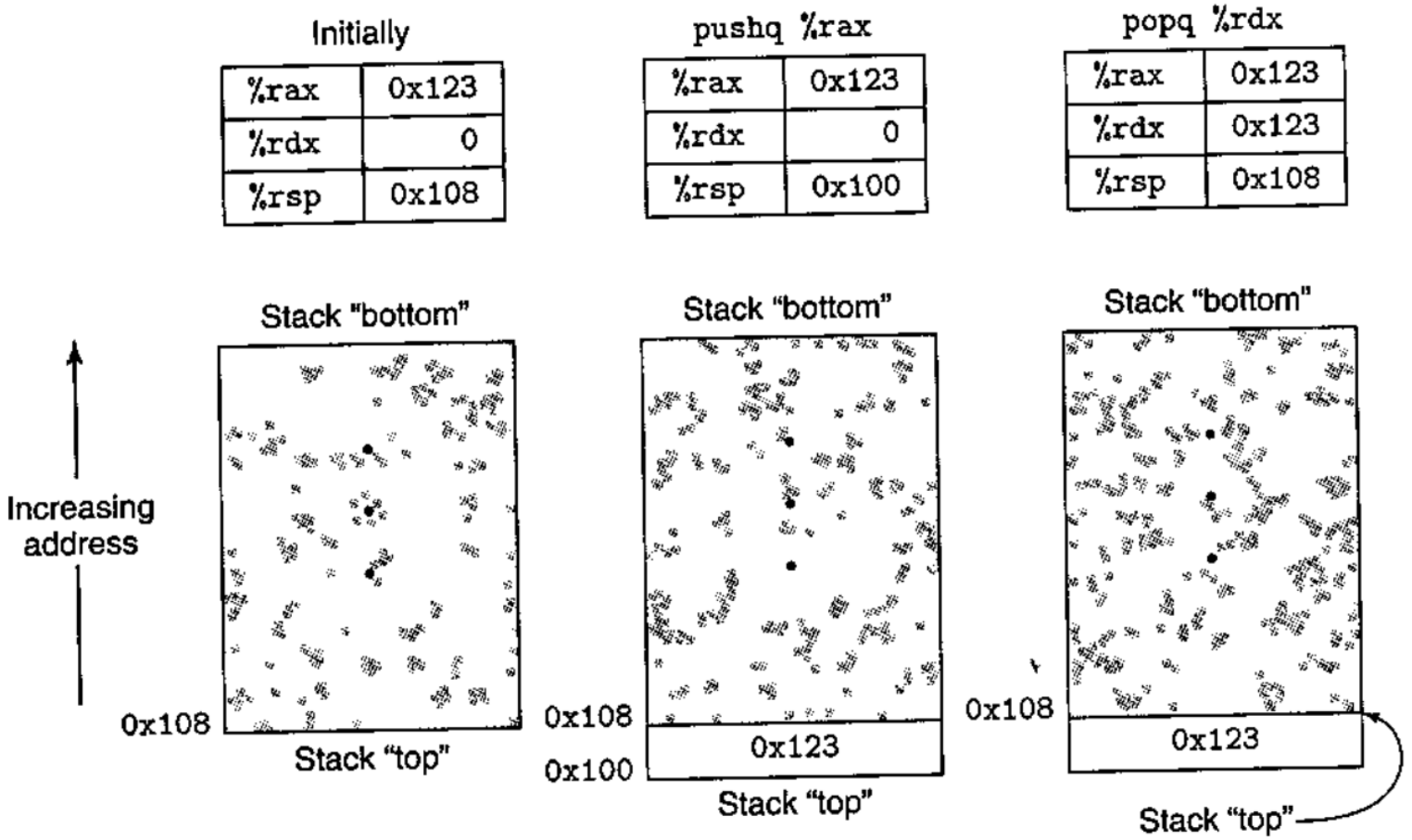


Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %rsp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

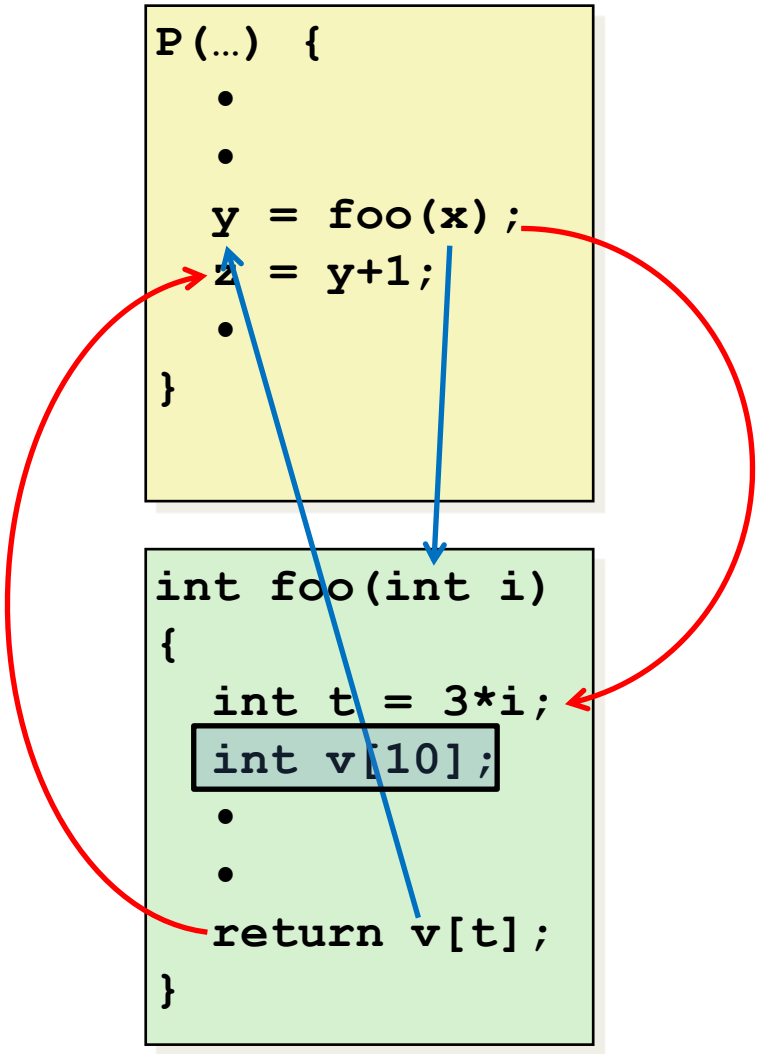
Topics

- Struct Layout
- Struct Padding and Alignment
- Procedures
  - C Memory Layout (review)
  - **X86-64 Calling Convention**

Mechanisms in Procedures

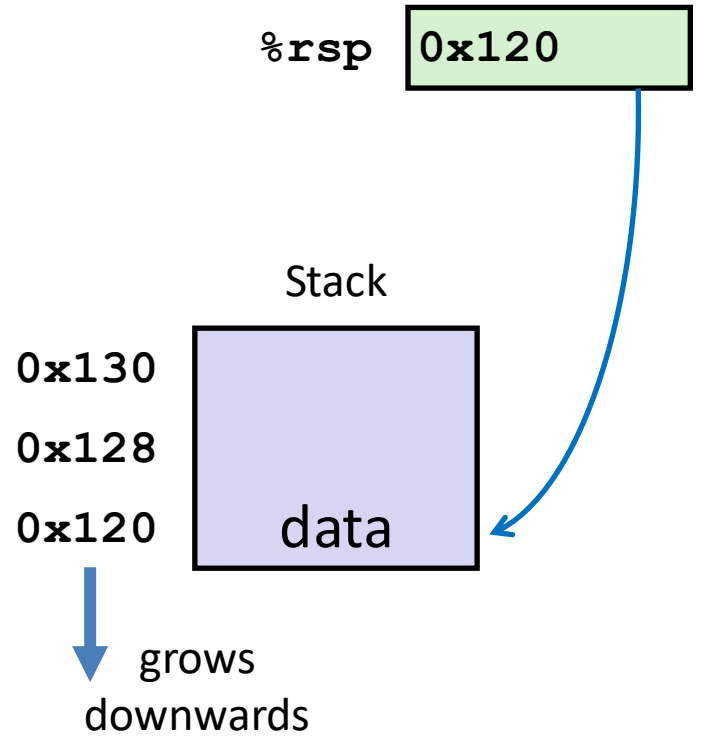
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Local memory management
  - Allocate during procedure execution
  - Deallocate upon return

- No one instruction does all that
  - Need instructions for each
- The stack is the key to all 3 of these!



The Stack in Assembly

- Chunk of memory that code can use
  - Think of it like scratch paper for taking notes
- `%rsp` points at most recently used position
- Stack grows downward
  - Subtract from `%rsp` to claim stack memory
  - Add to `%rsp` to release previously used memory
- Stack is used for multiple purposes
  - Scratch space if not enough registers
  - Function calls (extra arguments, return addresses)



Function control flow

- Two assembly instructions call functions and return from them
  - **`callq` *`label`***
    - Subtracts 8 from `%rsp` (allocates 8 bytes on the stack)
    - Saves the address of the *next* instruction to the stack at `%rsp`
    - Jumps to label
  - **`retq`**
    - Loads an address from the stack at `%rsp`
    - Adds 8 to `%rsp` (frees 8 bytes on the stack)
    - Jumps to loaded address

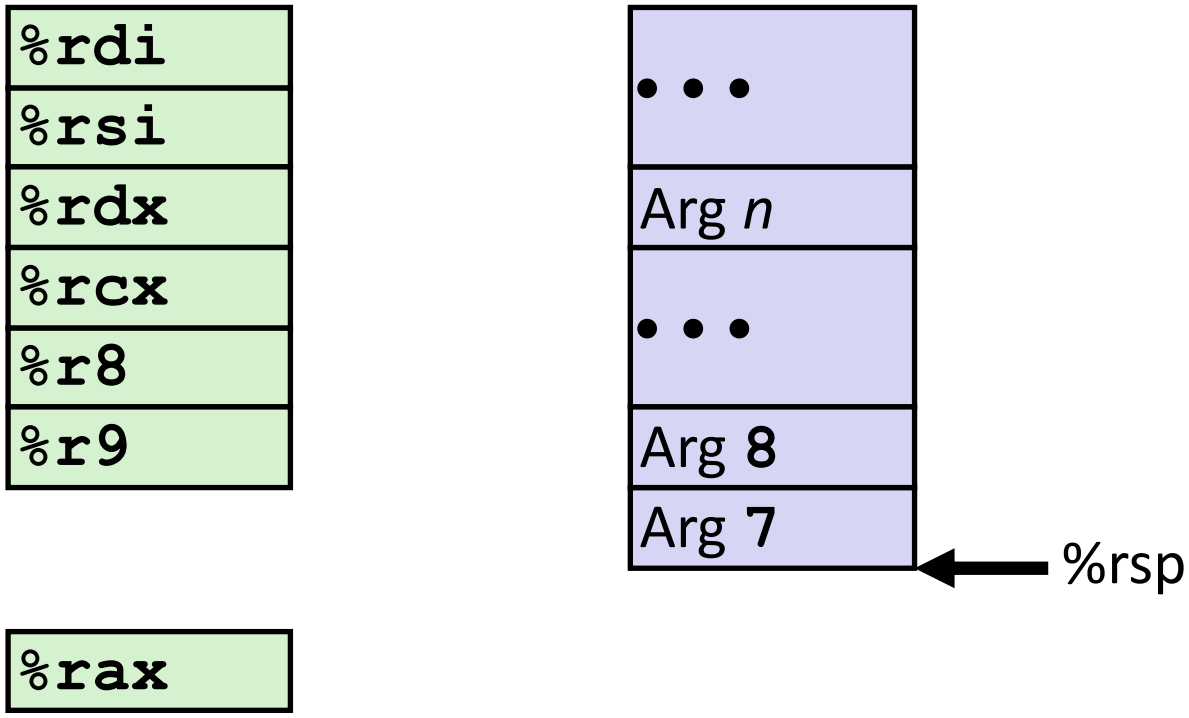
```
400544: call  400550 <mult2>
400549: mov    %rax,(%rbx)
```

Return address: **0x400549**

Just **`call`** and **`ret`** are fine by the way, the **`q`** is assumed (there is no other option)

Function data flow in x86-64

- First 6 arguments are in registers
  - `%rdi` is first argument
- Next **`n`** arguments are on the stack
  - Too many arguments hurts performance
- Return value is in **`%rax`**



Code Example

```
void multstore(long x, long y,
               long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
long mult2 (long a, long b){
  long s = a * b;
  return s;
}
```

Which registers hold arguments?

No return

%rdi %rsi %rdx

```
void multstore(long x, long y,
               long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: movq   %rax,(%rdx)      # Store at *dest
  40054d: retq                    # Return
```

%rax %rdi %rsi

```
long mult2 (long a, long b){
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax       # a
  400553:  imulq  %rsi,%rax       # a * b
  400557:  retq                   # Return
```

Control flow

Calls the mult2 function

```
void multstore(long x, long y,
               long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: movq   %rax,(%rdx)      # Store at *dest
  40054d: retq                    # Return
```

Return instructions

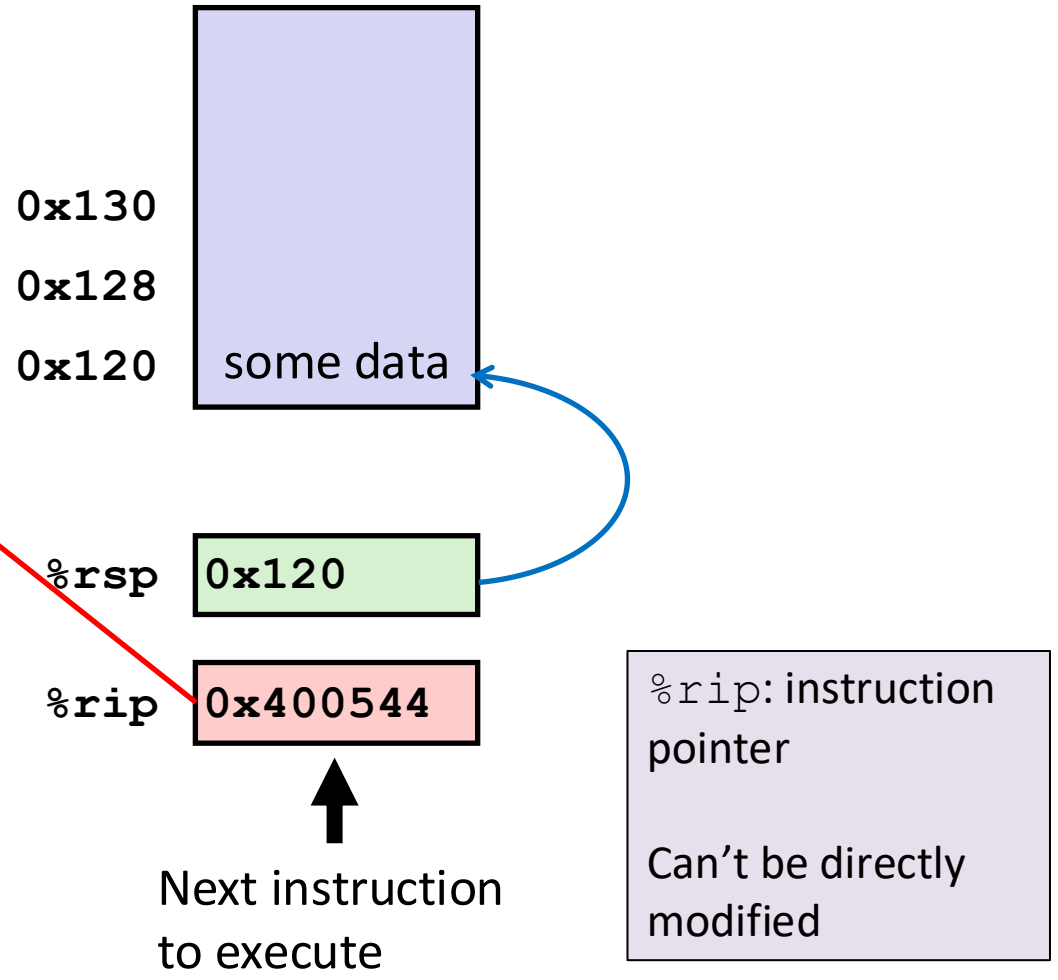
```
long mult2 (long a, long b){
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax       # a
  400553:  imulq  %rsi,%rax       # a * b
  400557:  retq                   # Return
```

Control Flow Example about to execute `callq`

```
0000000000400540 <multstore>:
   •
   •
   400544: callq  400550 <mult2>
   400549: movq   %rax,(%rdx)
   •
   •
```

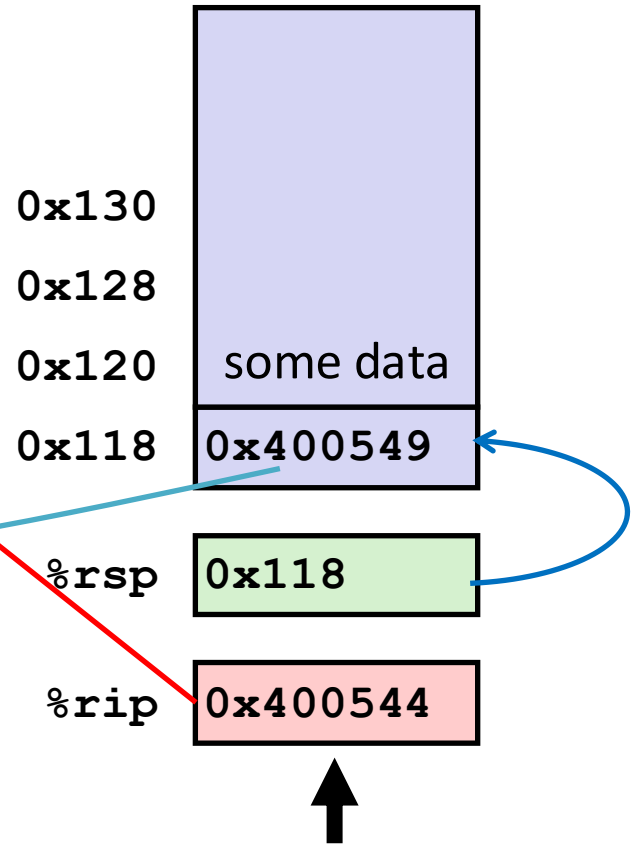
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  retq
```



Control Flow Example
`callq` step 1

```
0000000000400540 <multstore>:
   •
   •
   400544: callq  400550 <mult2>
   400549: movq   %rax,(%rdx)
   •
   •
   •
```

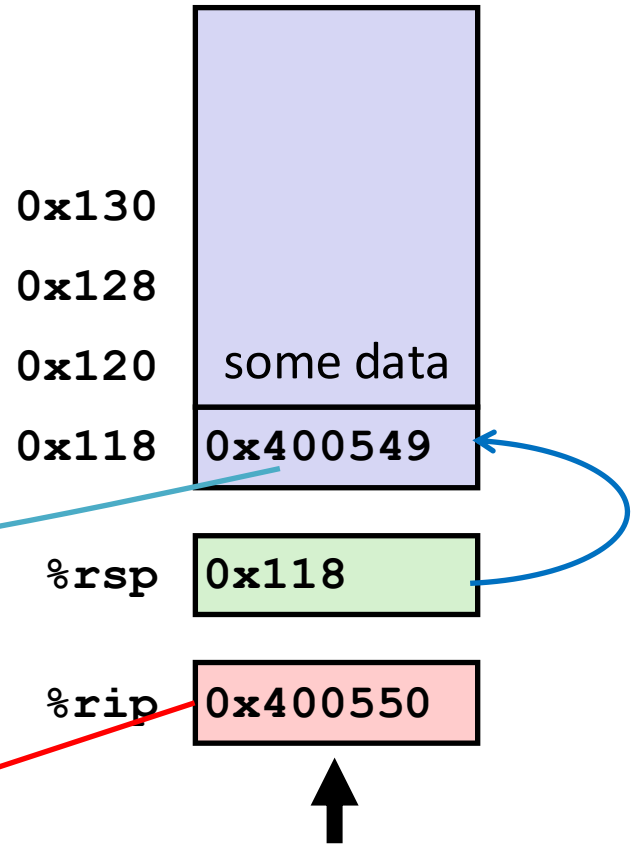
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  retq
```



Control Flow Example
`callq` step 2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rdx)
  •
  •
  •
```

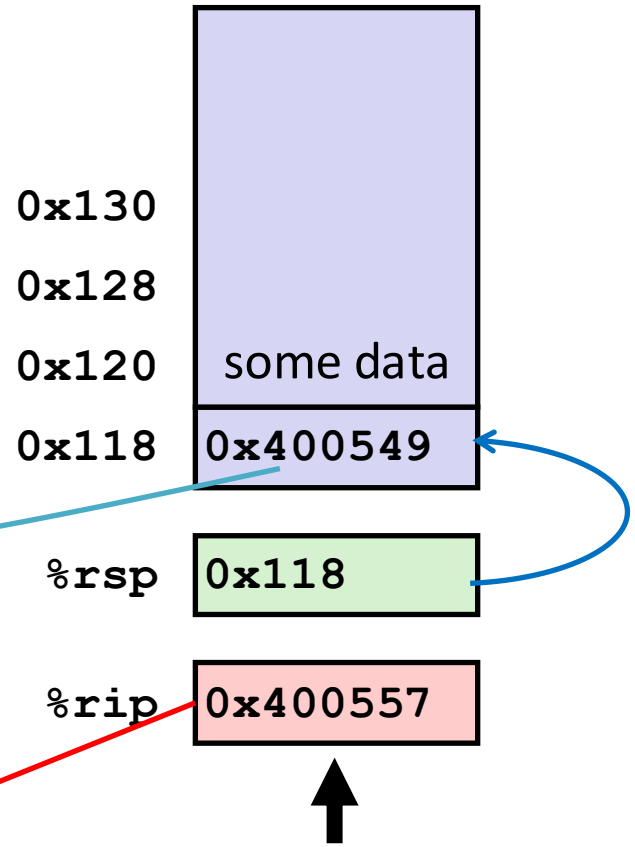
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example about to execute `retq`

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rdx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```

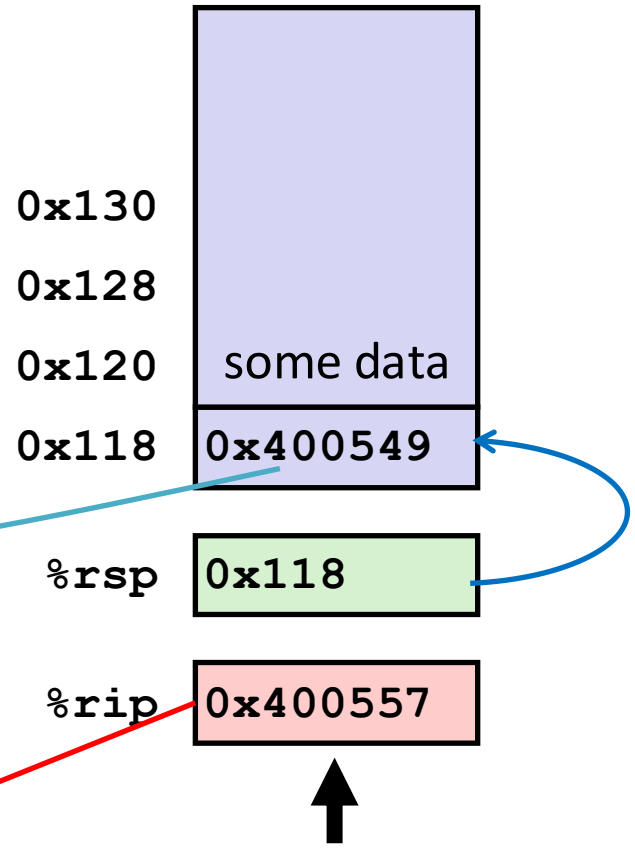


Q: What is the address of the instruction we execute after **`retq`**?

Control Flow Example about to execute `retq`

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rdx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```

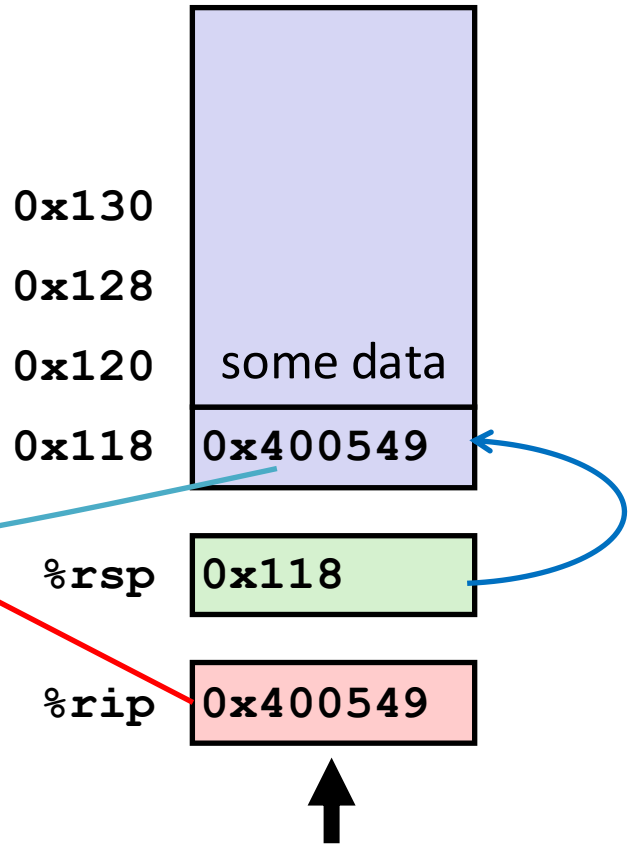


Q: What is the address of the instruction we execute after **`retq`**? **A:** 0x400549

Control Flow Example
`retq` step 1

```
0000000000400540 <multstore>:
   •
   •
   400544: callq  400550 <mult2>
   400549: movq   %rax,(%rdx)
   •
   •
   •
```

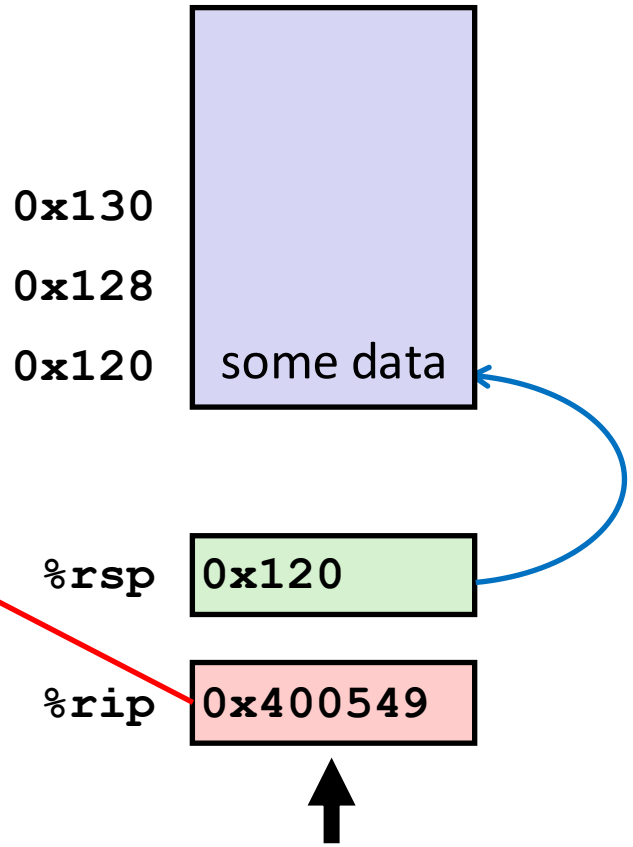
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  retq
```



Control Flow Example
`retq` step 2

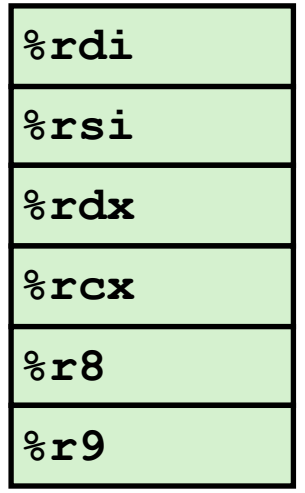
```
0000000000400540 <multstore>:
   •
   •
   400544: callq  400550 <mult2>
   400549: movq   %rax,(%rdx)
   •
   •
```

```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  retq
```

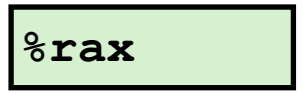


Question

- How did we decide how many registers to use for arguments and return values?

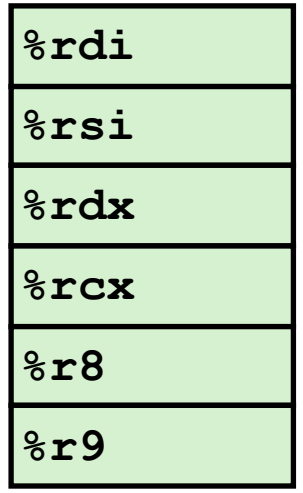


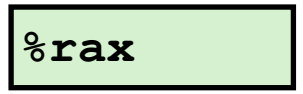
- Do all functions have to use this same convention?



Question

- How did we decide how many registers to use for arguments and return values?
  - Testing lots of real-world programs
  - Many style guides suggest you use four or less arguments
  - x86 (32-bit) only had four arguments
    - x86-64 added two more
  - C only has one return result, so one register is fine





- Do all functions have to use this same convention?
  - All functions within a program must, or they won't work
  - Different programs, or different OSes, could choose different