CSCI 2330

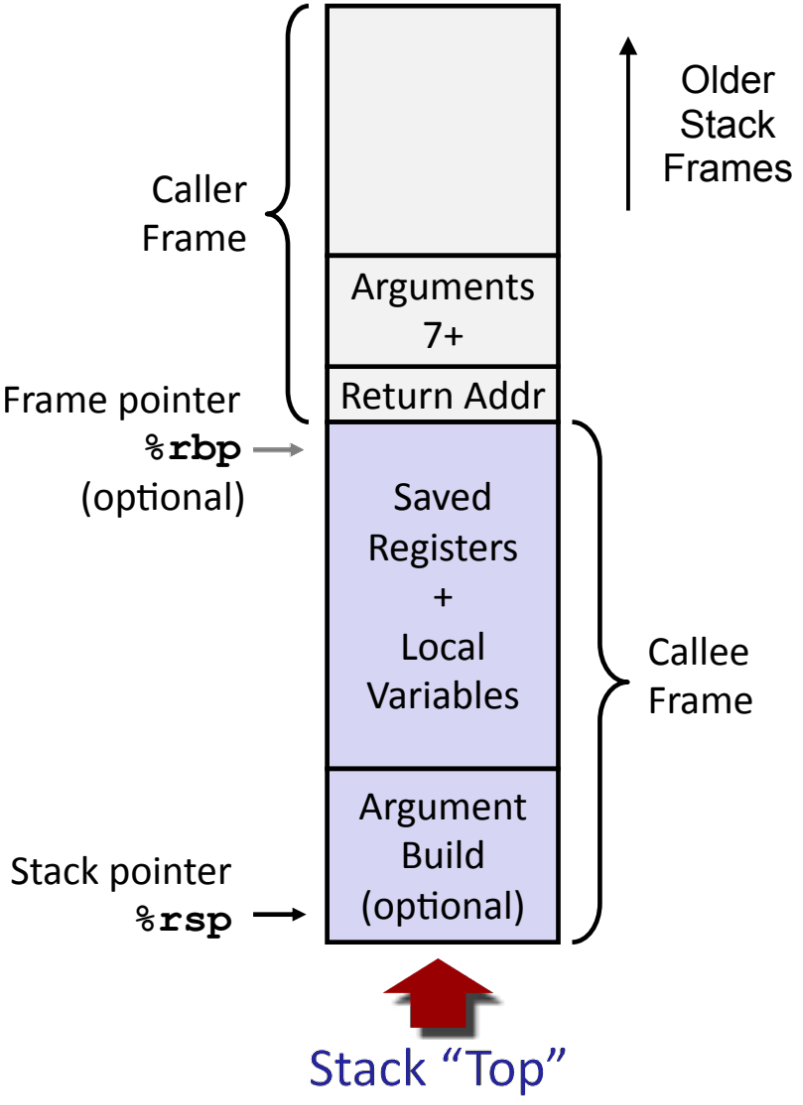

did you know....
fb.me/yuva.krishna.memes
in order to play the role of an insane and mentally depressed person in the movie "Joker", Joaquin Phoenix started learning programming in assembly.
ProgrammerHumor.io

TODO

- Assignment 3 assigned due October 28

Data organization in stack frames

- Suppose we are in function A() about to call function B(). What do we first push onto the stack?
  - Any arguments past 6
  - The return address
- These are at the end of A's stack frame
- Once we jump to B's address, we begin B's stack frame…


Older
Stack
Frames
Caller
Frame
Arguments
7+
Return Addr
Frame pointer
%rbp
(optional)
Saved
Registers
+
Local
Variables
Callee
Frame
Argument
Build
(optional)
Stack pointer
%rsp
Stack "Top"

Quiz

- Can the current (topmost) stack frame ever have a return address in it?

Local variables

- We have alluded to the idea that local variables can live in the stack
- But in our previous x86 code, we only considered local variables that lived in registers
  - Similar to how we only considered arguments that lived in registers
- But we can also put local variables on the stack
  - Especially we ran out of registers!

Local variables

- We must reserve room for stack variables in our stack frame
- We just have to grow the stack first to reserve space for them
  - And shrink the stack before we return
- Consider the following *leaf* function (a function that does not call any other function)…

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

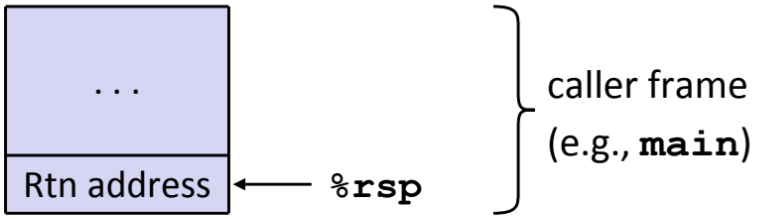
Local variables

- Now let’s take a look at a function that calls `incr()`…

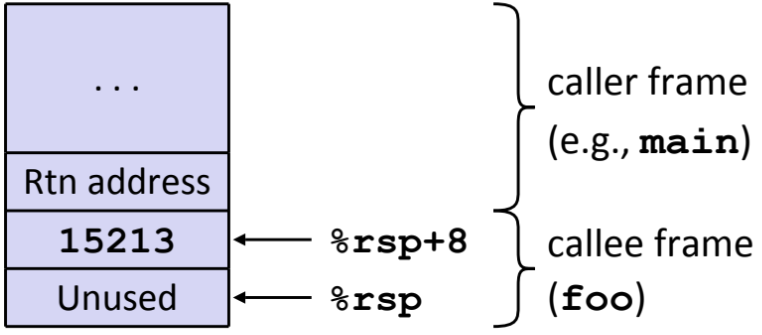
```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure



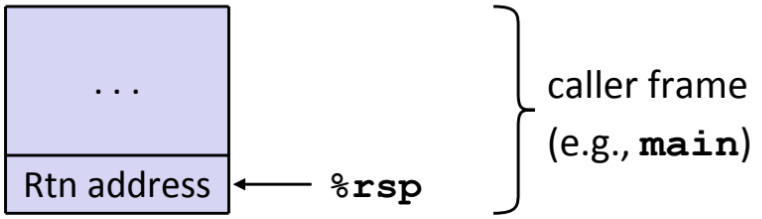
Local variables

- Note: we *grow* the stack by *subtracting* from `%rsp` 🥲
- Why did the compiler reserve more stack space than it needs?
  - For *alignment* – another x86-64 convention

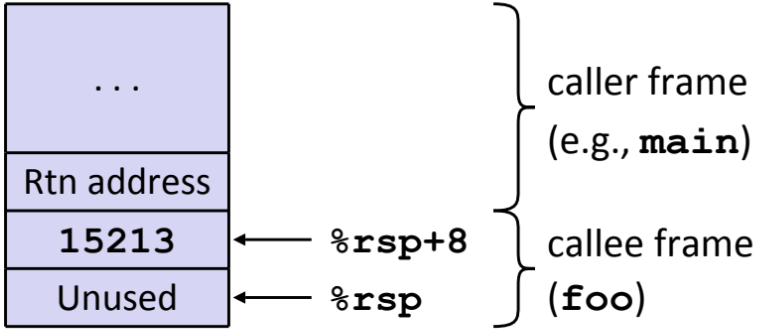
```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



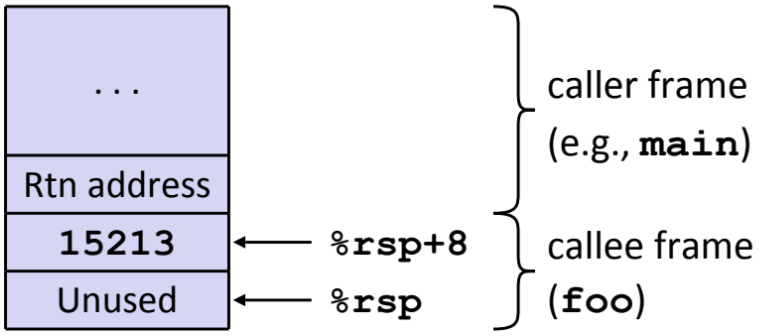
Resulting Stack Structure



```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

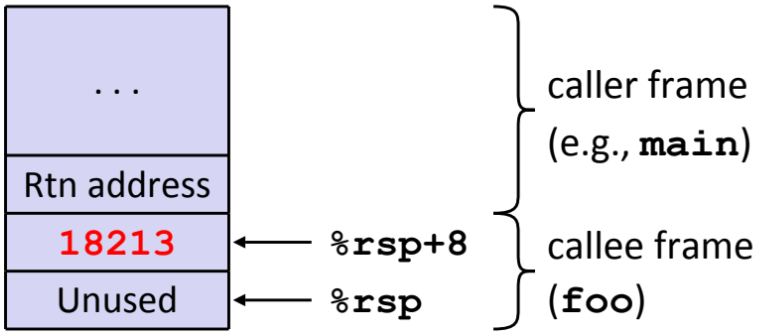


Register	Use(s)
%rdi	&v1
%rsi	3000

```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure
(after **incr** return)



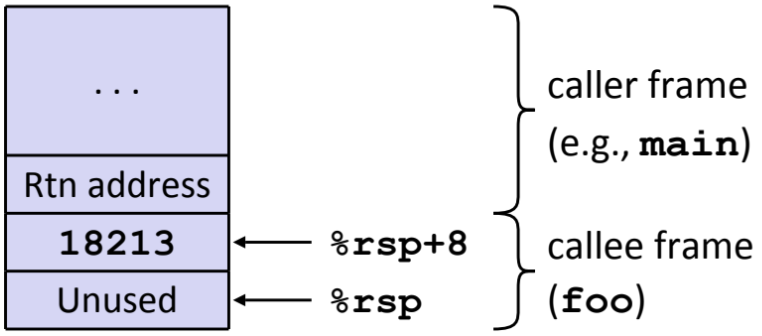
Register	Use(s)
%rdi	&v1
%rsi	3000

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

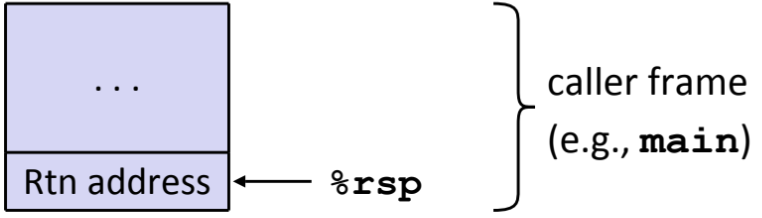
```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

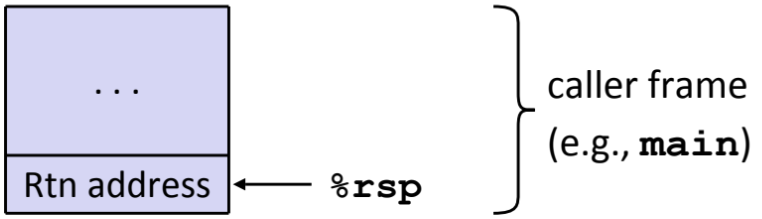
Updated Stack Structure



```
long foo() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

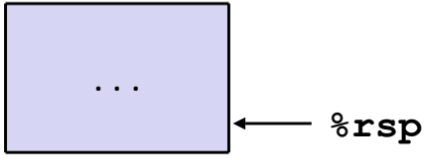
```
foo:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



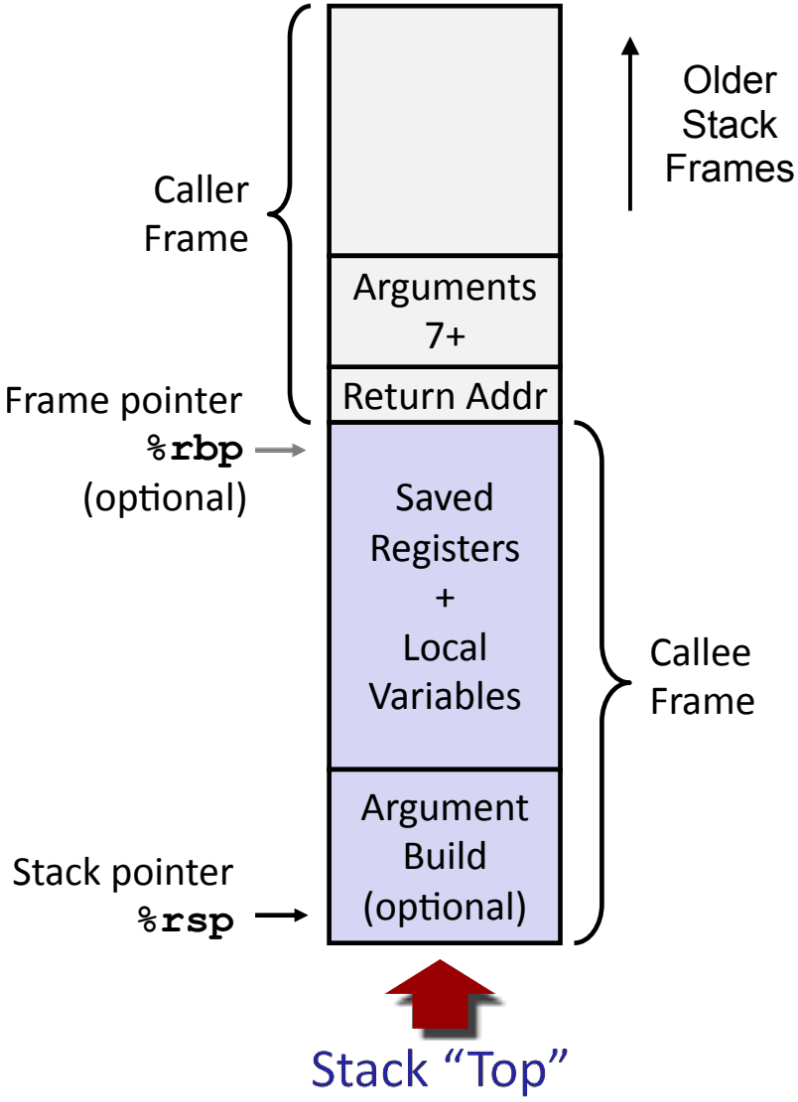
Register	Use(s)
%rax	Return value

Final Stack Structure



One more thing left

- We've covered everything in this figure!
- Except for:
  - *saved registers*



Suppose we have two functions

- Alice – the caller
- Bob – the callee
- Alice does a lot of work using temporary registers
- Alice calls Bob to do some more work…

Problem

```
alice:
  movq X, %reg # compute X
  callq bob
  movq %reg, ...

?
```

Problem

```
alice:
  movq X, %reg # compute X
  callq bob
  movq %reg, ...
```

- `bob` might clobber (overwrite) `%reg`

Problem

```
alice:
  movq X, %reg # compute X
  callq bob
  movq %reg, ...
```

- `bob` might clobber (overwrite) `%reg`
  - Or someone that `bob` calls might

Solution: Saved Registers

- Save register contents by pushing them to the stack
- Sometime later, restore by popping them off

Solution: Saved Registers

- Save register contents by pushing them to the stack
- Sometime later, restore by popping them off
- OK so who saves the registers? Alice or Bob?

Caller-saved registers

- This is where Alice saves the registers
- Push the register before calling Bob
- Pop the register after Bob returns
- This way Bob can safely do whatever Bob wants with the register…

```
alice:
  movq X, %reg     # compute X
  callq bob
  movq %reg, ...   # use X
```



Could overwrite `%reg`!

Caller-Saved `%reg`

```
alice:
  movq X, %reg     # compute X
  pushq %reg       # save X
  callq bob        # might change X
  popq %reg        # restore X
  movq %reg, ...   # use X
```

Callee-Saved `%reg`

```
alice:
  pushq %reg       # save caller's
  movq X, %reg     # compute X
  callq bob        # preserves X
  movq %reg, ...   # use X
  popq %reg        # restore caller
```

Caller-saved registers

- Who is this good for?
- Makes life easy for Bob
- Makes life tough for Alice

Callee-saved registers

- This is where Bob saves the registers
- Bob pushes the register before using it
- Bob pops the register when Bob is done
- This way Bob can still safely do whatever Bob wants with the register…

Callee-saved registers

- Who is this good for?
- Makes life easy for Alice
- Makes life tough for Bob

Callee-saved registers

- Who is this good for?
- ~~Makes life easy for Alice~~
- Makes life tough for Bob
- Remember: Alice is also a callee of some other function…

Callee vs. caller saved registers

- We could just pick one and use it consistently
- But we won’t
- And that’s because sometimes we might prefer one versus the other

Case 1: Alice is done

- Suppose Alice is done with the register before calling Bob
- So Alice doesn’t care if Bob clobbers the register
- In this scenario, we don’t want anyone to save the register!

Case 1

```
alice:
  movq X, %reg # compute X
  callq bob
  #movq %reg, …
  #we are now done with %reg
```

Caller vs. callee saved registers

- Caller-saved: Alice is in charge of saving the register, but Alice knows that we don’t need the register anymore, so Alice can skip saving it!
- Callee-saved: Bob is in charge of saving the register, but Bob doesn’t know Alice is done – in fact, Bob doesn’t know *anything* about his caller. Thus, Bob must still save the register.
  - Even if Alice doesn’t need the register, another caller Charlie might! You don’t know who is calling!

```
alice:
  movq X, %reg     # compute X
  callq bob
  ...              # don't need X again
```

Caller-Saved %reg

```
alice:
  movq X, %reg     # compute X
  callq bob        # might change X
  ...              # don't need X again
```



Skip save/restore!

Callee-Saved %reg

```
alice:
  pushq %reg       # save caller's
  movq X, %reg     # compute X
  callq bob        # preserves X
  ...
  popq %reg        # restore caller
```

Case 2: Bob doesn’t use it

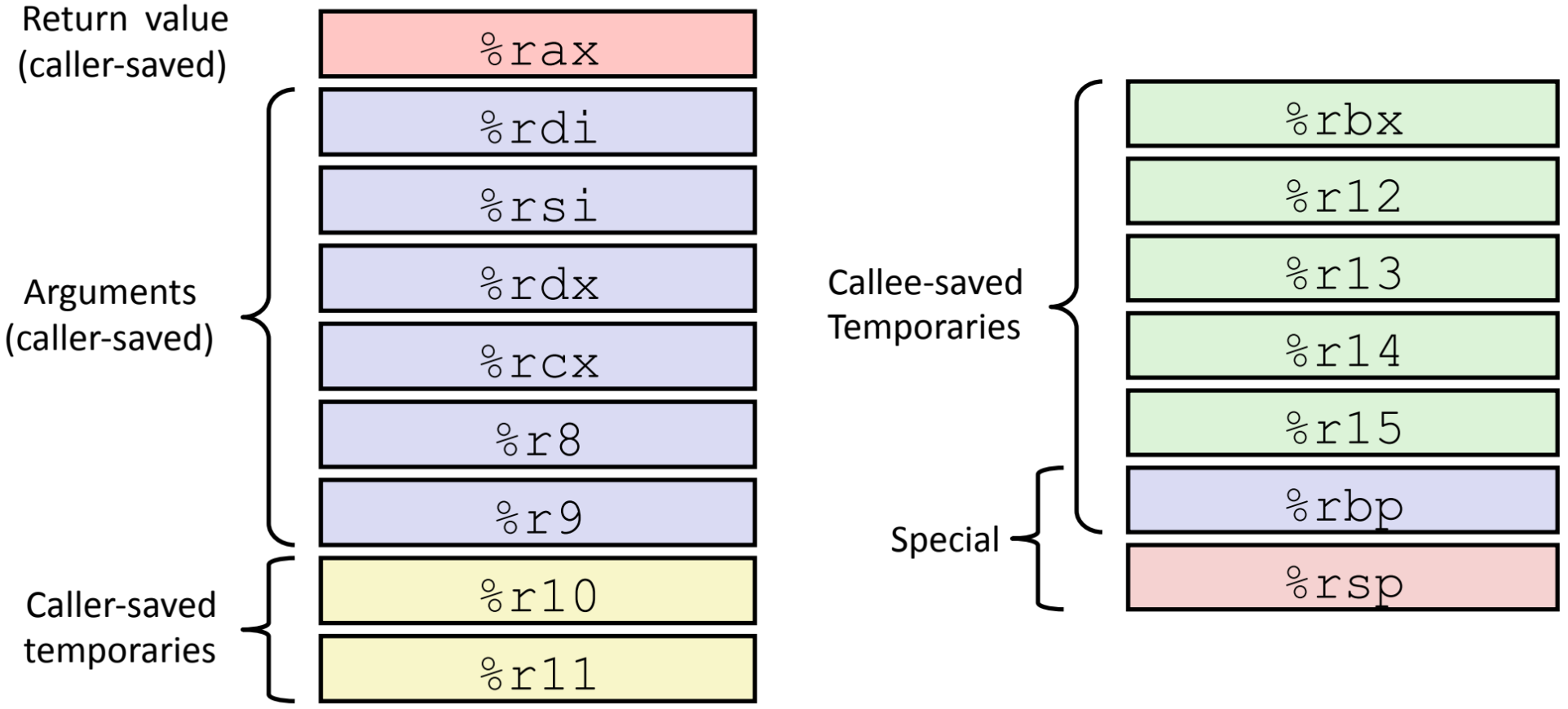
- Alice still needs the register after calling Bob, but Bob doesn’t need it
- So even though Alice cares that the register not be clobbered, Bob never will

Caller vs. callee saved registers

- Caller-saved: Alice is in charge of saving the register, but Alice doesn’t know whether Bob will use the register, so Alice must save it!
- Callee-saved: Bob is in charge of saving the register, and Bob knows that he doesn’t need it. Thus, Bob can skip saving the register!

Caller vs. callee saved registers

- Neither approach is always better
- The standard x86-64 *convention* takes a mixed approach
- Some registers are caller-saved
- Other registers are callee-saved…


Return value
(caller-saved)
%rax
Arguments
(caller-saved)
%rdi
%rsi
%rdx
%rcx
%r8
%r9
Caller-saved
temporaries
%r10
%r11
Callee-saved
Temporaries
%rbx
%r12
%r13
%r14
%r15
Special
%rbp
%rsp

Caller vs. callee saved registers

- Caller-saved registers: whatever you call can clobber these, and you need to save them if you want to use them again
  - Leaf functions love using these registers!
- Callee-saved registers: whatever you call can’t clobber these
  - Non-leaf *typically* love using these registers!

Register special cases

- `%rbp`: the frame/block pointer
  - If `%rbp` is used as a block pointer, it will always be saved by the callee at the beginning of the function
    - `push %rbp` at the very start of the function
    - `pop %rbp` at the very end of the function
  - Otherwise, it has normal callee-saved behavior

Register special cases

- `%rsp`: the stack pointer
  - It's not callee-saved in the sense that you ever push or pop `%rsp` to or from the stack to save or restore it
  - But it is a callee's responsibility to ensure that %rsp is the same as it was before the function call
  - So it is at least callee *preserved*

Function call special cases

- If a compiler is compiling `f()`, is there a function `f()` could call where the compiler would necessarily have complete information on the registers used by that function?

Function call special cases

- If a compiler is compiling `f()`, is there a function `f()` could call where the compiler would necessarily have complete information on the registers used by that function?
- Recursive functions!
- For many calls, including recursive calls, the compiler may have additional information to avoid unnecessary saves/restores

Saved registers summary

- Function calls have to share the same set of registers, so we need some sort of convention how we avoid clobbering each other
- Different registers have different rules concerning who should preserve them
- Some register preservation rules are more efficient in some scenarios than others, so compilers choose registers carefully

printf vs. scanf

- Both use format strings
- Printf is used to display output

Format specifiers

- `%d` – signed int in decimal (int)
- `%u` – unsigned int in decimal (unsigned int)
- `%x` – unsigned int in hex (unsigned int)
- `%s` – string (char *)
- `%f` – double (double)
- `%c` – char (char)

printf

- `int x = -3;`
- `unsigned int y = 5;`
- `char *z = "foo";`
- `printf("%d: %u, %s\n", x, y, z);`

scanf

- `int x;`
- `unsigned int y;`
- `char *z;`
- `scanf("%d: %u, %s\n", &x, &y, &z);`

sscanf

- `int x;`
- `unsigned int y;`
- `char *z;`
-
```
sscanf("-3: 5, barf\n",
        "%d: %u, %s\n", &x, &y, &z);
```

sscanf

- `int x;`
- `unsigned int y;`
- `char *z;`
- ```
  sscanf("-3: 5, barf\n",
         "%d: %u, %s\n", &x, &y, &z);
  ```
- `// x == ???, y == ???, z == ???`
  ```

sscanf

- `int x;`
- `unsigned int y;`
- `char *z;`
- `sscanf("-3: 5, barf\n",`
  `"%d: %u, %s\n", &x, &y, &z);`
- `// x == -3, y == 5, z == "barf"`

Multiple return values

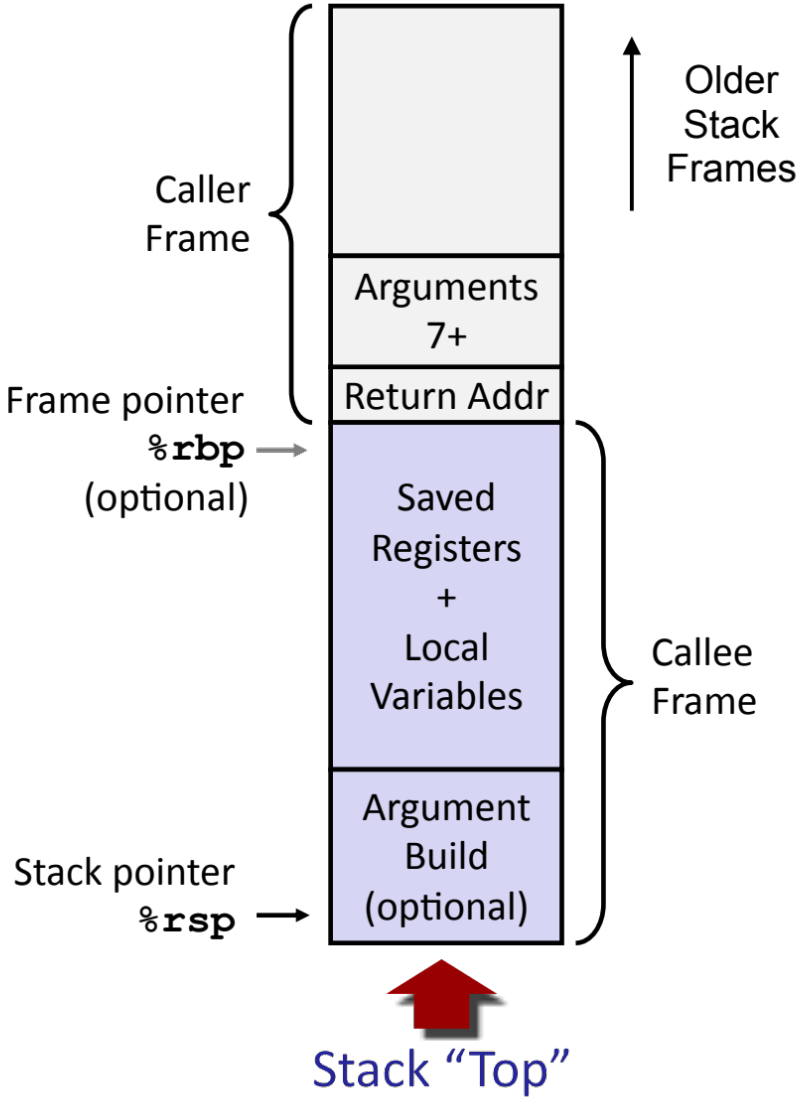
- Output pointers is a common pattern in C to facilitate multiple return values
- ```
  int f(input args..., int *int_out, char *str_out) {
      ...
      *int_out = 1234;
      *str_out = "barf";
      return 0; // some indication of success or failure
  }
  ```
- ```
  if (f(..., &some_int, &some_str) == 0) {
      ... safe to use some_int and some_str ...
  }
  ```

scanf/sscanf return value

- The # of values successfully read
- If your format string is "%d %d %d":
  - 0 is total failure
  - 1 or 2 is partial failure
  - 3 is success
- Because there are three format specifiers

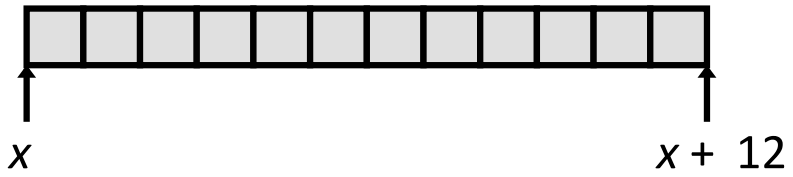
Functions exercises

```
printf(format, …)
sprintf(output, format, ...)
scanf(format, ...)
sscanf(input, format, …)
```

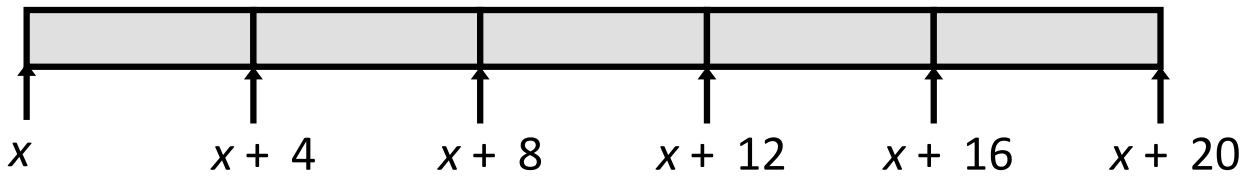


Arrays in x86-64

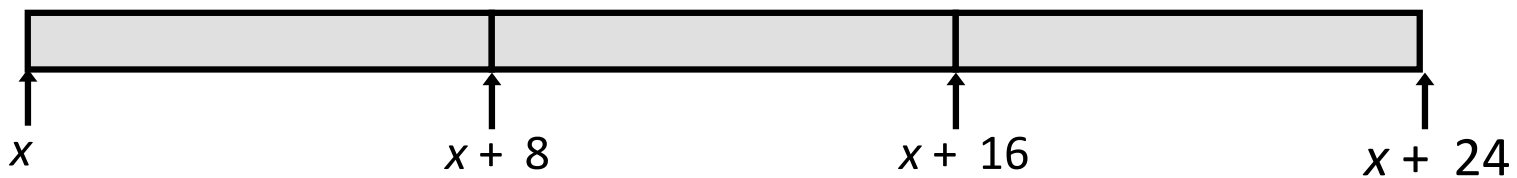
```
char string[12];
```



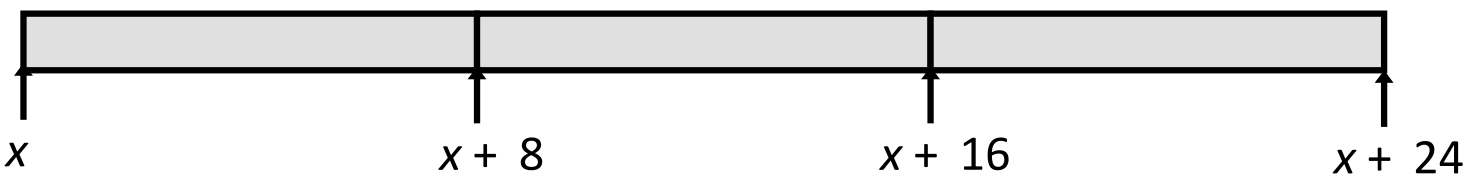

```
int val[5];
```




```
double a[3];
```




```
char* p[3];
```



Stack-based array

- `int x[10];`
- `int foo; // 4 bytes`
- `int bar; // 4 bytes`
- `eax = x[rcx] → movl 8(%rsp, %rcx, 4), %eax`
- `rbx = &x[rcx] → leaq 8(%rsp, %rcx, 4), %rbx`

Heap-based array

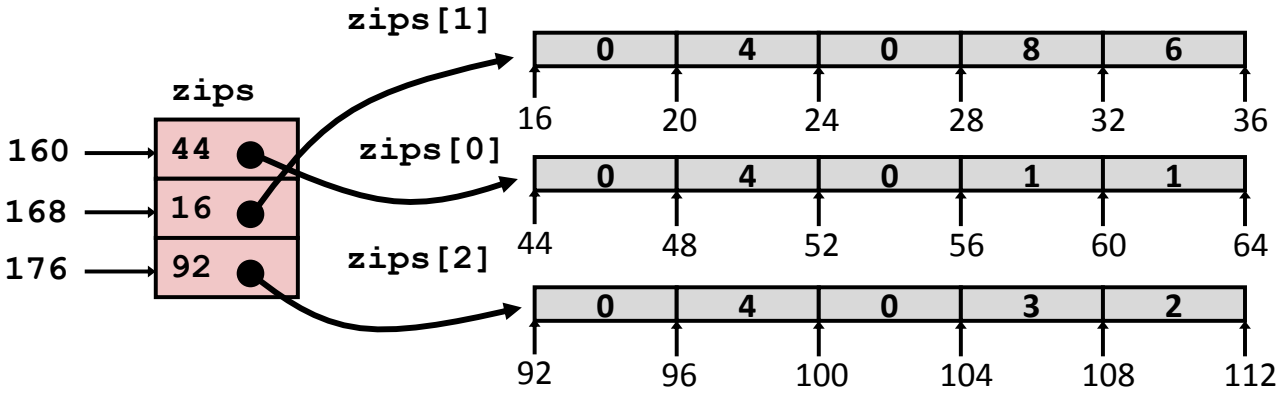
- `int *rsi = malloc(10 * sizeof(int));`
- `eax = rsi[rcx]` → `movl (%rsi, %rcx, 4), %eax`
- `rbx = &rsi[rcx]` → `leaq (%rsi, %rcx, 4), %rbx`

Multidimensional arrays

- Arrays that have two or more dimensions
- E.g., for three dimensions, we would index like `x[5][4][7]`

Multilevel arrays

```
int *zips[3];  // stored at address 160
zips[0] = (int *) malloc(sizeof(int) * 5);
zips[0][1] = 4;
...
```

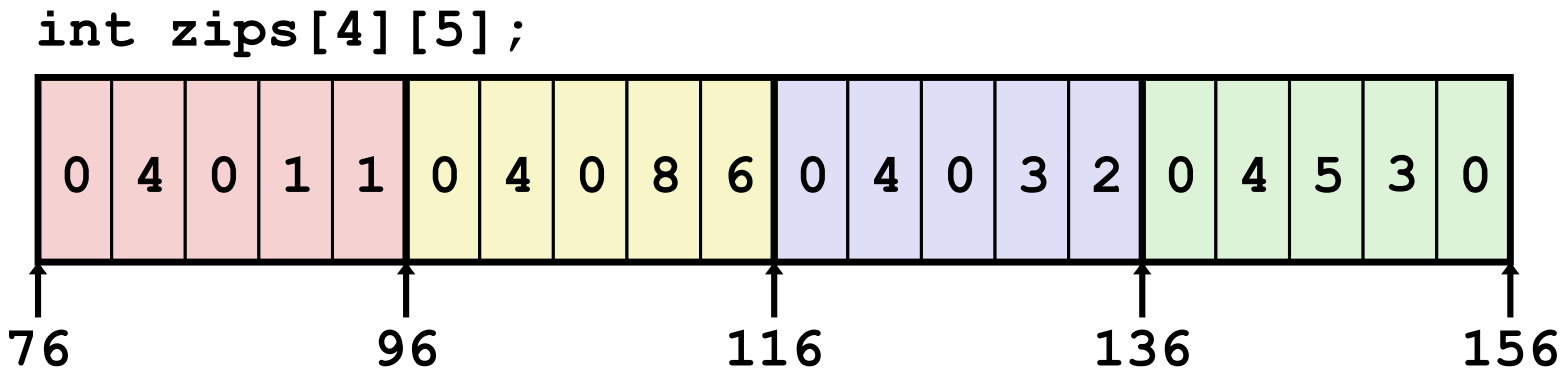


Multilevel arrays

- An array of pointers to arrays
- E.g., all arrays in Java are multilevel arrays
- You can do this in C
- But there is another approach…

Nested array on stack

```
int zips[4][5];  // stored at address 76
zips[0][1] = 4;
...
```



Two ways we could have done this

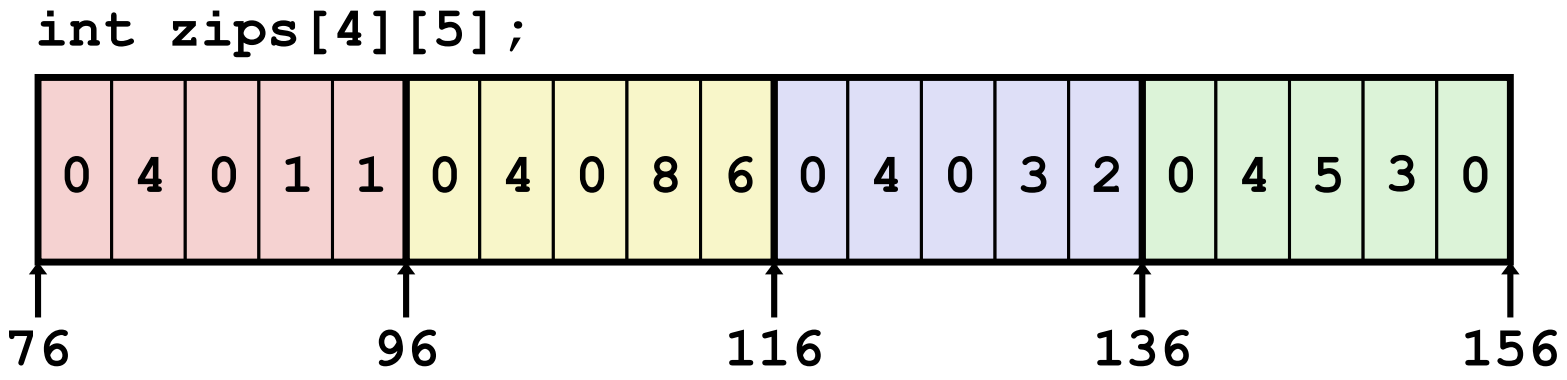
- int zips[4][5];
- Is this 4 rows of length 5?
- Or 5 rows of length 4?

Two ways we could have done this

- int zips[4][5];
- Is this 4 rows of length 5? *Row-major*
- Or 5 rows of length 4? *Column-major*

Nested array on stack

```
int zips[4][5];  // stored at address 76
zips[0][1] = 4;
...
```



Multidimensional arrays

- C uses row-major order
- Some languages (Fortran) use column-major
- To get the same layout in a column-major language, you reverse the order of the indexes:
- `int zip[5][4];  // hypothetical column-major C`
- `zips[1][0] = 4; // hypothetical column-major C`

Nested array on heap

```
int (*zips)[4][5] = malloc(sizeof(int[4][5]));
(*zips)[0][1] = 4
...
```

Nested array on heap

```
int (*zips)[4][5] = malloc(sizeof(int[4][5]));
(*zips)[0][1] = 4
...
```

- Note:

  `(*zips)[x][y]`

- is equivalent to:

```
int *ptr = (int *) zips;
ptr[x * 5 + y] // column-major would be different
```

Multilevel vs. nested arrays

- Multilevel arrays can be jagged: different rows can have different lengths
- Nested arrays are faster (no pointer chasing)
- Nested arrays take less memory (no overhead pointer)

TODO

- Assignment 3 assigned due October 28