CSE 351 Section 5

Stack & Procedures, Recursion

Administrivia

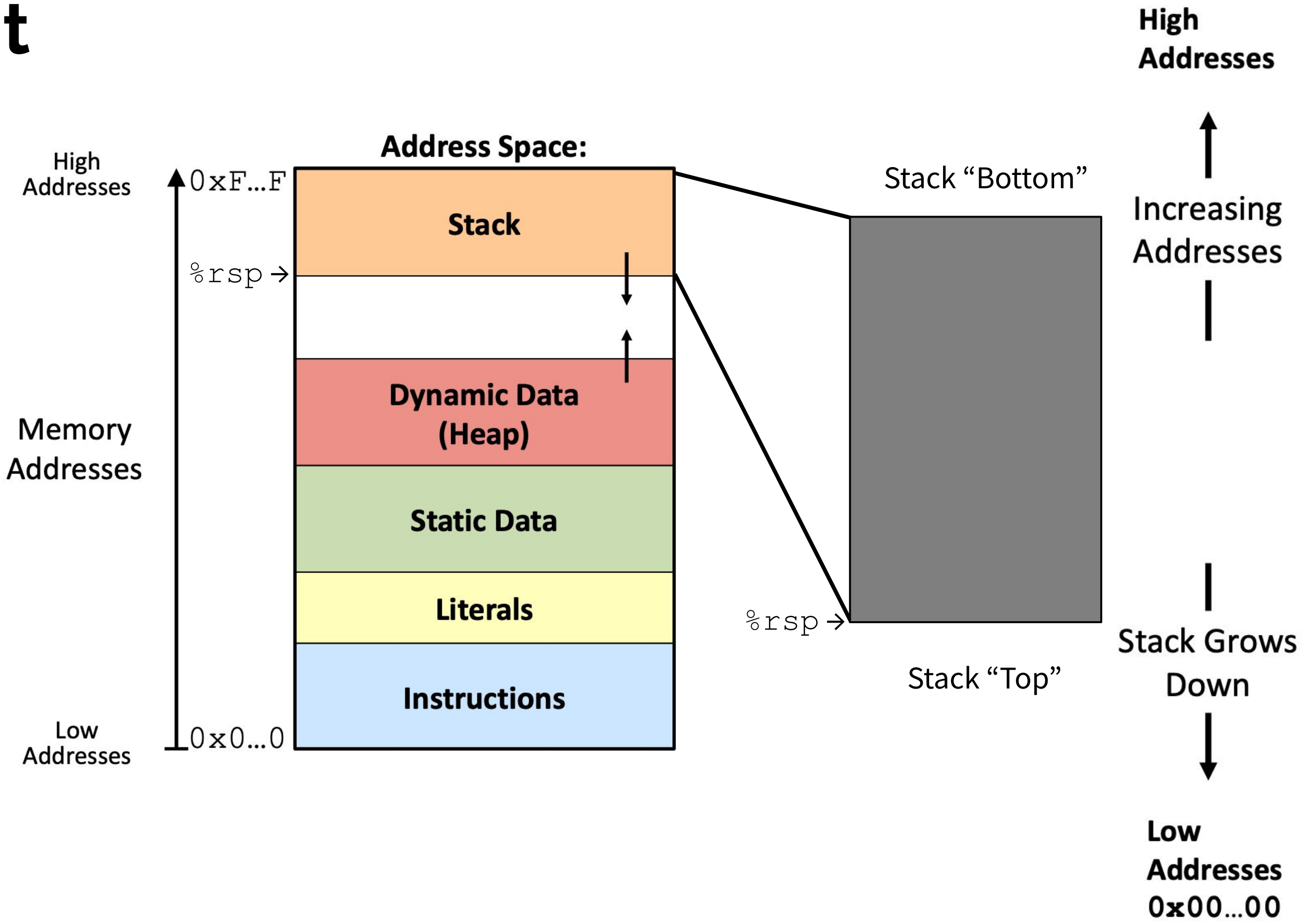
- **Homework 12**
  - Due Friday (10/28)
- **Homework 13**
  - Due Wednesday (11/2) – longer because it covers two lectures
- **Lab 2**:
  - Due this Friday (10/28)
  - Make sure you put each phase answer on a new line, and have an empty line after your last phase answer (don't actually type '\n')
  - One late day covers Saturday and Sunday

Stack & Procedures

(%rsp is the MVP)

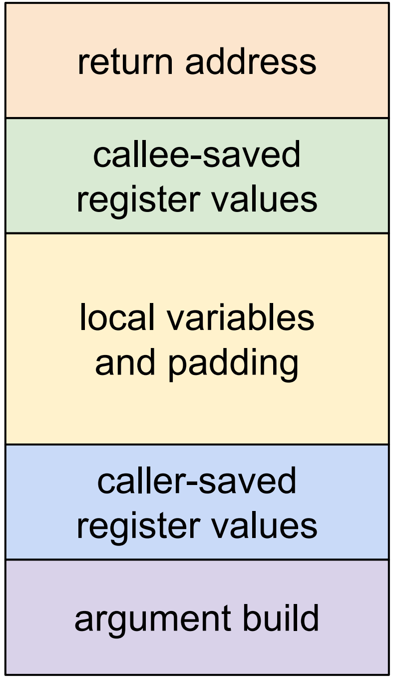
Memory Layout

- Stack is located at the top of our memory layout
- Stack is placed upside down in memory, with higher addresses considered the “bottom” and lower addresses considered the “top”
- There is a dedicated register `%rsp` that points to the current top of the stack



Stack Frame Structure

- Return address
  - Pushed by `callq`; address of instruction after `callq`
- Callee-saved registers
  - Only if function modifies/uses them
- Local variables
  - Variables that fit in a register may not be allocated on the Stack
  - Unavoidable if variable is too big for a register (*e.g.*, array)
  - Unavoidable if variable needs an address (*i.e.*, uses `&var`)
- Caller-saved registers
  - Only if values are needed *across* a function call
- Argument build
  - Only if function calls a function with more than six arguments



Calling Conventions

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

First 6 arguments are ordered in registers:
1: %rdi, 2: %rsi, 3: %rdx, 4: %rcx, 5: %r8, 6: %r9
Registers are not part of memory/the stack.

Additional arguments are pushed to the stack by the caller *before* invoking callq
In reverse order: arg n pushed first, arg 7 last.
Part of the caller's stack frame.

Return value
Placed in %rax.

Register Saving Conventions

Registers

Name	Convention	Name of "virtual" register: Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – Caller saved	%eax	%ax	%al
%rbx	Callee saved	%ebx	%bx	%bl
%rcx	Argument #4 – Caller saved	%ecx	%cx	%cl
%rdx	Argument #3 – Caller saved	%edx	%dx	%dl
%rsi	Argument #2 – Caller saved	%esi	%si	%sil
%rdi	Argument #1 – Caller saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	Callee saved	%ebp	%bp	%bpl
%r8	Argument #5 – Caller saved	%r8d	%r8w	%r8b
%r9	Argument #6 – Caller saved	%r9d	%r9w	%r9b
%r10	Caller saved	%r10d	%r10w	%r10b
%r11	Caller saved	%r11d	%r11w	%r11b
%r12	Callee saved	%r12d	%r12w	%r12b
%r13	Callee saved	%r13d	%r13w	%r13b
%r14	Callee saved	%r14d	%r14w	%r14b
%r15	Callee saved	%r15d	%r15w	%r15b

"Caller-saved" registers:

- %rax, %rcx, %rdx, %rsi, %rdi, %r8–%r11
- If caller needs to use their value(s) across a function call, then push onto the stack.
- Pushed just before function call; popped right after.

"Callee-saved" registers:

- %rbx, %rbp, %r12–%r15
- If callee wants to change their value(s), then push onto the stack.
- Pushed at beginning of function; popped just before ret.

Stack Frame Example

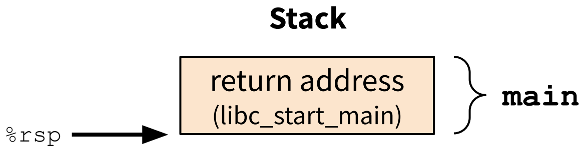
In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- `main` is actually started by a library routine, so it has arguments and a return address, too!



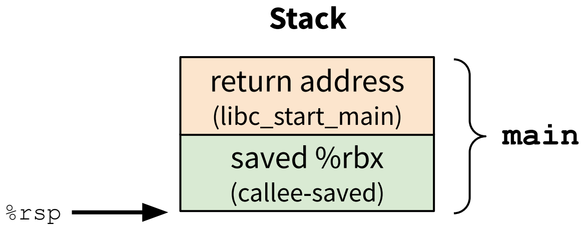

```
main:
        pushq   %rbx
        subq    $16, %rsp
        movl    %edi, %ebx
        movl    $351, 12(%rsp)
        movl    $1, (%rsp)
        movl    $2, 4(%rsp)
        movl    $3, 8(%rsp)
        pushq   $7
        movl    $6, %r9d
  ⋮
```

Stack Frame Example

- `main` uses `%ebx`, a callee-saved register, in Line 03, so it must save the old value on the stack before then.

`main:`

```
01    pushq   %rbx
02    subq    $16, %rsp
03    movl    %edi, %ebx
04    movl    $351, 12(%rsp)
05    movl    $1, (%rsp)
06    movl    $2, 4(%rsp)
07    movl    $3, 8(%rsp)
08    pushq   $7
09    movl    $6, %r9d
  ⋮
```



Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

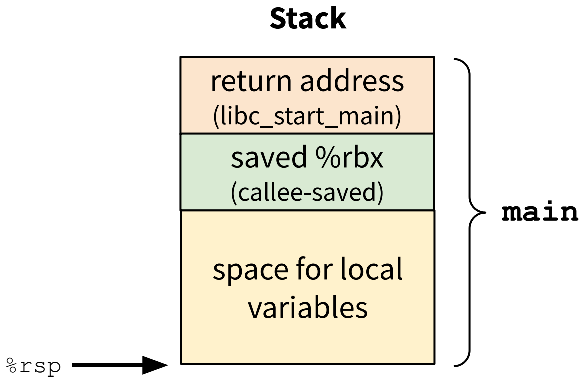
Let's look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- `main` then allocates space on the stack for local variables:
  - We use `&x`, so `x` must go on the stack
  - `a[]` takes up 12 bytes, so must go on the stack

main:

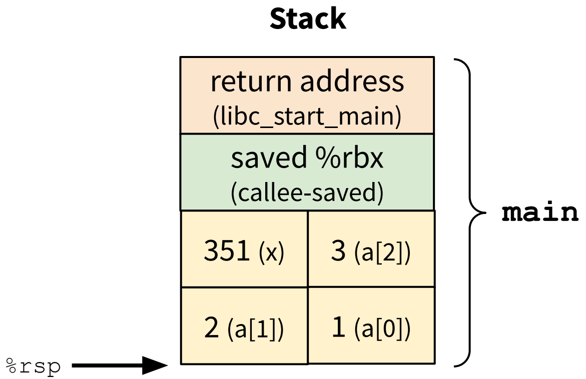
```
        pushq    %rbx
        subq     $16, %rsp
        movl     %edi, %ebx
        movl     $351, 12(%rsp)
        movl     $1, (%rsp)
        movl     $2, 4(%rsp)
        movl     $3, 8(%rsp)
        pushq    $7
        movl     $6, %r9d
⋮
```



Stack Frame Example

- `main` then stores the initial values of the local variables in their corresponding locations:
  - `x` is placed at `12(%rsp)`
  - `a[0]` is placed at `(%rsp)`, with the other two elements directly following that

```
main:
        pushq   %rbx
        subq    $16, %rsp
        movl    %edi, %ebx
        movl    $351, 12(%rsp)
        movl    $1, (%rsp)
        movl    $2, 4(%rsp)
        movl    $3, 8(%rsp)
        pushq   $7
        movl    $6, %r9d
        ⋮
```



Note: without an explicit reference to &x, x may not be stored on the stack depending on whether the compiler can optimize that memory access away

Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

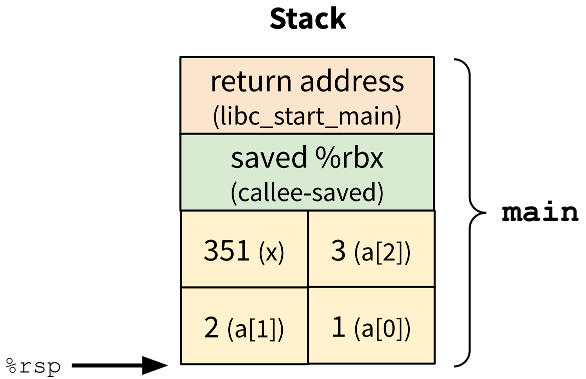
```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let's look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- Note that, in this example, the compiler chose to save `%rdi` (caller-saved) in `%rbx` back in Line 03 instead of pushing the old value of `%rdi` to the stack here so no caller-saved registers are pushed to the stack.

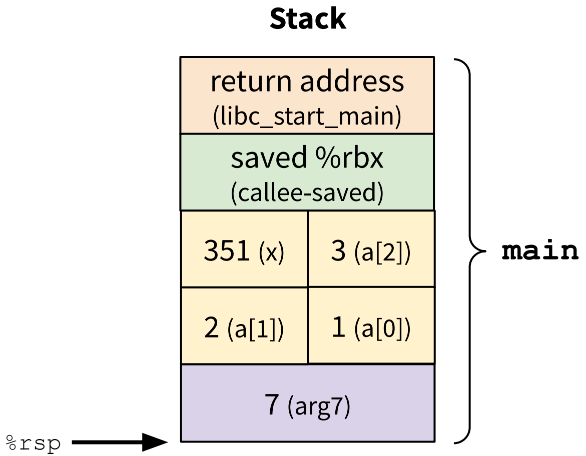
```
main:
01      pushq   %rbx
02      subq    $16, %rsp
03      movl    %edi, %ebx
04      movl    $351, 12(%rsp)
05      movl    $1, (%rsp)
06      movl    $2, 4(%rsp)
07      movl    $3, 8(%rsp)
08  →   pushq   $7
09      movl    $6, %r9d
  ⋮
```



Stack Frame Example

- `main` then prepares to call `foo`, which includes putting `arg7` (7) on the stack.
    - `arg1`/`%rdi` is a pointer; note that `&x` is now 20 bytes above `%rsp` after `arg7` is pushed

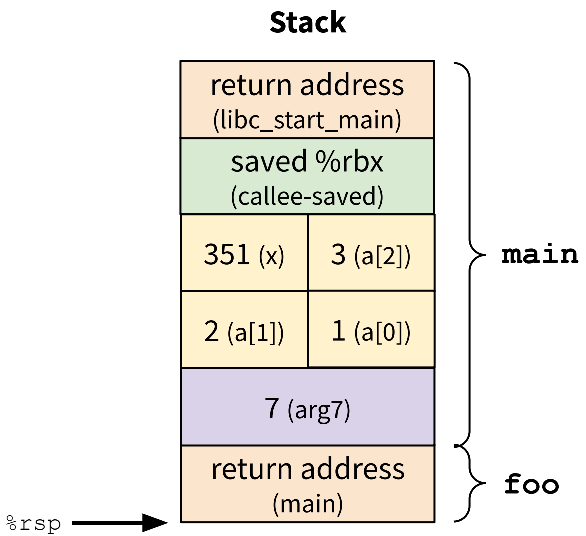
```
⋮
pushq   $7
movl    $6, %r9d
movl    $5, %r8d
movl    $4, %ecx
movl    $3, %edx
movl    $2, %esi
leaq    20(%rsp), %rdi
movl    $0, %eax
call    foo
⋮
```



Stack Frame Example

- `main` calls `foo`, which pushes the return address to `main` on the stack and marks the beginning of a new stack frame.

```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```



Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

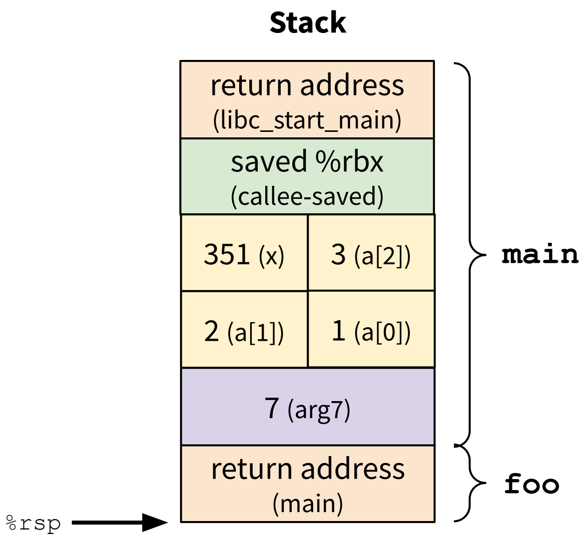
```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let's look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- `foo` has a minimal stack frame and doesn't put anything on the stack, however, it does access memory on the stack, since it sums `*arg1` and `arg7`.

```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```



Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

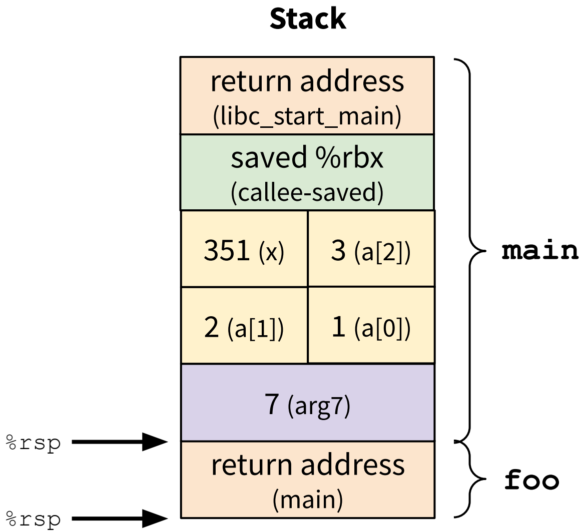
```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- `foo` returns to `main` using `ret`, which pops the return address to `main` from the stack into `%rip`.
  - `foo` has finished execution and its stack frame is now deallocated

```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```



Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

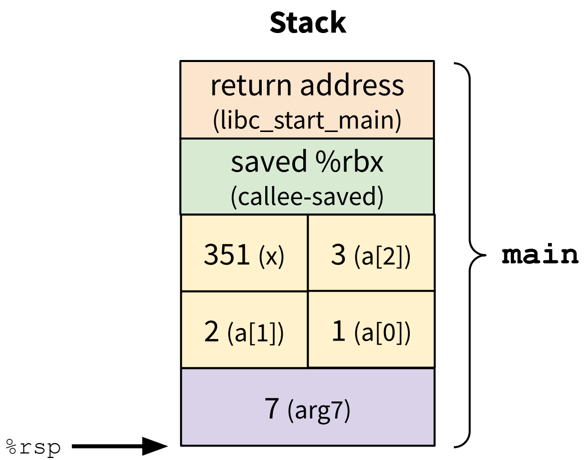
```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

- `main` returns the original value of `%rdi`, which was stored in `%rbx`, plus the return value of `foo`.

```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```



Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions.
Consider the following lines of code:

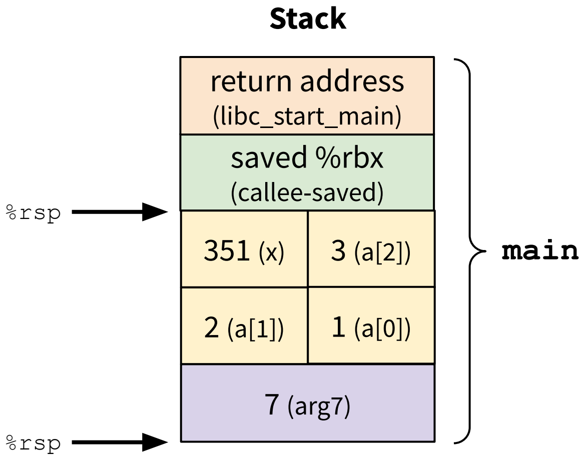
```
int main(int argc, char* argv[]) {
  int x = 351;
  int a[] = {1, 2, 3};
  int y = foo(&x, 2, 3, 4, 5, 6, 7);
  return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
  return *arg1 + arg7;
}
```

Let's look at how the stack grows and shrinks as the code above executes in assembly.

Stack Frame Example

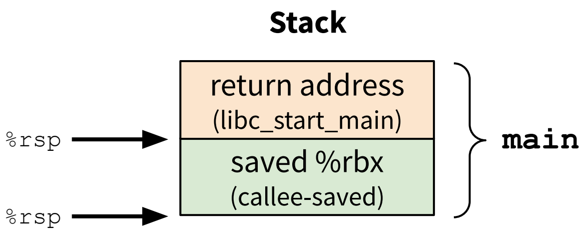
- `main` deallocates the argument build and local variables simultaneously (we must work our way up the stack).
    - Note that these would have been split up if there had been caller-saved registers on the stack

```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```



Stack Frame Example

- `main` must restore the saved value of `%rbx` before returning.




```
⋮
        movl    $0, %eax
        call    foo
        addl    %ebx, %eax
        addq    $24, %rsp
        popq    %rbx
        ret
foo:
        movl    (%rdi), %eax
        addl    8(%rsp), %eax
        ret
```

Stack Exercise

```c
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07          movzbl (%rdi),%eax
 4005e9: 84 c0             test %al,%al
 4005eb: 74 13             je 400600 <rfun+0x1a>
 4005ed: 53                push %rbx
 4005ee: 48 0f be d8       movsbq %al,%rbx
 4005f2: 48 83 c7 01       add $0x1,%rdi
 4005f6: e8 eb ff ff ff    callq 4005e6 <rfun>
 4005fb: 48 01 d8          add %rbx,%rax
 4005fe: eb 06             jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00    mov $0x0,%eax
 400605: c3                retq
 400606: 5b                pop %rbx
 400607: c3                retq
```

Stack Exercise

a) In terms of the C function, what value is being saved on the stack?

```c
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07          movzbl (%rdi),%eax
 4005e9: 84 c0             test %al,%al
 4005eb: 74 13             je 400600 <rfun+0x1a>
 4005ed: 53                push %rbx
 4005ee: 48 0f be d8       movsbq %al,%rbx
 4005f2: 48 83 c7 01       add $0x1,%rdi
 4005f6: e8 eb ff ff ff    callq 4005e6 <rfun>
 4005fb: 48 01 d8          add %rbx,%rax
 4005fe: eb 06             jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00    mov $0x0,%eax
 400605: c3                retq
 400606: 5b                pop %rbx
 400607: c3                retq
```

Stack Exercise

b) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)?

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07          movzbl (%rdi),%eax
 4005e9: 84 c0             test %al,%al
 4005eb: 74 13             je 400600 <rfun+0x1a>
 4005ed: 53                push %rbx
 4005ee: 48 0f be d8       movsbq %al,%rbx
 4005f2: 48 83 c7 01       add $0x1,%rdi
 4005f6: e8 eb ff ff ff    callq 4005e6 <rfun>
 4005fb: 48 01 d8          add %rbx,%rax
 4005fe: eb 06             jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00    mov $0x0,%eax
 400605: c3                retq
 400606: 5b                pop %rbx
 400607: c3                retq
```

c) `char *s = "CSE351"`

Stack Exercise

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (l
    s++;
    return temp +
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
 4005e6: 0f b6 07          movzbl (%rdi),%eax
 4005e9: 84 c0             test %al,%al
                           un+0x1a>
                           x
                           rfun>
 4005fe: eb 06             jmp 400606 <rfun+0x20>
 400600: b8 00 00 00 00    mov $0x0,%eax
 400605: c3                retq
 400606: 5b                pop %rbx
 400607: c3                retq
```

c) Assume `main` calls `rfun` with `char *s = "CSE351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack?

Stack Exercise

00000000004005e6 <rfun>:

4005e6: 0f b6 07	movzbl (%rdi),%eax
4005e9: 84 c0	test %al,%al
4005eb: 74 13	je 400600 <rfun+0x1a>
4005ed: 53	push %rbx
4005ee: 48 0f be d8	movsbq %al,%rbx
4005f2: 48 83 c7 01	add $0x1,%rdi
4005f6: e8 eb ff ff ff	callq 4005e6 <rfun>
4005fb: 48 01 d8	add %rbx,%rax
4005fe: eb 06	jmp 400606 <rfun+0x20>
400600: b8 00 00 00 00	mov $0x0,%eax
400605: c3	retq
400606: 5b	pop %rbx
400607: c3	retq

Memory Address	Value	Description
0x7fffffffdb48	Unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	Unknown	Original %rbx
0x7fffffffdb28	0x4005fb	Return address
0x7fffffffdb20	*s, "C", 0x43	Saved %rbx
0x7fffffffdb18	0x4005fb	Return address
0x7fffffffdb10	*s, *(s+1), "S", 0x53	Saved %rbx
0x7fffffffdb08	0x4005fb	Return address
0x7fffffffdb00	*s, *(s+2), "E", 0x45	Saved %rbx