

Data Structures in Assembly

❖ Arrays

- One-dimensional
- Multi-dimensional (nested)
- **Multi-level**

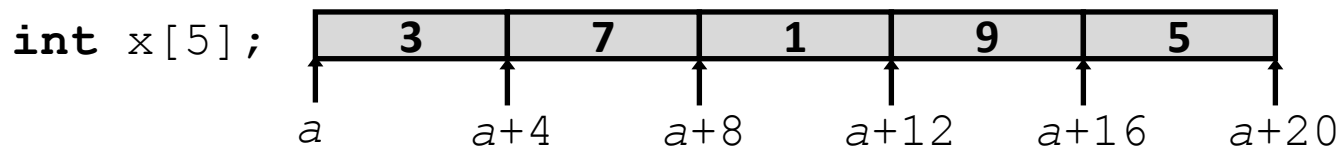
❖ Structs

- Alignment

Array Access

❖ Basic Principle

- **T** A[N] ; → array of data type **T** and length N
- Identifier A returns address of array (type **T***)



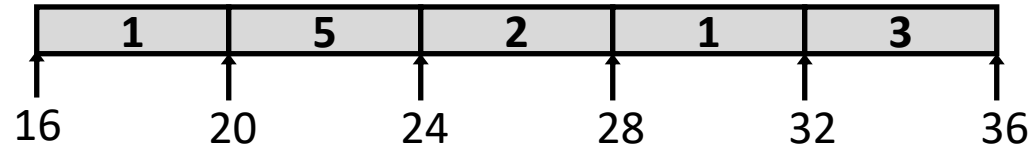
❖ Reference Type Value

<code>x[4]</code>	<code>int</code>	5
<code>x</code>	<code>int*</code>	<code>a</code>
<code>x+1</code>	<code>int*</code>	<code>a + 4</code>
<code>&x[2]</code>	<code>int*</code>	<code>a + 8</code>
<code>x[5]</code>	<code>int</code>	?? (whatever's in memory at addr <code>x+20</code>)
<code>*(x+1)</code>	<code>int</code>	7
<code>x+i</code>	<code>int*</code>	<code>a + 4*i</code>

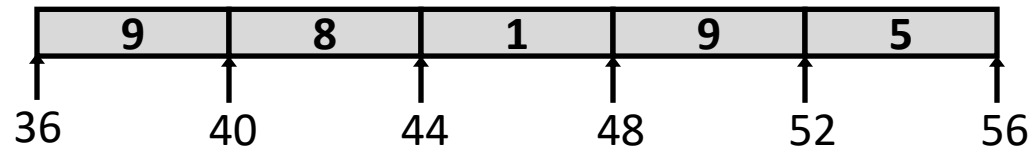
```
typedef int vec5[5];
```

Referencing Examples

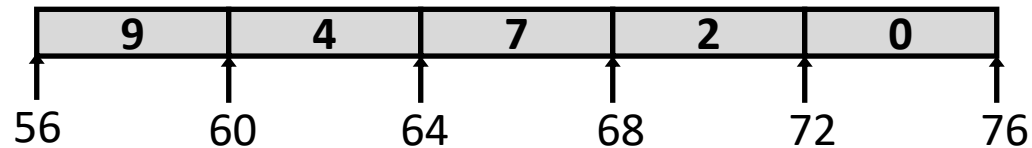
`vec5 cmu;`



`vec5 sfu;`



`vec5 ucb;`



<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
<code>sfu[3]</code>	$36 + 4 * 3 = 48$	9	Yes
<code>sfu[6]</code>	$36 + 4 * 6 = 60$	4	No
<code>sfu[-1]</code>	$36 + 4 * -1 = 32$	3	No
<code>cmu[15]</code>	$16 + 4 * 15 = 76$	??	No

- ❖ No bounds checking
- ❖ Example arrays happened to be allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Nested Array Example

```
typedef int vec5[5];
```

Remember, **T** A[N] is
an array with elements
of type **T**, with length N

```
vec5 sea[4] =  
{  
  { 9, 8, 1, 9, 5 },  
  { 9, 8, 1, 0, 5 },  
  { 9, 8, 1, 0, 3 },  
  { 9, 8, 1, 1, 5 }  
};
```

same as:

What is the layout in memory?

```
int sea[4][5];
```

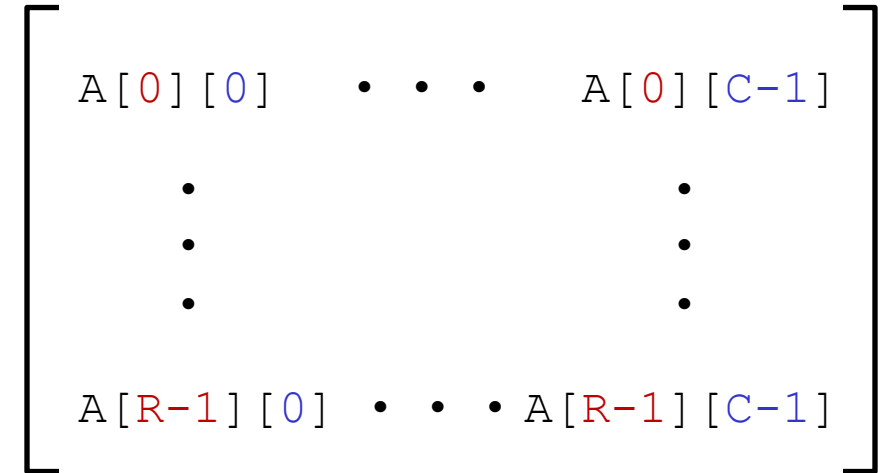
How is a 2D array packed into memory?

1D Array	2D Array
<pre>int array[10];</pre> Elements in sequence	<pre>int array[10][10];</pre> What is the layout?
$a[i] = \&a[0] + i$ Byte offset: $i * \text{sizeof}(\text{int})$	$a[i][j] ?$ Byte offset: ?

Two-Dimensional (Nested) Arrays

❖ Declaration: **T** A[R][C];

- 2D array of data type T
- R rows, C columns
- Each element requires **sizeof(T)** bytes

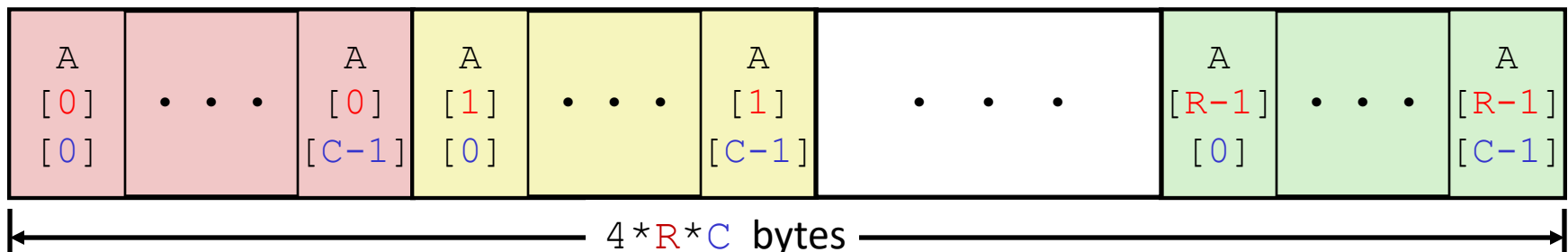


❖ Array size:

- $R * C * \text{sizeof}(T)$ bytes

❖ Arrangement: **row-major** ordering

int A[R][C];

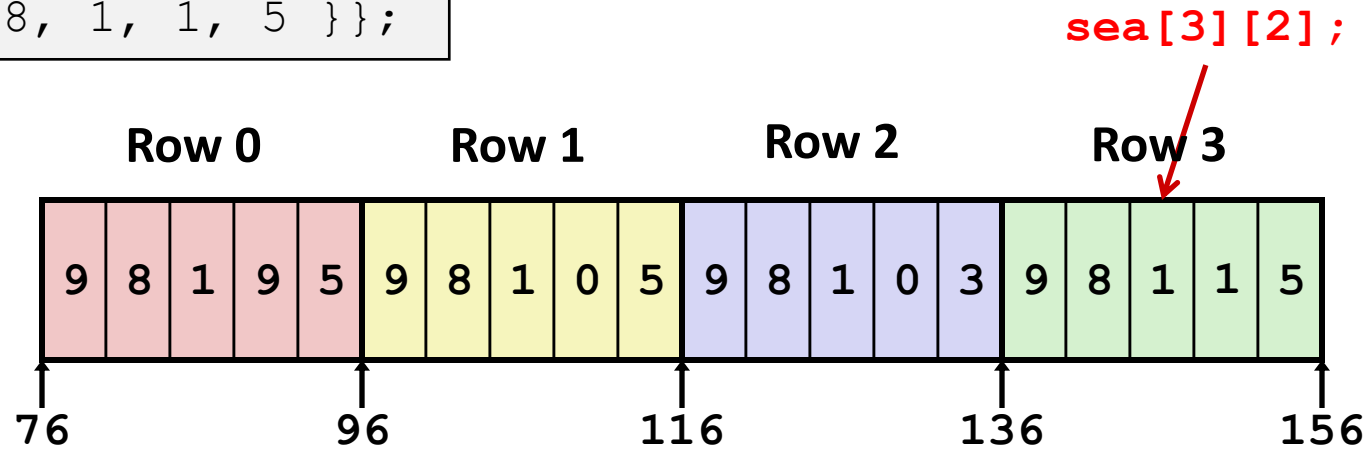


Nested Array Example

```
typedef int vec5[5];
```

```
vec5 sea[4] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

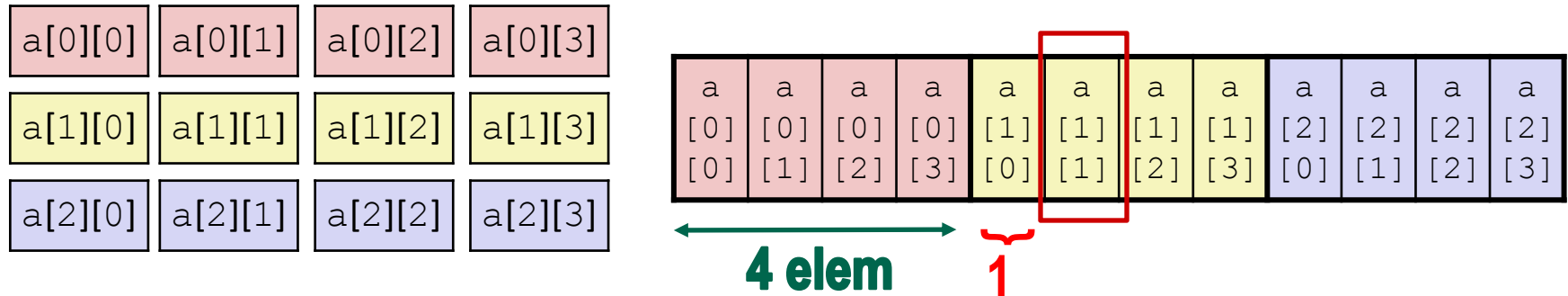
Remember, $\mathbf{T} \ A[N]$ is an array with elements of type $\mathbf{T}$, with length N



- ❖ “Row-major” ordering of all elements
- ❖ Elements in the same row are contiguous
- ❖ Guaranteed (in C)

Address calculation

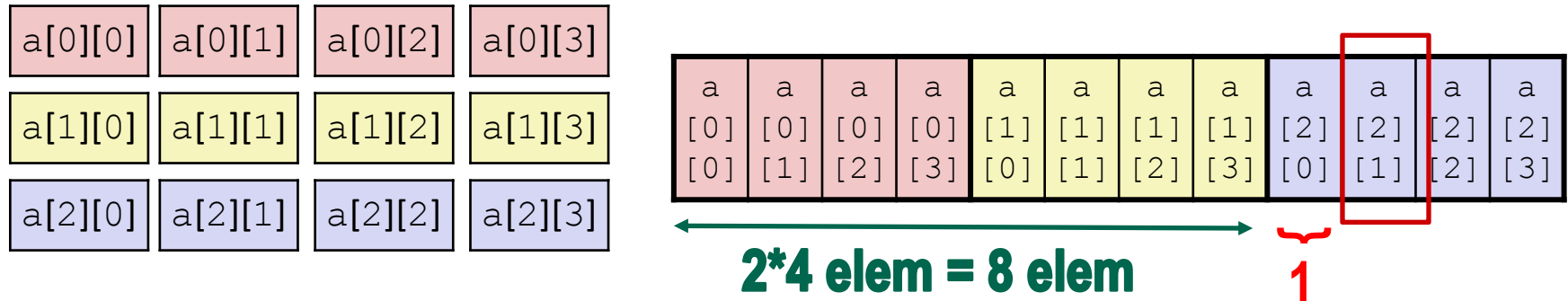
2D array: `int a[3][4]`. Find offset of `a[1][1]` in elements `i=1,j=1`



$$\begin{aligned}
 a[1][1] &= \text{Address of } A[1][0] + 1 \\
 &= 1 * \# \text{ of elem/1D array} + 1 \\
 &= 1 * 4 + 1 \\
 &= 5 \text{ elems from start} = 20 \text{ bytes (if int)}
 \end{aligned}$$

Address calculation

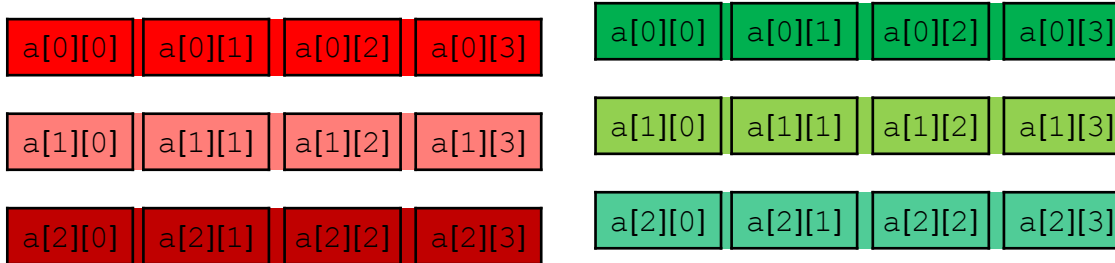
2D array: `int a[3][4]`. Find offset of `a[2][1]` in elements `i=1,j=1`



$$\begin{aligned}
 a[2][1] &= \text{Address of } A[2][0] + 1 \\
 &= 2 \times \# \text{ of elem/1D array} + 1 \\
 &= 2 \times 4 + 1 \\
 &= 9 \text{ elems from start} = 20 \text{ bytes (if int)}
 \end{aligned}$$

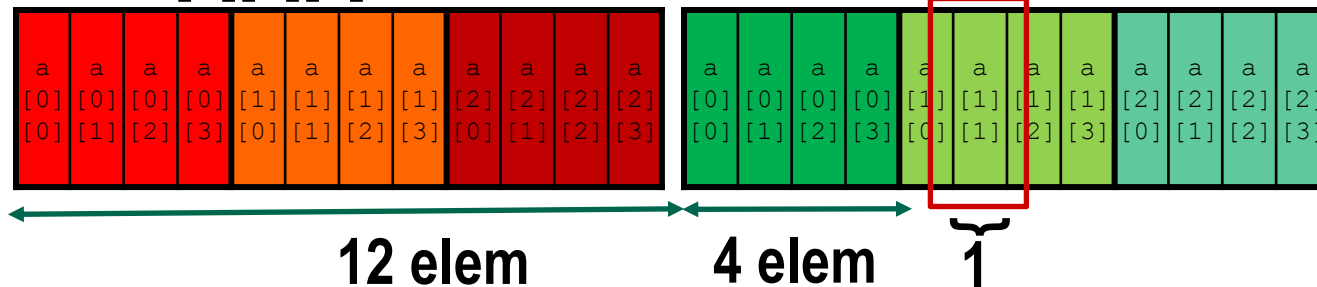
3D array

❖ 3D array: `int a[2][3][4]`. `A[0][1][1]` or `A[1][1][1]`



$a[1][1][1] = 1 * ([3][4]) + 1 * ([4]) + 1 \text{ elem}$

$a[1][1][1] = 1 * 12 + 1 * 4 + 1 = 17 \text{ elem}$



Multi-Level Array Example

Multi-Level Array Declaration(s):

```
int cmu[5] = { 1, 5, 2, 1, 3 };  
int sfu[5] = { 9, 8, 1, 9, 5 };  
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {sfu, cmu, ucb};
```

2D Array Declaration:

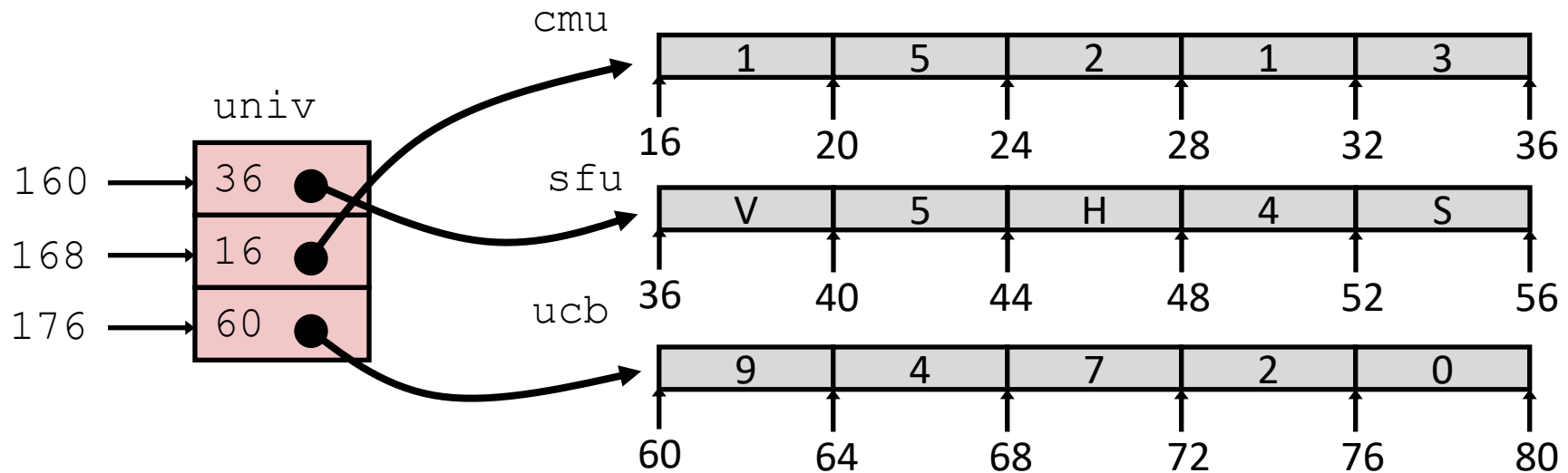
```
vec5 univ2D[3] = {  
    { 9, 8, 1, 9, 5 },  
    { 1, 5, 2, 1, 3 },  
    { 9, 4, 7, 2, 0 }  
};
```

Is a multi-level array the same thing as a 2D array?

NO

One array declaration = one contiguous block of memory

Multi-Level Referencing Examples



<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
------------------	----------------	--------------	--------------------

univ[2][3]			
------------	--	--	--

univ[1][5]			
------------	--	--	--

univ[2][-2]			
-------------	--	--	--

univ[3][-1]			
-------------	--	--	--

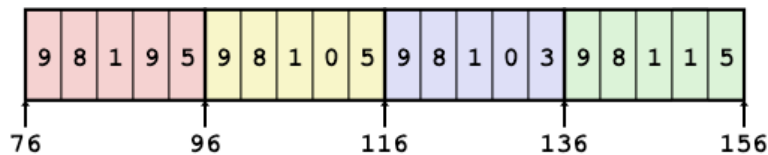
univ[1][12]			
-------------	--	--	--

- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed

Array Element Accesses

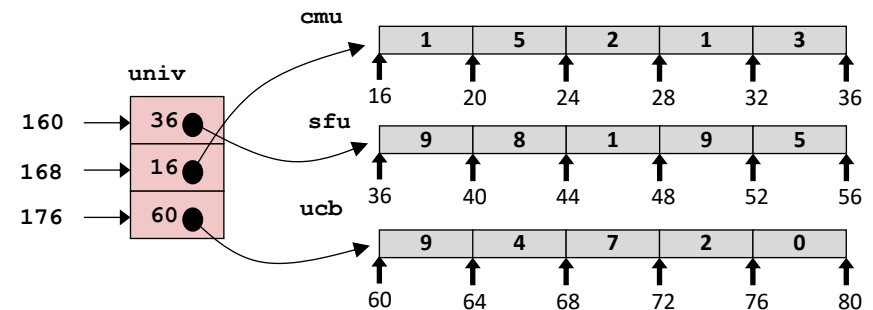
Nested array

```
int get_sea_digit
(int index, int digit)
{
    return sea[index][digit];
}
```



Multi-level array

```
int get_univ_digit
(int index, int digit)
{
    return univ[index][digit];
}
```



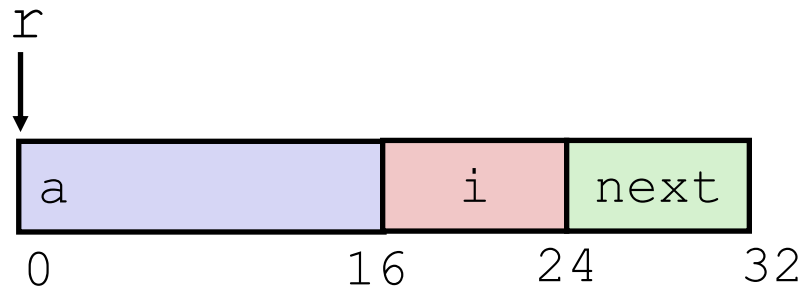
Access *looks* the same, but it isn't:

Mem[sea+20*index+4*digit]

Mem[**Mem**[univ+8*index]+4*digit]

C Structure Representation

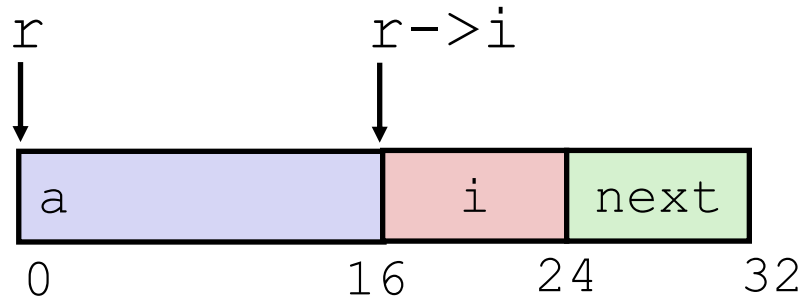
```
struct rec {  
    int a[4];  
    long i;  
    struct rec *next;  
};  
  
struct rec *r;
```



- ❖ Structure represented as block of memory
 - Big enough to hold all of the fields
- ❖ Fields ordered according to declaration order
 - Even if another ordering would be more compact
- ❖ Compiler determines overall size + positions of fields
 - Machine-level program has no understanding of the structures in the source code

Accessing a Structure Member

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



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

❖ Compiler knows the *offset* of each member within a struct

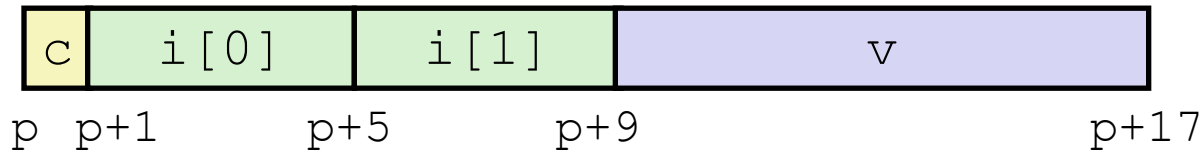
■ Compute as
`*(r+offset)`

- Referring to absolute offset, so no pointer arithmetic

```
add  a0, a1, 16 # Coming up in Week 3  
ret
```

Structures & Alignment

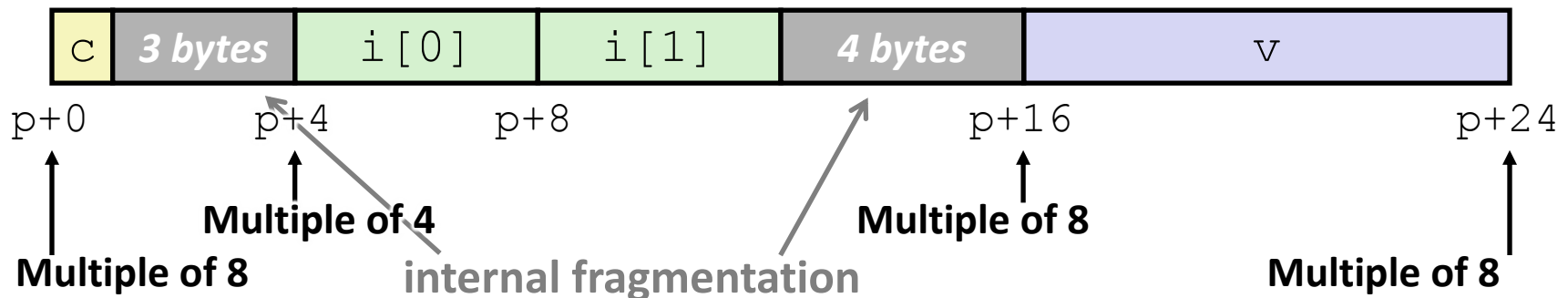
❖ Unaligned Data



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

❖ Aligned Data

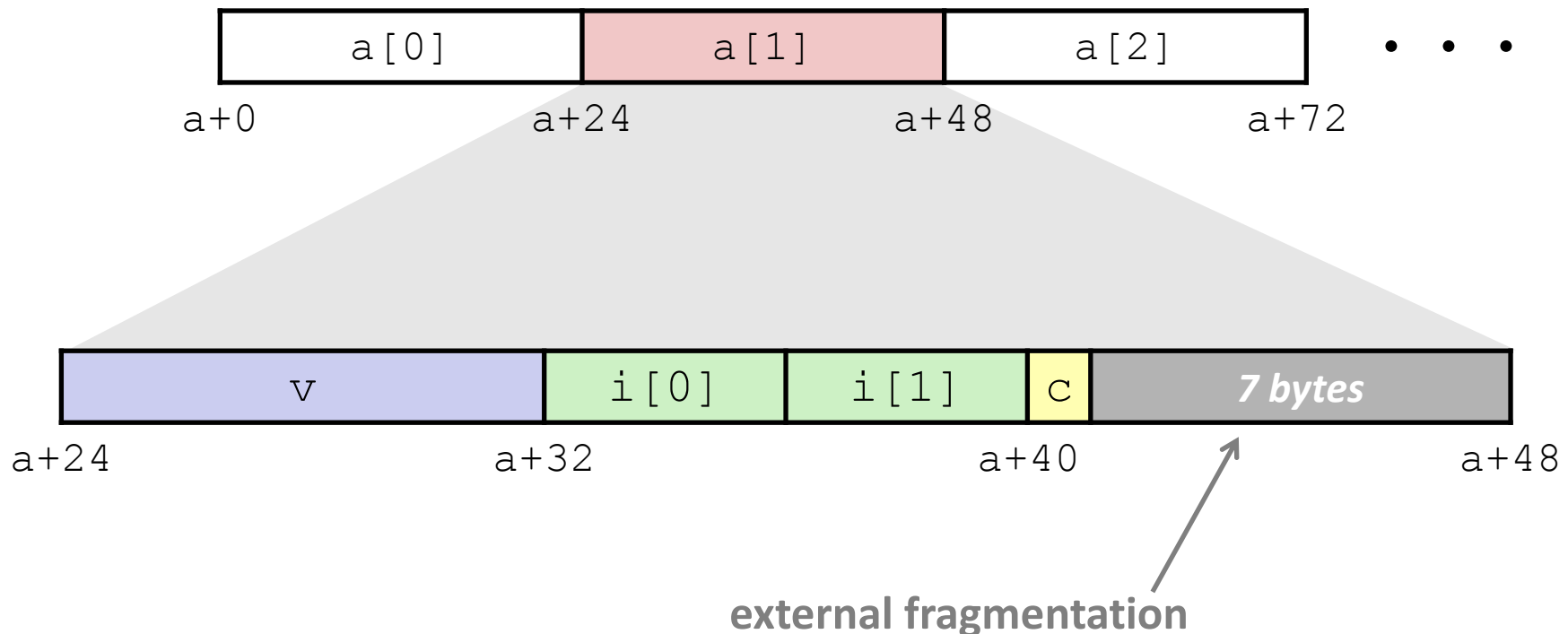
- Primitive data type requires K bytes
- Address must be multiple of K



Arrays of Structures

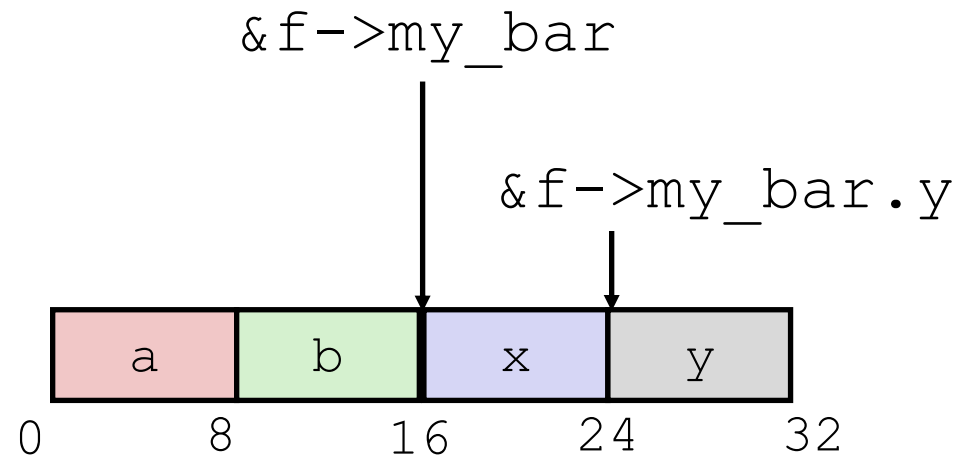
- ❖ Overall structure length multiple of K_{max}
- ❖ Satisfy alignment requirement for every element in array

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



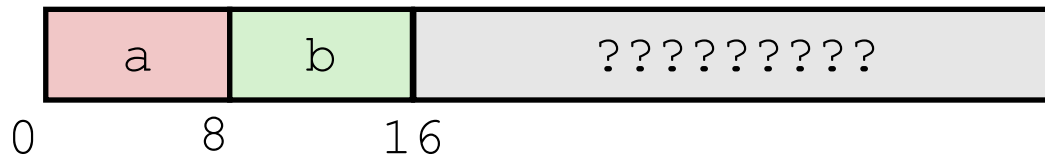
Nested Struct

```
struct foo {  
    long a;  
    long b;  
    struct bar my_bar;  
};  
  
struct bar {  
    long x;  
    long y;  
};  
  
struct foo *f;
```



Nested Struct

```
struct foo {  
    long a;  
    long b;  
    struct foo my_foo;  
};
```



Multiple Ways to Store Program Data

Multiple Ways to Store Program Data

❖ Static global data

- *Fixed size* at compile-time
- Entire *lifetime of the program* (loaded from executable)
- Portion is read-only (e.g. string literals)

```
int array[1024];

void foo(int n) {
    int tmp;
    int local_array[n];

    int* dyn =
        (int*)malloc(n*sizeof(int));
}
```

❖ Stack-allocated data

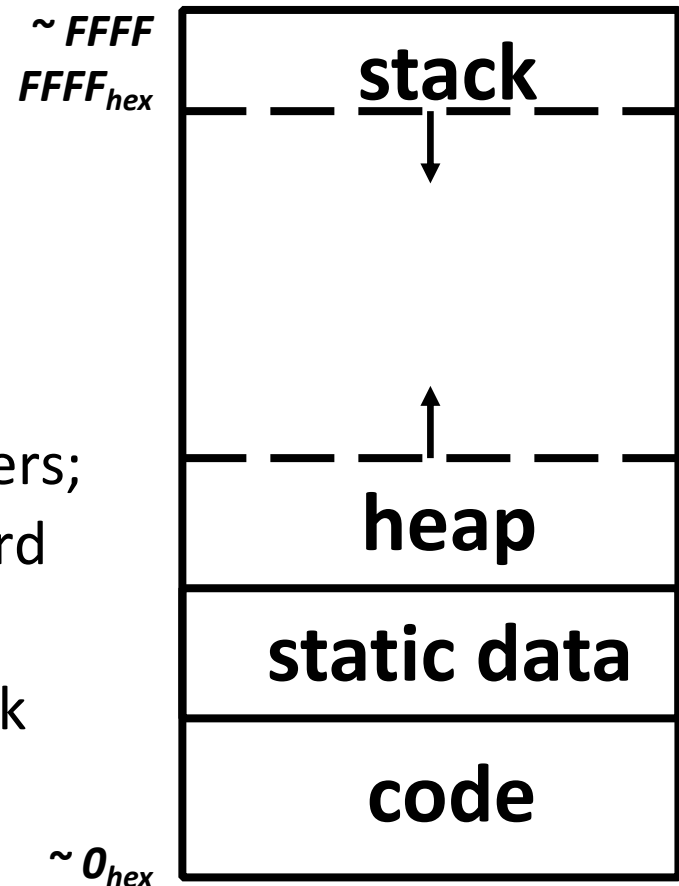
- Local/temporary variables
 - *Can* be dynamically sized (in some versions of C)
- *Known lifetime* (deallocated on `return`)

❖ **Dynamic (heap) data**

- Size known only at runtime (i.e. based on user-input)
- Lifetime known only at runtime (long-lived data structures)

C Memory Layout

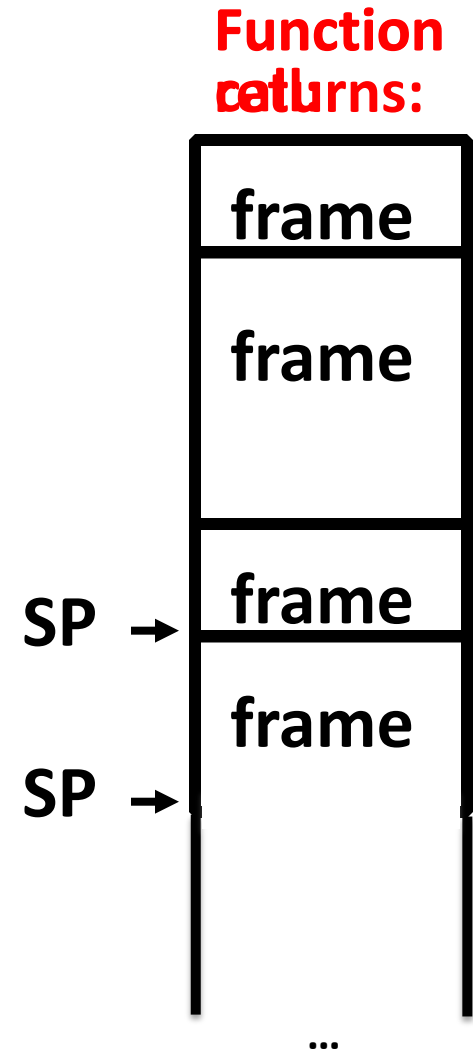
- Program's *address space* contains 4 regions:
 - **Stack**: local variables, grows downward
 - **Heap**: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
 - **Static Data**: global and static variables, does not grow or shrink
 - **Code**: loaded when program starts, does not change



*OS prevents accesses
between stack and heap
(via virtual memory)*

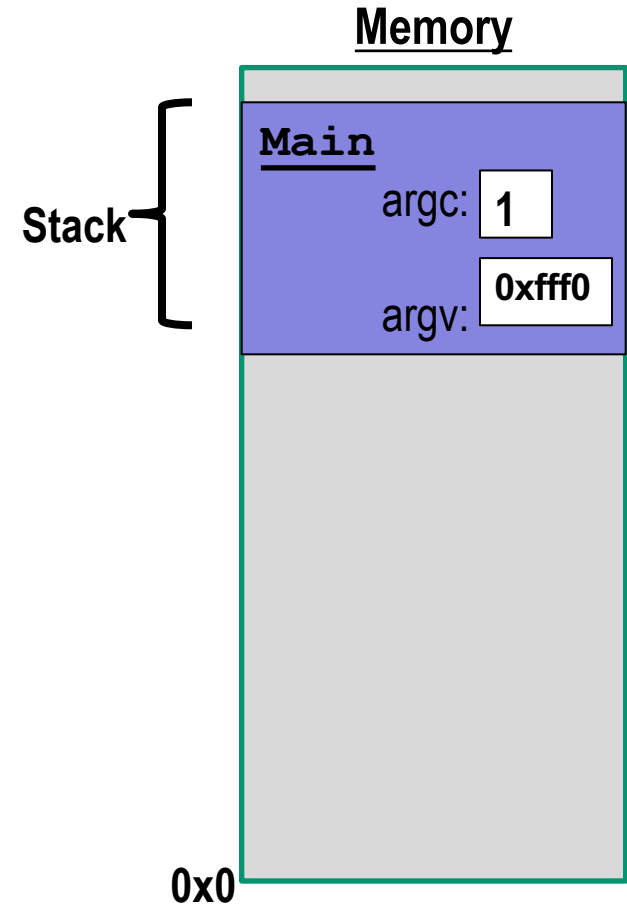
The Stack

- Each stack frame is a contiguous block of memory holding the local variables of a single procedure
- A stack frame includes:
 - Location of caller function
 - Function arguments
 - Space for local variables
- Stack pointer (SP) tells where lowest (current) stack frame is
- When procedure ends, stack pointer is moved back (but data remains (**garbage!**)); frees memory for future stack frames;



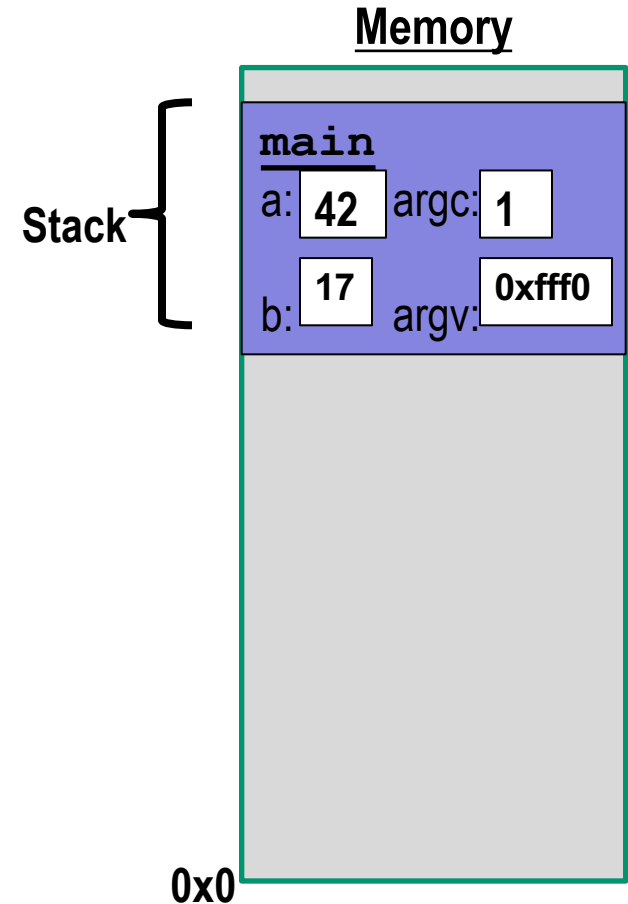
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



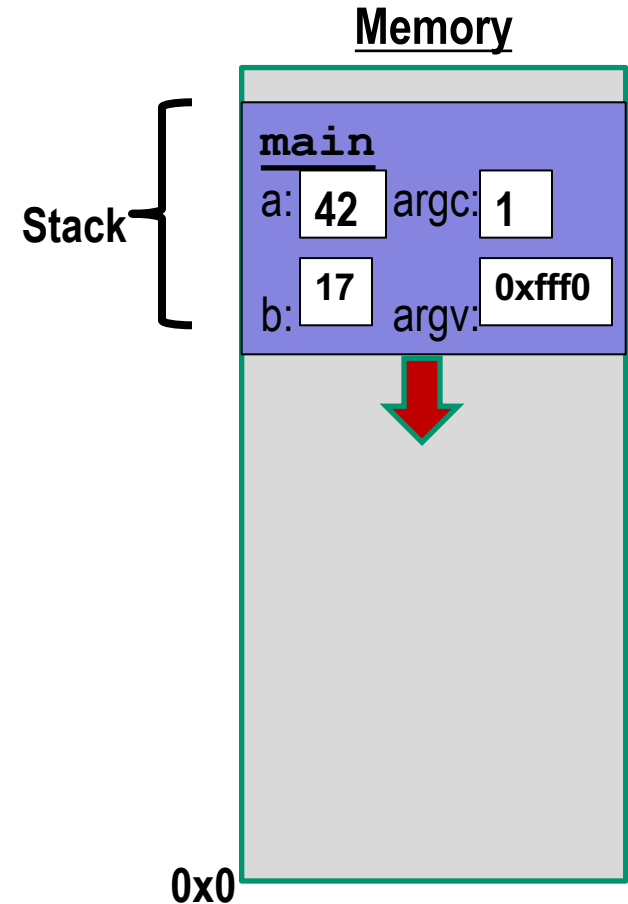
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



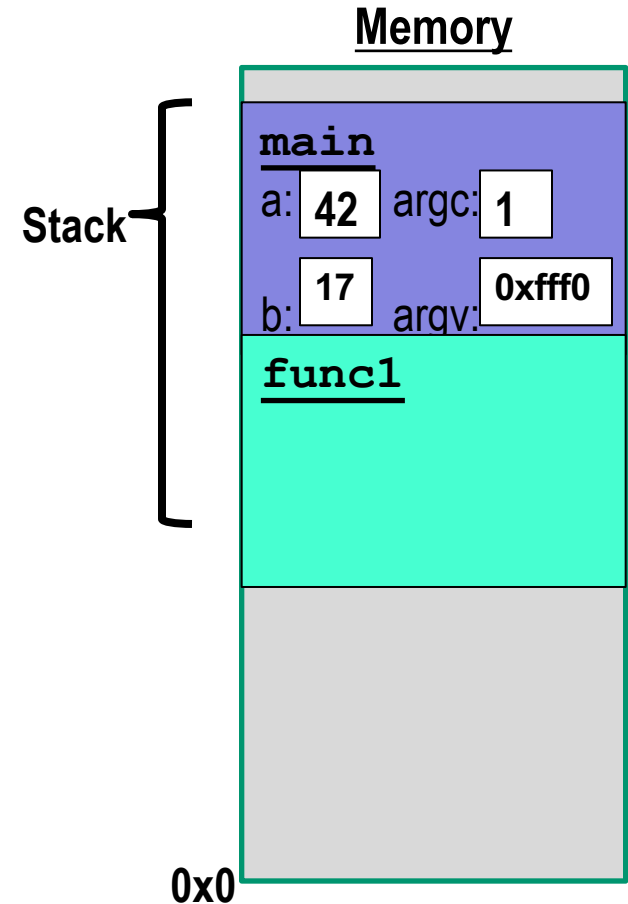
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



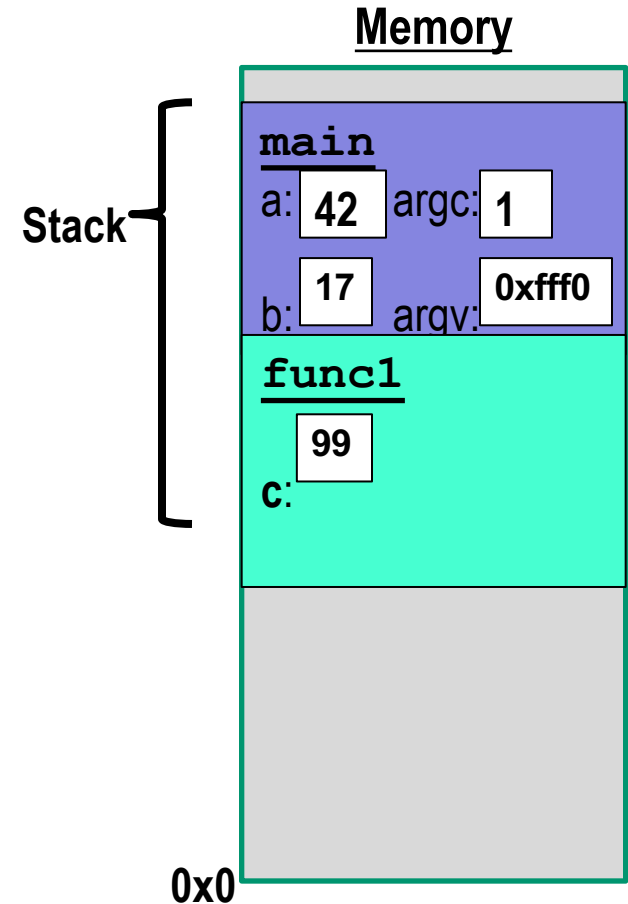
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



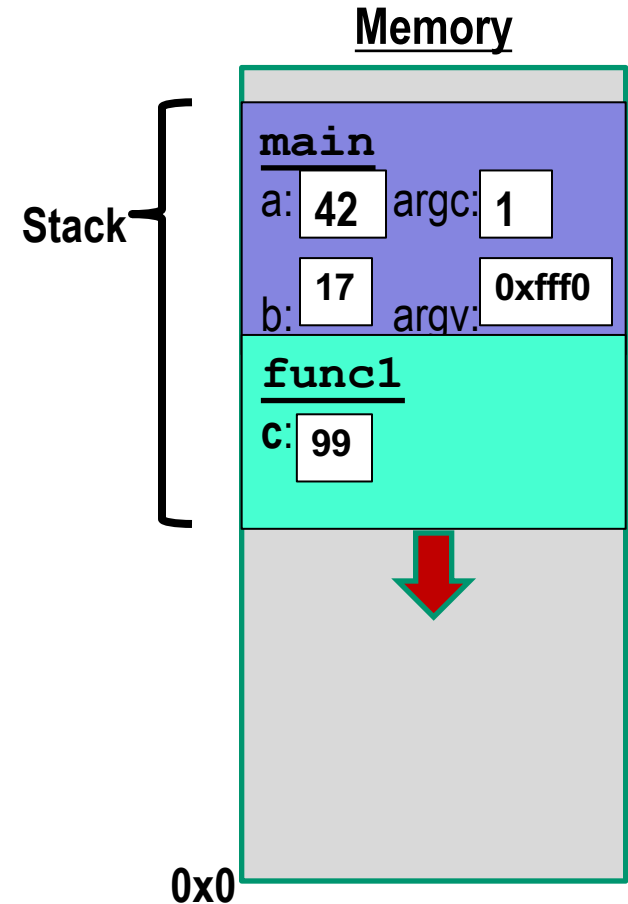
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



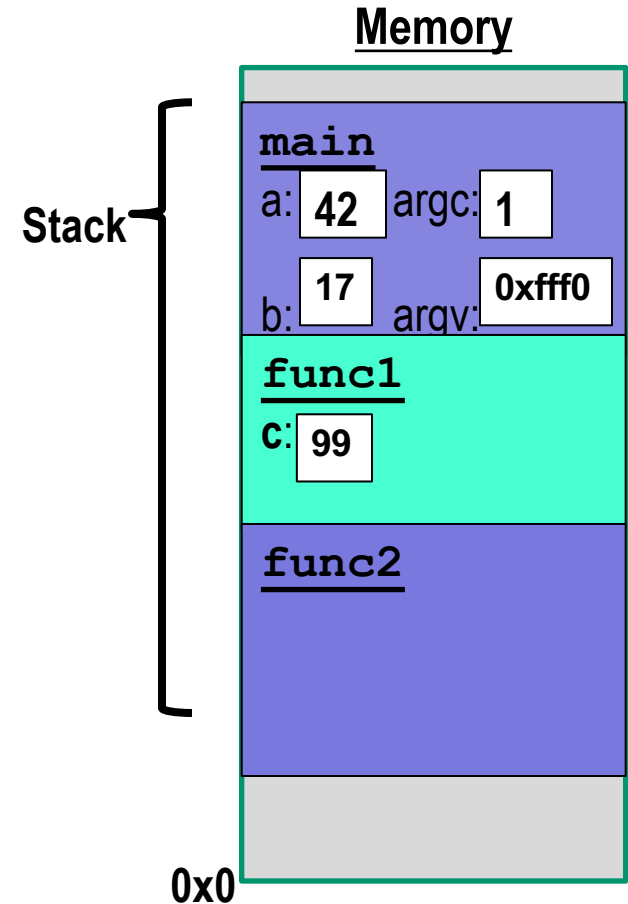
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



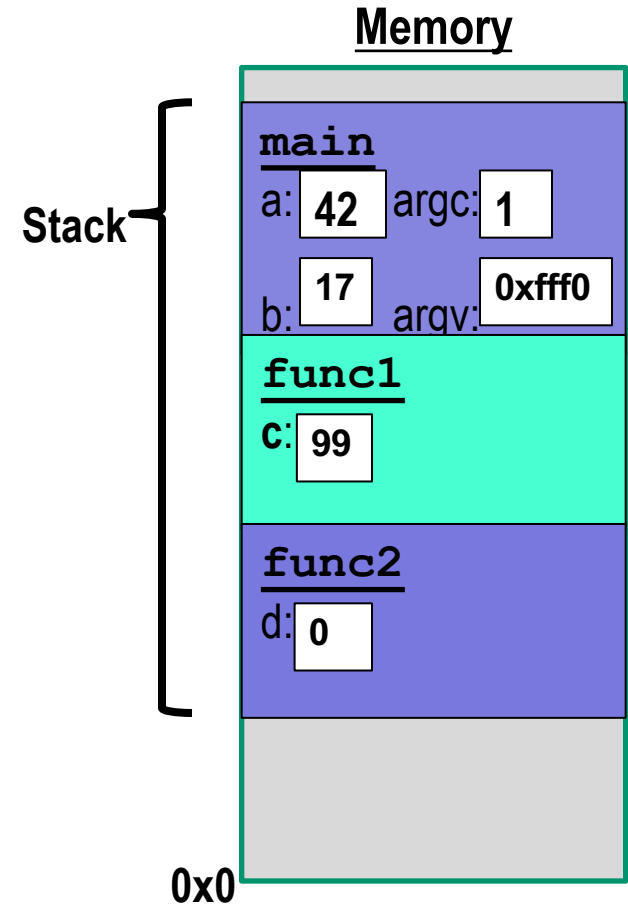
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



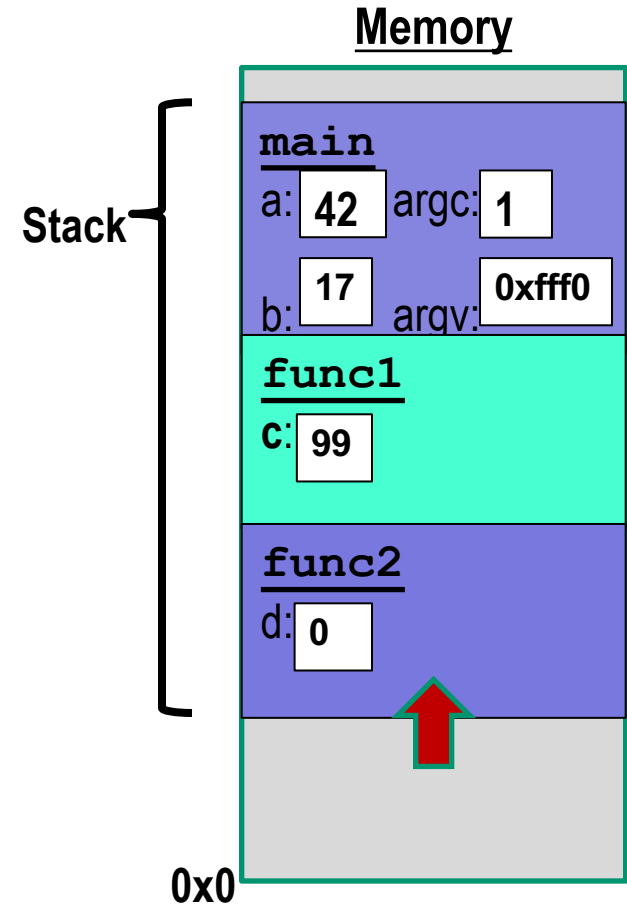
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



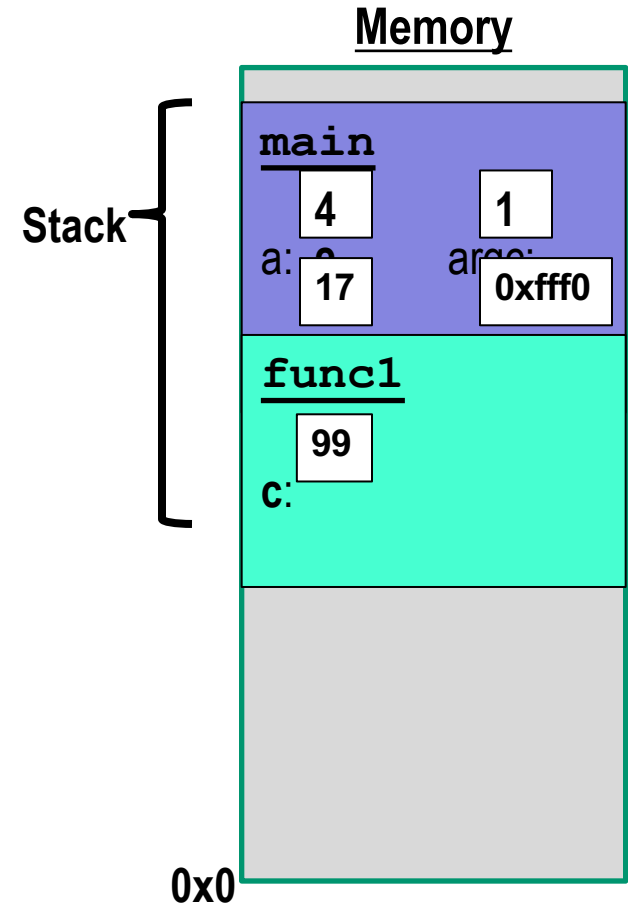
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



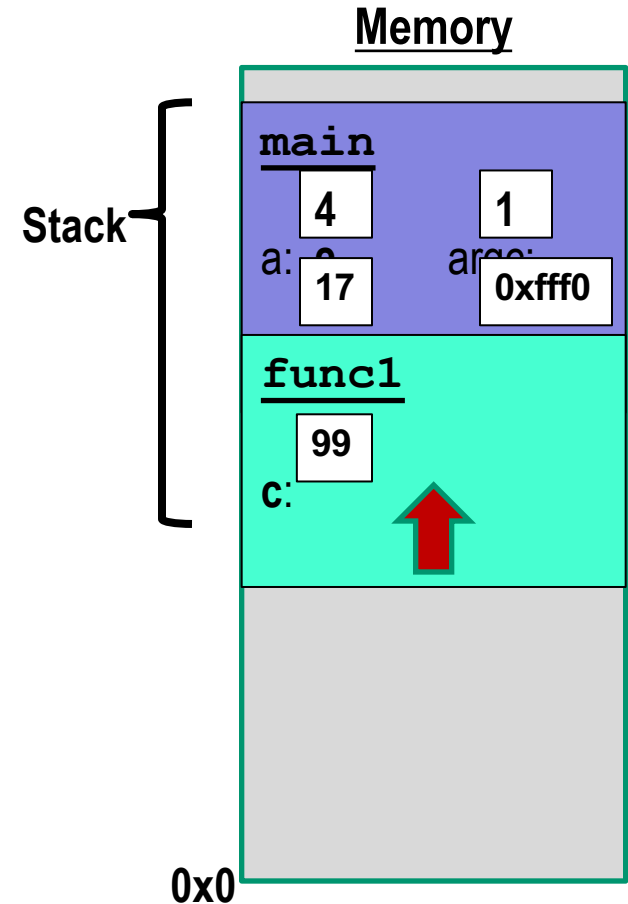
The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

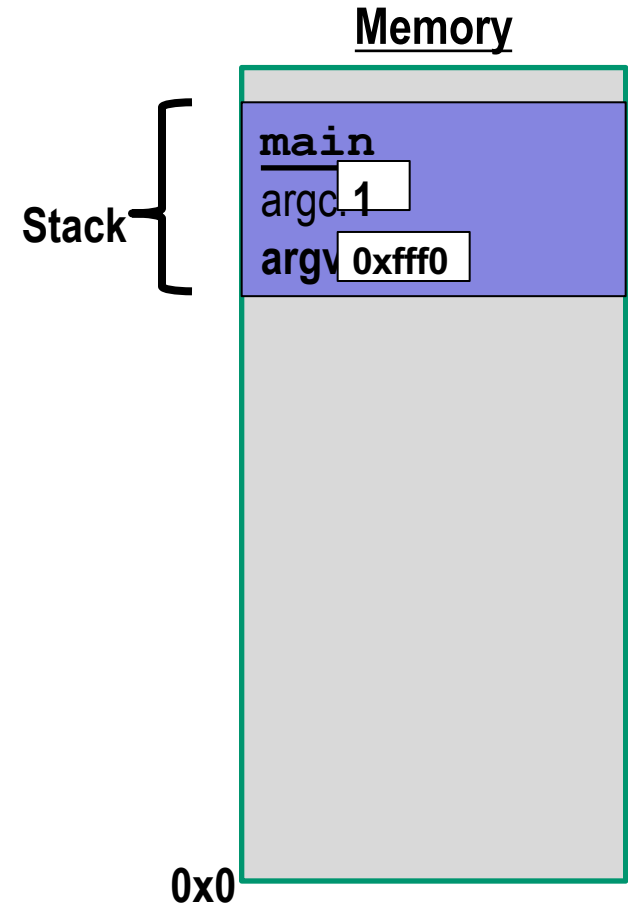
```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

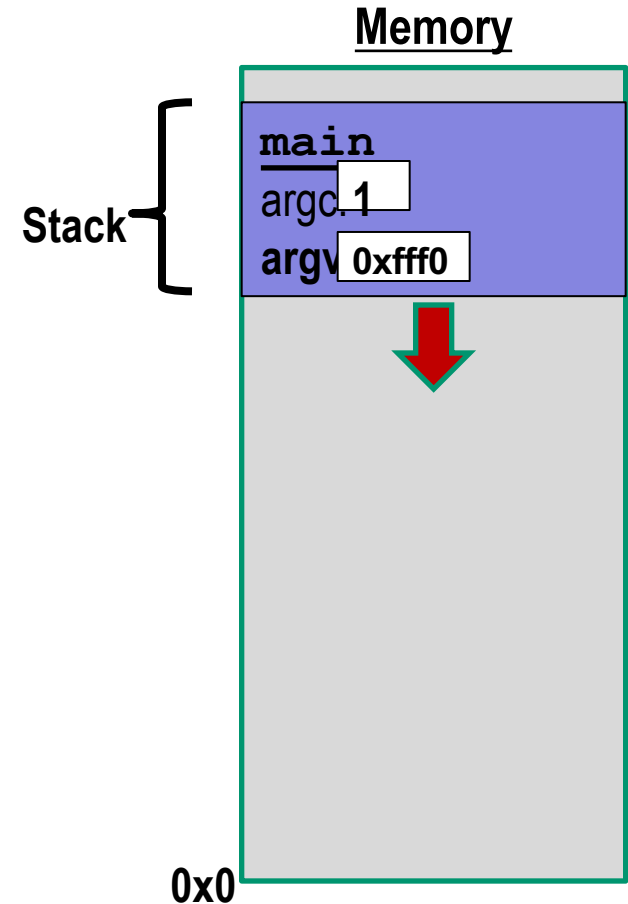
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

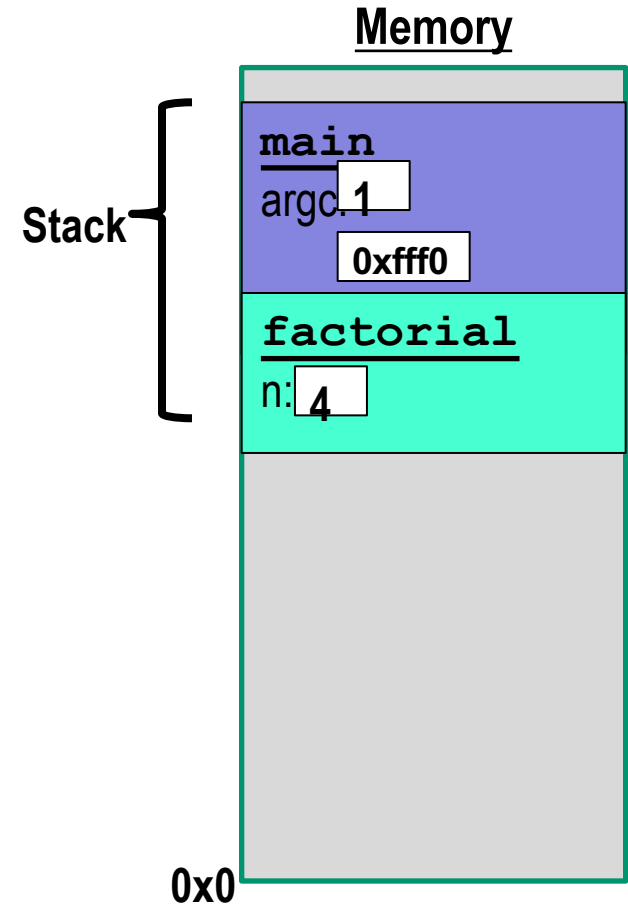
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

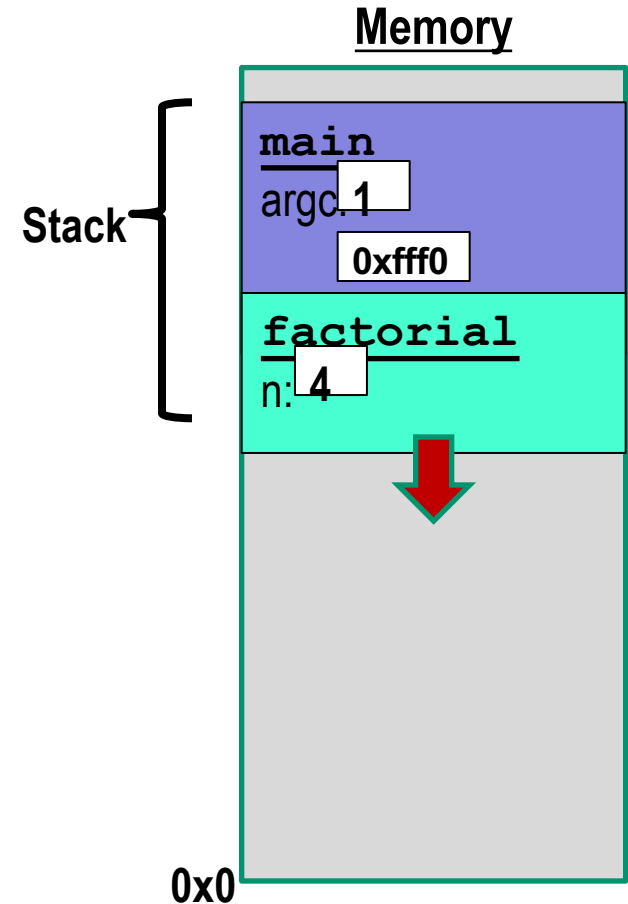
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

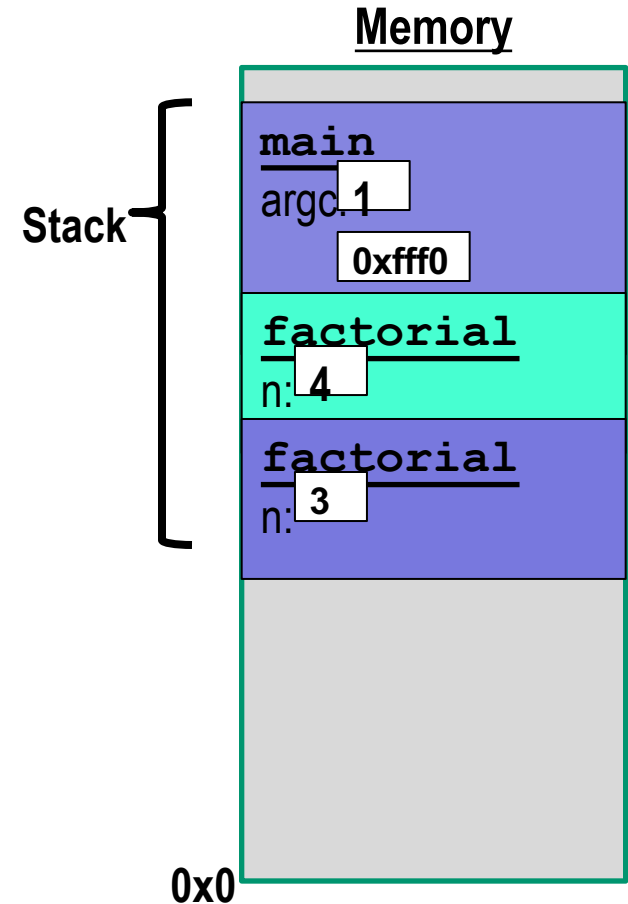
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

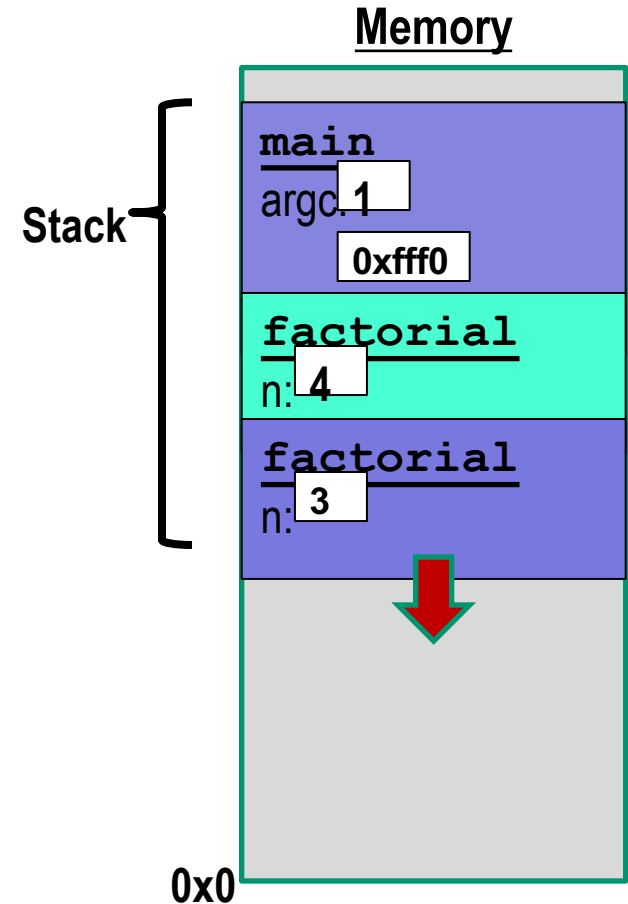
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

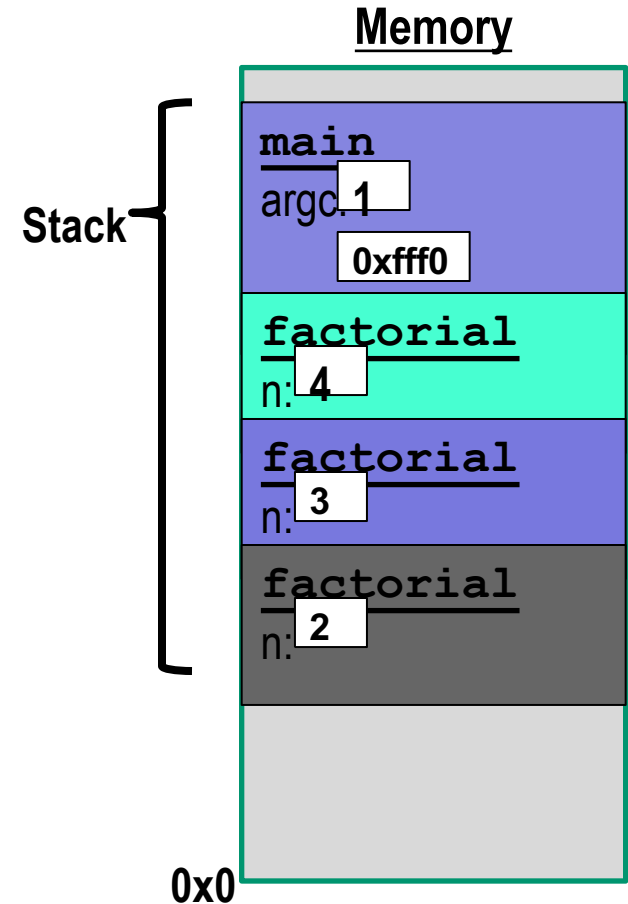
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

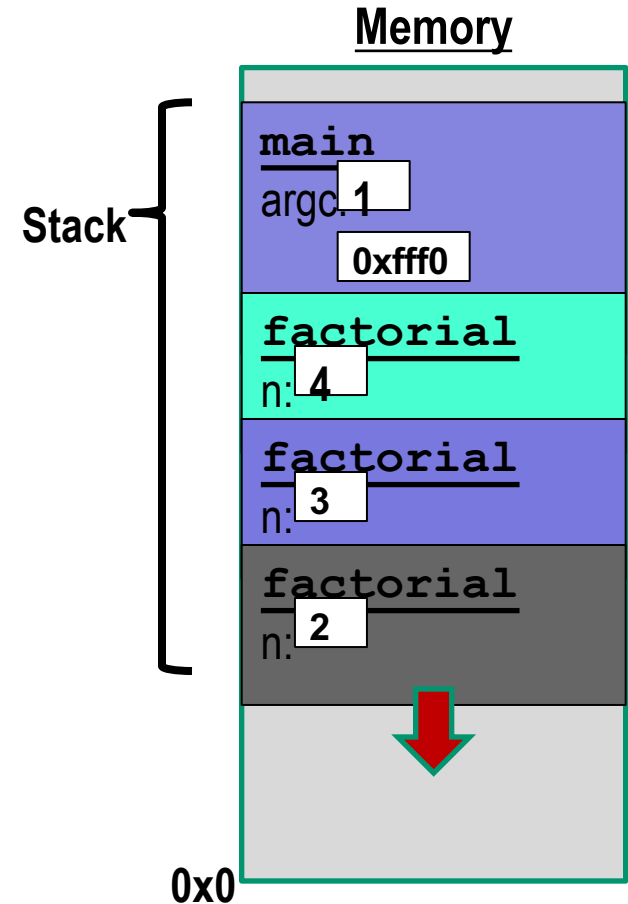
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

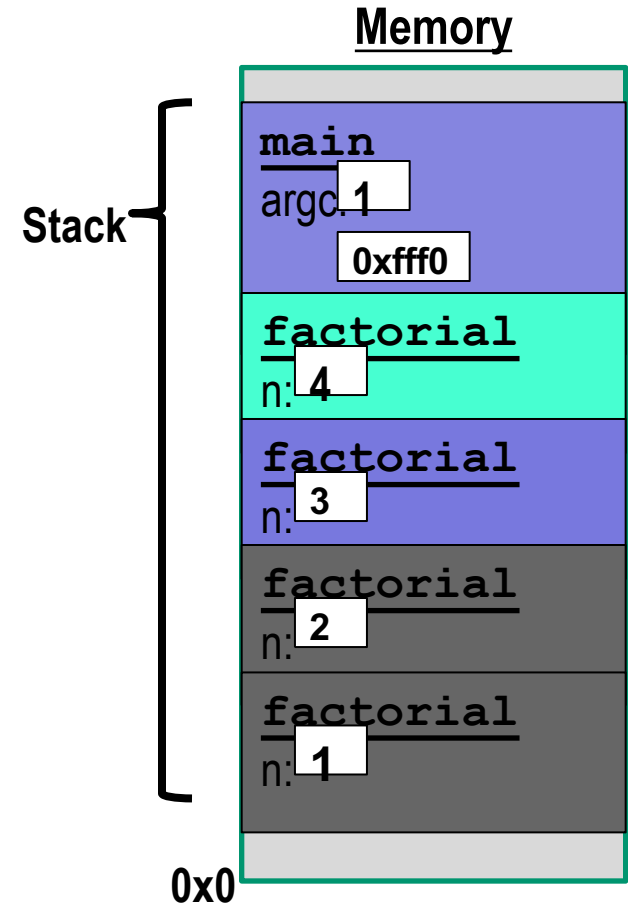
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

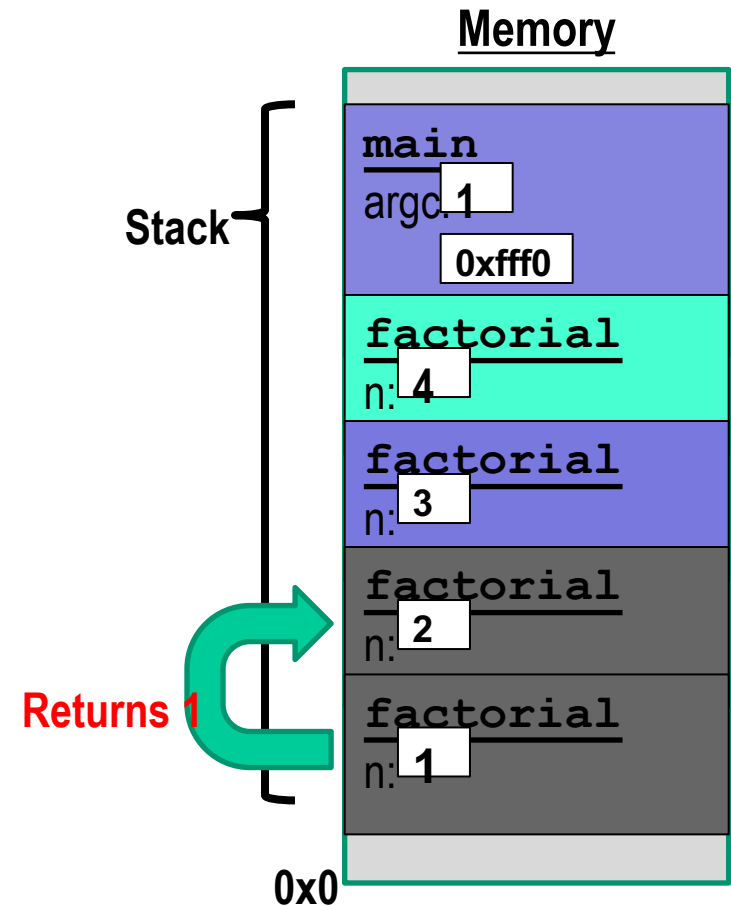
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

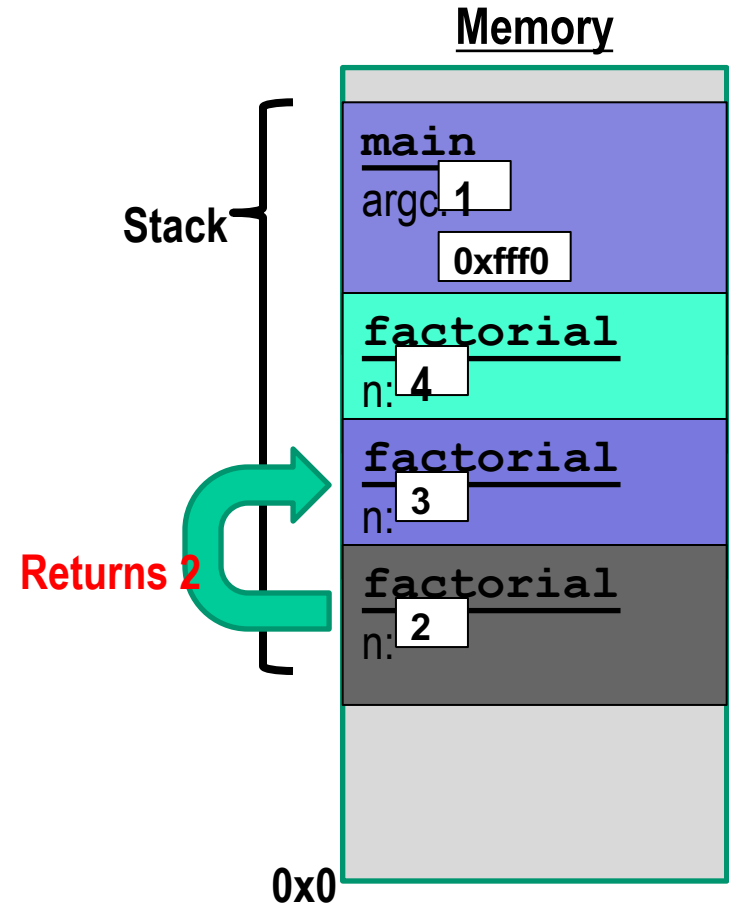
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

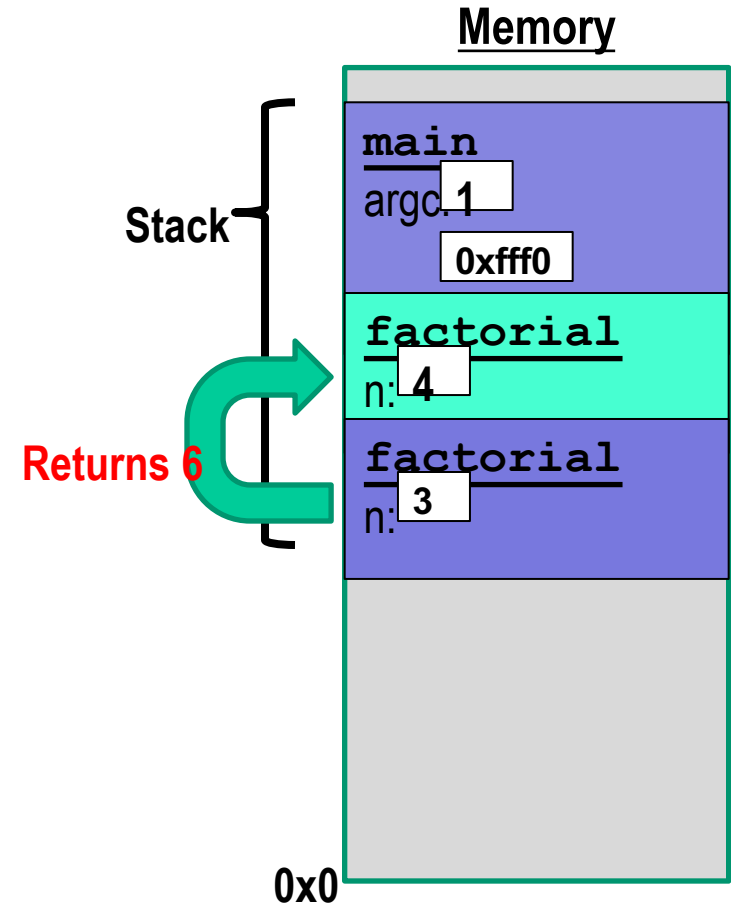
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```

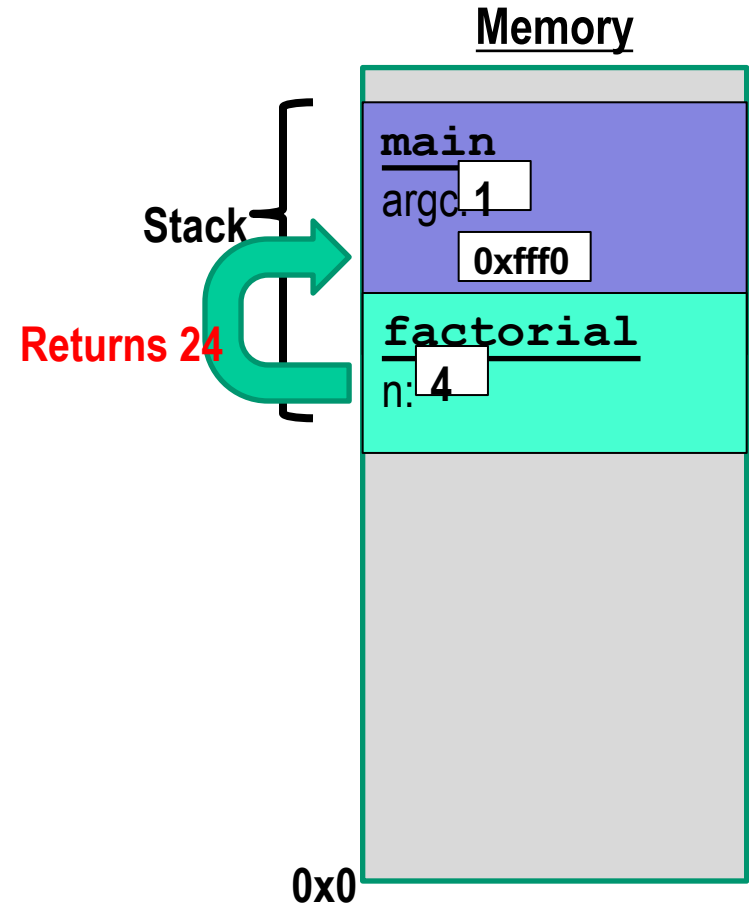


The Stack

Each function call has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

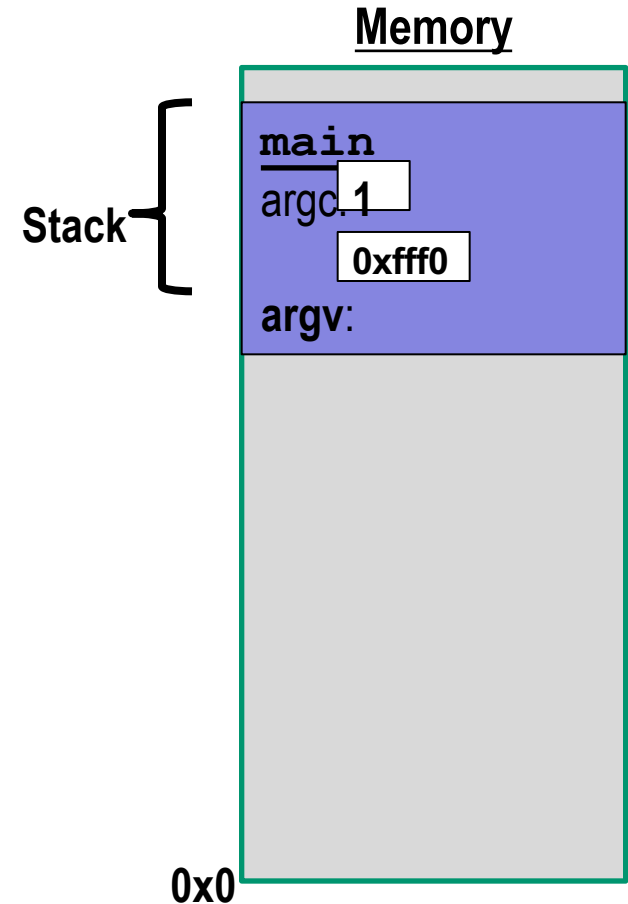
```
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

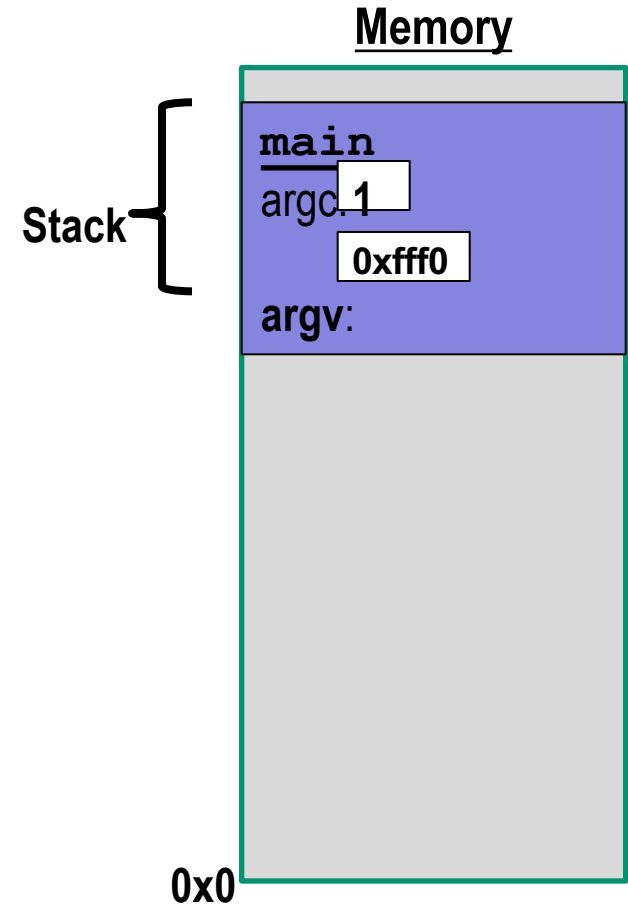
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

Each function call has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

- ❖ The stack behaves like a...well...stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.
- ❖ *Interesting fact:* C does not clear out memory when a function's frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!
- ❖ A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.
- ❖ What are the limitations of the stack?

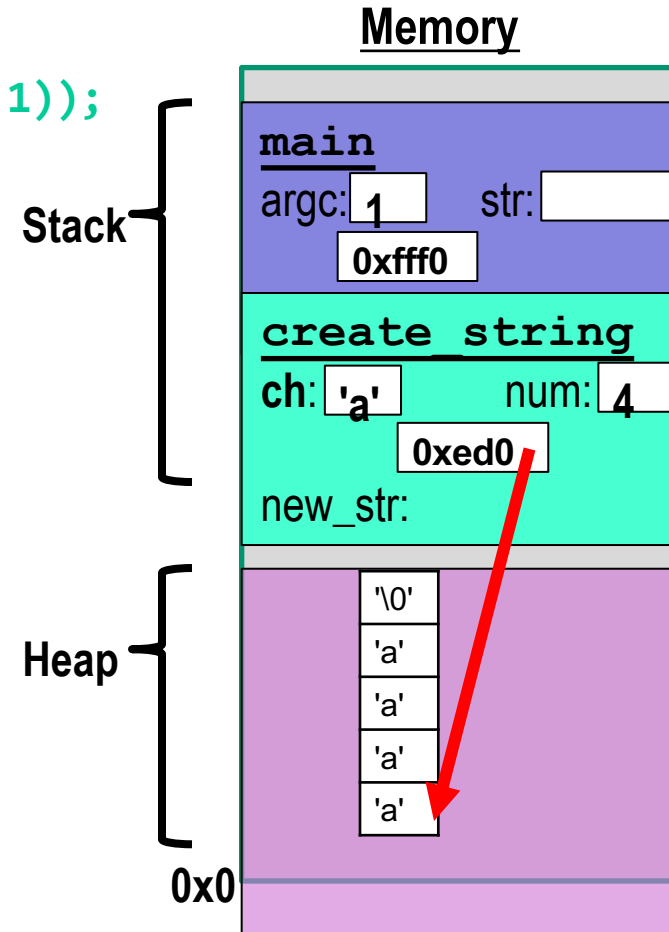
The Heap

- ❖ The **heap** is a part of memory that you can manage yourself.
- ❖ The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- ❖ Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.

The Heap

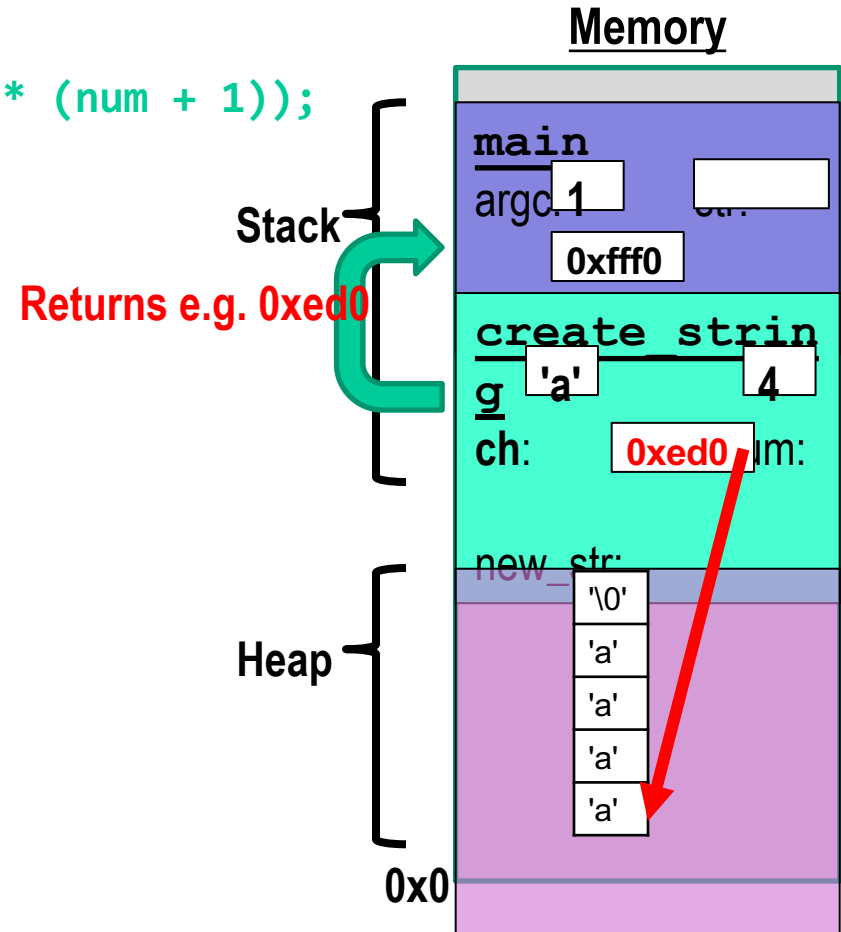
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str); // want "aaaa"
    return 0;
}
```



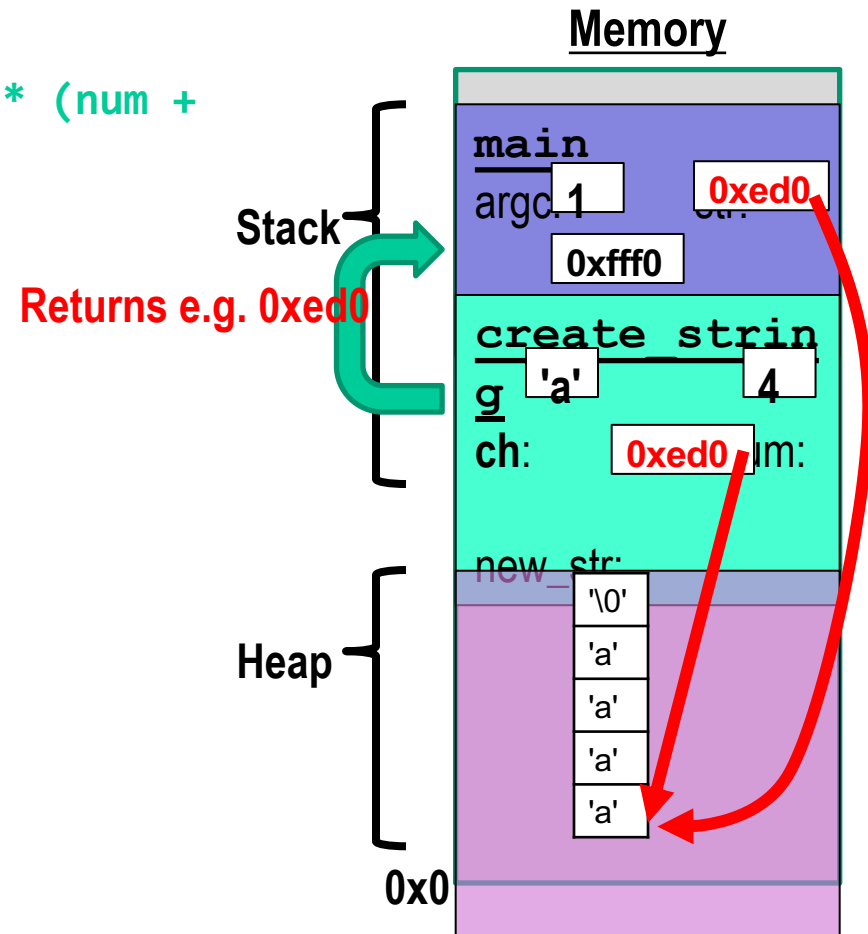
The Heap

```

char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num +
1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str); // want "aaaa"
    return 0;
}

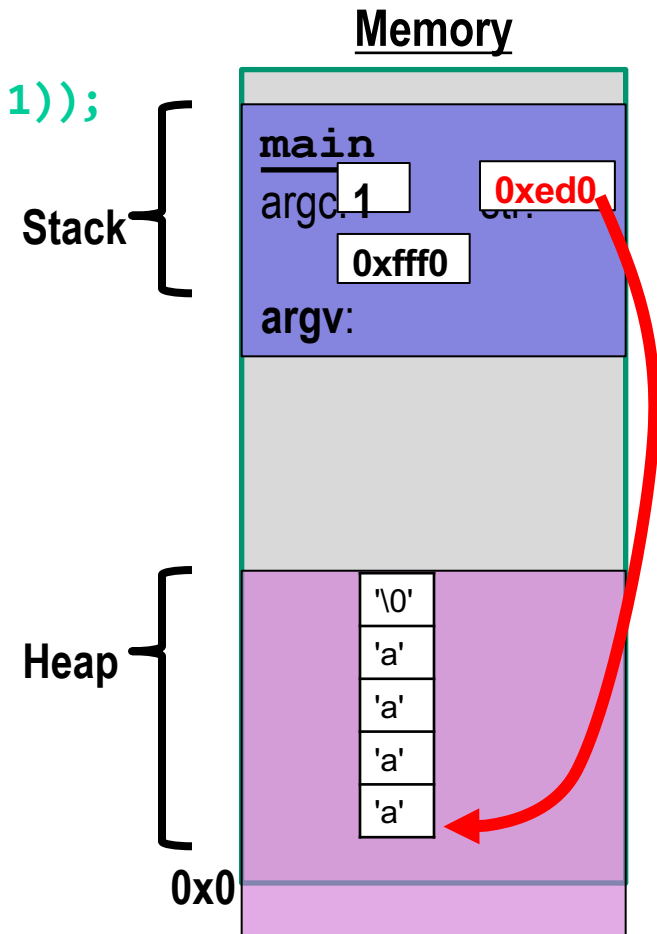
```



The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}
```

```
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str); // want "aaaa"
    return 0;
}
```



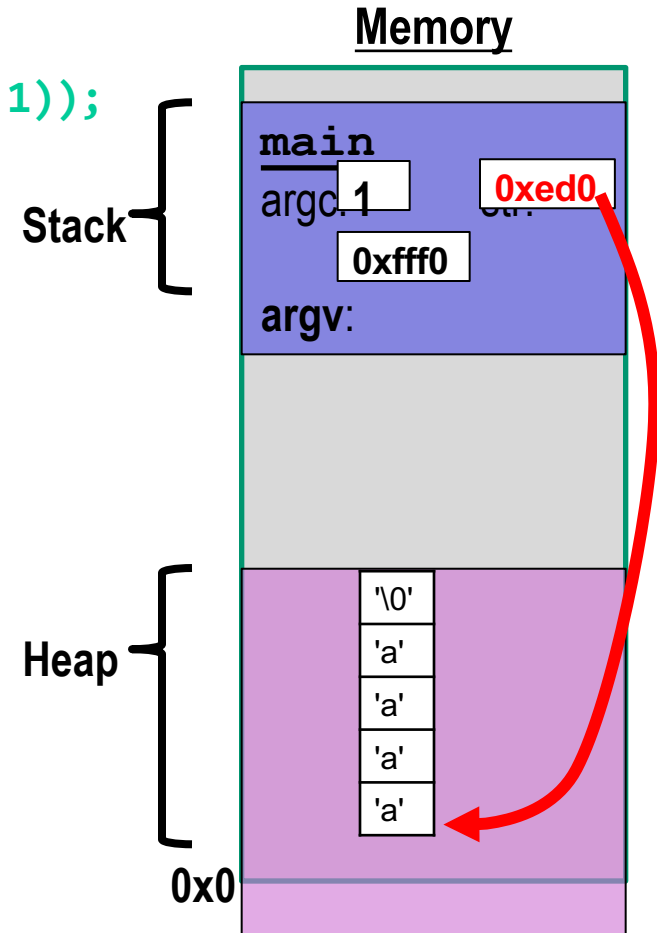
The Heap

```

char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str); // want "aaaa"
    return 0;
}

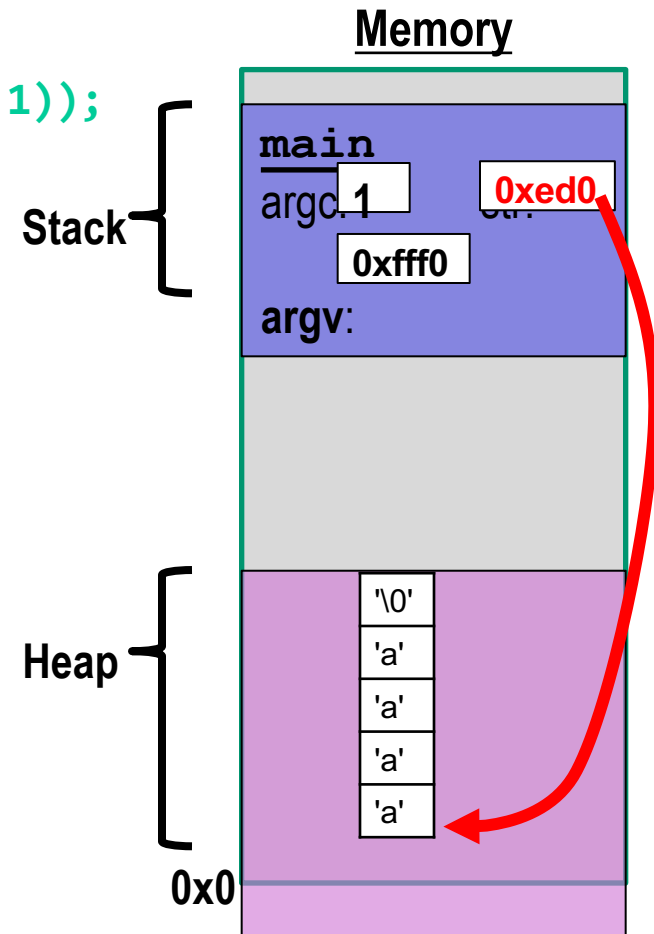
```



The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str); // want "aaaa"
    return 0;
}
```



Demo: Stack vs Heap Performance

Bit-Level Operations in C

❖ $\&$ (AND), $|$ (OR), $\wedge$ (XOR), $\sim$ (NOT)

- View arguments as bit vectors, apply operations bitwise
- Apply to any “integral” data type
 - long, int, short, char, unsigned

❖ Examples with char a, b, c;

- `a = (char) 0x41;` `// 0x41->0b 0100 0001`
`b = ~a;` `//                    0b                    ->0x`
- `a = (char) 0x69;` `// 0x69->0b 0110 1001`
`b = (char) 0x55;` `// 0x55->0b 0101 0101`
`c = a & b;` `//                    0b                    ->0x`
- `a = (char) 0x41;` `// 0x41->0b 0100 0001`
`b = a;` `//                    0b 0100 0001`
`c = a ^ b;` `//                    0b                    ->0x`

NOT

a	$\sim a$
0	1
1	0

AND

a	b	$a \& b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \text{ OR } b$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

a	b	$a \wedge b$
0	0	0
0	1	1
1	0	1
1	1	0

Demo: Bit vector vs Bool Array

Intuition for logical operators

Operator	Input	Output
& 1 : Copy	1001 & 1111	1001
 1 : Set	1001 1111	1111
& 0 : Reset	1001 & 0000	0000
 0 : Copy	1001 0000	1001
^ 1 : Flip	1001 ^ 1111	0110
^ : Match	1001 ^ 1001	0000

e.g., extract bits at position 3 and 2 from number
 number & 1100 - Copy positions 3 and 2. Reset positions 1 and 0
 1001 & 1100 = 1000

1000  --10

Shift Operations (also watch Lab 3 video)

❖ Left shift ($x \ll n$)

- Fill with 0s on right

❖ Right shift ($x \gg n$)

- Logical shift (for **unsigned** values)
 - Fill with 0s on left

- Arithmetic shift (for **signed** values)
 - Replicate most significant bit on left

❖ Notes:

- Shifts by $n < 0$ or $n \geq w$ (bit width of x) are *undefined*
- **In C:** behavior of $\gg$ is determined by compiler
 - depends on data type of x (signed/unsigned)
- **In Java:** logical shift is $\ggg$ and arithmetic shift is $\gg$

x	0010 0010
$x \ll 3$	0001 0 000
logical: $x \gg 2$	00 00 1000
arithmetic: $x \gg 2$	00 00 1000

x	1010 0010
$x \ll 3$	0001 0 000
logical: $x \gg 2$	00 10 1000
arithmetic: $x \gg 2$	11 10 1000

Extract and Concat

- ❖ Two numbers: A: 0xC(1101) and B: 0xA (1010)
- ❖ Question: Extract bits 3 and 2, and concatenate
- ❖ i.e., 1101 concat 1010 -> 1110 (red: bits of interest, black bits are dropped)
- ❖ Step 1: Extract bits from each
 - $1101 \& (3 \ll 2) = 1101 \& (0b0011 \ll 2) = 1101 \& 1100 = 1100$
 - $1010 \& (3 \ll 2) = 1010 \& (0b0011 \ll 2) = 1010 \& 1100 = 1000$
- ❖ Step 2: Prepare for concat:
 - $1100 \gg 2 = 0011$ (A), $1000 \gg 2 = 0010$ (B)
- ❖ Step 3: Concat: A.B. Place B in 0th position, Place A in 2nd bit position = $0011 \ll 2 \mid 0010 \Rightarrow 1110$

C bitfields vs. Packing

```
struct BitFields {
    unsigned int bit1 : 1;
    unsigned int bit2 : 1;
    unsigned int bit3 : 1;
};
Bitfield packed
packed.bit3 = 1
```

```
unsigned int packed = 0;
// Example 1: Setting 3rd bit
// Shift + OR
packed = packed | (1 << 2);
```

	Bitfields	Int packing
Space	Compiler chooses	User chooses
Speed	Potentially slower; compiler generates.	Generally faster; user creates.
Readability	Yes; like struct	No; very low level
Use Case	Network packets	Instruction rep

Printing bitfield vs packed

❖ Cast and print

```
Bitfield packed
packed.bit3 = 1
printf("%u",
(unsigned int) packed.bit3);
```

❖ Move and print

```
unsigned int packed = 0;
// Example 1: Setting 3rd bit
// Copy down bit3
bit3 = packed & (1 << 2);
// Move it the LSB.
Result = bit3 >> 2;
printf("%u",
(unsigned int) result);
```

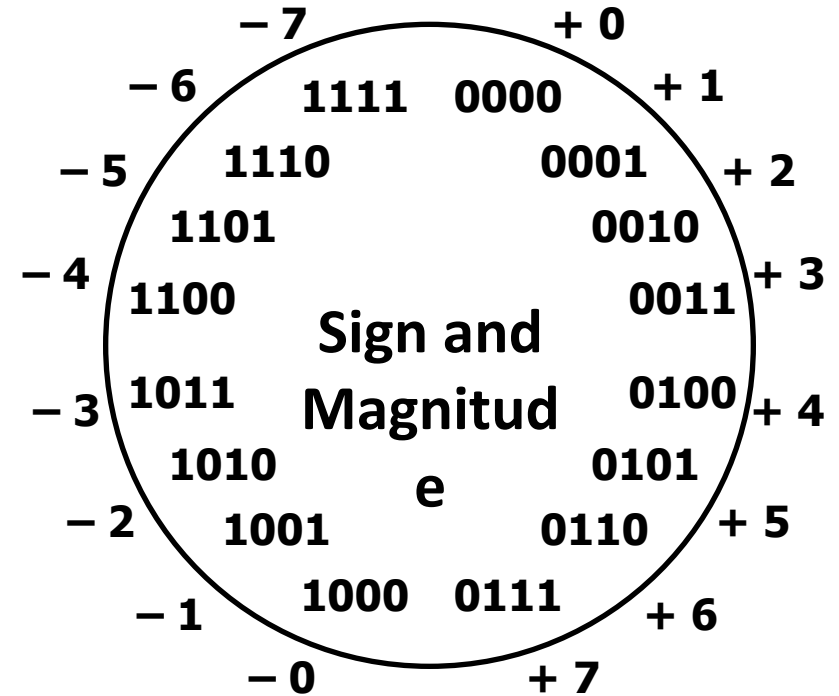
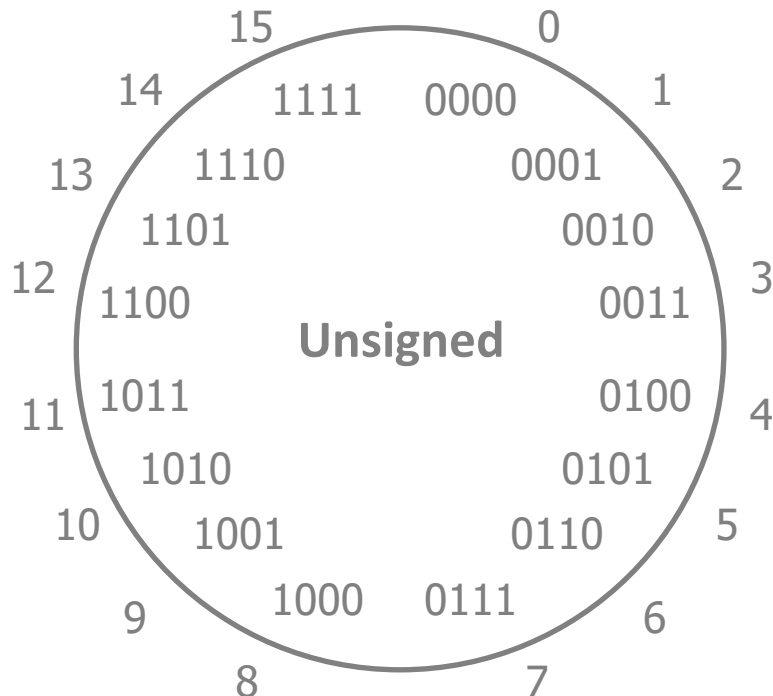
<https://replit.com/@ashriram/Extract-and-Print#main.c>

Integers

- ❖ Sign and unsigned variables in C
 - Bit pattern remains the same, just *interpreted* differently
 - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
 - Type of variables affects behavior of operators (shifting, comparison)
- ❖ We can only represent so many numbers in w bits
 - When we exceed the limits, *arithmetic overflow* occurs
 - *Sign extension* tries to preserve value when expanding
- ❖ Shifting is a useful bitwise operator
 - Right shifting can be arithmetic (sign) or logical (0)
 - Can be used in multiplication with constant or bit masking

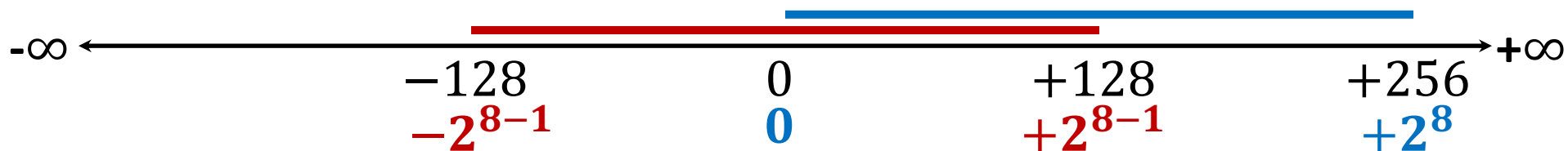
Sign and Magnitude

- ❖ MSB is the sign bit, rest of the bits are magnitude
- ❖ Drawbacks?



Encoding Integers

- ❖ The hardware (and C) supports two flavors of integers
 - *unsigned* – only the non-negatives
 - *signed* – both negatives and non-negatives
- ❖ Cannot represent all integers with w bits
 - Only 2^w distinct bit patterns
 - Unsigned values: $0 \dots 2^w - 1$
 - Signed values: $-2^{(w-1)} \dots 0 \dots 2^{(w-1)} - 1$
- ❖ **Example:** 8-bit integers (*e.g.* `char`)



Sign and Magnitude

❖ MSB is the sign bit, rest of the bits are magnitude

❖ Drawbacks:

■ Two representations of 0 (bad for checking equality)

■ **Arithmetic is cumbersome**

• Example: $4 - 3 \neq 4 + (-3)$

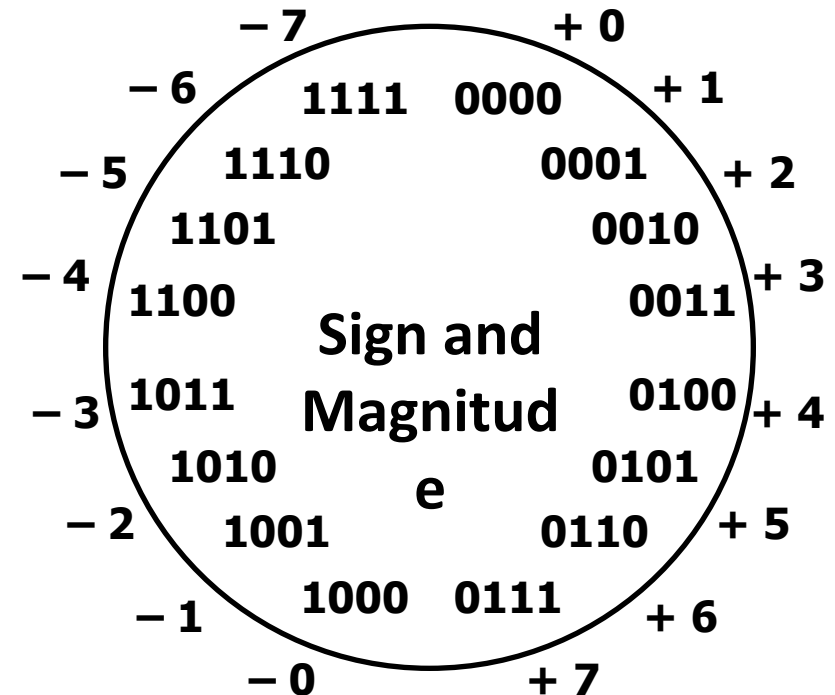
4	0100
- 3	- 0011
1	0001



4	0100
+ -3	+ 1011
-7	1111

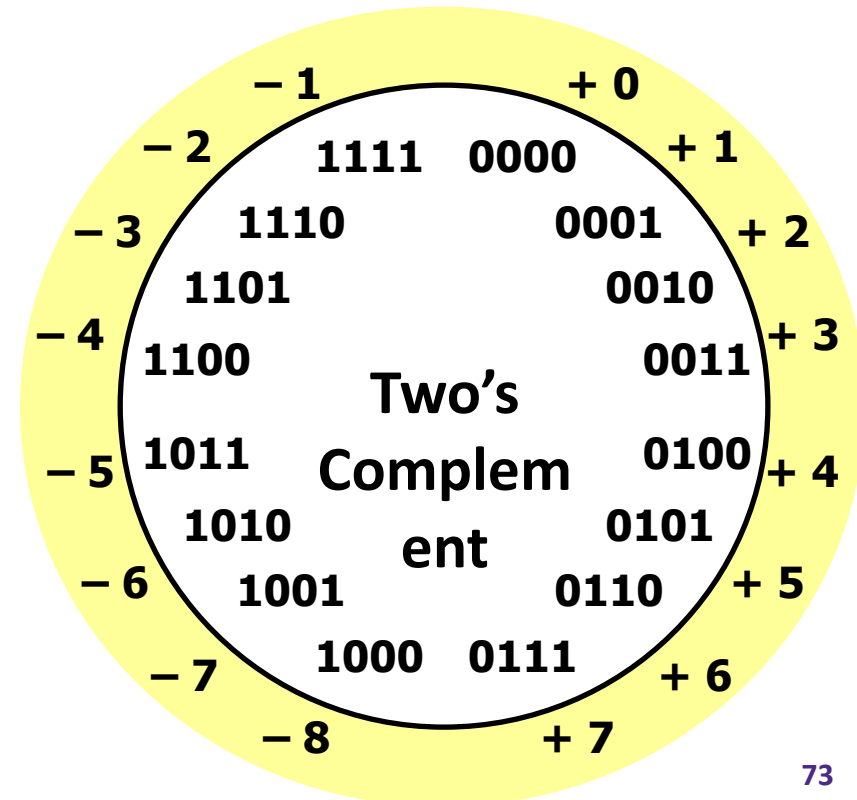


• Negatives “increment” in wrong direction!



Why Two's Complement is So Great

- ❖ Roughly same number of (+) and (−) numbers
 - ❖ Positive number encodings match unsigned
 - ❖ Simple arithmetic ($x + -y = x - y$)
 - ❖ Single zero
 - ❖ All zeros encoding = 0
 - ❖ Simple negation procedure:
 - Get negative representation of any integer by taking bitwise complement and then adding one!
- (~x + 1 == -x)



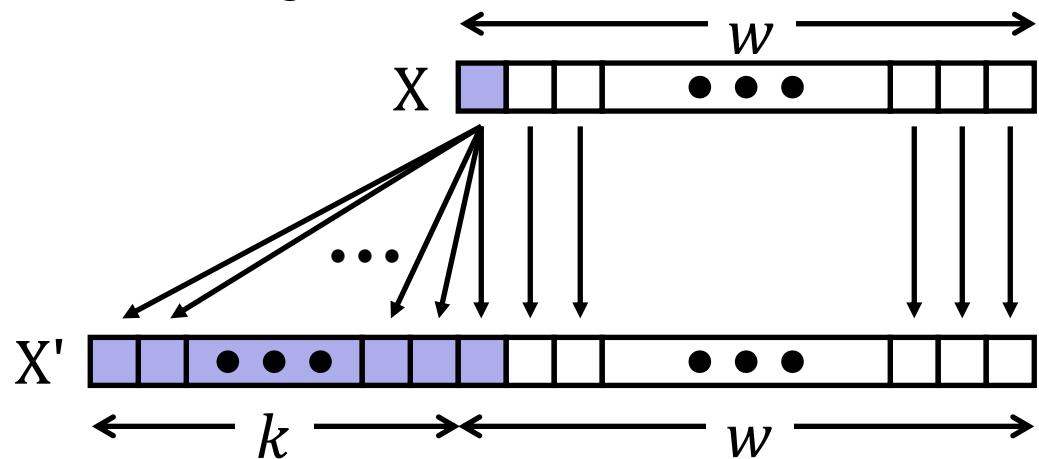
Sign Extension

❖ **Task:** Given a w -bit signed integer X , convert it to $w+k$ -bit signed integer X' *with the same value*

❖ **Rule:** Add k copies of sign bit

■ Let x_i be the i -th digit of X in binary

■ $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \dots, x_1, x_0}_{\text{original } X}$



45466