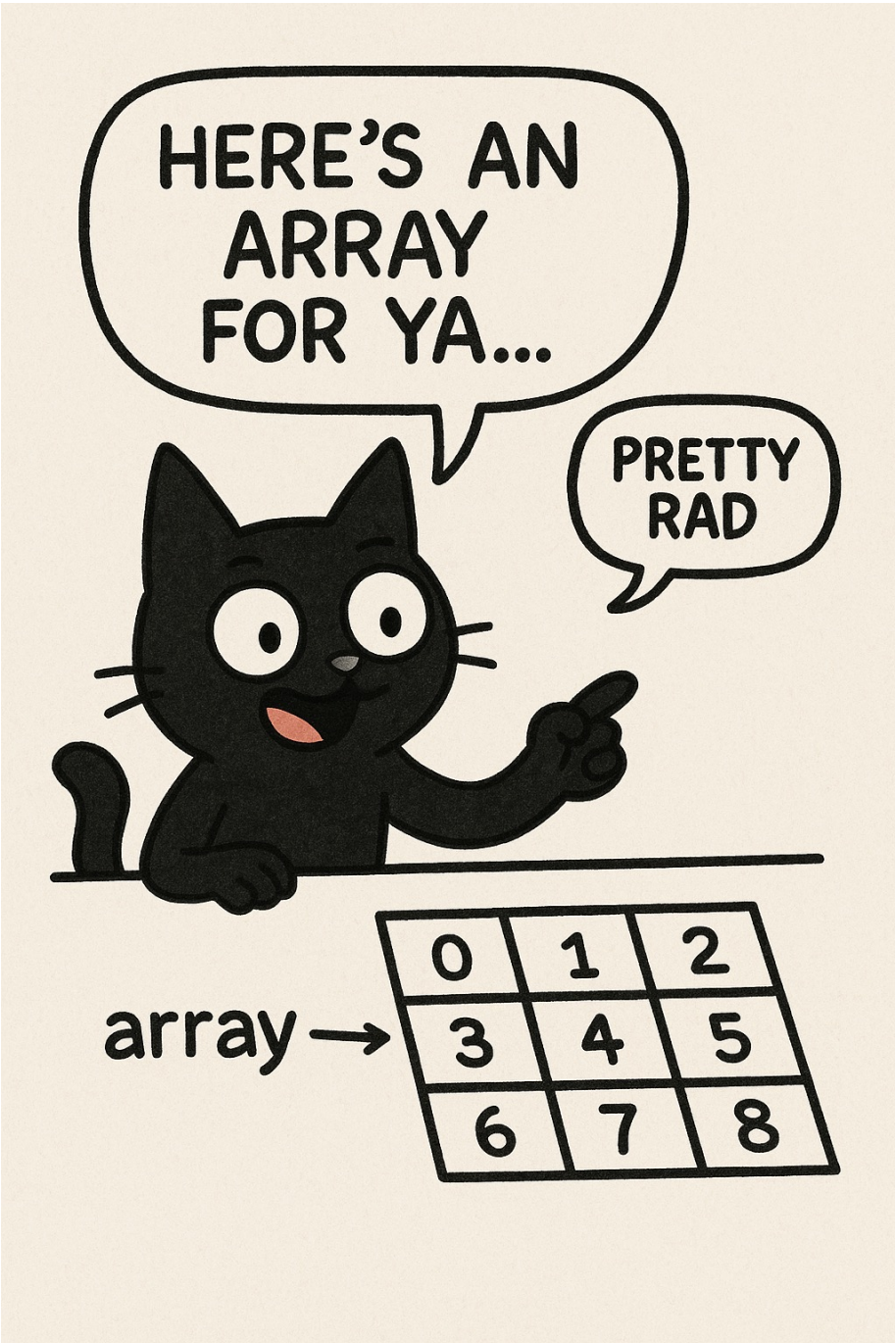
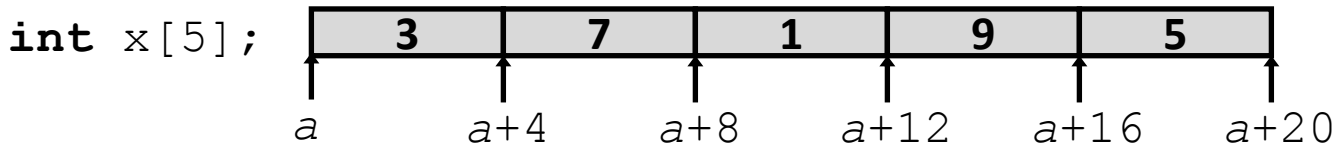
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - **Multi-level**
- Structs
  - Alignment


HERE'S AN ARRAY FOR YA...
PRETTY RAD
array
0 1 2
3 4 5
6 7 8

Array Access

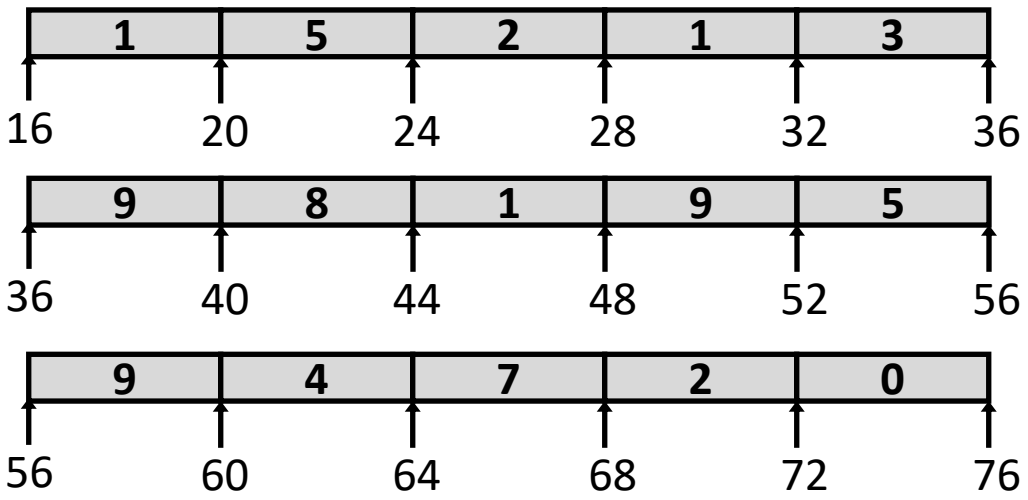
- Basic Principle
  - **T** A[N]; → array of data type **T** and length N
  - Identifier A returns address of array (type **T***)



Reference	Type	Value
x[4]	**int**	5
x	**int***	a
x+1	**int***	a + 4
&x[2]	**int***	a + 8
x[5]	**int**	?? (whatever's in memory at addr x+20)
*(x+1)	**int**	7
x+i	**int***	a + 4*i

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

Referencing Examples



Reference	Address	Value	Guaranteed?
`sfu[3]`	`36 + 4* 3 = 48`	`9`	Yes
`sfu[6]`	`36 + 4* 6 = 60`	`4`	No
`sfu[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`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

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

Nested Array Example

```
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:

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

What is the layout in memory?

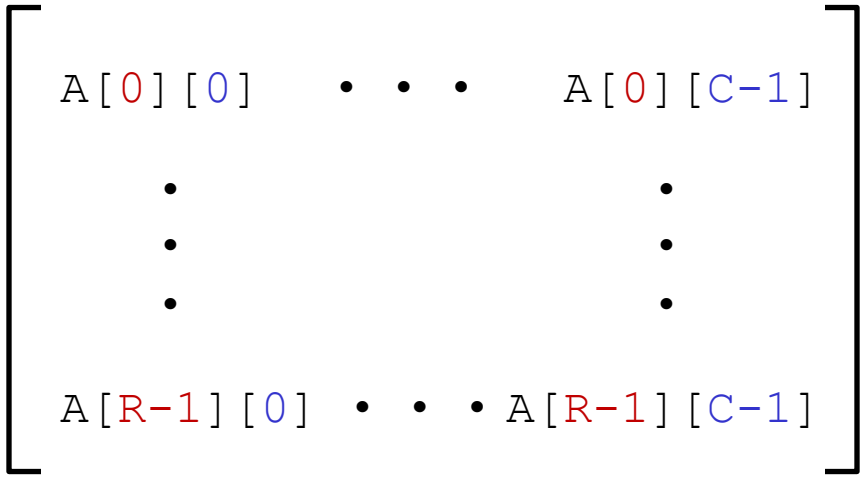
```
int sea[4][5];
```

How is a 2D array packed into memory?

1D Array	2D Array
int array[10]; Elements in sequence	int array[10][10]; What is the layout?
a[i] = &a[0]+i Byte offset: i*sizeof(int)	a[i][j] ? Byte offset: ?

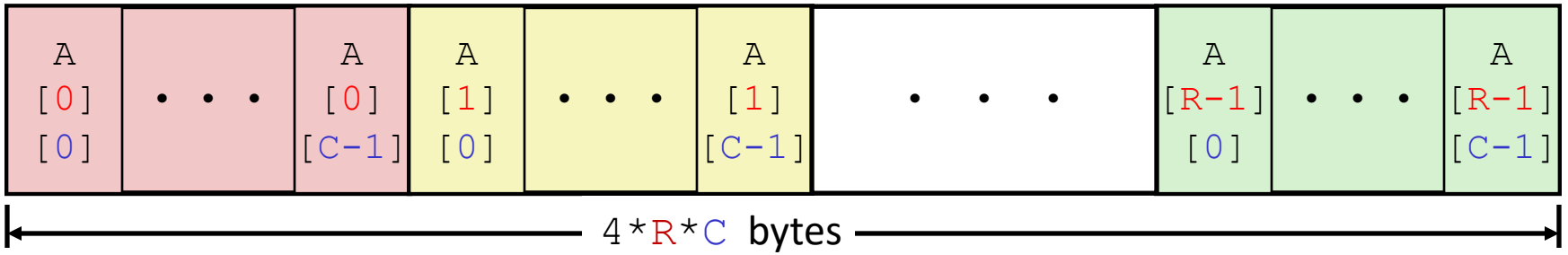
Two-Dimensional (Nested) Arrays

- 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***sizeof**(**T**) bytes
- Arrangement: **row-major** ordering

int A[R][C];

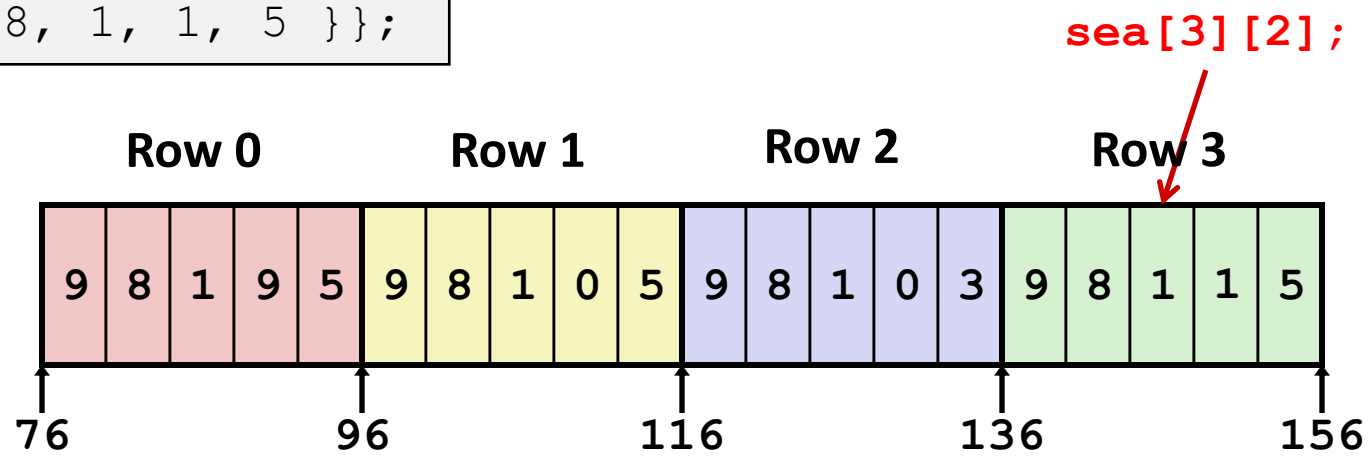


```
typedef int vec5[5];
```

Nested Array Example

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

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

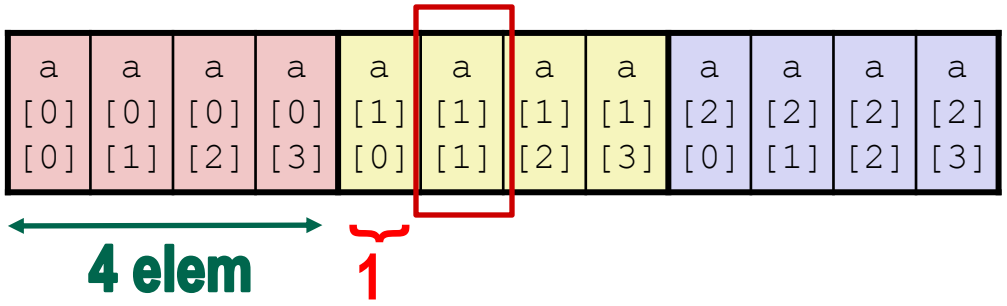


- "Row-major" ordering of all elements
- Elements in the same row are contiguous
- Guaranteed (in C)

Address calculation

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

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]



a[1][1] = Address of A[1][0] + 1

= 1*# of elem/1D array + 1

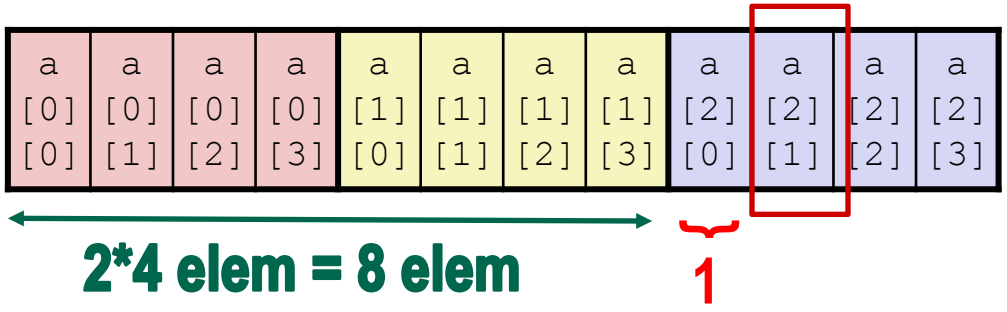
= 1 * 4 + 1

= 5 elems from start = 20 bytes (if int)

Address calculation

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

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]



a[2][1] = Address of A[2][0] + 1

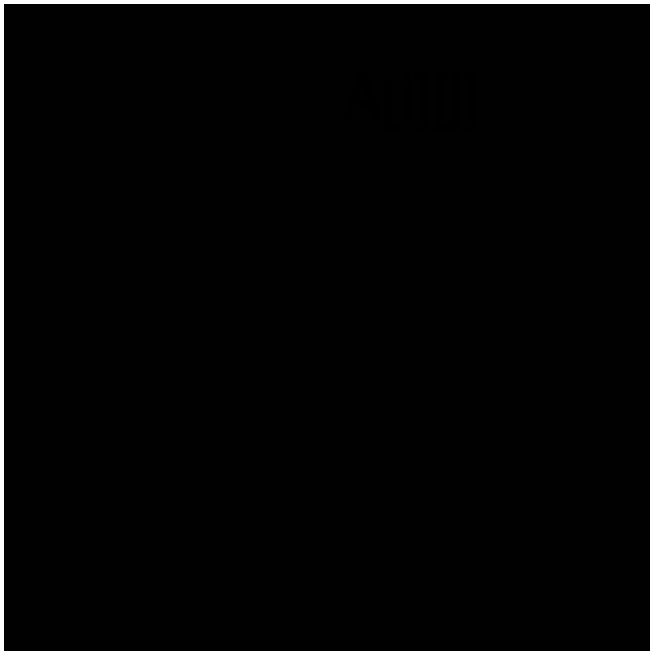
= 2*# of elem/1D array + 1

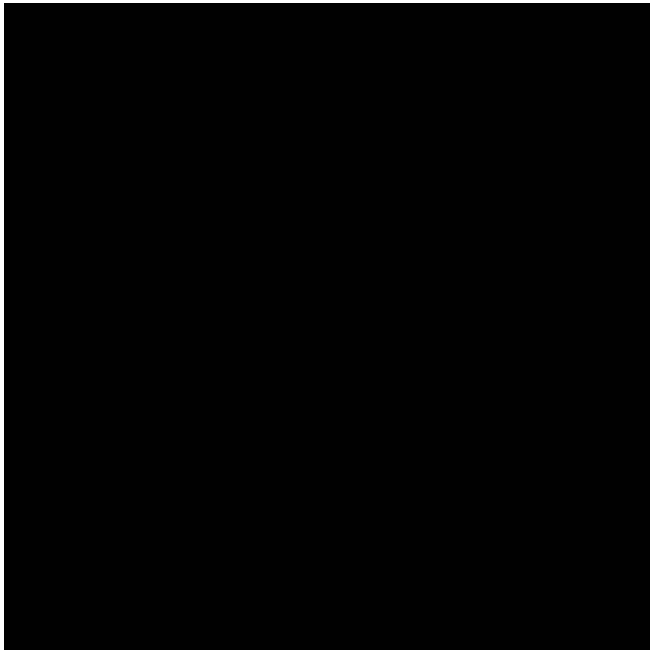
= 2 * 4 + 1

= 9 elems from start = 36 bytes (if int)

```
for (i = 0; i < 4){
   for(j = 0;  < 4) {
     A[i][j]
```

```
for (i = 0; i < 4){
   for(j = 0;  < 4) {
     A[j][i]
```

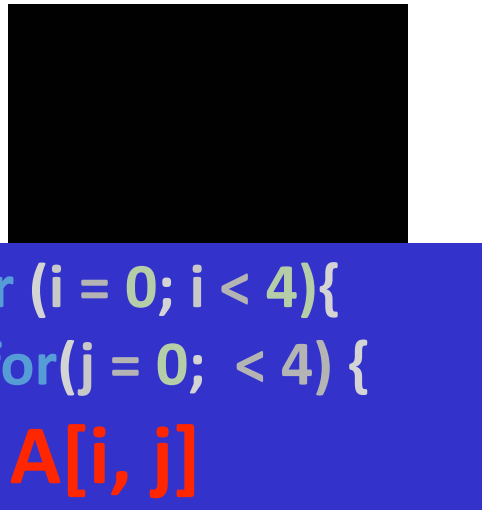




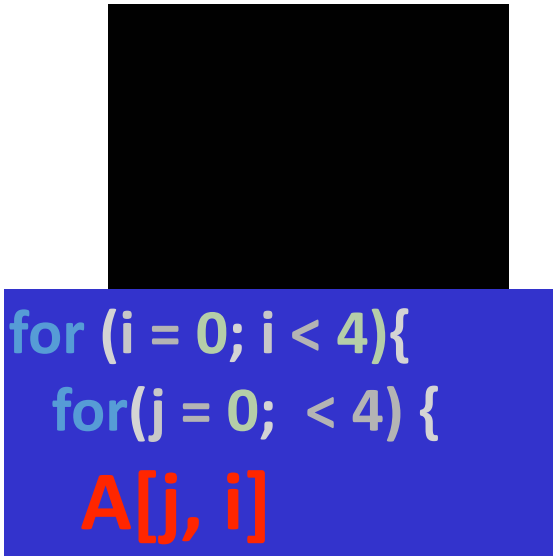
Stride

- Difference in address of two consecutive access to an array e.g., a[0] and a[1] (assume int a[]). Stride is 4.

a[0,0],a[0,1]……
Addr: 0,4,8,12
Stride = 4

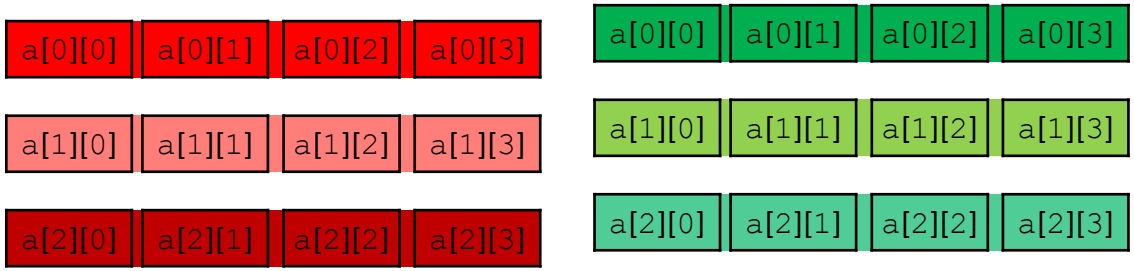


a[0,0],a[1,0]……
Addr: 0,16,32…
Stride = 16



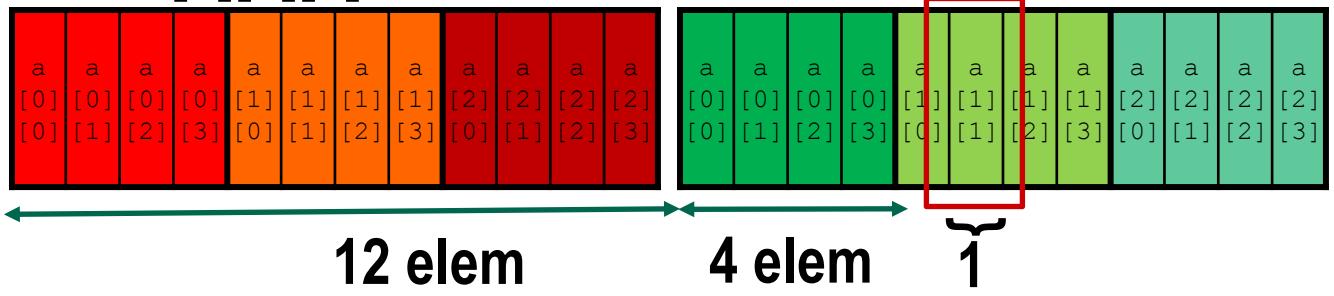
3D array

- 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 elem

a[1][1][1] = 1*12 + 1*4 + 1 = 17 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:

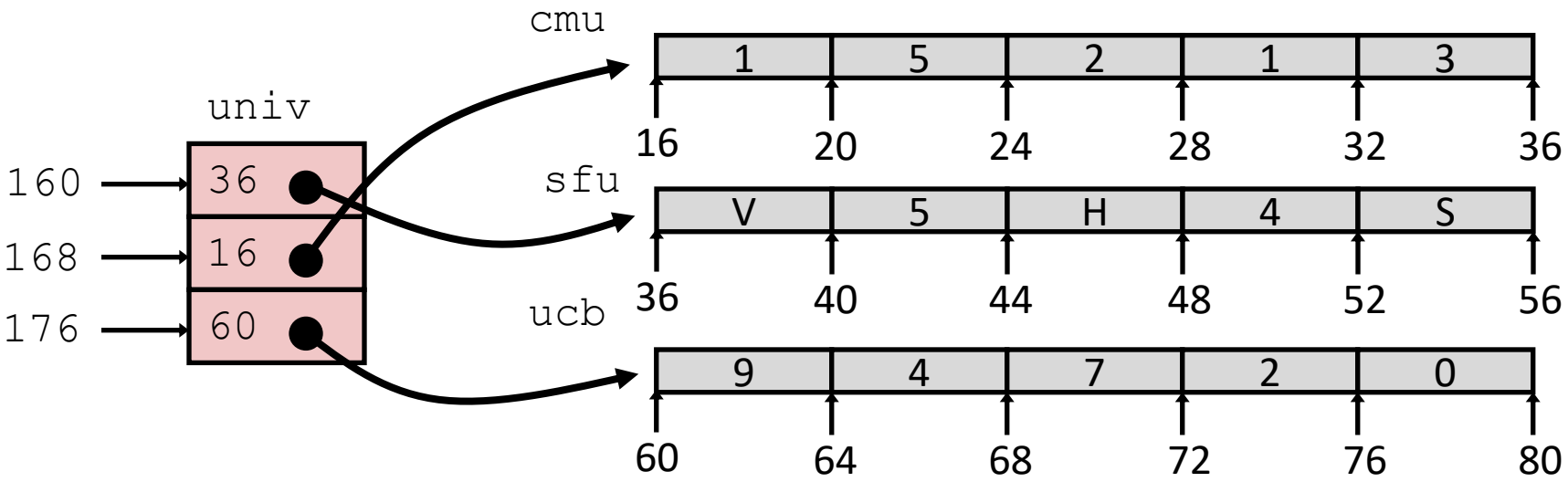
```
vec5 univ2D[3] = {
  { 9, 8, 1, 9, 5 },
  { 1, 5, 2, 1, 3 },
  { 9, 4, 7, 2, 0 }
};
```

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



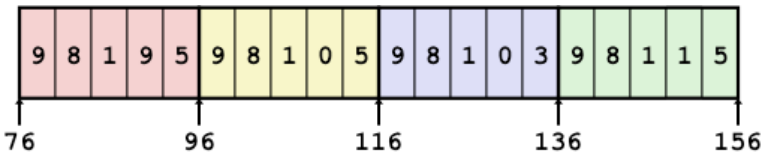
Reference	Address	Value	Guaranteed?
`univ[2][3]`			
`univ[1][5]`			
`univ[2][-2]`			
`univ[3][-1]`			
`univ[1][12]`			

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

Array Element Accesses

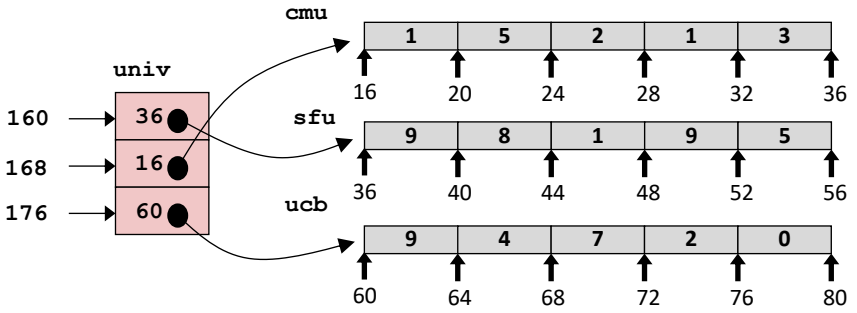
Nested array

```
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:

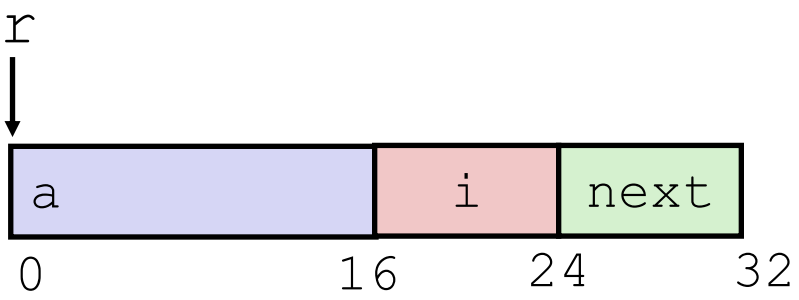
Mem[sea+20*index+4*digit]

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

C Structure Representation

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

struct rec *r;
```

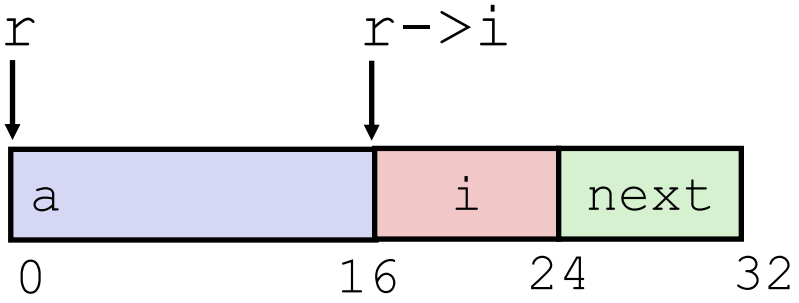


- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures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

Accessing a Structure Member

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

struct rec *r;
```



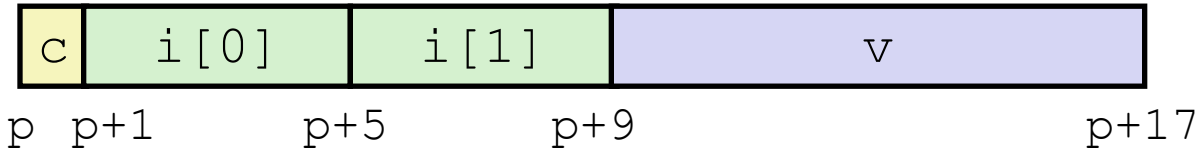

```
long get_i(struct rec *r)
{
  return r->i;
}
```

- 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
```

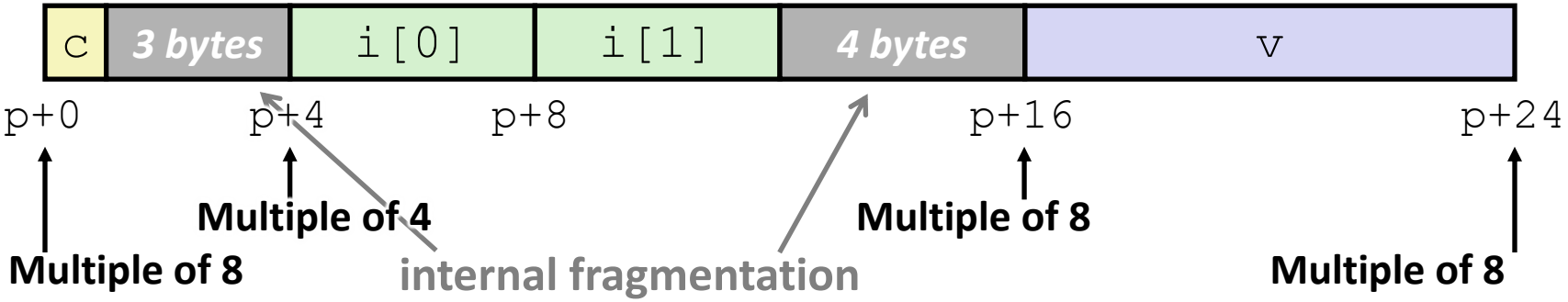
Structures & Alignment

- Unaligned Data




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

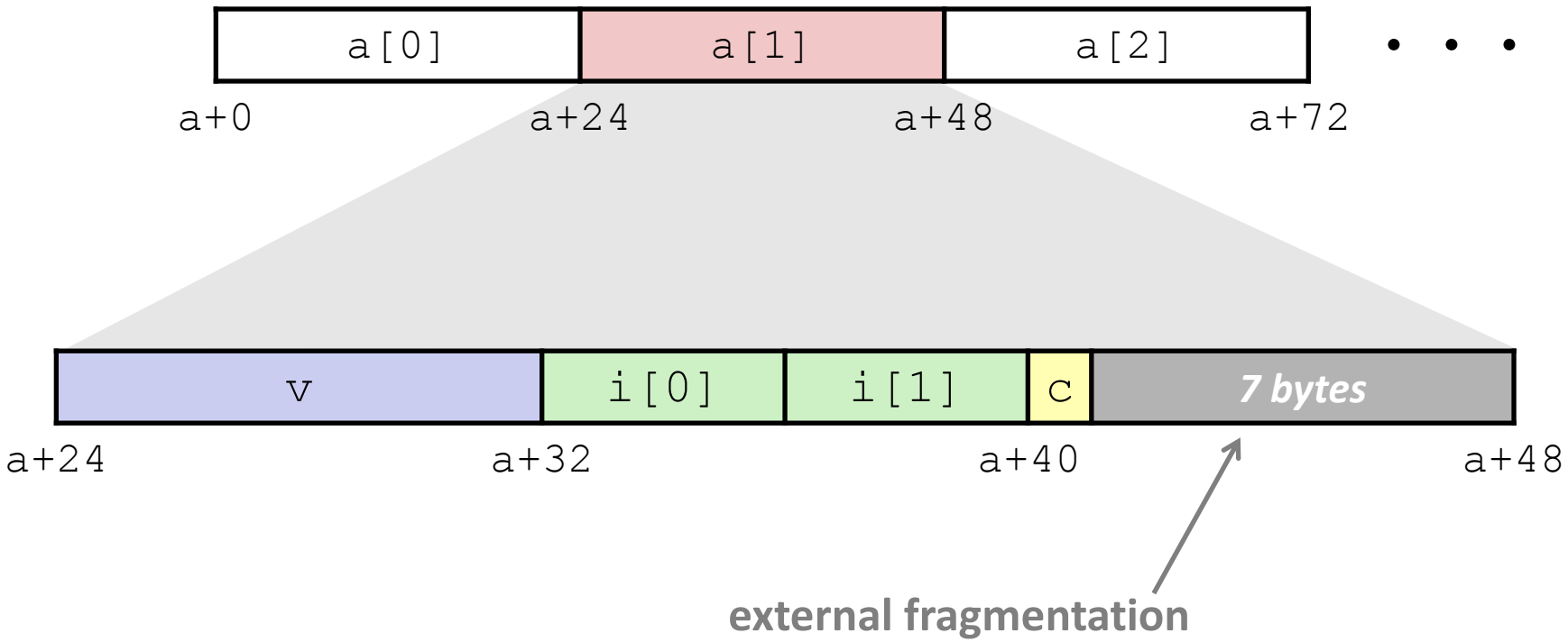
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



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];
```

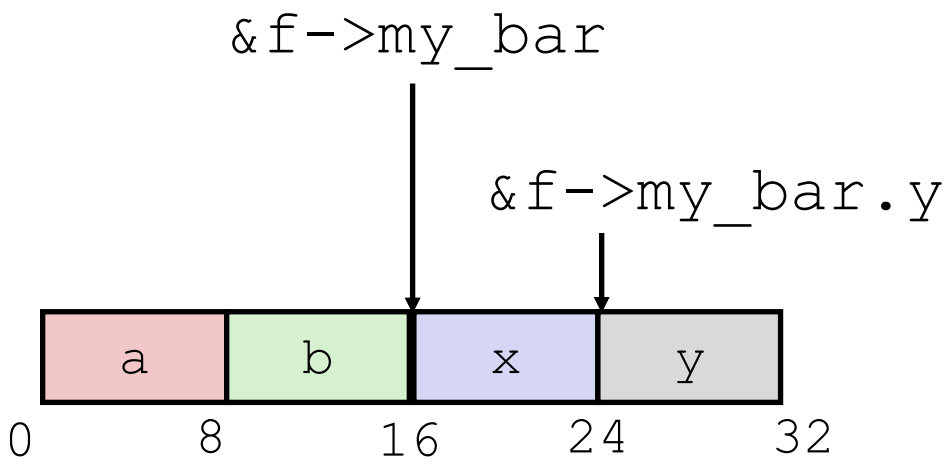


Nested Struct

```
struct foo {
    long a;
    long b;
    struct bar my_bar;
};

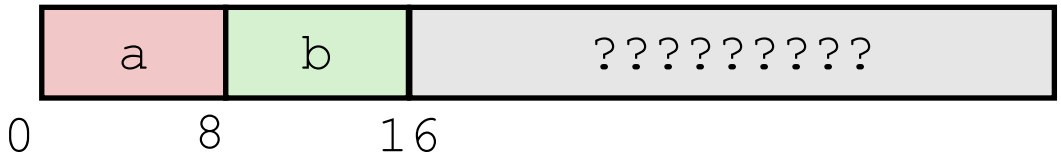
struct bar {
    long x;
    long y;
};

struct foo *f;
```

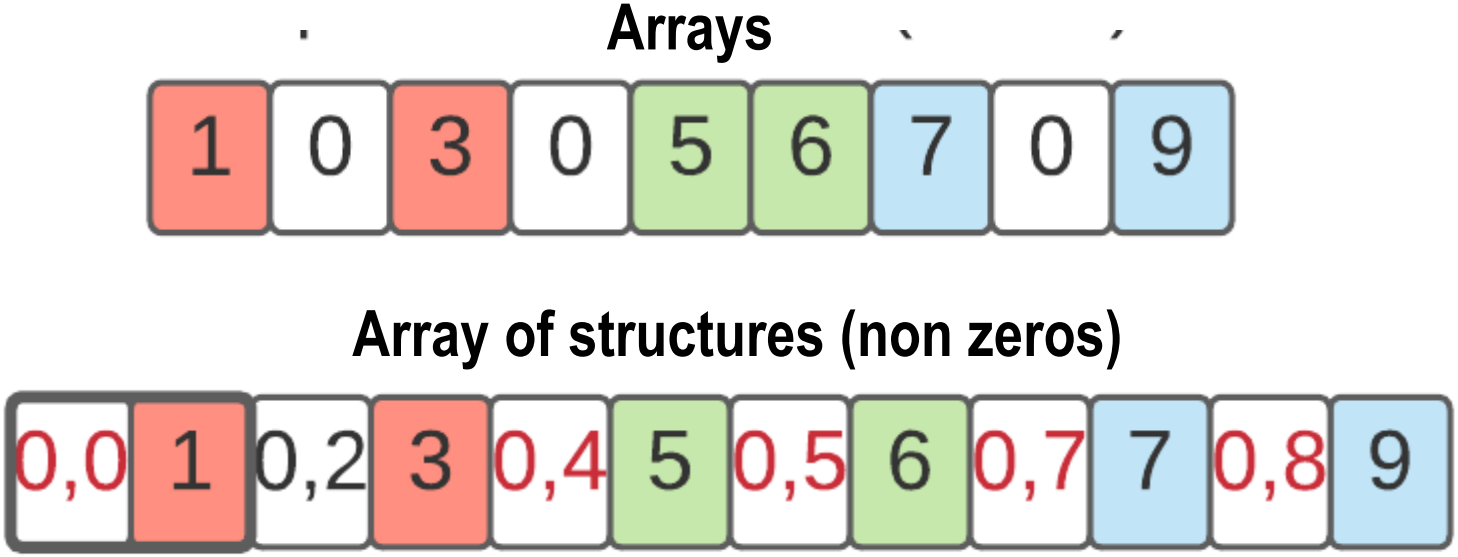


Nested Struct

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



Sparse Arrays (Array of Structs)




```
struct coo { int row; int col; int val; } // each non-zero
int nnz                                   // number of non zeros
struct coo_array[nnz]                     // Array of non zeros
```

```
for i = 0 to nnz
   int* base = &coo[i]
   int r = *base
   int c = *(base+1)
   int val = *(base+2)
```

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)
- 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)

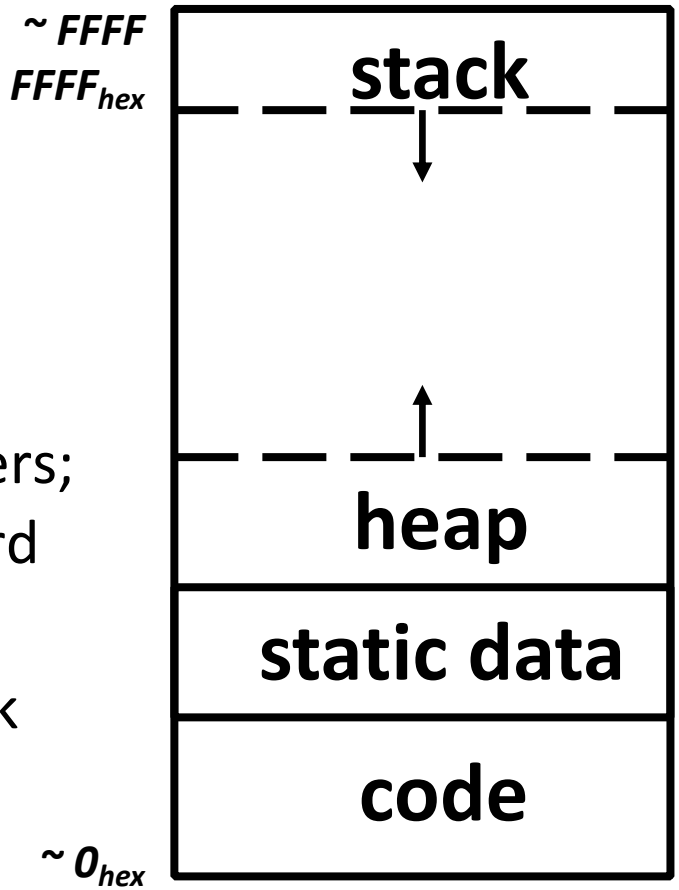
```
int array[1024];

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

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

C Memory Layout

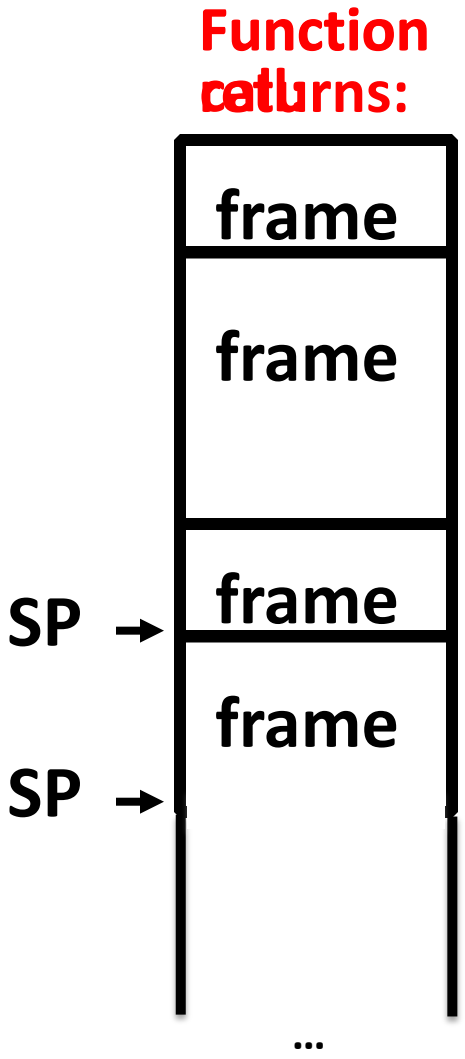
- Program's *address space* contains 4 regions:
  - Stack: local variables, grows downward
  - Heap: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
  - Static Data: global and static variables, does not grow or shrink
  - Code: loaded when program starts, does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

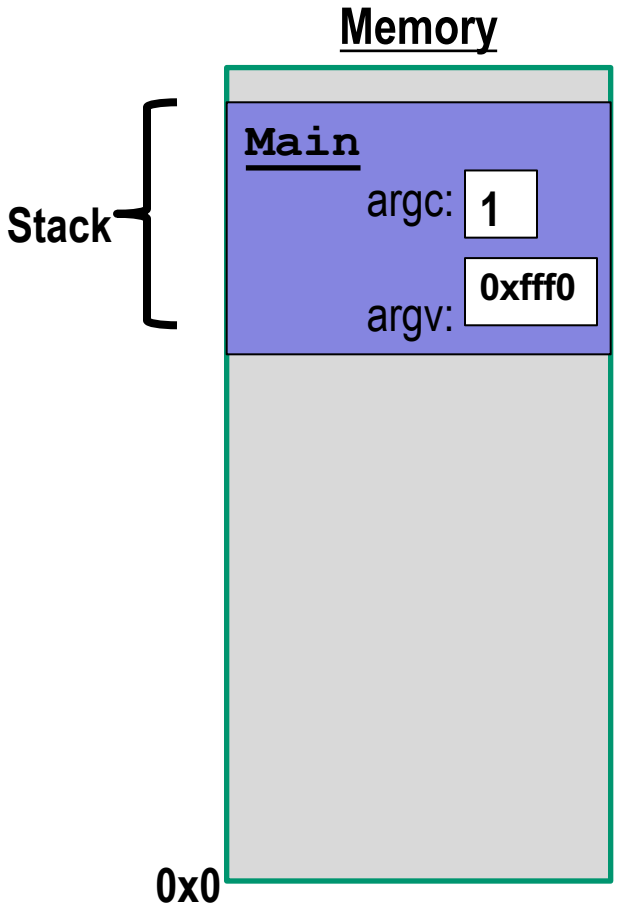


The Stack

```
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;
}
```

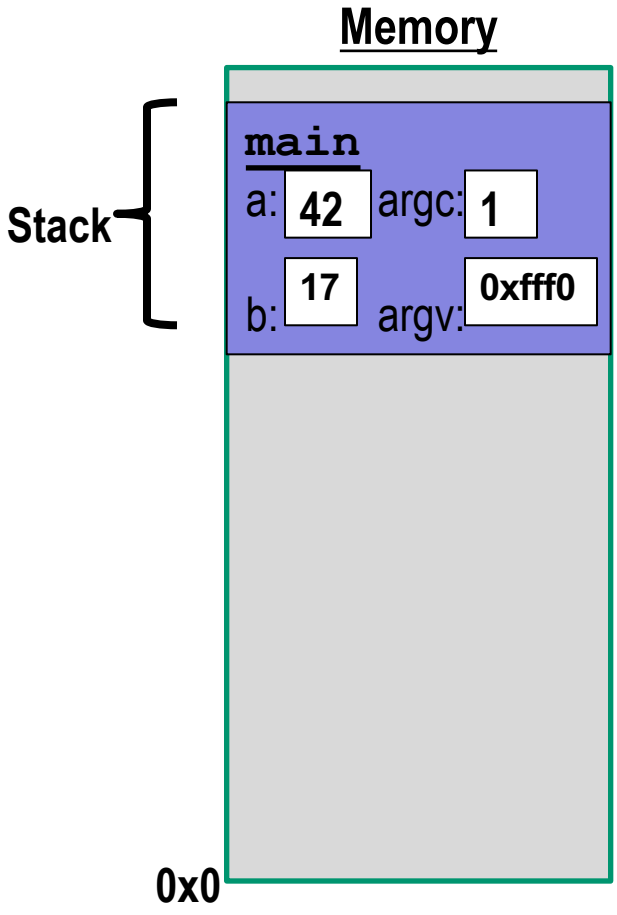


The Stack

```
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;
}
```

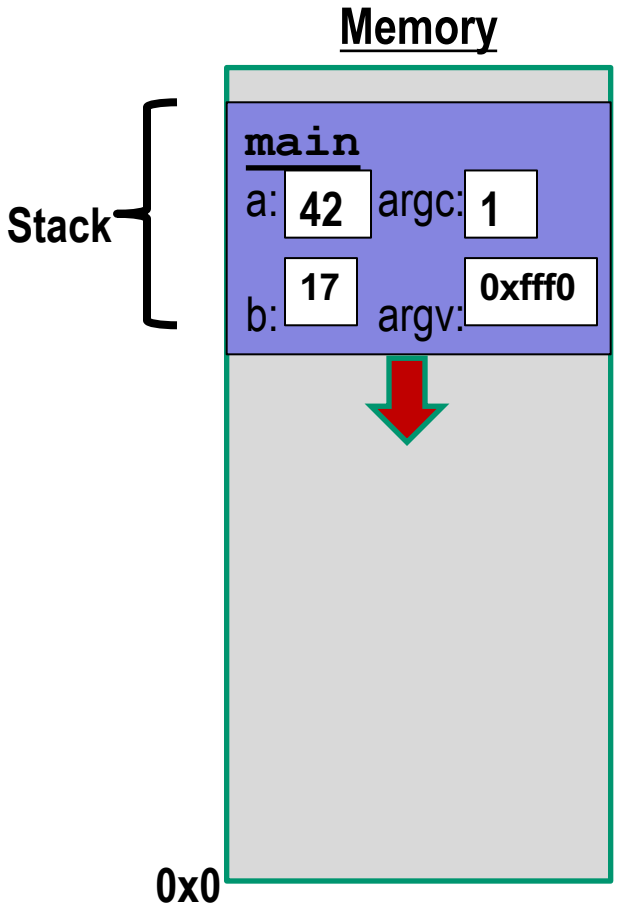


The Stack

```
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;
}
```

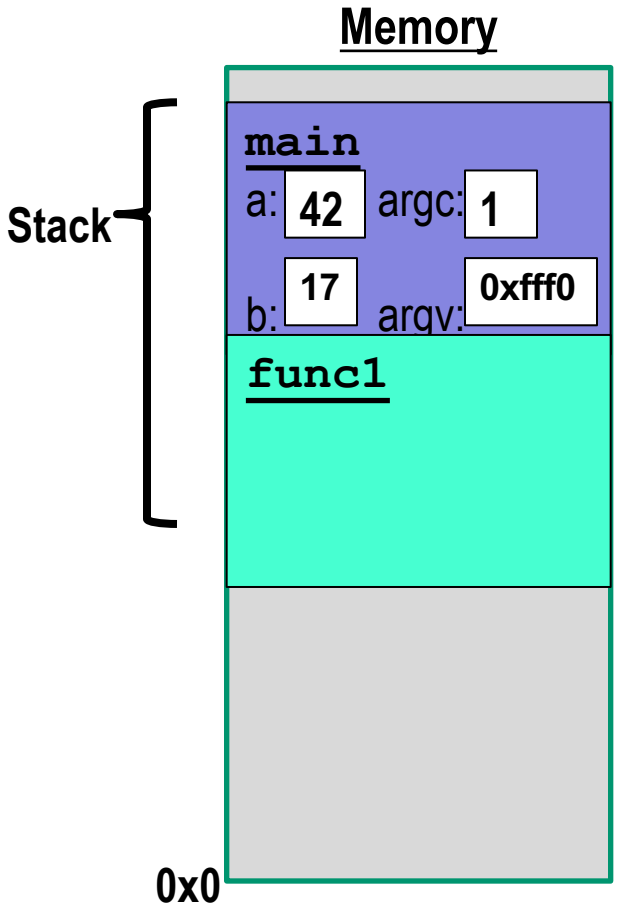


The Stack

```
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;
}
```

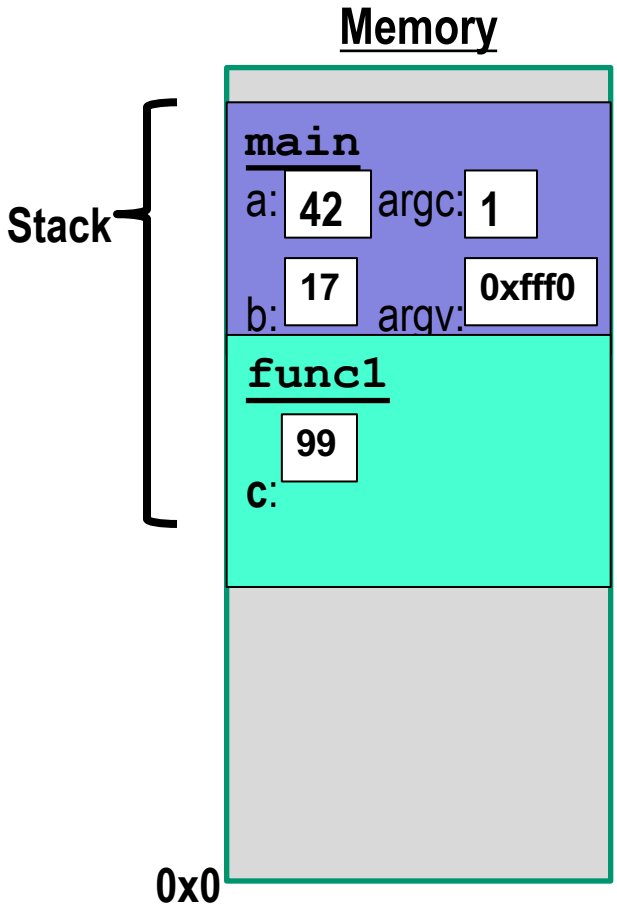


The Stack

```
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;
}
```

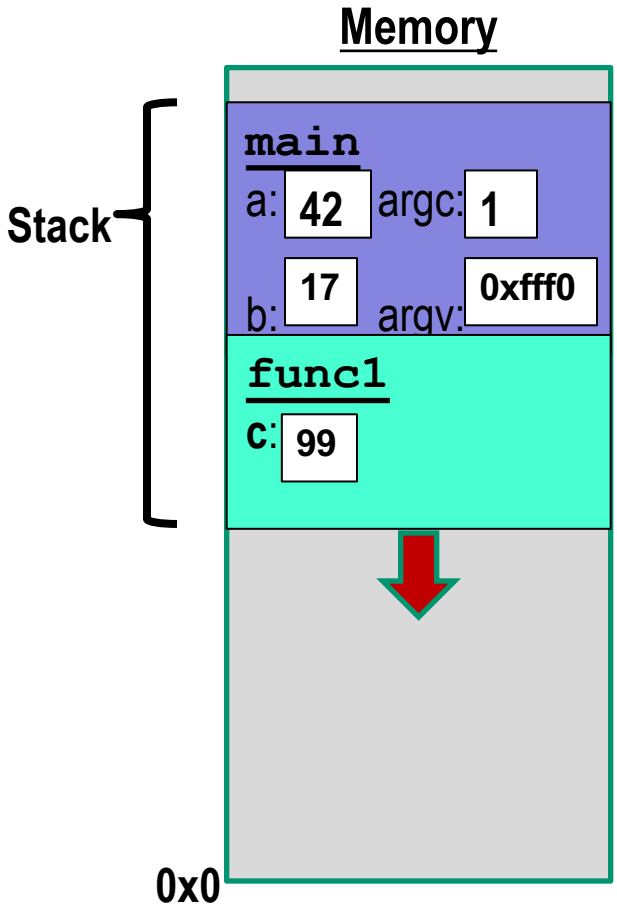


The Stack

```
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;
}
```

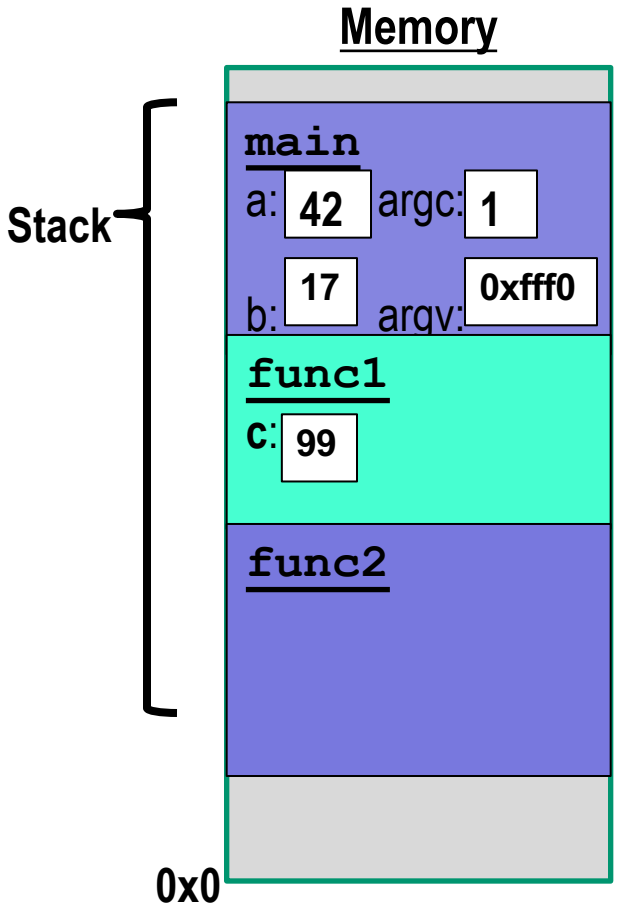


The Stack

```
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;
}
```

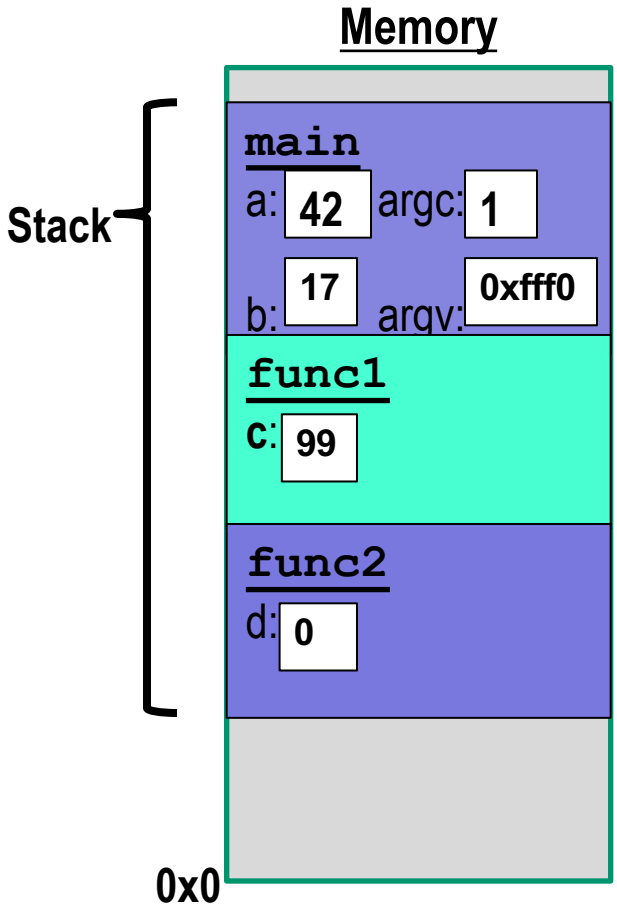


The Stack

```
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;
}
```

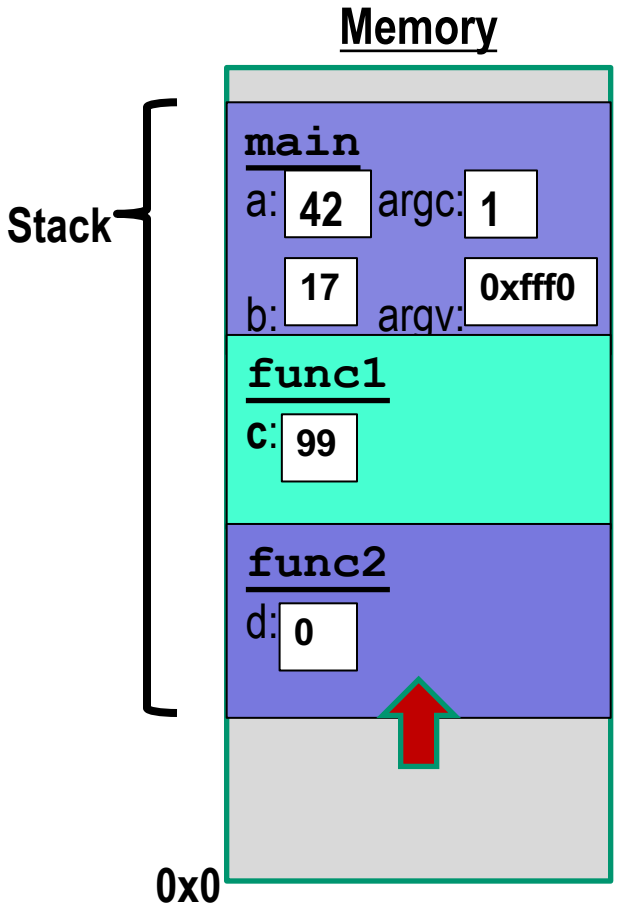


The Stack

```
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;
}
```

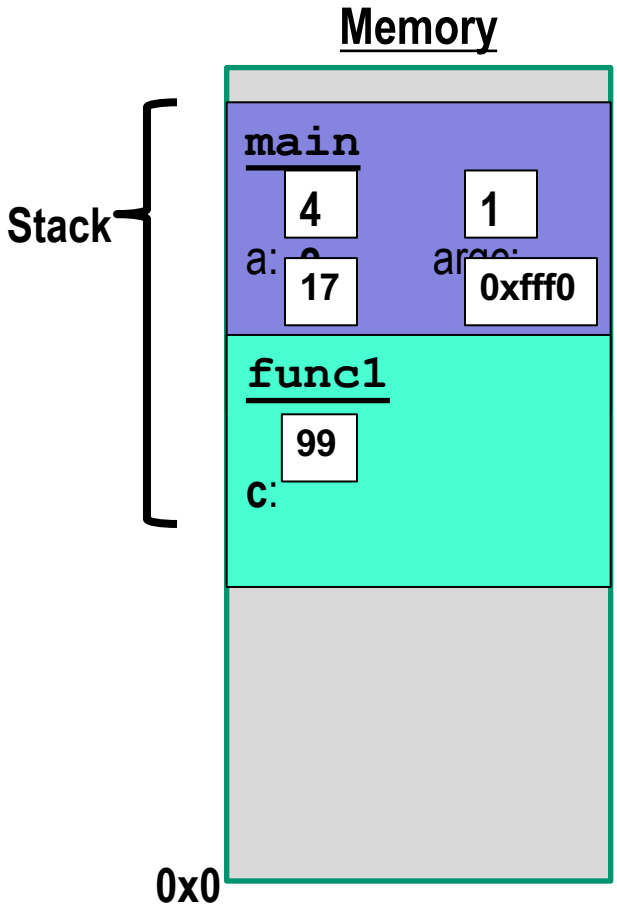


The Stack

```
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;
}
```

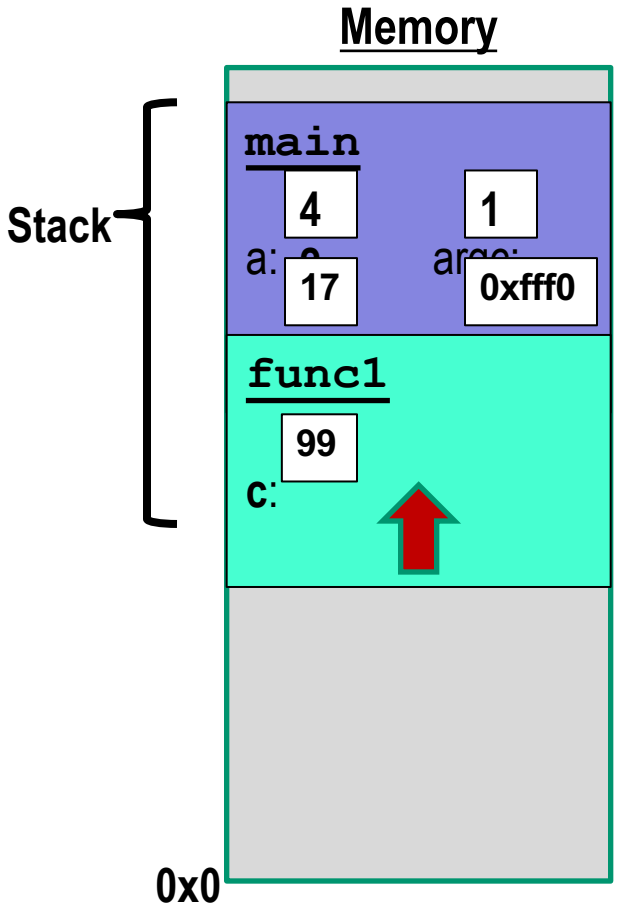


The Stack

```
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;
}
```

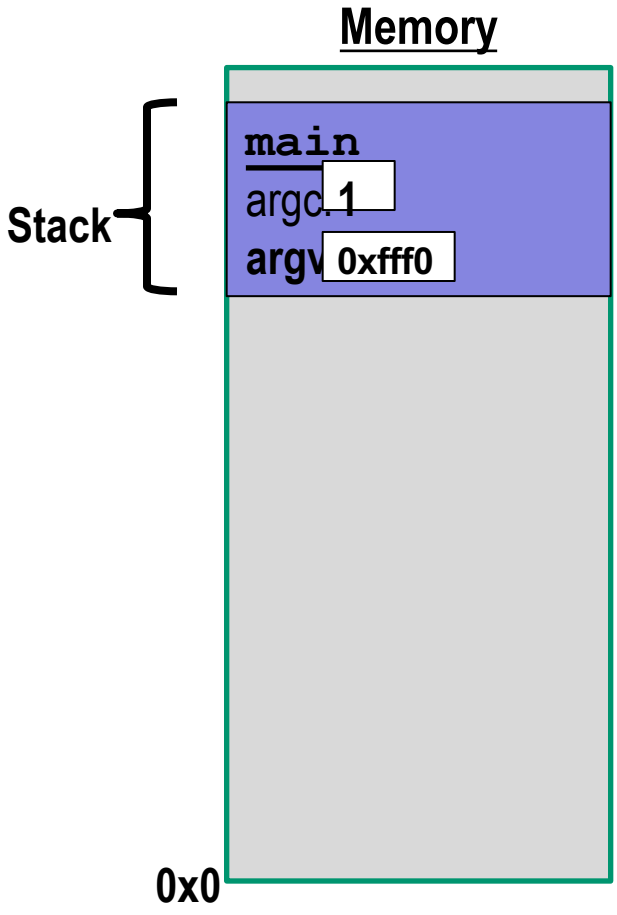


The Stack

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;
}
```

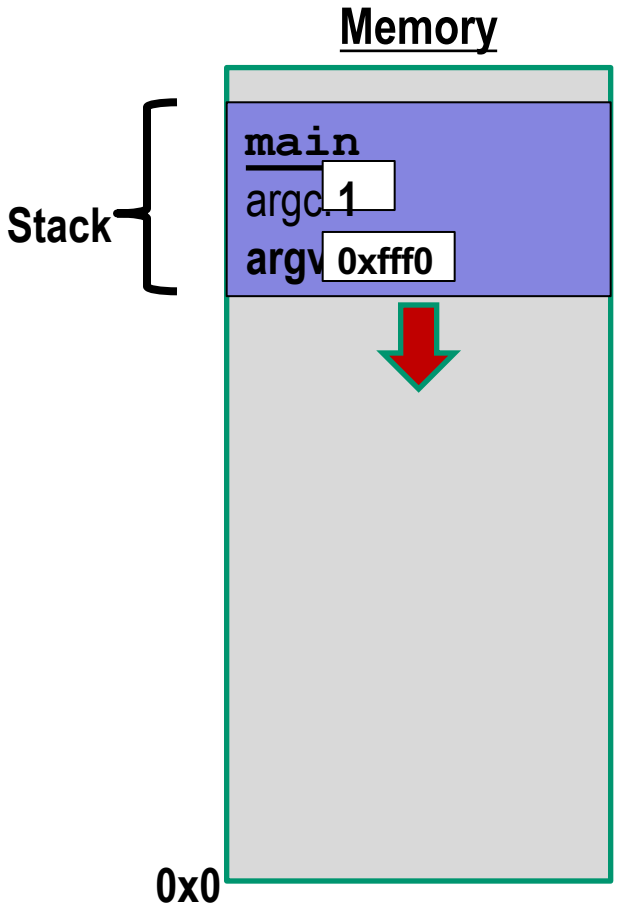


The Stack

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;
}
```

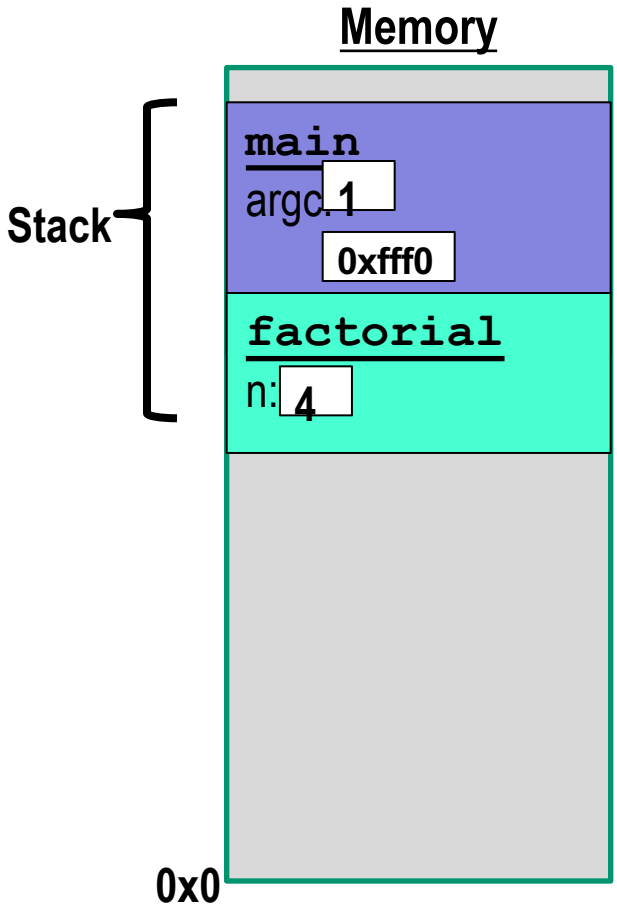


The Stack

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;
}
```

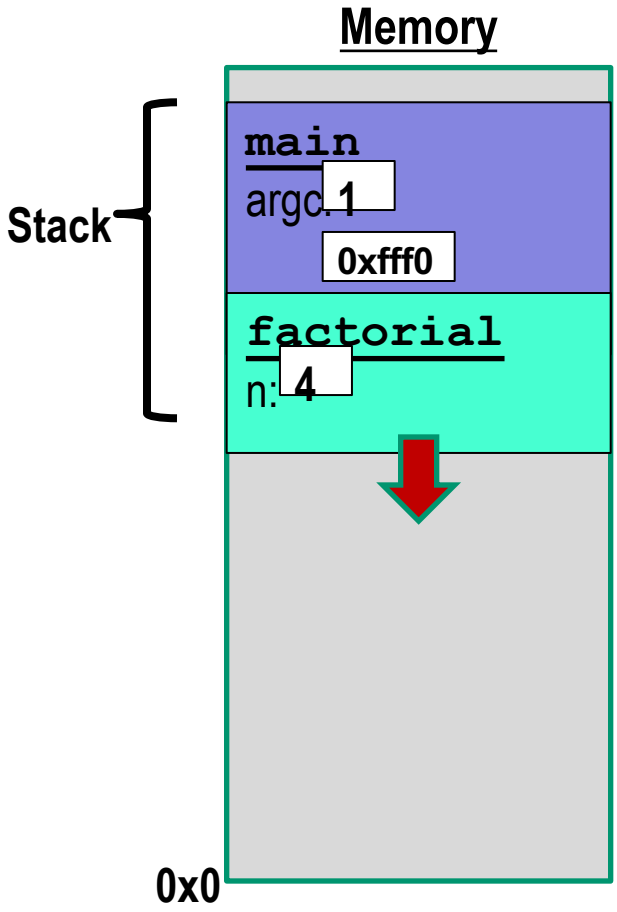


The Stack

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;
}
```

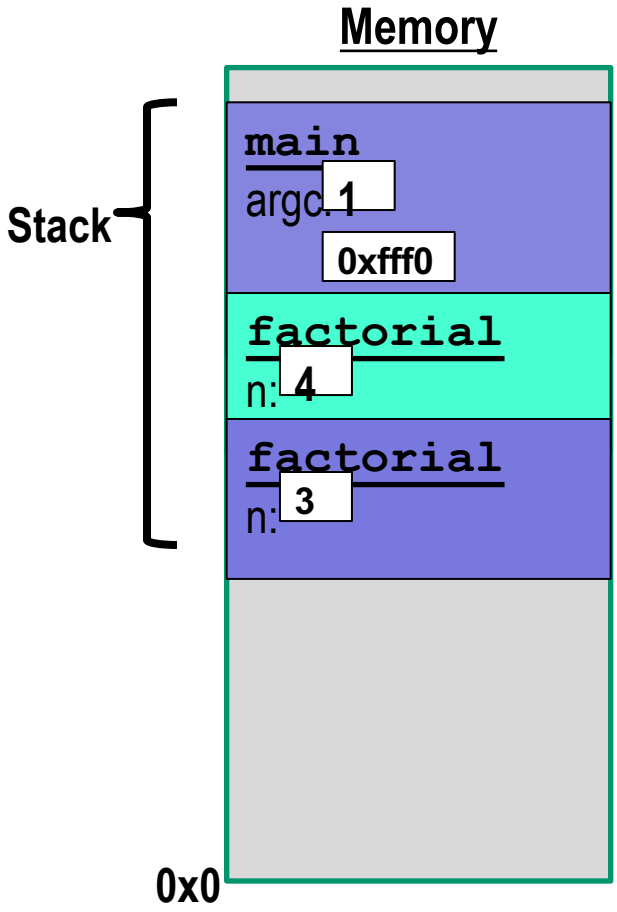


The Stack

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;
}
```

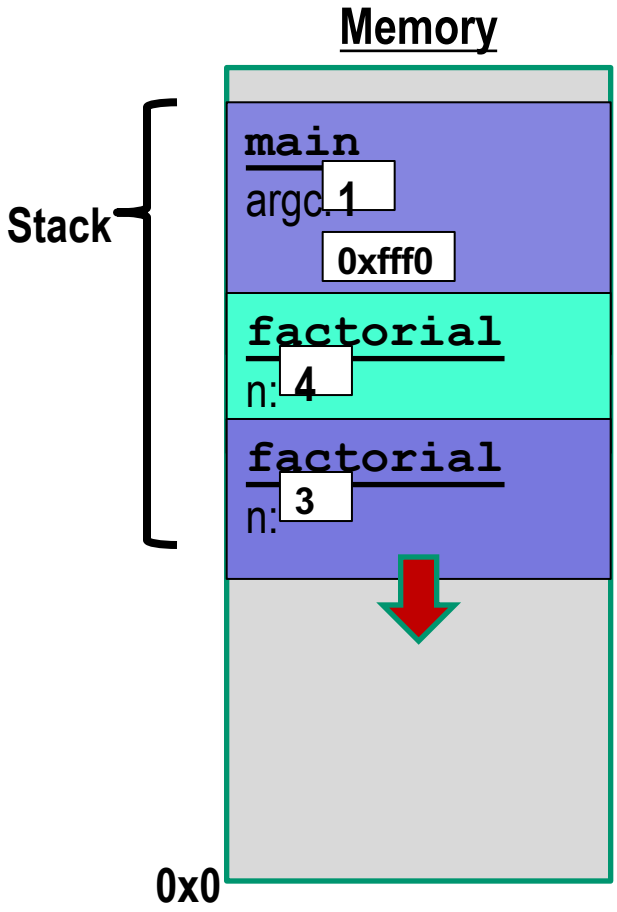


The Stack

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;
}
```

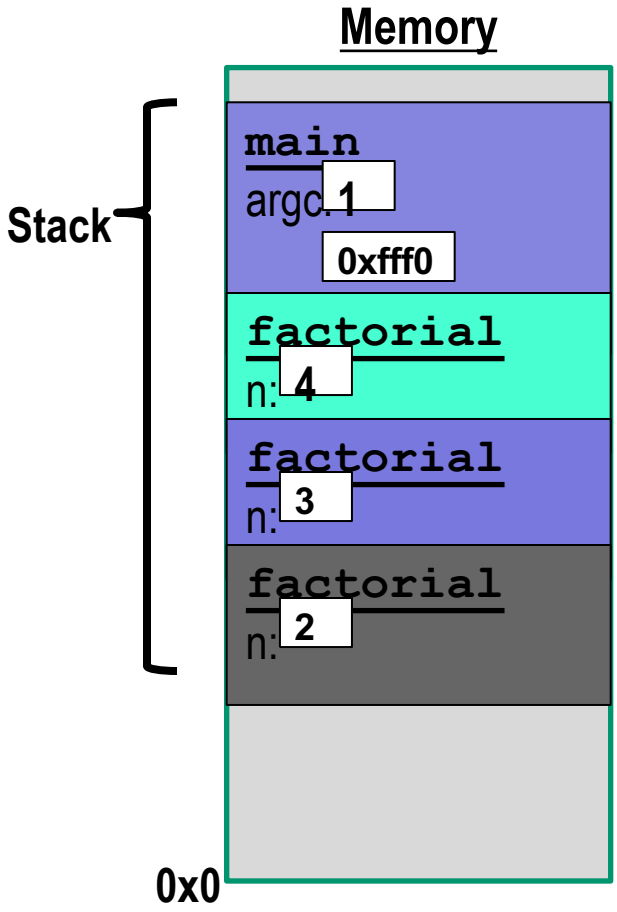


The Stack

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;
}
```

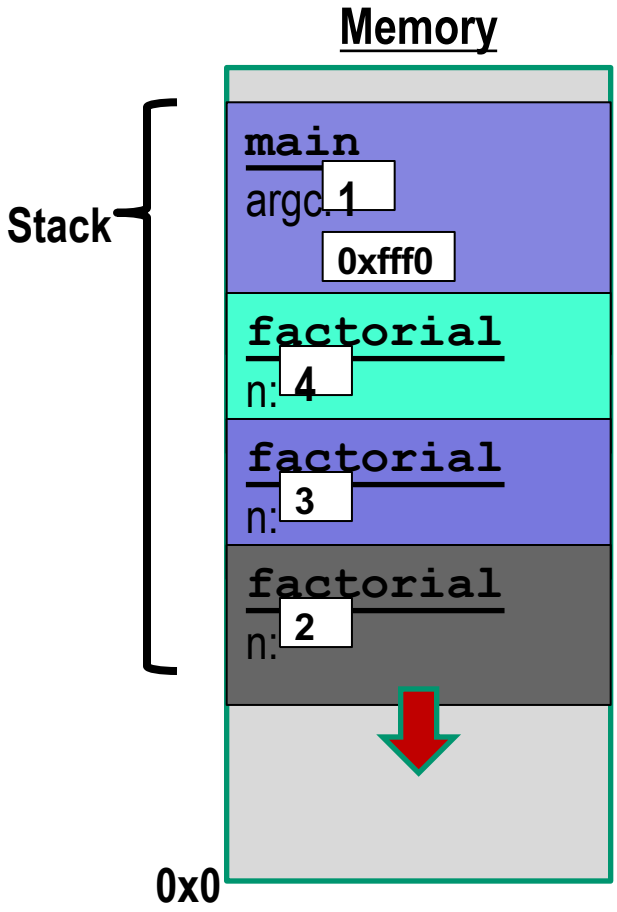


The Stack

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;
}
```

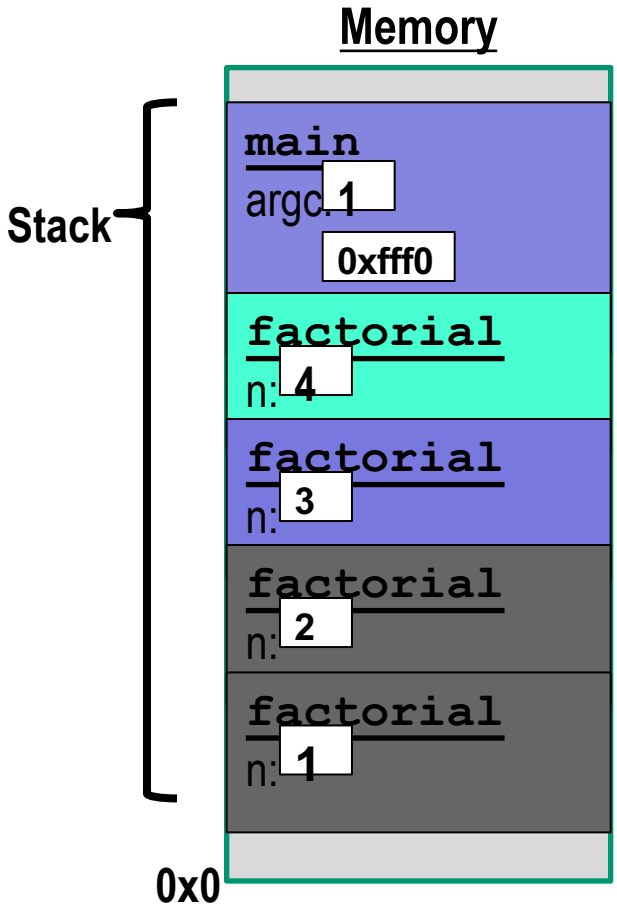


The Stack

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;
}
```

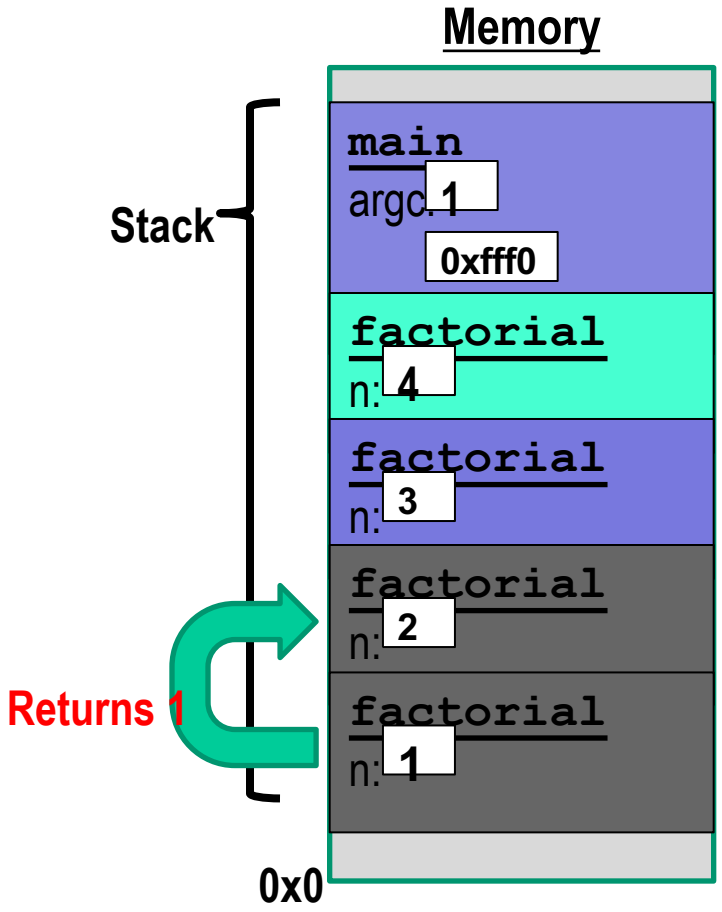


The Stack

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;
}
```

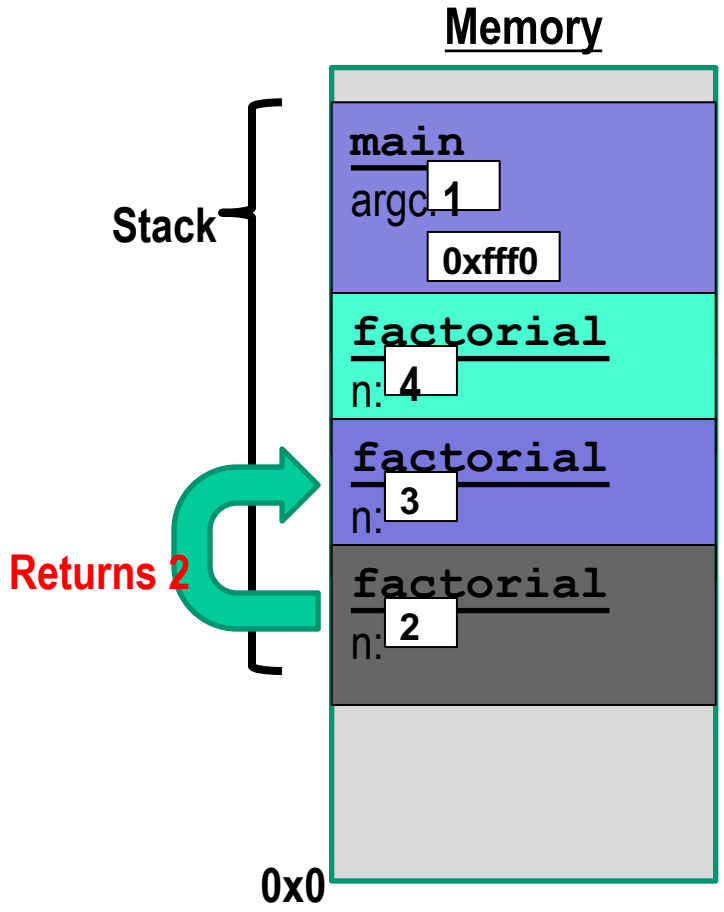


The Stack

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;
}
```

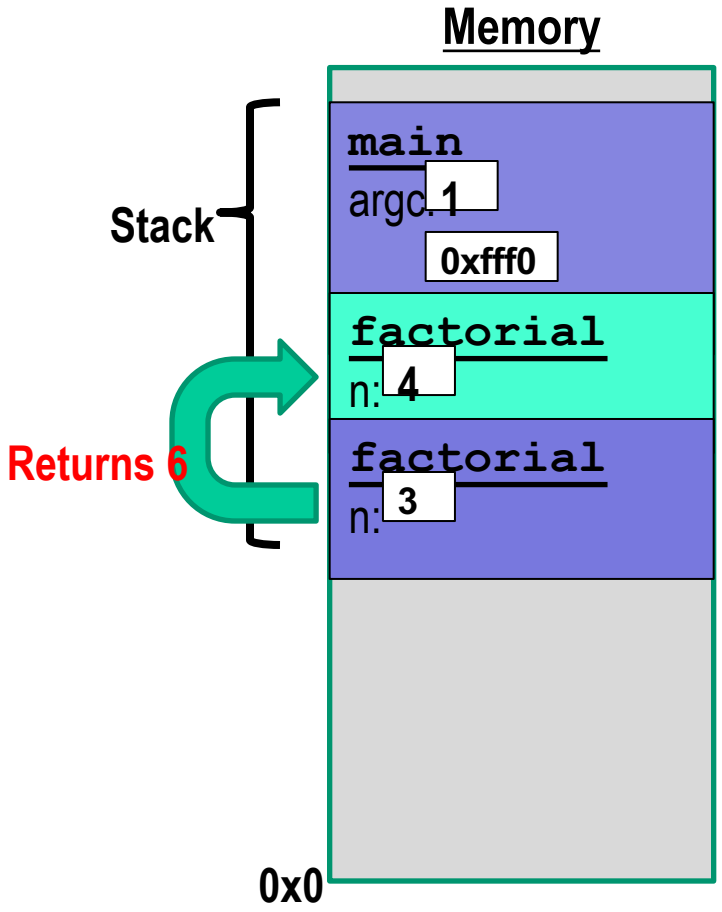


The Stack

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;
}
```

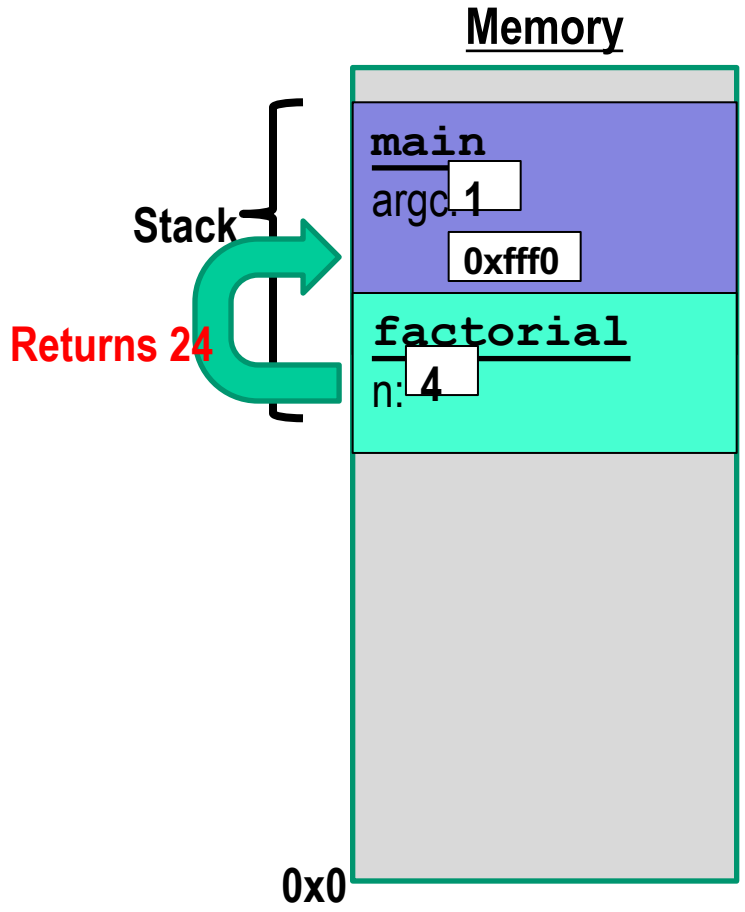


The Stack

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;
}
```

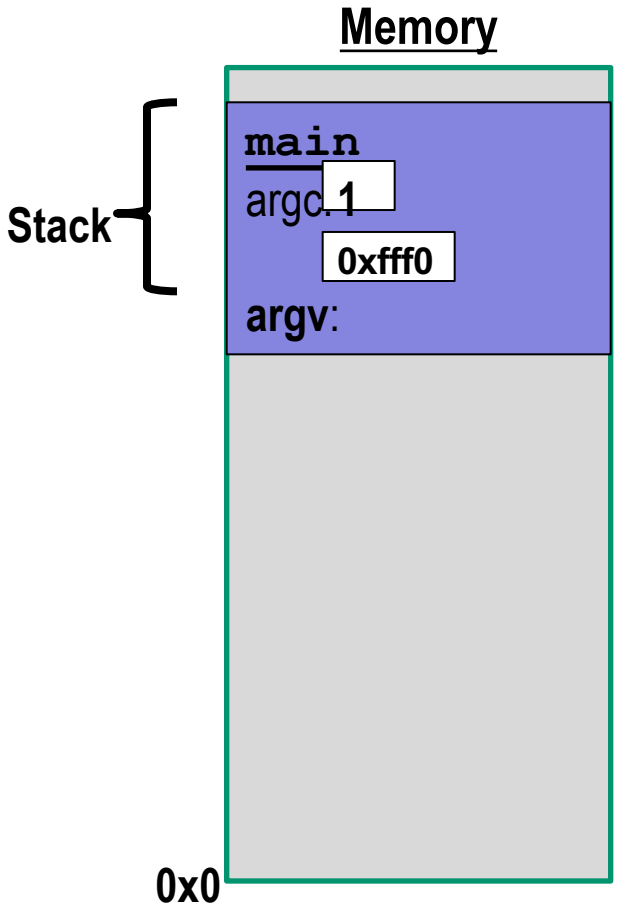


The Stack

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;
}
```

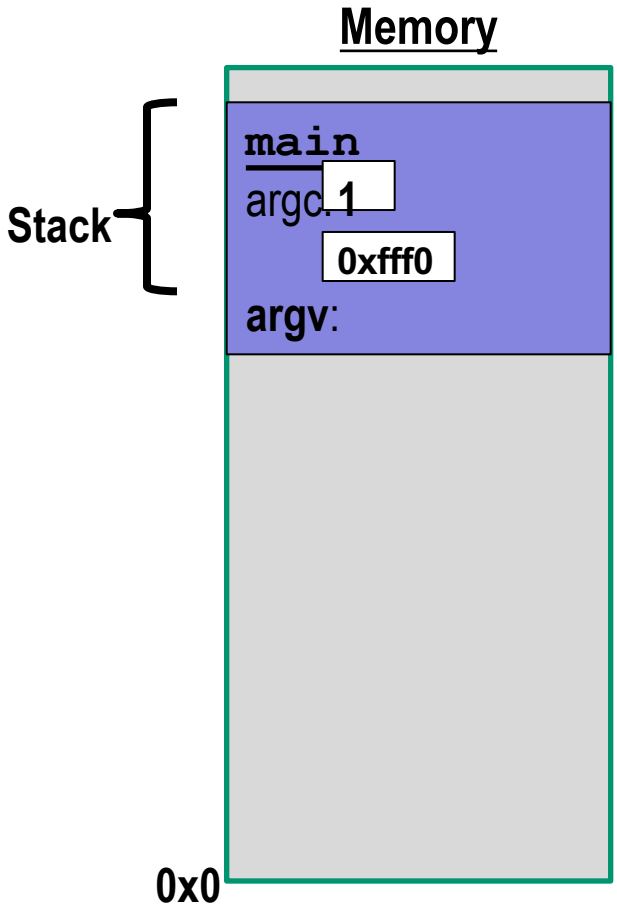


The Stack

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

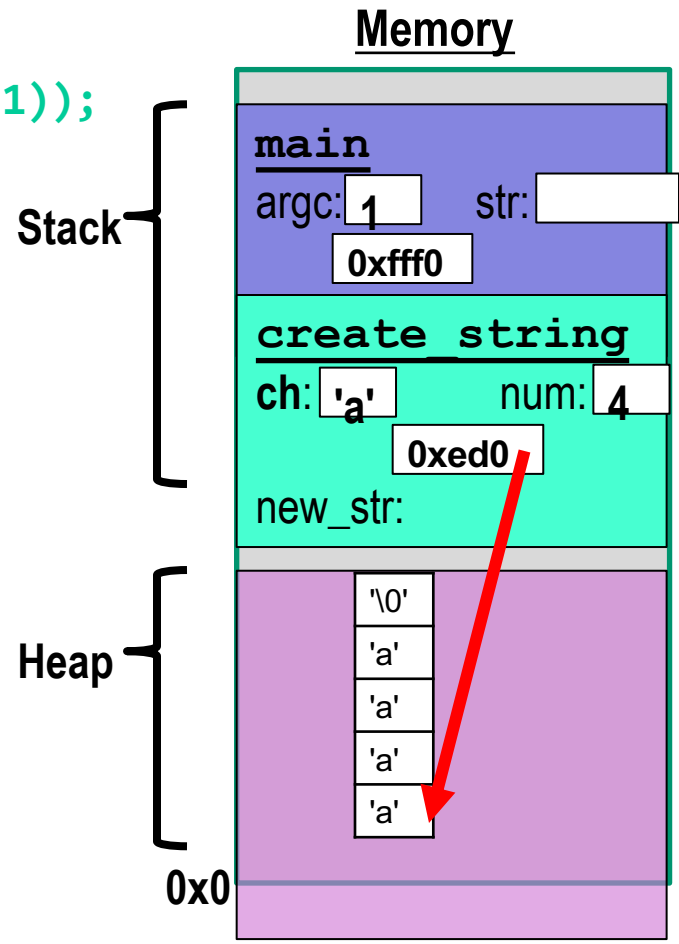
- The **heap** is a part of memory that you can manage yourself.
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.

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;
}

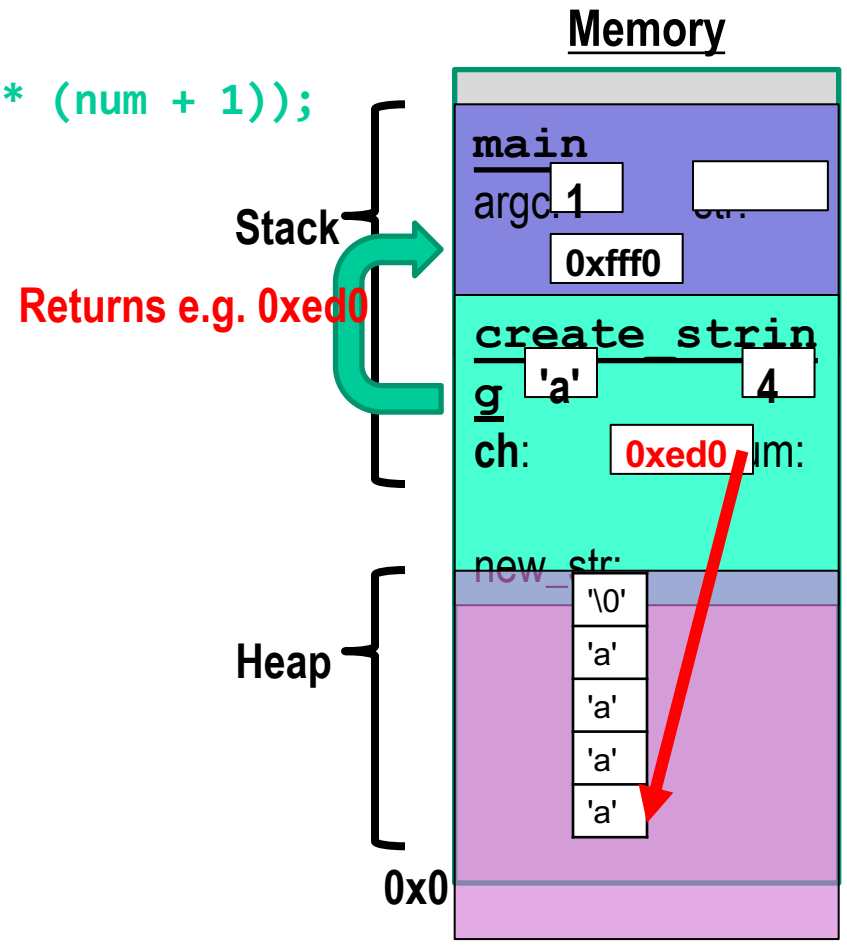
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



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;
}

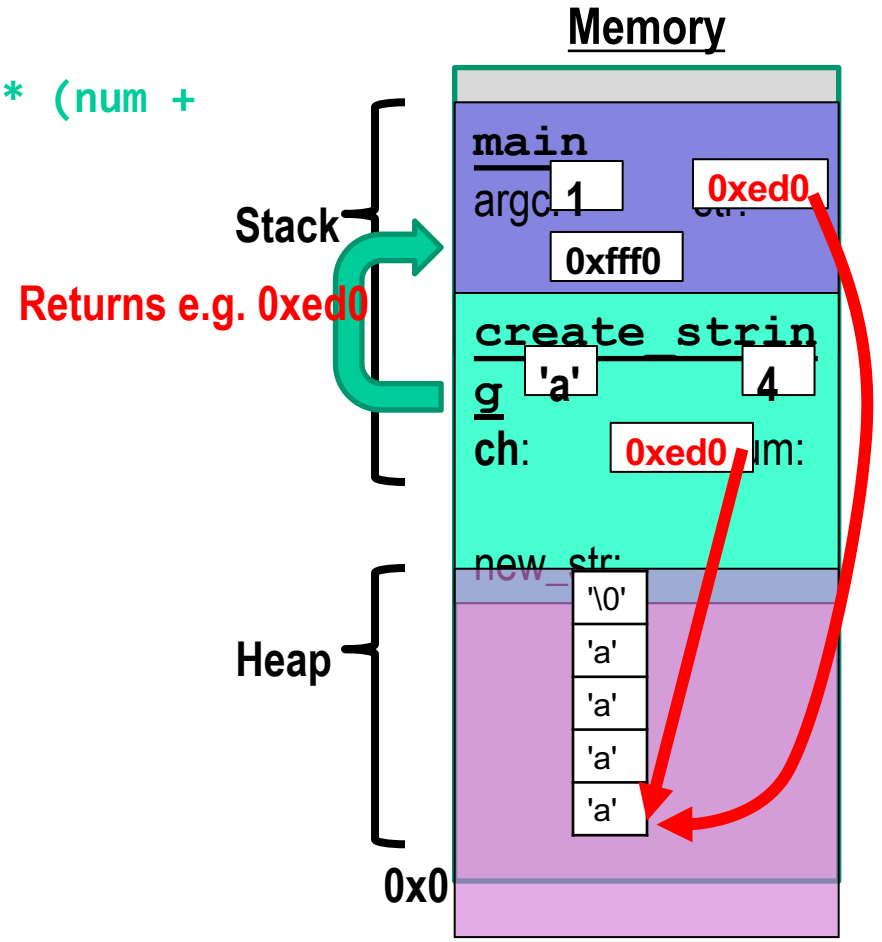
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



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;
}

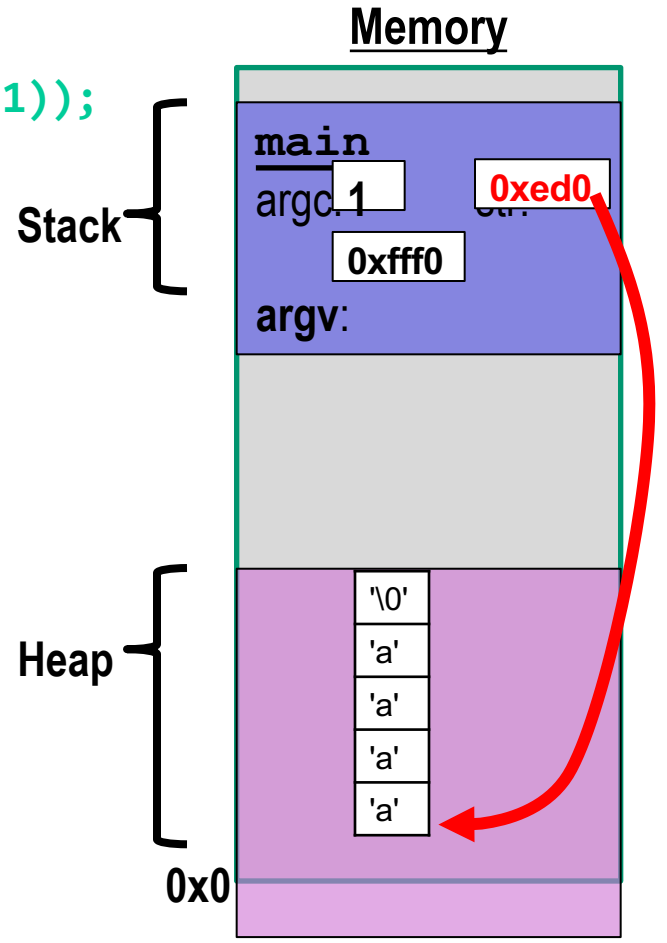
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



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;
}

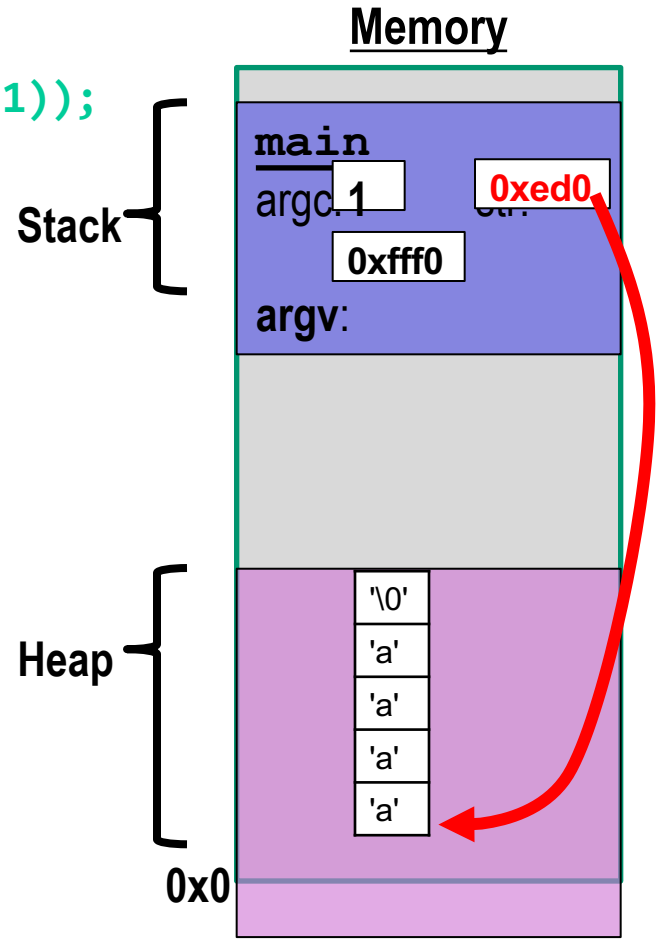
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



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;
}

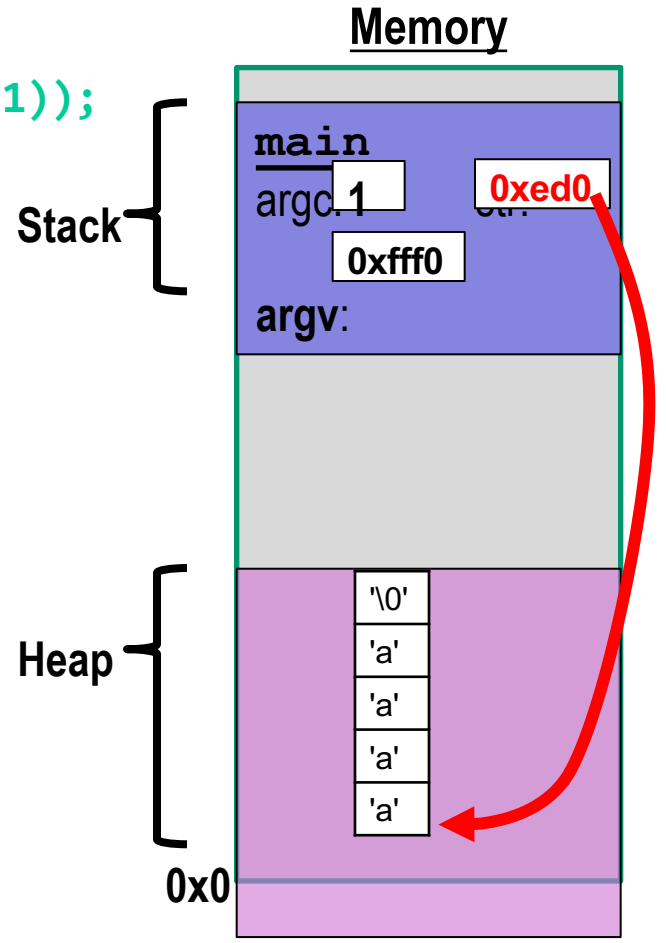
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



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), `^` (XOR), `~` (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	~ 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 OR b
0	0	0
0	1	1
1	0	1
1	1	1

XOR

a	b	a ^ 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<<n`)
  - Fill with `0`s on right
- Right shift (`x>>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≥w` (bit width of x) are *undefined*
  - **In C:** behavior of `>>` is determined by compiler
    - depends on data type of `x` (signed/unsigned)
  - **In Java:** logical shift is `>>>` and arithmetic shift is `>>`

	x	0010 0010
	x<<3	0001 0000
logical:	x>>2	0000 1000
arithmetic:	x>>2	0000 1000

	x	1010 0010
	x<<3	0001 0000
logical:	x>>2	0010 1000
arithmetic:	x>>2	1110 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<<2) = 1101 & (0b0011 << 2) = 1101 & 1100 = 1100
  - 1010 & (3<<2) = 1010 & (0b0011 << 2) = 1010 & 1100 = 1000
- Step 2: Prepare for concat:
  - 1100 >> 2 = 0011 (A), 1000 >>2 = 0010 (B)
- Step 3: Concat: A.B. Place B in 0th position, Place A in 2nd bit position = 0011 << 2 | 0010 => 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);
```

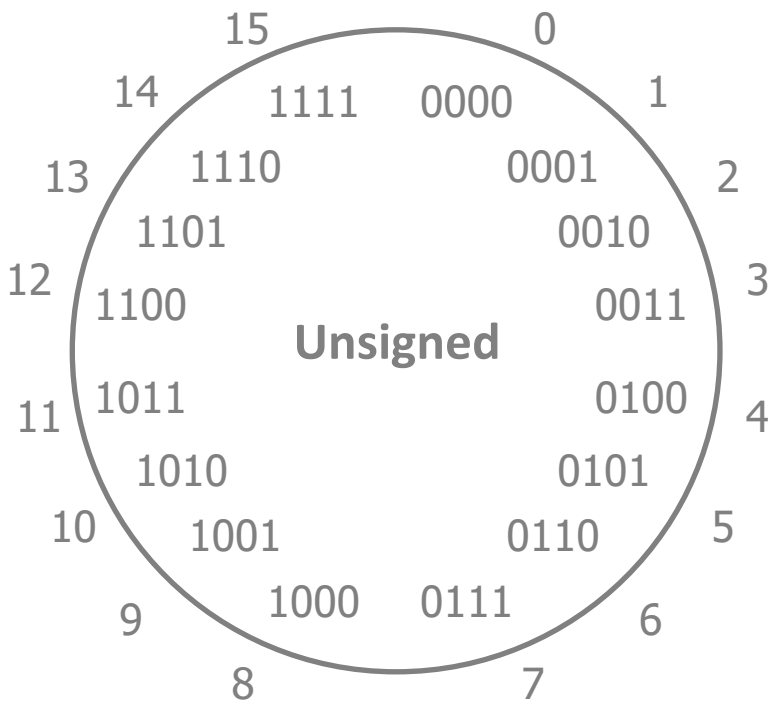
https://replit.com/@ashriram/Extract-and-Print#main.c

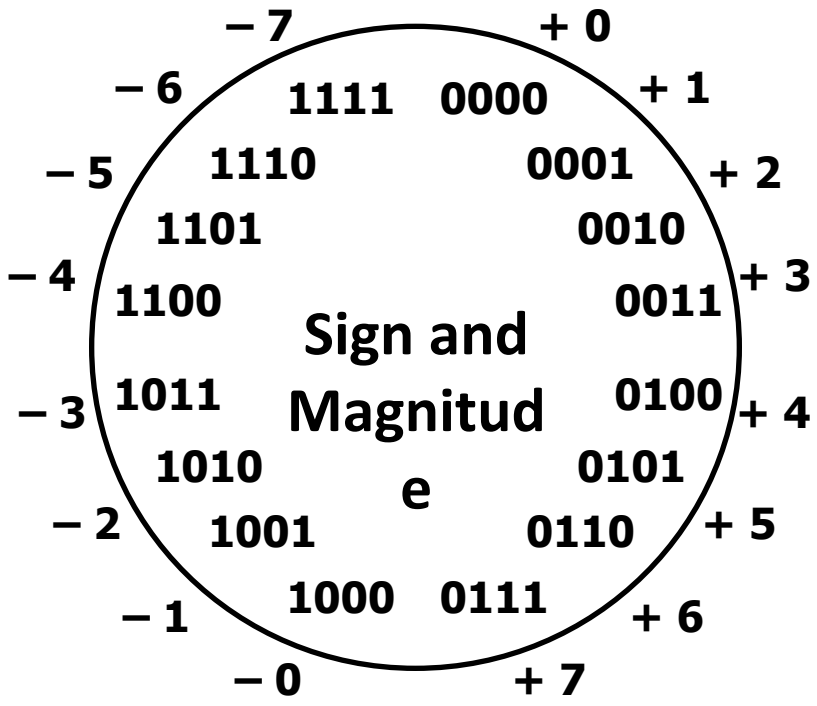
Integers

- 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

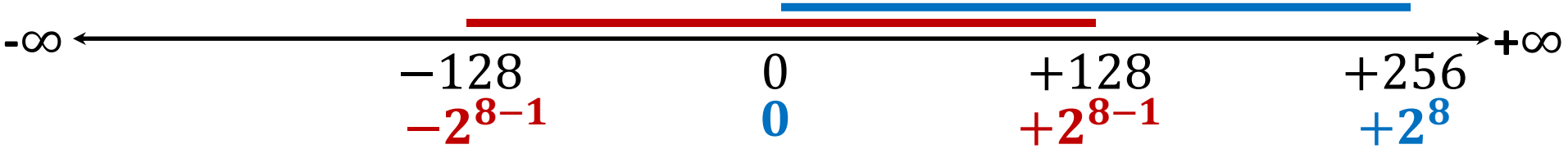
- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?





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: $0 \ldots 2^w - 1$
  - Signed values: $-2^{(w-1)} \ldots 0 \ldots 2^{(w-1)} - 1$
- **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 != 4+(-3)`

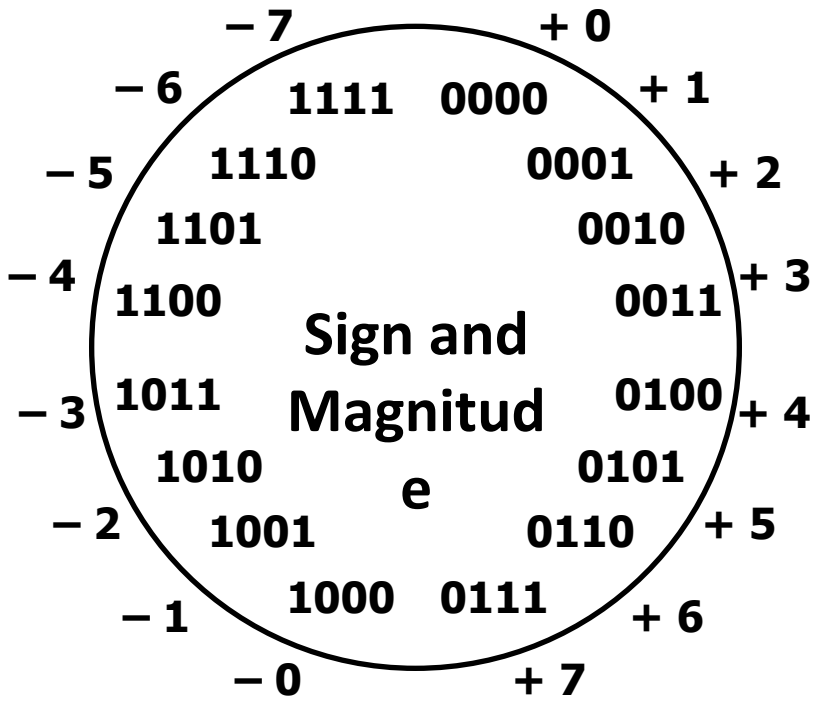
4	0100
- 3	- 0011
1	0001

✓

4	0100
+ -3	+ 1011
-7	1111

✗

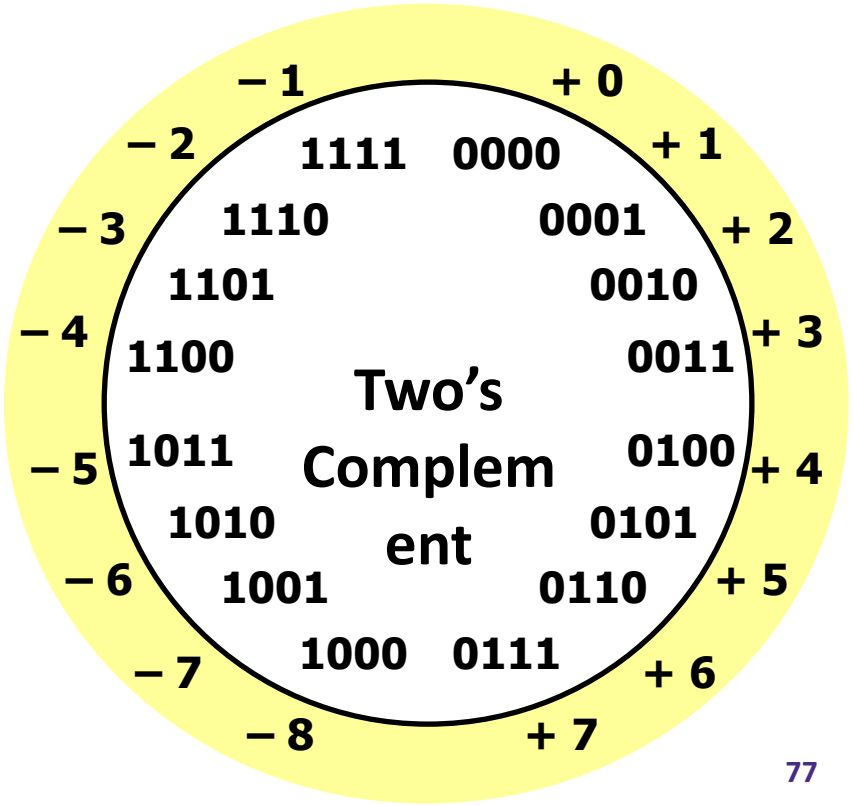
- Negatives "increment" in wrong direction!



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 )
```



3 bit

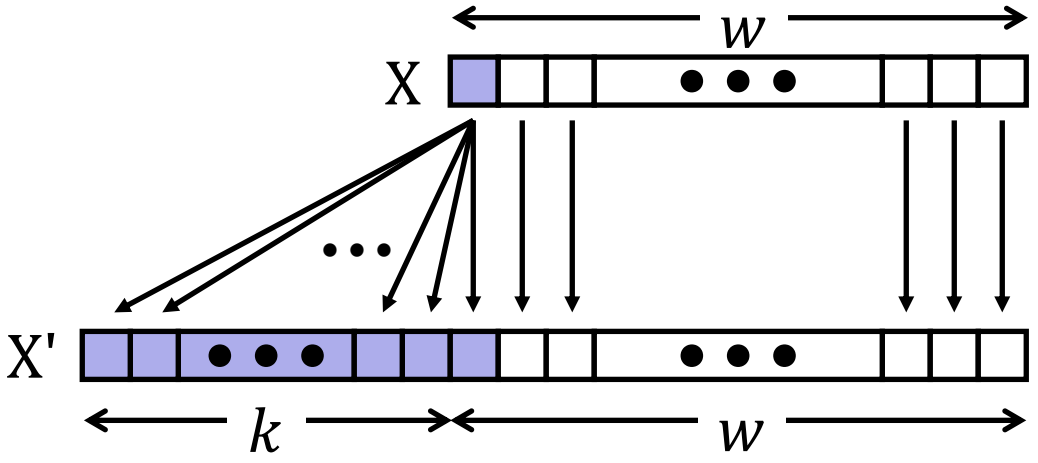
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

4 bit

0000	0
…	…
0100	4
…	
0111	7
1000	-8
…	…
1111	-1

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



45466