CS107 Lecture 10
Stack and Heap

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap

Array Trivia: `sizeof`

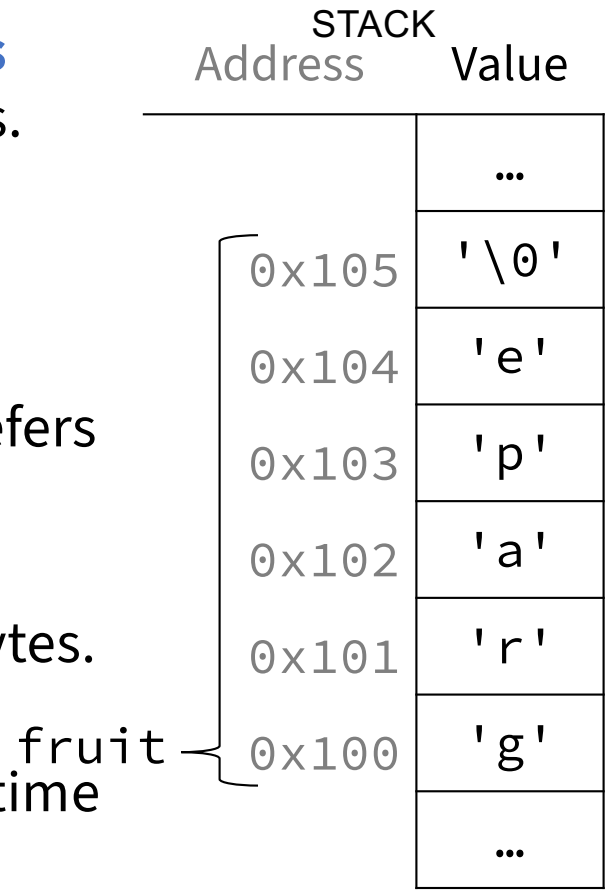
When you declare an array, **contiguous memory is allocated on the stack** to store each of its elements.

```
char fruit[6];
strcpy(fruit, "grape");
size_t size = sizeof(fruit); // 6
```

The array name (e.g., **`fruit`**) is not a true pointer. It refers to the entire array.

Here, **`sizeof`** returns the size of the entire array, in bytes.

That's because the array declaration is in scope at the time the compile-time **`sizeof`** operation is evaluated.



Array Trivia: `sizeof`

When you pass an **array** as a parameter, C really passes the address of the first element.

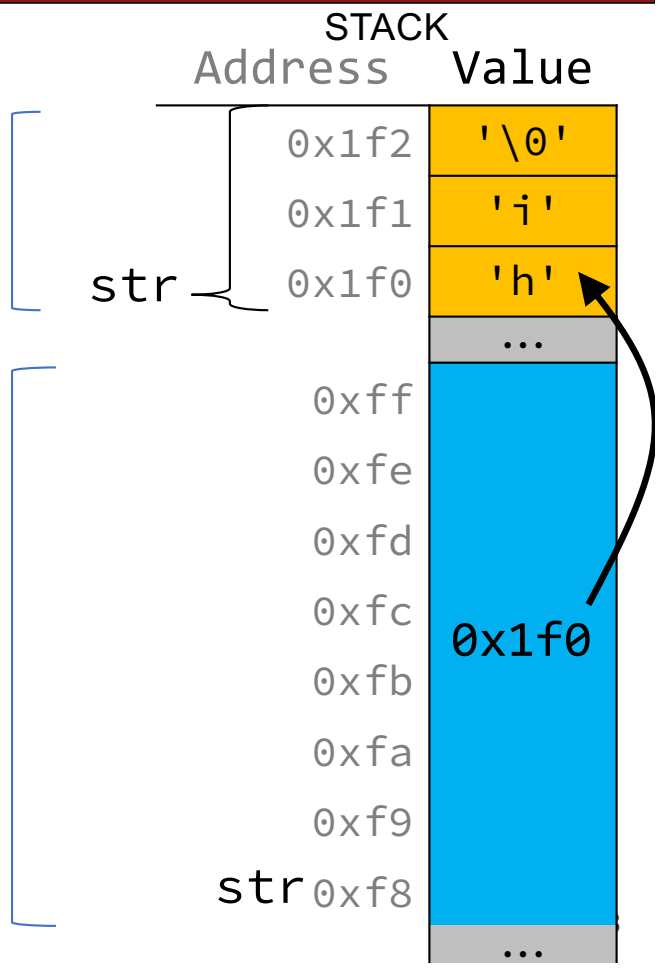
```
void func(char *str) {
   size_t size = sizeof(str); // 8
   ...
}

int main(int argc, char *argv[]) {
   char str[3];
   size_t size = sizeof(str); // 3
   strcpy(str, "hi");
   func(str);
   ...
}
```

The **amount of memory** set aside for **`func`**'s **`str`** is 8 bytes.

The **amount of memory** set aside for **`main`**'s **`str`** array is 3 bytes.

This is the **state of memory** just as **`func`** begins to execute.



Array Trivia: Pointer Arithmetic

When applying pointer arithmetic, we **advance a pointer by a certain number of bytes**.

```
char *a = "peach";          // e.g., 0xff0
char *b = str + 1;          // e.g., 0xff1
char *c = str + 3;          // e.g., 0xff3

printf("%s\n", a);          // prints peach
printf("%s\n", b);          // prints each
printf("%s\n", c);          // prints ch
```

READ-ONLY DATA

Address	Value
	…
0xff5	'\0'
0xff4	'h'
0xff3	'c'
0xff2	'a'
0xff1	'e'
0xff0	'p'
	…

Array Trivia: Pointer Arithmetic

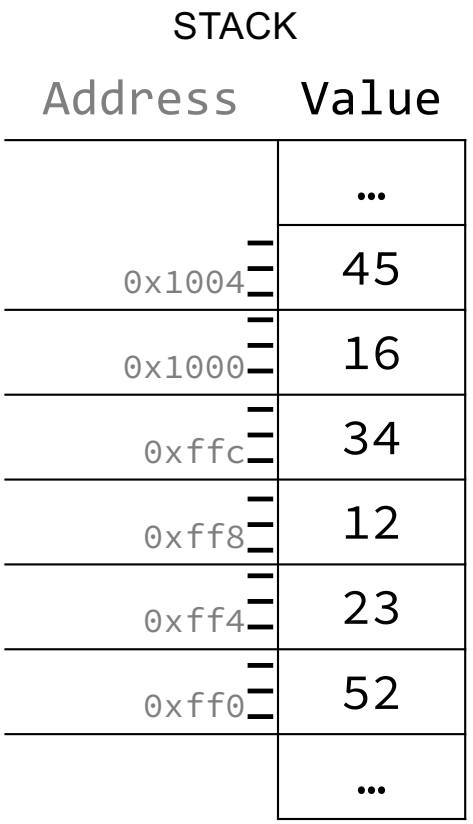
Pointer arithmetic advances a pointer by a **multiple of the pointee type's size**.

The 1, 2, and -1 offsets are **internally scaled** by **`sizeof(int)`**.

```
int numbers[] = {52, 23, 12, 34, 16, 45};
int *nums0 = numbers;   // shorter name
int *nums1 = nums0 + 1; // e.g., 0xff4
int *nums3 = nums1 + 2; // e.g., 0xffc
int *nums2 = nums3 - 1; // e.g., 0xff8

printf("%d\n", *nums1);              // 23
printf("%d\n", *nums2);              // 12
printf("%d\n", *nums3);              // 34
printf("%zu\n", nums3 - nums1);      // 2
```

Because **`nums1`** and **`nums3`** are both of type **`int *`**,
`nums3` – **`nums1`** evaluates to the **number of `int`s that fit in between them**.
Restated, if **`nums1 + 2`** equals **`nums3`**, then **`nums3 – nums1`** better equal 2.



CS107 Topic 3: Stack and Heap Memory

How can we effectively manage **all forms of memory** in our programs?

Why is answering this question useful?

- Illustrates how we can **efficiently pass data around using pointers**
- Illustrates how to construct data structures that **live longer than the function calls that create them**.

Learning goals:

- Understand the differences between **stack and heap memory** and **when to choose one over the other**.
- Become **fluent** in **`malloc`**, **`free`**, and **`realloc`** as standard C language directives used to **manage heap memory**.

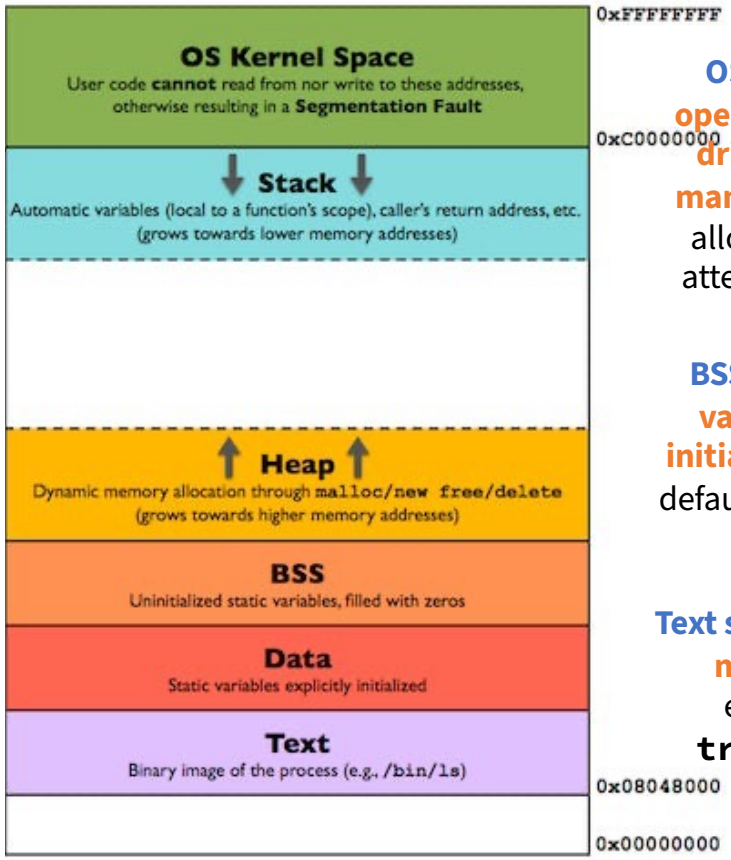
CS107 Topic 3: Stack and Heap Memory

this week's focus

Stack segment: Used for **function calls, local variables, and return addresses**. It typically occupies higher memory addresses and grows downwards towards the heap as it expands.

Heap segment: Used for **dynamic memory** allocation (e.g., via **`malloc`** in C, **`new`** in C++) and grows upwards towards higher addresses.

Data segment: Used to **store explicitly initialized global variables and constant**, including the string constants used throughout your code (e.g., `"apple"`, `"peach"`, `"rutabaga"`, `"absquatulation"`).



OS Kernel region: Used for the **operating system's kernel, device drivers,** and **low-level memory management code**. Your code isn't allowed to access this region and attempts to do so result in a crash.

BSS segment: Used for all **global variables that aren't explicitly initialized** before **`main`** is called. By default, all bytes in this segment start out as zeroes.

Text segment: Contains the **compiled machine code** of the running executable , like **`ls`**, **`emacs`**, **`triangle`**, **`sat`**, or **`automata`**

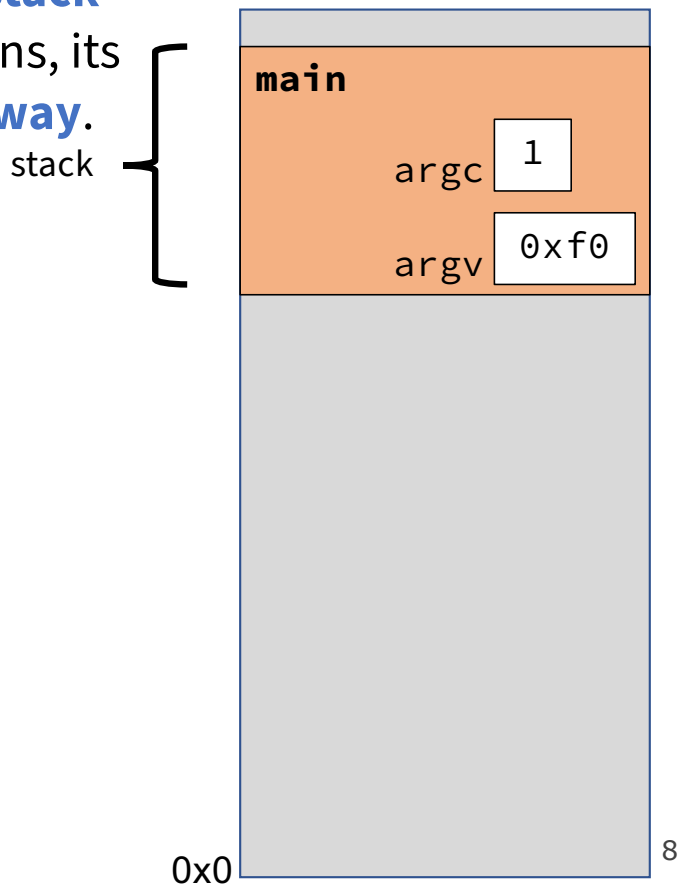
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



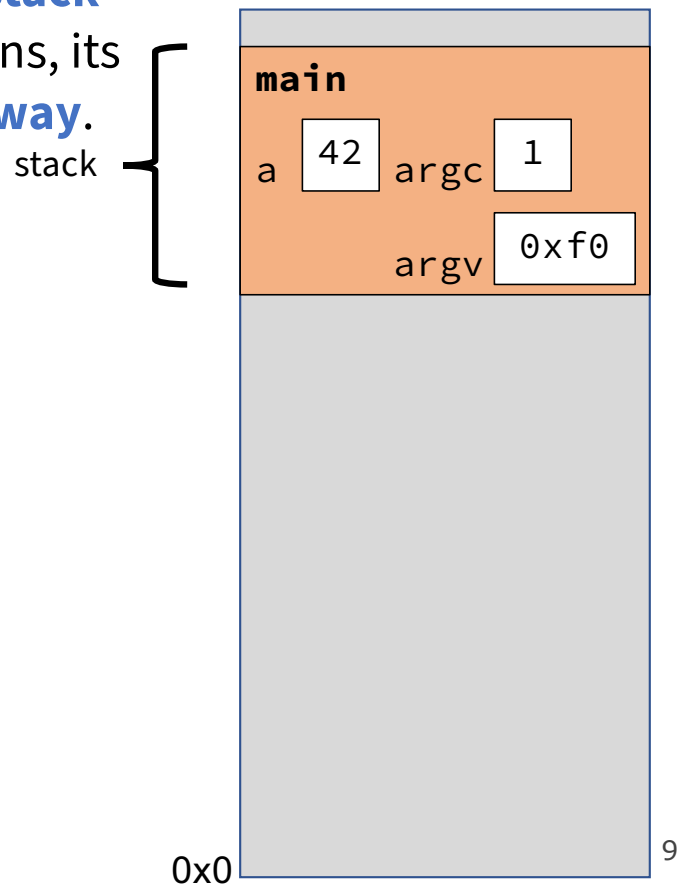
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



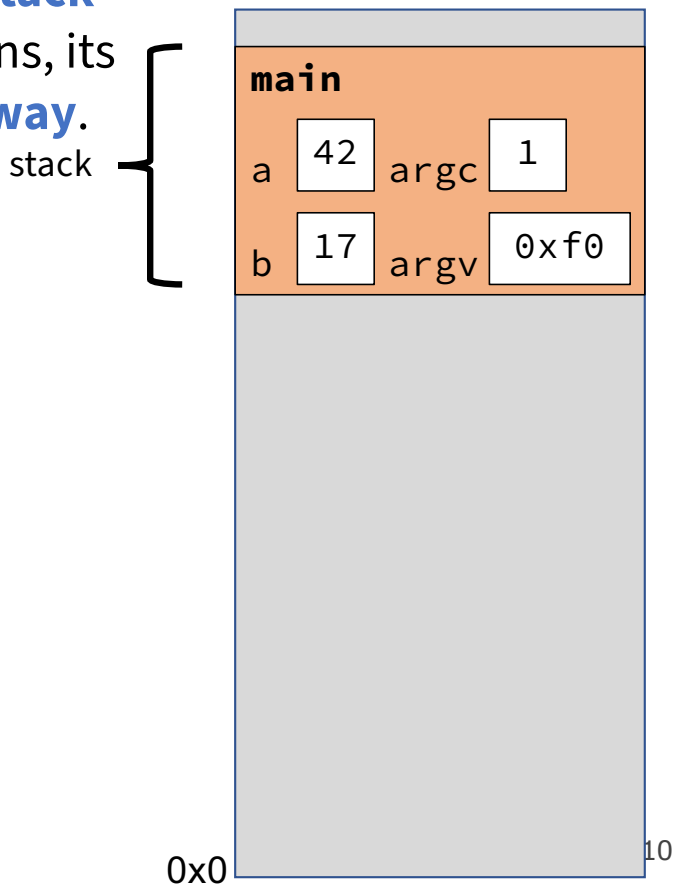
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



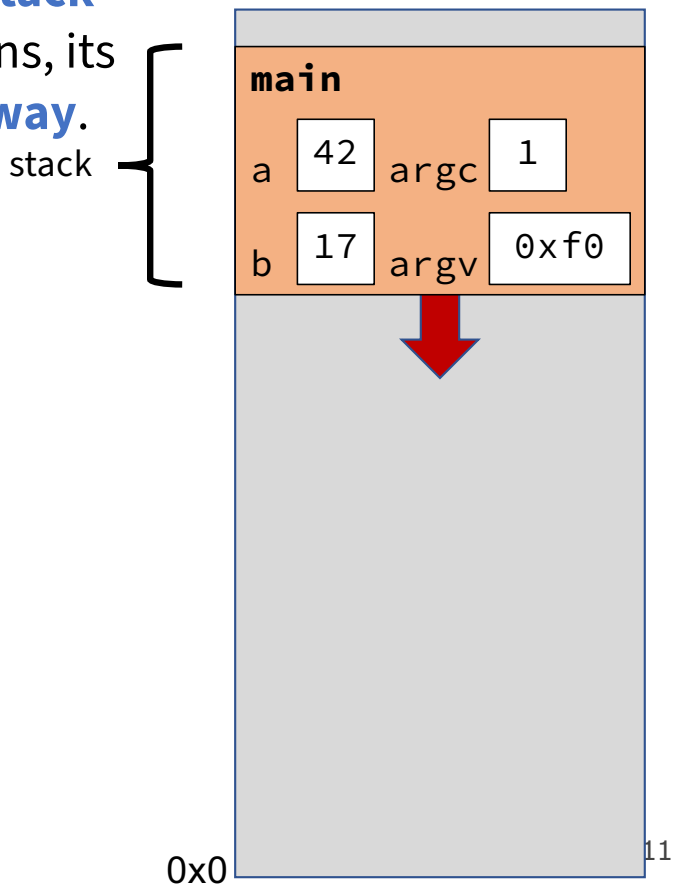
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



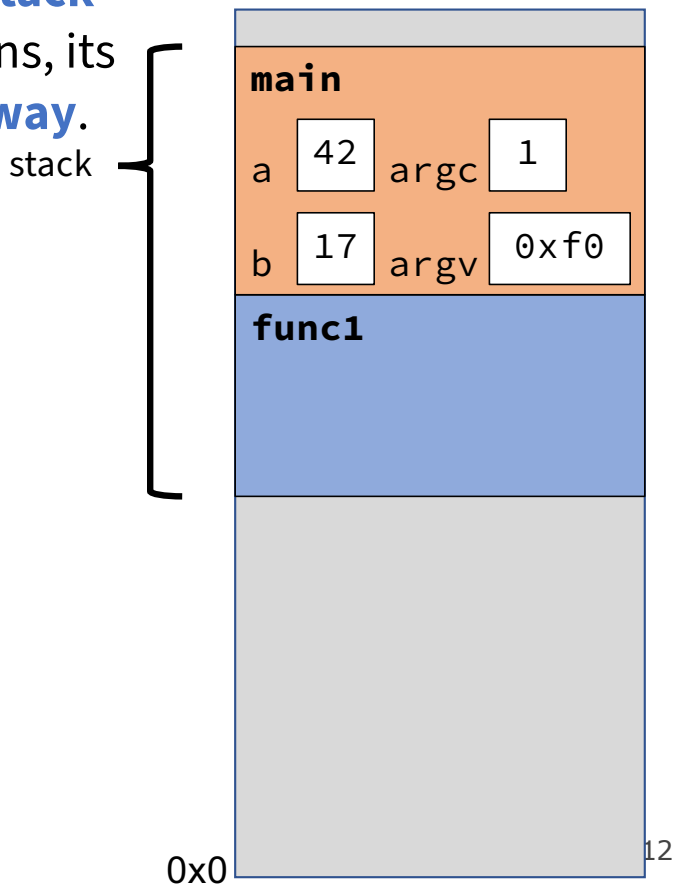
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



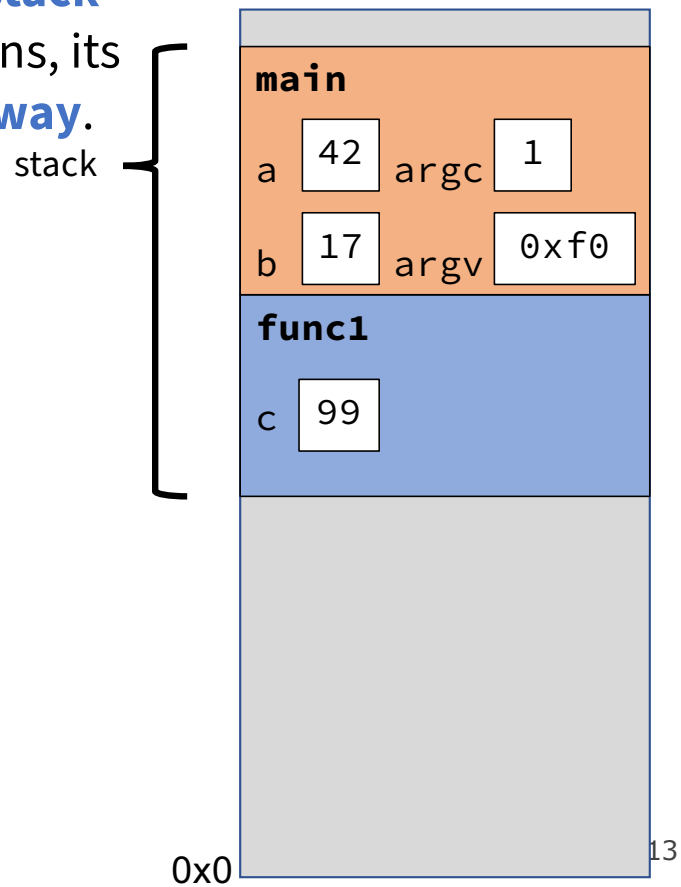
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



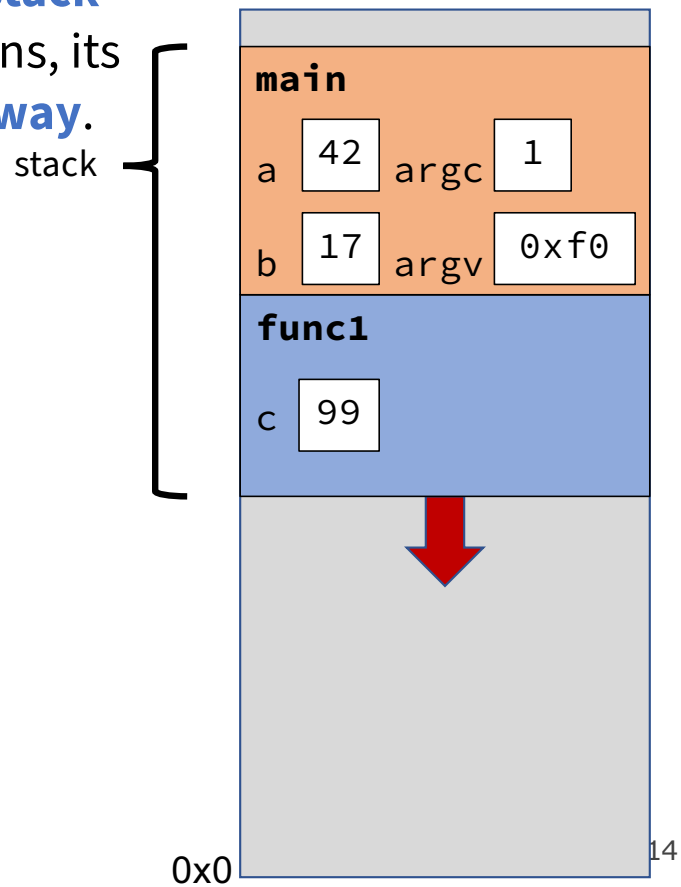
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



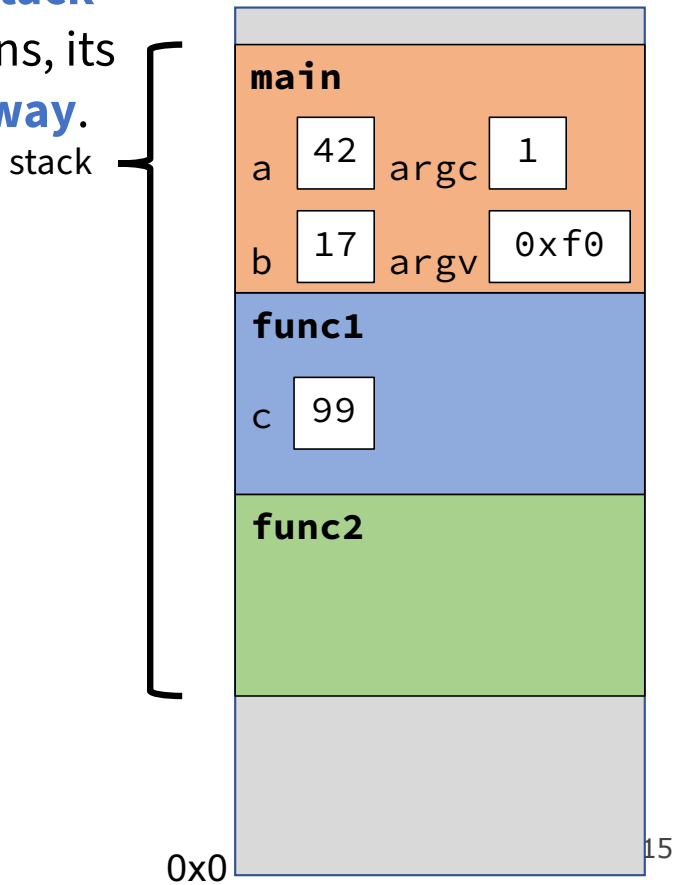
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



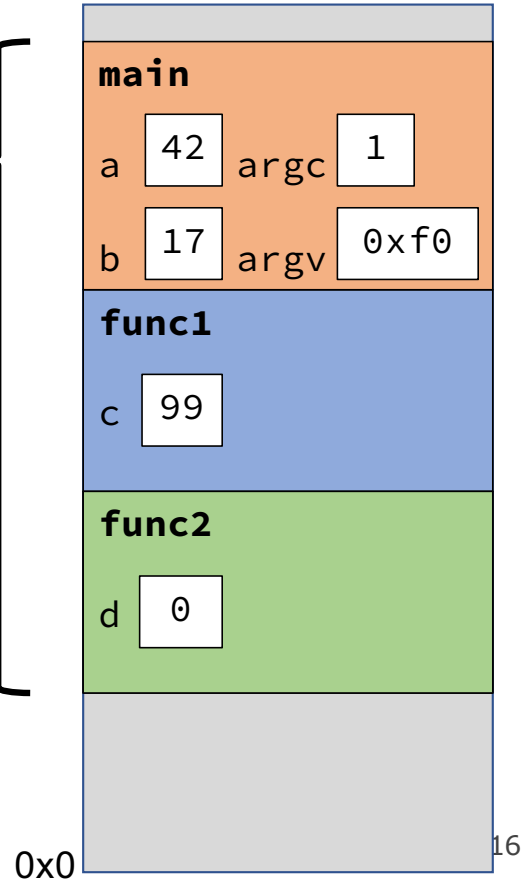
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



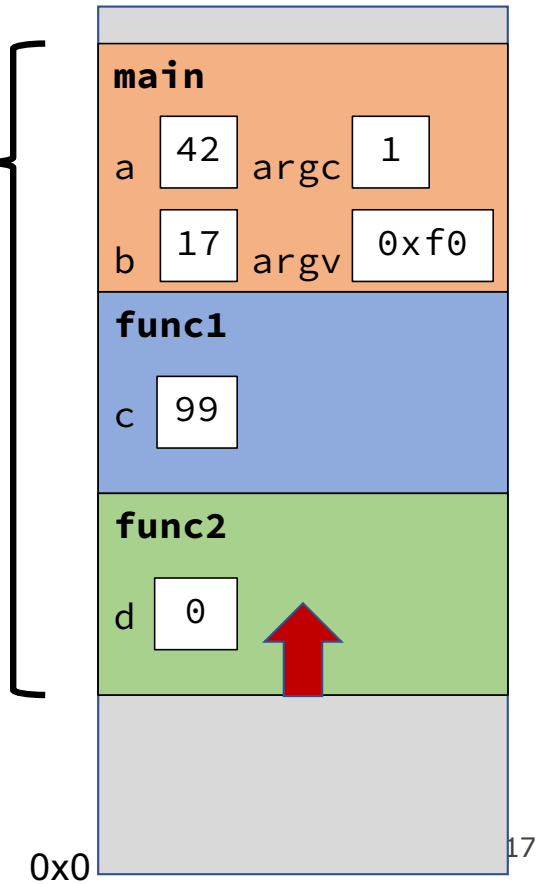
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



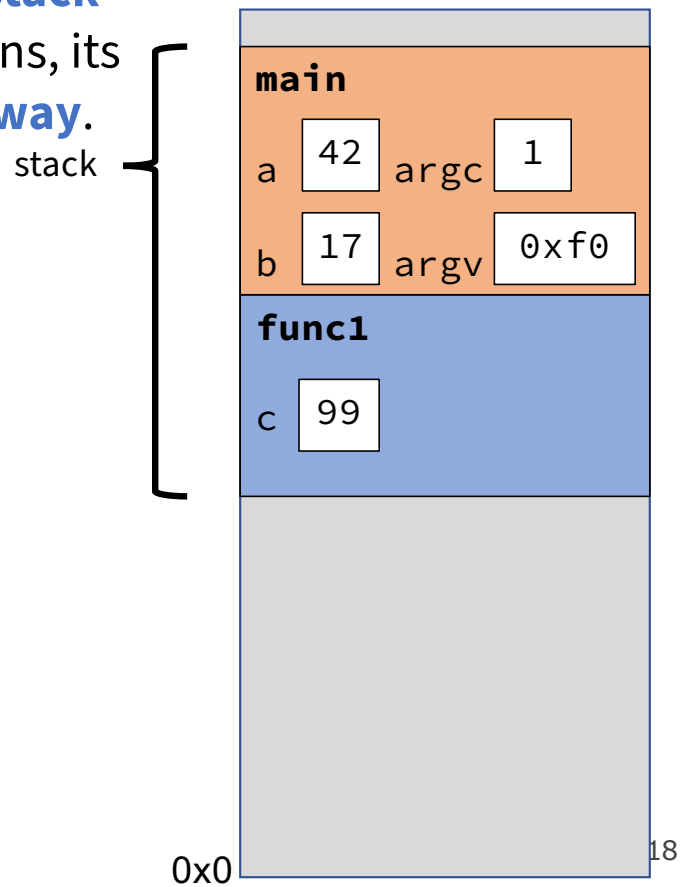
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



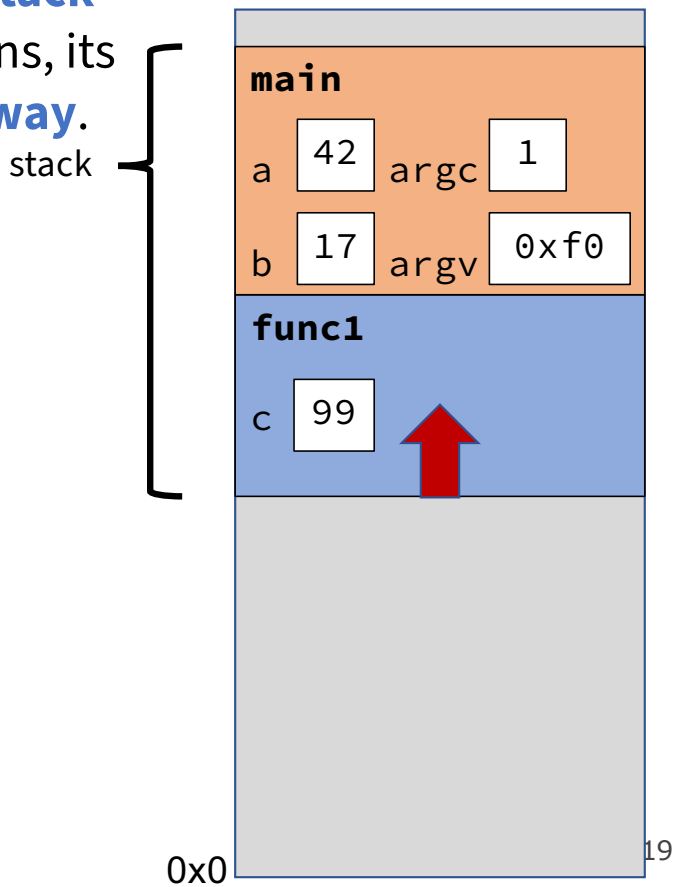
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



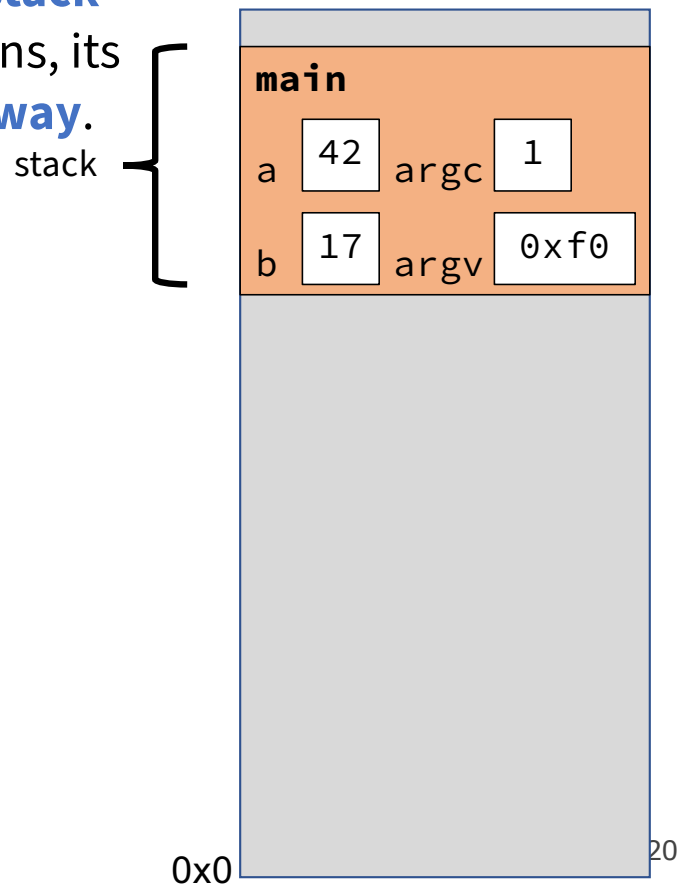
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



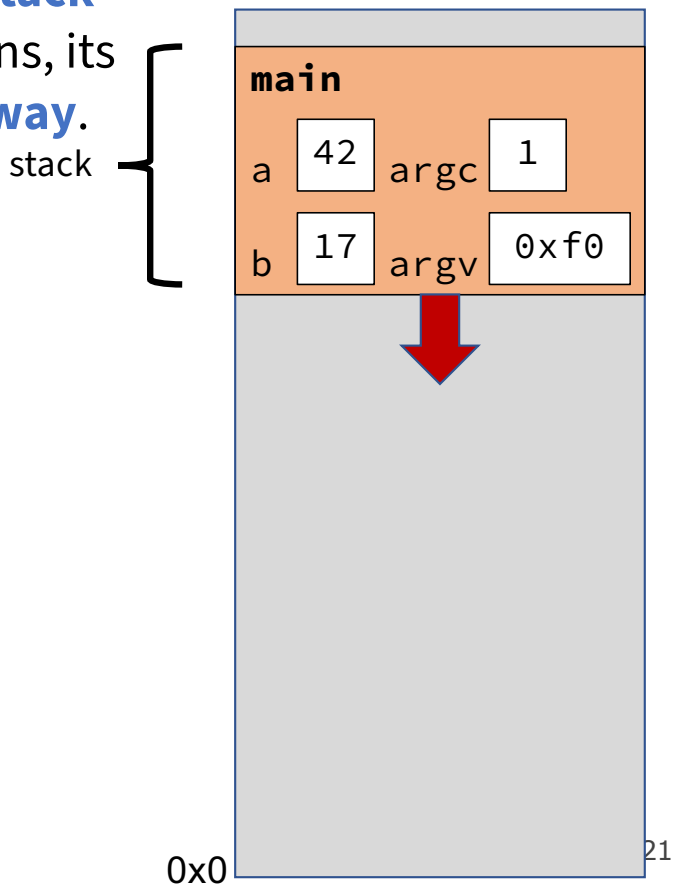
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



Stack Segment Case Study

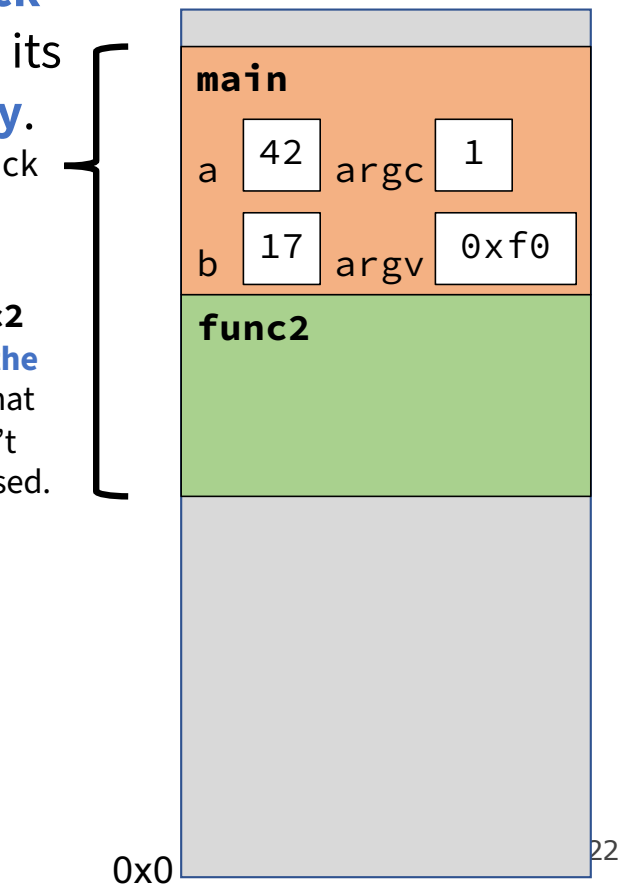
When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```

Note the stack frame for this call to **func2** **overlays the space previously used for the earlier call** to **func1**. The information that accumulated here by the **func1** call isn't relevant anymore, so the space can be reused.



Stack Segment Case Study

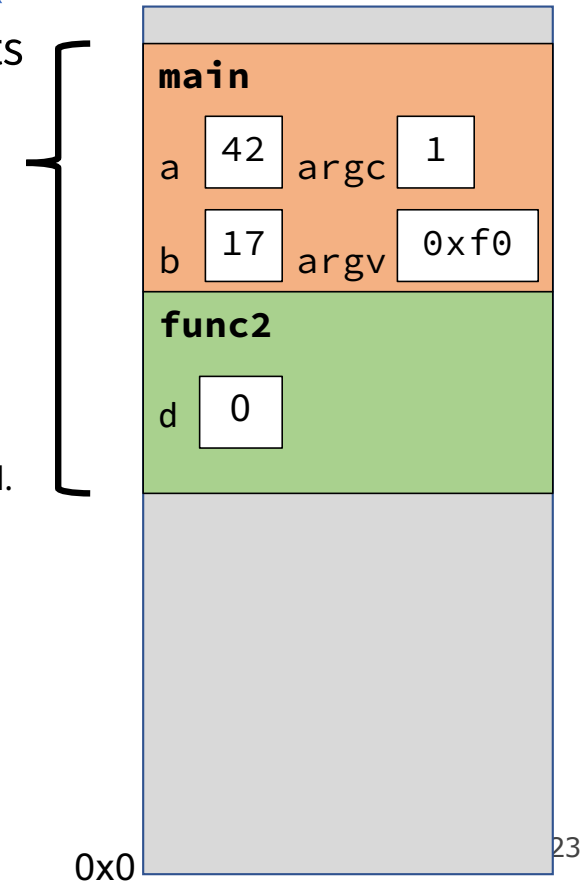
When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```

Note the stack frame for this call to **func2** **overlays the space previously used for the earlier call** to **func1**. The information that accumulated here by the **func1** call isn't relevant anymore, so the space can be reused.



Stack Segment Case Study

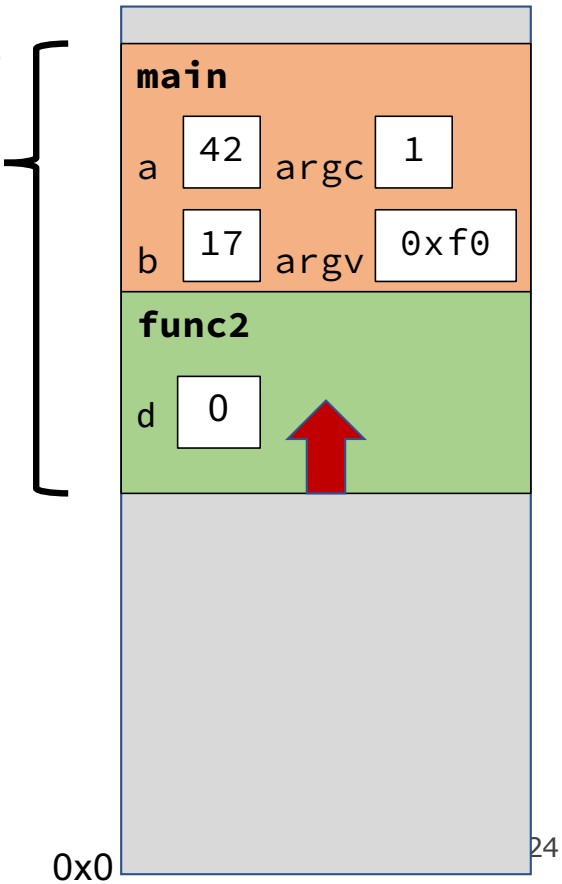
When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```

Note the stack frame for this call to **func2** **overlays the space previously used for the earlier call** to **func1**. The information that accumulated here by the **func1** call isn't relevant anymore, so the space can be reused.



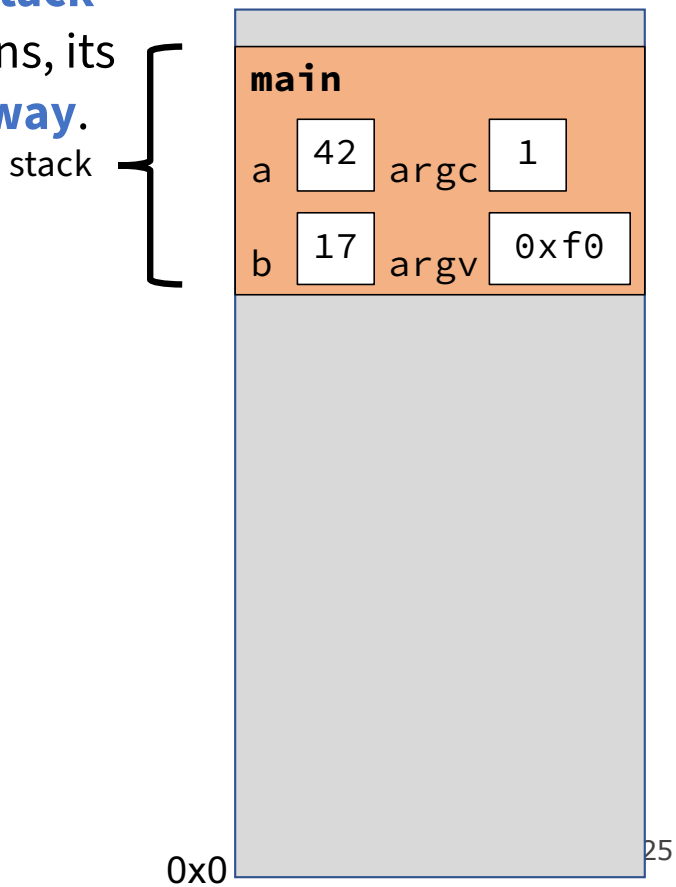
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



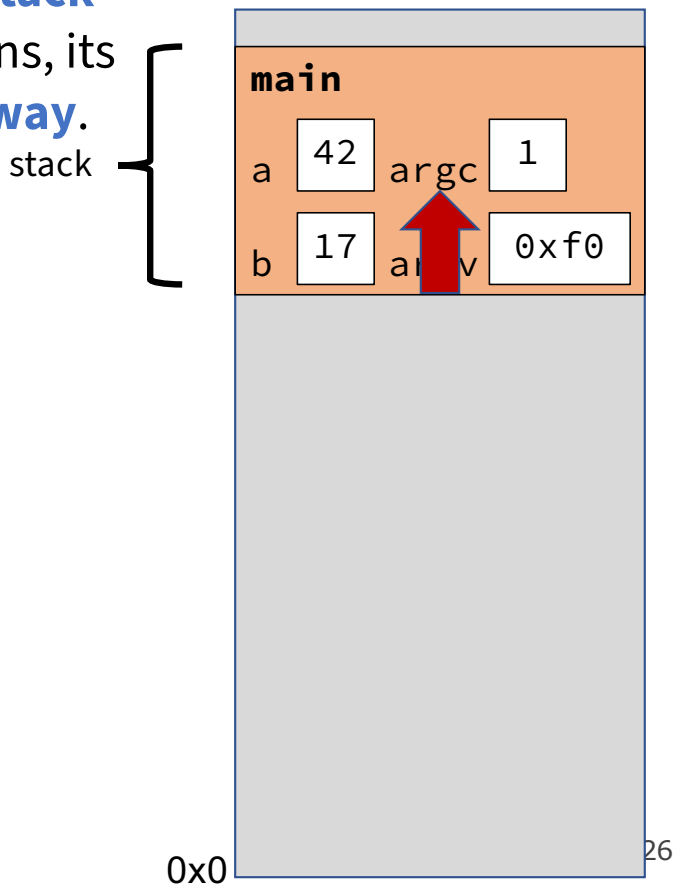
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```



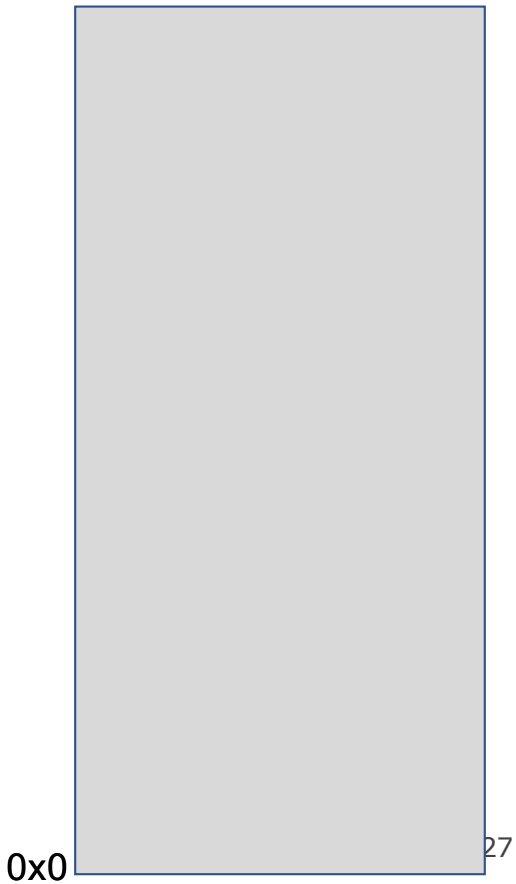
Stack Segment Case Study

When a function is called, a **new frame is pushed onto the stack** and makes space for **local variables**. When the function returns, its frame is **popped** and access to those locals **formally goes away**.

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    return 0;
}
```

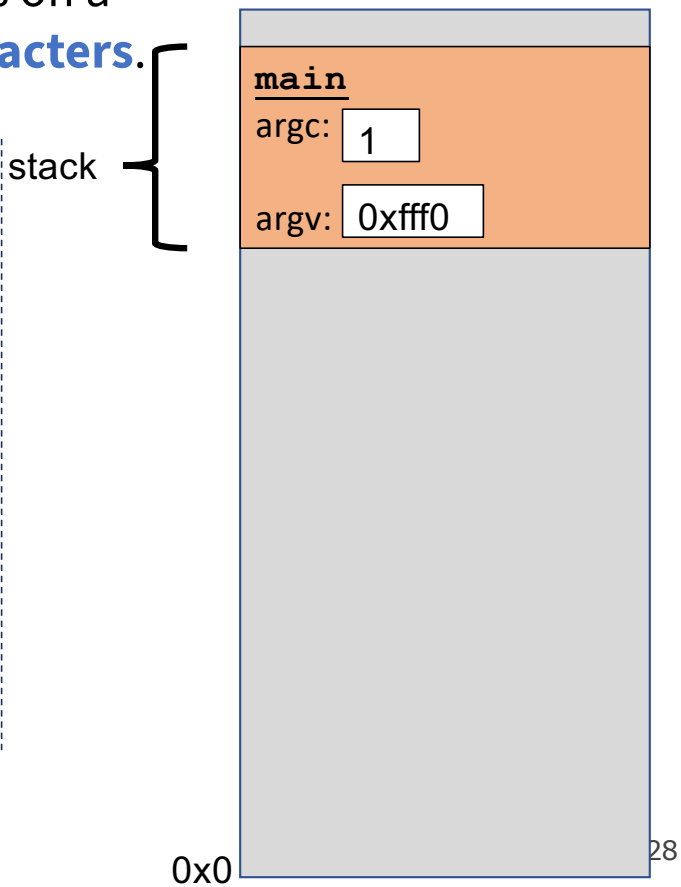


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

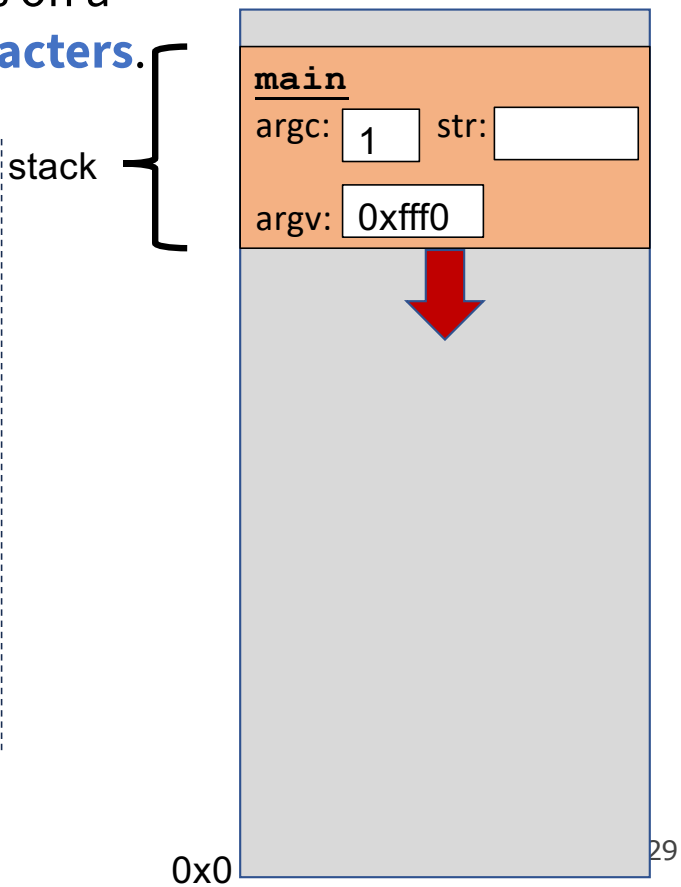


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

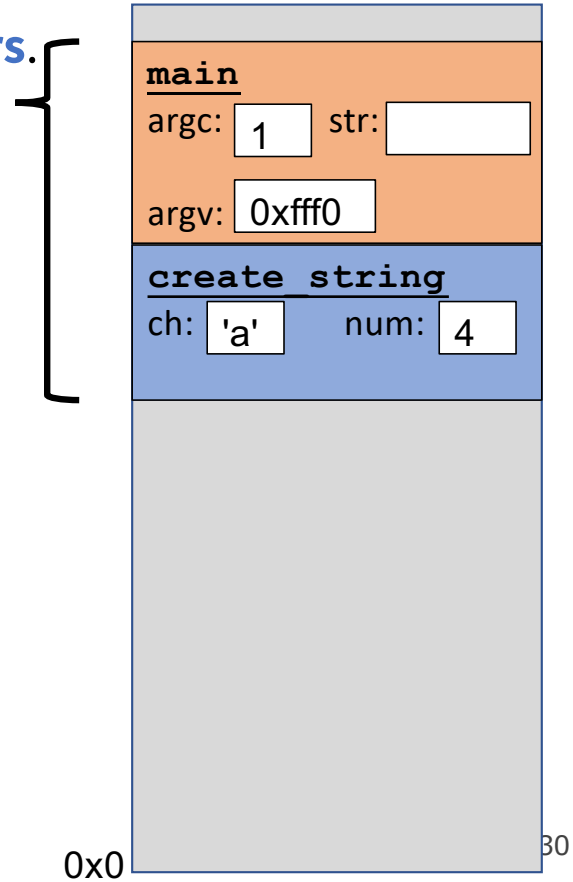


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

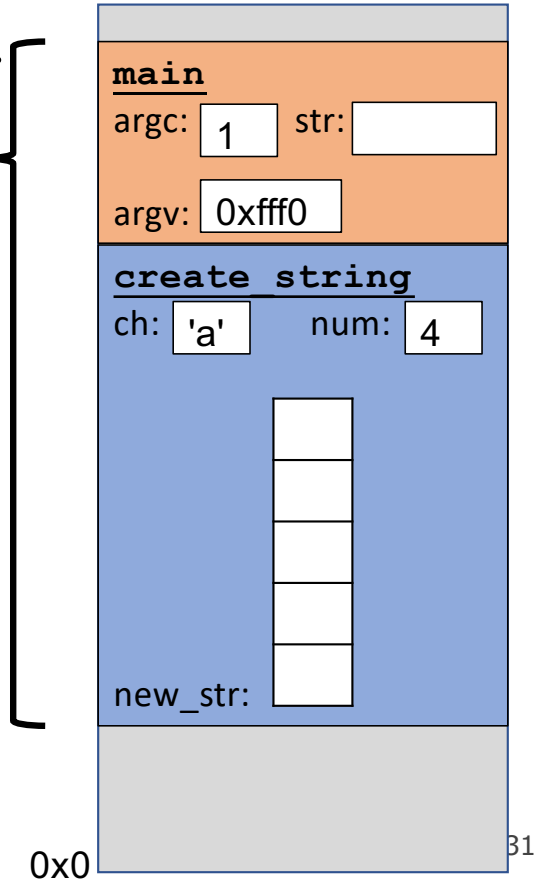


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

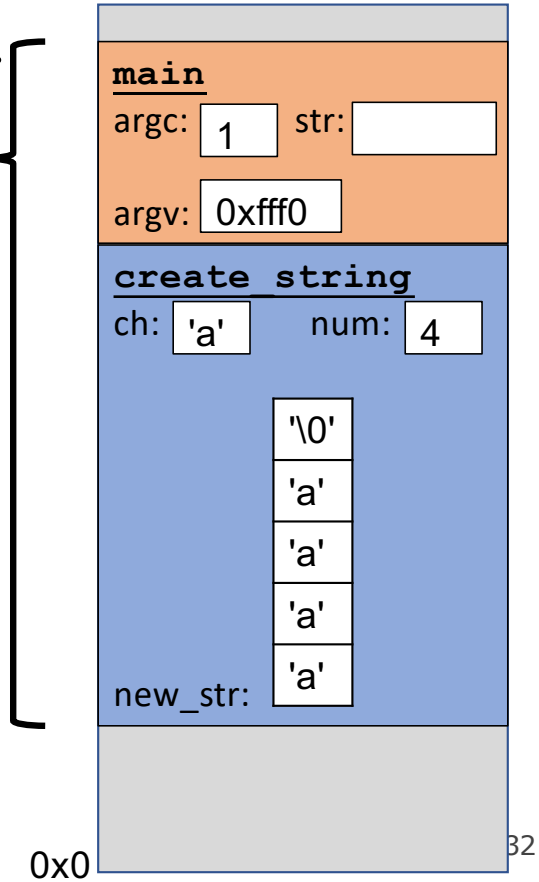


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```



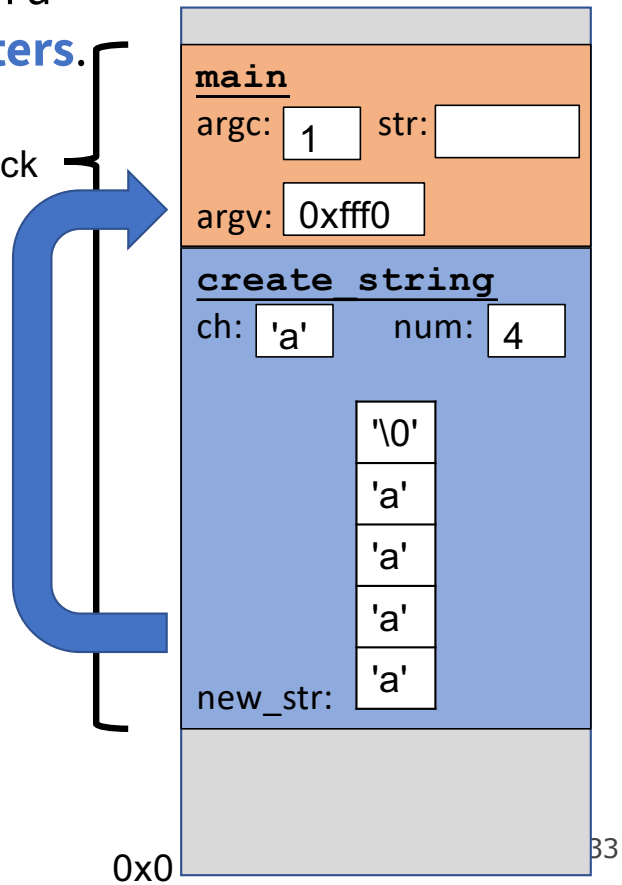
Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

returns e.g., 0xff50

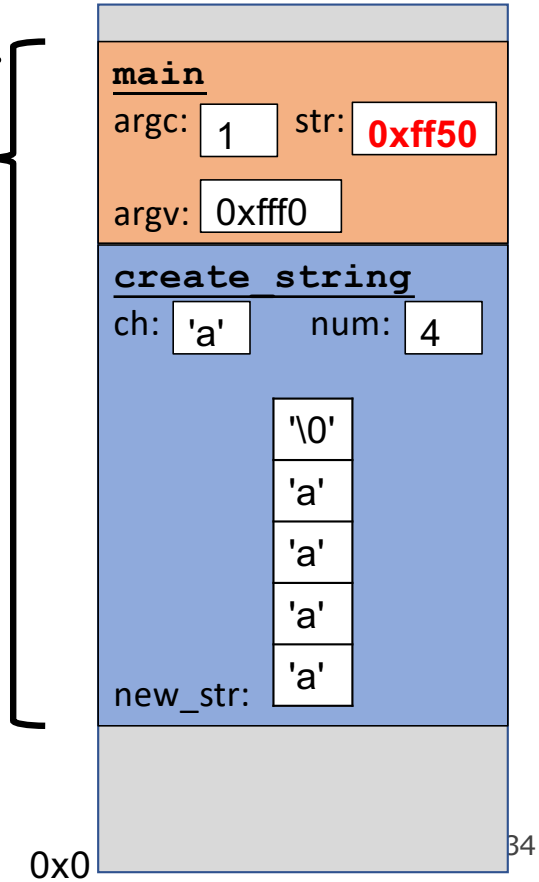


Stack Segment Case Study: Mayday, Mayday

Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```



Stack Segment Case Study: Mayday, Mayday

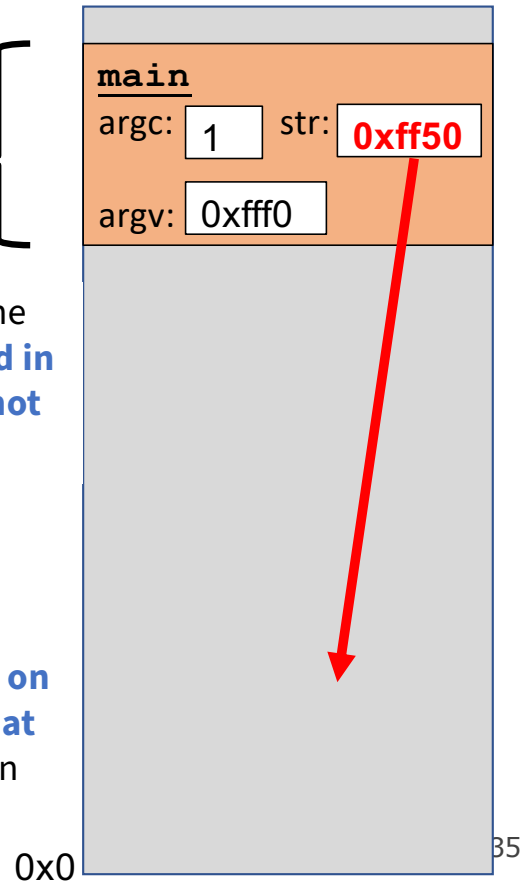
Here's a **well-intentioned** (but **broken**) **program** that relies on a (**broken**) helper function to **create a string of repeated characters**.

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (size_t 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\n", str); // want to print aaaa
    return 0;
}
```

Defcon 1: Local variables **disappear** when the function returns. These four **a**'s are **embedded in a local array that no longer exists**, so **str** is **not our address to dereference**.

In some cases, the array can be **pre-allocated on the stack** using **size information computed at runtime**. But that's **not ideal** if it requires an **inelegant restructuring of the code**.



Introducing `malloc`

Much as C++ allows you to via its **`operator new`**, C allows you to **dynamically allocate memory** using **`operator new`**'s equivalent: **`malloc`**

```
void *malloc(size_t size);
```

- **`malloc`** expects the **number of raw bytes** that should be allocated from the heap.
- **`malloc`** returns the **base address** of the freshly allocated memory.
  - **`malloc`** doesn't **know or care** if the memory is to be used as an **array**, a record, or something else.

Examples:

```
int *arr = malloc(n * sizeof(int));
char *t = malloc(strlen(s) + 1);
node *n = malloc(sizeof(node));
```

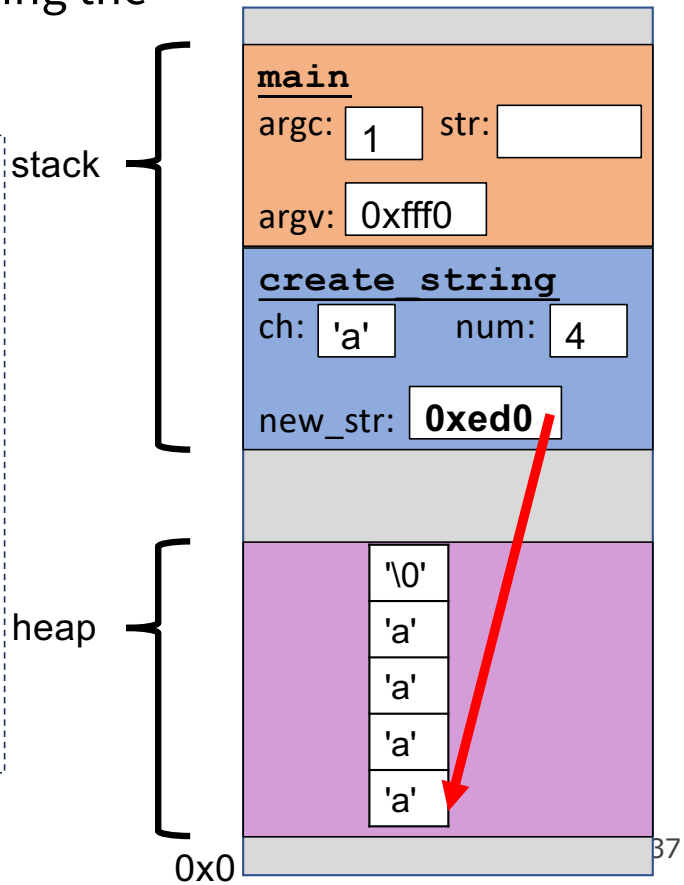
- The return type is a **`void *`** to denote an address to **generic (i.e., type-less) memory**.
  - You can set another pointer equal to it without casting.

Stack Segment Case Study: `malloc` Heals

Here's a **working version** that repairs the problem by allocating the new string on the heap using **`malloc`**.

```
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    assert(new_str != NULL);
    for (size_t 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\n", str); // want to print aaaa
    return 0; // should free str, will soon
}
```



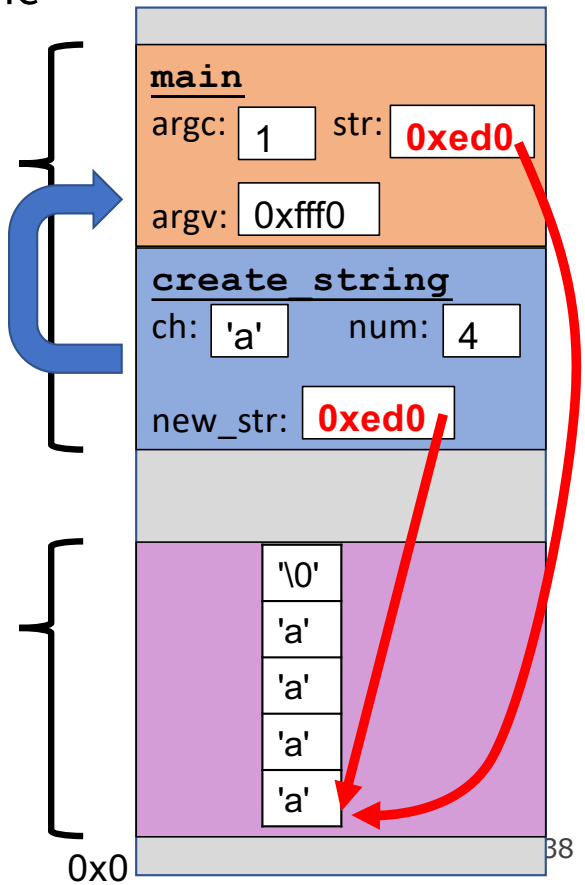
Stack Segment Case Study: `malloc` Heals

Here's a **working version** that repairs the problem by allocating the new string on the heap using **`malloc`**.

```
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    assert(new_str != NULL);
    for (size_t 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\n", str); // want to print aaaa
    return 0; // should free str, will soon
}
```

returns e.g., 0xed0

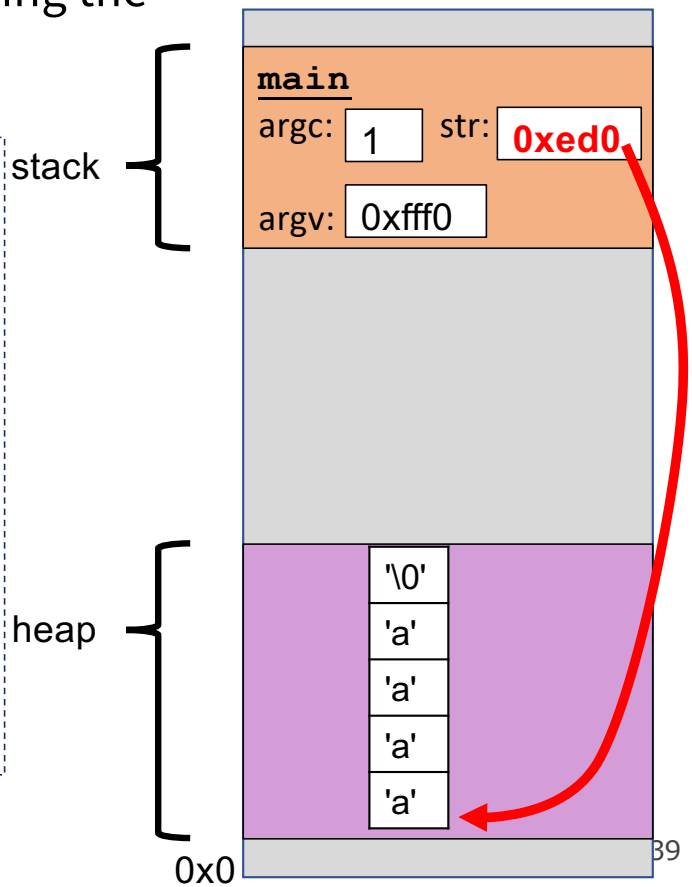


Stack Segment Case Study: `malloc` Heals

Here's a **working version** that repairs the problem by allocating the new string on the heap using **`malloc`**.

```
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    assert(new_str != NULL);
    for (size_t 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\n", str); // want to print aaaa
    return 0; // should free str, will soon
}
```

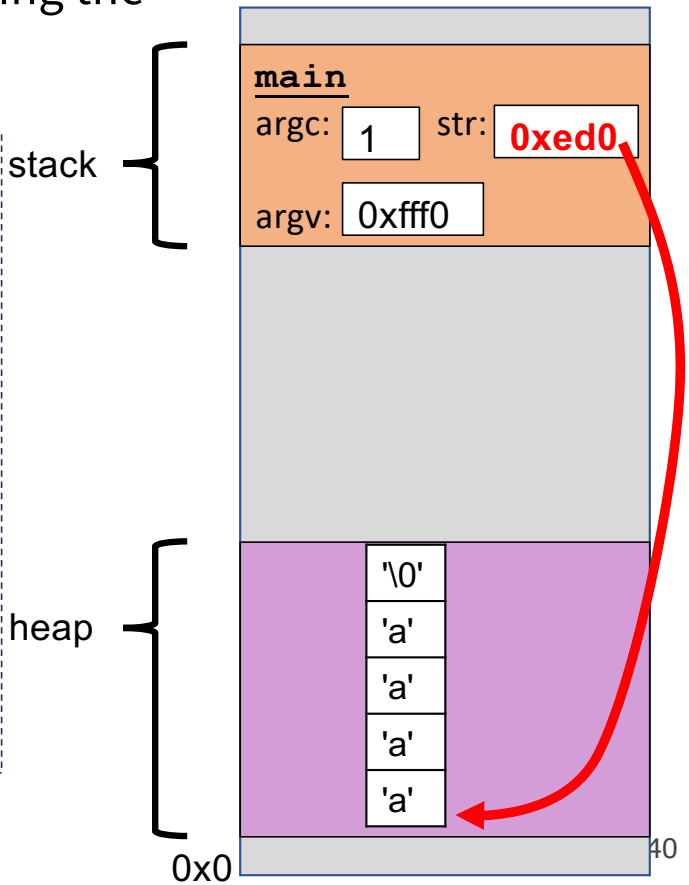


Stack Segment Case Study: `malloc` Heals

Here's a **working version** that repairs the problem by allocating the new string on the heap using **`malloc`**.

```
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    assert(new_str != NULL);
    for (size_t 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\n", str); // want to print aaaa
    return 0; // should free str, will soon
}
```



Dynamic Memory Etude

Let's write a function that returns an array of the first **`len`** multiples of **`mult`**.

```
int *array_of_multiples(int mult, int len) {
   /* TODO: arr declaration here */

   for (size_t i = 0; i < len; i++) {
      arr[i] = mult * (i + 1);
   }
   return arr;
}
```

Declare a pointer to **catch the heap-based address returned** by **`malloc`**.

Ensure **`malloc`**'s argument is the **number of bytes** needed to fulfill the allocation request.

CS107 policy: **`assert`** the address is non-**`NULL`**.

How should we declare and initialize **`arr`**?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(len * sizeof(int));`
D. `int *arr = malloc((len + 1) * sizeof(int));`

Dynamic Memory Etude

Let's write a function that returns an array of the first **`len`** multiples of **`mult`**.

```
int *array_of_multiples(int mult, int len) {
   int *arr = malloc(len * sizeof(int));
   assert(arr != NULL);
   for (size_t i = 0; i < len; i++) {
      arr[i] = mult * (i + 1);
   }
   return arr;
}
```

If an allocation error occurs (e.g., the process is **out of heap memory** or the heap has been **corrupted by flawed heap management**), **`malloc`** might return **`NULL`**.

`assert` will **intentionally terminate the program** if the condition evaluates to **`false`**. Memory allocation errors are a big deal, and we should end the program the moment we see them. Otherwise, your program will proceed and likely crash in a much more anonymous manner that might not be easily traced back to a failed **`malloc`** call.

In practice, you **never ship code** with active **`assert`** statements, but they are used during the development process to **verify that certain expectations are being met** and minimize the chances of runtime surprises. (The **`assert`** statements can be stripped out of the executable pretty easily during compilation.)

Dynamic Memory Etude

Let's write a function that returns an array of the first **`len`** multiples of **`mult`**.

```
int *array_of_multiples(int mult, int len) {
   int *arr = malloc(len * sizeof(int));
   assert(arr != NULL);
   for (size_t i = 0; i < len; i++) {
      arr[i] = mult * (i + 1);
   }
   return arr;
}
```

If an allocation error occurs (e.g., the process is **out of heap memory** or the heap has been **corrupted by flawed heap management**), **`malloc`** might return **`NULL`**.

`assert` will **intentionally terminate the program** if the condition evaluates to **`false`**. Memory allocation errors are a big deal, and we should end the program the moment we see them. Otherwise, your program will proceed and likely crash in a much more anonymous manner that might not be easily traced back to a failed **`malloc`** call.

In practice, you **never ship code** with active **`assert`** statements, but they are used during the development process to **verify that certain expectations are being met** and minimize the chances of runtime surprises. (The **`assert`** statements can be stripped out of the executable pretty easily during compilation.)

When Heap-Based Allocation Makes Sense

Here are **four common scenarios** where dynamic memory allocation is preferable to stack allocation:

- **When memory must outlive the current function call**
  Heap-based memory persists beyond the function call allocating it, unlike stack variables whose lifetime ends when the allocating function returns.
- **When building flexible or resizable data structures**
  Linked lists, trees, hash tables, and dynamically expanding arrays require memory that can be allocated, reallocated, and freed on demand.
- **When the size required is very large or not easily predicted**
  Larger arrays and data structures can exhaust the entire stack segment, whereas the heap segment can manage much larger allocations.
- **When data must be shared across multiple functions or modules**
  Heap memory allows multiple parts of a program to reference and modify the same data without copying, and that data isn't easily allocated in the `main` function.

Other heap allocators: `calloc`, `strdup`

`void *calloc(size_t count, size_t size);`

`calloc` is a **heap allocation function** **just** like **malloc**, with one important extra guarantee—it **zeroes out the memory** before it returns.

Examples:

```
int *counts = calloc(26, sizeof(int)); // all zeroes
bool *answers = calloc(n, sizeof(bool)); // all falses
struct node ** = calloc(num_buckets, sizeof(struct node *)); // all NULLs
```

`char *strdup(const char *str);`

`strdup` is a convenience function for **duplicating C strings onto the heap**.

Example:

```
char *news = strdup("disinformation");
news[0] = 'm';
char *british = strdup("disorganization")
british[9] = 's';
```

Like anything created using **`malloc`**, anything created using **`calloc`** or **`strdup`** **should be freed** when it's no longer needed. Again, soon!

Cleaning Up with `free`

```
void free(void *ptr);
```

`free` accepts an address previously returned by **`malloc`**, **`calloc`**, **`strdup`**, or (the soon to be discussed) **`realloc`** and donates the memory there back to the heap.

Simple Example:

```
char *earth = strdup("earth");
char *quake = strdup("quake");
char *earthquake = malloc(strlen(earth) + strlen(quake) + 1);
strcpy(earthquake, earth);
free(earth);
strcat(earthquake, quake);
free(quake);
printf("%s\n", earthquake);
free(earthquake); // three allocations reverted by three frees
```

When you **fail to free heap memory** you no longer need before letting go of it, you **leak** that memory.