CS107 Fall 2019, Lecture 7
Stack and Heap

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

Warmup exercise

Welcome to Week 4! We are knee-deep in C and system memory management.

Share your CS107 experience with your neighbor:

- What is one concept you've found interesting/exciting so far?
- What is one tool/strategy you've found helpful for assignments (diagrams, man pages, gdb print, gdb break, valgrind, printf, etc.)?

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

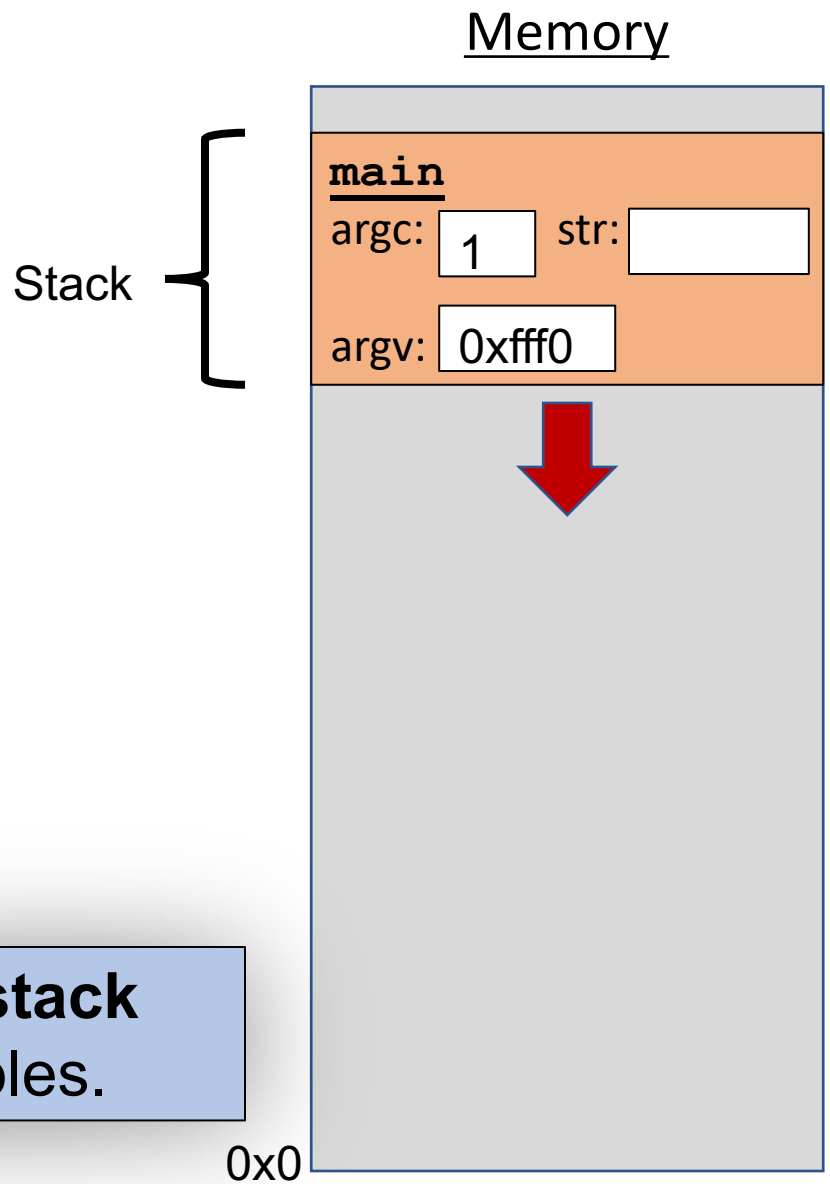
```
cp -r /afs/ir/class/cs107/samples/lectures/lect7 .
```

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

The Stack

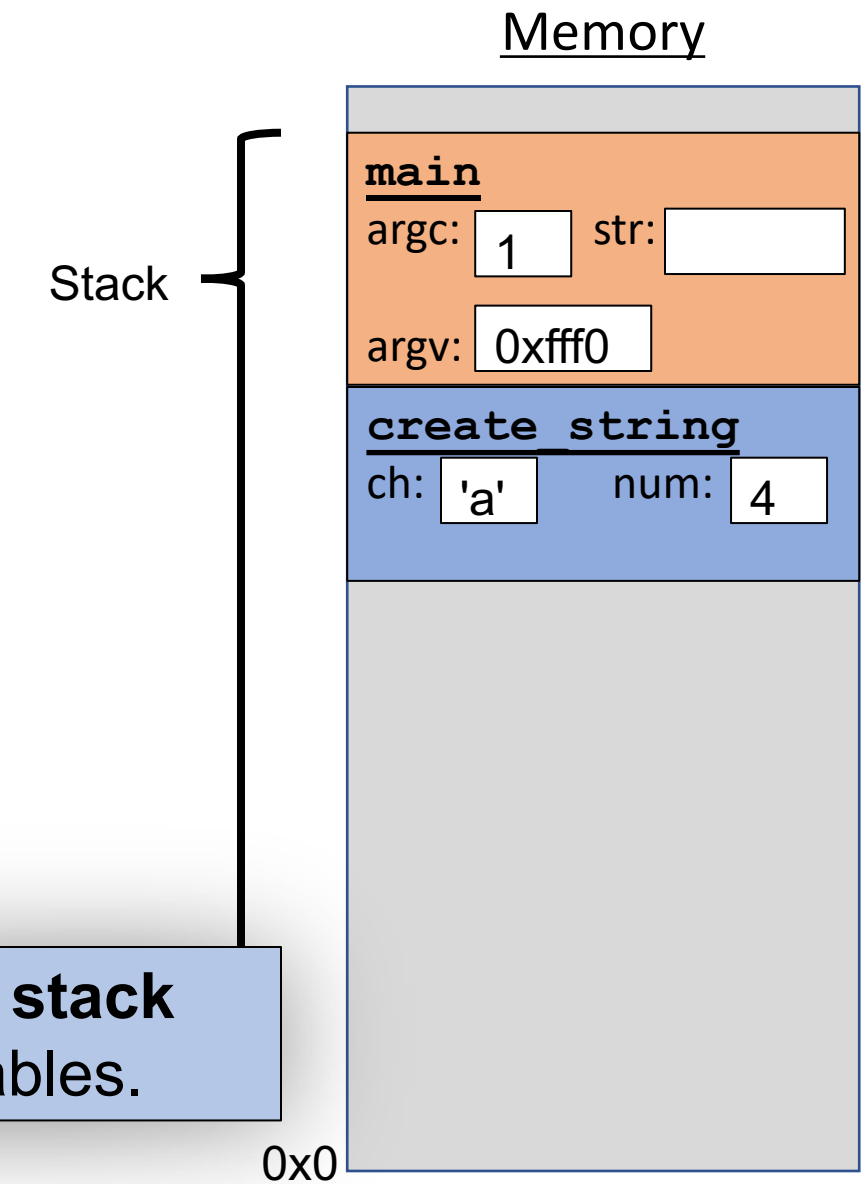

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```



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

The Stack

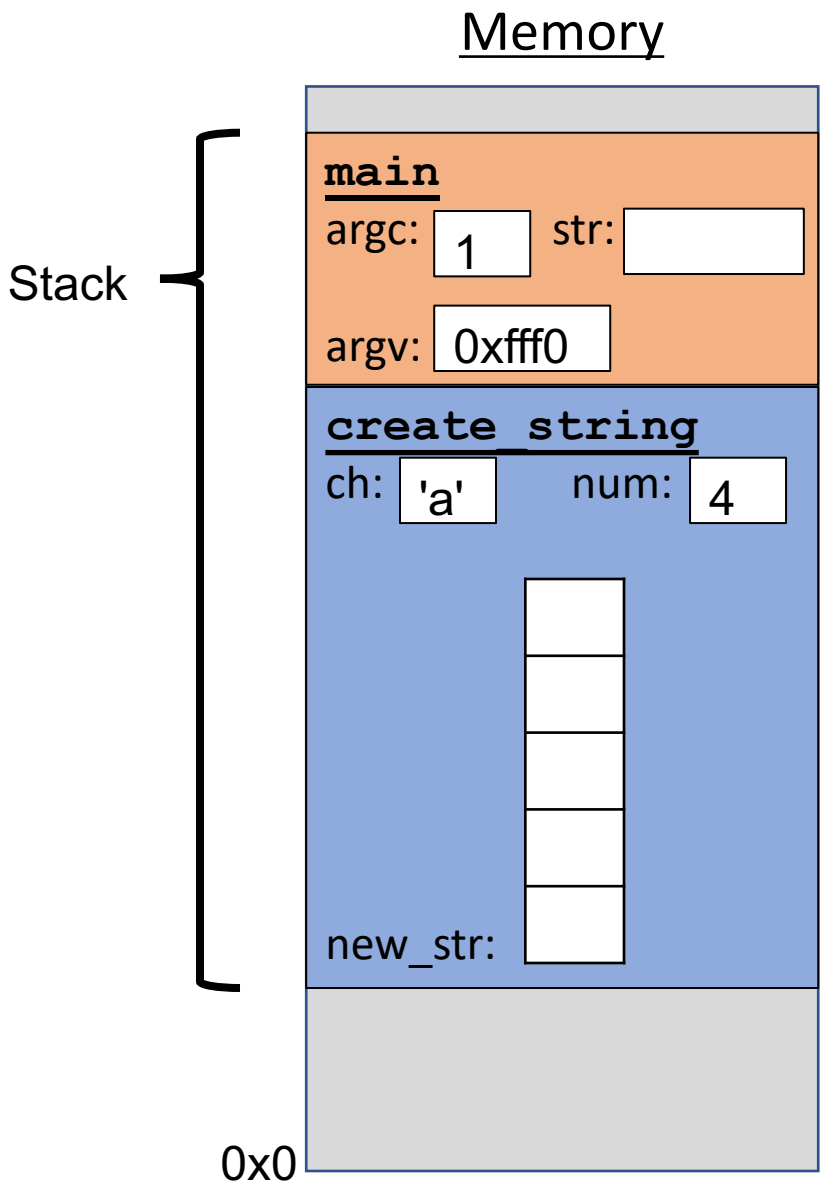

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```



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

The Stack

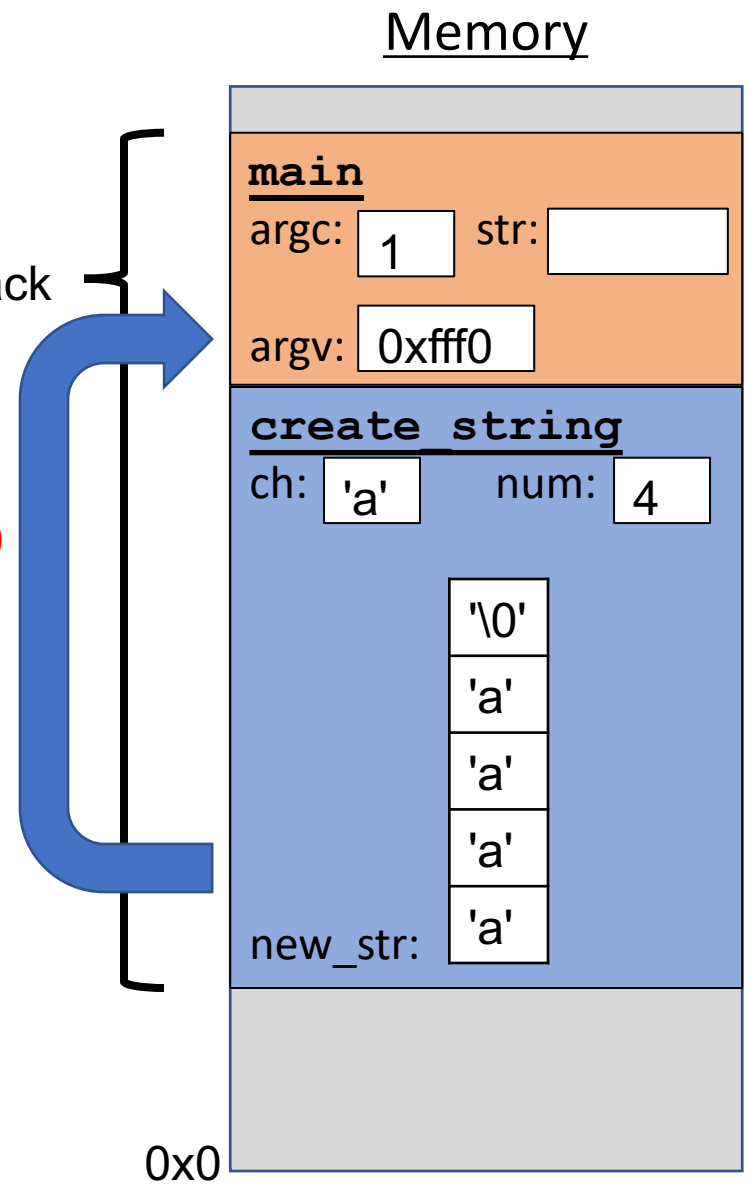

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```



The Stack


```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

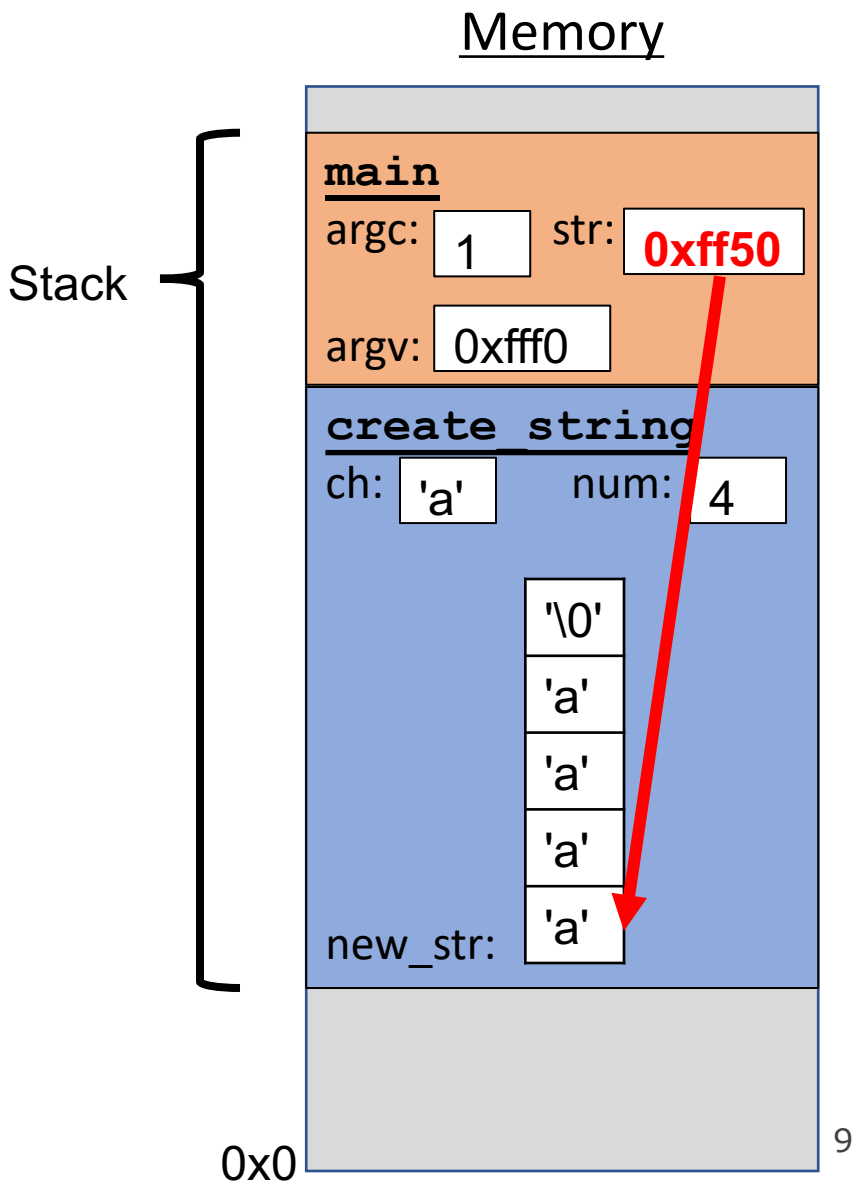
Returns e.g. 0xff50



The Stack


```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

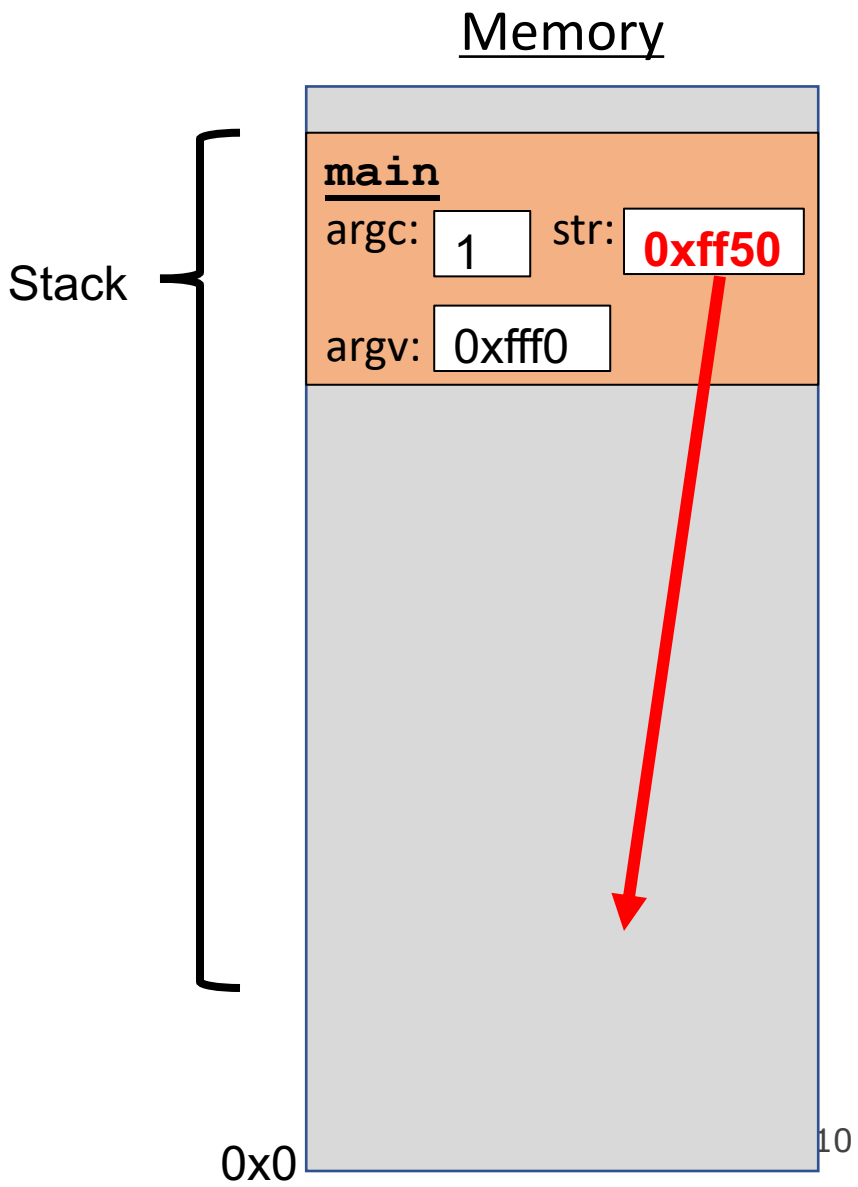
stack-allocated



The Stack


```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

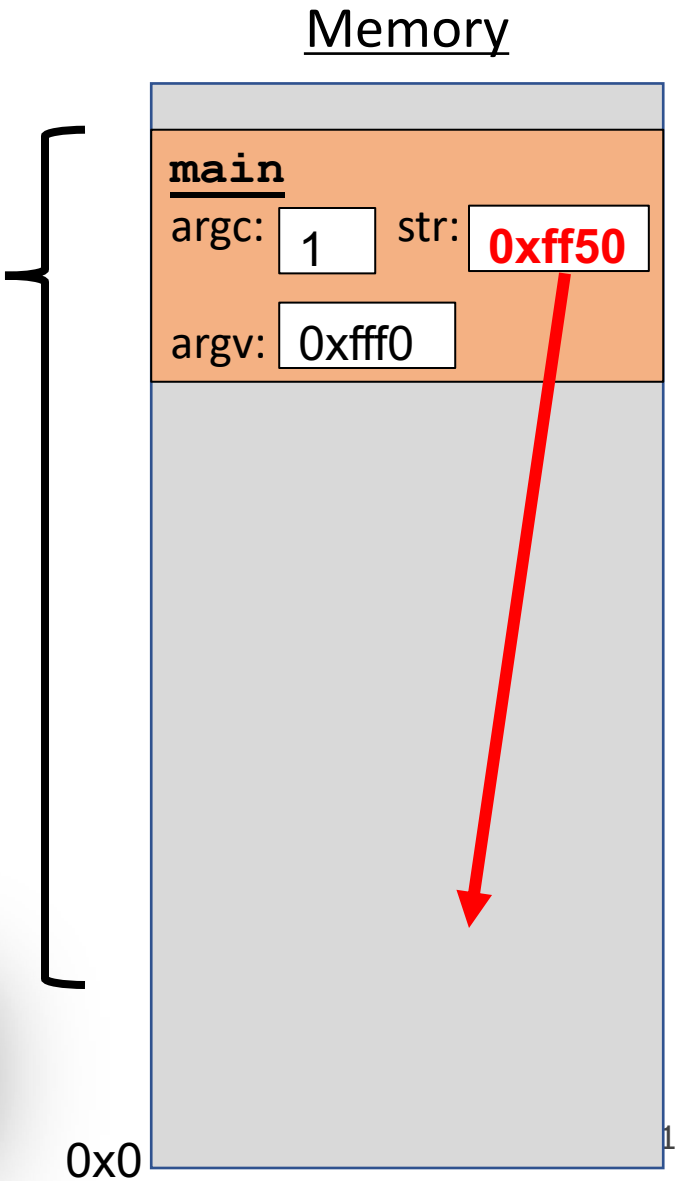
stack-allocated



The Stack


```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack-allocated



⚠ Stack memory gets reused by other function calls when a function returns!

Stacked Against Us

How do we make sure memory content persists after a function exits?

If we're committed to stack-only memory: let the caller allocate memory and pass it to the callee.

(we'll learn heap memory later today)

Programming with the stack (1/2)

Strategy #1: Allow callee to modify existing arrays.

```
void binky(char *ptr) {
    ptr[0] = 'B';
}

int main(int argc, char *argv[]) {
    char arr[] = "asdf";
    modify_array(arr);
    printf("%s", arr);        // Bsdf
    return 0;
}
```

Programming with the stack (2/2)

Strategy #2: Pass in a pre-allocated **buffer** that the caller can write to.

```
void pig_latin(char *out, char *in) {
    /* write changes to out */
}

int main(int argc, char *argv[]) {
    char str[] = "be";
    char converted[50];                    // output buffer
    pig_latin(converted, str);
    printf("%s %s", str, converted);       // be ebay
    return 0;
}
```

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

A heap of possibilities

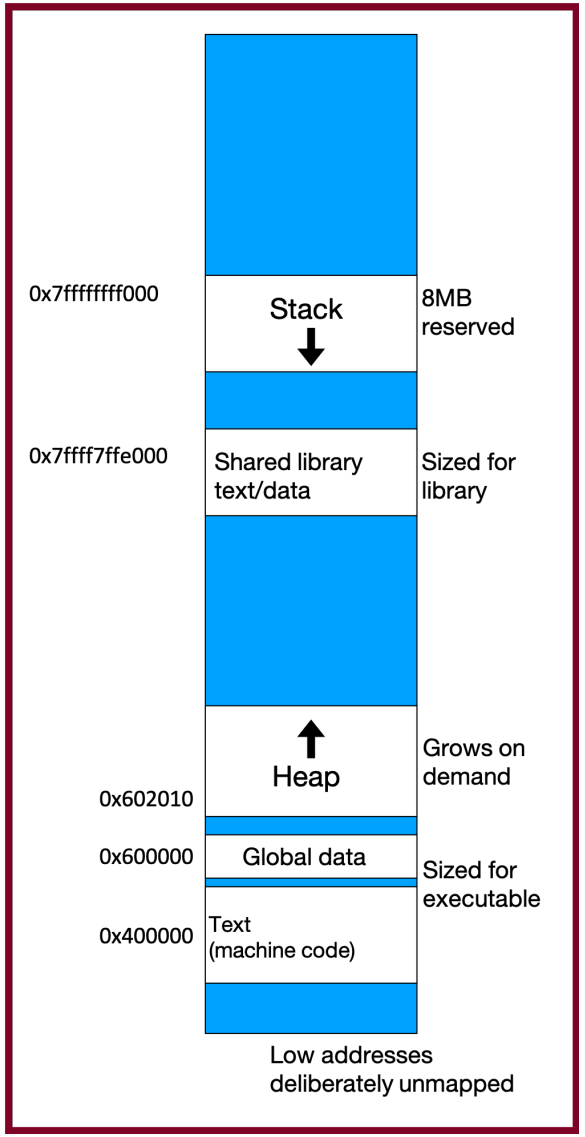
How do we make sure memory content persists after a function exits?

Let the callee allocate memory on the heap, with the guarantee that the caller will later free this memory.

The Heap

- The **heap** is a part of memory that you can manage yourself.
- Unlike the stack, the memory will only get reused when you explicitly clean it up.
- 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**.



malloc

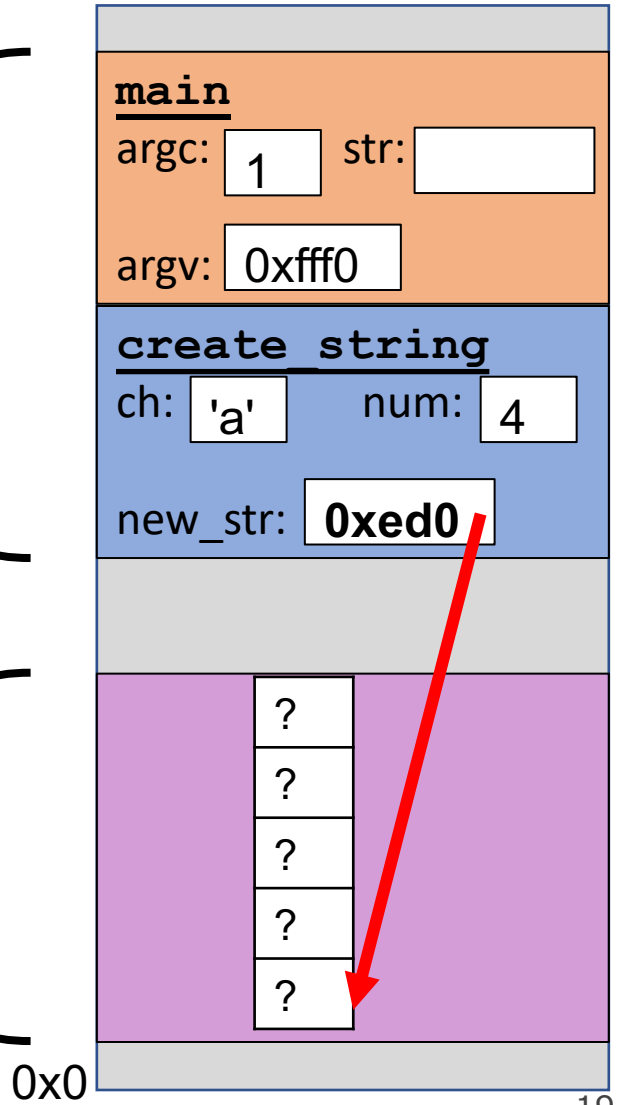
```
void *malloc(size_t size);
```

To allocate memory on the heap, use the **`malloc`** function ("memory allocate") and specify the number of bytes you'd like.

- This function returns a pointer to *the **starting address** of the new memory*.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns NULL, then there wasn't enough memory for this request.

The Heap

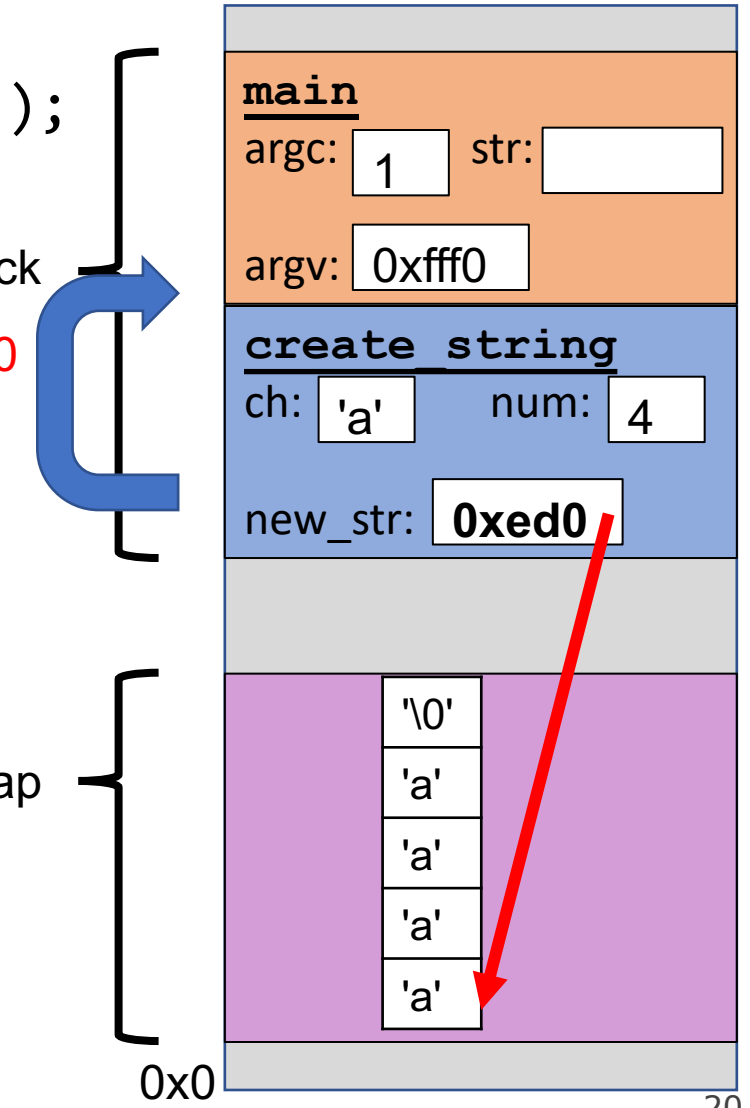
```
1 char *create_string(char ch, int num) {
2     char *new_str = malloc(sizeof(char) * (num + 1));
3     assert(new_str != NULL);
4     for (int i = 0; i < num; i++) {
5         new_str[i] = ch;
6     }
7     new_str[num] = '\0';
8     return new_str;
9 }

1 int main(int argc, char *argv[]) {
2     char *str = create_string('a', 4);
3     printf("%s", str);    // want "aaaa”
4     ...                   // free str memory
5     return 0;
6 }
```



The Heap

```
1  char *create_string(char ch, int num) {
2      char *new_str = malloc(sizeof(char) * (num + 1));
3      assert(new_str != NULL);
4      for (int i = 0; i < num; i++) {
5          new_str[i] = ch;
6      }
7      new_str[num] = '\0';
8      return new_str;
9  }
```

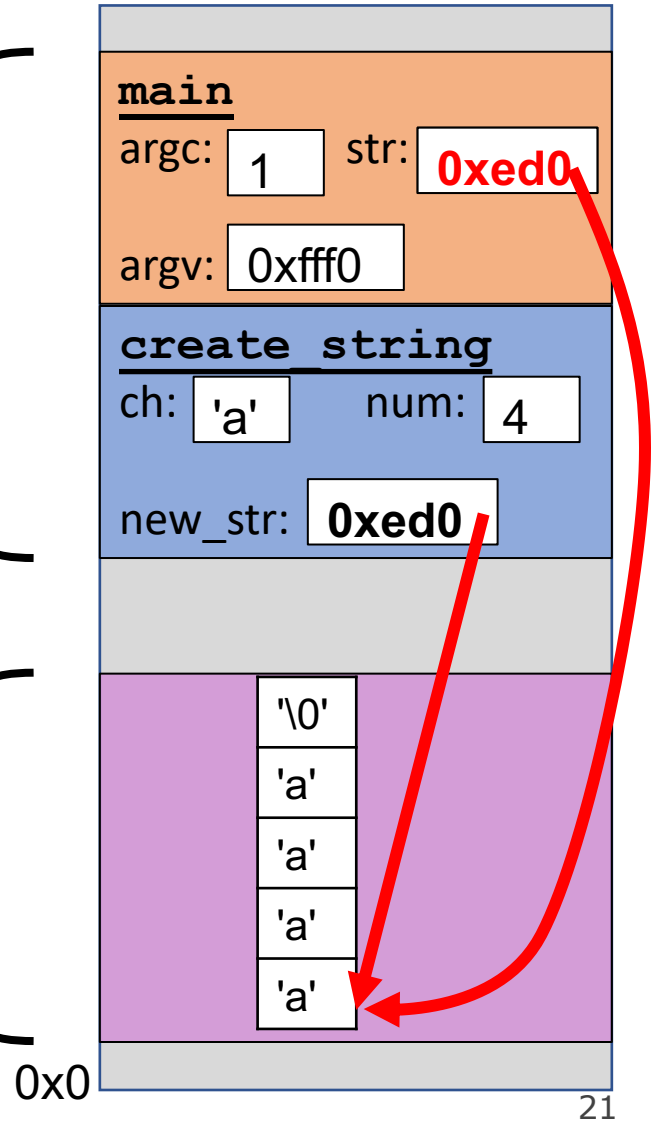
```
1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);    // want "aaaa”
4      ...                   // free str memory
5      return 0;
6  }
```



The Heap

```
1  char *create_string(char ch, int num) {
2      char *new_str = malloc(sizeof(char) * (num + 1));
3      assert(new_str != NULL);
4      for (int i = 0; i < num; i++) {
5          new_str[i] = ch;
6      }
7      new_str[num] = '\0';
8      return new_str;
9  }
```

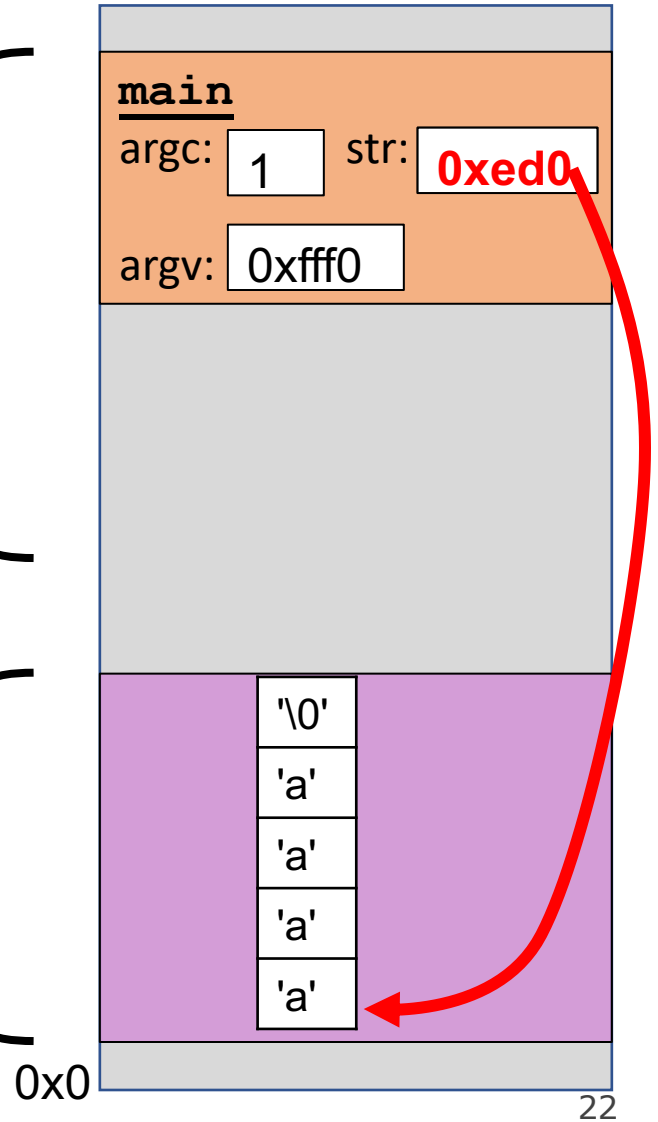
```
1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);    // want "aaaa”
4      ...                   // free str memory
5      return 0;
6  }
```



The Heap

```
1  char *create_string(char ch, int num) {
2      char *new_str = malloc(sizeof(char) * (num + 1));
3      assert(new_str != NULL);
4      for (int i = 0; i < num; i++) {
5          new_str[i] = ch;
6      }
7      new_str[num] = '\0';
8      return new_str;
9  }
```

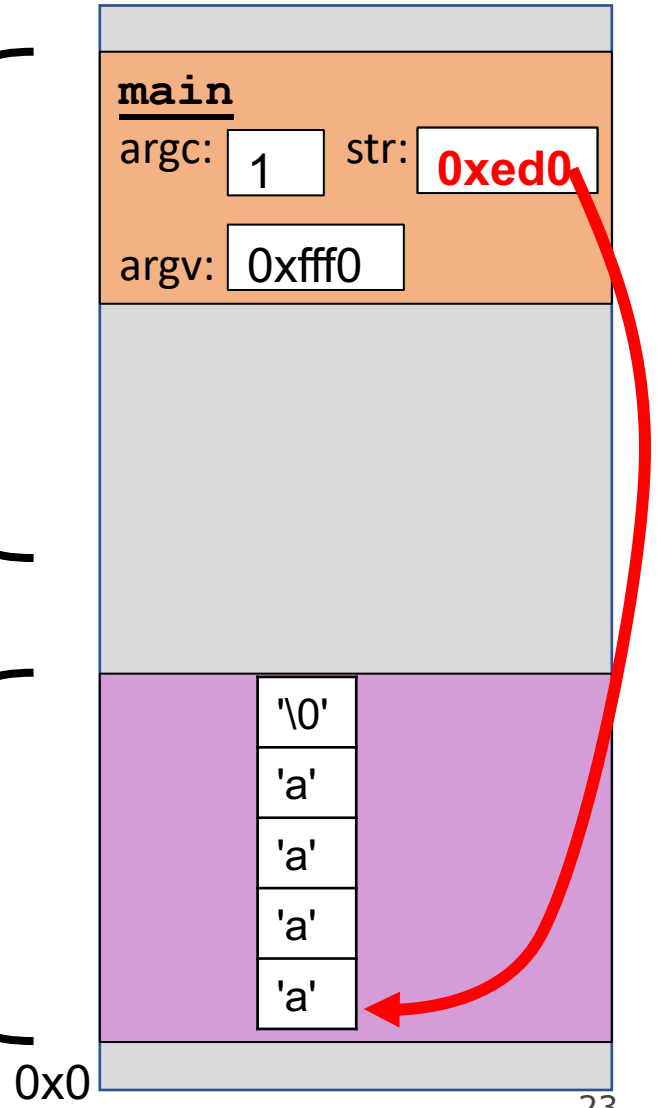
```
1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);    // want "aaaa”
4      ...                   // free str memory
5      return 0;
6  }
```



The Heap

```
1  char *create_string(char ch, int num) {
2      char *new_str = malloc(sizeof(char) * (num + 1));
3      assert(new_str != NULL);
4      for (int i = 0; i < num; i++) {
5          new_str[i] = ch;
6      }
7      new_str[num] = '\0';
8      return new_str;
9  }
```

```
1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);    // want "aaaa”
4      ...                   // free str memory
5      return 0;
6  }
```



Exercise: `malloc` multiples

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

```
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

Line 2: How should we declare `arr`?

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

Exercise: `malloc` multiples

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

```
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- Use a pointer to store the address returned by malloc.
- Malloc's argument is the **number of bytes** to allocate.

⚠ **This code is missing an assertion.**

Line 2: How should we declare `arr`?

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

Always assert with the heap

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

```
1 int *array_of_multiples(int mult, int len) {
2     int *arr = malloc(sizeof(int) * len);
3     assert(arr != NULL);
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- If an allocation error occurs (e.g. out of heap memory!), malloc will return NULL. This is an important case to check **for robustness**.
- **assert** will crash the program if the provided condition is false. A memory allocation error is significant and we should terminate the program.

Other heap allocations: `calloc`

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

`calloc` is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- calloc is more expensive than malloc because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

```
char *str = strdup("Hello, world!");  // on heap
str[0] = 'h';
```

Cleaning Up with `free`

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

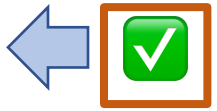
- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

free details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

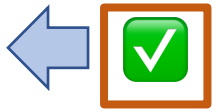
```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);
…
free(ptr);
```

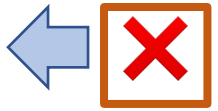


❌ Memory at this address was already freed!

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);
…
free(ptr + 1);
```





Memory Leaks

A memory leak is when you allocate memory on the heap, but you fail to free it.

- Your program should be responsible for cleaning up any memory that it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!

However, memory leaks rarely (if ever) cause crashes.

- For your own programs, focus first on your logic and implementation, then ensure that you free memory as appropriate.
- **Valgrind** is a very helpful tool for finding memory leaks!

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc`—the previous (smaller) one was already reclaimed by `realloc`.

```
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Heap allocator analogy: A hotel

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, bellman moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you…

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes…
- Return to room after checkout?

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

Announcements

- We will send out exam accommodation emails this week (OAE, university athletics)
- Assignment 1 grades released later today
- Assignment 3 released tonight
  - Memory, pointers, and stack/heap
  - Emphasis on debugging

Debugging

1. **Observe** the bug.
2. Create a **simple**, **reproducible** input.
3. **Narrow** the search space.
4. **Analyze** using GDB and pictures.
5. **Devise and run experiments** until you identify the root cause.
6. **Modify** code to squash bug.

Starting with this assignment, we will only be able to help you with debugging in helper hours if you fill out the signup form with information from these steps!

Joke break

Code golf

From Wikipedia, the free encyclopedia

Code golf is a type of recreational computer programming competition in which participants strive to achieve the shortest possible source code that implements a certain algorithm.


```
a=5;while(*p++){if(p[a])--a;++b;}
```

```
for(a=5;*p++;++b)if(p[a])--a;
```

You should not be code golfing with your CS107 assignment grade ☺

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Compare and contrast the heap memory functions we've learned today.

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- NULL on failure, so check with `assert`
- Memory is contiguous; it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location.
- If you `realloc`/`free` non-heap address

Engineering principles: stack vs heap

Stack ("local variables")

Heap (dynamic memory)

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient**.
  Automatic allocation/ deallocation;
  declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠ **Somewhat inflexible**
Cannot add/resize at runtime, scope
dictated by control flow in/out of functions

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient**.
  Automatic allocation/ deallocation; declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful**.
  Can provide more memory on demand!
- **Very flexible.**
  Runtime decisions about how much/when to allocate, can resize easily with realloc
- **Scope under programmer control**
  Can precisely determine lifetime

⚠ **Lots of opportunity for error**
Low type safety, forget to allocate/free before done, allocate wrong size, etc., Memory leaks (much less critical)

pig_heap
(Pig Latin redux)

`pig_heap.c`

pig_cat
(Pig Latin concatenation)

`pig_heap.c`

An exercise in using `realloc`
You will study this code in lab3

Plan For Today

- The Stack
- The Heap and Dynamic Memory
- Announcements
- **Practice:** Pig Latin
- Miscellaneous topics: **const**, **struct** and ternary operator

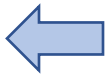
const

Use `const` with pointers to indicate that the *data* pointed to cannot change.

```
char str[6] = "Hello";
const char *s = str;
s[0] = 'h';     // ❌ Cannot use s to change characters it points to
s = &str[2];    // ✅ You can modify the actual pointer itself
```

C complains if you set a **non-**`const` pointer equal to a `const` pointer:

```
int binky(const char *str) {
    char *modifiable_str = str;
    ...
}
```



warning: initialization discards '**const**' qualifier from pointer target type

Const

`const` can be confusing to interpret in some variable types. General rule of thumb is that `const` restricts the type immediately following it.

```
// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = …

// cannot modify chars pointed to by *strPtr
const char **p_str = …
```

For CS107, your goal is to be able to use `const` variables when they are declared for you.

Structs

The ***struct*** keyword defines a new variable type as a group of other variables.

declaration

```
struct date {
    int month;          // members of each
    int day;            // date structure
};
```

instantiation

1.
```
struct date today;
today.month = 1;
today.day = 27;
```

2.
```
struct date new_years_eve = {12, 31};
```

Structs

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

declaration

```
typedef struct date {
    int month;
    int day;
} date;
```

instantiation

1.
```
date today;
today.month = 1;
today.day = 27;
```

2.
```
date new_years_eve = {12, 31};
```

Structs are also pass-by-value

⚠️ Like other parameter types, a **copy** of the entire struct gets passed in for function calls.

```
void advance_day(date d) {
    d.day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 27};
    advance_day(my_date);
    printf("%d", my_date.day);
    // 27
    return 0;
}
```

my_date is **unchanged**

```
void advance_day(date *d) {
    (*d).day++;
    // equivalent: d->day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 27};
    advance_day(&my_date);
    printf("%d", my_date.day);
    // 28
    return 0;
}
```

You can use the arrow operator -> on struct pointers.

Structs work as expected

```
typedef struct date {
    int month;
    int day;
} date;

date lunar_newyear() {
    date d = {1, 25};
    return d;          // or (date) {1, 25};
}

int main(int argc, char *argv[]) {
    date my_date = lunar_newyear();
    printf("%d", my_date.day);   // 25
    int size = sizeof(date);     // 8
    return 0;
}
```

You can return structs from functions

`sizeof` a struct == sum of the `sizeof`s of its members

Arrays of structs also work as expected

```
typedef struct my_struct {
      int x;
      char c;
} my_struct;

int main(int argc, char *argv[]) {
    my_struct array_of_structs[5];

    array_of_structs[0] = (my_struct){0, 'A'};

    array_of_structs[1].x = 2;
    array_of_structs[1].c = 'B';
}
```

Need the cast to confirm struct type to C

Array indexing works too. Why don't we use -> here?

Ternary Operator

The ternary operator is a shorthand for using if/else to evaluate to a value.

`condition ? expressionIfTrue : expressionIfFalse`

```
char *param;
if (argc > 1) {
      param = argv[1];
} else {
      param = "default";
}

// equivalent to
char *param = (argc > 1) ? argv[1] : "default";
```

⚠ It is bad style to stuff complex/ magic conditions into the ternary operator, so use this wisely.