COMP201

Computer Systems & Programming

Lecture #9 – `realloc`, Memory Bugs

Aykut Erdem // Koç University // Fall 2025

KOÇ
UNIVERSITY

Recap: Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

This stores 5 **char *s**, *not* all of the characters for 5 strings!

```
char *str0 = stringArray[0];    // first char *
```

Recap: Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
char *str = "apple"; // e.g. 0xff0
char *str1 = str + 1;    // e.g. 0xff1
char *str3 = str + 3;    // e.g. 0xff3


printf("%s", str);       // apple
printf("%s", str1); // pple
printf("%s", str3); // le
```

DATA SEGMENT

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

Recap: Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …              // e.g. 0xff0
int *nums1 = nums + 1;     // e.g. 0xff4
int *nums3 = nums + 3;     // e.g. 0xffc

printf("%d", *nums);       // 52
printf("%d", *nums1);      // 23
printf("%d", *nums3);      // 34
```

STACK

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

4

Recap: Pointer Arithmetic

How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;


char str[6];
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Recap: Pointer Arithmetic

How does the code know how many bytes it should add when performing

At compile time, C can figure out the sizes of different data types, and the sizes of what they point to. Hence, when the program runs, it knows the correct number of bytes to address or add/ subtract for each data type

```
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Recap: Pointer arithmetic

Array indexing is "syntactic sugar" for pointer arithmetic:

$$\texttt{ptr + i} \qquad \Leftrightarrow \qquad \texttt{\&ptr[i]}$$

$$\texttt{*(ptr + i)} \qquad \Leftrightarrow \qquad \texttt{ptr[i]}$$

⚠️ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)`.
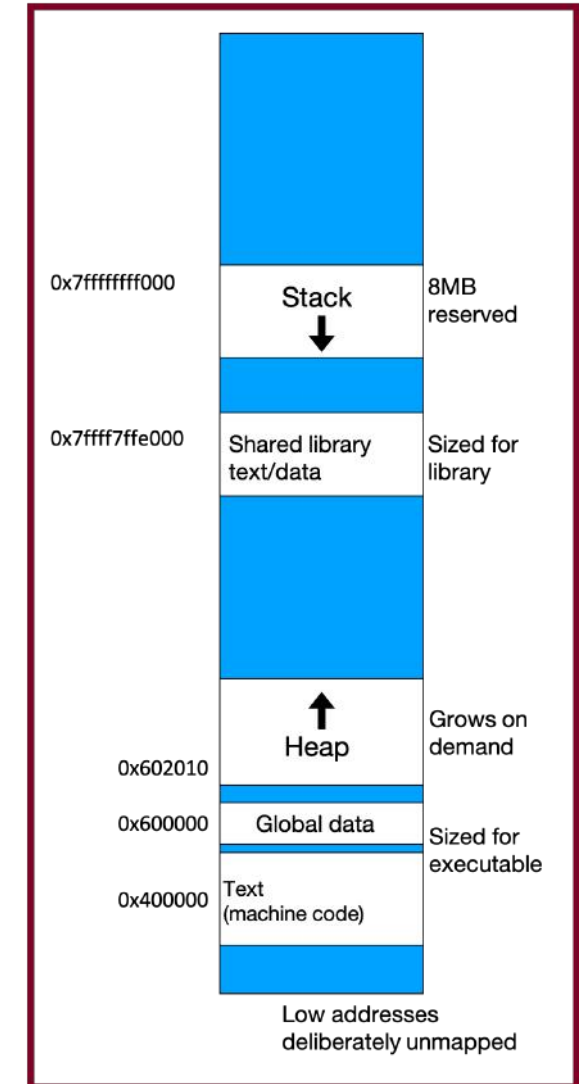
- This means too-large/negative subscripts will compile ☺

  `arr[99]`   `arr[-1]`

- You can use either syntax on either pointer or array.
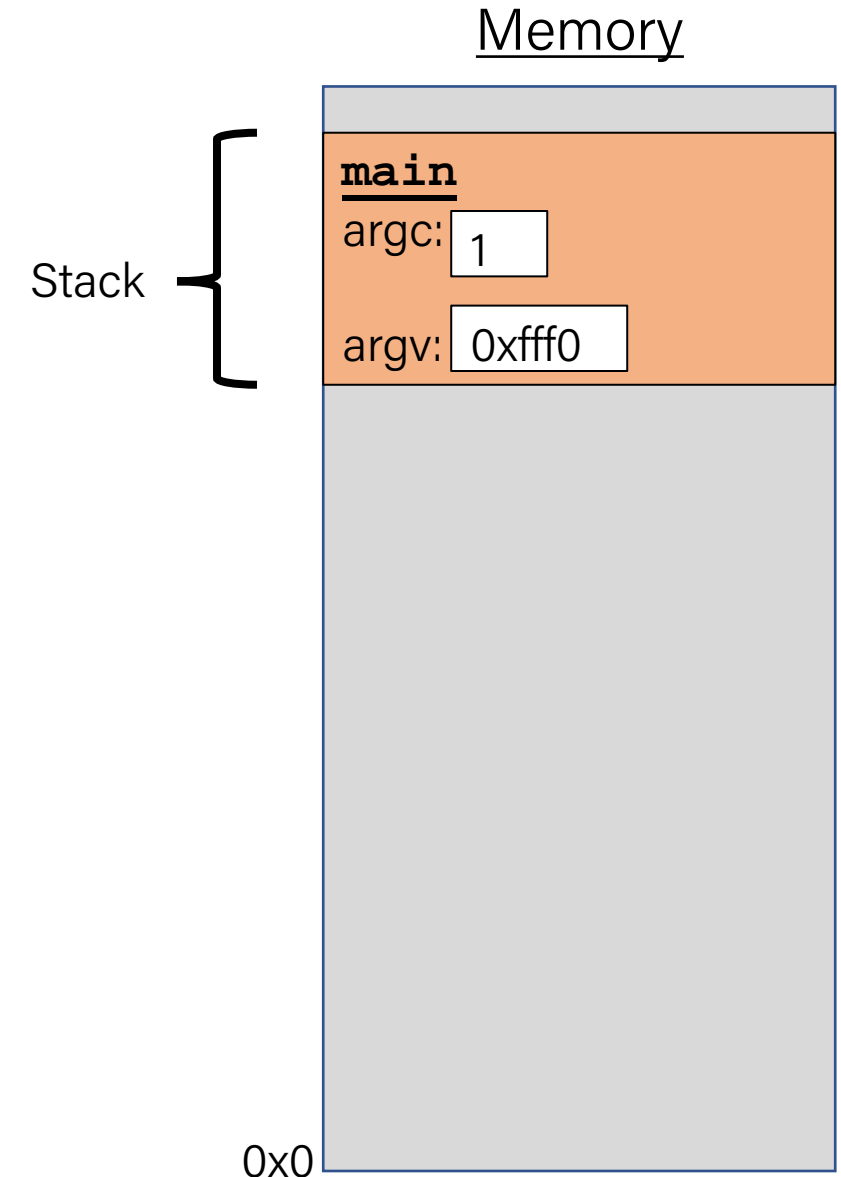
Recap: The Stack

- We are going to dive deeper into different areas of memory used by our programs.

- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.

- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.

8

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```
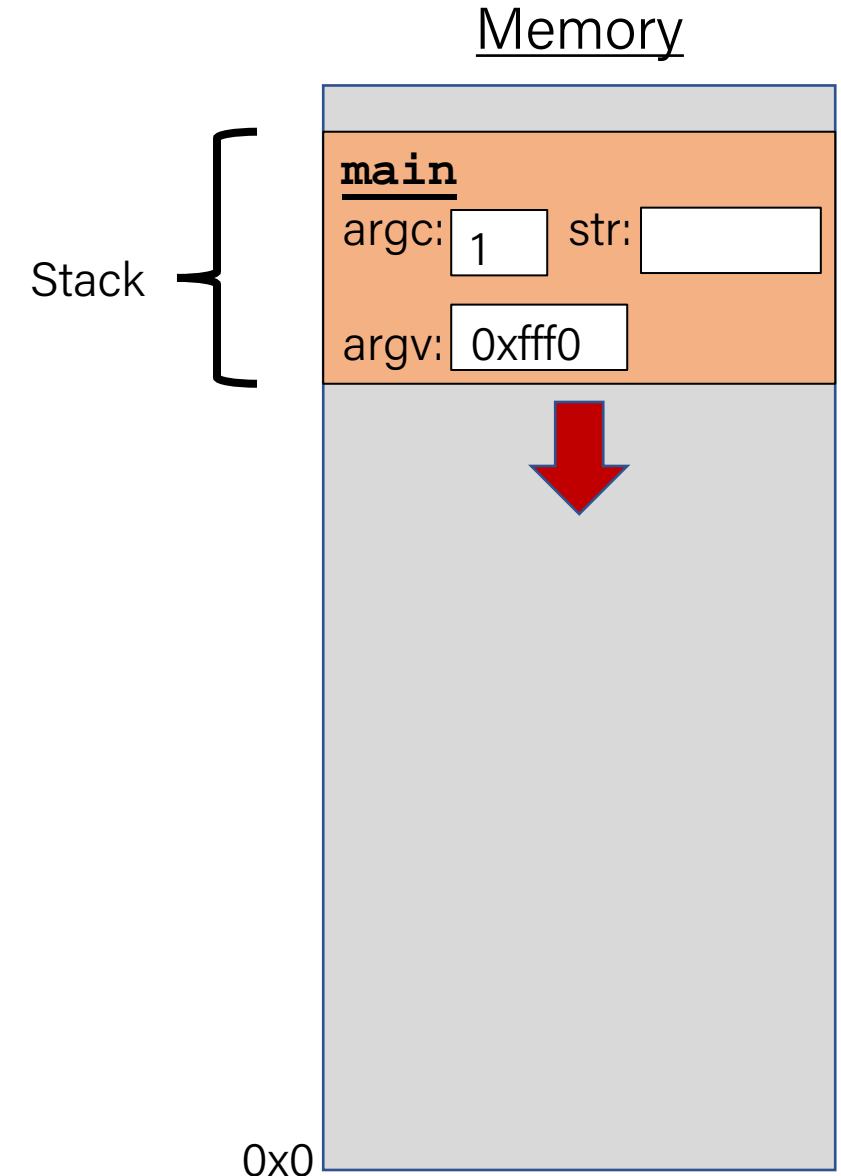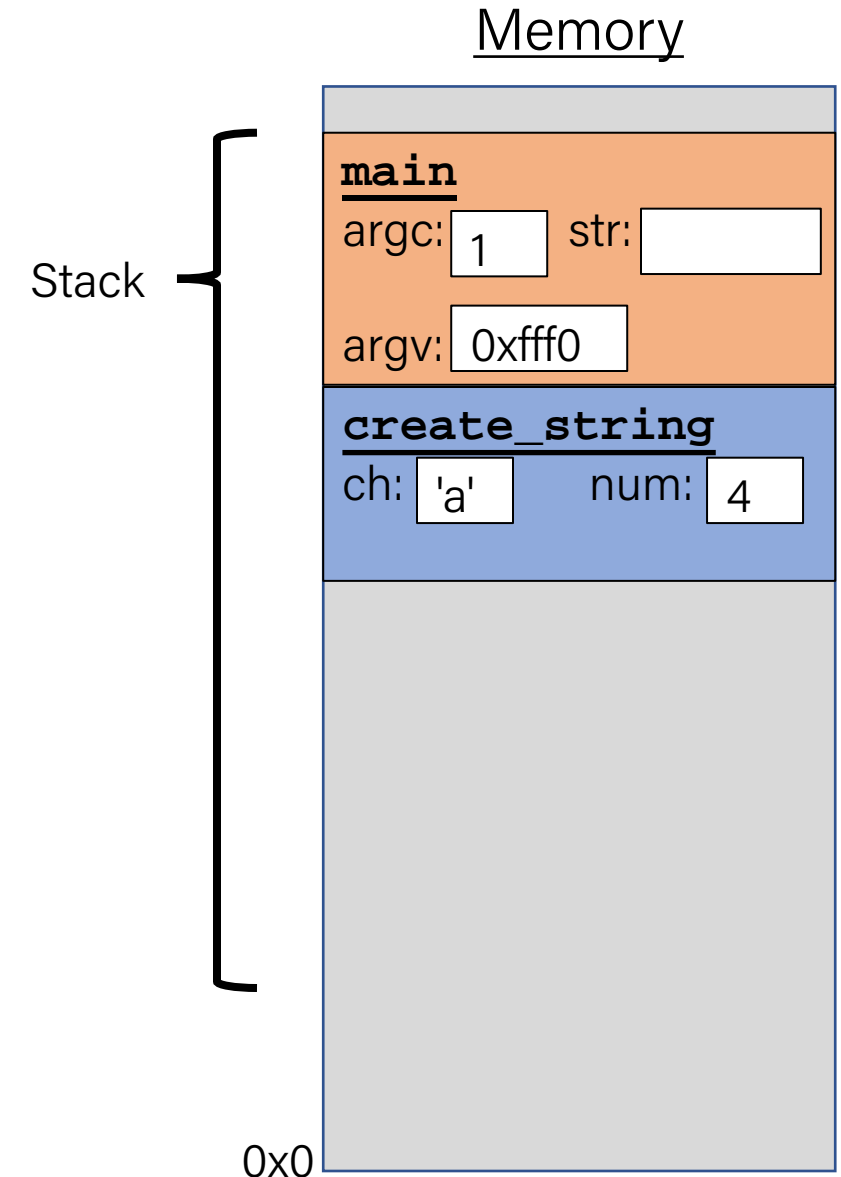
Memory

Stack

main
argc: 1

argv: 0xfff0

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```

Memory

Stack

main

argc: 1   str:
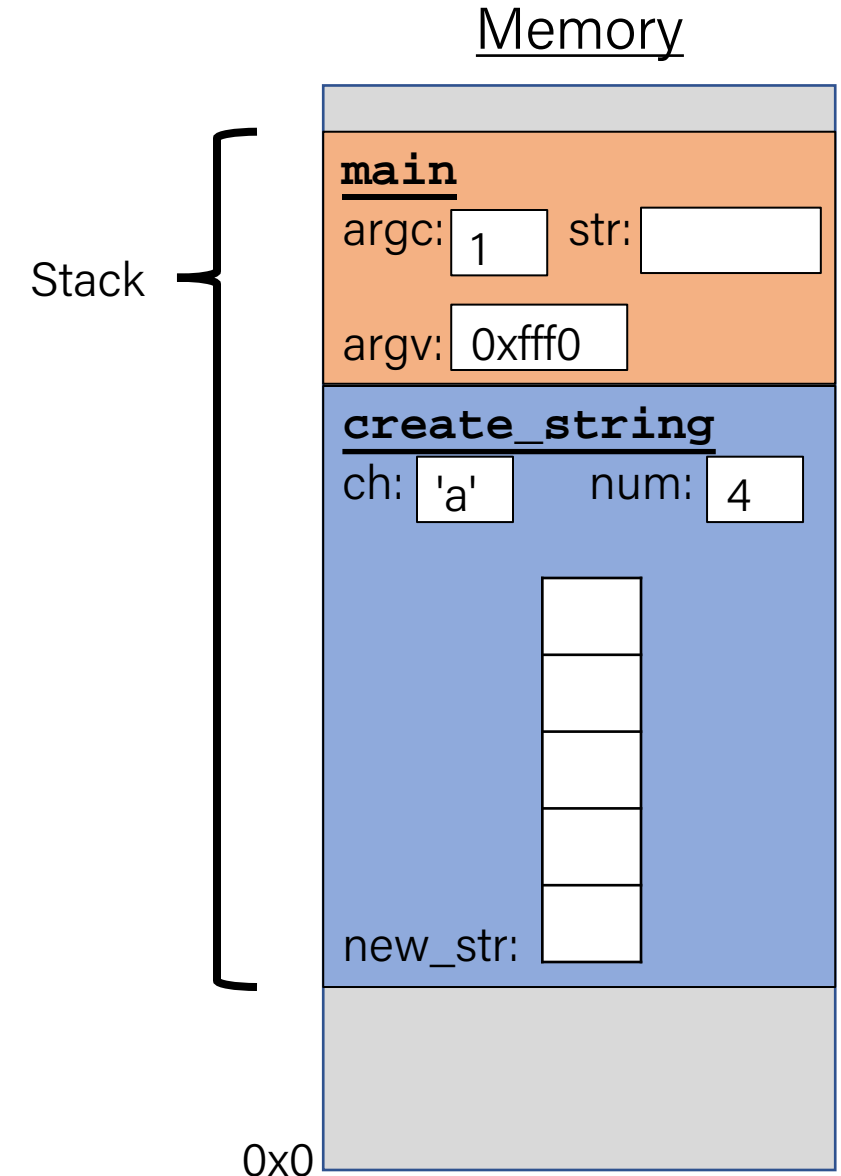
argv: 0xfff0

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```
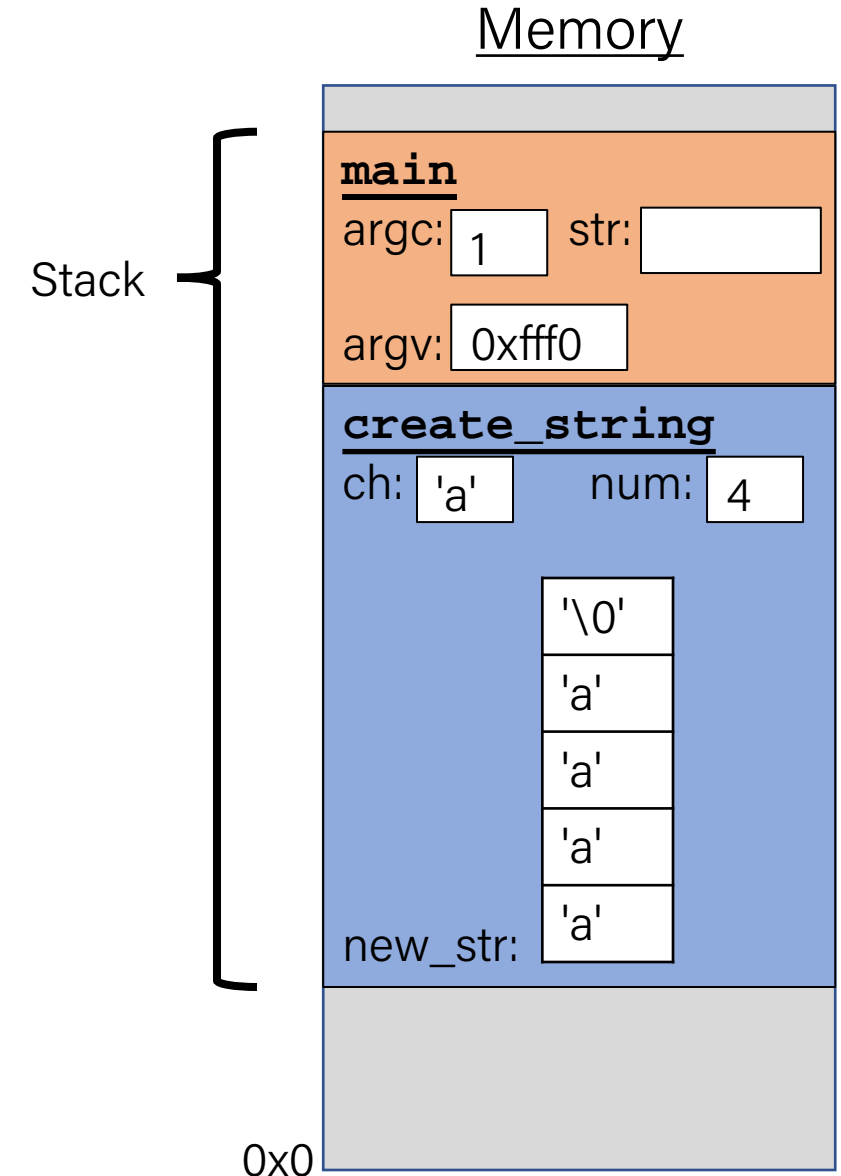
Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```

Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```
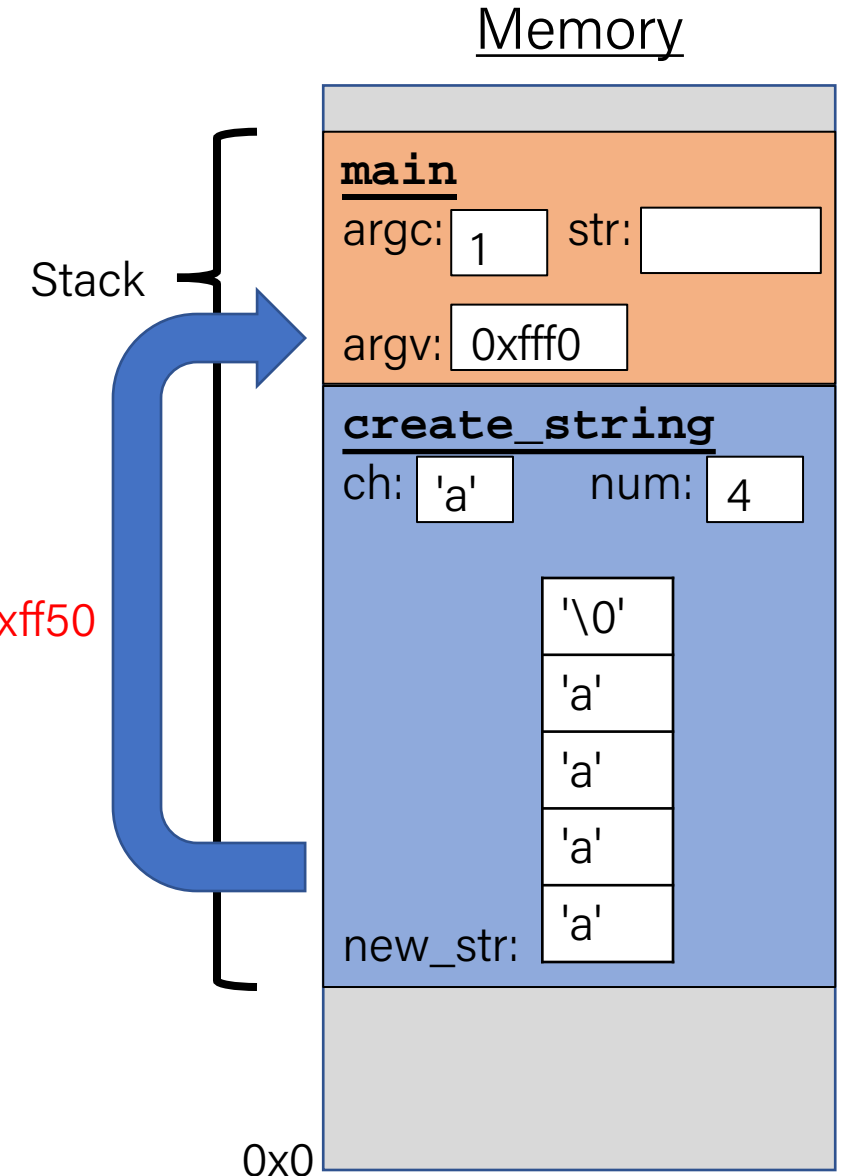
Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:
'\0'
'a'
'a'
'a'
'a'

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```
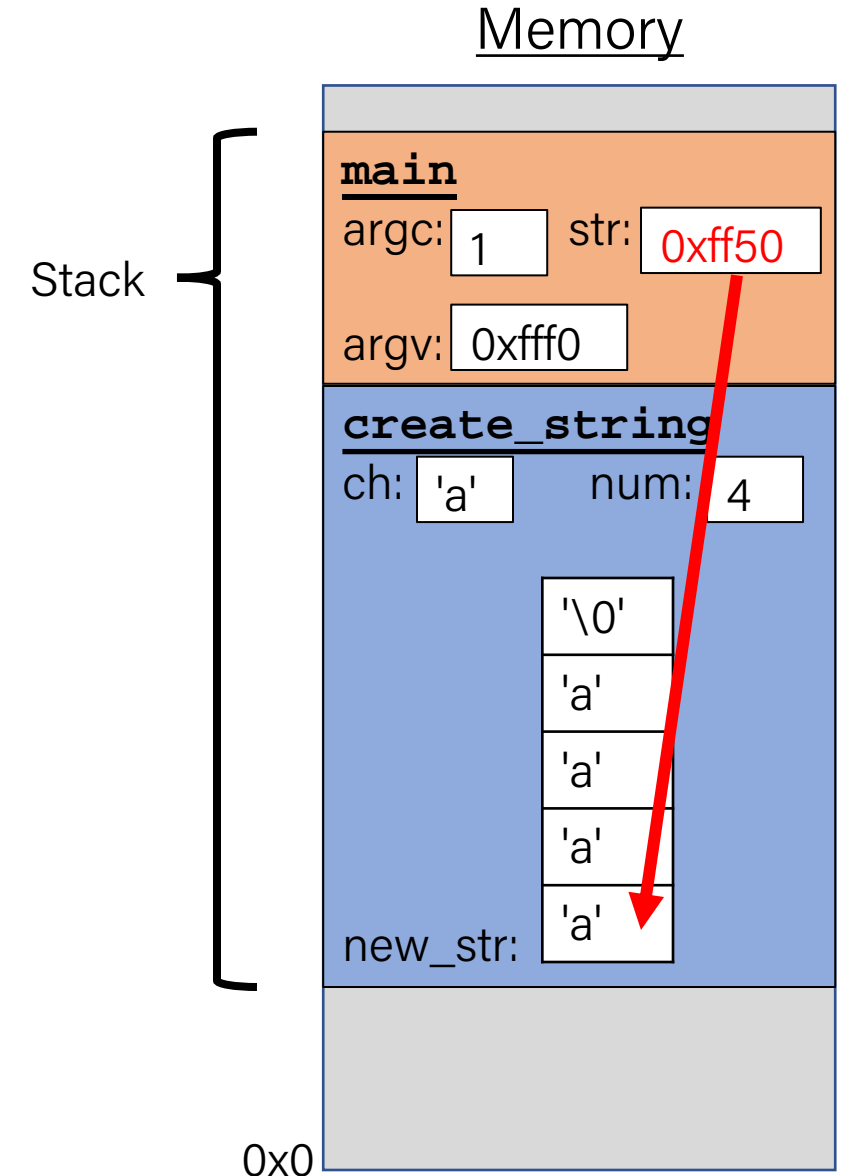
Returns e.g. 0xff50

Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:
| '\0' |
| 'a' |
| 'a' |
| 'a' |
| 'a' |

0x0

Recap: The Stack

```c
char *create_string(char ch, int num) {
    char new_str[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;
}
```
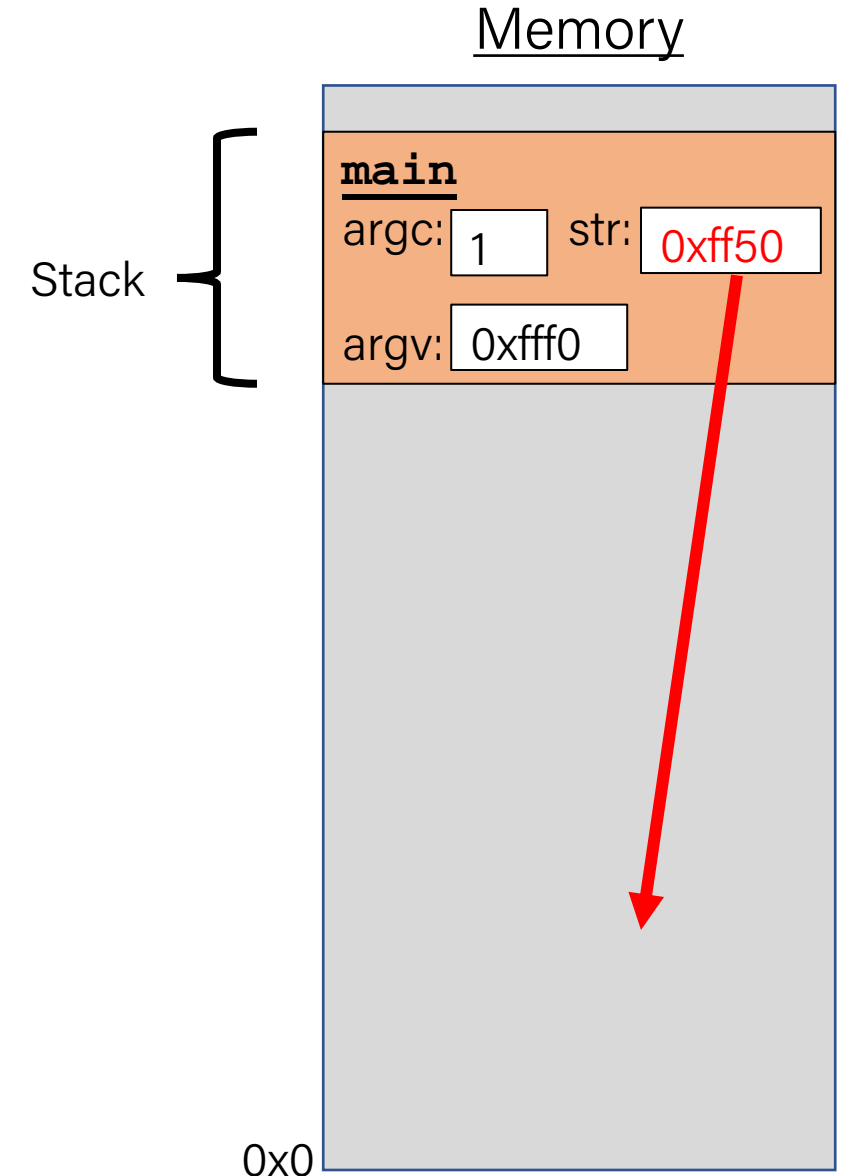
Memory

Stack

main

argc: 1   str: 0xff50

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'

'a'

'a'

'a'

new_str: 'a'

0x0

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[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;
}
```
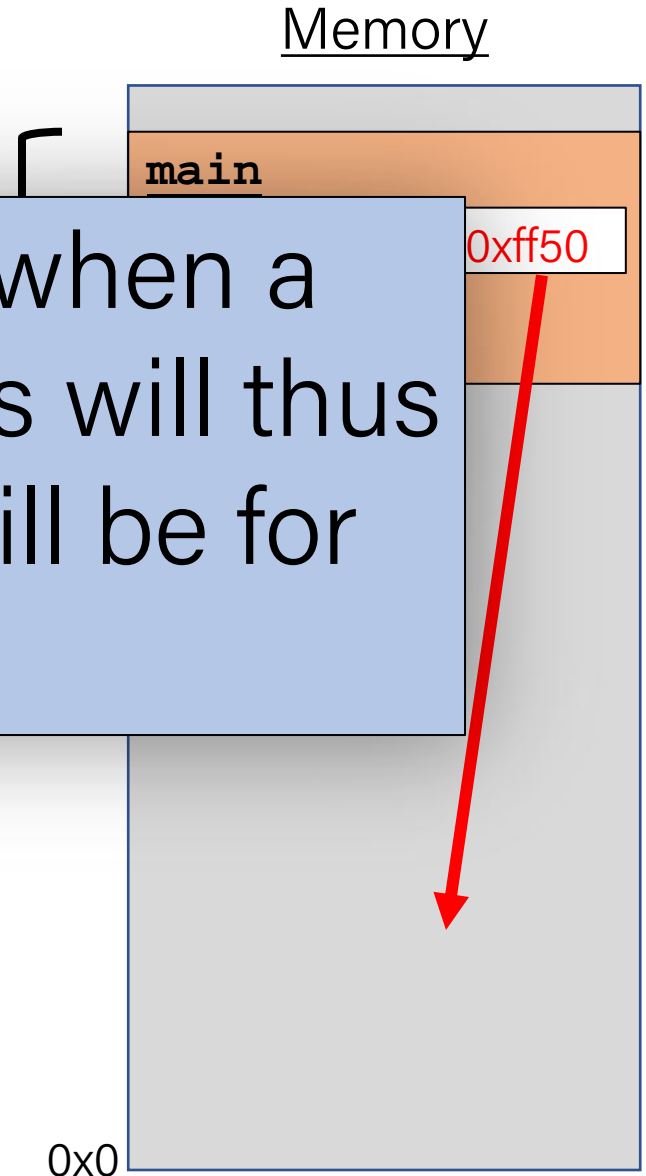
Memory

Stack

main

argc: 1   str: 0xff50

argv: 0xfff0

0x0

Recap: The Stack

```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
```

Problem: local variables go away when a function finishes. These characters will thus no longer exist, and the address will be for unknown memory!

```
}

int m
```
```
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

main

0xff50

0x0

Recap: 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**.
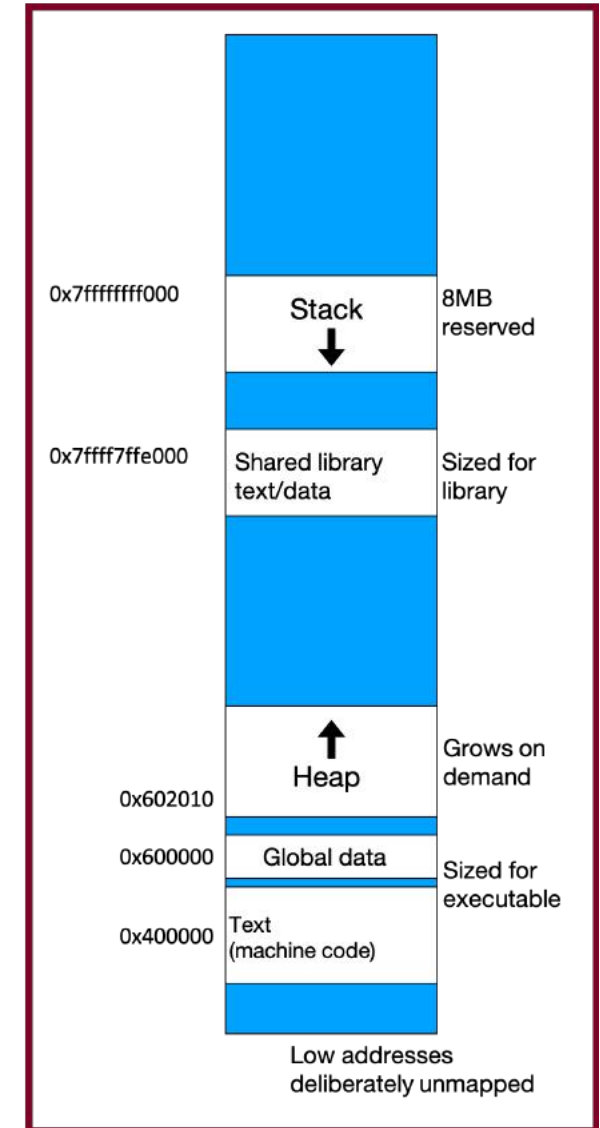
Recap: `malloc`

```
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.* It doesn't know or care whether it will be used as an array, a single block of memory, etc.

- **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.

Recap: The Heap

```c
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;
}
```
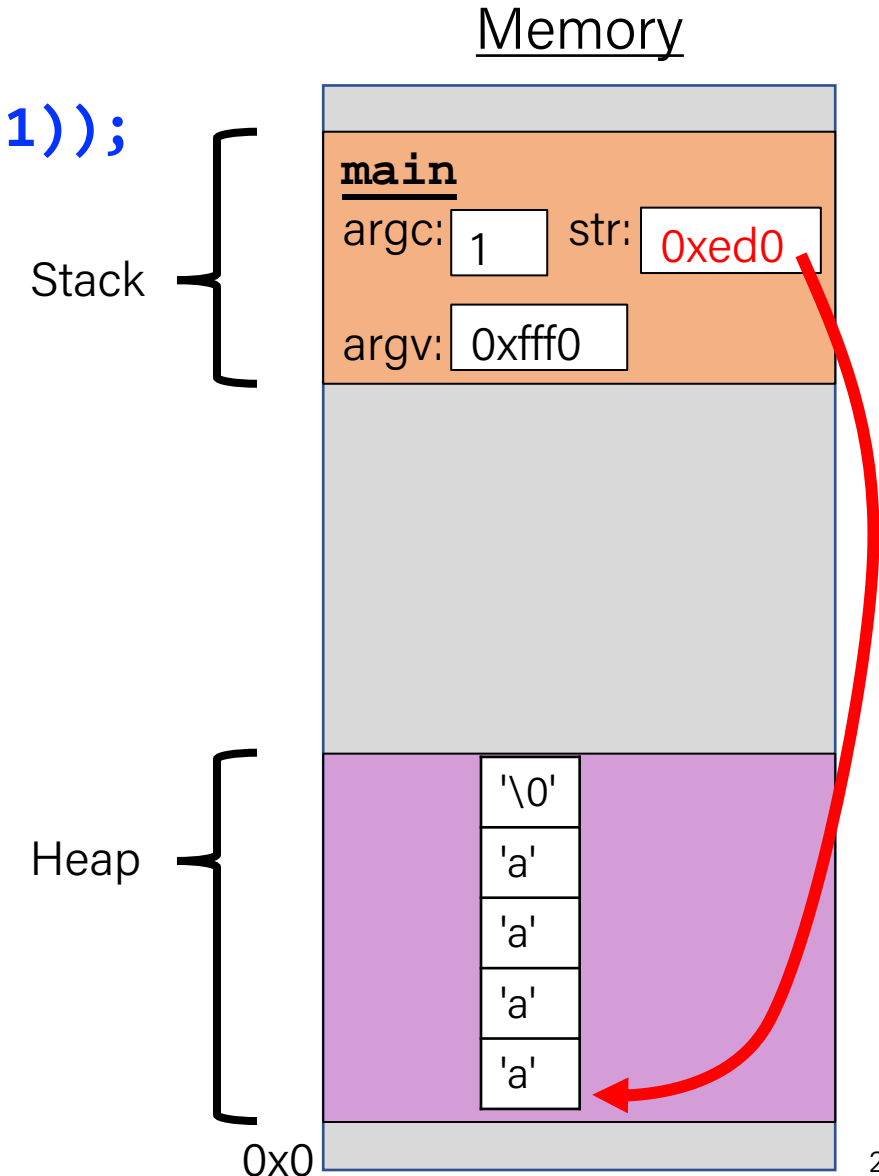
Returns e.g. 0xed0

Recap: The Heap

```c
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;
}
```


Memory

Stack

main

argc: 1   str: 0xed0

argv: 0xfff0

Heap

'\0'
'a'
'a'
'a'
'a'
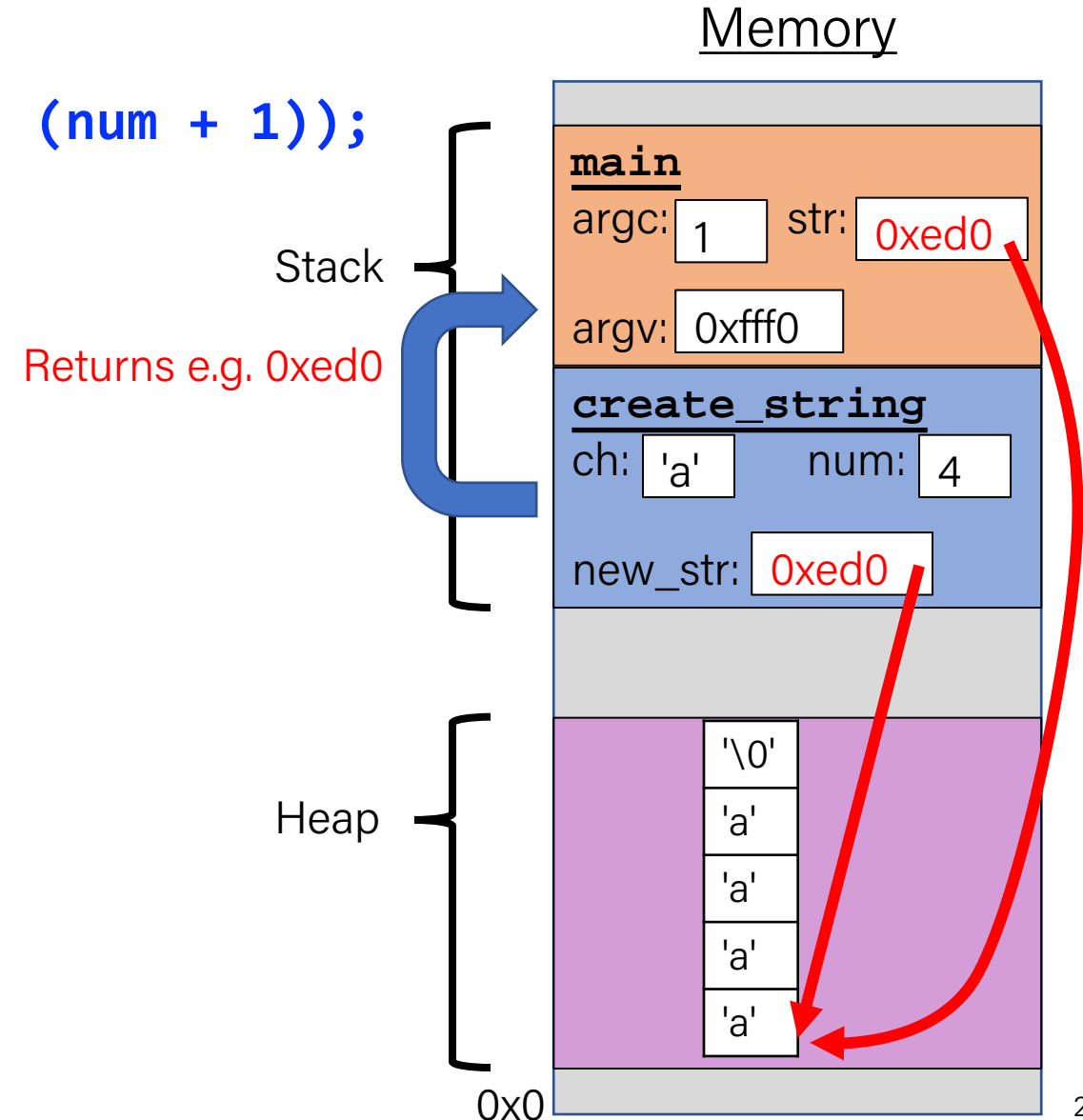
0x0

Recap: 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.

Plan for Today

- Other heap allocations
- `free`
- Practice: Pig Latin
- `realloc`
- Memory bugs

Plan for Today

- Other heap allocations

- free

- Practice: Pig Latin

- realloc

- Memory bugs

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';
```

Implementing strdup

How can we implement **strdup** using functions we've already seen?

```
char *myStrdup(char *str) {
    char *heapStr = malloc(strlen(str) + 1);
    assert(heapStr != NULL);
    strcpy(heapStr, str);
    return heapStr;
}
```

Plan for Today

- **free**

- Practice: Pig Latin

- realloc

- Memory bugs

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);
```

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 *str = strdup("Hello!");

…

free(str);     // our responsibility to free!
```

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);  ⬅ ❌
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");

…

free(str);     // our responsibility to free!
```

Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.

- Your program should be responsible for cleaning up any memory 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.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.

- Valgrind is a very helpful tool for finding memory leaks!

free Practice

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
1     char *str = strdup("Hello");
2     assert(str != NULL);
3     char *ptr = str + 1;
4     for (int i = 0; i < 5; i++) {
5          int *num = malloc(sizeof(int));
6           assert(num != NULL);
7          *num = i;
8          printf("%s %d\n", ptr, *num);
9     }
10     printf("%s\n", str);
```

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
1    char *str = strdup("Hello");
2    assert(str != NULL);
3    char *ptr = str + 1;
4    for (int i = 0; i < 5; i++) {
5        int *num = malloc(sizeof(int));
6        assert(num != NULL);
7        *num = i;
8        printf("%s %d\n", ptr, *num);
9        free(num);
10   }
11   printf("%s\n", str);
12   free(str);
```

Lecture Plan

- free

- Practice: Pig Latin

- realloc

- Memory bugs

Demo: Pig Latin

pig_latin.c

Lecture Plan

- free

- Practice: Pig Latin

- `realloc`

- Memory bugs

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*.

realloc

```c
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);
```

realloc

- **realloc** only accepts pointers that were previously returned my **malloc**/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

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`.

```c
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, employee 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?

Demo: Pig Latin Part 2

`pig_latin.c`

Heap allocation interface: A summary

```c
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 about.

🤔

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")

- **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)

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)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.

- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Lecture Plan

- free

- Practice: Pig Latin

- realloc

- Memory bugs

Pointers and Working with Dynamic Memory

Here are some common errors and mistakes that may happen if you're not careful enough:

- storage used after free,
- allocation freed repeatedly,
- insufficient space for a dynamically allocated variable,
- freeing unallocated storage,
- freeing of the stack space,
- memory leakage,
- assignment of incompatible types,

- returning (directly or via an argument) of a pointer to a local variable,
- dereference of wrong type,
- dereference of uninitialized or invalid pointer,
- incorrect use of pointer arithmetic,
- array index out of bounds

54

Exercise 1

```c
void myfunc(int *arr) {
  int *p_arr = (int*) malloc(2*sizeof(int));
  p_arr[0] = 42;
  p_arr[1] = 24;
  arr = p_arr;
}

int main(int argc, char *argv[]) {
  int *arr = NULL;
  myfunc(arr);
  printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
  free(arr);
  return 0;
}
```

Exercise 1

```c
void myfunc(int *arr) {
    int *p_arr = (int*) malloc(2*sizeof(int));
    p_arr[0] = 42;
    p_arr[1] = 24;
    arr = p_arr;
}

int main(int argc, char *argv[]) {
    int *arr = NULL;
    myfunc(arr);
    printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
    free(arr);
    return 0;
}
```


main

argc

argv

NULL arr

myfunc

0x1c

NULL arr

0x1c p_arr

0x1c
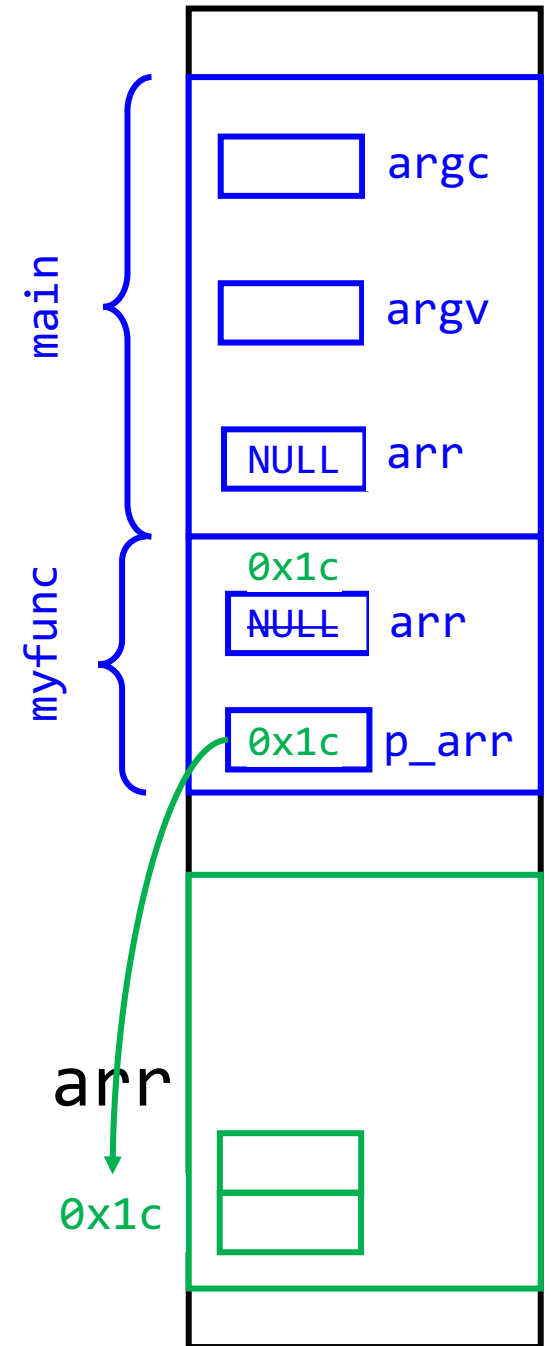
Exercise 1

```c
void myfunc(int *arr) {
    int *p_arr = (int*) malloc(2*sizeof(int));
    p_arr[0] = 42;
    p_arr[1] = 24;
    arr = p_arr;
}

int main(int argc, char *argv[]) {
    int *arr = NULL;
    myfunc(arr);
    printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
    free(arr);
    return 0;
}
```

1. dereference of uninitialized or invalid pointer: `arr` in `main` is still `NULL`

Exercise 1

```c
void myfunc(int *arr) {
  int *p_arr = (int*) malloc(2*sizeof(int));
  p_arr[0] = 42;
  p_arr[1] = 24;
  arr = p_arr;
}

int main(int argc, char *argv[]) {
  int *arr = NULL;
  myfunc(arr);
  printf("arr[0] = %d\n arr[1] = %d", arr[0], arr[1]);
  free(arr);
  return 0;
}
```

2. freeing unallocated storage!

Exercise 2

```c
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}

int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

Exercise 2

```
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}
```

1. insufficient space for a dynamically allocated variable: `malloc` should use `sizeof(int*)`

```
int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

Exercise 2

```c
int myfunc(int **array, n) {
  int** int_array = (int**) malloc(n*sizeof(int));
  array = int_array;
  return 0;
}

int main(int argc, char *argv[]) {
  int **array = NULL;
  myfunc(array,10);
  array[0] = (int*) malloc(4*sizeof(int));
  return 0;
}
```

2. dereference of uninitialized or invalid
   pointer: `array` in `main` is still `NULL`

Exercise 3

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

    char *param1 = *argv[1];
    char *param2 = *argv[2];
    char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != '\0')
      ;

  strcat(ptr+strlen(param1)+1, param2);
  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

Exercise 3

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

    char *param1 = *argv[1];
    char *param2 = *argv[2];
    char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != '\0')
      ;

strcat(ptr+strlen(param1)+1, param2);
printf("%s\n", ptr);
ptr = NULL;
return 0;
}
```

1. Dereference of invalid pointer: `strcat` could not find end of `dest`

Exercise 3

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);

  while ((*ptr++ = *param1++) != '\0')
    ;

  strcat(ptr+strlen(param1)+1, param2);
  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

2. memory leakage: `ptr = NULL;` should be `free(ptr);`

Exercise 4

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(param1);

  while ((*ptr++ = *param2++) != '\0')
    ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

Exercise 4

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

    char *param1 = *argv[1];
    char *param2 = *argv[2];
    char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(param1);

  while ((*ptr++ = *param2++) != '\0')
     ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

1. memory leakage: `ptr = NULL;` should be `free(ptr);`

Exercise 4

```c
int main(int argc, char *argv[]) {
  if (argc!=3) {printf("wrong number of arguments\n"); return 1;}

  char *param1 = *argv[1];
  char *param2 = *argv[2];
  char *ptr;

  ptr = (char *) malloc(strlen(param1)+strlen(param2)+1);
  strcpy(ptr, param1);
  ptr += strlen(param1);

  while ((*ptr++ = *param2++)
    ;

  printf("%s\n", ptr);
  ptr = NULL;
  return 0;
}
```

2. memory leakage:
   `ptr+=strlen(param1);`
   no way to free memory originally
   pointed by `ptr`

Microsoft IT outage

CrowdStrike global outage to cost US Fortune 500 companies $5.4bn

Banking and healthcare firms, major airlines expected to suffer most losses, according to insurer Parametrix

Nick Robins-Early

Wed 24 Jul 2024 18.19 CEST

< Share

News | Opinion | S

World   UK   Climate crisis   Ukraine   Environme

Microsoft IT outage

Crowd

Fortu

Banking a
suffer mo

Nick Robins-Early

Wed 24 Jul 2024 18.19 CEST

< **Share**

Zach Vorhies / Google Whistleblower ✓
@Perpetualmaniac

Crowdstrike Analysis:

It was a NULL pointer from the memory unsafe C++ language.

Since I am a professional C++ programmer, let me decode this stack trace dump for you.

```
EXCEPTION_RECORD:  fffffb0d18d3ec28 -- (.exr 0xfffffb0d18d3ec28)
ExceptionAddress: fffff8021df335a1 (csagent+0x00000000000e35a1)
   ExceptionCode: c0000005 (Access violation)
   ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000000
   Parameter[1]: 000000000000009c
Attempt to read from address 000000000000009c

CONTEXT:  fffffb0d18d3e460 -- (.cxr 0xfffffb0d18d3e460)
rax=fffffb0d18d3f2b0 rbx=0000000000000000 rcx=0000000000000003
rdx=fffffb0d18d3f280 rsi=ffff9a81b596f9a4 rdi=ffff9a81b596605c
rip=fffff8021df335a1 rsp=fffffb0d18d3ee60 rbp=fffffb0d18d3ef60
 r8=000000000000009c  r9=0000000000000000 r10=0000000000000000
r11=0000000000000014 r12=fffffb0d18d3ef28 r13=fffffb0d18d3f0d0
r14=000000000000001a r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b           efl=00050206
csagent+0xe35a1:
fffff802`1df335a1 458b08          mov     r9d,dword ptr [r8] ds:002b:00000000`0000009c=????????
Resetting default scope

BLACKBOXBSD: 1 (!blackboxbsd)

BLACKBOXNTFS: 1 (!blackboxntfs)

BLACKBOXPNP: 1 (!blackboxpnp)

BLACKBOXWINLOGON: 1

PROCESS_NAME:  System

READ_ADDRESS:   000000000000009c

ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%p referenced memory at 0x%p. The memory could not

EXCEPTION_CODE_STR:  c0000005

EXCEPTION_PARAMETER1:  0000000000000000

EXCEPTION_PARAMETER2:  000000000000009c

EXCEPTION_STR:  0xc0000005

STACK_TEXT:
fffffb0d`18d3ee60 fffff802`1df09152  : 00000000`00000000 00000000`e01f008d fffffb0d`18d3f202 fffff802`1e
fffffb0d`18d3f000 fffff802`1df0a3e9  : 00000000`00000000 00000010`00000000 fffff9a81`b5
fffffb0d`18d3f130 fffff802`1e14954f  : 00000000`00000000 00000000`00000000 00000000`00
fffffb0d`18d3f260 fffff802`1e145d9b  : ffff9a81`93735280 fffffb0d`18d3f5d0 00000000`00000000`00
fffffb0d`18d3f4d0 fffff802`1deb8fd0  : 00000000`000030f1 fffffb0d`18d3f790 ffff9a81`992cbb30 fffe409`b7
```

10:08 PM · Jul 19, 2024 · **34.7M** Views

Recap

- `free`
- Practice: Pig Latin
- `realloc`
- Memory bugs

Next time: *C Generics – `void *`*