



C Review Session

Zoom Logistics

- This presentation will be recorded, so please have your videos off
- If you have any questions, please post them in chat!
- We'll also try to leave some time for Q&A after the presentation for any other questions

Disclaimer

- These slides are not a comprehensive overview of everything you need from 61C to succeed in 162
 - Other concepts not mentioned will likely be brought up when relevant
- This class has A LOT of C coding
 - If you choose to take this class you are committing to that workload
 - Almost everyone taking this class is rusty in C but this review alone will not make you comfortable enough for this class

Other Resources

- [CS 162 Ladder](#)
 - Overview of C and 61C topics
- [CS 61C Resources Page](#)
 - C staff notes, GDB reference card
- [Python Tutor C](#)
 - Just like CS 61A's Python Tutor, but for C

Outline

- We'll do brief conceptual review on topics and then do a review section. We'll close with notes on Git and other tips.
- Topics List:
 - C Basic Syntax
 - C Pointers and Arrays
 - C Structs
 - C Memory Layout
 - Review
 - Git

C Basics

Types in C

- C is a statically typed language
 - Type is known at compile-time (instead of at run-time)
- All types are numerical or a composition of other types:
 - Ex. `int`, `char`, `struct`, `union`, `typedef`, pointer
- Every variable in C has a type:
 - `int i;` // Declares `i` to be an integer.
 - `int j = 5;` // Defines `j` to be an integer initialized to 5.
- All variables in C are just bytes under the hood with some length
 - Ex: `int32_t` is a 4 byte integer interpreted as a signed 2's complement number
- There is no built-in `bool`
 - Must import `stdbool.h` with `#include <stdbool.h>`

Basic Operations

- Adding (+), subtracting (-), dividing (/), multiplying (*).
- Pre-increment and post-increment for numeric types:
 - Ex: `int`, `long`, `char`, `size_t`

```
int i = 0;
printf("%d\n", ++i); // Increments and then returns the value of i.
printf("%d\n", i++); // Increments i but returns the value before increment.
printf("%d\n", i);
```

1

1

2

Multiple Declarations

- You can declare/define multiple variables in the same line.

```
int x, y;  
char c = 'a', d;  
float f, g = 3.0, h = 162.0;
```

Conditionals and Loops

- Change control flow/iterate. Basic syntax:

if-else if-else blocks:

```
int i = 5;
if (i == 5) {
    i = 6;
    printf("This is executed.\n");
} else if (i == 6) {
    printf("This is not
executed.\n");
} else {
    printf("Not executed either.\n");
}
if (i == 6) {
    printf("This will be executed,
however.\n");
}
```

For loops:

```
for (int i = 0; i <= 162; i++) {
    printf("%d\n", i);
}

// Not limited to integers!
for (char c = fgetc(infile); c !=
EOF; c = fgetc(infile)) {
    putchar(c);
}
```

While loops:

```
int i = 0;
while (i <= 162) {
    printf("%d\n", i);
    ++i;
}
```

Do-while loops:

```
int i = 0;
do {
    printf("%d\n", i);
    ++i;
} while (i <= 162);
```

Switch Statements

Switch statements are a nice way to do conditional logic based on the value of an expression:

```
char c = 'c';
printf("Make sure to break out of your
cases!\n");
switch (c) {
    case 'a':
        printf("Why?\n");
        break;
    case 'c':
        printf("Otherwise you will ");
    case 'b':
        printf("fall through!\n");
        break;
    default:
        break;
}
```

Type Casting

- We can convert between types with casting either implicitly or explicitly
 - Ex: `unsigned int i = -1;`
 - Ex: `long s = (unsigned int) -1;`
- C does not have the concept of generics the way other languages do
- Instead C uses `void *` and `char *` to generalize pointer types. Can use `char *` to modify data at the byte level.
- It's easy to cast to fix compilation errors but break your program as a result.

Truthiness in C

- In C, the only false values are things that evaluate to 0
 - 0, NULL, false (w/ #include <stdbool.h>)
- All other values are truthy.
- It's not uncommon in C to see while (1) { ... }
 - This is an infinite loop in C. (Can also do for(;;) {})

Pointers

Memory Manipulation



Pointer Syntax

- Denote variables as pointers with *
 - `int* my_pointer` means a pointer to an integer
 - `char** my_db_pointer` means a pointer to a pointer to a character
- & – Address operator: get the memory address of a variable
 - `int x = 5;`
 - `int* my_pointer = &x;`
- * – Dereference operator: get the value of the variable that is pointed to
 - `int my_int = *my_pointer; // Gets the value of integer pointed to by my_pointer.`
 - `*my_pointer = 7; // Changes the value of the integer pointed to by my_pointer.`
- Pitfall with Multiple Pointer Declarations in one line:
 - `int *x, y;` declares `x` to be a pointer to `int` but `y` to be an `int`. Use `int *x, *y` instead.

Pointers in C and Memory

	x ⁰	x ¹	x ²	x ³	
0x7FFFFFFC	00	00	00	04	int x = 4;
0x7FFFFFF8	FF	FF	FF	FF	int y = -1;
0x7FFFFFF4	7F	FF	FF	FC	int* p = 0x7FFFFFFC;
0x7FFFFFF0	00	AB	CD	EF	garbage
0x7FFFFEC	'6'	'1'	'C'	'!'	(string literal) "ILUV61C!"
0x7FFFFE8	'I'	'L'	'U'	'V'	
0x7FFFFE4	7F	FF	FF	E8	char* s = 0x7FFFFE8;
	...				

- Can think of memory as one big array
- A *memory address* is just the *index* of the memory array
- Each memory address holds one byte of data (*byte addressing*)
- **Q: Is this a 32-bit or 64-bit system?**
- **A: 32-bit (memory addresses)**

Pointers in C

- The **value of a pointer** is the **memory address** of what it's pointing to

0xFFFFFFFF

...

0xF93209B0

0x61C

0xF93209AC

0x2A

...

0xF9320904

0xF93209AC

0xF9320900

0xF9320904

...

0x00000000

```
int x = 1564;
```

```
int y = 42;
```

...

```
int* p = _____;
```

```
int** db_p = _____;
```

*The stack grows down.

Task: Make p point to y,
and make db_p be pointer
that points to p.

Pointers in C

- The **value of a pointer** is the **memory address** of what it's pointing to

0xFFFFFFFF

...

0xF93209B0

0x61C

0xF93209AC

0x2A

...

0xF9320904

0xF93209AC

0xF9320900

0xF9320904

...

0x00000000

```
int x = 1564;
```

```
int y = 42;
```

...

```
int* p = &y;
```

```
int** db_p = &p;
```

*The stack grows down.

Pointers in C

- The **value of a pointer** is the **memory address** of what it's pointing to

0xFFFFFFFF

...

0xF93209B0

0x61C

0xF93209AC

0x2A

...

0xF9320904

0xF93209AC

0xF9320900

0xF9320904

...

0x00000000

--

```
int x = 1564;  
int y = 42;  
...  
int* p = &y;  
int** db_p = &p;
```

What do the following evaluate to?

- `&db_p`
- `db_p`
- `*db_p`
- `**db_p`

Pointers in C

- The **value of a pointer** is the **memory address** of what it's pointing to

0xFFFFFFFF

...

0xF93209B0

0x61C

0xF93209AC

0x2A

...

0xF9320904

0xF93209AC

0xF9320900

0xF9320904

...

0x00000000

--

```
int x = 1564;
```

```
int y = 42;
```

```
...
```

```
int* p = &y;
```

```
int** db_p = &p;
```

What do the following evaluate to?

- `&db_p` — 0xF9320900
- `db_p` — 0xF9320904
- `*db_p` — 0xF93209AC
- `**db_p` — 0x2A

Pointers in C

- Think of “*” as the inverse of “&” and vice versa.
- NULL is used as the value for an invalid/uninitialized pointer
- Just passing an address doesn't make it valid
 - Using memory that is not legal/in scope can lead to program crashes (segfaults)

sizeof

- In C, the sizes of types may depend on the system you're targeting:
 - Ex: Size of a pointer. Could be 4 bytes (32-bit) or 8 bytes (64-bit).
- In C, we use `sizeof` to determine the size
 - Ex: `sizeof(int)` == # bytes in an integer
 - `sizeof(int*)` == `sizeof(char*)` == # bytes in a pointer
- `sizeof` is important to use for memory allocation
 - Malloc calls should include `sizeof` somewhere
 - Ex: `int* x = malloc(sizeof(int))`

Writeable Function Parameters

- When a function is called, its parameters are *copied* onto its stack
 - C is a pass by value language, like Java
- Changes to these values will be lost when the function returns
- Solution:
 - pass in a pointer as a parameter
 - function receives copy of pointer
 - function uses this pointer to access/edit the contents it points to
 - the changes will persist after the function returns

Writable Function Parameters Example

```
void f(int x) {  
    x += 1;  
}  
...  
int y = 1;  
f(y);  
printf("%d", y); // "1"
```


Writable Function Parameters Example

```
→ 1 void f(int x) {  
→ 2     x += 1;  
3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(y);  
8     printf("%d", y); // "1"  
9  
10 }
```

Stack

main

y | int
1

f

x | int
1

Writable Function Parameters Example

```
1 void f(int x) {  
→ 2     x += 1;  
→ 3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(y);  
8     printf("%d", y); // "1"  
9  
10 }
```

Stack

main

y	int 1
---	----------

f	
x	int 2

Writable Function Parameters Example

```
1 void f(int x) {  
2     x += 1;  
3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(y);  
8     printf("%d", y); // "1"  
9  
10 }
```

Stack

main

y

int
1

C/C++ [details](#)

Print output (drag lower right corner to resize)

1

Writable Function Parameters Example

```
1 void f(int* x) {  
2     *x += 1;  
3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(&y);  
8     printf("%d", y); // "2"  
9  
10 }
```

Stack

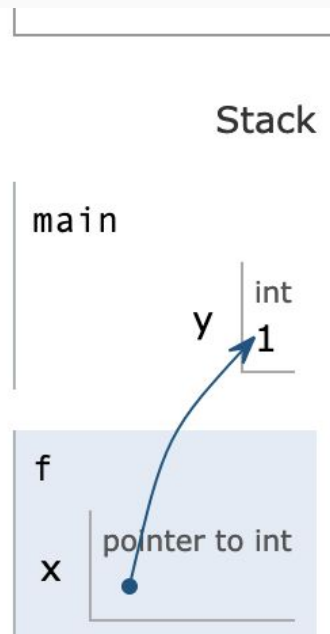
main

y	int 1
---	----------

C/C++ [details](#)

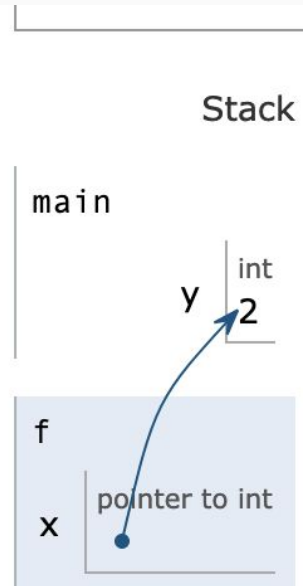
Writable Function Parameters Example

```
→ 1 void f(int* x) {  
→ 2     *x += 1;  
3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(&y);  
8     printf("%d", y); // "2"  
9  
10 }
```



Writable Function Parameters Example

```
1 void f(int* x) {  
→ 2     *x += 1;  
→ 3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(&y);  
8     printf("%d", y); // "2"  
9  
10 }
```



Writable Function Parameters Example

```
1 void f(int* x) {  
2     *x += 1;  
3 }  
4  
5 int main() {  
6     int y = 1;  
7     f(&y);  
8     printf("%d", y); // "2"  
9  
10 }
```

Stack

main

y

int
2

Print output (drag lower right corner to resize)

2

Arrays in C

- An array is a contiguous region of memory of fixed size
 - `int array[3];`
- Either static or allocated on the stack.
- Referenced by an address to their first element
 - `array == &array[0]`
- Access elements via pointer arithmetic
 - `array[i] == *(array + i)`
- Arrays don't have an end marker
 - programmer's responsibility to keep track of the size of an array

Pointer Arithmetic

- `int lottery_numbers[3] = {62, 55, 30};`
- `lottery_numbers[0] == *lottery_numbers == 62`
- Q: `*(lottery_numbers + 2)` or `*(lottery_numbers + 2*sizeof(int))`?
- A: `*(lottery_numbers + 2)`

The compiler knows to multiply by `2*sizeof(int)`, and does it for you.

- This is valid for any type-defined pointer, the compiler will automatically advance the correct number of bytes based on the pointer's type

Arrays are not Pointers!

Arrays and pointers share a few characteristics, but they are not the same.

1. Pointers can point to anything. Even invalid memory. Arrays are either static or on the stack (valid by definition).
2. Arrays are not reassignable. Ex: `int arr[10]; arr = arr + 1;` will error. Similar to const pointers in that regard. Non const pointers can be reassigned.
3. Arrays include size in their type. An int array of 9 elements is a different type than an int array of 10 elements, for example.
4. Arrays that have their size determined at compile time can have their size queried using `sizeof`.

Arrays Carry Size Information

```
int arr[10];  
int *p = arr;  
printf("%lu\t%lu\n", sizeof(arr), sizeof(p));
```

40 8

Pointers to Arrays

```
int lottery_numbers[3] = {62, 55, 30}; (pointer w/ space for 3 integers)
```

Q: Which of the following gives a pointer to the `lottery_numbers` array?

```
void* ptr_one = lottery_numbers;  
void* ptr_two = &lottery_numbers;  
void* ptr_tre = &lottery_numbers[0];
```

Pointers to Arrays

```
int lottery_numbers[3] = {62, 55, 30}; (pointer w/ space for 3 integers)
```

Q: Which of the following gives a pointer to the `lottery_numbers` array?

```
void* ptr_one = lottery_numbers;  
void* ptr_two = &lottery_numbers;  
void* ptr_three = &lottery_numbers[0];
```

A: All of them are correct and equivalent to each other!

For more information read [here](#).

Pointers to Arrays

- `int lottery_numbers[3] = {62, 55, 30};`

How can `lottery_numbers == &lottery_numbers`?

- `lottery_numbers` -> gives you the address of `lottery_numbers[0]`
 - `&lottery_numbers` -> gives you the address of the entire array, which is the same thing
-
- `&lottery_numbers` and `lottery_numbers` evaluate to pointers to different things
 - `lottery_numbers` decays to **an int ***
 - `&lottery_numbers` points to **an int array of size 3**
 - Try adding 1 to each pointer!

Data Structures in C

structs, typedefs.

Typedef

- typedef creates a new type that has the exact same structure as a data type
- Syntax for typedef
 - `typedef <data type name> <new data type name>;`
 - `typedef int my_int;`
 - `typedef long long LL;`
- Commonly used to create new types from structs

Structs

- structs allow us to create groups of different types (arrays, int, char)
- struct syntax
 - `struct my_struct { int x; char* y; };`
- Usually used in conjunction with typedef
 - `typedef struct { int x; char* y; } my_struct;`
 - `my_struct instance;`
- structs DO NOT have methods, constructors, or destructors
- Alternate notation when accessing fields of structs via pointers:
 - `my_struct* instance_ptr = &instance;`
 - `(*instance_ptr).x == instance_ptr->x`
 - `(*struct_instance).field == struct_instance->field`

Initializing a struct

```
typedef struct my_struct {  
    int my_int;  
    char my_char;  
} my_struct_t;  
// typedef means "my_struct_t" is equivalent to  
"struct my_struct"
```

```
// Method 1: declare struct on stack  
// initialize fields manually  
void main() {  
    my_struct_t s1;  
    s1.my_int = 1;  
    s1.my_char = 'a';  
}
```

```
// Method 2: malloc and initialize manually  
// (most common, especially if struct  
// ptr is being returned)  
// Note the use of -> instead of .  
my_struct_t* s2 = malloc(sizeof(my_struct_t));  
s2->my_int = 3;  
s2->my_char = 'c';
```

```
// The following is incorrect  
my_struct s3;
```

```
// The following is correct  
struct my_struct s4;
```

Ex: Using typedef

(1)

```
struct ListNode {  
    char* value;  
    struct ListNode* next;  
};  
  
typedef struct ListNode LinkNode;
```

(2)

```
typedef struct ListNode {  
    char* value;  
    struct ListNode* next;  
} LinkNode;
```

types.h

`off_t`: **signed integer**, used for file offsets

`pid_t`: **signed integer**, used for process IDs

`pthread_t`: **unsigned integer**, used to identify a thread

`size_t`: **unsigned integer**, used for sizes of objects

Why so many types? Can't we have `int` instead of `pid_t`?

- This is for accommodating different types of devices. Some devices may be 32-bit machines vs 64-bits.

Pointers and Structs

- Alternate notation when dealing with pointers to structs.
 - `(*struct_pointer).value == struct_pointer->value`

Working with C Data Structures + Pointers!

- Complete the example function that adds a new node to the back of the list.

```
void add_node_to_back (ListNode* head, char* str) {  
    while (head->next != NULL) { // Iterate through list to find end  
        _____;  
    }  
    _____;  
    _____;  
    _____;  
    _____;  
}
```

```
typedef struct ListNode {  
    char* value;  
    struct ListNode* next;  
} ListNode;
```

Working with C Data Structures + Pointers!

```
void add_node_to_back (LinkNode* head, char* str) {  
    while (head->next != NULL) { // Iterate through list to find end  
        head = head->next;  
    }  
    LinkNode* new_node = (LinkNode*) malloc(sizeof(LinkNode));  
    // +1 copies over null terminator  
    new_node->value = (char*) malloc(sizeof(char)*(strlen(str)+1));  
    strncpy(new_node->value, str, strlen(str)+1);  
    new_node->next = NULL;  
    head->next = new_node;  
}
```

C Basics

The following code has undefined behavior. Identify all of the following bugs in the code.

Pointer Basics

```
// Print out all array elements,  
// each element on a newline
```

```
void print_array (int* arr) {  
    while (*arr != NULL) {  
        printf ("%d\n", *arr);  
        arr += sizeof (int);  
    }  
}
```

```
int main () {  
    // Array intended to consist of  
    // 1, 2, 0, 5  
    int a[] = {1, 2, 0, 5, NULL};  
    // Should print:  
    // 1  
    // 2  
    // 0  
    // 5  
    print_array (a);  
}
```

Pointer Basics

```
// Print out all array elements,  
// each element on a newline
```

```
void print_array  
(int* arr, size_t size) {  
    int* endpoint = arr + size;  
    while (arr < endpoint) {  
        printf ("%d\n", *arr);  
        arr += 1;  
    }  
}
```

```
int main () {  
    // Array intended to consist of  
    // 1, 2, 0, 5  
    // No need for NULL  
    int a[] = {1, 2, 0, 5};  
    // Should print:  
    // 1  
    // 2  
    // 0  
    // 5  
    print_array (a, 4);  
}
```

Strings in C

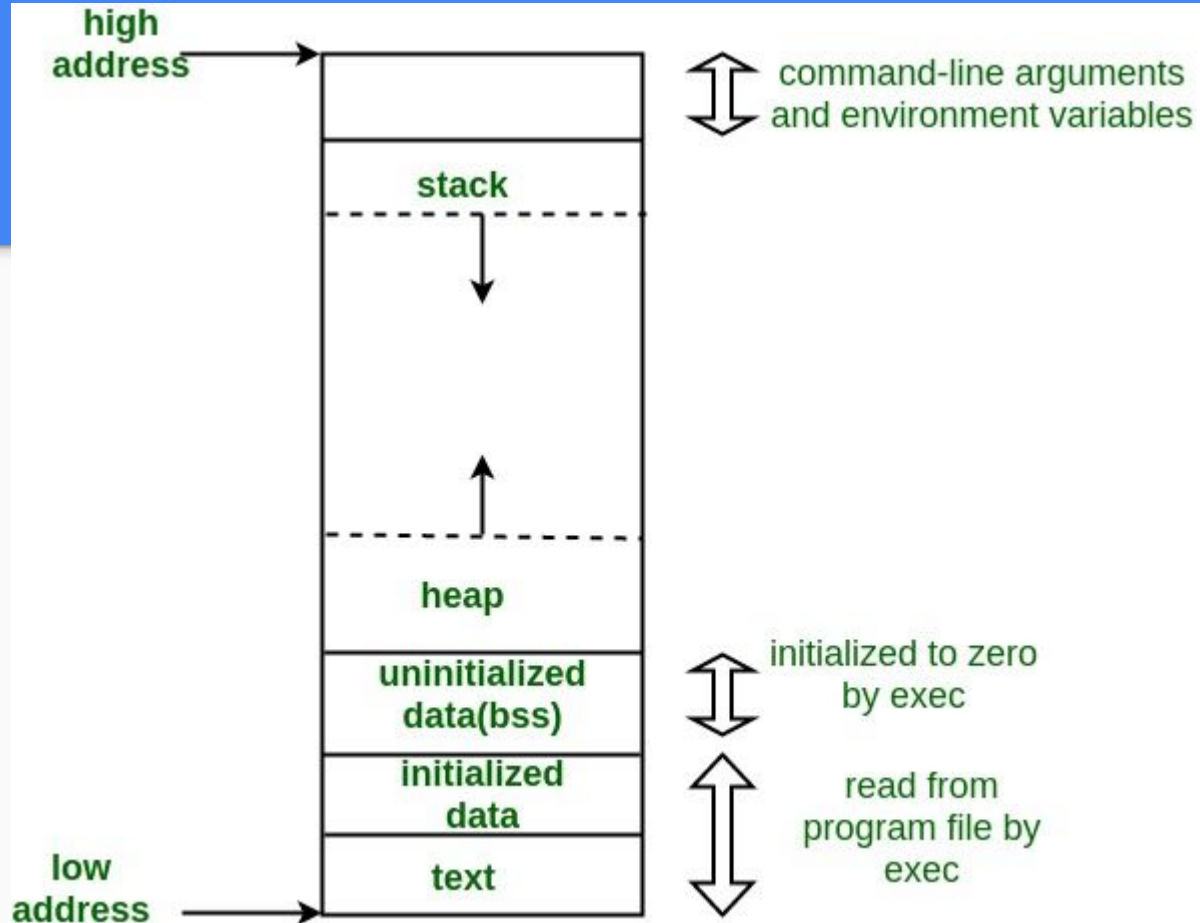
- A string in C is just an array of characters (type is `char*`)
 - A proper string always ends with a null terminator '`\0`'
- C Library functions assume proper strings (e.g. `strlen`, `strcpy`, `strcmp`)
 - very unsafe assumption, errors can occur
- Functions that have length as a parameter are safer (still not good though)
 - ex) `strncpy`, `strncat`, etc.

<code>char a[10] = "abcd";</code>	<code>char *p = "efgh";</code>
1. a is an array	1. p is a pointer variable
2. <code>sizeof(a)</code> = 10 bytes	2. <code>sizeof(p)</code> = 4 bytes
3. a and &a are same	3. p and &p aren't same
4. abcd is stored in stack section of memory	4. p is stored at stack but efgh is stored at code section of memory
5. <code>char a[10] = "abcd";</code> <code>a = "hello";</code> // invalid --> a, itself being an address and string constant is also an address, so not possible.	5. <code>char *p = "efgh";</code> <code>p = "india";</code> // valid
6. <code>a++</code> is invalid	6. <code>p++</code> is valid
7. <code>char a[10] = "abcd";</code> <code>a[0] = 'b';</code> // valid	7. <code>char *p = "efgh";</code> <code>p[0] = 'k';</code> // invalid. -->Code section is r- only.

C Memory Layout

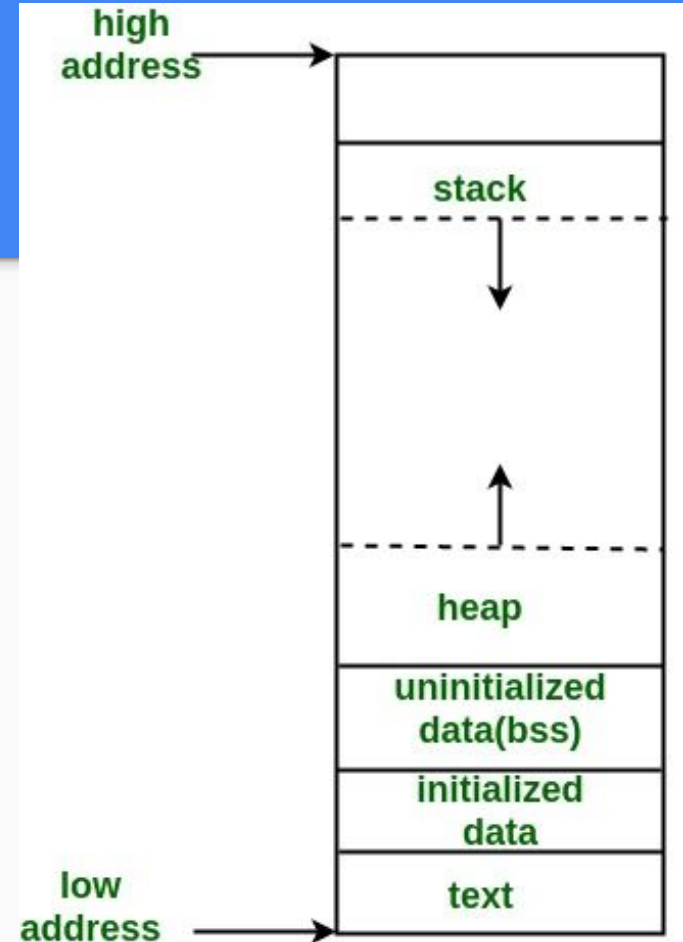


Memory Layout



Memory Layout (explained)

- Text: Actual instructions of the Program
- Data (+ BSS): Statically allocated data (global and static variables, Strings, constants)
- Stack: Local variables for each function call
- Heap: Dynamic memory that persists beyond function calls (malloc)



C Memory (Global)

- Global variables can be accessed by all functions and exist throughout the duration of a program's lifetime
- Convenient, but dangerous
 - no access control, namespace pollution, testing/confinement issues (hard to unit test), concurrency issues, etc.
 - bad practice in large-scale software engineering projects

C Memory (Stack)

- Stores local arguments and function parameters in a stack frame
- When a function is invoked, a stack frame is pushed to the stack
- When a function returns, its stack frame is removed from the stack

Do not return a pointer to the stack.

Instead, store and return data on the heap (with malloc/calloc) or make them global if it's appropriate.

C Memory (Heap)

- Memory on the heap is requested with the alloc series of functions
- `void* malloc (size_t nbytes)`
 - Return a pointer to n bytes of data
- `void* calloc (size_t nelems, size_t elemsize)`
 - Return a pointer to elemsize * nelems bytes of data (recommended for arrays, strings)
 - Data is zeroed-out
- `void* realloc (void* ptr, size_t nbytes)`
 - Resizes a block of memory that was previously allocated (without losing old data)
 - Realloc can return a NEW pointer—be sure to update any references to the old pointer after
- Release heap-allocated memory with `free (void* ptr)`

Helpful Tip: Use `malloc(sizeof(<insert type>))` instead of `malloc(<number>)`

Avoid padding issues!

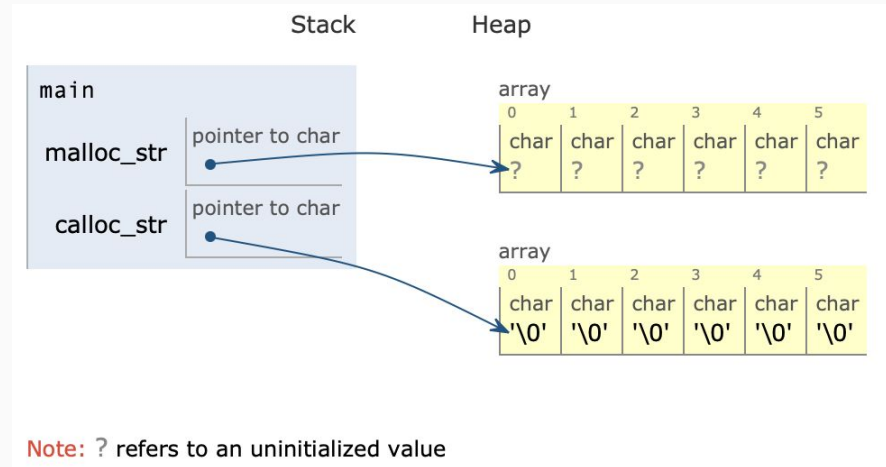
C Memory (Heap) String Example

`malloc((5+1)*sizeof(char))`

`calloc(5+1, sizeof(char))`

Note how `calloc` zeroes out the memory.

The +1 represents the null terminator.



C Memory (Heap)

WARNING: For strings:

calloc “provides” null-terminators (because it zeros out all allocated memory); however, malloc **does not** provide null-terminators. You will need to set the null terminator appropriately. Good practice to put null terminator manually anyway.

For both malloc and calloc, be sure to request one additional byte for the null-terminator. For **malloc**, be sure to actually set that byte to the null-terminator.

About Free

- Every time you add a malloc/calloc, think about where you will free that data so it happens every time.
 - But of course, don't free memory that will be used in other places :)
- We will explicitly check for memory leaks (Unfreed, unused heap allocations) in projects
 - Run valgrind!

Let's take a break!

Review/Examples

C Memory

In the following program we have provided 5 print statements. State which print statements will not always succeed and why. Assume all necessary includes.

C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello;
}
```

```
char* f1 () {
    return "hello";
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
}
```

```
char* f2 () {
    return global;
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr;
}
```


C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr;
}
```

```
char* f1 () {
    // string literals stored in data segment
    // (null termination is done by compiler)
    return "hello";
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello;
}
```

```
char* f2 () {
    return global;
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
}
```

C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr;
}
```

```
char* f1 () {
    // string literals stored in data segment
    // (null termination is done by compiler)
    return "hello";
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello;
}
```

```
char* f2 () {
    return global; // no null terminator
    // would work if global had a null terminator
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
}
```

C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr;
}
```

```
char* f1 () {
    // string literals stored in data segment
    // (null termination is done by compiler)
    return "hello";
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello; // cannot return stack array
}
```

```
char* f2 () {
    return global; // no null terminator
    // would work if global had a null terminator
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
}
```

C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr;
}
```

```
char* f1 () {
    // string literals stored in data segment
    // (null termination is done by compiler)
    return "hello";
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello; // cannot return stack array
}
```

```
char* f2 () {
    return global; // no null terminator
    // would work if global had a null terminator
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
    // no null terminator
}
```

C Memory

```
char global[] = {'h', 'e', 'l', 'l', 'o'};
int main () {
    printf ("%s\n", f1());
    printf ("%s\n", f2());
    printf ("%s\n", f3());
    printf ("%s\n", f4());
    printf ("%s\n", f5());
}
```

```
char* f5 () {
    char* arr = calloc(strlen("hello") + 1,
                       sizeof (char));
    for (int i = 0; i < 5; i++) {
        arr[i] = "hello"[i];
    }
    return arr; // calloc adds null terminator
}
```

```
char* f1 () {
    // string literals stored in data segment
    // (null termination is done by compiler)
    return "hello";
}
```

```
char* f3 () {
    char hello[] = "hello";
    return hello; // cannot return stack array
}
```

```
char* f2 () {
    return global; // no null terminator
    // would work if global had a null terminator
}
```

```
char* f4 () {
    return malloc (strlen ("hello") + 1);
    // no null terminator
}
```

libc

string manipulation, file i/o



Useful Functions to Know

- `strlen` - returns the length of a string (not including the null terminator)
- `strcpy` - copies the characters from `src` string to `dest` string
- `strcmp` - compares two strings lexicographically; returns an integer
- `fprintf` - print formatted strings to a specified file
- `fopen` - opens a `FILE*`
- `fclose` - close a `FILE*`
- `fread` - read contents from a `FILE*`
- `fwrite` - write contents to a `FILE*`

File I/O Example

```
int write_char(char* filename, char c) {  
    FILE* file = fopen(filename, "w");  
    if (file == NULL) return 0;  
    if (fwrite(&char, 1, 1, file) != 1) {  
        fclose(file);  
        return 0;  
    }  
    fclose(file);  
    return 1;  
}
```

(void* ptr, size_t size, size_t
nmemb, FILE* stream)



strcpy Implementation

```
char* strcpy(char* dest, const char* src) {  
    if (dest == NULL) {  
        return NULL;  
    }  
    while (*src != '\0') {  
        *dest = *src;  
        dest++;  
        src++;  
    }  
    return dest;  
}
```

strcpy() copies the string pointed to by src into the buffer pointed to by dest, and returns the pointer to dest.

What's the issue with this solution?

strcpy Implementation

```
char* strcpy(char* dest, const char* src) {  
    if (dest == NULL) {  
        return NULL;  
    }  
    char* ptr = dest;  
    while (*src != '\\0') {  
        *dest = *src;  
        dest++;  
        src++;  
    }  
    *dest = '\\0';  
    return ptr;  
}
```

strcpy() copies the string pointed to by src into the buffer pointed to by dest, and returns the pointer to dest.

strlen Implementation

```
long int strlen_staff(const char* src) {  
    /* INSERT CODE HERE */  
}
```

strlen() returns how many characters are in the string pointed to by *src.

strlen Implementation (sol'n)

```
long int strlen_staff(const char* src) {  
    long int count = 0;  
    while(*src != '\\0') {  
        count++;  
        src += 1;  
    }  
    return count;  
}
```

strlen() returns how many characters are in the string pointed to by *src.

fprintf formatting

- Prints a formatted string to the specified `FILE*`
- For each conversion specification (`%[char]`), provide an argument to be printed
 - `fprintf(file1, "Line %d: %s", 1, "Segmentation Fault")`
- Some common specifications: `%d` [decimal], `%u` [unsigned decimal], `%c` [character], `%s` [string], `%f` [double], `%p` [address]
- `printf(...)` = `fprintf(stdout, ...)`

Swap

```
void swap(int *a, int *b) {  
    int* temp;  
    temp = a;  
    a = b;  
    b = temp;  
}  
  
int main () {  
    int a = 1;  
    int b = 2;  
    swap(&a, &b);  
    printf("%d %d\n", a, b);  
}
```

What does the following print?

Swap

```
void swap(int *a, int *b) {  
    int* temp;  
    temp = a;  
    a = b;  
    b = temp;  
}  
  
int main () {  
    int a = 1;  
    int b = 2;  
    swap(&a, &b);  
    printf("%d %d\n", a, b);  
}
```

What does the following print?

1 2

While we are swapping the pointers, these changes are only applicable inside the swap function. To persist them to main, we can swap the values instead.

```
int temp;  
temp = *a;  
*a = *b;  
*b = temp;
```

Strings

```
char* create_reversed_string(char* str) {  
    int length = _____;  
    char* new_str = malloc(____);  
    for (int i = ____; i < ____; ____) {  
        new_str[i] = str[_____];  
    }  
    _____;  
    return new_str;  
}
```

Fill in the blanks to create a reversed string.

Do not reverse the string in-place, use malloc to create a new string instead.

Strings

```
char* create_reversed_string(char* str) {  
    int length = strlen(str);  
    char* new_str = malloc(length+1);  
    for (int i = 0; i < length; i++) {  
        new_str[i] = str[length-i-1];  
    }  
    new_str[length] = '\\0';  
    return new_str;  
}
```

Important Notes

- Allocate space for null-terminator
- Set null-terminator after reversing string
- Followup: Do we need to set the null terminator if we use calloc?

Git

Everyone's Favorite Version Control

git checkout

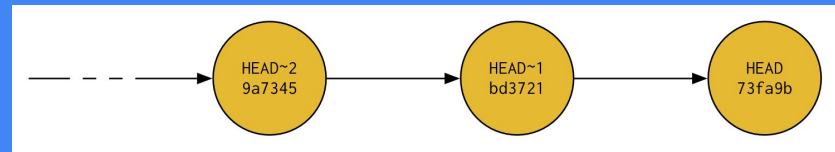
- I want to go back (temporarily) to a previous commit!
 - **Git checkout** <commit hash> or HEAD
- You can also checkout to remotes
 - Git checkout **staff main**
 - Checkout staff skeleton codes!

git reset

- ***What if I mess up?*** Git reset to the rescue!
- Go back to a specific commit and preserve changes as unstaged edits
 - `$ git reset <commit>`
- Different from git checkout
 - Git checkout: Go back to the commit but keep the entire history intact
 - Git reset: unstage the edits and remove the commits before
- *Please do not use `-hard` with git reset, if possible!*

git log

- Sometimes we need to look at the project's history
- The commit history, or git log, shows us all of the commits that preceded the most recent commit
- To view it, use the **git log** command (depending on your terminal, you may need to press "q" to return to the command line)
- To show an abbreviated version, add the **--oneline** option:
 - **git log --oneline**
- Notice our new commit appearing in the log!



git diff <commit1> <commit2>

- We've used the log to confirm that a new commit was made, but how do we know that the new commit contains the changes that we want?
- Use the **git diff <commit1> <commit2>** command! ("diff" = difference)
- It shows us the *difference* between the two provided commits
- Usually, we are comparing the latest commit to some other commit
 - The latest commit is always called HEAD
 - The commit immediately prior to HEAD is HEAD~1
 - The commit immediately prior to HEAD~1 is HEAD~2 (two commits before HEAD)
- To compare the latest commit and the one prior to that:
 - **git diff HEAD~1 HEAD**

A Few Pieces of Advice

- Make sure to git pull before working on any files
- Let people know what and what files you are working on to **avoid merge conflicts**
 - Make new branches for portions of your project
 - Pair programming and VSCode Liveshare!
 - If encountered merge conflicts, resolve them carefully
- Be careful when working off of previous commits (ie. git checkout)
 - Can result in diverges in history and messy git stuff

More Advice

- Use 'git checkout staff/main -- file_name' to replace your version with the staff version.
 - Particularly useful when you accidentally modify a test file in projects!
- Avoid using 'git push -f'