

# **CS 211: Intro to Computer Architecture**

## ***7.1: Dynamic Memory Management***

**Minesh Patel**

Spring 2025 – Tuesday 4 March

# Announcements

- PA2 survey announced via Canvas
- This week:
  - **WA4 + WA5**: assigned through Canvas; due **next Tuesday @ 23:59**
    - Short and simple: combined difficulty == WA1/2/3
  - **PA3**: TBA tomorrow or Thursday, due after spring break
    - Encompasses pointers, arrays, and memory management
    - Please skim the doc and code immediately so you know what you're getting into
- Next week:
  - No recitations, but extra office hours instead
  - Tuesday's lecture will be partially exam review

# Midterm Preparation

- The WAs will be representative for questions
- We will draw on material from lectures, WAs, and PAs
- We are working on:
  - Some sort of “practice sheet” for later this week
  - A list of topics that will be covered

# The CSL and RUCATS are Both Hiring

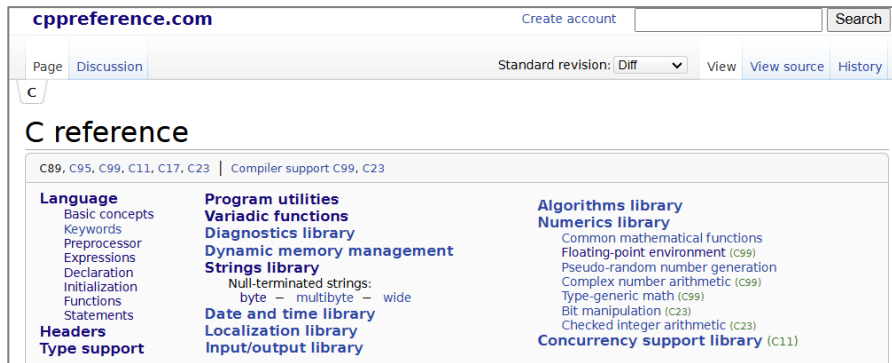
- [RUCATS hiring form](#)
- [CSL hiring form](#)



# Reference Material

- Today's lecture draws heavily from:
  - [CS 61C @ UC Berkeley](#) (Prof. Dan Garcia)

## And Various C and Linux Reference Materials



cppreference.com

Create account Search

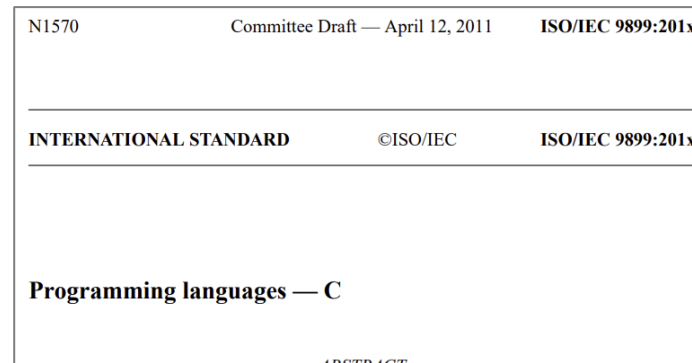
Page Discussion Standard revision: Diff View View source History

C

### C reference

C89, C95, C99, C11, C17, C23 | Compiler support C99, C23

<b>Language</b> <ul style="list-style-type: none"><li>Basic concepts</li><li>Keywords</li><li>Preprocessor</li><li>Expressions</li><li>Declaration</li><li>Initialization</li><li>Functions</li><li>Statements</li></ul>	<b>Program utilities</b> <ul style="list-style-type: none"><li><b>Variadic functions</b></li><li><b>Diagnostics library</b></li><li><b>Dynamic memory management</b></li><li><b>Strings library</b><ul style="list-style-type: none"><li>Null-terminated strings:<ul style="list-style-type: none"><li>byte – multibyte – wide</li></ul></li></ul></li><li><b>Date and time library</b></li><li><b>Localization library</b></li><li><b>Input/output library</b></li></ul>	<b>Algorithms library</b> <ul style="list-style-type: none"><li><b>Numerics library</b><ul style="list-style-type: none"><li>Common mathematical functions</li><li>Floating-point environment (C99)</li><li>Pseudo-random number generation</li><li>Complex number arithmetic (C99)</li><li>Type-generic math (C99)</li><li>Bit manipulation (C23)</li><li>Checked integer arithmetic (C23)</li></ul></li><li><b>Concurrency support library</b> (C11)</li></ul>
--	---	--



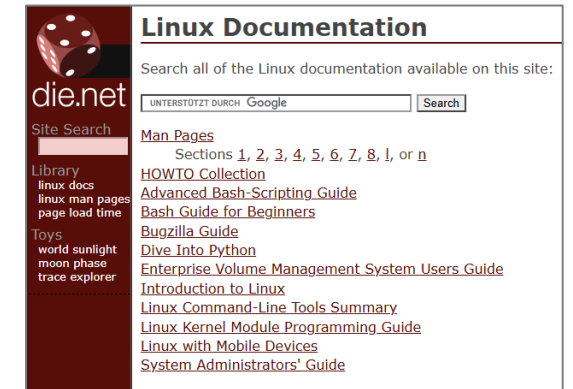
N1570 Committee Draft — April 12, 2011 ISO/IEC 9899:201x

---

**INTERNATIONAL STANDARD** ©ISO/IEC ISO/IEC 9899:201x

---

**Programming languages — C**



die.net

Linux Documentation

Search all of the Linux documentation available on this site:

UNTERSTÜTZT DURCH Google Search

Site Search

Library

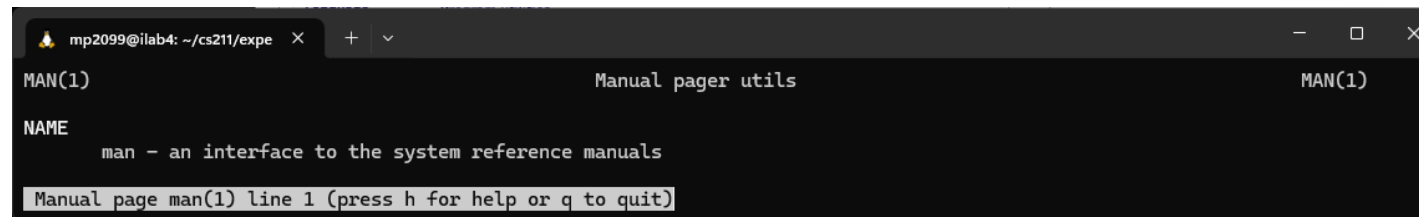
- linux docs
- linux man pages
- page load time

Toys

- world sunlight
- moon phase
- trace explorer

**Man Pages**

- Sections 1, 2, 3, 4, 5, 6, 7, 8, l, or n
- HOWTO Collection
- Advanced Bash-Scripting Guide
- Bash Guide for Beginners
- Bugzilla Guide
- Dive Into Python
- Enterprise Volume Management System Users Guide
- Introduction to Linux
- Linux Command-Line Tools Summary
- Linux Kernel Module Programming Guide
- Linux with Mobile Devices
- System Administrators' Guide



```
mp2099@ilab4: ~/cs211/expe x + v - □ x
MAN(1) Manual pager utils MAN(1)
NAME
man - an interface to the system reference manuals
Manual page man(1) line 1 (press h for help or q to quit)
```

# A Simplified C Library Reference: [manual.cs50.io](http://manual.cs50.io)

CS50 Manual Pages

Manual pages for the C standard library, the C POSIX library, and the CS50 Library for those less comfortable.

frequently used in CS50

## cs50.h

- `get_char` - prompt a user for a `char`
- `get_double` - prompt a user for a `double`
- `get_float` - prompt a user for a `float`
- `get_int` - prompt a user for an `int`
- `get_long` - prompt a user for an `long`
- `get_string` - prompt a user for a `string`

## ctype.h

- `isalnum` - check whether a character is alphanumeric
- `isalpha` - check whether a character is alphabetical
- `isblank` - check whether a character is blank (i.e., a space or tab)
- `isdigit` - check whether a character is a digit
- `islower` - check whether a character is lowercase
- `ispunct` - check whether a character is punctuation
- `isspace` - check whether a character is whitespace (e.g., a newline, space, or tab)
- `isupper` - check whether a character is uppercase
- `tolower` - convert a `char` to lowercase
- `toupper` - convert a `char` to uppercase

CS50 Manual Pages

Manual pages for the C standard library, the C POSIX library, and the CS50 Library for those less comfortable.

frequently used in CS50

## stdio.h

- `fprintf` - print to a file
- `printf` - print to the screen
- `sprintf` - print to a string

CS50 Manual Pages

Manual pages for the C standard library, the C POSIX library, and the CS50 Library for those less comfortable.

frequently used in CS50

## string.h

- `strchr` - locate character in string
- `strchrnul` - locate character in string

CS50 Manual Pages

## NAME

less comfortable

`printf` - print to the screen

## LIBRARY

[Show](#)

## SYNOPSIS

less comfortable

**Header File**

```
#include <stdio.h>
```

**Prototype**

```
int printf(string format, ...);
```

Note that `...` represents zero or more additional arguments.

# Agenda

- **Functions**

- Loaders and the C Memory Layout

- Dynamic Memory Allocation

- C Generics

- C Library Functions

# Recap: C's Type System

- void
- basic types
  - char
  - signed integers
  - unsigned integers
  - floating-point
- enumerated types
- **derived types**
  - structures
  - pointers
  - arrays
  - unions
  - **functions** ← **Almost done!**



# Recap: A 64-bit Address Space

## 64-bit address space ( $2^{64}$ bytes)

*0x0000\_0000\_0000\_0000*



```
int16_t i = INT16_MAX;  
int16_t *p = &i;
```

**Somewhere  
in memory**

**Where?**

*0xffff\_ffff\_ffff\_ffff*

# Functions in Memory

## 64-bit address space ( $2^{64}$ bytes)

0x0000\_0000\_0000\_0000

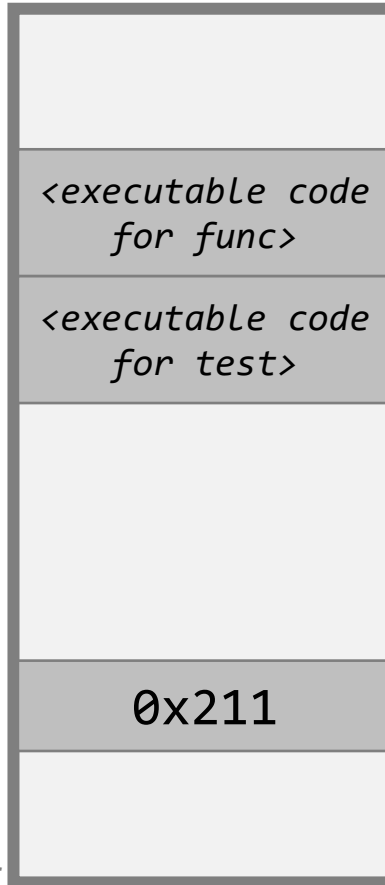
Addresses of  
"func" and "test"

$\rightarrow$   $\&func$

$\rightarrow$   $\&test$

```
void func(int a) {...}

void test(void)
{
    uint64_t u = 0x211;
}
```



**Code lives in  
memory, too**

*its binary representation  
is **numbers** called "machine code"  
(more on this after spring break)*

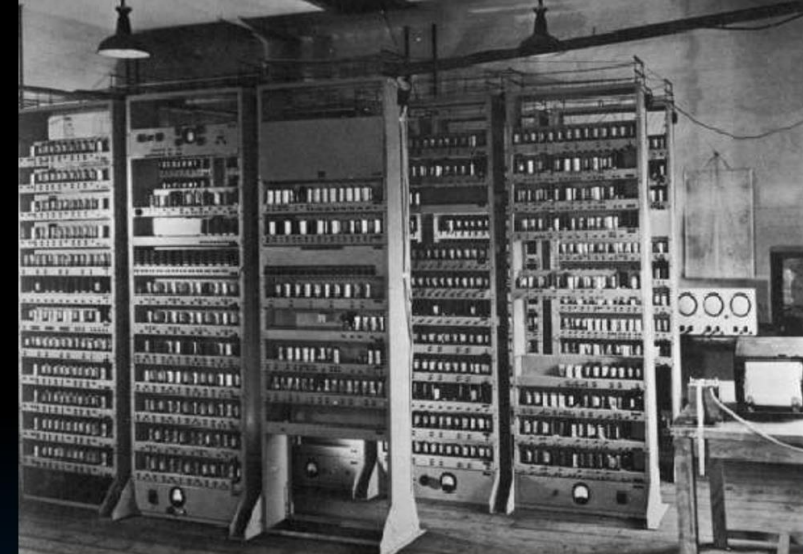
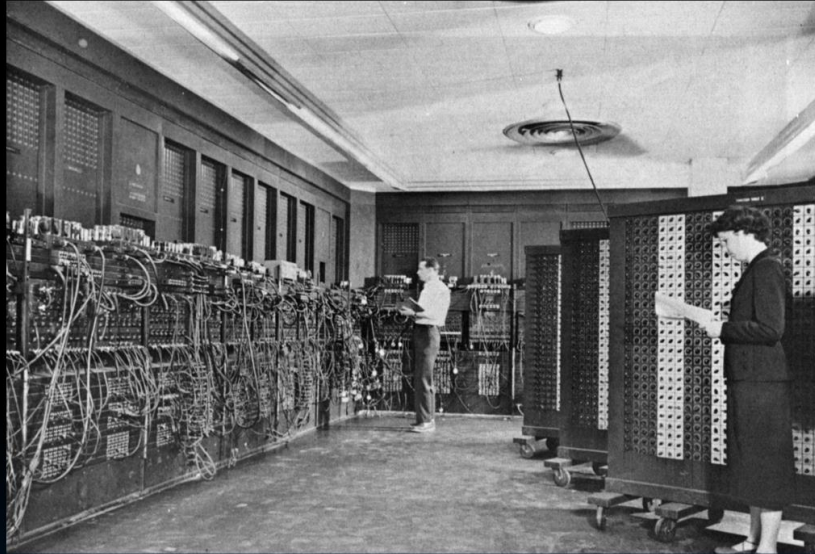
**Data**

0xffff\_ffff\_ffff\_ffff

# Code Lives in Memory, Too



## From ENIAC (1946) to EDSAC (1949)



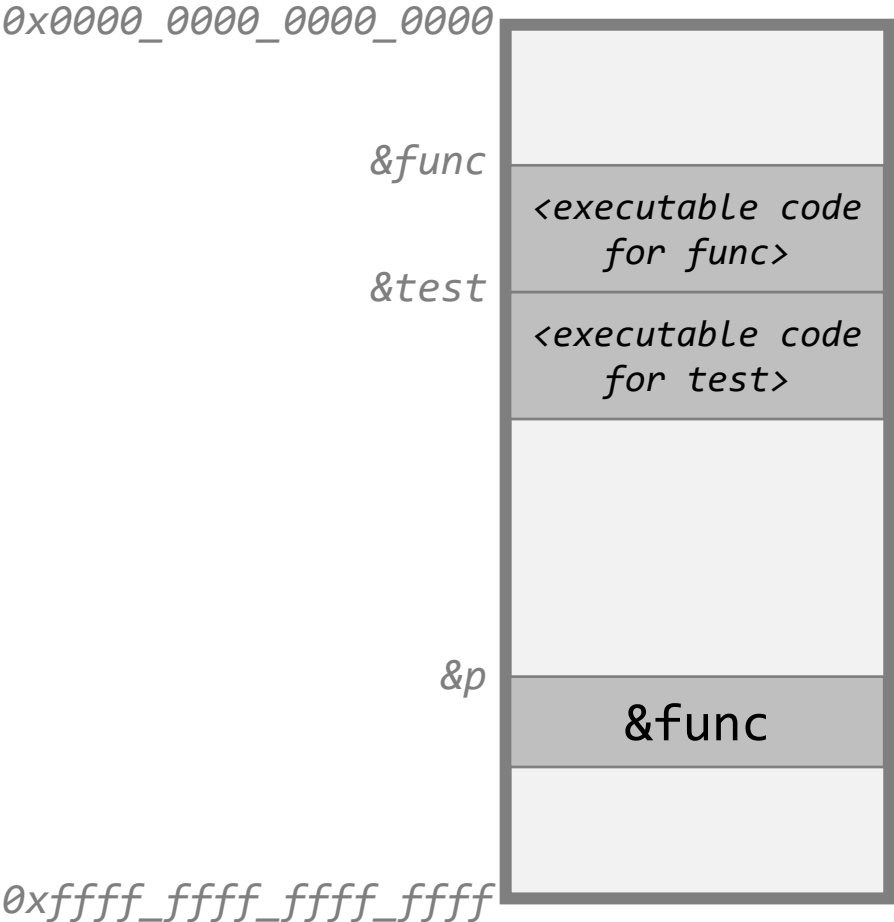
- ENIAC: First Electronic General-Purpose Computer
- Needed 2-3 days to setup new program
- Programmed with patch cords and switches
  - At that time & before, "computer" mostly referred to people who did calculations
  - Mostly women! (See *Hidden Figures*, 2016)
- EDSAC: First General **Stored-Program** Computer
- Programs held as **numbers in memory**
  - Revolution! Program is also data!
- 35-bit binary **two's complement** words

# Function Pointers

## 64-bit address space ( $2^{64}$ bytes)

```
void func(int a) {...}  
  
void test(void)  
{  
    void (*p)(int a) = &func;  
    (*p)(1);  
}
```

Identifier 'p'  
points to 'func'



# Function Pointer Syntax: Best Practice

Function identifier

```
void test(int a)
{
    void (*fn)(int a) = &test;
    (*fn)(1);
}
```

Equivalent Code

```
void test(int a)
{
    void (*fn)(int a) = test;
    fn(1);
}
```

- The compiler **implicitly converts** function identifiers to pointers
- Best practice: prefer **explicitness**

We will come back to function pointers on Thursday

# Agenda

- Functions
- **Loaders and the C Memory Layout**
- Dynamic Memory Allocation
- C Generics
  - C Library Functions

# Loading a Program into Memory

- **Q:** How does your program (all the code, C objects, etc.) get into memory?

```
netid@ilab:~$ gcc -o hello hello.c
netid@ilab:~$ ./hello world
Hello, World
netid@ilab:~$
```

- A special program called the “loader” parses the **executable file** generated by GCC and loads it bit-by-bit into memory

```
mp2099@ilab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/toolchain_glibc2/sysroot/lib/ld-linux-riscv64-lp64.so.1
/common/users/shared/cs211_s25_5678/toolchain_glibc2/sysroot/lib/ld-linux-riscv64-lp64.so.1: missing program name
Try '/common/users/shared/cs211_s25_5678/toolchain_glibc2/sysroot/lib/ld-linux-riscv64-lp64.so.1 --help' for more information.
```

```
mp2099@ilab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/toolchain_glibc2/sysroot/lib/ld-linux-riscv64-lp64.so.1 --help
Usage: /common/users/shared/cs211_s25_5678/toolchain_glibc2/sysroot/lib/ld-linux-riscv64-lp64.so.1 [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked 'ld.so', the program interpreter for dynamically-linked ELF programs. Usually, the program interpreter is invoked automatically when a dynamically-linked executable is started.
```

You may invoke the program interpreter program directly from the command line to load and run an ELF executable file; this is like executing that file itself, but always uses the program interpreter you invoked, instead of the program interpreter specified in the executable file you run. Invoking the program interpreter directly provides access to additional diagnostics, and changing the dynamic linker behavior without setting environment variables (which would be inherited by subprocesses).

# A C Program's Address Space

- The loader's job: **fill in memory**
- So far, we've treated all bytes in memory as equal
  - Surprise: **they're not equal**

*0x0000\_0000\_0000\_0000*

Memory

**64-bit  
address space**  
( $2^{64}$  bytes)

*0xffff\_ffff\_ffff\_ffff*



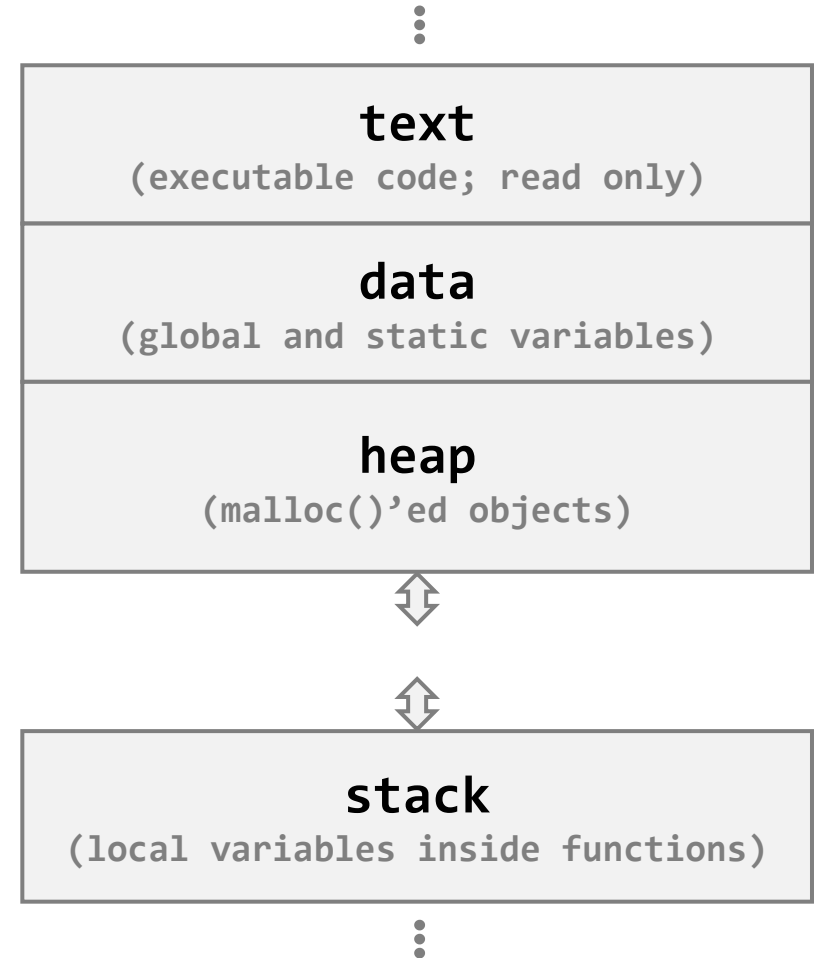
# Sections of the Address Space

```
netid@ilab:~$ gcc -o hello hello.c
netid@ilab:~$ ./hello world
Hello, World
netid@ilab:~$
```

**Loader**

- Many parts of a C program
  - **Executable code** (i.e., your functions)
  - **Data** (i.e., your C objects)
    - Initialized and uninitialized data
    - Compiler-allocated and dynamically-allocated
  - **Debugging information**
  - ...

## Typical C Program's Address Space



# Sections of the Address Space: text

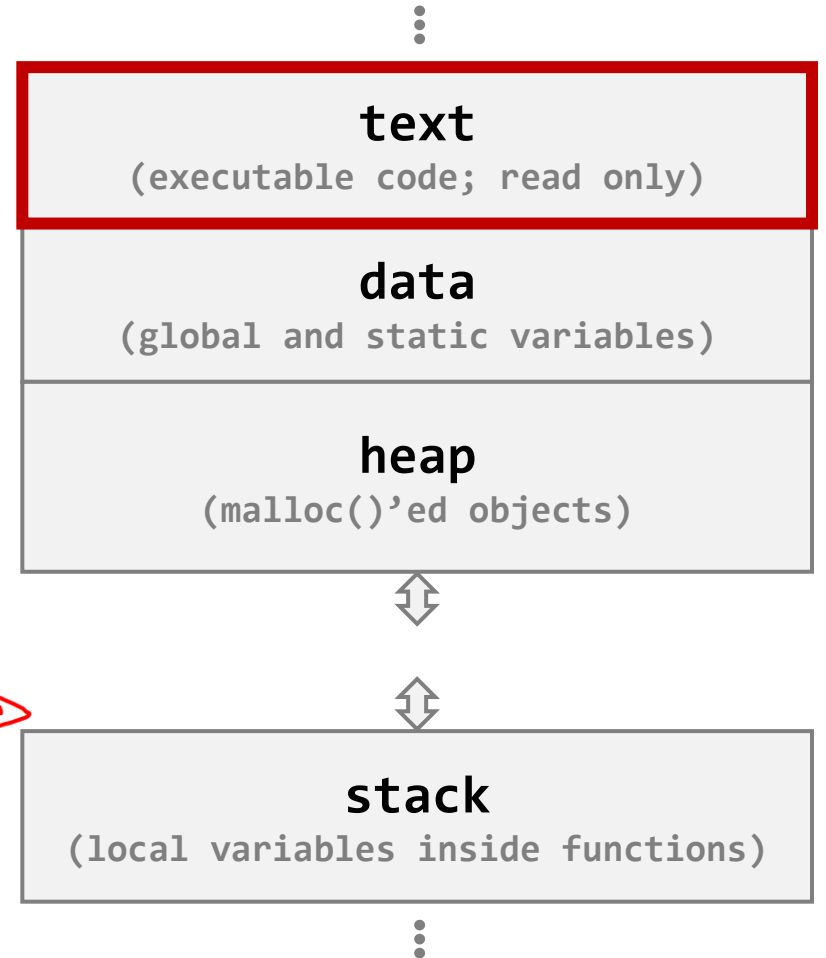
- Read-only section for executable code

```
void func(int a) {...}

void test(void)
{
  uint64_t u = 0x211;
}
```

&func →  
&test →  
&main →

## Typical C Program's Address Space



- Just a bunch of numbers
  - Representation = “machine code”
  - More on this after Spring Break

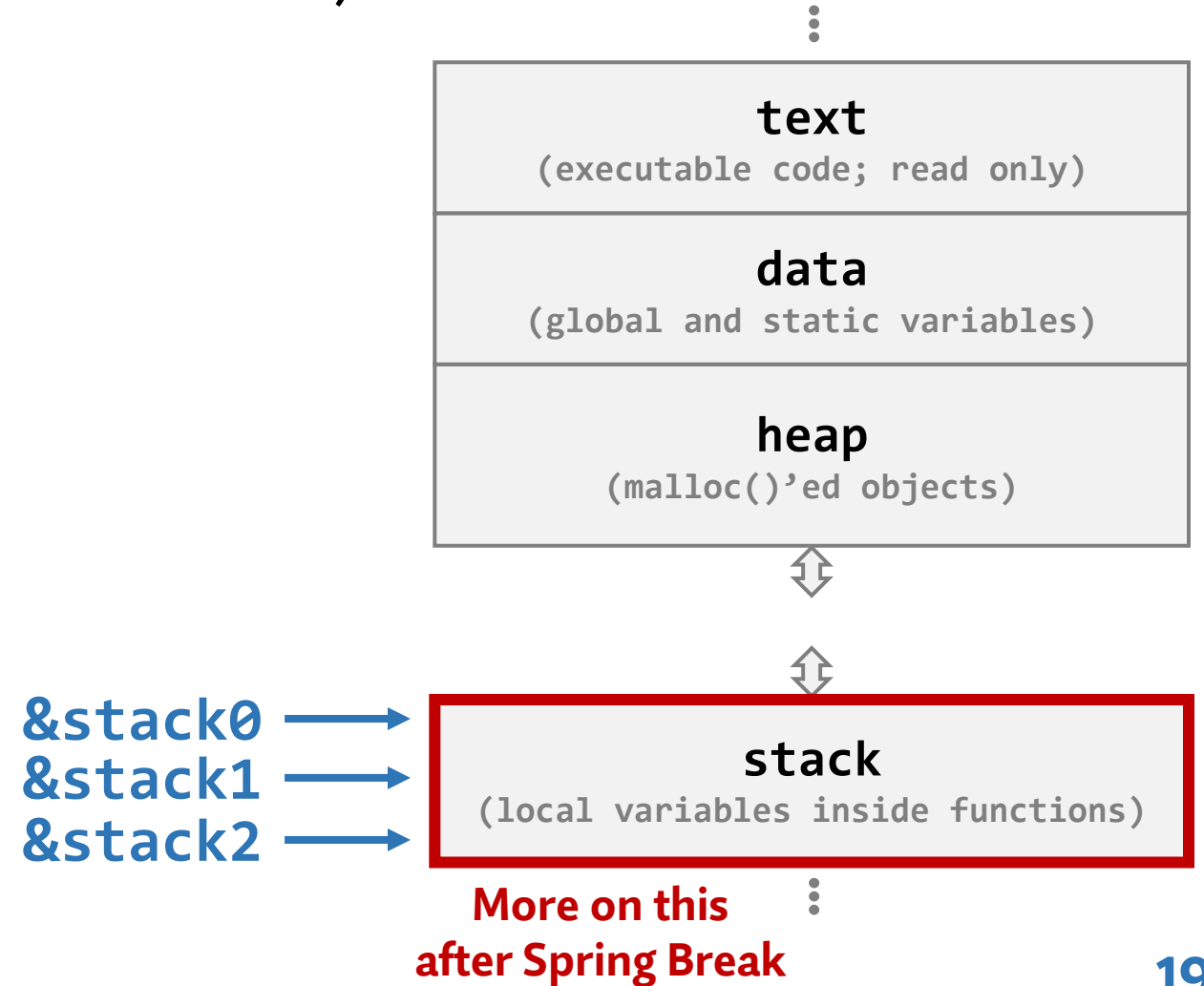
# Sections of the Address Space: stack

- Any variable inside a function (“local” variables)
- Lives **only** within the function

```
int main(int argc, char **argv)
{
    uint64_t stack0[100] = {0x211};
    uint64_t *stack1 = stack0;
    ...
}

void func(void)
{
    struct test stack2;
    ...
}
```

## Typical C Program's Address Space



# Aside: Variable Scope in C

- C objects exist only within their “scope”

## “Local”: Block Scope

- The innermost block { ... } is in scope
- Live as long as the scope

## “Global”: File Scope

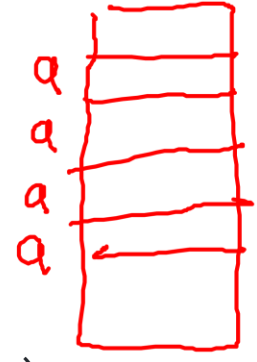
- In scope for everything in the file
- Live as long as the program

```
#include <stdlib.h>

int64_t a = 0;

void func_0(void)
{
    int64_t a = 1;
    for(int a = 2; a <= 2; a++)
    {
        {
            int a = 3;
            printf("%d\n", a); // a = 3
        }
        printf("%d\n", a); // a = 2
    }
    printf("%d\n", a); // a = 1
}

void func_1(void)
{
    printf("%d\n", a); // a = 0
}
```



# Aside: Two Types of Global Variables

## Global

*visible to all functions in the file*

```
#include <stdlib.h>

int64_t a = 0;

void func_0(void)
{
    a++;
    printf("%d\n", a); // prints 2, then 4
}

int main(int argc, int *argv[])
{
    a++;
    func_0();
    a++;
    func_0();
}
```

## Static

*persists between function calls*

```
#include <stdlib.h>

void func_0(void)
{
    static int64_t a = 1;
    a++;
    printf("%d\n", a); // prints 2, then 3
}

int main(int argc, int *argv[])
{
    static int64_t a = 2; // different 'a'
    func_0();
    func_0();
}
```

# Sections of the Address Space: data

- Static and global variables
- Live **as long as** the entire program

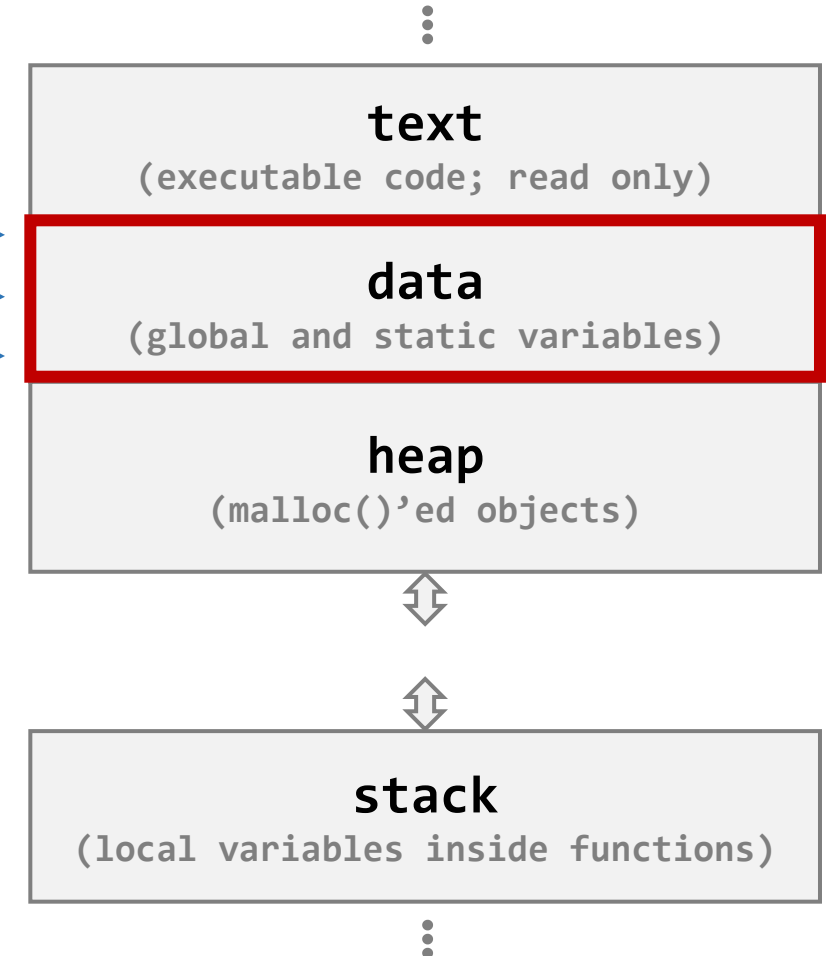
```
uint64_t data2[100];

int main(int argc, char **argv)
{
    static uint64_t *data0 = NULL;
    ...
}

void func(void)
{
    static uint64_t *data1;
    ...
}
```

&data0 →  
&data1 →  
&data2 →

## Typical C Program's Address Space



# Sections of the Address Space: heap

- Any object created with `malloc()`
  - Lives **until it is free()'d**

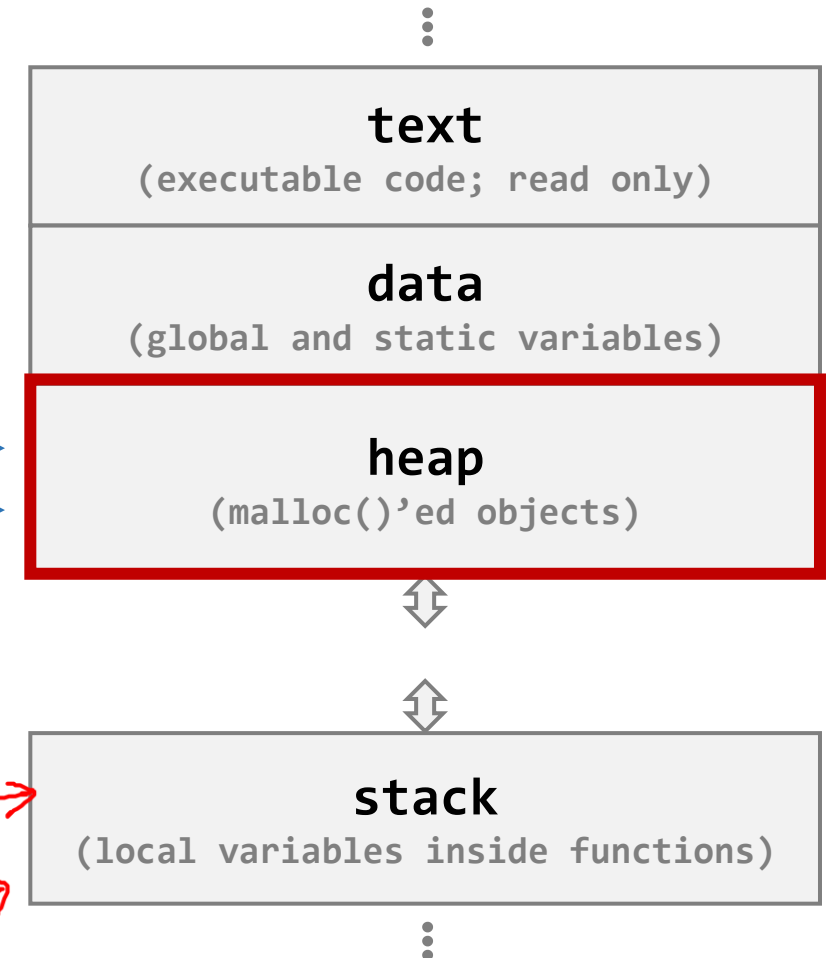
```
int main(int argc, char **argv)
{
    uint64_t *heap0 = (uint64_t *)malloc(sizeof(uint64_t));
    uint64_t *heap1 = (uint64_t *)malloc(sizeof(uint64_t) * N);

    *heap0 = 1;
    heap1[N - 1] = 0x211;

    free(heap0);
    free(heap1);
}
```

heap0 →  
heap1 →

## Typical C Program's Address Space



# Inspecting Sections of an Executable File

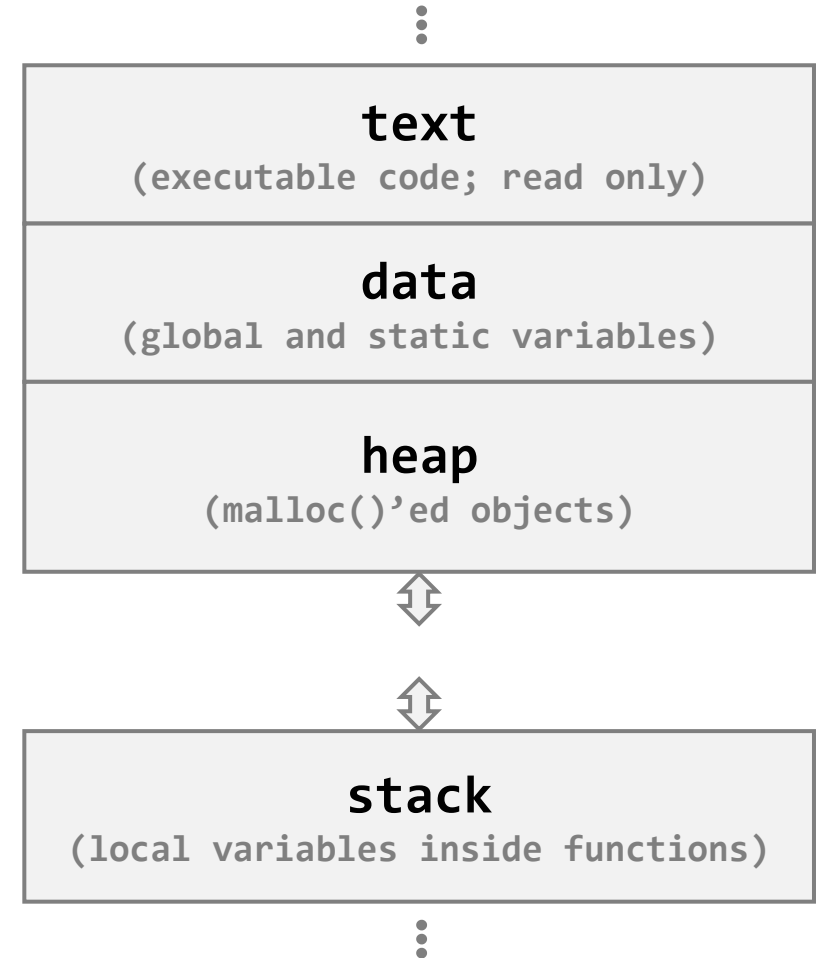
```
mp2099@lab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/to
olchain_glibc2/bin/riscv64-unknown-linux-gnu-gcc -o hello hello.c
mp2099@lab1:~/cs211/experiment$ readelf -S hello
There are 28 section headers, starting at offset 0x1a88:

Section Headers:
[Nr] Name              Type             Address           Offset
     Size             EntSize          Flags   Link Info  Align
[ 0] 0000000000000000    NULL            0000000000000000 0 0 0 0
[ 1] .interp             PROGBITS        000000000010270 0 0 1 0
     0000000000000020 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag      NOTE            000000000010290 0 0 4 0
     0000000000000020 0000000000000000 A 0 0 4
[ 3] .hash              HASH            0000000000102b0 0 0 8 0
     0000000000000024 0000000000000004 A 5 0 8
[ 4] .gnu.hash          GNU_HASH        0000000000102d8 0 0 8 0
     0000000000000030 0000000000000000 A 5 0 8
[ 5] .dynsym            DYNSYM          000000000010308 0 0 8 0
     0000000000000060 0000000000000018 A 6 1 8
[ 6] .dynstr            STRTAB          000000000010368 0 0 1 0
     000000000000004a 0000000000000000 A 0 0 1
[ 7] .gnu.version       VERSYM          0000000000103b2 0 0 2 0
     0000000000000008 0000000000000002 A 5 0 2
[ 8] .gnu.version_r     VERNEED        0000000000103c0 0 0 8 0
     0000000000000030 0000000000000000 A 6 1 8
[ 9] .rela.plt          RELA            0000000000103f0 0 0 8 0
     0000000000000030 0000000000000018 AI 5 20 8
[10] .plt               PROGBITS        000000000010420 0 0 16 0
     0000000000000040 0000000000000010 AX 0 0 16
[11] .text              PROGBITS        000000000010460 0 0 4 0
     00000000000000c2 0000000000000000 AX 0 0 4
[12] .rodata            PROGBITS        000000000010528 0 0 8 0
     0000000000000015 0000000000000000 A 0 0 8
[13] .eh_frame_hdr     PROGBITS        000000000010540 0 0 4 0
     0000000000000024 0000000000000000 A 0 0 4
[14] .eh_frame          PROGBITS        000000000010568 0 0 8 0
     000000000000006c 0000000000000000 A 0 0 8
[15] .preinit_array    PREINIT_ARRAY  000000000010d40 0 0 1 0
     0000000000000008 0000000000000008 WA 0 0 1
[16] .init_array        INIT_ARRAY      000000000010d48 0 0 8 0
     0000000000000008 0000000000000008 WA 0 0 8
[17] .fini_array        FINI_ARRAY      000000000011e00 0 0 8 0
     0000000000000008 0000000000000008 WA 0 0 8
[18] .dynamic           DYNAMIC         000000000011e08 0 0 8 0
     00000000000001e0 0000000000000010 WA 6 0 8
[19] .got               PROGBITS        000000000011fe8 0 0 8 0
     0000000000000008 0000000000000008 WA 0 0 8
[20] .got.plt           PROGBITS        000000000011ff0 0 0 8 0
     0000000000000020 0000000000000008 WA 0 0 8
[21] .sdata             PROGBITS        000000000012010 0 0 8 0
     0000000000000008 0000000000000000 WA 0 0 8
[22] .bss               NOBITS          000000000012018 0 0 1 0
     0000000000000008 0000000000000000 WA 0 0 1
[23] .comment           PROGBITS        000000000012018 0 0 1 0
     000000000000002b 0000000000000001 MS 0 0 1
[24] .riscv.attributes RISCV_ATTRIBUTE 0000000000000000 0 0 1 0
     0000000000000066 0000000000000000 0 0 1
[25] .symtab            SYMTAB          0000000000000000 0 0 8 0
     0000000000000078 0000000000000018 26 52 8
[26] .strtab            STRTAB          0000000000000000 0 0 1 0
     0000000000000262 0000000000000000 0 0 1
[27] .shstrtab          STRTAB          0000000000000000 0 0 1 0
     00000000000000fc 0000000000000000 0 0 1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), p (processor specific)
mp2099@lab1:~/cs211/experiment$ |
```

- Clearly, there's more
- Extra sections for
  - Initialization
  - Debugging (e.g., GDB)
  - Metadata (e.g., supported platforms)
- Sub-sections of data
  - e.g., read-only
  - Uninitialized

## Typical C Program's Address Space





# Agenda

- Functions
- Loaders and the C Memory Layout
- **Dynamic Memory Allocation**
- C Generics
  - C Library Functions

# Malloc

An **unsigned integer** type: wide enough to count the entire memory's worth of bytes

## 7.22.3.4 The malloc function

### Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

### Description

The **malloc** function allocates space for an object whose size is specified by **size** and whose value is indeterminate.

## 7.22.3.3 The free function

### Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

### Description

The **free** function causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs.

ISO Standard 9899:2011

- Allocates the requested number of **uninitialized bytes**
  - Malloc **does not care** how you will use the allocated memory
  - Assuming 'size' is very error-prone: use "**sizeof(type) \* N**"
- Returns NULL on failure (e.g., run out of memory): **always check!**

# Dynamic Memory Allocation

- You can dynamically allocate anything that you can statically allocate

## Basic Objects

```
uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
if(u != NULL)
{
    *u = 1;
    u[0]++;
    free(u);
}
```

## Array Objects

```
#define N (2 * 1024 * 1024)
uint64_t *arr = (uint64_t *)malloc(sizeof(uint64_t) * N);
if(arr != NULL)
{
    *arr = 0;
    arr[N - 1] = N - 1;
    free(u);
}
```

# Malloc'ing Pointers

- Dynamically allocating **a pointer** requires a pointer-to-pointer

```
struct my_struct
{
    uint8_t u;
    ...
}

// pointer to a pointer-to-struct
struct my_struct **pps = (struct my_struct **)malloc(sizeof(struct my_struct*));
if(pps != NULL)
{
    // pointer-to-struct
    *pps = (struct my_struct *)malloc(sizeof(struct my_struct));
    if(*pps != NULL)
    {
        (*pps).u = 0;
        (*pps)->u = 0;
        free(*pps);
    }
    free(pps); // one free() for every successful malloc()
}
```

# Friends of malloc()

```
malloc(3)                Library Functions Manual                malloc(3)

NAME
    malloc, free, calloc, realloc, reallocarray - allocate and free
    dynamic memory

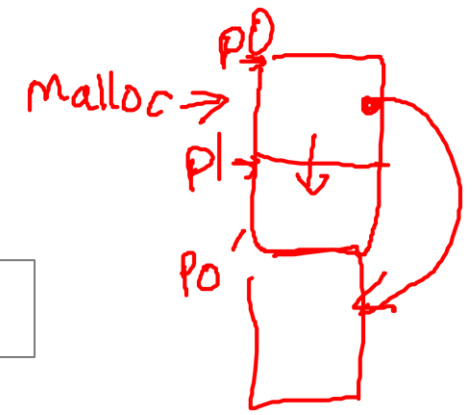
LIBRARY
    Standard C library (libc, -lc)

SYNOPSIS
    #include <stdlib.h>

    void *malloc(size_t size);
    void free(void *ptr);
    void *calloc(size_t nmemb, size_t size);
    void *realloc(void *ptr, size_t size);
    void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

- Helper functions:
  - **calloc()**: also initialize the allocated bytes to '0'
  - **realloc()**: change the size of a previously-malloc'd region

# Realloc for Resizing



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

- Resizes a previously-allocated block at **ptr** to a new **size**
  - Might require moving the block to a new location!
  - Automatically copies the data and free's the old block, if necessary
- On success:
  - New pointer is **valid**
  - Old pointer is **invalid** – do not try to use or free() it
- On failure:
  - New pointer is **NULL**
  - Old pointer is unchanged

```
void func(void)
{
    uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
    ...
    uint64_t *v = realloc(u, sizeof(uint64_t) * 2);
}
```

# Four Common Bugs with Malloc

All can cause a **program crash or security vulnerability**

## Memory Leak

```
void func(void)
{
    uint64_t *u = (uint64_t *)
        malloc(sizeof(uint64_t));
    ...
    return;
}
```

## Use After Free

```
void func(void)
{
    uint64_t *u = (uint64_t *)
        malloc(sizeof(uint64_t));
    free(u);
    u[0] = 1;
}
```

## Double Free

```
void func(void)
{
    uint64_t *u = (uint64_t *)
        malloc(sizeof(uint64_t));
    free(u);
    free(u);
}
```

## Incorrect Allocation Size

```
void func(void)
{
    uint64_t *u = (uint64_t *)
        malloc(1);
    u[0] = 0xffffffffffffffff;
}
```

# Example: Use After Free

- Your code will usually crash **some time later** than the actual bug

```
void func(void)
{
    uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
    if(u != NULL)
        free(u);

    ... // meanwhile, 'u' was reallocated to a different object

    printf("%lx", u); // prints data from somewhere else in the program
}
```

```
void func(void)
{
    uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
    if(u != NULL)
        free(u);

    ... // meanwhile, 'u' was NOT reallocated

    printf("%lx", u); // crash trying to access memory at 'u'
}
```



# Example: Double-Free

- Your code will usually crash **some time later** than the actual bug

```
void func(void)
{
    uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
    if(u != NULL)
        free(u);

    ... // meanwhile, 'u' was reallocated to a different object

    if(u != NULL)
        free(u); // free's somebody else's memory

    ... // bad things happen: the other object will experience a use-after-free
}
```

# Example: realloc() misuse

- Your code will usually crash **some time later** than the actual bug

```
void func(void)
{
    uint64_t *u = (uint64_t *)malloc(sizeof(uint64_t));
    ...
    uint64_t *v = (uint64_t *)realloc(u, sizeof(uint64_t) * 2);
    // IF realloc failed, v == NULL; u == still valid
    // IF realloc passed, v == new memory; u == invalid

    ... // need to be careful to use the right pointer!
}
```

# Avoiding Memory Bugs

## Proactive measures:

1. Always initialize pointers to NULL
2. Always set pointers to NULL after free() or successful realloc()
3. Always check the results of malloc() and friends
4. Avoid magic numbers in the code for allocation/array sizes

## Debugging tools:

1. **GDB:** line-by-line debugging
2. **Valgrind:** slows down your code, but checks for memory misuse
  - Memory leaks
  - Use-after-free
  - Out of bounds accesses

# **CS 211: Intro to Computer Architecture**

## ***7.1: Dynamic Memory Management***

**Minesh Patel**

Spring 2025 – Tuesday 4 March