

CS 211: Intro to Computer Architecture

6.1: C Data Representation III: Derived Types Cont.

Minesh Patel

Spring 2025 – Tuesday 25 February

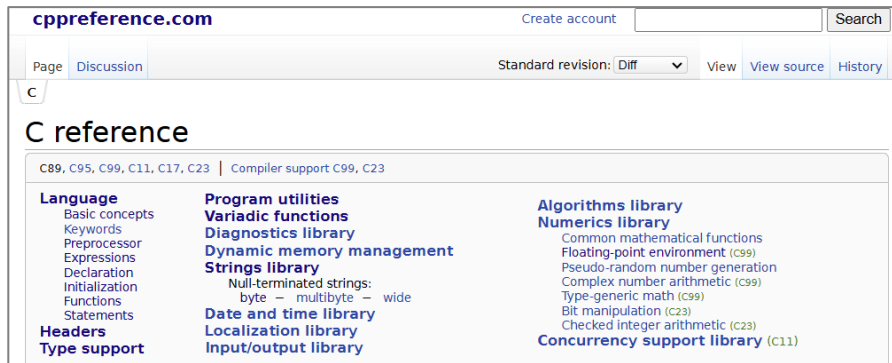
Announcements

- **PA2** due Wednesday, Feb 23 @ 23:59
 - We will have a post-PA2 survey during the next class
- **PA3** and **WA4** to be released at the end of the week

Reference Material

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

And Various C and Linux Reference Materials



cppreference.com

Create account Search

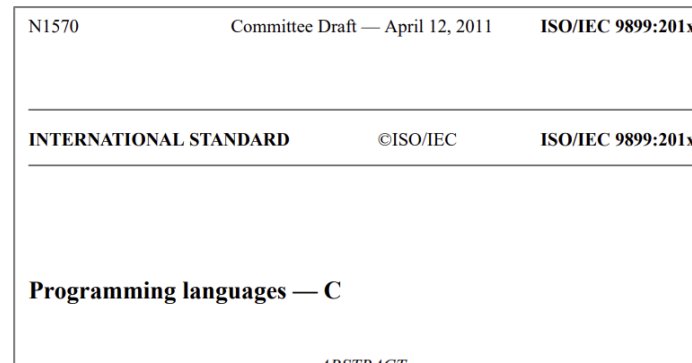
Page Discussion Standard revision: Diff View View source History

C

C reference

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

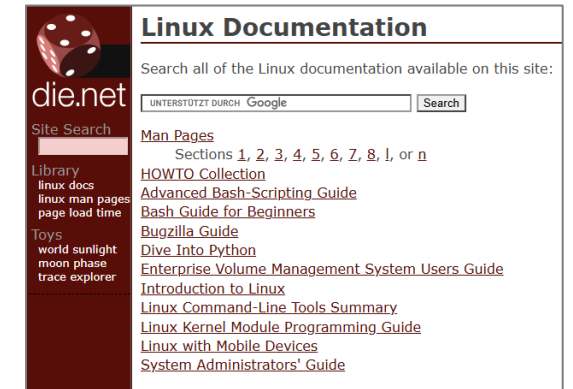
Language <ul style="list-style-type: none">Basic conceptsKeywordsPreprocessorExpressionsDeclarationInitializationFunctionsStatements	Program utilities <ul style="list-style-type: none">Variadic functionsDiagnostics libraryDynamic memory managementStrings library<ul style="list-style-type: none">Null-terminated strings:<ul style="list-style-type: none">byte – multibyte – wideDate and time libraryLocalization libraryInput/output library	Algorithms library <ul style="list-style-type: none">Numerics library<ul style="list-style-type: none">Common mathematical functionsFloating-point environment (C99)Pseudo-random number generationComplex number arithmetic (C99)Type-generic math (C99)Bit manipulation (C23)Checked integer arithmetic (C23)Concurrency support library (C11)
--	---	--



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

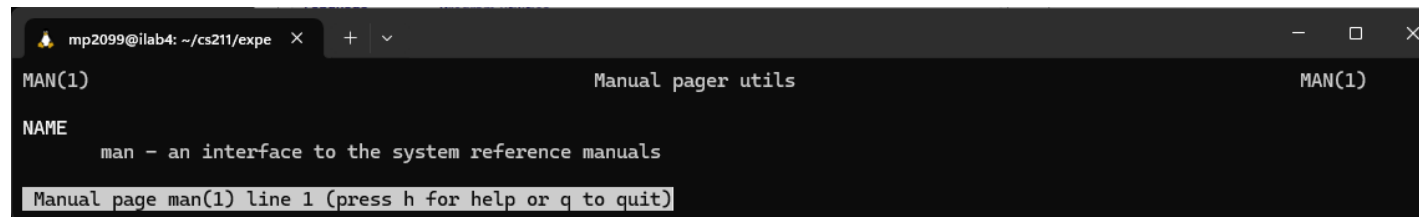
Site Search

Man Pages
Sections 1, 2, 3, 4, 5, 6, 7, 8, l, or n

Library
linux docs
linux man pages
page load time

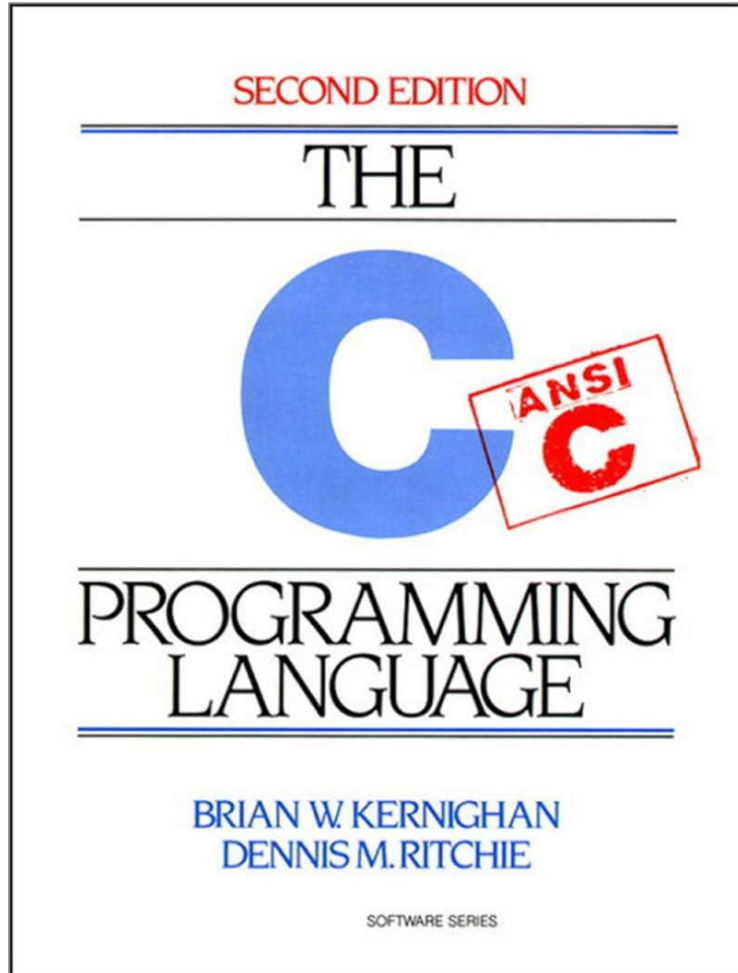
Toys
world
sunlight
moon phase
trace explorer

- [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](#)

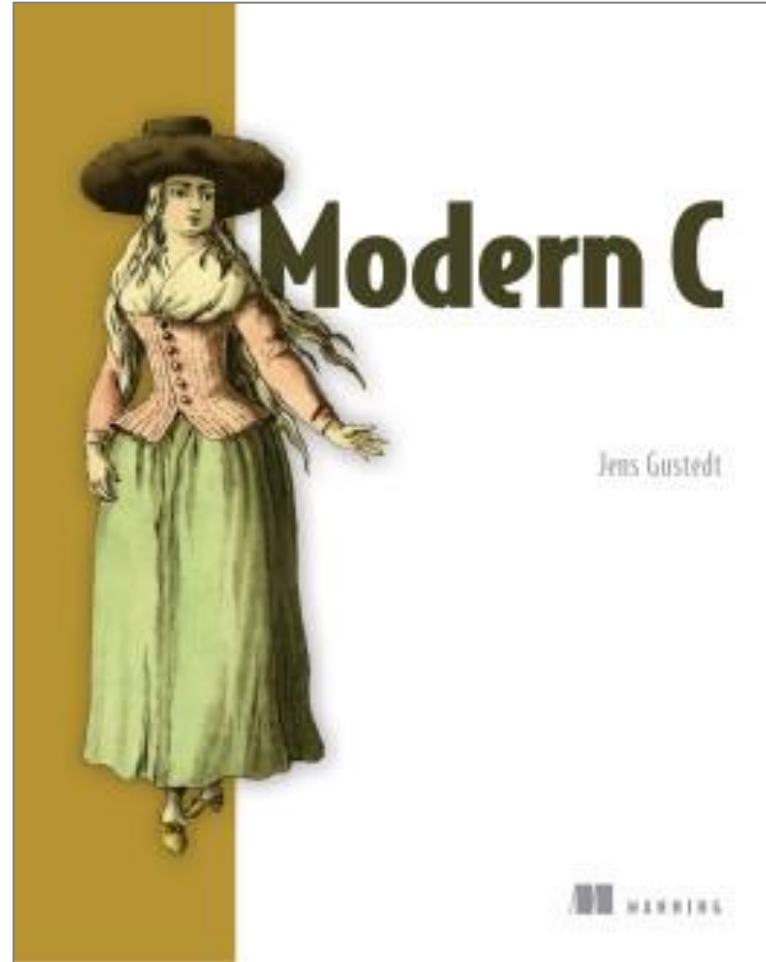


```
mp2099@ilab4: ~/cs211/expe x + v
MAN(1) Manual pager utils MAN(1)
NAME
    man - an interface to the system reference manuals
Manual page man(1) line 1 (press h for help or q to quit)
```

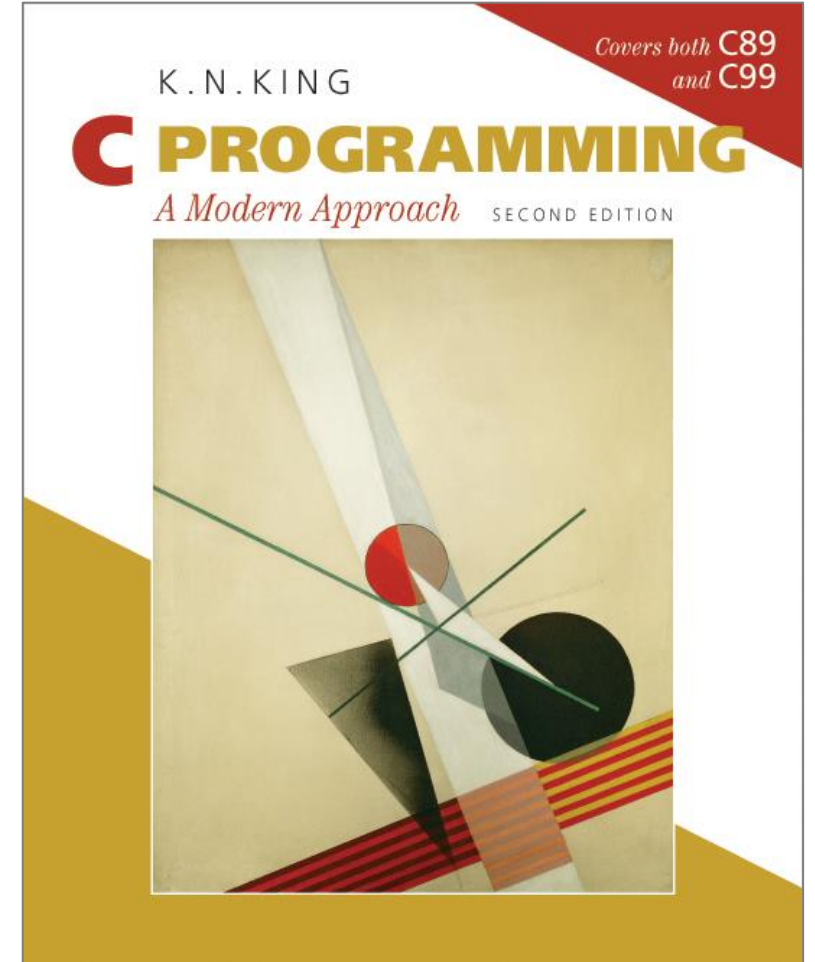
Standards are Great, But...



https://m.media-amazon.com/images/I/51EyaJeebHL._SL1056_.jpg



<https://www.manning.com/books/modern-c>



<http://knking.com/books/c2/cover.html>

Recommended GCC Warning Options

`-Wall`
`-Wextra`

Enables additional (mostly useful) warnings

`-Werror`

Treat all warnings as errors (don't create the executable)

`-Wshadow`

Warns when you use the same name for variables in different scopes

`-pedantic`

Follow the C standard strictly (often helpful, but very... pedantic)

`-Wenum-compare`

Warn about a comparison between values of different enumerated types. In C++ enumerated type mismatches in conditional expressions are also diagnosed and the warning is enabled by default. In C this warning is enabled by `-Wall`.

`-Wenum-conversion`

Warn when a value of enumerated type is implicitly converted to a different enumerated type. This warning is enabled by `-Wextra` in C.

Example of Variable Shadowing

```
void func(int a)
{
    for(uint32_t a = 0; a < 10; a++)
    {
        printf("%d", a); // which a?
    }
}
```

Agenda

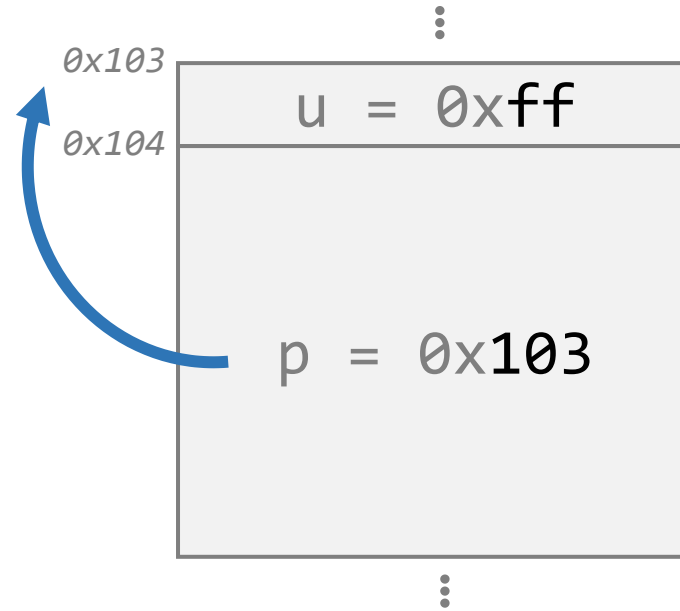
- void
- basic types
 - char
 - signed integers
 - unsigned integers
 - floating-point
- enumerated types
- **derived types**
 - structures
 - **pointers**
 - arrays
 - unions
 - functions

• **Pointers**

- **Basics (continued)**
 - Why pointers?
 - Pointer nuances
-
- Arrays and Strings
 - Array nuances

Recap: Pointers “Point” to Other Objects

```
uint8_t u = 0xff;  
uint8_t *p = &u;
```



Pointer types are **objects in memory**

Pointers contain the **memory address** of other objects

Pointer Example

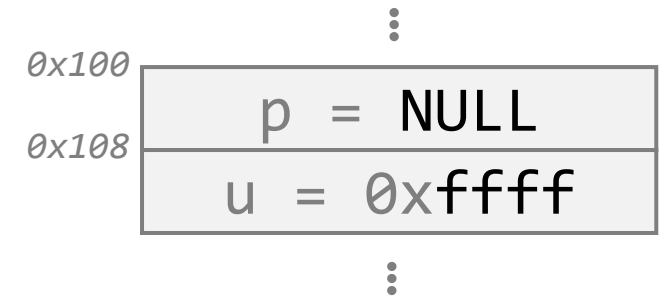
```
#include <stdlib.h>

void func(void)
{
    uint16_t *p = NULL; // NULL = pointing nowhere
    uint16_t u = 0xffff;

    p = &u;
    printf("%p %x %x\n", p, *p, u);

    u >>= 4;
    printf("%p %x %x\n", p, *p, u);

    *p >>= 4;
    printf("%p %x %x\n", p, *p, u);
}
```



- Declare an **identifier** p
 - Type “**pointer to uint16_t**”
 - Initialized to **NULL**
 - i.e., pointing to “nothing”

Pointer Example

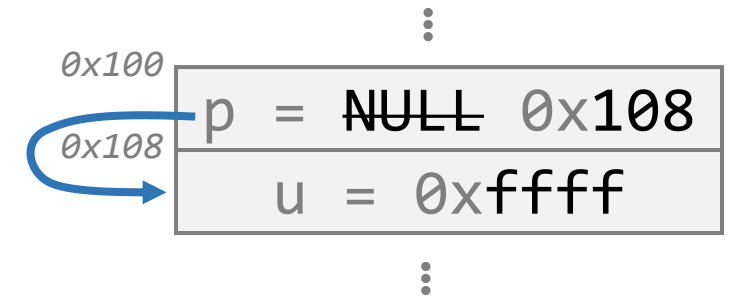
```
#include <stdlib.h>

void func(void)
{
    uint16_t *p = NULL; // NULL = invalid address
    uint16_t u = 0xffff;

    p = &u;
    printf(“%p %x %x\n”, p, *p, u);

    u >>= 4;
    printf(“%p %x %x\n”, p, *p, u);

    *p >>= 4;
    printf(“%p %x %x\n”, p, *p, u);
}
```



- Declare an **identifier** `p`
 - Type “**pointer to uint16_t**”
 - Initialized to `NULL`
 - i.e., pointing to “nothing”
- Set `p` to the **address of u**
 - “`p` points to `u`”
- Print the **value** pointed to by `p`
 - `*p`: “**dereferencing** the pointer”

Pointer Example

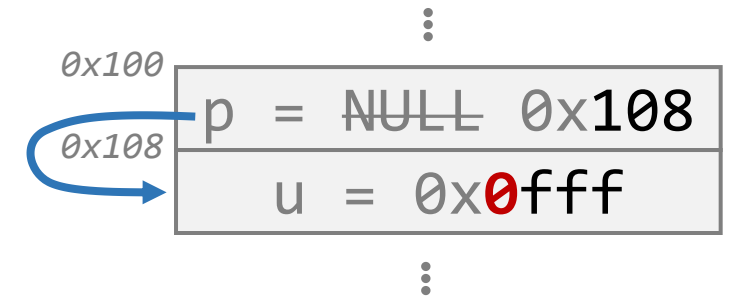
```
#include <stdlib.h>

void func(void)
{
    uint16_t *p = NULL; // NULL = invalid address
    uint16_t u = 0xffff;

    p = &u;
    printf(“%p %x %x\n”, p, *p, u);

    u >>= 4;
    printf(“%p %x %x\n”, p, *p, u);

    *p >>= 4;
    printf(“%p %x %x\n”, p, *p, u);
}
```



- Declare an **identifier** p
 - Type “**pointer to uint16_t**”
 - Initialized to **NULL**
 - i.e., pointing to “nothing”
- Set p to the **address of u**
 - “p points to u”
- Print the **value** pointed to by p
 - *p: “**dereferencing** the pointer”
- P still points to u

Pointer Example

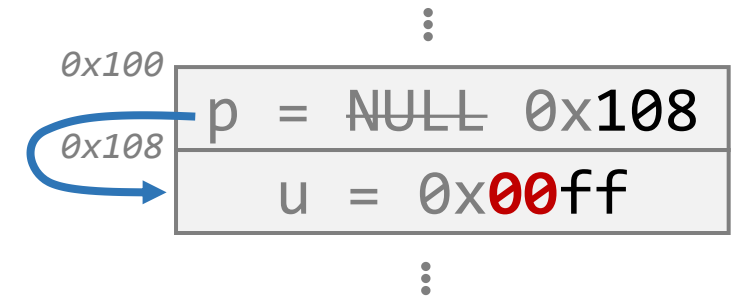
```
#include <stdlib.h>

void func(void)
{
    uint16_t *p = NULL; // NULL = invalid address
    uint16_t u = 0xffff;

    p = &u;
    printf(“%p %x %x\n”, p, *p, u);

    u >>= 4;
    printf(“%p %x %x\n”, p, *p, u);

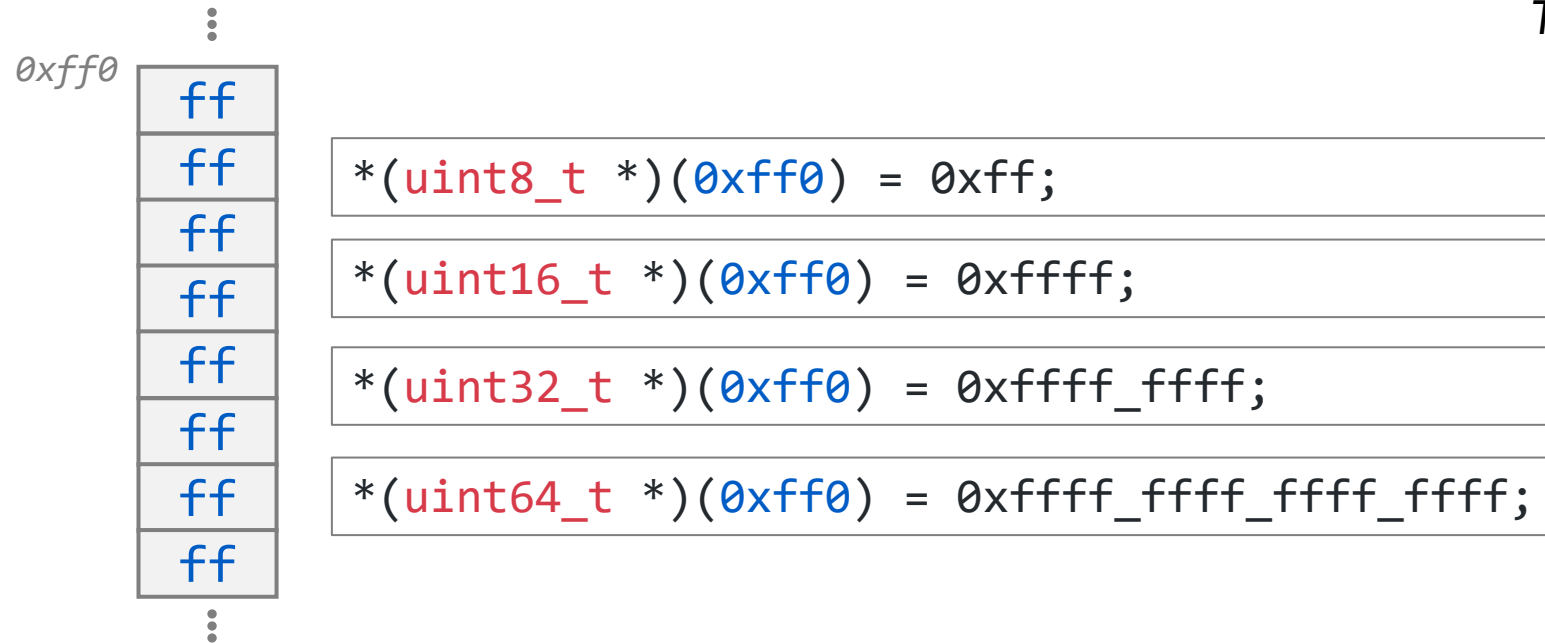
    *p >>= 4;
    printf(“%p %x %x\n”, p, *p, u);
}
```



- Declare an **identifier** p
 - Type “**pointer to uint16_t**”
 - Initialized to **NULL**
 - i.e., pointing to “nothing”
- Set p to the **address of u**
 - “p points to u”
- Print the **value** pointed to by p
 - *p: “**dereferencing** the pointer”
- P still points to u
- Changes the **value pointed to** by p

Interpreting Memory Locations

- Memory is just an **array of bytes**
- A memory location can hold **any type of object**



This will compile + run, but reading from invalid memory locations is **undefined behavior**

```
mp2099@ilab3:~/cs211/experiment$ cat pointer2.c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int main(int argc, char *argv[])
{
    printf("%x\n", *(uint8_t *)0x102);

    return EXIT_SUCCESS;
}
mp2099@ilab3:~/cs211/experiment$ /common/system/riscvi
/bin/riscv64-unknown-elf-gcc -o pointer2 pointer2.c
mp2099@ilab3:~/cs211/experiment$ ./pointer2
Segmentation fault
mp2099@ilab3:~/cs211/experiment$ |
```

- **Types** tell the compiler how to interpret those bytes
 - Beware of endianness ☺

Pointer Example 2

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

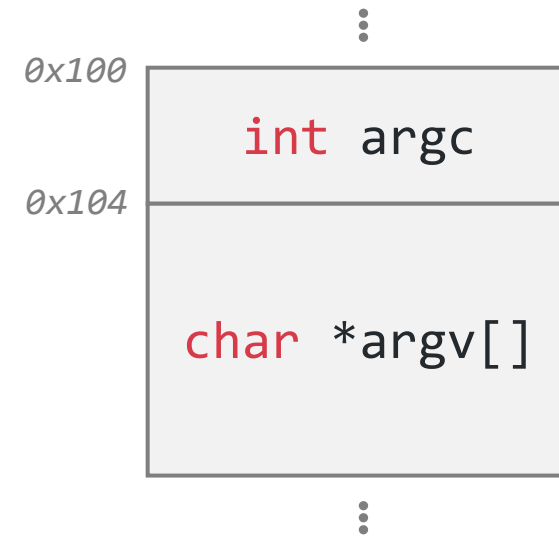
int main(int argc, char *argv[])
{
    uint8_t u8 = 0xff;

    uint8_t *p_u8 = &u8;
    printf("as int: %d @ %p\n", (int)*p_u8, p_u8);

    int8_t *p_i8 = (int8_t *)p_u8;
    printf("as int: %d @ %p\n", (int)*p_i8, p_i8);

    return EXIT_SUCCESS;
}
```

Memory



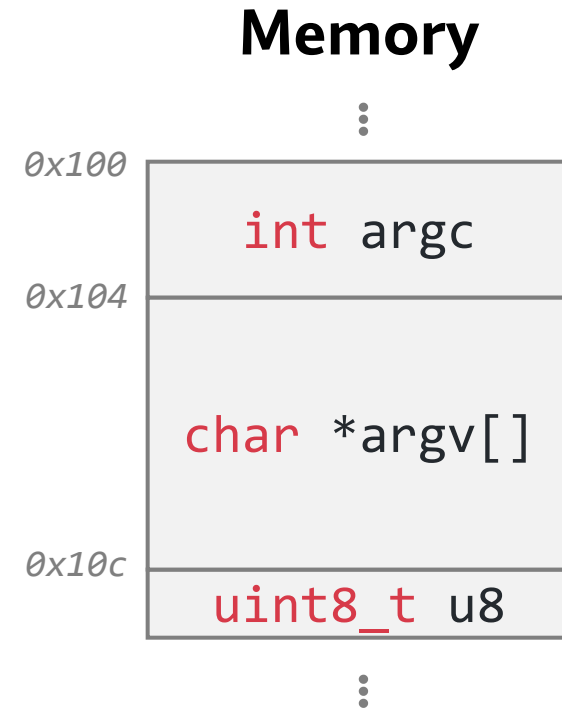
Pointer Example 2

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

int main(int argc, char *argv[])
{
    uint8_t u8 = 0xff;
    ← uint8_t *p_u8 = &u8;
    printf("as int: %d @ %p\n", (int)*p_u8, p_u8);

    int8_t *p_i8 = (int8_t *)p_u8;
    printf("as int: %d @ %p\n", (int)*p_i8, p_i8);

    return EXIT_SUCCESS;
}
```



Pointer Example 2

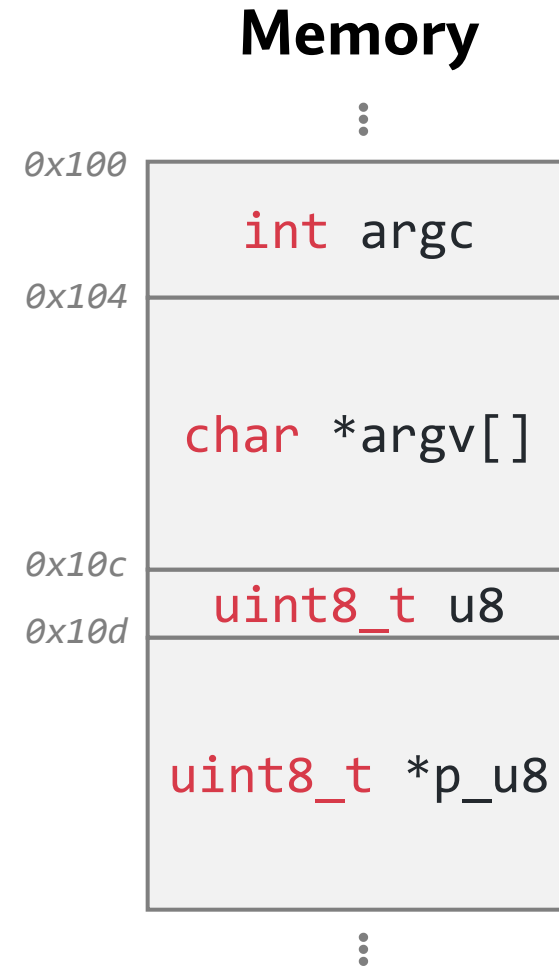
```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int main(int argc, char *argv[])
{
    uint8_t u8 = 0xff;

    uint8_t *p_u8 = &u8;
    printf("as int: %d @ %p\n", (int)*p_u8, p_u8);

    int8_t *p_i8 = (int8_t *)p_u8;
    printf("as int: %d @ %p\n", (int)*p_i8, p_i8);

    return EXIT_SUCCESS;
}
```



```
mp2099@ilab1:~/cs211/experiment$ /common/system/riscvi/bin
/riscv64-unknown-elf-gcc -o pointer_types pointer_types.c
mp2099@ilab1:~/cs211/experiment$ ./pointer_types
as int: 255 @ 0x400080030f
```

Pointer Example 2

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

int main(int argc, char *argv[])
{
    uint8_t u8 = 0xff;

    uint8_t *p_u8 = &u8;
    printf("as int: %d @ %p\n", (int)*p_u8, p_u8);

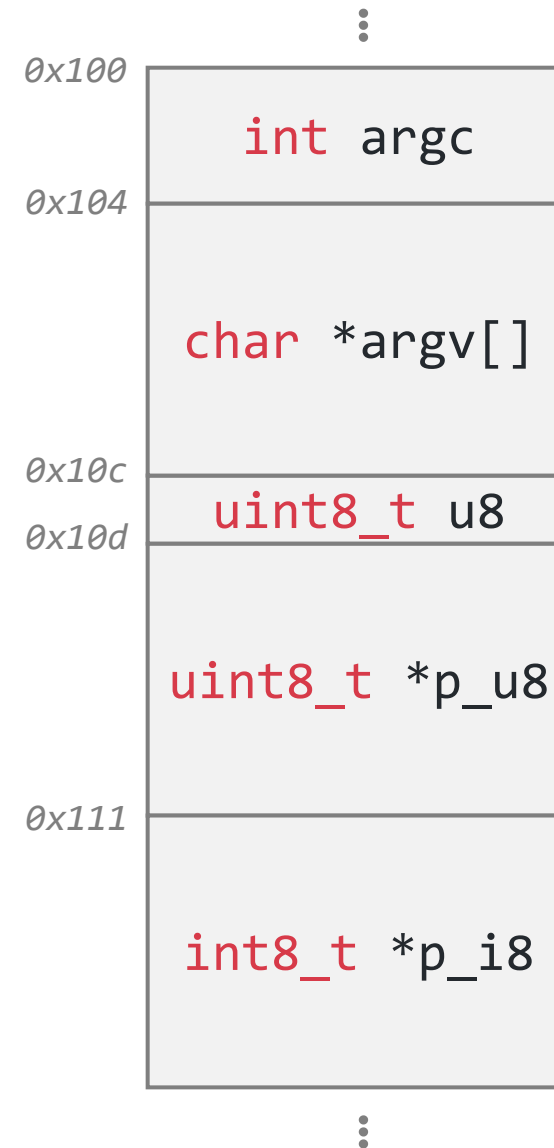
    int8_t *p_i8 = (int8_t *)p_u8;
    printf("as int: %d @ %p\n", (int)*p_i8, p_i8);

    return EXIT_SUCCESS;
}
```



```
mp2099@ilab1:~/cs211/experiment$ /common/system/riscvi/bin
/riscv64-unknown-elf-gcc -o pointer_types pointer_types.c
mp2099@ilab1:~/cs211/experiment$ ./pointer_types
as int: 255 @ 0x400080030f
as int: -1 @ 0x400080030f
```

Memory



Agenda

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

• **Pointers**

- Basics (continued)
 - **Why pointers?**
 - Pointer nuances
-
- Arrays and Strings
 - Array nuances

Passing Parameters in C

- C function arguments are **always copies** rather than the **original object**
 - **Call by value:** arguments are copies of the original object(s) ← **C**
 - **Call by reference:** arguments refer to the original object(s) ← **Not C**

Modifying Copies of {a, b}

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

void func(void)
{
    int a = 0, b = 1;
    swap(a, b); // does nothing
}
```

Copy of an int object

Modifying the Original {a, b}

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void func(void)
{
    int a = 0, b = 1;
    int *p_a = &a, *p_b = &b;
    swap(p_a, p_b); // swaps a and b
}
```

Copy of an int * object

Pointer Use-Case 1: Passing Parameters

- **Mutability:** pointers emulate passing by reference

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

- **Performance:** Pointers let us avoid copying gigantic objects every time

```
void func_0(struct big_data s);
```

Copying: entire object (e.g., 100 GB)

```
void func_0(struct big_data *s);
```

Copying: pointer object (e.g., 8B)

Pointer Use-Case 2: Memory Management

- So far, we've relied on the compiler to **create/destroy objects**

Static Memory Allocation

```
void func(int print)
{
    if(print != 0)
    {
        uint32_t u = 0; // compiler creates object 'u'
        printf("%d", u); // we use the object
    } // compiler destroys 'u' (out of scope)
}
```

- Unfortunately, we don't always know our objects at **compile time**
 - Maybe dependent on **runtime inputs** (e.g., modify a data structure, read a file)
 - Maybe we don't know an **object's type**

Pointer Use-Case 2: Dynamic Memory Management

- Dedicated functions `malloc()` / `free()` to create/destroy objects of size N

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

Description

2 The `malloc` function allocates space for an object whose size is specified by `size`

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

Description

The `free` function causes the space pointed to by `ptr` to be deallocated

ISO Standard 9899:201x

Static Memory Allocation

```
void func(void)
{
    // compiler allocates the object
    uint32_t u;

    // use the object
    u = 0;
    u = u + 1;

} // compiler destroys u (out of scope)
```

Dynamic Memory Allocation

```
void func(void)
{
    // programmer allocates object
    uint64_t *p = (uint64_t *)malloc(sizeof(uint64_t));

    // use your object
    *p = 0;
    *p = *p + 1;

    free(p); // programmer destroys object

}
```

Pointer Use-Case 3: Low-Level Programming

- Many CPU features are configured via **hardwired memory locations**
 - Initializing (booting) the system
 - Configuring hardware/software policies
 - Interfacing with devices (e.g., accelerators, peripherals, I/O)
 - ...

X86 “System Address Map” on Bootup

start	end	size	description	type	
Real mode address space (the first MiB)					
0x00000000	0x000003FF	1 KiB	Real Mode IVT (Interrupt Vector Table)	unusable in real mode	640 KiB RAM ("Low memory")
0x00000400	0x000004FF	256 bytes	BDA (BIOS data area)		
0x00000500	0x00007BFF	29.75 KiB	Conventional memory	usable memory	
0x00007C00	0x00007DFF	512 bytes	Your OS BootSector		
0x00007E00	0x0007FFFF	480.5 KiB	Conventional memory		
0x00080000	0x0009FFFF	128 KiB	EBDA (Extended BIOS Data Area)	partially used by the EBDA	
0x000A0000	0x000BFFFF	128 KiB	Video display memory	hardware mapped	
0x000C0000	0x000C7FFF	32 KiB (typically)	Video BIOS	ROM and hardware mapped / Shadow RAM	384 KiB System / Reserved ("Upper Memory")
0x000C8000	0x000EFFFF	160 KiB (typically)	BIOS Expansions		
0x000F0000	0x000FFFFF	64 KiB	Motherboard BIOS		

Caution: Pointer Bugs

- Pointers are powerful, but also a huge source of real-world **bugs and vulnerabilities**

The screenshot shows the CVE website interface. At the top, there are navigation links for CVE List, CNAs, WGs, Board, About, and News. Below this is a search bar and several buttons: Search CVE List, Downloads, Data Feeds, Update a CVE Record, and Request CVE IDs. A banner indicates a total of 240,830 CVE records and provides notices about the new website and legacy download formats. The search results section shows 5,283 records matching the search term 'pointer'. A table lists several CVEs with their names and descriptions, highlighting the 'pointer' vulnerability in each.

Name	Description
CVE-2025-27113	libxml2 before 2.12.10 and 2.13.x before 2.13.6 has a NULL pointer dereference in xmlPatMatch in pattern.c.
CVE-2025-25475	A NULL pointer dereference in the component /libsrc/dclreccd.cc of DCMTK v3.6.9+ DEV allows attackers to cause a Denial of Service (DoS) via a crafted DICOM file.
CVE-2025-25473	FFmpeg git master before commit c08d30 was discovered to contain a NULL pointer dereference via the component libavformat/mov.c.
CVE-2025-25471	FFmpeg git master before commit fd1772 was discovered to contain a NULL pointer dereference via the component libavformat/mov.c.
CVE-2025-24483	NULL pointer dereference vulnerability exists in Defense Platform Home Edition Ver.3.9.51.x and earlier. If an attacker provides specially crafted data to the specific process of the Windows system where the product is running, the system may cause a Blue Screen of Death (BSOD), and as a result, cause a denial-of-service (DoS) condition.
CVE-2025-24177	A null pointer dereference was addressed with improved input validation. This issue is fixed in macOS Sequoia 15.3, iOS 18.3 and iPadOS 18.3. A remote attacker may be able to cause a denial-of-service.
CVE-2025-24031	PAM-PKCS#11 is a Linux-PAM login module that allows a X.509 certificate based user login. In versions 0.6.12 and prior, the pam_pkcs11 module segfaults when a user presses ctrl-c/ctrl-d when they are asked for a PIN. When a user enters no PIN at all, `pam_get_pwd` will never initialize the password buffer pointer and as such `cleanse` will try to dereference an uninitialized pointer . On my system this pointer happens to have the value 3 most of the time when running sudo and as such it will segfault. The most likely impact to a system affected by this issue is an availability impact due to a daemon that uses PAM crashing. As of time of publication, a patch for the issue is unavailable.
CVE-2025-24014	Vim is an open source, command line text editor. A segmentation fault was found in Vim before 9.1.1043. In silent Ex mode (-s -e), Vim typically doesn't show a screen and just operates silently in batch mode. However, it is still possible to trigger the function that handles the scrolling of a gui version of Vim by feeding some binary characters to Vim. The function that handles the scrolling however may be triggering a redraw, which will access the ScreenLines pointer , even so this variable hasn't been allocated (since there is no screen). This vulnerability is fixed in 9.1.1043.
CVE-2025-21697	In the Linux kernel, the following vulnerability has been resolved: drm/v3d: Ensure job pointer is set to NULL after job completion After a job completes, the corresponding pointer in the device must be set to NULL. Failing to do so triggers a warning when unloading the driver, as it appears the job is still active. To prevent this, assign the job pointer to NULL after completing the job, indicating the job has finished.
CVE-2025-21695	In the Linux kernel, the following vulnerability has been resolved: platform/x86: dell-uart-backlight: fix serdev race The dell_uart_bl_serdev_probe() function calls devm_serdev_device_open() before setting the client ops via serdev_device_set_client_ops(). This ordering can trigger a NULL pointer dereference in the serdev controller's receive_buf handler, as it assumes serdev->ops is valid when SERPORT_ACTIVE is set. This is similar to the issue fixed in commit 5e700b384ec1 ("platform/chrome: cros_ec_uart: properly fix race condition") where devm_serdev_device_open() was called before fully initializing the device. Fix the race by ensuring client ops are set before enabling the port via devm_serdev_device_open(). Note, serdev_device_set_baudrate() and serdev_device_set_flow_control() calls should be after the devm_serdev_device_open() call.
CVE-2025-21688	In the Linux kernel, the following vulnerability has been resolved: drm/v3d: Assign job pointer to NULL before signaling the fence In commit e4b5ccd392b9 ("drm/v3d: Ensure job pointer is set to NULL after job completion"), we introduced a change to assign the job pointer to NULL after completing a job, indicating job completion. However, this approach created a race condition between the DRM scheduler workqueue and the IRQ execution thread. As soon as the fence is signaled in the IRQ execution thread, a new job starts to be executed. This results in a race condition where the IRQ execution thread sets the job pointer to NULL simultaneously as the `run_job()` function assigns a new job to the pointer . This race condition can lead to a NULL pointer dereference if the IRQ execution thread sets the job pointer to NULL after `run_job()` assigns it to the new job. When the new job completes and the GPU emits an interrupt, `v3d_irq()` is triggered, potentially causing a crash. [466.310099] Unable to handle kernel NULL pointer dereference at virtual address 00000000000000c0 [466.318928] Mem abort info: [466.321723] ESR = 0x0000000096000005 [466.325479] EC = 0x25: DABT (current EL), IL = 32 bits [466.330807] SET = 0, Frv = 0 [466.333864] EA = 0, S1PTW = 0 [466.337010] FSC = 0x05: level 1 translation fault [466.341900] Data abort info: [466.344783] ISV = 0, ISS = 0x00000005, ISS2 = 0x00000000 [466.350285] CM = 0, WnR = 0, TagAccess = 0 [466.355350] GCS = 0, Overlay = 0, DirtyBit = 0, Xs = 0 [466.360677] user pgtable: 4k pages, 39-bit VAs, pgdp=0000000089772000 [466.367140] [0000000000000000] pgd=0000000000000000, p4d=0000000000000000, pud=0000000000000000 [466.375875] Internal error: Oops: 0000000096000005 [#1] PREEMPT SMP [466.382163] Modules linked in: rfcmm snd_seq_dummy snd_hrtimer snd_seq_device algif_hash algif_skcipher af_alg bnep binfmt_misc vc4 snd_soc_hdmi_codec drm_display_helper cec brcmfmac_wcc spidev rpidvid_hevc(C) drm_client_lib brcmfmac hci_uart drm_dma_helper pisp_be btbcm brcmutil snd_soc_core aes_ce_blk v4l2_mem2mem bluetooth aes_ce_cipher snd_compress videobuf2_dma_contig ghash_ce cfg80211 gf128mul

Common Pointer Bugs

Uninitialized Pointer (BAD)

```
void func(void)
{
    char *p;
    *p = 0x100; // undef behavior
}
```

Initialized Pointer (OK)

```
void func(void)
{
    char *p = NULL;
    *p = 0x100; // defined behavior (crash)
}
```

Dangling Pointer

```
void func(void)
{
    char *p;

    if(1)
    {
        char c = 'a';
        p = &c;
    }

    *p = 0x100; // undef behavior (object c is gone)
}
```


Agenda

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

• **Pointers**

- Basics (continued)
 - Why pointers?
 - **Pointer nuances**
-
- Arrays and Strings
 - Array nuances

Initializing Multiple Pointers

- C allows multiple declarations on one line

```
char a, b; // neither initialized
char a = 'a', b; // b uninitialized
char a = 'a', b = 'b'; // both initialized
```

- It gets messy and confusing with pointers

```
char *a, b; // b is NOT a pointer; neither are initialized
char *a = NULL, *b; // b uninitialized
char a = 0, *b = &a; // both initialized
```

- We recommend sticking to one line per declaration/initializer

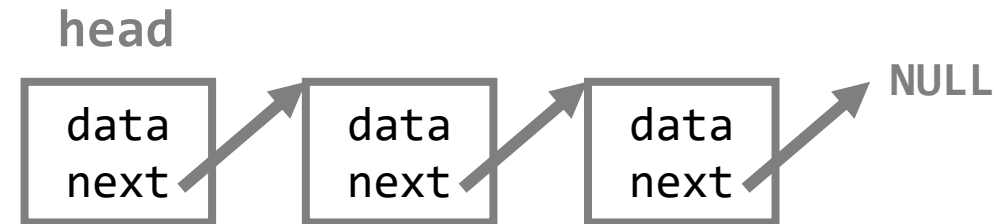
```
char *a = NULL,
      *b = NULL; // if you must
```

Pointers to Structs

- A pointer can point to **any C object** (`int`, `struct`, `enum`, pointer, etc.)

Singly-Linked List

```
struct node
{
    uint8_t data;
    struct node *next;
}
```



Passing a Struct By Value

```
void initialize_node(struct node n)
{
    new_node.data = 0;
    new_node.next = NULL;
} // n is destroyed here (original unmodified)
```

Passing a Struct By Pointer

```
void initialize_node(struct node* n)
{
    new_node->data = 0; // (*new_node).data
    new_node->next = NULL; // (*new_node).next
} // original n is modified
```

The Void Pointer

“p” is a pointer to an object of type “void”

```
void *p;
```

Cannot dereference a “void *”
(compiler doesn't know how to interpret
what is at the memory address)

```
void v = *p;
```

Type “void” is incomplete

6.3.2.3 Pointers

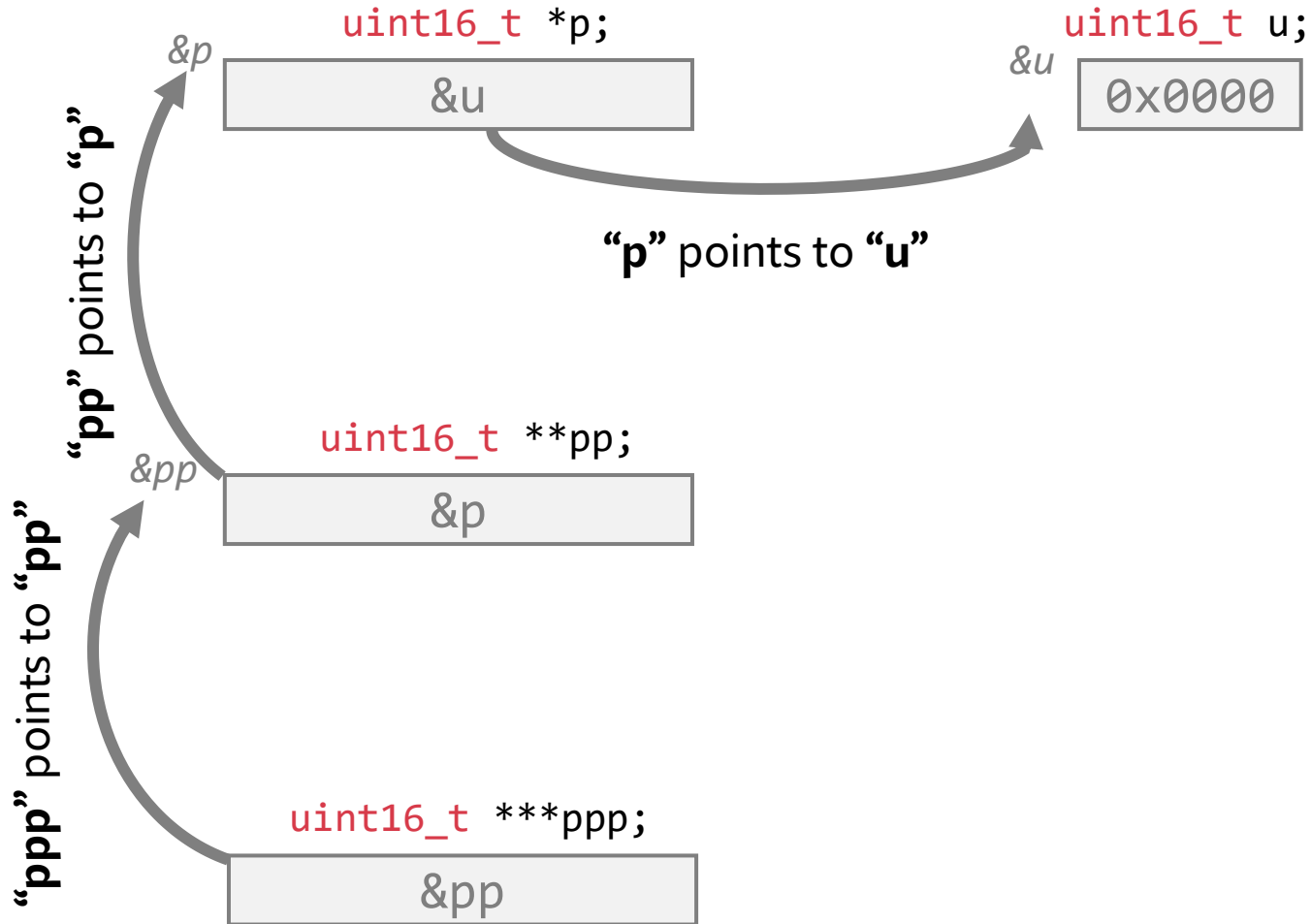
1 A pointer to `void` may be converted to or from a pointer to any object type.

ISO Standard 9899:201x (6.3.2.3)

- Implies we can convert **any pointer type** to **any other type**
 - Enables *reinterpreting* an object as a different type (**more on this later**)

```
uint8_t u = 0xff;  
struct node *p = (struct node *)&u;  
p->next = NULL; // very bad things happen!
```

Pointers to Pointers



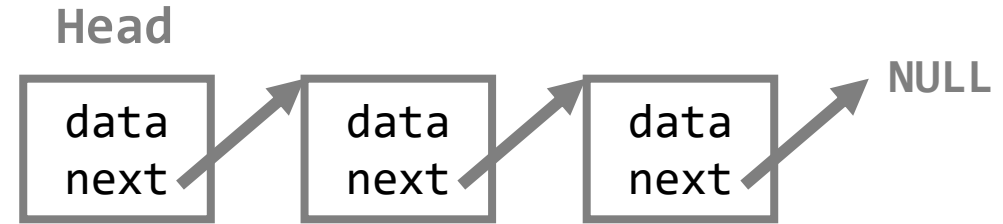
```
uint16_t u = 0;  
uint16_t *p = &u;
```

```
uint16_t **pp = &p;
```

```
uint16_t ***ppp = &pp;
```

Pointer-to-pointer Example: Singly Linked List

```
struct node
{
    uint8_t data;
    struct node *next;
}
```



```
struct node * create_node(void);

void insert_at_head(
    struct node **head, int data);

int main(int argc, char *argv[])
{
    struct node *head = create_node();
    insert_at_head(&head, 0);
    insert_at_head(&head, 1);
}
```

```
void insert_at_head(struct node **head, int data)
{
    // initialize the new node
    struct node *new_node = create_node();
    new_node->data= data;
    new_node->next = *head;

    // update head
    head->next = *head;
    *head = new_node;
}
```

Agenda

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

- **Pointers**
 - Basics (continued)
 - Why pointers?
 - Pointer nuances

- **Arrays and Strings**
 - Array nuances

Arrays and Strings

- C arrays are **objects in memory**
 - An **array object** contains a sequential list of some other type of object
 - No metadata for array length

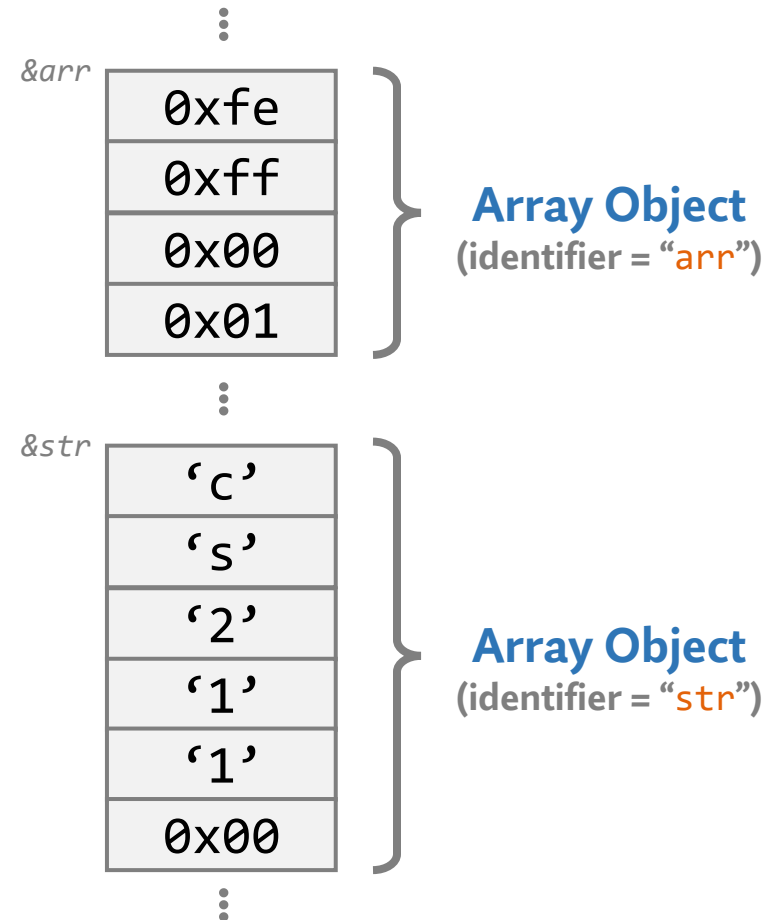
```
void func(void)
{
    int8_t arr[4] = {-2, -1, 0, 1};

    // some code
}
```

```
void func(void)
{
    char str[] = "cs211";

    // some code
}
```

C strings are just
arrays of characters
(with a null terminator)

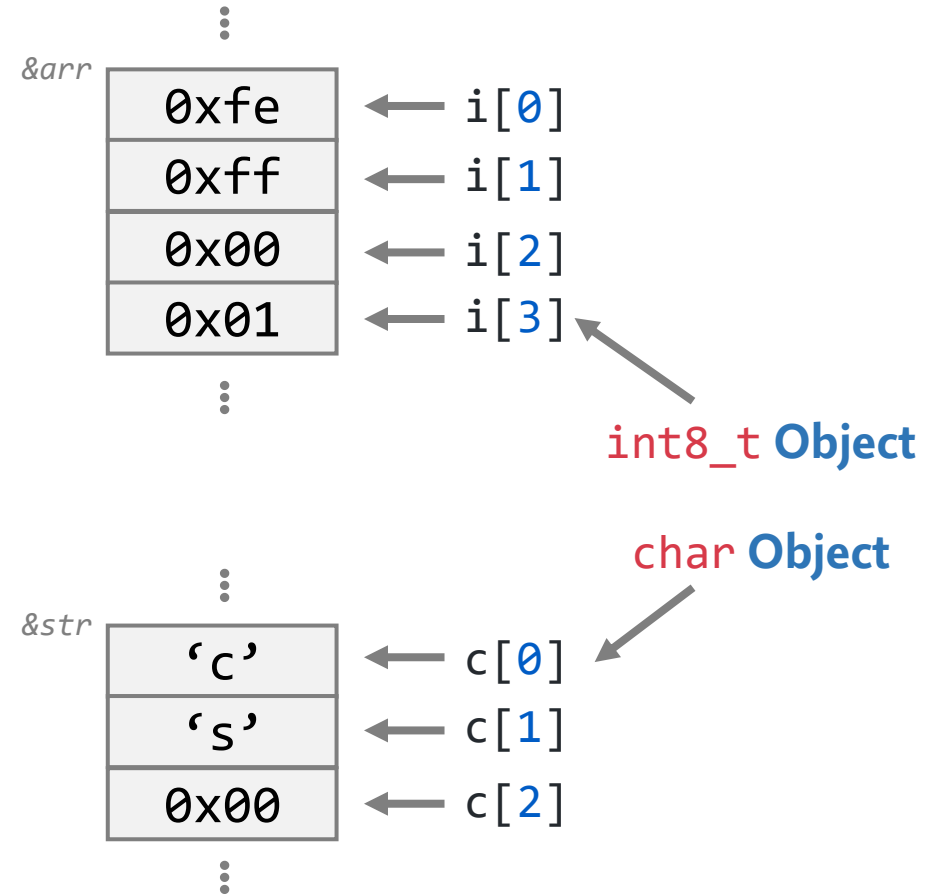


Indexing C Arrays

- Array indexing is very straightforward

```
void func(void)
{
    int8_t i[4] = {-2, -1, 0, 1};
    printf("%d%d%d%d", i[0], i[1], i[2], i[3]);
}
```

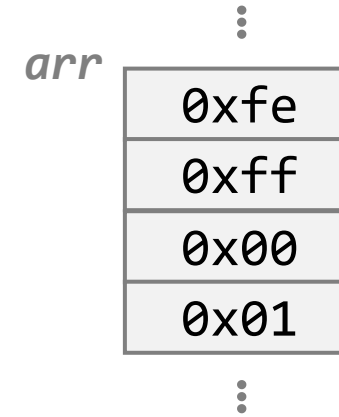
```
void func(void)
{
    char c[3] = {'c', 's', '\0'};
    printf("%c%c", c[0], c[1]);
    printf("%s", c);
}
```



Under the Hood, Indexing is Just Pointers

- **Array identifiers** act like **pointers** (to the first element)

```
void func(void)
{
    uint8_t arr[4] = {0, 1, 2, 3};
    uint8_t a0 = *arr;           // == arr[0]
    uint8_t a1 = *(arr + 1);    // == arr[1]
    ...
}
```



- **Element access** is just shorthand for **pointer arithmetic**

```
uint8_t arr[]; ↔ uint8_t *arr;
arr[0] ↔ *arr
arr[1] ↔ *(arr + 1)
⋮
arr[N] ↔ *(arr + N)
```

Pointer Arithmetic

Element access is just shorthand for **pointer arithmetic**

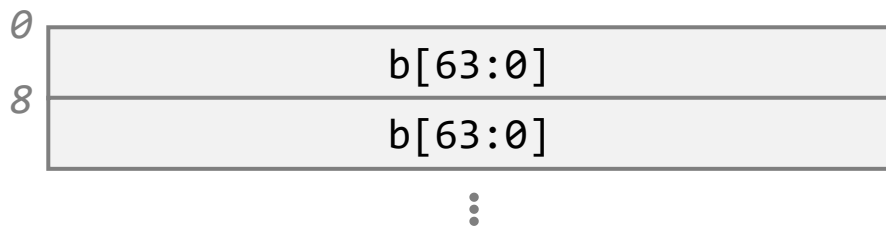
$$p[n] \Leftrightarrow *(p + n)$$

- Adding/subtracting **integers** operates on units of 1

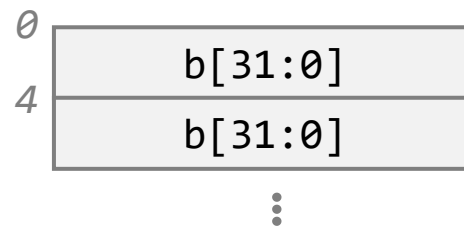
```
uint64_t u = 0;  
u = u + 1; // u = 1
```

- Adding/subtracting **pointers** operates on units of "**sizeof(pointee)**"

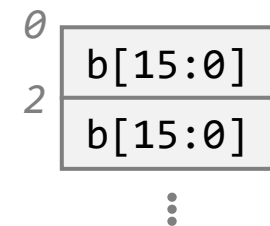
```
uint64_t *p = 0;  
p = p + 1; // p = 8
```



```
uint32_t *p = 0;  
p = p + 1; // p = 4
```



```
uint16_t *p = 0;  
p = p + 1; // p = 2
```



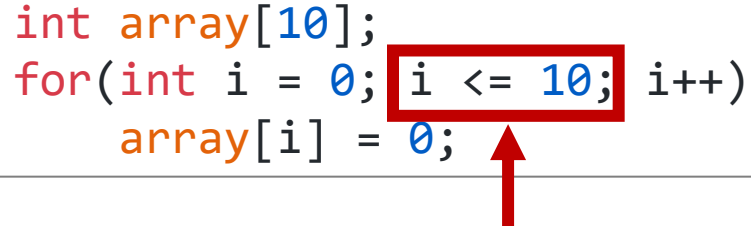
Agenda

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

- **Pointers**
 - Basics (continued)
 - Why pointers?
 - Pointer nuances
- **Arrays and Strings**
 - **Array nuances**

Caution: Array Off-By-One Errors

```
int array[10];  
for(int i = 0; i <= 10; i++)  
    array[i] = 0;
```



Array “out of bounds” access

- C does NOT check for you!
- Undefined behavior: can be hard to debug

- Avoid magic numbers – error prone and hard to read/change

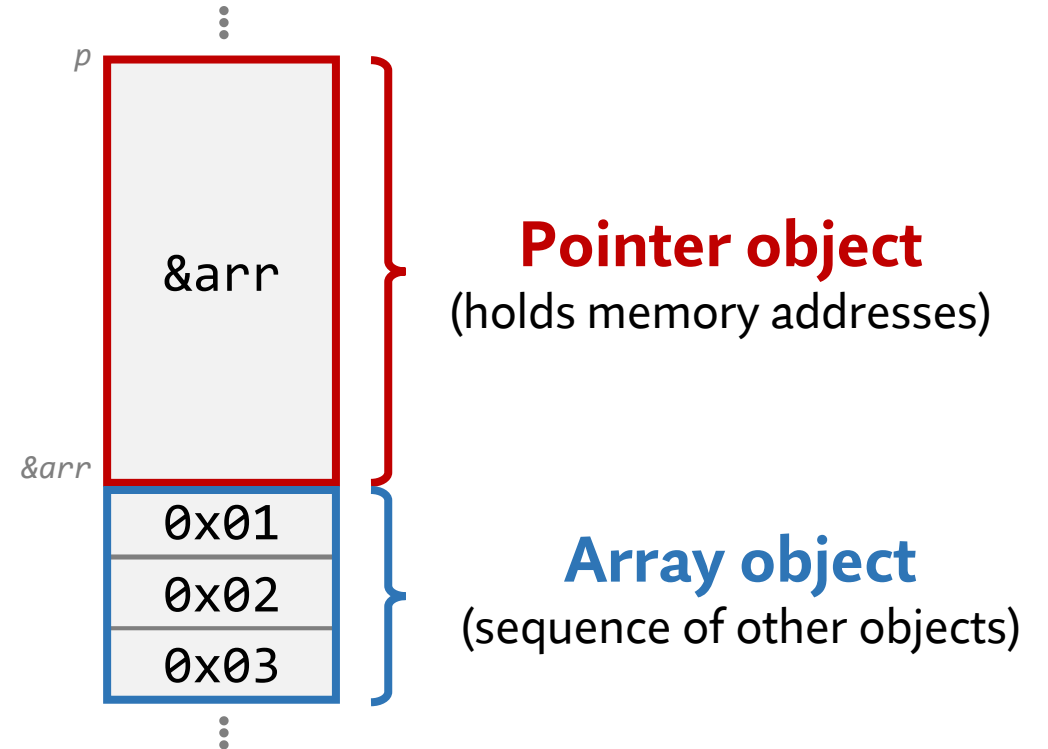
```
#define ARR_LEN 10  
  
int array[ARR_LEN];  
for(int i = 0; i < ARR_LEN; i++)  
    array[i] = 0;
```

Array vs. Pointer Identifiers

Array identifiers refer to
one array object

```
uint8_t arr[3] = {1, 2, 3};  
uint8_t *p = &arr;
```

Pointer identifiers refer to
one pointer object



```
sizeof(arr) == 3  
sizeof(arr[0]) == 1  
sizeof(p) == 8
```

Size of an Array

```
uint8_t arr[3] = {1, 2, 3};  
uint8_t *p = &arr;
```

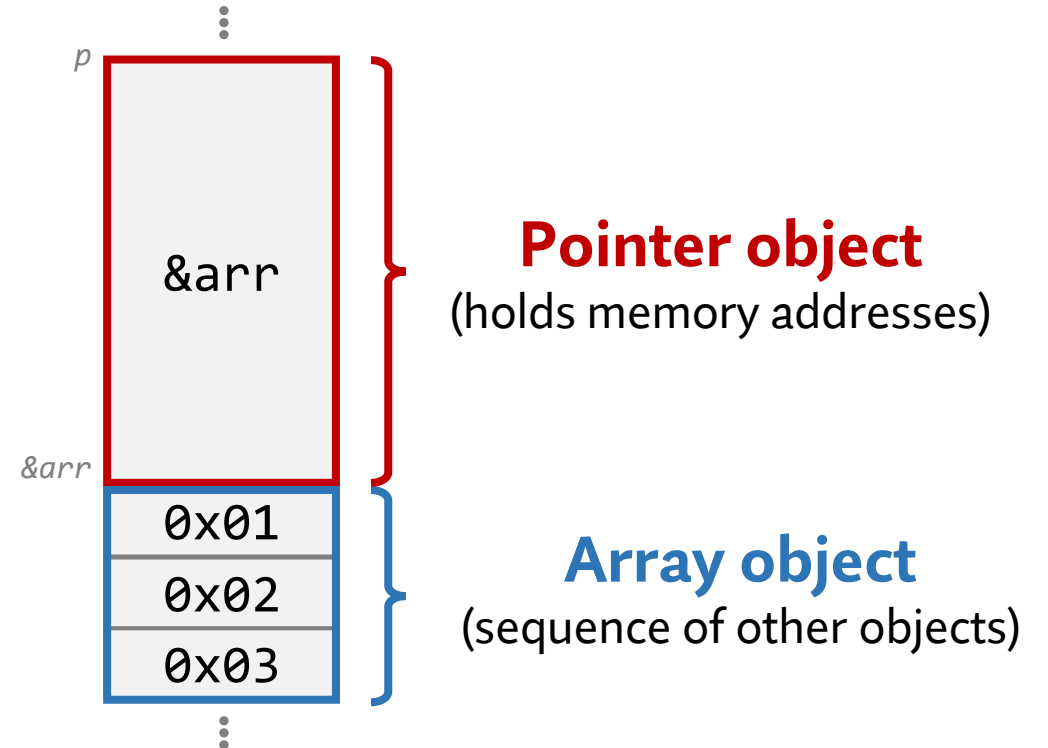
Compiler knows the length of `arr`

```
num_elements = sizeof(arr) / sizeof(arr[0]);
```

(the information is embedded in the type)

Compiler does NOT know the length of an array pointed to by `p`

```
num_elements = ??????(p);
```



```
sizeof(arr) == 3
```

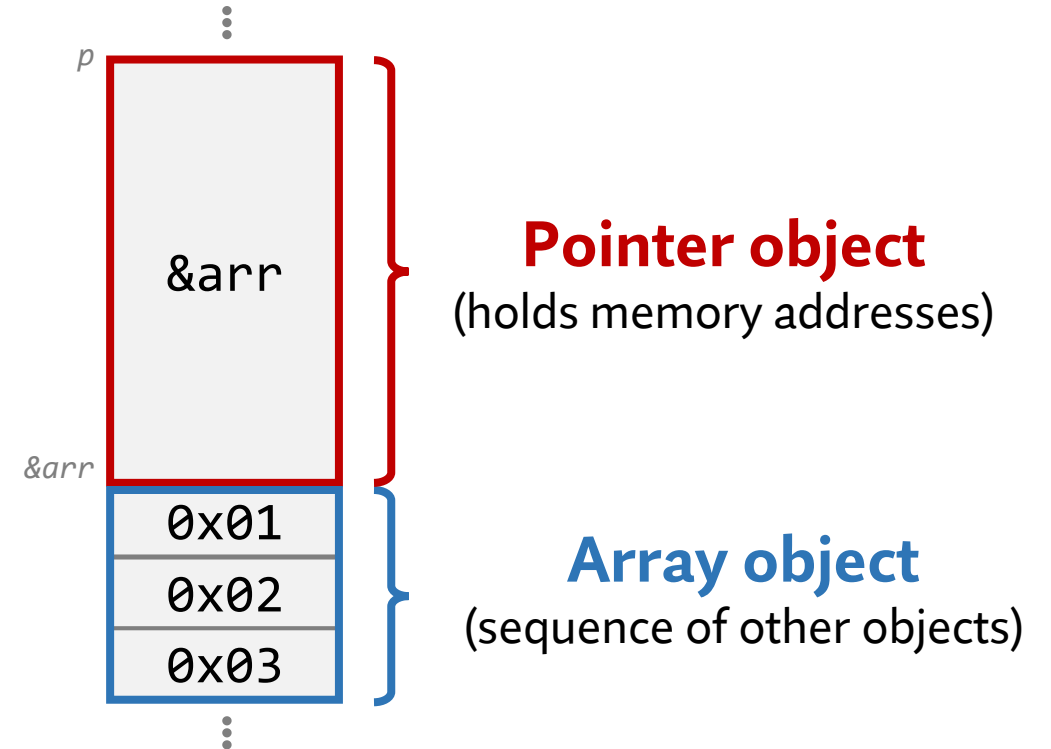
```
sizeof(arr[0]) == 1
```

```
sizeof(p) == 8
```

Caution: Pointers and Arrays are Not Identical

```
uint8_t arr[3] = {1, 2, 3};  
uint8_t *p = &arr;
```

- Do not confuse pointer and array types
 - `arr` does NOT “store” a memory address
 - Can’t change the address it represents
- Type of `arr` = array of `uint8_t[3]`
 - `uint8_t[3]`
- Type of `&arr` = pointer to array
 - `uint8_t(*)[3]`
- Type of `p` = pointer to `uint8_t`
 - `uint8_t *`

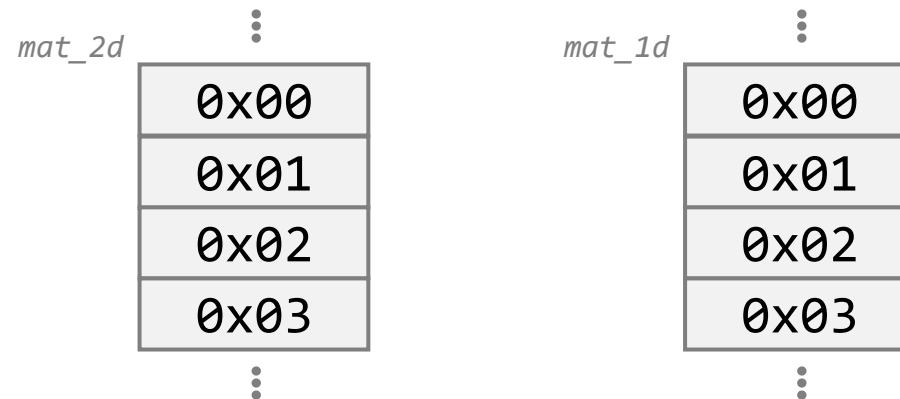


```
sizeof(arr) == 3  
sizeof(arr[0]) == 1  
sizeof(p) == 8
```


Multidimensional Arrays

- The compiler “does the right thing” for multidimensional arrays

```
uint8_t mat_2d[2][2] = {{0, 1}, {2, 3}};  
uint8_t mat_1d[4]    = {0, 1, 2, 3};  
  
mat_2d[0][0] == mat_1d[0];  
...
```



Demotion to Pointers

- Array objects are **treated as pointers** when passed to a function

```
void func(int *arr);  
void func(int arr[]);
```

 } **Treated as identical**

```
int main(int argc, char *argv[]);  
int main(int argc, char **argv);
```

 } **Treated as identical**

- Function arguments lose the array type information
 - Can no longer use `sizeof(arr)`
 - Need to explicitly pass the array length or use a sentinel to mark the end

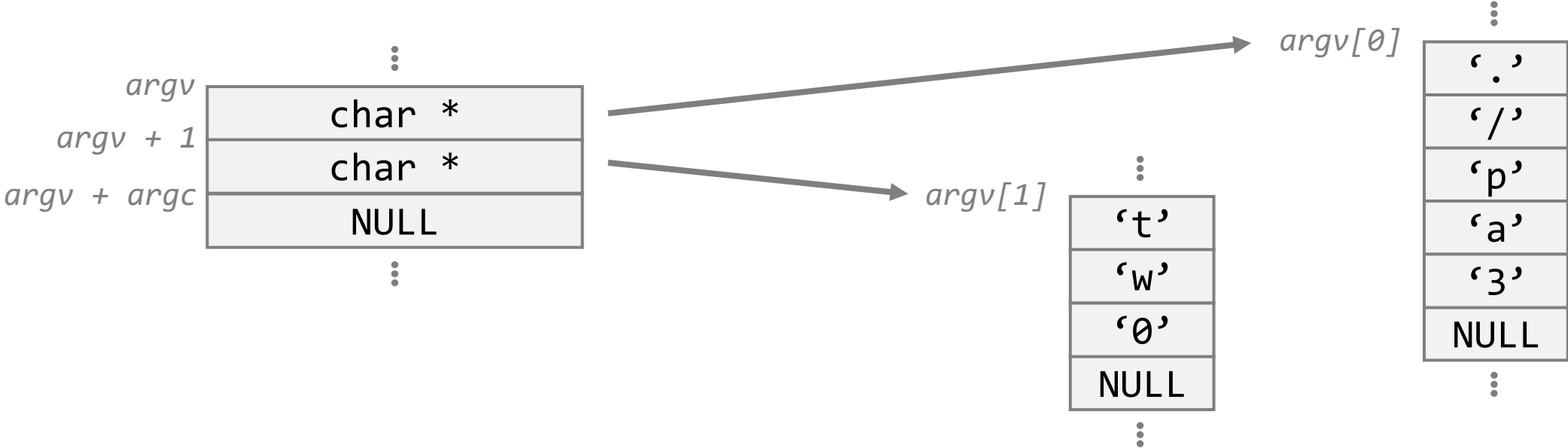
Multidimensional Arrays: Example “argv”

```
netid@ilab:~/cs211$ ./pa3 two
```

```
int main(int argc, char *argv[])  
{  
    // code  
}
```

argv is an **array of pointers**

argv[n] are **C strings**



Next Week

- Functions (and parameter passing)
- Memory layout of a C program
- Dynamic memory allocation

CS 211: Intro to Computer Architecture

6.1: C Data Representation III: Derived Types Cont.

Minesh Patel

Spring 2025 – Tuesday 25 February