CS 211: Intro to Computer Architecture

5.2: C Data Representation: Derived Types

Minesh Patel

Spring 2025 – Thursday 20 February

Announcements

- **PA2** due Sunday, Feb 23 @ 23:59

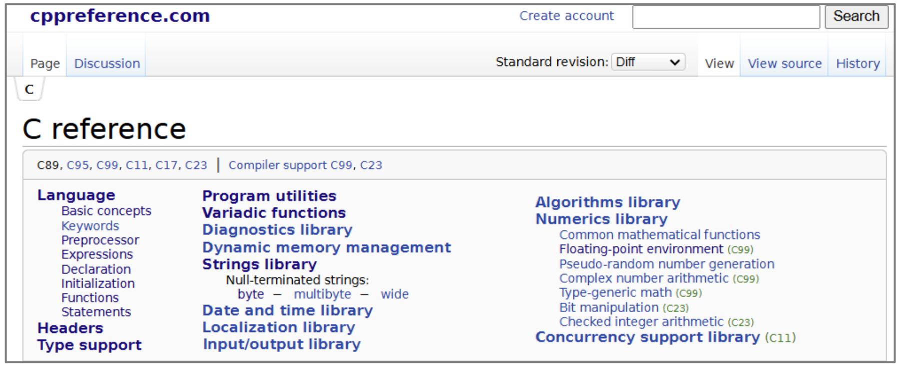
- **WA3** due next Monday, Feb 24 @ 23:59
  - **On Canvas this time** instead of Gradescope

- **WA4** to be assigned sometime in the next several days

Reference Material

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

And Various C and Linux Reference Materials



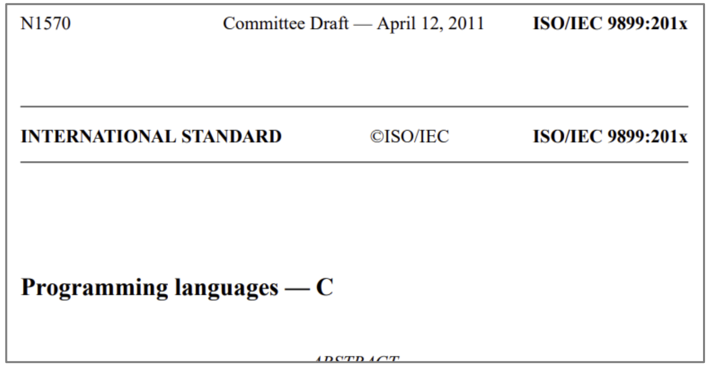
C reference

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

Language
Basic concepts
Keywords
Preprocessor
Expressions
Declaration
Initialization
Functions
Statements
Headers
Type support

Program utilities
Variadic functions
Diagnostics library
Dynamic memory management
Strings library
Null-terminated strings:
byte — multibyte — wide
Date and time library
Localization library
Input/output library

Algorithms library
Numerics library
Common mathematical functions
Floating-point environment (C99)
Pseudo-random number generation
Complex number arithmetic (C99)
Type-generic math (C99)
Bit manipulation (C23)
Checked integer arithmetic (C23)
Concurrency support library (C11)

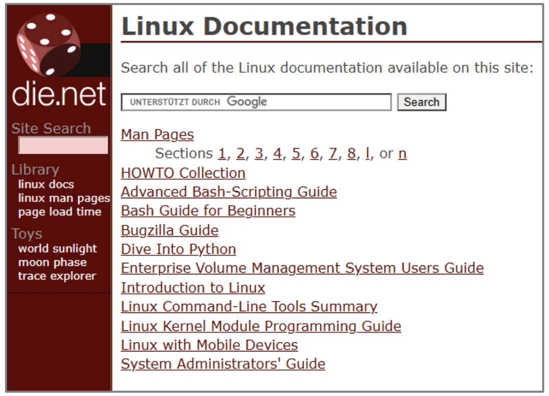


N1570 Committee Draft — April 12, 2011 **ISO/IEC 9899:201x**

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

Programming languages — C

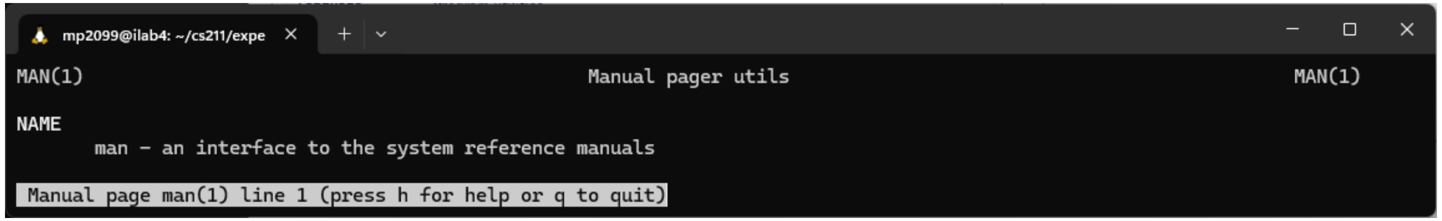
ABSTRACT



Linux Documentation

Search all of the Linux documentation available on this site:

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



```
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)
```

Pedantic Correction: Literal vs. Constant

- This is a complex topic (specifically compound literals `9899:201x 6.5.2.5`)
- For our purposes:

Literal

Creates an object in memory

(may or may not be modifiable)

```c
char *s = "hello";
char *c = (char []){"abc"};
int *i = (int[]){-1, 1};

struct vector
{
    int x;
    int y;
} *v = &(struct vector){.x = 1, .y = 0};
```

Constant

Does NOT occupy memory

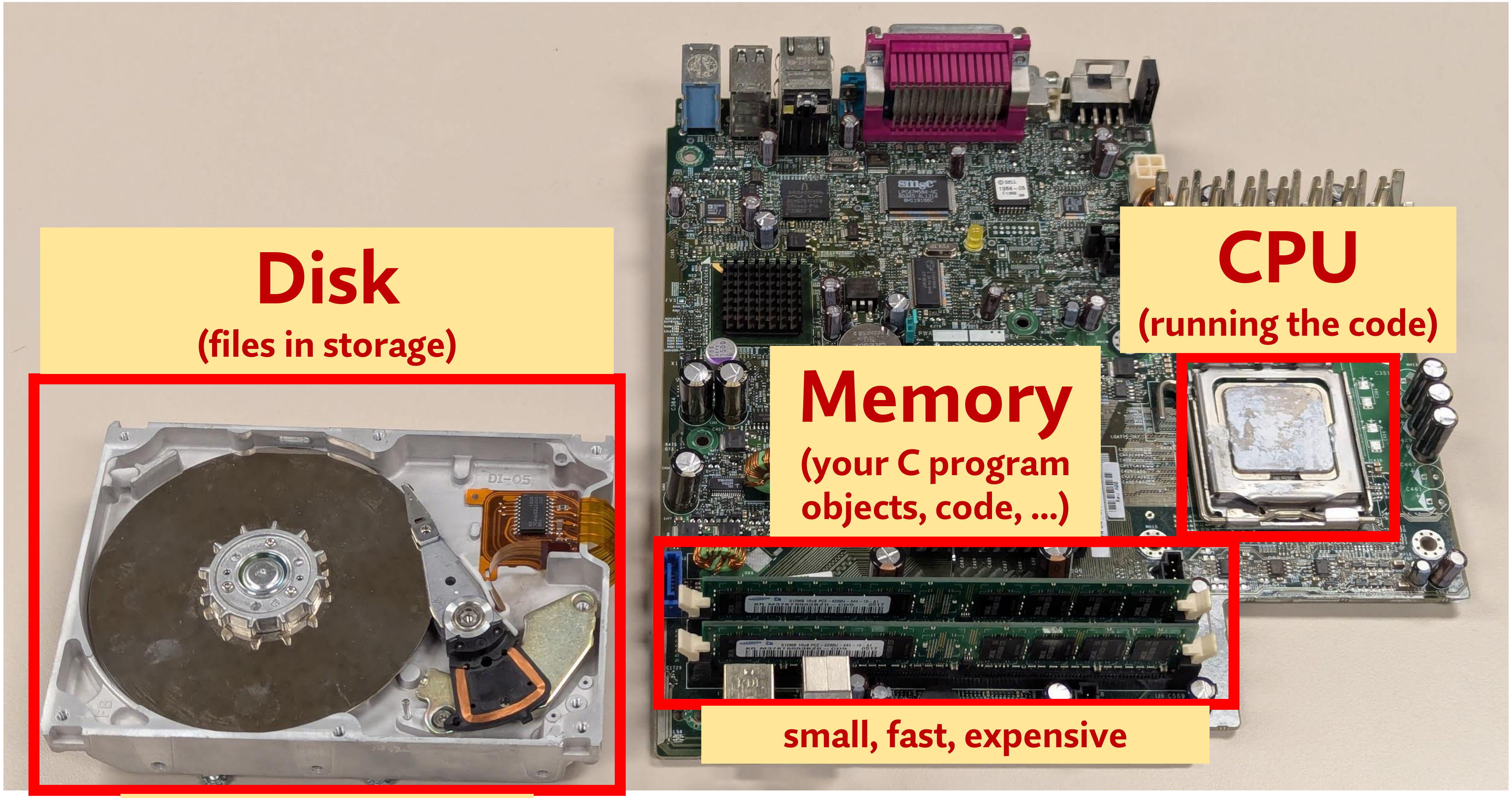
(can be evaluated at compile-time)

```c
int a = 10;
char c = 'c’;
int e = VAL; // enum value
uint64_t u = 0xfull;
```

Agenda

- **Enumerated and Derived Types**

Recap: What is this "Memory"?



Disk

(files in storage)

Memory

(your C program objects, code, ...)

CPU

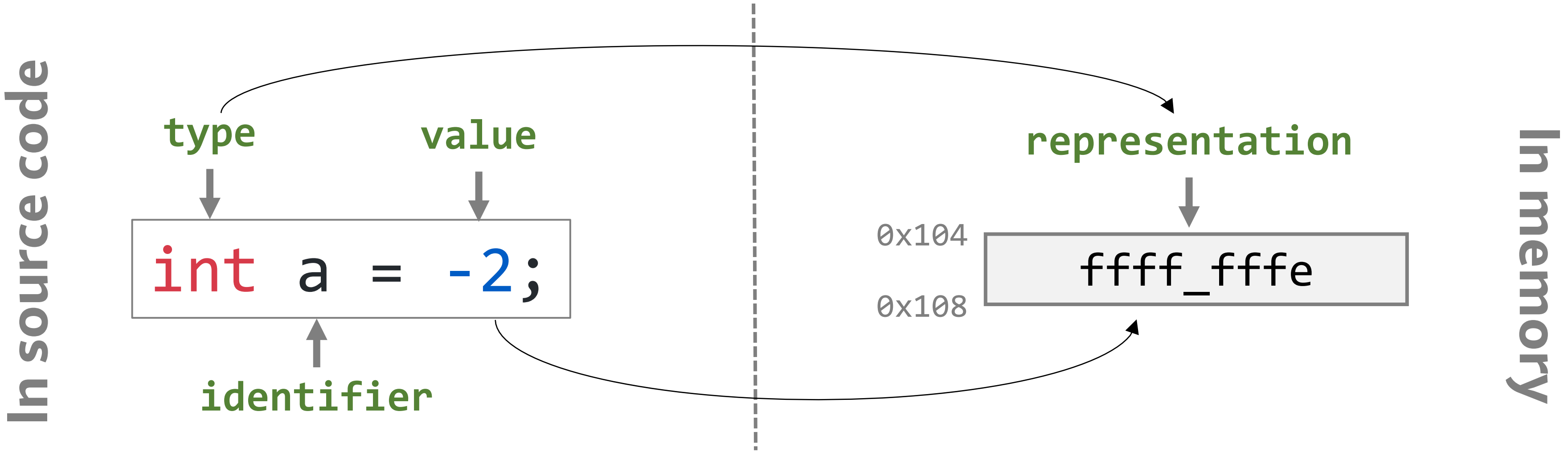
(running the code)

small, fast, expensive

big, slow, cheap

Recap: Why Does C Have Types?

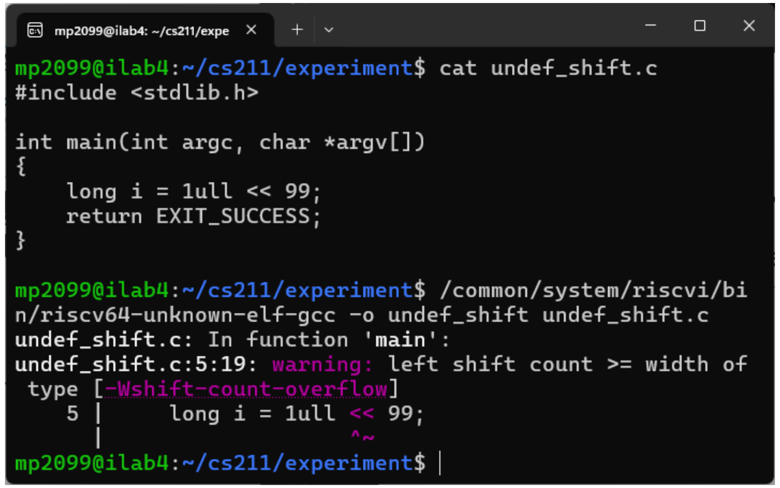
- **Types** tell the compiler how to represent **objects** in memory



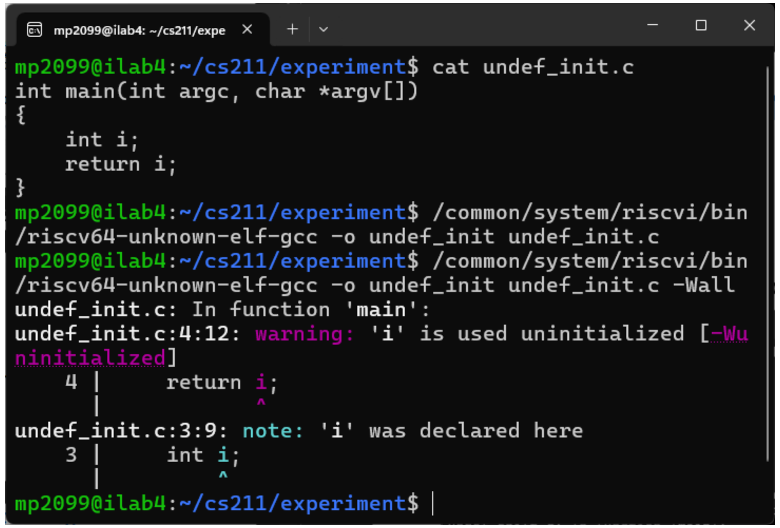
Recap: Undefined/Impl. Defined Behavior

Undefined Behavior: BAD

- Compiler will often warn you (in most cases)



```
mp2099@ilab4:~/cs211/experiment$ cat undef_shift.c
#include <stdlib.h>

int main(int argc, char *argv[])
{
    long i = 1ull << 99;
    return EXIT_SUCCESS;
}

mp2099@ilab4:~/cs211/experiment$ /common/system/riscvi/bi
n/riscv64-unknown-elf-gcc -o undef_shift undef_shift.c
undef_shift.c: In function 'main':
undef_shift.c:5:19: warning: left shift count >= width of
 type [-Wshift-count-overflow]
    5 |     long i = 1ull << 99;
      |                   ^~
mp2099@ilab4:~/cs211/experiment$ |
```

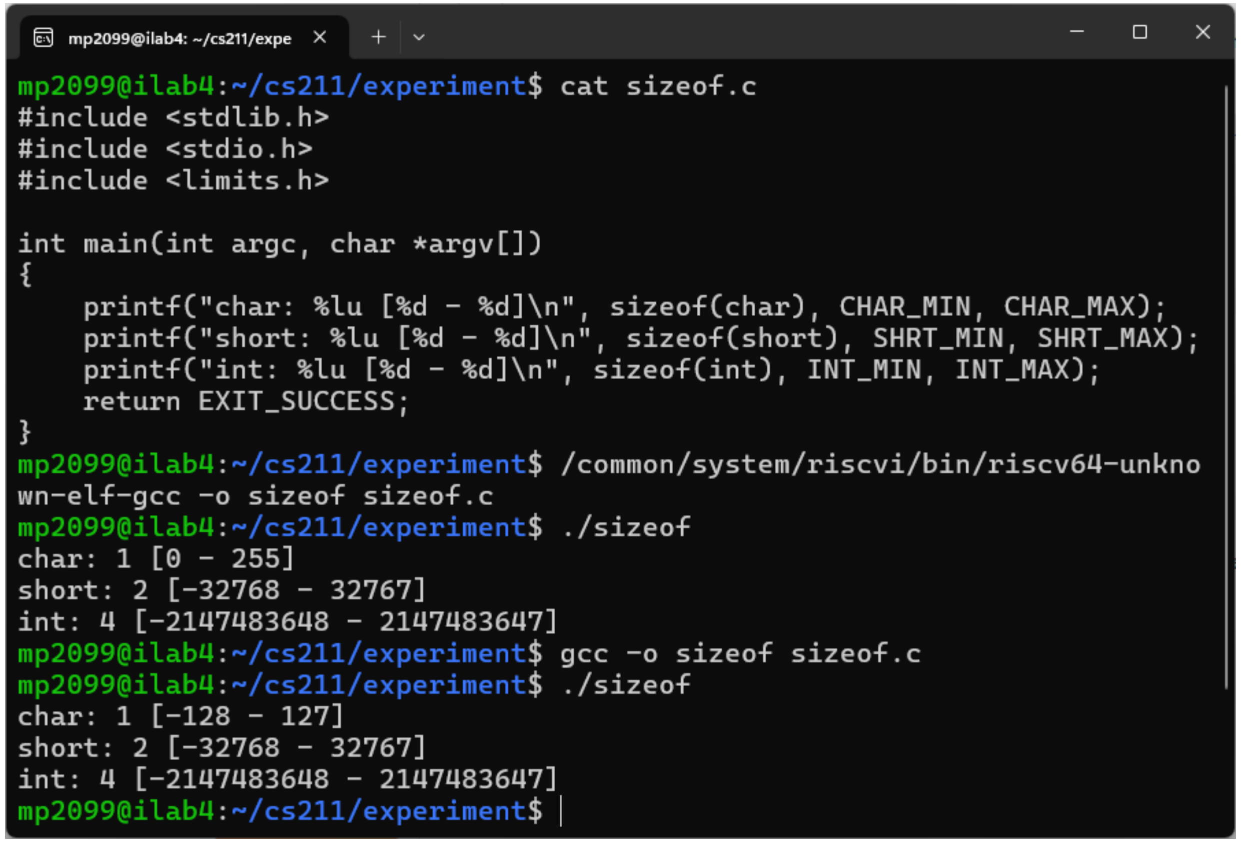


```
mp2099@ilab4:~/cs211/experiment$ cat undef_init.c
int main(int argc, char *argv[])
{
    int i;
    return i;
}
mp2099@ilab4:~/cs211/experiment$ /common/system/riscvi/bin
/riscv64-unknown-elf-gcc -o undef_init undef_init.c
mp2099@ilab4:~/cs211/experiment$ /common/system/riscvi/bin
/riscv64-unknown-elf-gcc -o undef_init undef_init.c -Wall
undef_init.c: In function 'main':
undef_init.c:4:12: warning: 'i' is used uninitialized [-Wu
ninitialized]
    4 |     return i;
      |            ^
undef_init.c:3:9: note: 'i' was declared here
    3 |     int i;
      |         ^
mp2099@ilab4:~/cs211/experiment$ |
```

Use –Wall!

Implementation Defined Behavior: OK

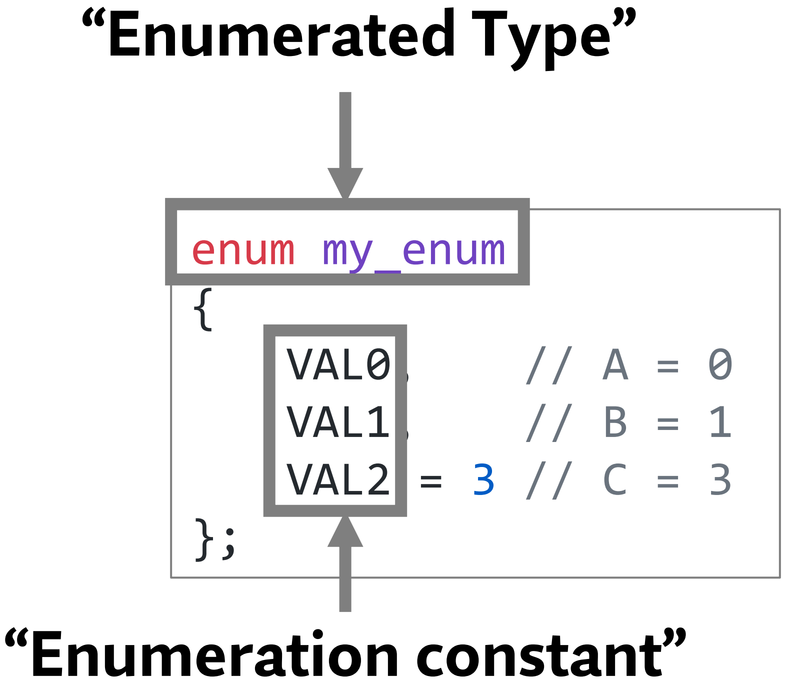
- Just remember that this varies between platforms



```
mp2099@ilab4:~/cs211/experiment$ cat sizeof.c
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main(int argc, char *argv[])
{
    printf("char: %lu [%d - %d]\n", sizeof(char), CHAR_MIN, CHAR_MAX);
    printf("short: %lu [%d - %d]\n", sizeof(short), SHRT_MIN, SHRT_MAX);
    printf("int: %lu [%d - %d]\n", sizeof(int), INT_MIN, INT_MAX);
    return EXIT_SUCCESS;
}
mp2099@ilab4:~/cs211/experiment$ /common/system/riscvi/bin/riscv64-unkno
wn-elf-gcc -o sizeof sizeof.c
mp2099@ilab4:~/cs211/experiment$ ./sizeof
char: 1 [0 - 255]
short: 2 [-32768 - 32767]
int: 4 [-2147483648 - 2147483647]
mp2099@ilab4:~/cs211/experiment$ gcc -o sizeof sizeof.c
mp2099@ilab4:~/cs211/experiment$ ./sizeof
char: 1 [-128 - 127]
short: 2 [-32768 - 32767]
int: 4 [-2147483648 - 2147483647]
mp2099@ilab4:~/cs211/experiment$ |
```

Agenda

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

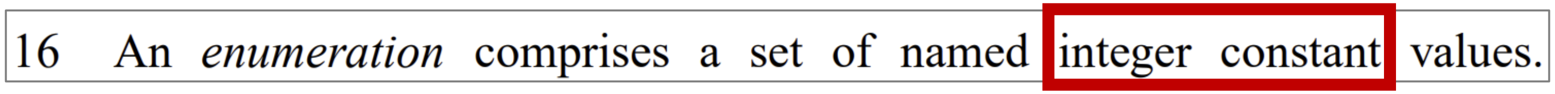


“Enumerated Type”

```c
enum my_enum
{
    VAL0,       // A = 0
    VAL1,       // B = 1
    VAL2 = 3    // C = 3
};
```

“Enumeration constant”

```c
void func(void)
{
    enum my_enum a = VAL0;
}
```



16 An *enumeration* comprises a set of named integer constant values.

ISO Standard 9899:201x

Closer Look: Enumeration Constants

- Enumerated constants are really just **named integer constants**

```c
enum days_in_a_month
{
    JAN = 31,
    FEB = 28,
    MAR = 31,
    APR = MAR - 1,
    ...
};
```

Non-monotonic

Non-unique

Usable once defined

You Write	Equivalent Code
JAN	(int)31
FEB	(int)28
MAR	(int)31

```c
void func(void)
{
    int days_in_a_year = JAN + FEB + MAR + ...;


    printf(“Year: %d\n", days_in_a_year);
}
```

Just integer values

Enums: Two Independent Types

- Enum definitions contain **two independent types**

1 “Enumerated Type”

Defines a **new type** with an implementation defined representation

- Compatible with {`char`, `int`, or `unsigned int`}
- Whichever is large enough to hold all values

2 “Enumeration constant”

- Type is always **`int`**
- Completely independent of the “Enumerated Type”

```
enum days_in_a_month
{
     JAN = 31,
     FEB = 28,
     MAR = 31,
     APR = MAR - 1,
     ...
};

void func(void)                           id           const.
{                  type
    enum days_in_a_month jan_days = JAN;
    int days_in_a_year =
         JAN + FEB + MAR + ...;

    printf(“January: %d\n", jan_days);
    printf(“Year: %d\n", days_in_a_year);
}
```

Type Checking and Implicit Casting

- Enums have **very limited** type checking

```
enum enum_type_0
{
    A = -(1u << 17),
    B = (1u << 17)
}
```

Could be any of:
{~~char~~, int, or ~~unsigned int~~}

```
enum enum_type_1
{
    C,
    D = 10,
};
```

Could be any of:
{char, int, or unsigned int}

```
enum enum_type_2
{
    E,
    F = (1ull << 62)
};
```

ISO 1988:2011 says that Enumerated Types can only be {~~char~~, ~~int~~, or ~~unsigned int~~}

Sensible Code (Allowed)

```
void func(void)
{
    enum enum_type_0 et0 = A;
    enum enum_type_1 et1 = C;
    enum enum_type_2 et2 = E;
}
```

Ignored Typing (Allowed)

```
void func(void)
{
    int et0 = A;
    int et1 = C;
    int et2 = E;
}
```

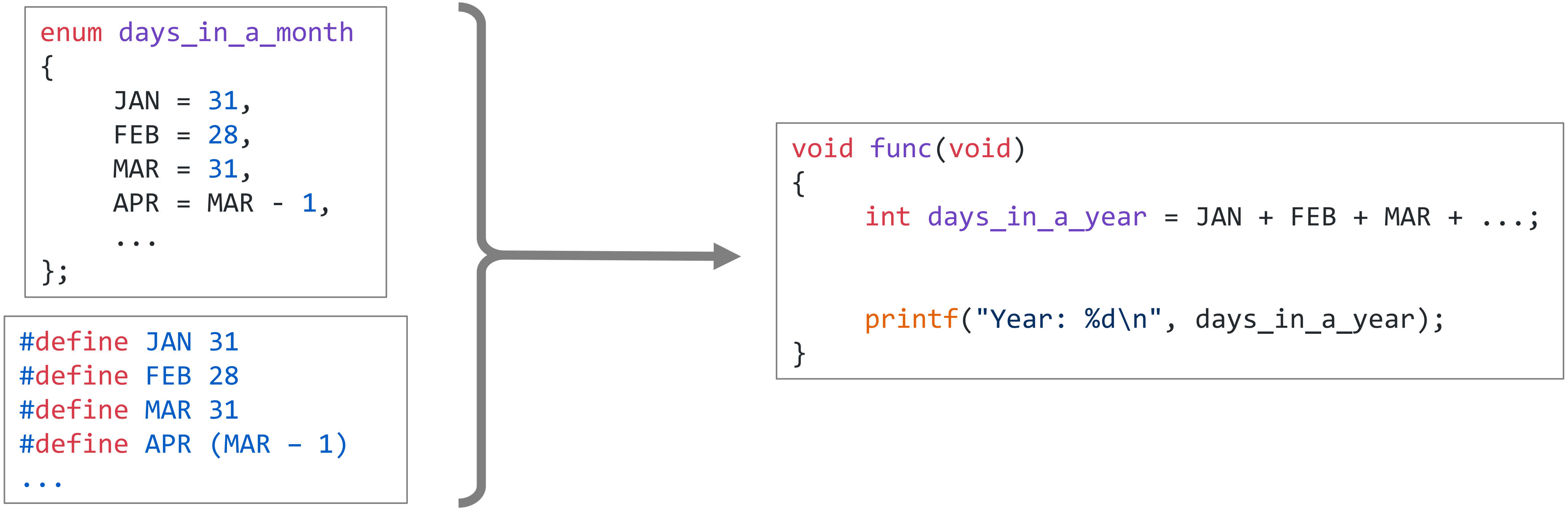
Misleading Code (Still Allowed)

```
void func(void)
{
    enum enum_type_0 et0 = C;
    enum enum_type_0 et1 = D;
    enum enum_type_0 et2 = E;
}
```

No specific compiler warnings/errors about mismatched enum types

Enums vs. Macros

- If there aren’t strong type checks, why not just use **macros**?



```
enum days_in_a_month
{
    JAN = 31,
    FEB = 28,
    MAR = 31,
    APR = MAR - 1,
    ...
};
```

```
#define JAN 31
#define FEB 28
#define MAR 31
#define APR (MAR – 1)
...
```

```
void func(void)
{
    int days_in_a_year = JAN + FEB + MAR + ...;


    printf("Year: %d\n", days_in_a_year);
}
```

- Unlike macros (which are processed before compilation), enums:
    - Are visible in the debugger (gdb)
    - Obey scoping rules
    - Follow `int` typing rules

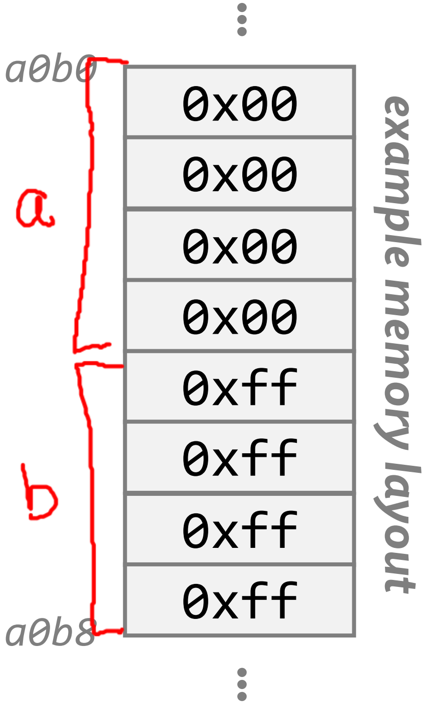
Agenda

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

- Structs are a **set of objects** under **one identifier**
  - Simply for programming convenience
  - NOT related to OOP/classes/methods

```
struct my_struct
{
    int a;
    int b;
};

void func(void)
{
    struct my_struct i = {.a = 0, .b = -1};
    printf("%d %d\n", i.a, j.a);
}
```

It's not always this simple.
More on that next week



Structures

- You can have whatever objects you want in a struct
  - There are no “method calls”, “public/private access”, or “inheritance” like C++/Java
  - It’s just a sugar-coated way of accessing the members

```
struct my_struct
{
    unsigned char a;
    int64_t b;
    enum my_enum e;
};

void func(void)
{
    struct my_struct i;
    i.a = 0;
}
```

- Three **separate memory locations**
- Updating one does NOT affect another

- We will revisit structs after covering pointers

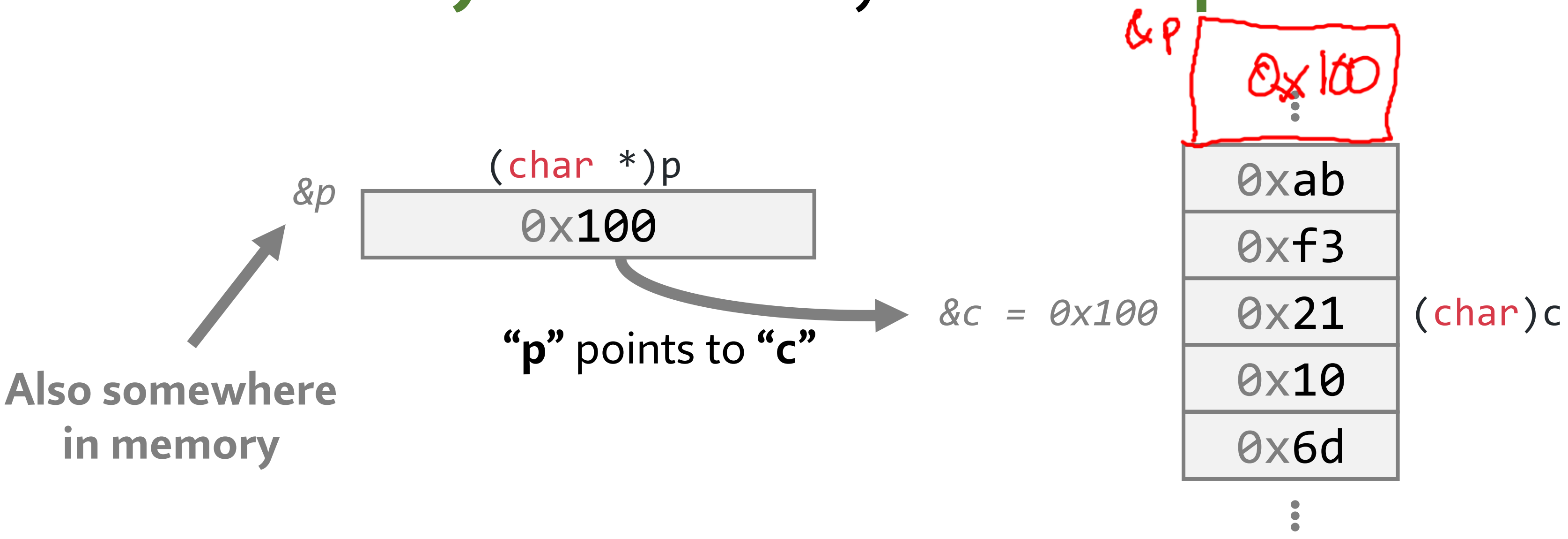
Pointers Store Memory Addresses

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

- Pointer objects represent **memory addresses**
  - Compatible with an unsigned integer type (ilab: `uint64_t`)

Recall:

1. **Memory** is a contiguous sequence of **bytes**.
2. **Each byte** in memory has a **unique address**.

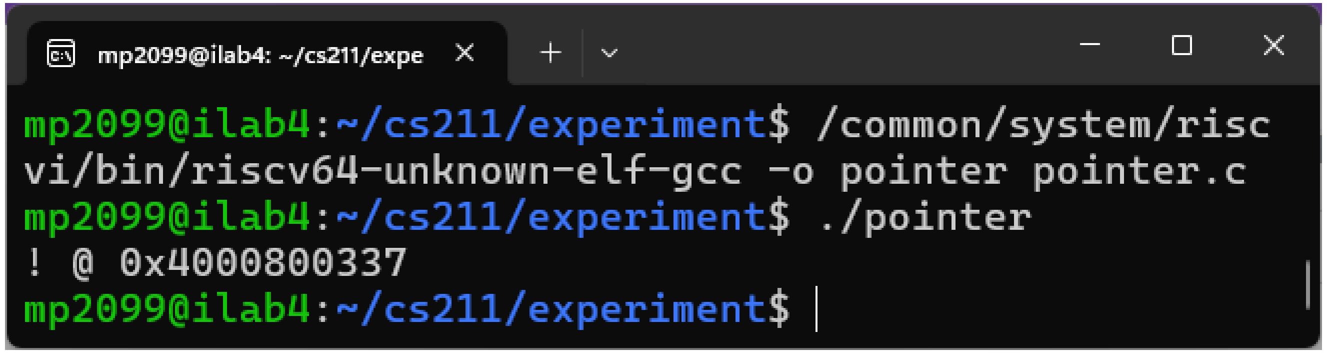


The Width of a Pointer Object

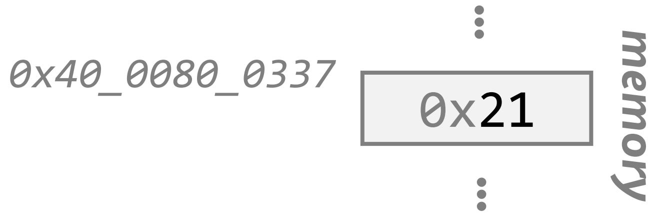
- Width of a memory address (pointer) is **implementation defined**
  - **64 bits** on RISC-V-64 (and x86_64, AArch64)
  - **<= 32 bits** on many older or low-power systems

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

int main(int argc, char *argv[])
{
    char c = '!'; // ascii 0x21

    printf("%c @ %p\n", c, &c);
    return EXIT_SUCCESS;
}
```

PRI_CHAR



```
mp2099@ilab4:~/cs211/experiment$ /common/system/risc
vi/bin/riscv64-unknown-elf-gcc -o pointer pointer.c
mp2099@ilab4:~/cs211/experiment$ ./pointer
! @ 0x4000800337
mp2099@ilab4:~/cs211/experiment$
```



[Demo] Examining Memory in GDB

Terminal Emulator 1:

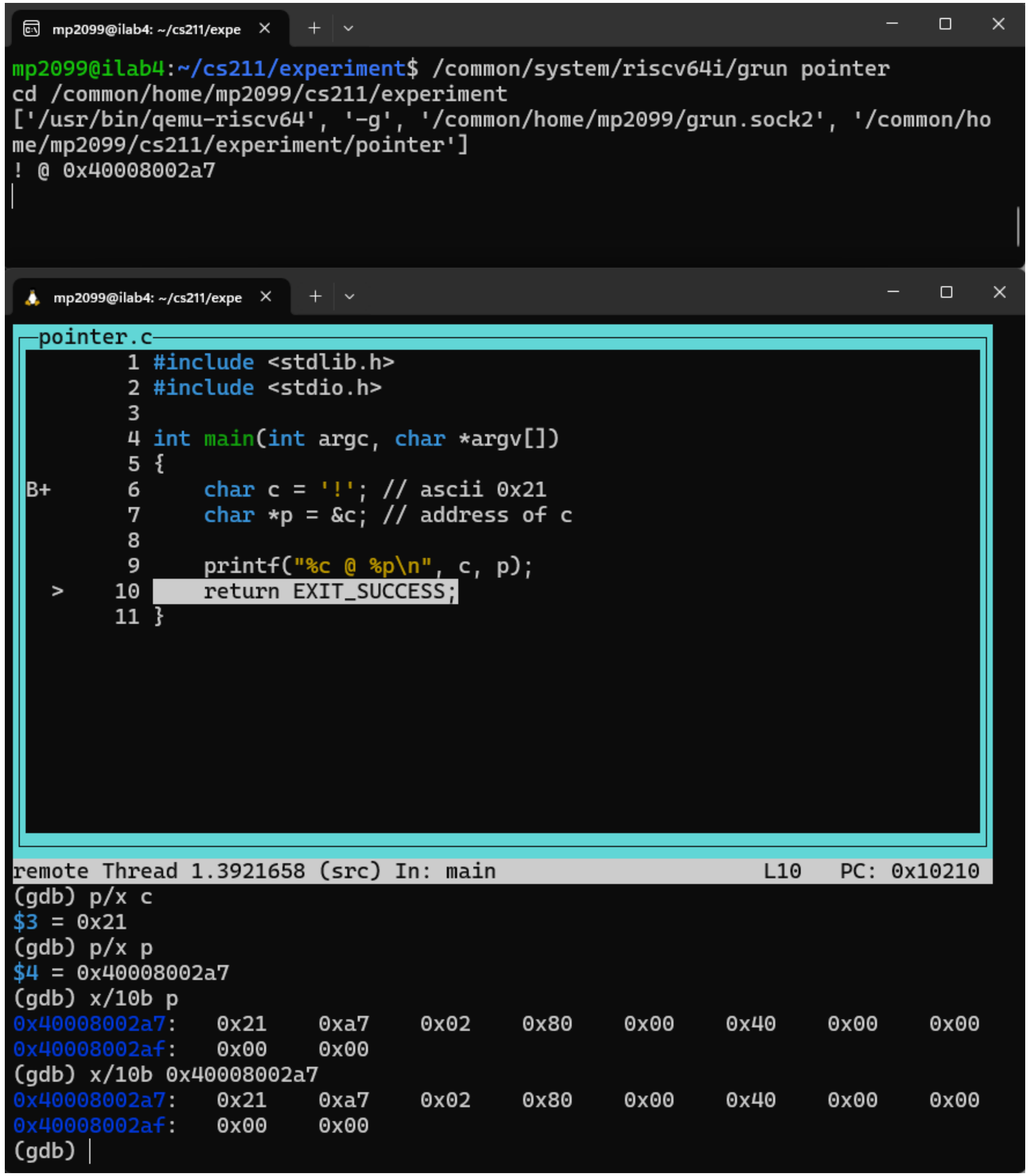
```
netid@ilab4:~/cs211/experiment$ /common/system/riscv64i/grun pointer
```

same ilab machine!   same directory!

Terminal Emulator 2:

```
netid@ilab4:~/cs211/experiment$ /common/system/riscv64i/gdb pointer
```

- **start** or **run** – start the session
- **break <location>** - set a breakpoint
  - **b main** – break at the main function
  - **b main.c:11** – break there, for example
- **p <variable>** - print a variable
  - **p/x <variable>** – print in hex
  - **p/t <variable>** – print in binary
- **x <address>** - print memory at the address
  - **x/10b <address>** – print 10 bytes
  - **x/10b <variable>** - get the address from a variable
- **layout <command>** - change the GDB layout
- **help <command>** - get some help ☺



```
mp2099@ilab4:~/cs211/experiment$ /common/system/riscv64i/grun pointer
cd /common/home/mp2099/cs211/experiment
['/usr/bin/qemu-riscv64', '-g', '/common/home/mp2099/grun.sock2', '/common/ho
me/mp2099/cs211/experiment/pointer']
! @ 0x40008002a7
```

```
pointer.c
         1 #include <stdlib.h>
         2 #include <stdio.h>
         3
         4 int main(int argc, char *argv[])
         5 {
B+       6     char c = '!'; // ascii 0x21
         7     char *p = &c; // address of c
         8
         9     printf("%c @ %p\n", c, p);
  >     10     return EXIT_SUCCESS;
        11 }

remote Thread 1.3921658 (src) In: main                    L10   PC: 0x10210
(gdb) p/x c
$3 = 0x21
(gdb) p/x p
$4 = 0x40008002a7
(gdb) x/10b p
0x40008002a7:   0x21    0xa7    0x02    0x80    0x00    0x40    0x00    0x00
0x40008002af:   0x00    0x00
(gdb) x/10b 0x40008002a7
0x40008002a7:   0x21    0xa7    0x02    0x80    0x00    0x40    0x00    0x00
0x40008002af:   0x00    0x00
(gdb)
```

Aside: Size of Integer and Pointer Types

- A pointer represents a **memory address** as an **unsigned number**
- How many bits do we need in a pointer?

1B

1 KiB = 1024 B = 2^{10} B

1 MiB = 1024 KiB = 2^{20} B

1 GiB = 2^{30} B

⋮

Depends on how much memory we have

10^6 B

MiB = 2^{20}

64 ~~MB~~ of RAM

$64 \cdot 2^{20}$ B

$2^6 \cdot 2^{20}$ B

2^{26} B → 26 bits

512 MB of RAM

$512 \cdot 2^{20} = 2^{29}$

⇒ 29-bit pointer

4 GiB → 32-bit pointers

8 GB of RAM

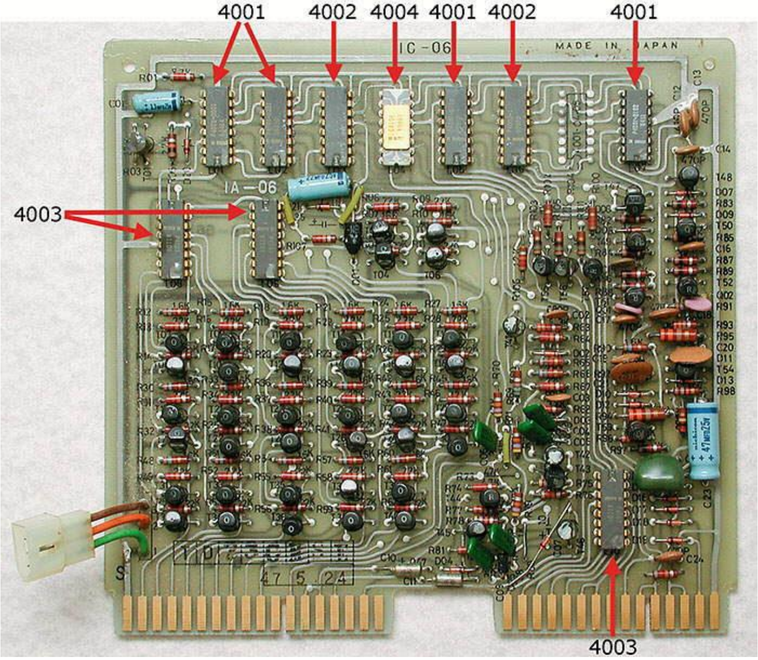
2^{33} ⇒ 33-bit pointer

Aside: Size of Integer and Pointer Types

- A pointer represents a **memory address** as an **unsigned number**

		Data Width	Pointer size	Max. Memory
1971	Intel 4004	4 bits	12 bits	640 bytes ← `uint64_t[10]`

http://www.vintagecalculators.com/html/busicom_141-pf.html



http://www.vintagecalculators.com/html/busicom_141-pf.html

Aside: Size of Integer and Pointer Types

- A pointer represents a **memory address** as an **unsigned number**

		Data Width	Pointer size	Max. Memory
1971	Intel 4004	4 bits	12 bits	640 bytes
1977	MOS 6502 (Apple II, Atari 2600)	8 bits	16 bits	64 Kbytes
1982	MOS 6510 (Commodore 64)			
1983	Richo RP2A03 (NES)	8 bits	16 bits	2 Kbytes

Gregory, Klahn, Bonilla, *"NES Hardware Emulation"*

https://cdn-blog.adafruit.com/uploads/2019/01/Untitled-17.png

https://upload.wikimedia.org/wikipedia/commons/0/02/Atari-2600-Wood-4Sw-Set.png

Aside: Size of Integer and Pointer Types

- A pointer represents a **memory address** as an **unsigned number**

		Data Width	Pointer size	Max. Memory
1971	Intel 4004	4 bits	12 bits	640 bytes
1977	MOS 6502 (Apple II, Atari 2600)	8 bits	16 bits	64 Kbytes
1982	MOS 6510 (Commodore 64)			
1983	Richo RP2A03 (NES)	8 bits	16 bits	2 Kbytes
1997	PowerPC 750 (iMac)	32 bits	32 bits	4 Gbytes
2000	Pentium 4 (desktops/laptops/servers)	32 bits	32 bits	4 Gbytes
2003	Athlon 64 (desktops/laptops/servers)	64 bits	64 bits	16 Ebytes

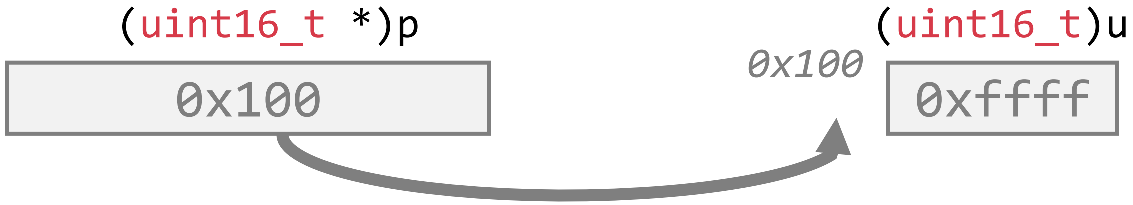
Upper Bound: no vendor actually builds this much

Pointers Enable Read/Write to Other Objects

Declaring a Pointer Object

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

Address of object “u”

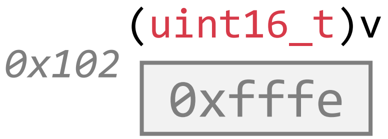


“p” points to “u”

Reading the Value Pointed To

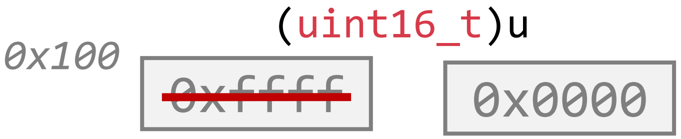
```
uint16_t v = *p - 1;
```

“dereference the pointer”



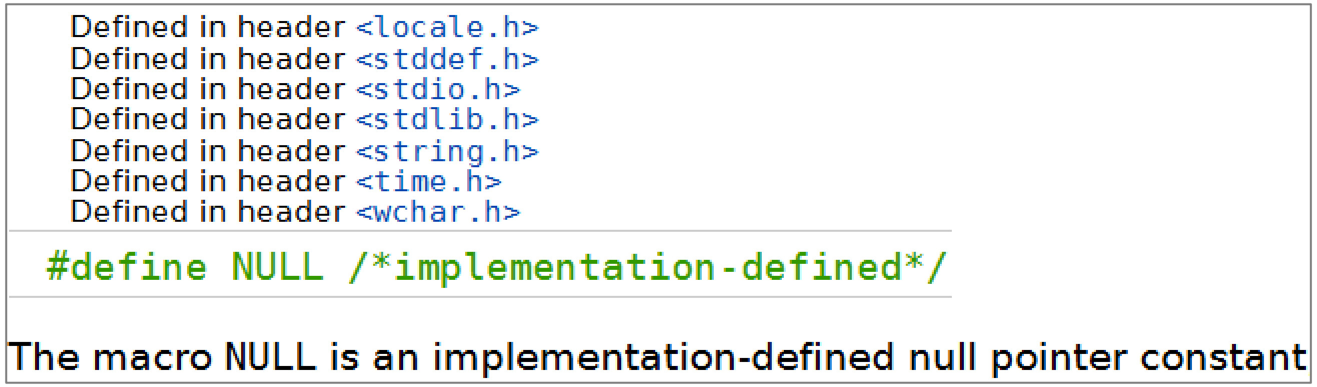
Writing the Value Pointed To

```
*p = *p + 1;
```



Initializing Pointer Objects

- The C library provides a macro **NULL** to represent "**pointing to nothing**"



Defined in header `<locale.h>`
Defined in header `<stddef.h>`
Defined in header `<stdio.h>`
Defined in header `<stdlib.h>`
Defined in header `<string.h>`
Defined in header `<time.h>`
Defined in header `<wchar.h>`

```
#define NULL /*implementation-defined*/
```

The macro NULL is an implementation-defined null pointer constant

https://en.cppreference.com/w/c/types/NULL

```
uint16_t *p = NULL;
if(p == NULL)
    handle_error();
```

Uninitialized Object

```
uint16_t *p;
```

Value is **undefined**.

Points to "something" (e.g., random location in memory, invalid location, etc.

Correctly Initialized

```
uint16_t *p = NULL;
```

Value is **well-defined**.

Points to "no object".

Correctly Initialized

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

Value is **well-defined**.

Points to "object u".

Using NULL Pointers

- Any type of pointer can be set to **NULL**
  - Best-practice to initialize pointers to **NULL**
  - Pointer-returning functions often return **NULL** to signal an error

```
int *p = NULL;
```

```
uint64_t *p = NULL;
```

```
char *p = NULL;
```

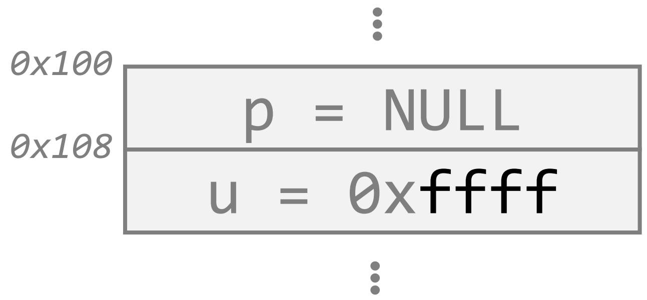
- Dereferencing NULL **will always** crash the program
  - Helpful for debugging! Much better than undefined behavior

```
uint16_t *p = NULL;
uint16_t u = *p; // crash
```

- Note: do not confuse **0** (for integers) with **NULL** (for pointers)
  - Further reading: https://c-faq.com/null/

Pointer Example

```c
#include <stdlib.h>

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

    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);
}
```



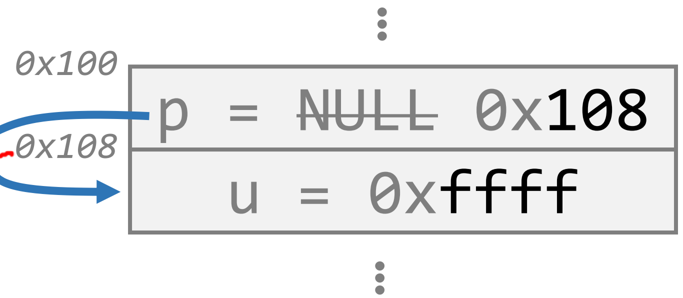
- Declare an **identifier** p
  - Type “pointer to uint16_t”
  - Initialized to NULL
    - i.e., pointing to “nothing”

Pointer Example

```
#include <stdlib.h>

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

    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);
}
```

ffff

ffff &u = 0x108



0x100 p = ~~NULL~~ 0x108

0x108 u = 0xffff

- 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”

type(p) = uint_16t*

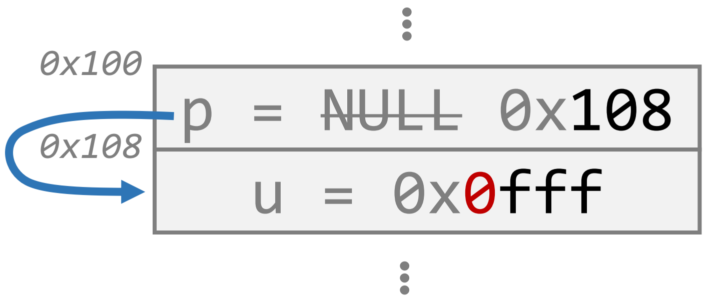
type(*p) = uint_16t

type(&u) = uint_16t*

type(&p) = uint_16t**

Pointer Example

```
#include <stdlib.h>

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

    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);
}
```



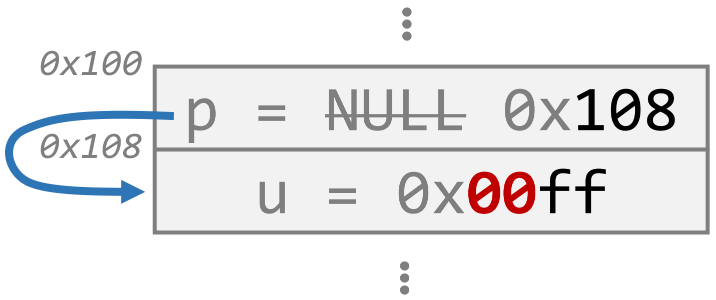
0x100 p = ~~NULL~~ 0x108

0x108 u = 0x0fff

- 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 = 0xffffu;

    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

CS 211: Intro to Computer Architecture

5.2: C Data Representation: Derived Types

Minesh Patel

Spring 2025 – Thursday 20 February