Version 2.00

UCSD CSE 30 Section B

Computer Organization and Systems Programming

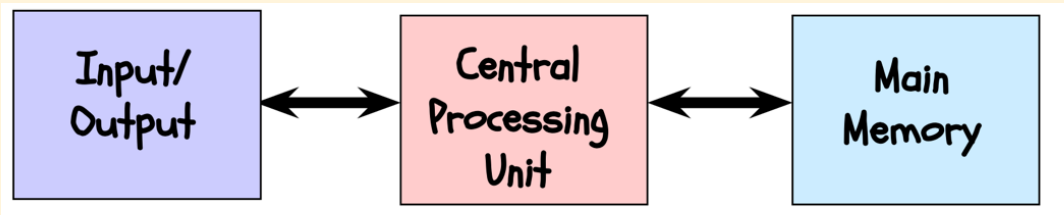
Midterm Review

Keith Muller

DEC PDP 11/45 - 1973

A General-Purpose Computer – Von Neuman Architecture

- Since the middle of the 20th century, many architectural approaches to the **general-purpose computer** have been tried
- The **architecture** which **nearly all modern computers** are based was proposed by John Von Neuman in the late 1940's
- The **major components** are:



- Central Processing Unit (CPU): a device which fetches, interprets (decodes), and executes a specified set of operations called instructions
- Memory: Storage of N words of W bits, where W is a fixed architectural parameter, and N can can be expanded to meet **workload** (the programs running on the CPU) and **cost requirements**
- I/O: Devices for communication with the outside world (including external persistent storage)
  - External connections (from CPU to memory and I/O) typically use industry **"standards"**
  - **Standards** enable technologies from **different companies to interoperate**

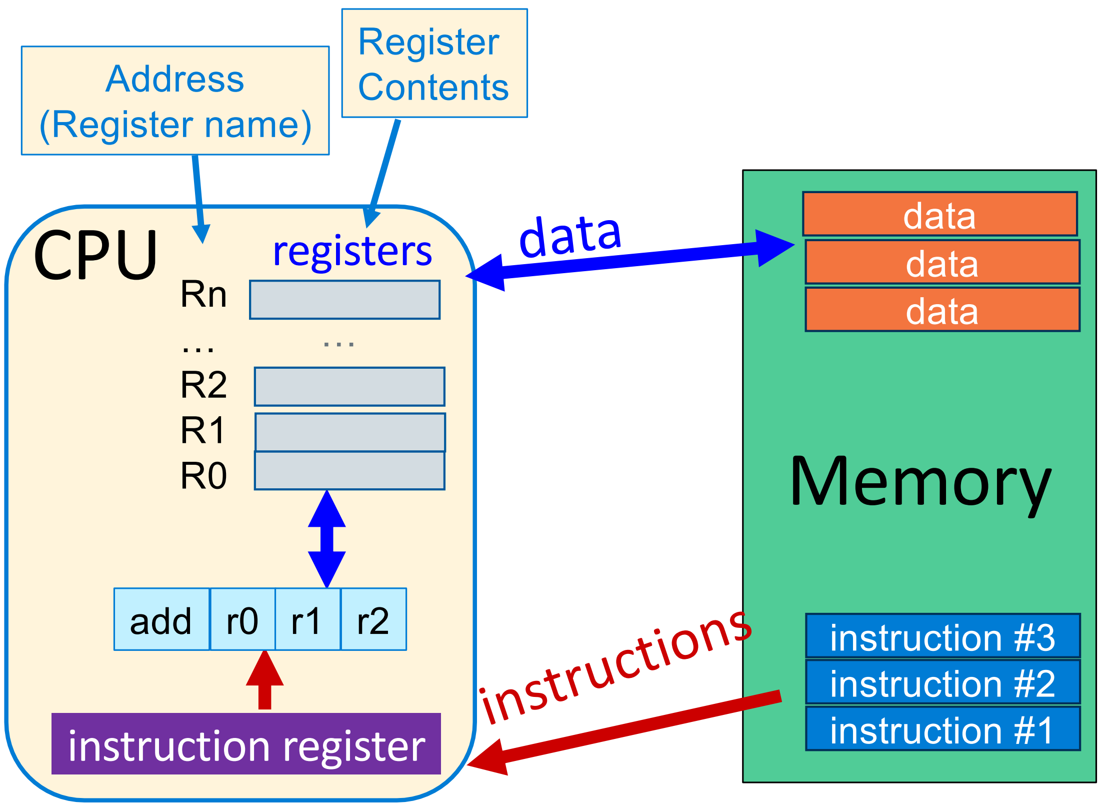
x

Von Neuman Architecture

- **Distinguishing feature**: Memory contains **both** program CPU instructions and data
- **CPU Instructions** are encoded in memory with patterns of ones and zeros (similar to binary numbers)
    - **Encoded CPU instructions** are called **machine code (or machine language)**
- **Example**: three 32-bit instructions (shown in hexadecimal format below)

```
81 fe 89 32
81 54 22 af
81 22 10 9A
```

- **Instructions** operate on **data** that is stored in a small capacity volatile (fast) memory in the CPU
    - This memory is called registers
    - ARM-32 has 32-bit registers
- CPU **reads/writes data** from **memory** from these **data registers** to **operate on the data**

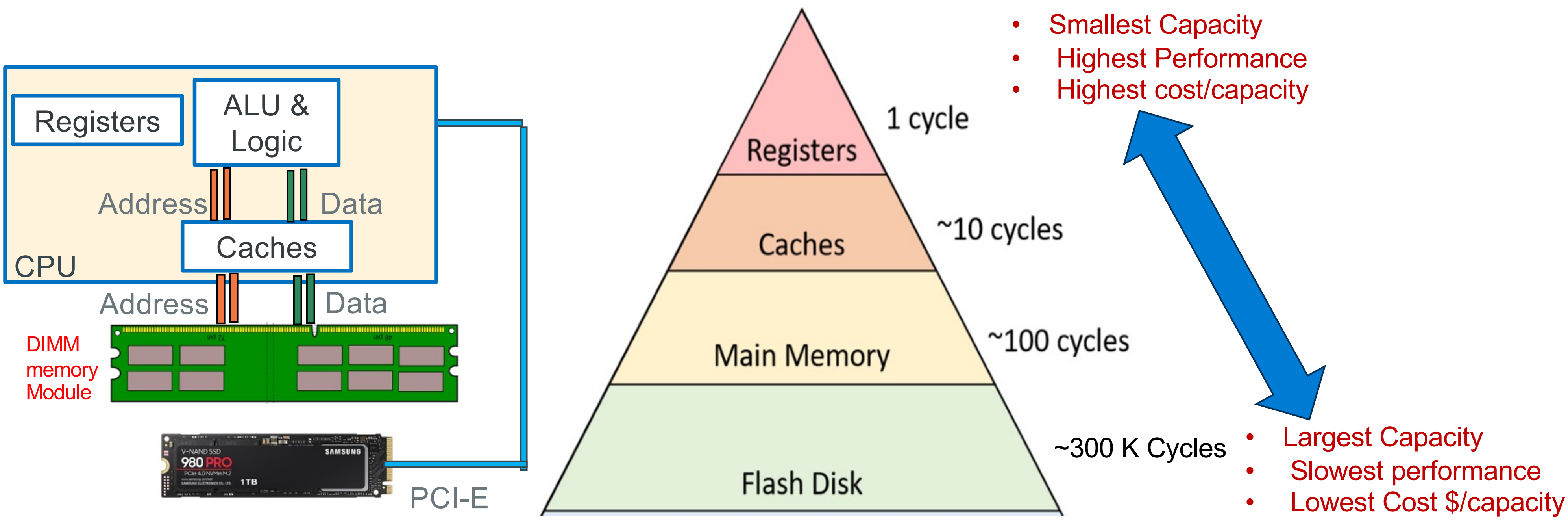


x

Memory Triangle: Hardware Cost/Performance/Capacity Tiers

Assume each instruction takes 1 clock cycle

Clock cycle =~ time to access; larger is slower

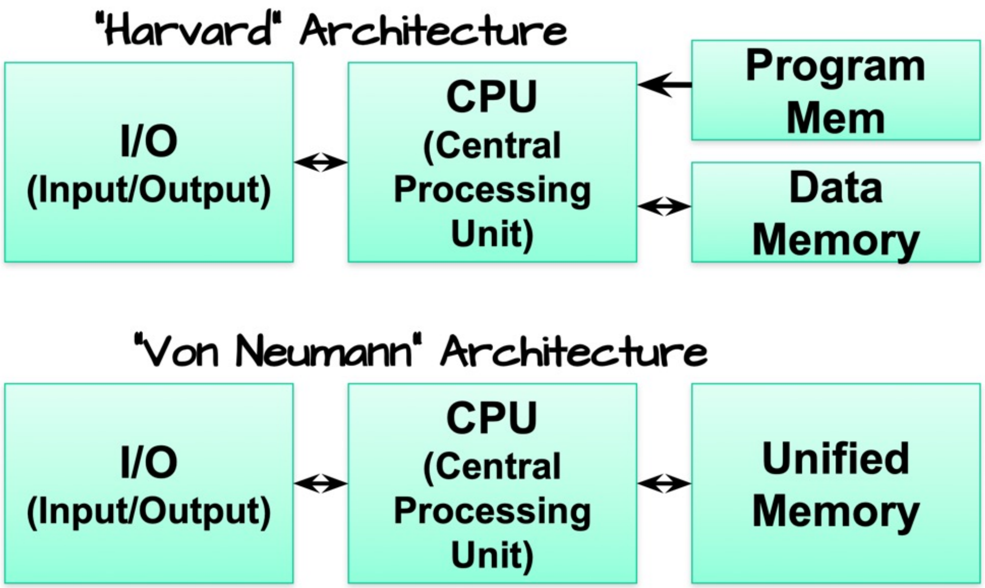


Design Goal: best performance at the lowest (or specific) cost
Other goals: performance/energy (operating cost), expandability, high margin (price/cost)

x

An Alternative that was not successful: Harvard Architecture

- **Harvard architecture premise:** Instructions and data should not interact (claim is higher higher performance), and they can have different word sizes
  - **Observation:** Two memory subsystems (using similar state of the art technologies) can be accessed concurrently for higher throughput
- **Distinguishing feature: Independent instruction** and **data memories**
- Do you agree and why?

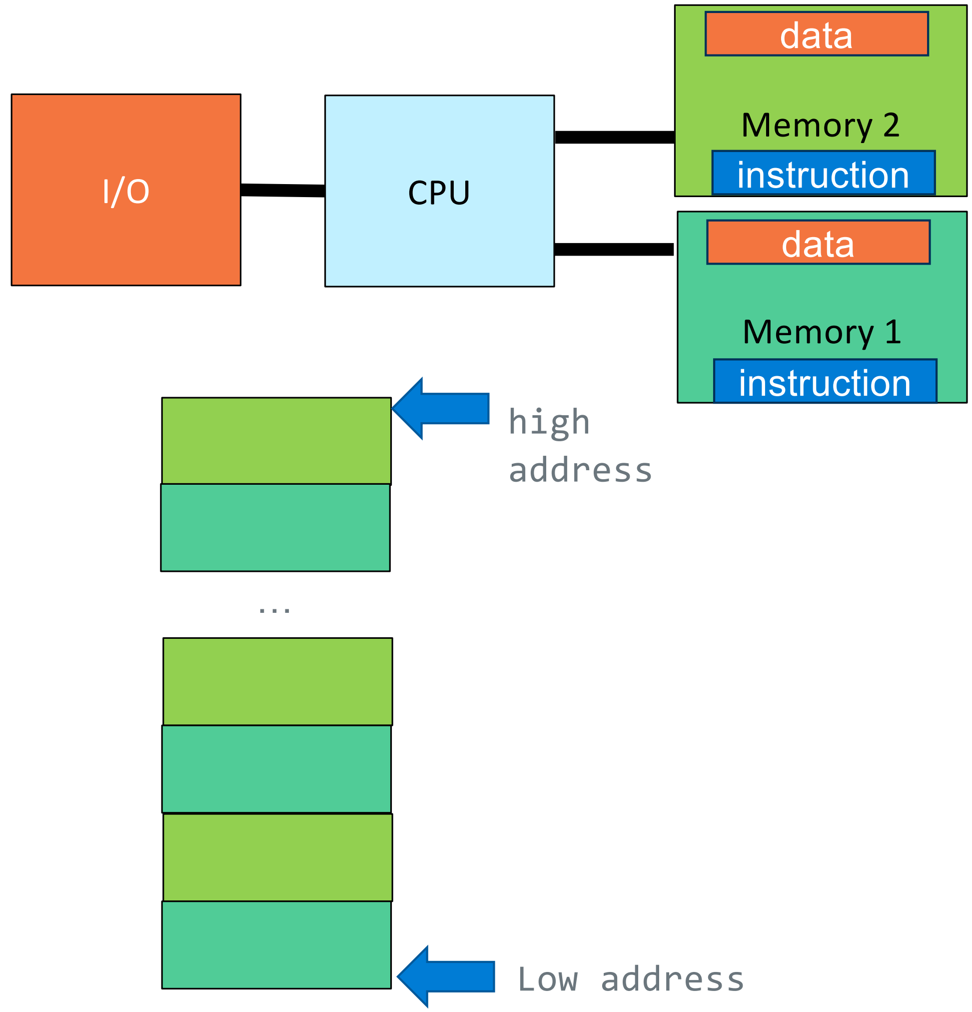


X

Machine Organization Example – Which Architecture is it?

- A good exam question
- Answer: Either you must be told where the Instructions and data are placed
- How can this be a Harvard architecture?

- Harvard Architecture: Use physical **memory interleaving** to achieve the performance increase with having to scale and size two different memory subsystems
- The size of the interleave is some multiple of bytes (like 1024)



x

C, Assembly and Executable Programs

- **Assembly language** is a symbolic version of the **machine code (language)**
  - **Instructions** describe operations the hardware can perform (e.g., =, +, -, *)
  - **Unique to a specific ISA**: e.g., ARM-32 versus IA-64
  - May be stored in a human readable text file
  - You can write in assembly language just like C or Java
    - Assembly is much easier to program than machine code
- A **high-level language** (like C) is **compiled** into an **assembly language equivalent**
  - A statement in C is represented by a sequence of one or more assembly language instructions (why a do you think it is a sequence?)
- **Assembly language program**
  - assembly language program is **translated (assembled)** into **machine code**
- An **executable program** contains
  - **series of instructions in machine code** (the program)
  - (maybe some) **data** to operate on

x

From Source code to Execution

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
int main (void)
{
    printf("Hello!\n");
    return EXIT_SUCCESS;
}
$ gcc -Wall -Wextra -Werror -c -S test.c
$ ls -ls
total 8
4 -rw-r--r-- 1 kmuller kmuller 109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller 725 Mar 14 15:58 test.s
$ gcc test.s
$ ls -ls
total 16
8 -rwxr-xr-x 1 kmuller kmuller 7708 Mar 14 15:58 a.out
4 -rw-r--r-- 1 kmuller kmuller  109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller  725 Mar 14 15:58 test.s
$ ./a.out
Hello!
```

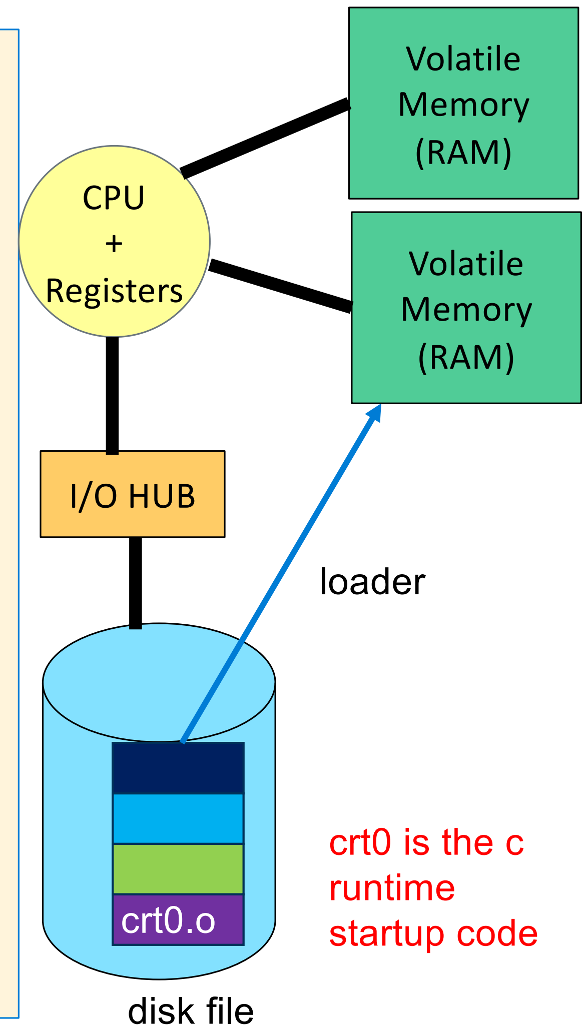
Source to Execution Steps

1. **Compile** (c source to assembler)
2. **Assemble** (assembler source to object)
3. **Link** (Combine object files to executable)
4. **Load** (Copy executable from into memory)
5. **Execute** (OS runs the code)

compile: -S -c tells the compiler to only compile to the assembly file
without –S –c compiles + assembles + links in one step
then the next step is not needed

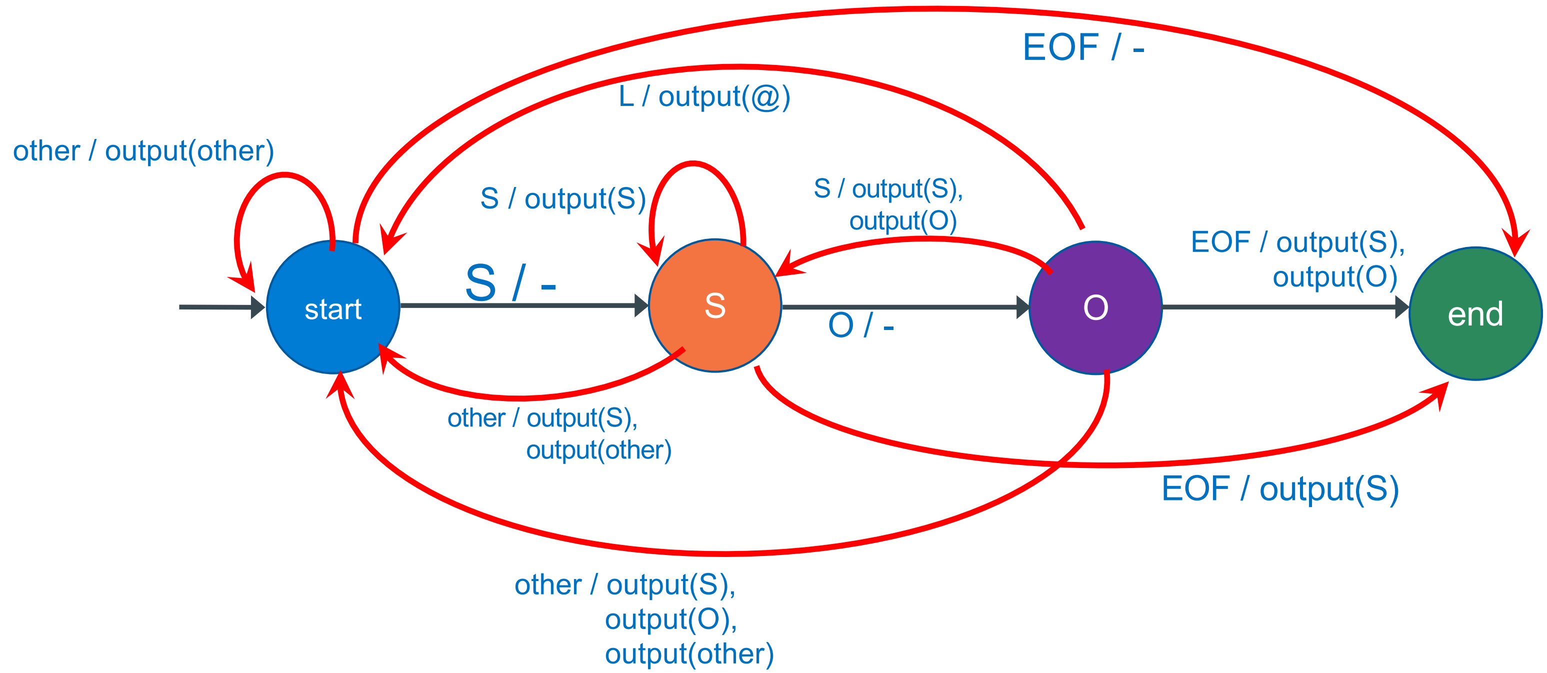
assemble and link the
gcc automatically calls the assembler with .S or .s files

load and then execute



x

Merging DFA's: Step one design each sequence -1



This DFA replaces SOL with a @

x

cpp conditional (and macro) only operations

- You can use **conditional preprocessor tests** (like if-else statements) around blocks of code
  `#ifdef MACRO, #ifndef MACRO, #else, #endif`
- In this use, MACRO is called the guard MACRO ("guards" entry to the following block)

`#ifdef MACRO` if MACRO is defined, then the block is included, otherwise the `#else` block (if any) is included

`#ifndef MACRO` if MACRO is NOT defined, then the block is included, otherwise the `#else` block (if any) is included

`#endif` is the end of a block

```
#define MACRO          // defines MACRO  -- #define MACRO 8 defines macro and assigns a value of 8
#undef MACRO           // undefines MACRO
```

```
#define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    int x[8];
    ….
    return;
}
```

```
// #define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    short x[8];
    ….
    return;
}
```

First Look at Header Files (also called .h or "include" files)

- Header file: a file whose only purpose is to be **`#include`'d** by the **preprocessor**
  - Contains: **Exported (public) Interface declarations**
    - Examples: function prototypes, user defined types, global variable, macros, etc.
  - Used to import the public interface of another C source file
    - `#include` its header (interface) file
- **NEVER EVER use cpp to** `#include` a .c file, a .S or a .s file
- **Convention (strongly enforced):** header files use a .h filename extension (example: filename**.h**)
  - **Example**: Source file src.**c** exported (public) interface is in the header file src.**h**
- How to specify the file to be **`#include`'d**
  - <system-defined> are system header files (typically located under /usr/include/…)

    ```
    #include <stdio.h>   // located in /usr/include/stdio.h
    ```
  - "programmer-defined" header files usually in a relative Linux path (see `–I` flag to gcc)

    ```
    #include "else.h"    // looks in the current directory first
    ```
- **Convention:** `#include` directives are usually placed near the top of a source file above any code

x

Compilation Process Operations

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

// A simple C Program
int
main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

preprocessor: inserts and processes the contents of files here.
Inserts: Function protype for printf (later in course)
macro value for EXIT_SUCCESS
File locations: /usr/include/stdio.h & /usr/include/stdlib.h

preprocessor: replaces the line Comment with one blank

compiler generates assembly code to call the library function printf() and pass the string "Hello World!"

cpp: replaces EXIT_SUCCESS with 0 on Linux

compile: **gcc –Wall –Wextra prog.c -o prog**

1. cpp first processes the file (cpp is called by gcc)
2. Compiler (gcc) compiles main to assembly
3. Assembler (gas – called by gcc) translates the assembly to machine code
4. Linker (ld) merges the machine code for printf() (from a library) with your programs machine code to create the executable file **prog** (machine code)
   - -o specifies the name of the executable (default: **a.out**)

X

cpp conditional tests: header guards

- Header guards ensure that only **one copy of a .h file** is **included in a source file**
- **A Convention**: header guard (macro) NAME (all capital letters) is created as follows:
  - use the **filename of header file but** in all caps
  - **replace** the **period in** header file **name** with an _
  - Example: file sum.h header guard macro name is SUM_H

- How do you use "header guards" in your code?

```
#ifndef NAME_H          // first line in the file
#define NAME_H
      . . .
#endif                  // last line in the file
```

```
#include "sum.h"
// file ex.c
int main(void)
{
    int x[MAX];
```

ex.i

```
int main(void)
{
    int x[8];
```

header guards (two lines)

header guards (one line)

```
#ifndef SUM_H
#define SUM_H

#define MAX 8

#endif
```

file sum.h

```
#include "sum.h"
// file func.c
int func(void)
{
    int z[MAX];
```

func.i

```
int func(void)
{
    int z[8];
```


x

Background: What is a Definition?

- **Definition**: creates an instance of a *thing*
- There **must be exactly one** definition of each *function or variable* (no duplicates)
- **Function definition (compiler actions)**
  1. **creates code** you wrote in the functions body
  2. **allocates** memory to store the code
  3. **binds** the function name to the allocated memory
- **Variable definitions (compiler actions)**
  1. **allocates memory: generate code** to allocate space for local variables
  2. **initialize memory: generate code** to initialize the memory for local variables
  3. **binds (or associates)** the variable name to the allocated memory

x

C Function Definitions - 3

- In standard C, functions **cannot be nested (defined)** inside of another function (called *local functions in other languages)*

```
int outer(int i)
{
    int inner(int j) // do not do this, not in standard c
    {
    }
}
```

- **Assignment inside conditional test** with **a function call** (this is very common!)

```
if ((i = SomeFunction()) != 0)
   statement1;
else
   statement2;
```

assignment returns the value that is placed into the variable to the **left of the = sign**, then the test is made

x

Background: What is a Declaration?

Declaration: describes a ***thing*** – specifies types, **does not create** an instance

- **Each declaration** has an associated ***identifier*** (the name)

1. **Function prototype:** describes how to **write the code to call a function** defined elsewhere
   - **Identifier** is the **function name**
   1. Describes the **type of the function return value**
   2. Describes the **types of each of the parameters**
2. **Variable declaration:** describes how to **write the code to use a variable** in a statement
   - **Identifier** is the **variable name**
   - Describes the **type of a variable** that is **defined elsewhere**
3. **Derived and defined type description**
   - **Identifier** describes the derived/defined type
   - struct, arrays, plus others (covered later)

- An **identifier** may be **declared multiple times**, but **only defined once**
- A **definition is also a declaration in C**

x

Definitions and Declarations Use in C

You must **declare a function or variable <u>before</u> you use it**

- **Warning:** Use before declaration will implicitly cause types to default to be of type **int**

sumit() is BOTH defined and declared here

Independent Translation Unit: the granularity (unit) of source which is compiled or assembled

Default Definition and declaration ***range of validity:***

1. Restricted to the file (translation unit) where they are located **<u>and</u>**
2. **Start at the point** of definition or declaration in the file, stopping at the end of the source file (**translation unit**)

Observation: Requiring function order in a file is a pain….
(1) sum() must be defined in the same source files
(2) sum() appear before it is used by main()
Question: How do we remove this limitation?

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 8
int sumit(int max)
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
// observe sumit() is declared above main()
int main(void)
{
  printf("sum: %d\n", sumit(MAX));
  return EXIT_SUCCESS;
}
```

`i, sum,` are both defined and declared here

sumit() is used here

x

C and Scope Review

- **Scope**: **Range** (or the extent) of instructions over which a name/identifier **is allowed be referenced** by C instructions/statements
  1. **File Scope: Range is within** a **single source file** (**translation unit**)
  2. **Block Scope: Range is within** an **enclosing block** (for variables only)

```c
int global;                              // global variable with file scope

void                                     // function foo with file scope
foo(int parm)                            // parameter parm block scope begins
{                                        // function body (block) begins
    int i, j = 5;                        // variables with block scope
    for (int k = 0; k < 10; i++) {       // inner block scope
        // some code
    }
}                                        // function body ends
```


x

Nested Scope

- **Nested Scope:** When two different variables have the same name are in scope at the same time, the declaration (*remember **definitions are also declarations***) that appears in the inner scope **hides the declaration** that appears **in an outer scope**

```
void funcA(int n)                     // scope of the function parameter 'n' begins
{                                     // the body of the function begins
    ++n;                              // 'n' is in scope and refers to the function parameter
    // int n = 2;                     // error: cannot redeclare identifier in the same scope

    for(int n = 0; n < 10; ++n) {    // scope of loop-local 'n' begins
         printf("%d\n", n);          // prints 0 1 2 3 4 5 6 7 8 9
    }                                // scope of the loop-local 'n' ends

                                      // the function parameter 'n' is back in scope
    printf("%d\n", n);               // prints the value of the parameter

}                                    // scope of function parameter 'n' ends
```


x

C Variable Storage Lifetime

1. **Static Storage Lifetime:** valid while program is executing
   - Storage allocated and initialized **prior to program start** (implicit default = 0)

2. **Automatic Storage Lifetime:** valid while enclosing block is activated
   - **Storage allocated and is not implicitly initialized (value = unspecified)** by **executing code** when entering scope and **made available for reuse** by **executing code** when exiting scope
   - It is **not correct to say that automatic storage has been deallocated on exit** (it ***might be)*** but more often is *still part of your program* **and may be referenced from the viewpoint of the OS without causing a runtime fault** if you have an address (pointers later in course)
   - Contents of storage after exiting scope is not changed (why would C act this way?)

3. **Allocated Storage Lifetime:** valid from point of allocation until freed or program termination
   - Storage allocated by call to an allocator function (malloc() etc.) at runtime and is not implicitly initialized (value = garbage) - one allocator does initialize to zero at runtime calloc() – later in course

4. **Thread Storage Lifetime:** valid while thread is executing (not CSE 30)

x

Variables in C

- **Global variables**
  - **Defined at file scope** (outside of a block)
  - have static storage duration
  - global variables defined without an initial value default to 0 (set prior to program execution start)
  - global variables defined with an initial value are set at program start
- **Local (block scope, or automatic) variables** (including function parameter variables)
  - **Defined at block scope** (inside of a block)
  - have automatic storage duration, **with one exception (see below)**
  - block scope variables defined without an initial value have an **unspecified** initial value
  - block scope variables defined with an initial are set each time by code when the block is entered
  - All block scope variables become **unspecified** at block exit
- **Variable definitions preceded by the keyword** *`static`* always have static storage duration including variables defined with block scope (when used global variables it restricts scope – later slides)

```c
int global;          // global with static storage duration, initial value = 0
int foo(void)
{
    static int s;    // "local" with static storage duration, initial value = 0
    int x;           // "local" with automatic storage duration
}
```


x

Example:
Block scope (local) static storage duration variables

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int foo(void)
{
    static int s;  //static storage duration, set to 0 at program start
    return s += 1;
}

int main(void)
{
    for (int i = 0; i < MAX; i++)
        printf("%d ", foo());
    printf("\n");
    return EXIT_SUCCESS;
}
```

```
% ./a.out
1 2 3 4 5
%
```


x

Creating Public Interface files (header files)

- To enable a **source file** to **use any of the functions, global variables**, and **MACROS** defined in another file (separate translation unit)
  - You must create a file that exports all permitted accesses so the compiler can generate the correct code
- **Convention:** For each source file, `file.c`, the **public interface file** is `file.h`
- If a file has no external interfaces, then it does not need a .h file

file.h

declarations | exported information how to use functions etc. in file.h

file.c

definitions | the definitions of functions etc.

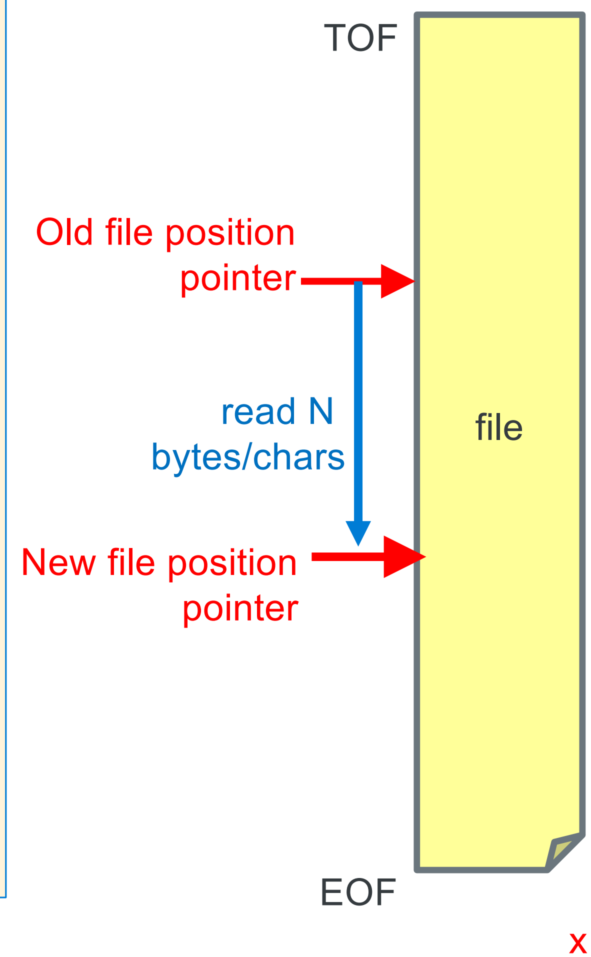
- `file.h` contains any
  - public preprocessor macros
  - **function prototypes** for the functions defined in the source file, `file.c` **that you want visible (exported)** for use (called) by functions defined in **other source files**
  - *global variable declarations (external linkage)*
  - **Do not put any** *definition statements* in a header file

- `file.c` contains
  - All function and global variable definitions (internal and external linkage)
  - Any private preprocessor macros
  - Any private (internal linkage) function prototypes

x

C standard I/O Library (stdio) File I/O
File Position Pointer and EOF

- Read/write functions in the standard I/O library *advances* the ***file position pointer*** from the ***top of a file*** (before the 1st byte if any) towards the ***end of the file*** **after each call** to a read/write function
  - **Side effect of call:** file position pointer moves towards the **end of file** by number of bytes read/written
- **standard I/O File position pointer** indicates where in the file (byte distance from the top of the file) the next read/write I/O will occur
- Performing a sequence of read/write operations (without using any other stdio functions to move the file pointer between the read/write calls) performs what is called **Sequential I/O** (sequential read & sequential write)
- EOF condition state may be set after a **read operation**
  - After the last byte is read in a file, additional reads results in a **function return value** of EOF
  - **EOF signals** no more data is available to be read
  - EOF is **NOT a character in the file**, but a condition state on the stream
  - EOF is usually a #define EOF -1 macro located in the file stdio.h (later in course)



C Library Function API : Simple Character I/O – Used in PA3

Operation	Usage Examples
Write a char	`int status; int c;` `status = putchar(c);          /* Writes to screen stdout */`
Read a char	`int c;` `c = getchar();                 /* Reads from keyboard stdin */`

```
#include <stdio.h>  // import the public interface
```

```
int putchar(int c);
```

- writes c (demoted to a char) **to** `stdout`
- returns either: c on success *OR* EOF (a macro often defined as -1) on failure
- see % man 3 putchar

```
int getchar(void);
```

- returns the next input character (if present) **promoted to an int** read **from** `stdin`
- see % man 3 getchar

- Make sure you use int variables with putchar() and putchar()
- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition (**-1) which is outside the range of valid characters

Why is character I/O using an int?

Answer: Needs to indicate an EOF (-1) condition that is not a valid char

x

Character I/O (Also the Primary loop in PA3)

```c
// copy stdin to stdout one char at a time
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int c;


    while ((c = getchar()) != EOF) {
        (void)putchar(c);   // ignore return value
    }

    return EXIT_SUCCESS;
}
```

Always check return code to handle EOF
EOF is a macro integer in stdio.h

```
% ./a.out
thIS is a TeSt
thIS is a TeSt
^d
%
%./a.out < a > b
```

Typed on keyboard

Printed by program

Typed on keyboard

Copies file a to file b

Always check return codes unless you do not need it

Sometimes you may see a (void) cast which indicates ***ignoring the return value is deliberate*** this is often required by many coding standards

Make sure you use int variable with getchar() and putchar()!

X

C Library Function: Simple Formatted Printing

Task	Example Function Calls

Write formatted data	`int status;`
	 `status = fprintf(stderr, "%d\n", i);`
	 `status = printf("%d\n", i);      /* Writes to stdout */`

```c
#include <stdio.h>  // import the public interface

int fprintf(FILE *file, const char *format, ...);
```

- Write chars to the file identified by **file** (stdout, stderr are already open)
- Convert values to chars, as directed by **format**
- Return count of chars successfully written
- **Format** is the output specifications enclosed in a "string"
- Returns a negative value if an error occurs

```c
int printf(const char *format, ...); // *format - later in course
```

- Equivalent to `fprintf(stdout, format, ...);`

- Type **%** `man 3 printf` for more information on *format*

x

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal** evaluation

```
if ((x == 5) || (y > 3))  // if x == 5 then y > 3 is not evaluated
```

- **Each** expression argument is evaluated **in sequence** from left to right including any side effects (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated *(for performance)*

x

Program Flow – Short Circuit or Minimal Evaluation

```
if ((a != 0) && func(b))     // if a is 0, func(b) is not called
    do_something();
```

```
// if (((x > 0) && (c == 'Q')) evaluates to non zero (true)
// then (b == 3) is not tested

while (((x > 0) && (c == 'Q')) || (b == 3)) {  // c short circuit
    x = x / 2;
    if (x == 0) {
        return 0;
    }
}
```


x

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$

$$\texttt{0x}\ \overbrace{1111}^{f}\ \overbrace{1010}^{a}\ \overbrace{0101}^{5}\ \overbrace{0011}^{3}$$

$$\boxed{\texttt{0b}1111101001010011}$$

binary start with a 0b in C

x

Binary to Hex (group 4 bits per digit from the right)

- 4 binary bits is one Hex digit $2^4 = 16^1$

$$0b\ \underbrace{0110}_{6}\ \underbrace{1010}_{a}\ \underbrace{0011}_{3}\ \underbrace{1111}_{f}$$

$$\boxed{0x6a3f}$$

hex start with 0x in C

x

sizeof(): Variable Size (number of bytes) *Operator*

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns a value of type size_t:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
        or
size_t size = sizeof(variable_name); // preferred!
```

- The argument to `sizeof()` is often an expression:

```
size = sizeof(int * 10);
```

  - reads as:
    - number of bytes required to store **10 integers (an array of [10])**

x

Memory Addresses & Memory Content

Variable names in a C statement evaluation

```
x = x + 1;     // Lvalue = Rvalue
```

- **Lvalue:** when on the left side (Lside or Left value) of the = sign
  - address where it is stored in memory – a constant
  - **Address assigned to a variable cannot be changed at runtime**
  - Does not require a memory read
  - **Lside Must evaluate to an address**
- **Rvalue:** when on the right side (Rside or Right value) of an = sign
  - **contents or value stored** in the variable (at its memory address)
  - requires a memory read to obtain contents

x

Introduction: Address Operator: &

- Requirement: **identifier must have a Lvalue**
  - Cannot be used with constants (e.g., 12) or expressions (e.g., x + y)
  - Example: **&12** does not have an *Lvalue*,
    - so, &12 is **not** a legal expression
- How can I get an address for use on the **Rside**?
  - **&var** (any variable identifier or name)
  - **function_name** (name of a function, not func());
    - &funct_name is equivalent
  - **array_name** (name of the array like array_name[5]);
    - &array_name is equivalent

x

Pointer Variables - 2

- A pointer **cannot** point at itself, why?

```
int *p = &p; /* is not legal – type mismatch */
```

  - p is defined as (int *), a pointer to an int, **but**
  - the type of &p is (int **), a pointer to a pointer to an int
- Pointer variables typically use the **same amount of memory** no matter what they point at (in all but very tiny special purpose, often old design, cpu's)

```
int *iptr;
char *cptr;

printf("iptr(%u) cptr(%u)\n", sizeof(iptr), sizeof(cptr));
```

- Above prints on a 32-raspberry pi

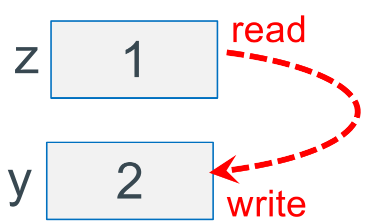
```
% ./example
iptr(4) cptr(4)
```

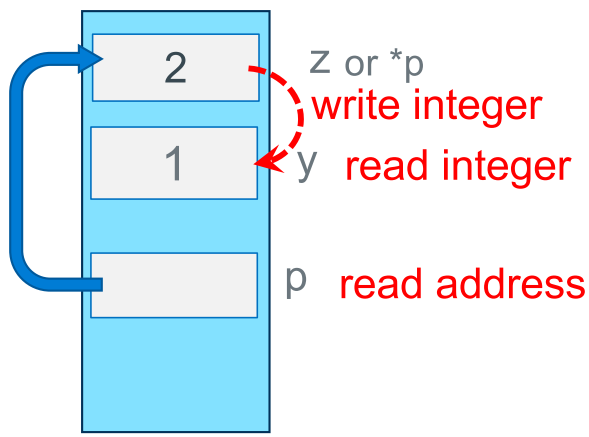

x

Each use of a * operator results in one additional read: Rside

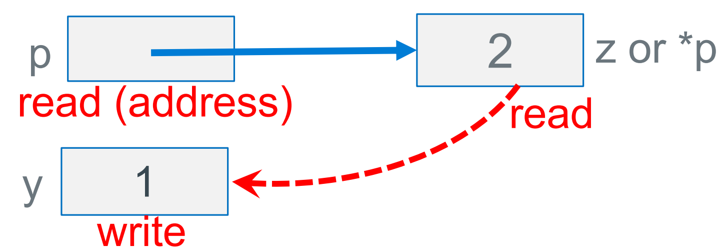
RULE: Each * when used as a dereference operator in a statement (either Lside or Rside) it causes an <u>additional</u> read to be performed

```
int z = 2;
int y = 1;

y = z; // one read
```






```
int z = 2;
int y = 1;
int *p;

p = &z;
y = *p; // two reads on rside
```

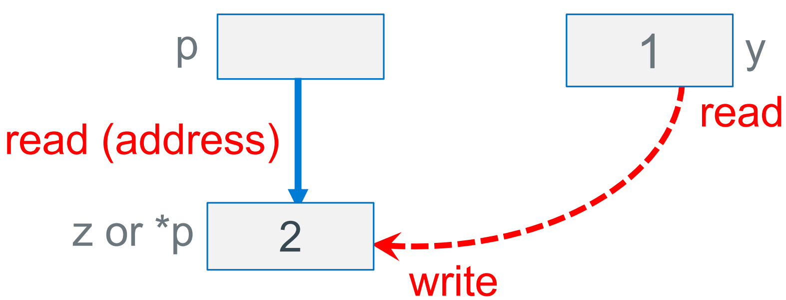


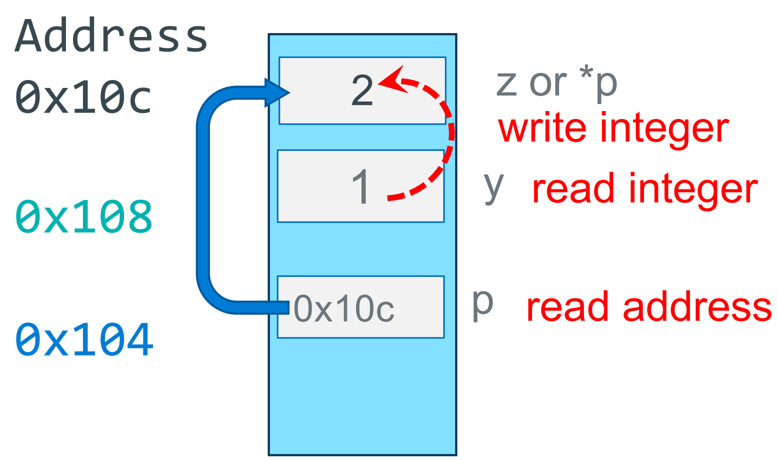
Aside: `y = *(&z); // same as y = z`

x

Each use of a * operator results in one additional read: Lside

```
int z = 2;
int y = 1;
int *p;

p = &z;
*p = y;     // one read on lside
```

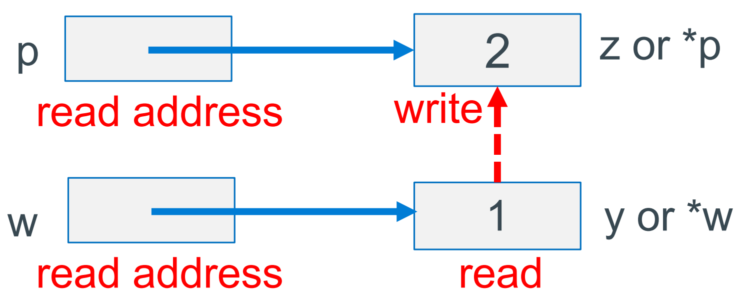


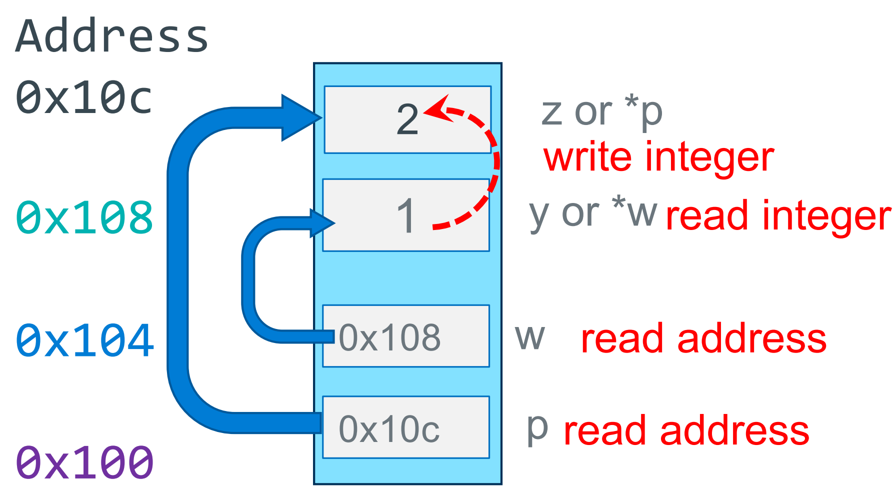


x

Each use of a * operator results in one additional read : both sides

```
int z = 2;
int y = 1;
int *w;
int *p;

p = &z;
w = &y;
*p = *w;
```

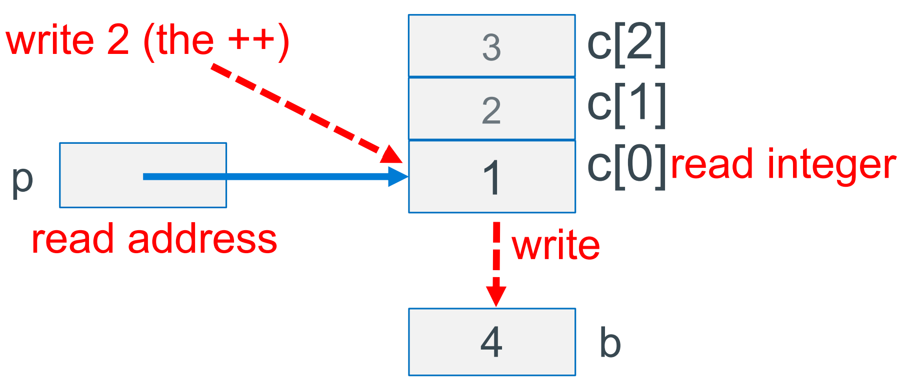


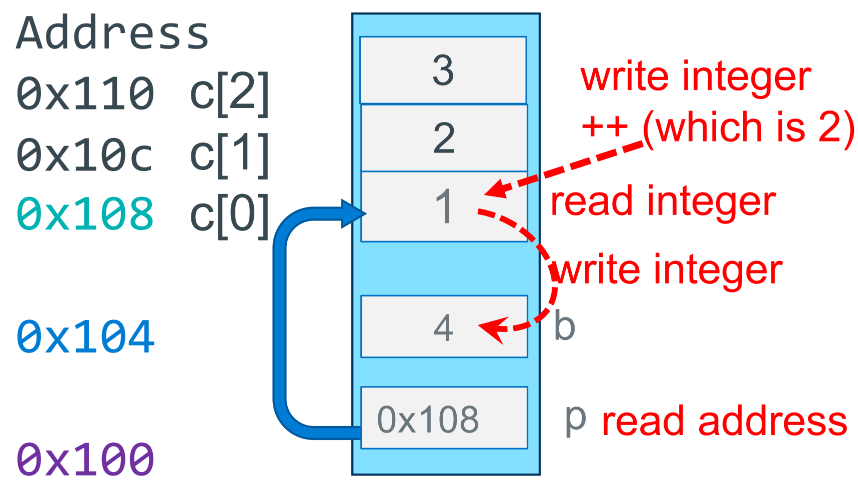


x

Each use of a * operator results in one additional read : both sides

```
int c[] = {1, 2, 3};
int b = 4;
int *p;

p = c;
b = (*p)++;
```



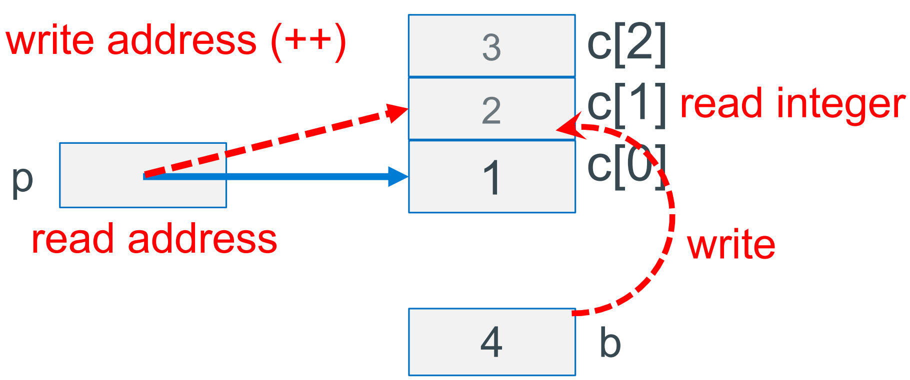


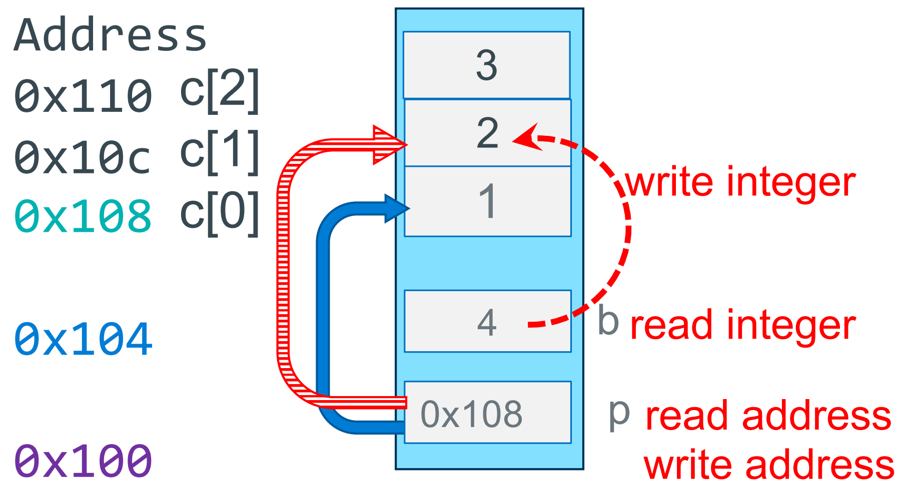
2 reads and 2 writes

x

Each use of a * operator results in one additional read : both sides

```
int c[] = {1, 2, 3};
int b = 4;
int *p;

p = c;
*(++p) = b;
```





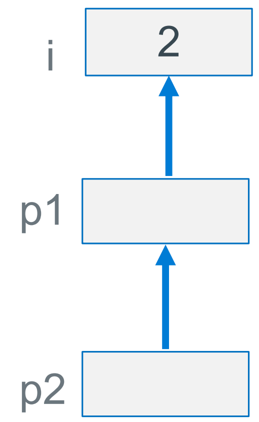
2 reads and 2 writes

x

Pointer to Pointers (Double Indirection)

- Define a pointer to a pointer (p2 below)

```
int i = 2;
int *p1;
int **p2; // pointer to a pointer to an int

p1 = &i;
p2 = &p1;
printf("%d\n", (**p2) * (**p2));
```



- C allows any number of pointer indirections
  - more than two levels is very uncommon in real applications as it reduces readability and generates at lot of memory reads
- **RULE (important):** number of * **in the variable definition** tells you **how many reads it takes to get to the base type**

  #reads to base type = number of * (in the definition) + 1

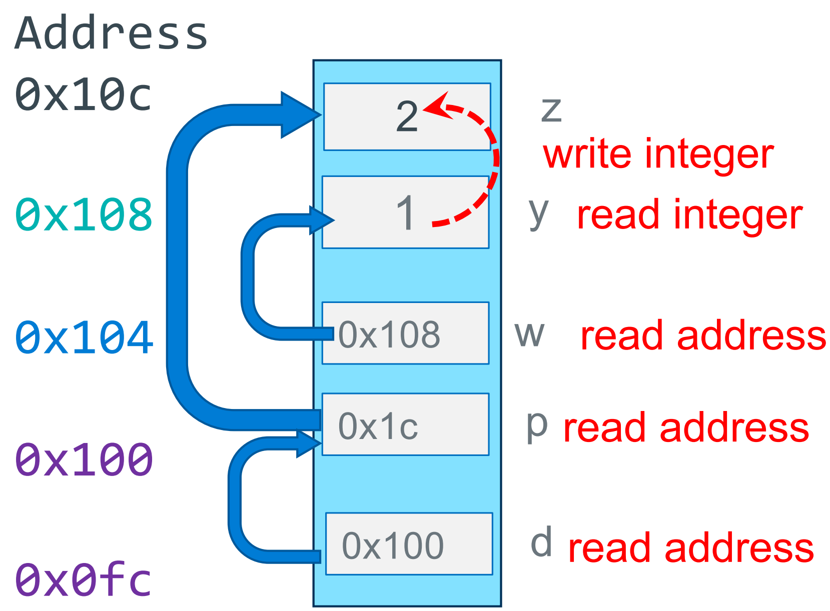
- Example:

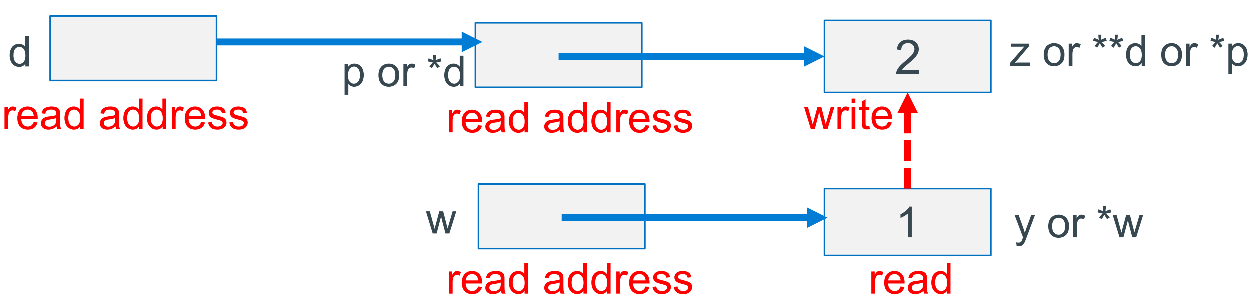
  `int **p2;      //` requires 3 reads to get to the int

x

Double Indirection: Lside

```
int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = *w;
```

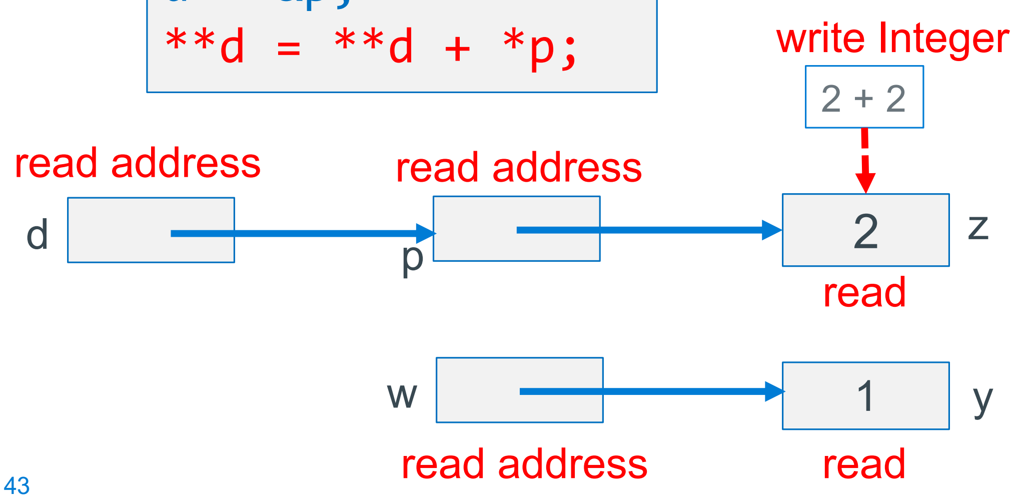


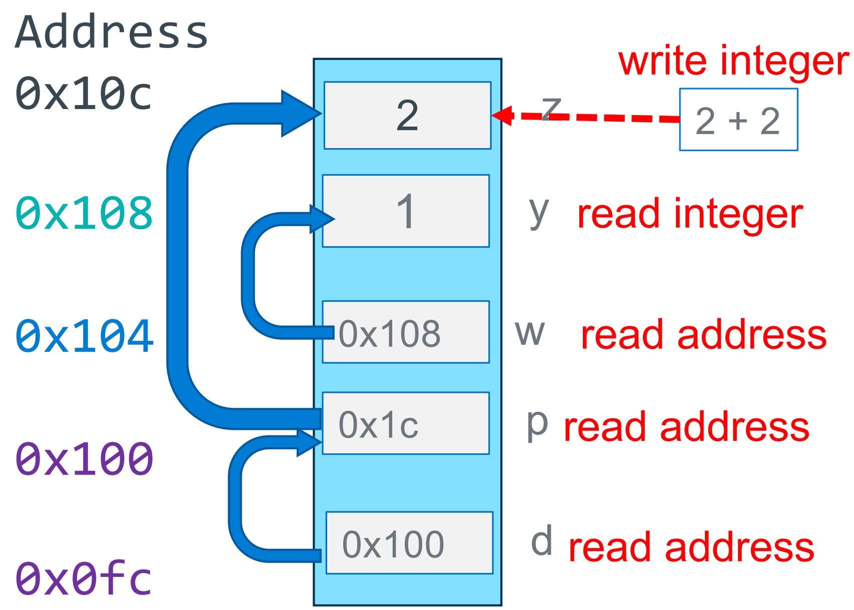


x

Double Indirection: Rside

```
int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = **d + *p;
```





Important Observation

**d on Lside is two reads

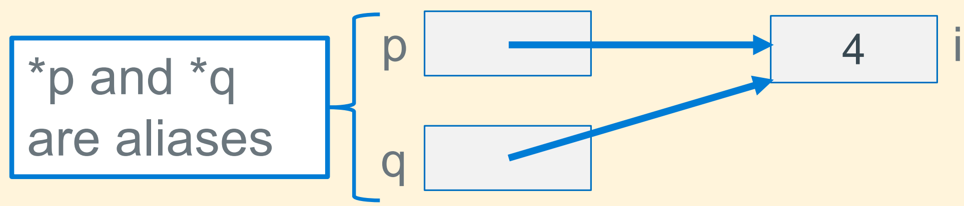
**d on Rside is three reads

x

What is Aliasing?

- Two or more variables are aliases of each other when they all reference the same memory (so different names, same memory location)
- **Example**: When one pointer is copied to another pointer it *creates an* ***alias***
- ***Side effect***: Changing one variables value (content) changes the value for other variables
  - Multiple variables all read and write the **same** memory location
  - Aliases occur either by accident (coding errors) or deliberate (careful: readability)

```c
int i = 5;
int *p;
int *q;

p = &i;
q = p;    // *p & *q now aliases
*q = 4;   // changes i and *p
```



Result *p, *q and i all have the value of 4

x

Determining Element Count: compile time calculation

- Programmatically determining the element count in a compiler calculated array

  `sizeof(array) / sizeof(of just one element in the array)`

- `sizeof(array)` **only works** when used in the SAME **scope** where the array variable was defined

```
#include <stddef.h>
int main()
{

    int block[] =
       {2, 3, 5, 6, 11, 13};    // automatic: compiler calculates array size

    int cnt = (int)(sizeof(block) / sizeof(block[0]));    // in this case cnt = 6

    for (int indx = 0; indx < cnt; indx++)
       block[indx] = 0;
```


x

Pointers and Arrays - 1

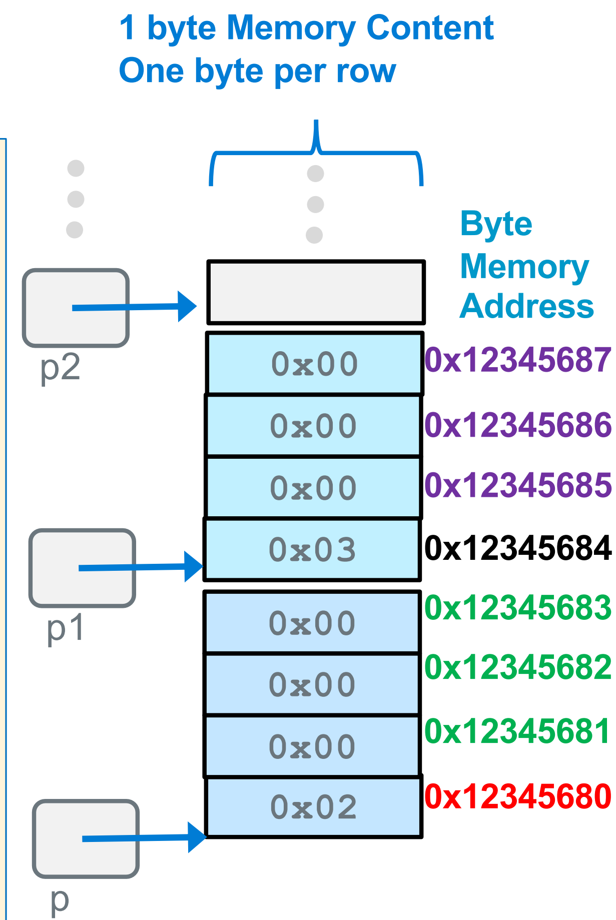
- A few slides back we stated: Array name (by itself) on the Rside evaluates to the address of the first element of the array

```
int buf[] = {2, 3, 5, 6, 11};
```

- Array indexing syntax ([ ]) an operator that performs *pointer arithmetic*

- **buf and &buf[0]** on the **Rside are equivalent**, ***both evaluate*** to the address of the first array element

```
int *p = buf;           // or int *p = &buf[0];
int *p1 = &buf[1];
int *p2 = &buf[2];
int *p3 = &buf[3];
```

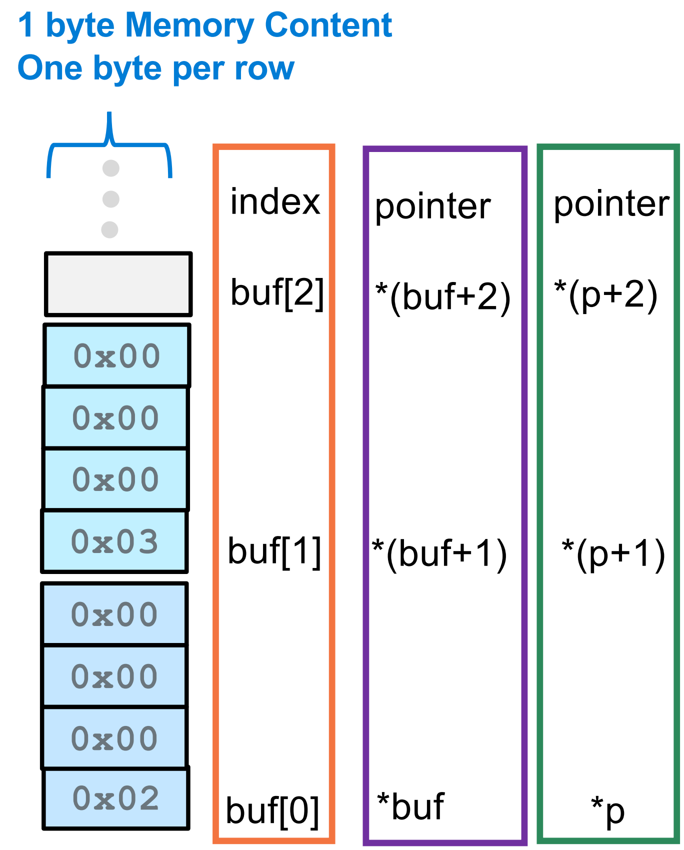


x

Pointers and Arrays - 2

- When p is a pointer, the actual evaluation of the address:
  - (p+1) **depends on the base type** the pointer p points at
- (p+1) adds 1 x sizeof(what p **points at**) bytes to p
  - ++p is equivalent to p = p + 1
- Using pointer arithmetic to find array elements:
  - Address of the second element &buf[1] is (buf + 1)
  - It can be referenced as *(buf + 1) or buf[1]

```
int buf[] = {2, 3, 5, 6, 11};
int *p;
p = buf;

*p = *p + 10;                  // {12, 3, 5, 6, 11}
*(p + 1) = *(buf + 1) + 10;   // {12, 13, 5, 6, 11}
```

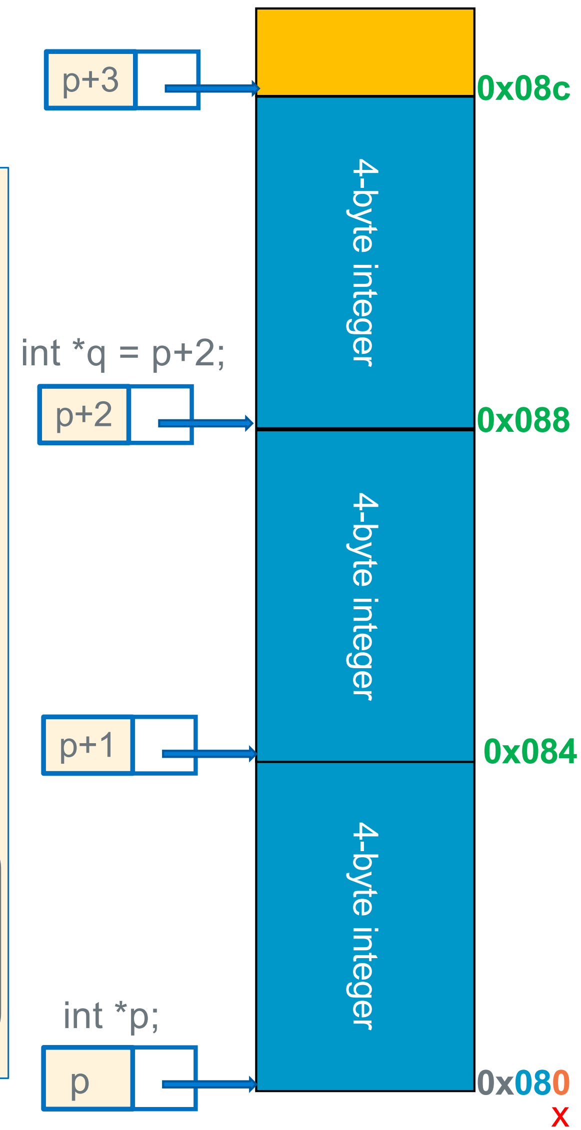


x

Pointer Arithmetic

- **You <u>cannot</u> add two pointers** *(what is the reason?)*
- A pointer q <u>can be subtracted</u> from another pointer p when the pointers are the same type – best done only within arrays!
- The value of **`(p-q)`** is the number of **elements between** the two pointers
  - Using memory address arithmetic (p and q Rside are both byte addresses):
  - Notice that it is sizeof(*p) below: it is what **p points at** and not **sizeof(p) which is the size of the pointer**!

```
distance in elements = (p - q) / sizeof(*p)

(p + 3) - p = 3 = (0x08c - 0x080)/4 = 3
```



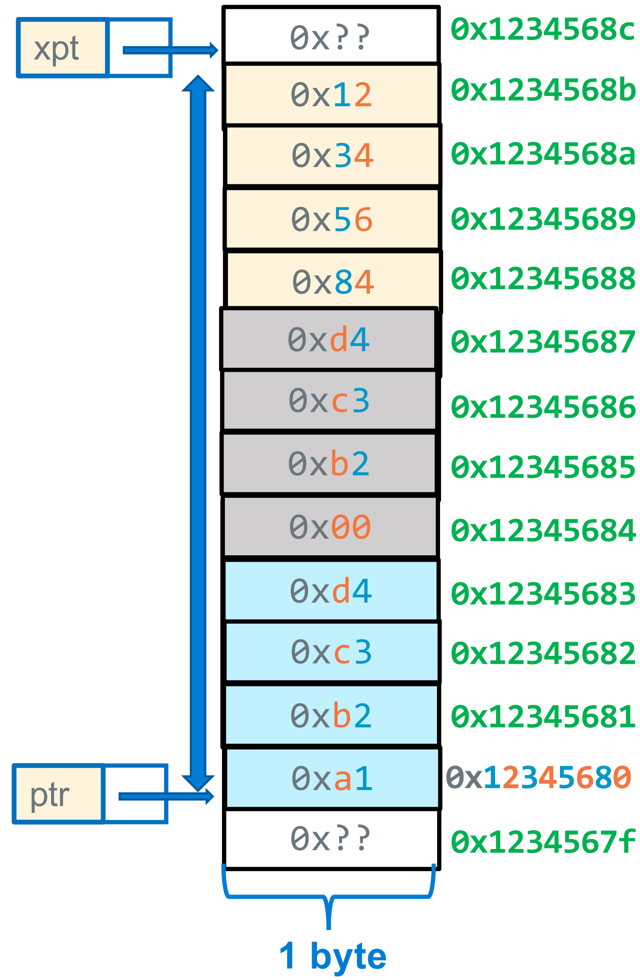
Fast Ways to Traverse an Array: Use a Limit Pointer

```c
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

int *ptr;
int *xptr;
ptr = x;                    //or &x[0]
xptr = ptr + cnt;
```

xpt is a **loop limit pointer** it points **1 element past** the end of the array

```c
while (ptr < xptr) {
    printf("%#x\n", *ptr);
    ptr++;
}
```

```
 % ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```



x

C Precedence and Pointers

- ++ -- pre and post increment combined with pointers can create code that is complex, hard to read and difficult to maintain
- Use () to help readability

common	With Parentheses	Meaning
`*p++`	`*(p++)`	(1)The Rvalue is the object that p points at (2)increment pointer p to next element ++ is higher than *
`(*p)++`		(1)Rvalue is the object that p points at (2)increment the object
`*++p`	`*(++p)`	(1)Increment pointer p first to the next element (2)Rvalue is the object that the incremented pointer points at
`++*p`	`++(*p)`	Rvalue is the incremented value of the object that p points at

Operator	*Description*	*Associativity*
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

X

C Strings - 1

- **C** **<u>does not</u>** have a **dedicated type** for strings
- **Strings are** an **array of characters terminated by** a sentinel termination **character**
- `'\0'` is the **Null termination character;** has the **value of zero (do not confuse with '0')**
- An **array of chars** contains **a string only <u>when</u>** it is terminated by a '\0'
- **Length of a string** is the number of characters in it, <u>not including</u> the '\0'
- Strings in C are **<u>not</u>** objects
  - No embedded information about them, you just have a name and a memory location
  - You cannot use **+** or **+=** to concatenate strings in C
  - For example, you must **calculate string length** using code at runtime looking for the sentinel

length of the string: number of char (= 5)

index	*0*	*1*	*2*	*3*	*4*	*5*
char	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`'\0'`

length (size) of the array in number of char (= 6)

x

C Strings - 2

- **First `'\0'` <u>encountered</u> from the start of the string** always indicates the end of a string
- The **'\0' does not have to be** in the **last element in the space allocated to the array**
  - But String length is always less than the size of the array it is contained in
- In the example below, the array buf contains <u>two strings</u> (but only *cat* is seen as the string)
  - One string starts at &(buf[0]) is "cat" with a string length of 3
  - The other string starts at &(b[4]) is "o" with a string length of 1
  - "o" has two bytes: 'o' and '\0'

string length: number of char (= 3)

string length: number of char (= 1)

index	*0*	*1*	*2*	*3*	*4*	*5*
buf	`'c'`	`'a'`	`'t'`	`'\0'`	`'o'`	`'\0'`
	`0x63`	`0x61`	`0x74`	`0x00`	`0x6f`	`0x00`

length (size) of the array in number of char (= 6)

Defining Strings: Initialization

- When you combine the automatic length definition for arrays with double quote(") **initialization**
  - Compiler automatically adds the null terminator '\0' for you

```
char a[4] = {'c', 'a', 't', '\0'};
char b[] = "cat";                          // compiler calculates size, adds '\0'
char c[] = {'c', 'a', 't', '\0', 'a, 'b'}; // array size 6, first string length 3
char empty[] = "";                         // empty string - contains '\0'
                                           // string length = 0
```


x

Output Parameters (Mimics Call by Reference)

- Passing a pointer parameter with the **intent** that the called function will use the address it to store values for use by the calling function, then pointer parameter is called an **output parameter**
- To pass the address of a variable x use the **address operator** (&x) **or** the contents of a pointer variable that points at x, or the name of an array (the arrays address)
- To be receive an address in the called function, define the corresponding parameter type to be a pointer (add *)
  - It is common to describe this method as: "pass a pointer to x"
- C is still using "*pass by value*"
  - we pass the **value** of the address/pointer in a **parameter copy**
  - **The called routine** uses the address to change a variable in the caller's scope

```
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);


}

void
inc(int *p)
{



}
```


x

Example Using Output Parameters

```
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
    printf("%d\n", x);
    return EXIT_SUCCESS;
}

void
inc(int *p)
{
    if (p != NULL)
        *p += 1;    // or (*p)++
}
```

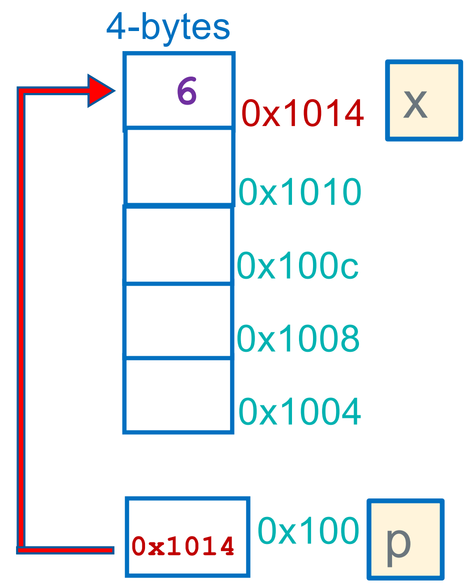
Pass the address of x (&x)

Receive an address copy in the variable p (int *p)

Write to the output variable (*p)

At the Call to inc() in main()

1. Allocate space for p
2. Copy x's address into p



With a pointer to X,

inc() can change x in main()

this is called a side effect

p just like any other local variable

x

Array Parameters: Call-By-Value or Call-By-Reference?

- **`Type[]`** array parameter is automatically "**promoted**" to a pointer of type **`Type *`**, and a **copy** of the ***pointer*** is ***passed by value***

```
void passa(int []);
int main(void)
{
  int numbers[] = {9, 8, 1, 9, 5};

  passa(numbers);
  printf("numbers size:%lu\n", sizeof(numbers)); // 20
  return EXIT_SUCCESS;
}
```

the name is the address, so this is passing a pointer to the start of the array

```
void passa(int a[])
{
    printf("a size:%lu\n", sizeof(a)); // 4
    return;
}
```

IMPORTANT:
See the size difference 20 in main() in passa() is 4 bytes (size of a pointer) on pi-cluster and 8 on ieng6

- Call-by-value pointer (callee can change the pointer parameter to point to something else!)
- Acts like call-by-reference (called function can change the contents caller's array)

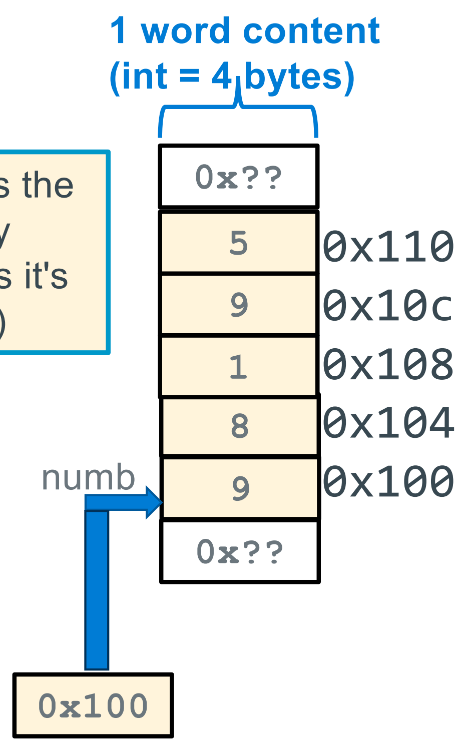
X

Arrays As Parameters: What is the size of the array?

- It's tricky to use arrays as parameters, as **they are passed as pointers to the start of the array**
  - In C, **Arrays do not know their own size** and at runtime there is no "bounds" checking on indexes

Observe numb is the name of an array (whose Rvalue is it's starting address)

Remember a is parameter copy so is a separate variable that contains a pointer to num




```c
int sumAll(int *);

int main(void)
{
   int numb[] = {9, 8, 1, 9, 5};
   int sum = sumAll(numb);

   return EXIT_SUCCESS;
}

int sumAll(int *a)
{
   int i, sum = 0;
   int sz = (int) (sizeof(a)/sizeof(*a));
   for (i = 0; i < sz; i++) // this does not work
        sum += a[i];
   }
}
```

this is a POINTER to the first element…..
so sizeof(a) is the size of a pointer, not the array it points at
Net result:
sz is 4/4 = 1 on picluster

x

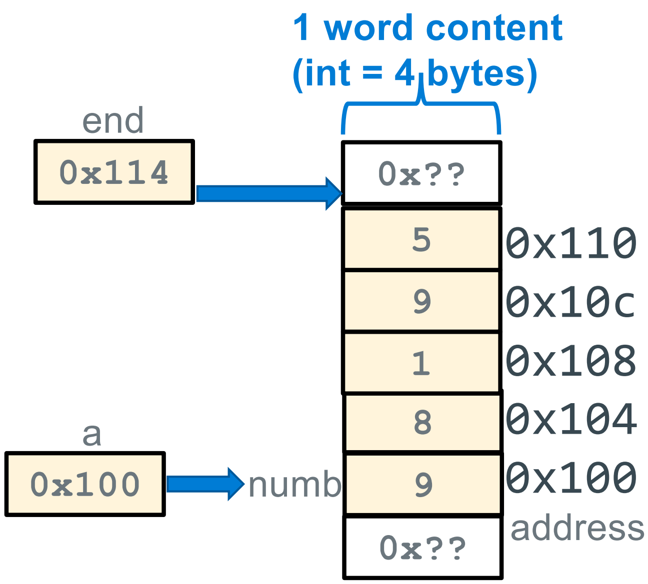
Arrays As Parameters, Approach 1: Pass the size

Two ways to pass array size

1. pass the count as an additional argument
2. add a sentinel element as the last element

remember you can only use sizeof() to calculate element count where the array is <u>defined</u>

```c
int sumAll(int *a, int size);
int main(void)
{
  int numb[] = {9, 8, 1, 9, 5};
  int cnt = (int)(sizeof(numb)/sizeof(numb[0]));

  printf("sum is: %d\n", sumAll(numb, cnt););
  return EXIT_SUCCESS;
}
```




```c
int sumAll(int *a, int size)
{
  int sum = 0;
  int *end;
  end = a + size;

  while (a < end)
    sum += *a++;
  return sum;
}
```

same as:
sum = sum + *a;
a++;

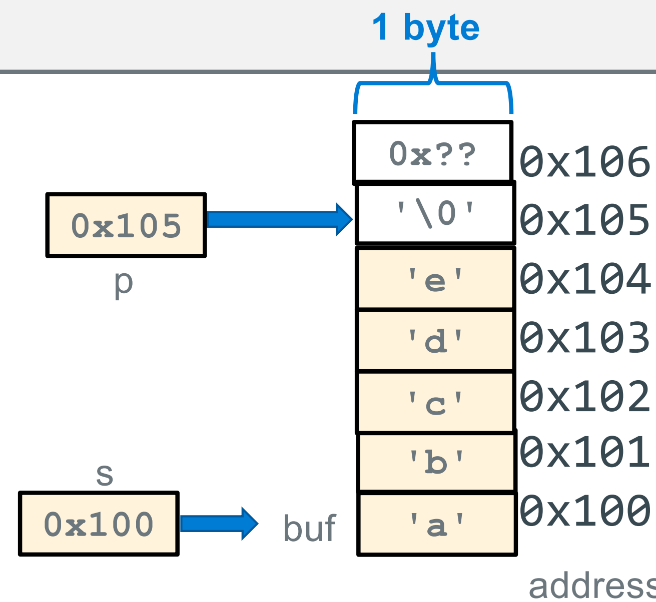
x

Arrays As Parameters, Approach 2: Use a sentinel element

- A sentinel is an element that contains a value that is not part of the normal data range
  - Forms of 0 are often used (like with strings). Examples: '\0', NULL

```
int strlen(char *a);  // returns number of chars in string, not counting \0
int main(void)
{
  char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // or buf[] = "abcde";

  printf("Number of chars is: %d\n", strlen(buf));
  return EXIT_SUCCESS;
}
```

```
/* Assumes parameter is a terminated string */
int strlen(char *s)
{
   char *p = s;
   if (p == NULL)
       return 0;
   while (*p != '\0')
       p++;
    return (p - s);
}
```



x

The NULL Constant and Pointers

- **NULL is a constant** that **evaluates to zero (0)**
- You assign a pointer variable to contain NULL to indicate that the pointer does not point at anything
- A pointer variable with a value of NULL is called a "NULL pointer" (invalid address!)
- Memory location 0 (address is 0) is not a valid memory address in any C program
- Dereferencing NULL at runtime will cause a program fault (segmentation fault)!

```
p = NULL;
i = *p;                 /* segmentation fault! */
*(int *)900000 = 25;    /* cast 900000 to a pointer */
                        /* if writeable address space, it works */
                        /* that memory location just changed */
```


x

Simple String IO - Reading

Task	Example Function Calls

Read a string	`#include <stdio.h>`
	`char *strpt;`
	`char myStr[BFSZ];`
	`strptr = fgets(myStr, BFSZ, stdin);`

must pass the size of the array so fgets() knows how much space there is

```
char *fgets(char array[ ], int size, FILE *stream)
```

- `char *` is a pointer (address) to an **array of char**
- reads in at most **one less than** ***size*** characters from ***stream*** and stores them into **array**
- Reading stops after an **EOF** or a newline '\n'
  - If a newline ('\n') is read, it is stored into the buffer
  - **A terminating null byte ('\0') is always stored after the last character in the buffer**

t	h	i	s		i	s		a		s	t	r	i	n	g	\n	\0

- Returns a **NULL at end of file** (or a read failure), otherwise a pointer to array (pointers later…)
- See `man 3 fgets`

X

Returning a Pointer To a Local Variable (Dangling Pointer)

- There are many situations where a function will return a pointer, but a function must never return a pointer to a memory location that is no longer valid such as:

1. Address of a passed parameter copy as the caller may or will deallocate it after the call
2. Address of a local variable (automatic) that is invalid on function return

- These errors are called a dangling pointer

n is a parameter with the scope of bad_idea it is no longer valid after the function returns

```
int *bad_idea(int n)
{
    return &n; // NEVER do this
}
```

a is an automatic (local) with a scope and **lifetime** within bad_idea2 a is no longer a valid location after the function returns

```
int *bad_idea2(int n)
{
    int a = n * n;
    return &a; // NEVER do this
}
```

```
/*
 * this is ok to do
 * it is NOT a dangling
 * pointer
 */

int *ok(int n)
{
    static int a = n * n;
    return &a; // ok
}
```


X

String Literals (Read-Only) in Expressions

- When strings in quotations (*e.g.*, "string") are **part of** an **expression** (*i.e.*, *not* part of an *array initialization*) they are called ***string literals***

```
printf("literal\n");
printf("literal %s\n", "another literal");
```

- What is a ***string literal:***
  - Is a null-terminated string in a **`const char array`**
  - Located in the **read-only data** segment of memory
  - Is not assigned a variable name by the compiler, so it is only accessible by the location in memory where it is stored
- **String literals** are a type of ***anonymous variable***
  - Memory containing data without a name bound to them (only the **address** is known)
- The *string literal* in the printf()'s, are replaced with the starting address of the corresponding array (first or [0] element) when the code is compiled

x

String Literals, Mutable and Immutable arrays - 1

- mess1 is a **mutable** array (type is char [ ]) with enough space to hold the string + '\0'

```
char mess1[] = "Hello World";
*(mess1 + 5) = '\0'; // shortens string to "Hello"
```

mess1[ ] `Hello World\0`

- mess2 is a **pointer** to an **immutable** array with space to hold the string + '\0'

```
char *mess2 = "Hello World";      // "Hello World" read only string literal
                                  // mess2 is a pointer NOT an array!
*(mess2 + 1)  = '\0';             // Not OK (bus error)
```

mess2 → `Hello World\0` ← read only string literal

- mess3 is a pointer to a mutable array

```
char *mess3 = (char []) {"Hello World"};  // mutable string
*(mess3 + 1)  = '\0';                     // ok
```

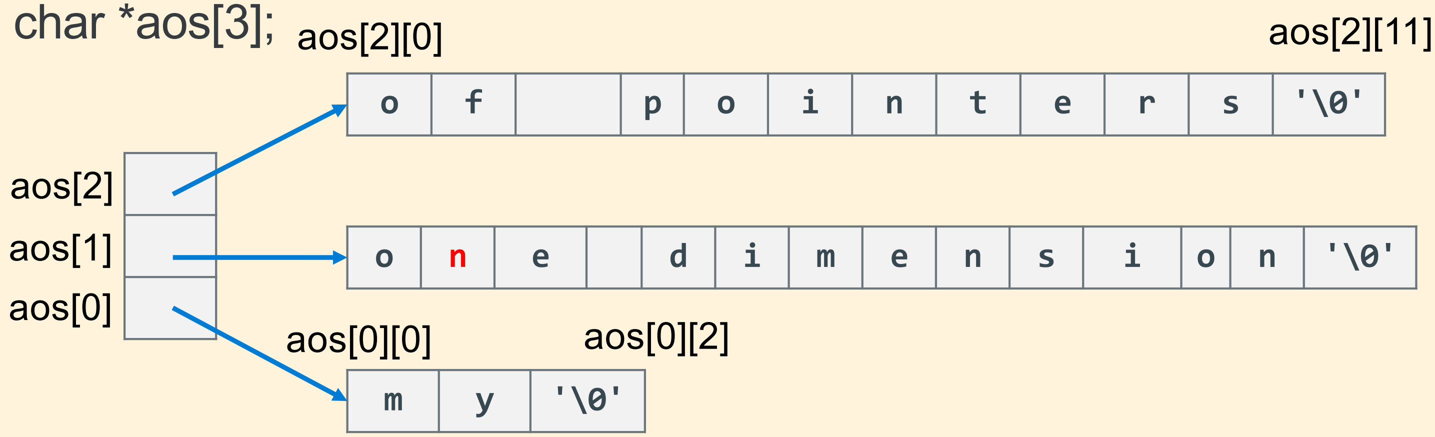
using the cast (char [ ]) makes it mutable

mess3 → `Hello World\0` ← mutable string

x

Array of Pointers to Strings (This is NOT a 2D array)

- 2D char arrays are an inefficient way to store strings (wastes memory) unless all the strings are similar lengths, so 2D char arrays *are rarely used* with string elements
- **An array of pointers** is common for strings as *"rows"* can very in length
- char *aos[3];



- **`aos`** is an array of pointers; each pointer points at a character array (also a string here)
- Not a 2D array, but any char can be accessed as if it was in a 2D array of chars
  - When I was learning, this was the most confusing syntax aspects of C

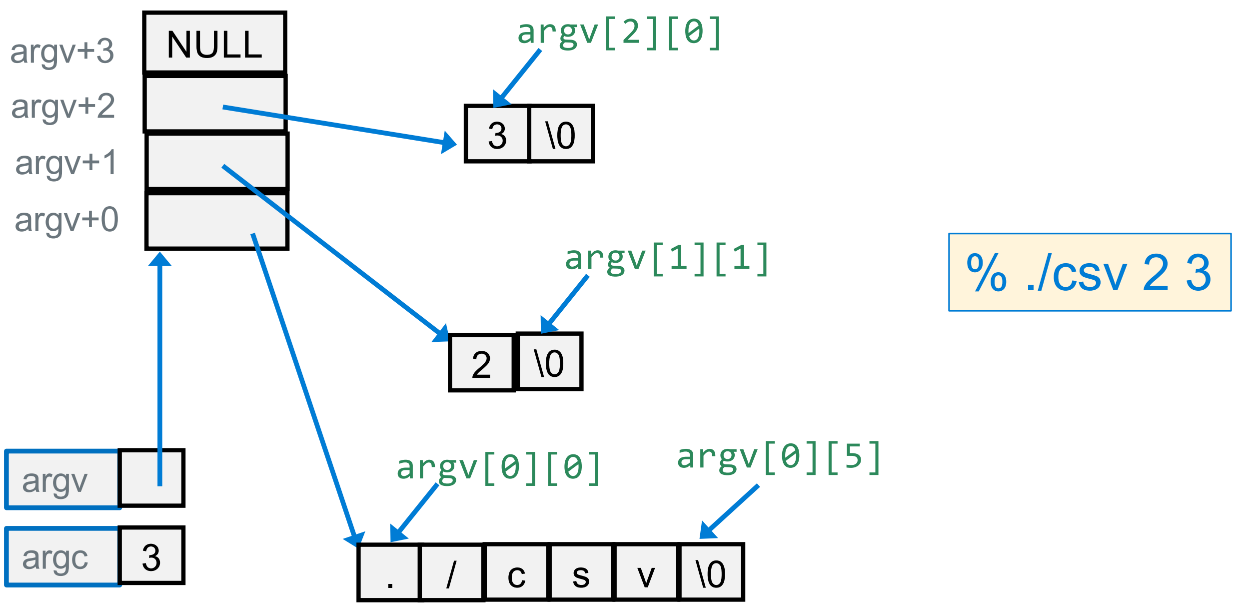
x

main() Command line arguments: argc, argv

- Arguments are passed to main() as a pointer to an array of pointers to char arrays (strings)`(**argv)`

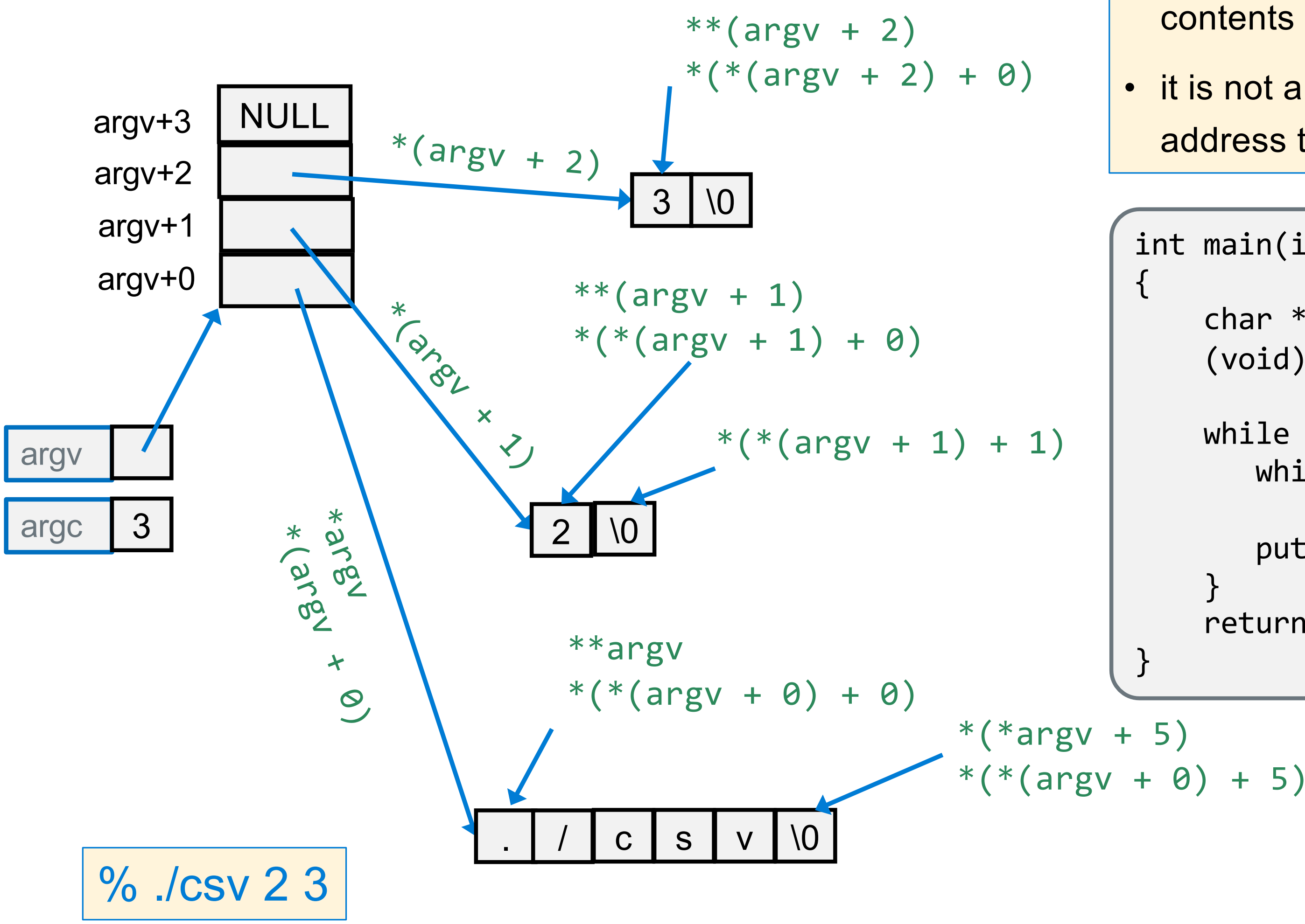
  `Conceptually: % *argv[0] *argv[1] *argv[2] ….`

- `argc` is the number of VALID elements (they point at something)
- `*argv (argv[0]) is` **usually** is the name of the executable file (% ./vim file.c)
- `argv[argc] or *(argv + argc)` always contains a NULL (0) sentinel
- `argv` elements point at **mutable** (fixed size) **strings**!



x

Accessing argv char at a time



- argv is a pointer variable, whose contents can be changed
- it is not an array name, which is just an address that cannot be changed

```
int main(int argc, char **argv)
{
    char *pt;
    (void)argc; // shut up the compiler

    while ((pt = *argv++) != NULL) {
        while (*pt != '\0')
            putchar(*pt++);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

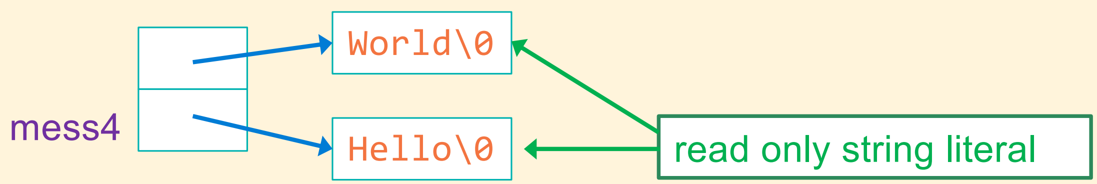

x

Defining an Array of Pointer to Strings

- mess4 is an array of pointers to immutable arrays

```
char *mess4[] = {"Hello","World"};  // immutable string
*(*mess4 + 1)  = '\0';              // bus error
```

Bus error: writing read only memory
Seg fault: writing unallocated memory



- mess5 is an array of pointers to mutable arrays

```
char *mess5[] = { (char []){"Hello"}, (char []){"World"}};
*(*mess5 + 1)  = '\0';                    // OK!
```