CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 5: Strings, C Memory Model, Endianness

Instructors: Dan Garcia, Justin Yokota

Agenda

- Strings
- C Memory model
- Heap usage
- C Demos
- Endianness

Agenda

- **Strings**
- C Memory model
- Heap usage
- C Demos
- Endianness

Strings

- Strings in C are defined as arrays of characters, similar to Python.
  - Note that there isn't a primitive type called "string" in C. Instead, C strings are given the type "char[]" or "char*".
- Recall: C arrays differ from arrays in other languages in a key way: the array itself does not explicitly keep track of its length.
  - To the computer, an array of length 50 looks indistinguishable from an array of length 5, unless you adopt some other convention.
- Most arrays end up storing their length in a variable or parameter. For strings, though, we follow a different convention:
- The character NUL (ASCII value 0, also sometimes written as '\0') is never considered a valid character in a string. Its meaning is instead "null terminator". In C, a string is defined to end at the first null terminator.

Strings

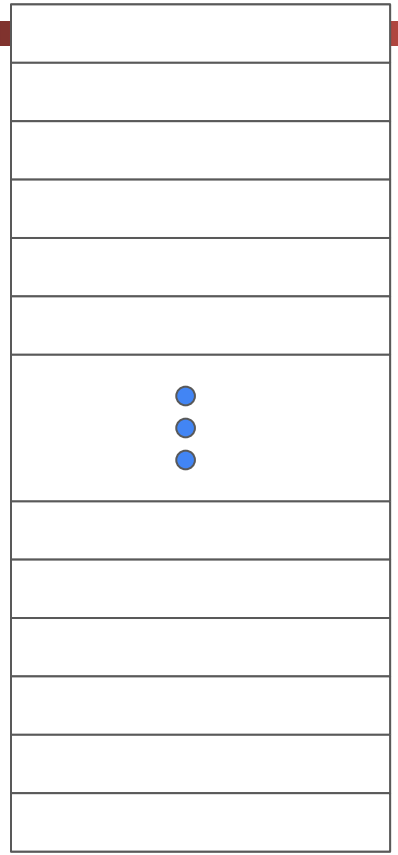
- The character NUL (ASCII value 0, also sometimes written as ‘\0’) is never considered a valid character in a string. Its meaning is instead “null terminator”. In C, a string is defined to end at the first null terminator.
- Example: The string “Hi” is stored in memory as an array of characters. This array would look like: `{‘H’, ‘i’, ‘\0’}`, or in number form, `{72, 105, 0}`
- As a result, strings need one more byte of memory than their length to be stored.
- Any string literals you use in your code will include a null terminator. However, if you create your own strings, you must remember to add a null terminator yourself. **If the null terminator is missing, C will continue to read memory as if it was part of the string until it comes across a byte that happens to have value 0.**
- This is a very common source of bugs.

Agenda

- Strings
- **C Memory model**
- Heap usage
- C Demos
- Endianness

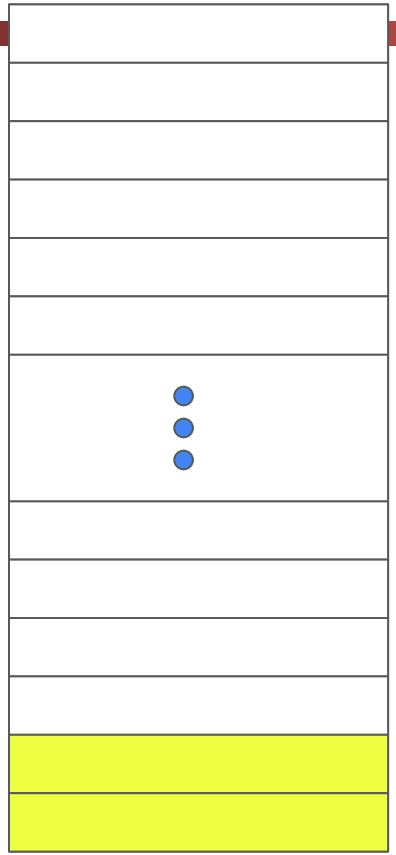
Memory Structure

- Main memory can be thought of as a large array of bytes.
- When running a C program, memory assignments largely fall under four distinct "chunks":
  - Stack
  - Heap
  - Static/data
  - Text/Code
- Generally speaking, you don't use all 2^{32} bytes; if you try to access a random memory address you didn't get assigned, your code will likely crash. This is known as a *segmentation fault*



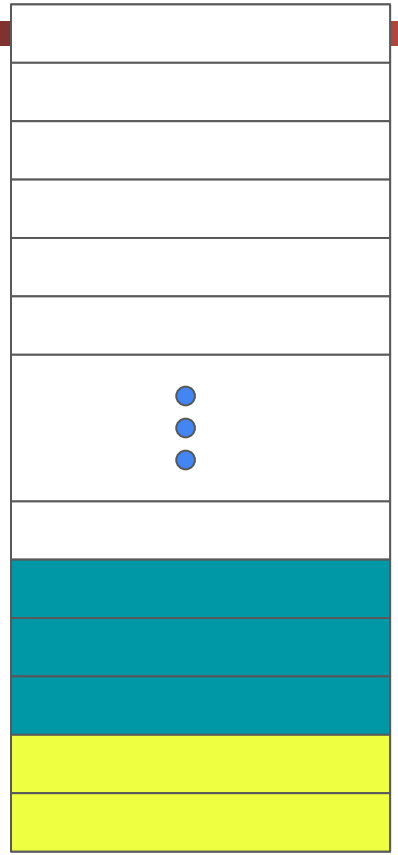
Memory Structure

- Code/text
  - One of the key advancements in CS was to design programs as software instead of hardware.
    - Instead of messing with wires to program a computer, you can just write your program as data
  - The code segment stores the actual bytecode that comprises your program.
  - Fixed size, ideally never changed after loading the program
  - Includes some constants!
    - Constants that are considered "built-in" to the code
    - Ex. x = y + 1; the 1 is part of the code.



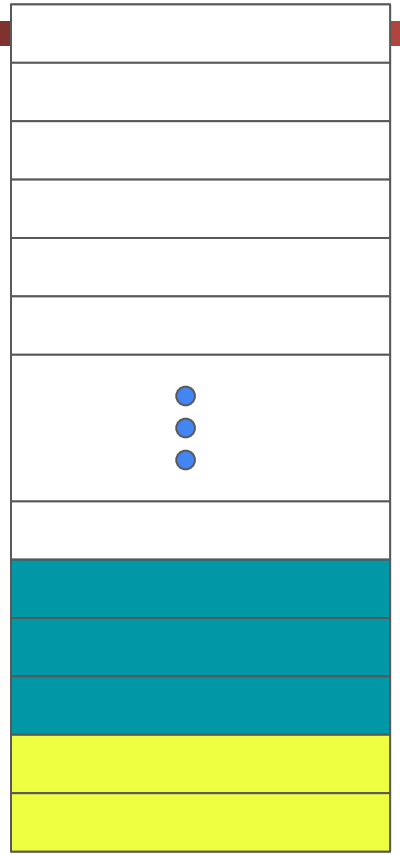
Memory Structure

- Static/data
  - Main idea: It's possible to do some analysis before the code starts running to determine a set of variables that must be allocated space. The data segment contains as many of those variables as possible.
  - Fixed size, for efficiency. Two main data values:
  - Global variables (you only ever need one copy of a global variable, so fixed size)
  - String literals (you can determine how many string literals there are in a program)



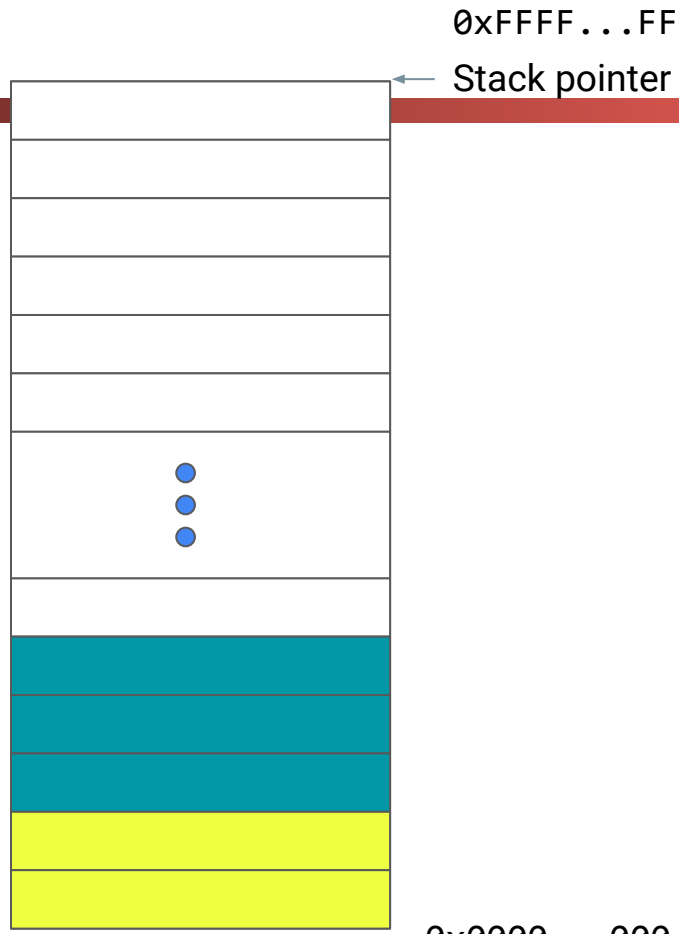
Memory Structure

- Static/data
  - String literals
  - Every unique string gets stored here (if you have multiple parts of your code that use the same string, they all end up pointing to the same memory address)
    - Corollary: String literals are **immutable** if they are stored in data
    - ```
      char* i = "Hi";
      i[0]="P"; // Error
      char* j = "Hi";
      ```
  - Only applies to string literals which need pointers!
  - Ex. `char[] i = "Hi";` doesn't need a pointer to "Hi", so it treats the string the same was as an integer literal.


      ```

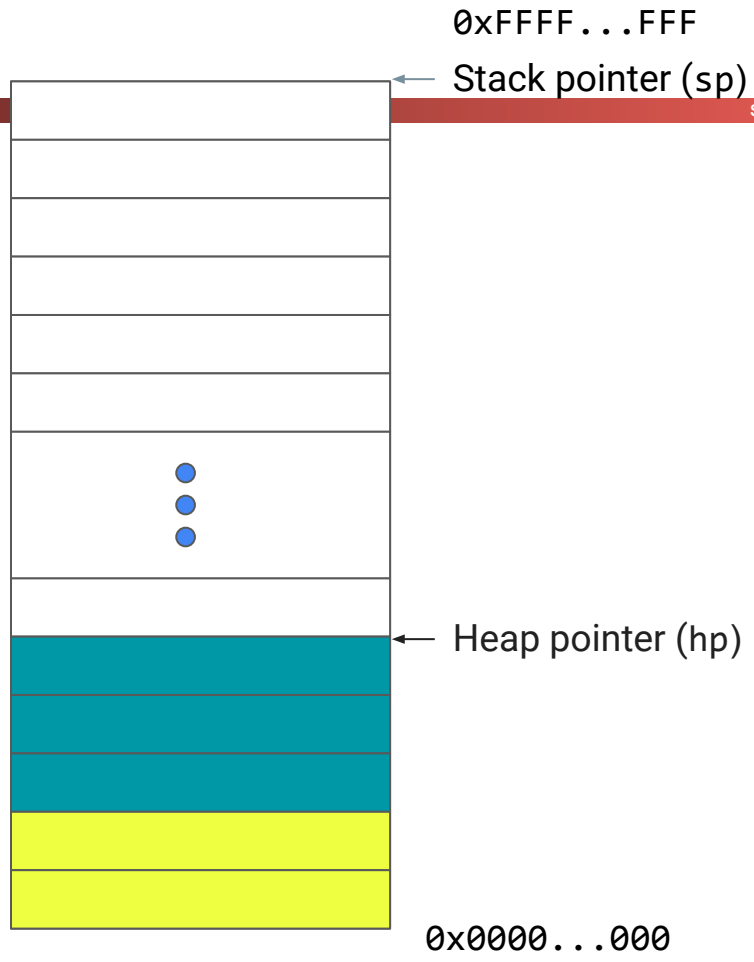
Memory Structure

- Stack
  - Acts similarly to the "Environment Diagrams" you might have seen in 61A
  - Every function call sets aside some space on the stack for its local variables (plus some metadata)
  - Designed for **temporary** storage; after a function returns, all data in that function's stack frame gets freed.
  - Variable size; grows downward as you call functions and shrinks as you return from functions.
  - All local variables are defined on the stack



Memory Structure

- Heap
  - For any variables that need to persist across functions, the heap is available
  - In C, all heap memory must be manually allocated; a C program won't allocate heap space unless you tell it to explicitly.
  - Similarly, heap memory is only freed when manually requested; if you forget to free something, it takes up heap space forever
  - Grows upward as memory is allocated
  - Often the source of a lot of bugs when starting off working with C.



Agenda

- Strings
- C Memory model
- **Heap usage**
- C Demos
- Endianness

Heap Usage

- The heap is run by a *memory manager*, which is part of the *operating system* (or OS)
- In order to use the heap, our program needs to request memory from the OS, and promises to return that memory when it's no longer needed.
- `void* malloc(size_t n)`
  - Requests n consecutive bytes from the memory manager. The memory manager finds a block of open space and reserves that space for your personal use. This is known as a *Memory ALLOCation.*
  - Does NOT clean out any data that used to be there (garbage data)
  - If the memory manager can't find enough memory, returns a NULL pointer
- `void free(void* ptr)`
  - Gives the block pointed to by ptr back to the memory manager
  - ptr MUST be a pointer that was returned by the memory manager earlier that hasn't been freed; if you free a pointer to the stack, a pointer in the middle of an allocated block, or a pointer that was already freed, your code crashes
  - Does NOT clean out any data you put in the allocated block

Heap Usage

- The heap is run by a *memory manager*, which is part of the *operating system* (or OS)
- In order to use the heap, our program needs to request memory from the OS, and promises to return that memory when it's no longer needed.
- `void* malloc(size_t n)`
  - Requests n consecutive bytes from the memory manager. The memory manager finds a block of open space and reserves that space for your personal use. This is known as a *Memory ALLOCation.*
  - Does NOT clean out any data that used to be there (garbage data)
  - If the memory manager can't find enough memory, returns a NULL pointer
- `void free(void* ptr)`
  - Gives the block pointed to by ptr back to the memory manager
  - ptr MUST be a pointer that was returned by the memory manager earlier that hasn't been freed; if you free a pointer to the stack, a pointer in the middle of an allocated block, or a pointer that was already freed, your code crashes
  - Does NOT clean out any data you put in the allocated block

Heap Usage

- `void* malloc(size_t n)`
- `void free(void* ptr)`
- `void* calloc(size_t nitems, size_t size)`
  - Allocates (nitems * size) bytes of data, and guarantees that the returned block is cleared (Clear ALLOCation)
  - Two main ways the memory manager does this:
    - Find a block that's already empty
    - Run malloc, then clear the block by zeroing out its contents
- `void* realloc(void* ptr, size_t n)`
  - "Resizes" the block pointed to by ptr, so that it now contains n bytes.
  - ptr must be a pointer that was returned by the memory manager
  - Most of the time, it just resizes the box and returns the same pointer back. However, if it can't do so (ex. if the data immediately after the malloc'ed block is already allocated), the memory manager can choose to move the block to a new place in memory, freeing the old pointer.
  - Otherwise, acts as malloc; newly allocated memory contains garbage, and realloc returns NULL if it can't find a new block.

Heap Example


```
int* i = malloc(sizeof(int) * 5);
for(int j = 0; j < 5; j++) i[j] = j;
int* k = calloc(3, sizeof(int));
i = realloc(i, sizeof(int)*2);
i = realloc(i, sizeof(int)*10);
free(k);
k = malloc(sizeof(int)*6);
for(int j = 0; j < 5; j++) k[j] = j;
free(i);
free(k+2);
```

Best Practices for Heap Use

- If you allocate something, make sure you free it eventually.
  - If you don't free a pointer before you lose track of the pointer, it'll stay allocated forever. This is known as a *memory leak*, and results in code that eventually breaks
  - Keep in mind that frees are NOT recursive; if you have a complicated struct, you need to free all components of the struct BEFORE you free the struct itself
  - Corollary: For large structs, a common practice is to make a *constructor* that initializes all needed memory for a struct, and a *destructor* that frees all needed memory for a struct.
- The built-in `sizeof` function can be used to determine the size of a struct (or primitive) in bytes.
  - Ex. `sizeof(char) == 1, sizeof(int) == 4` on most computers, but can differ
  - Use `sizeof` when calling `malloc` instead of doing the math manually, for portability
- Generally speaking, the stack is faster to use (no need to search for memory), but also can't store as much data. Also keep in mind that the stack frees all data after the function returns. Thus, efficient code uses the heap minimally, and uses as few distinct mallocs as possible.

Agenda

- Strings
- C Memory model
- Heap usage
- **C Demos**
- Endianness

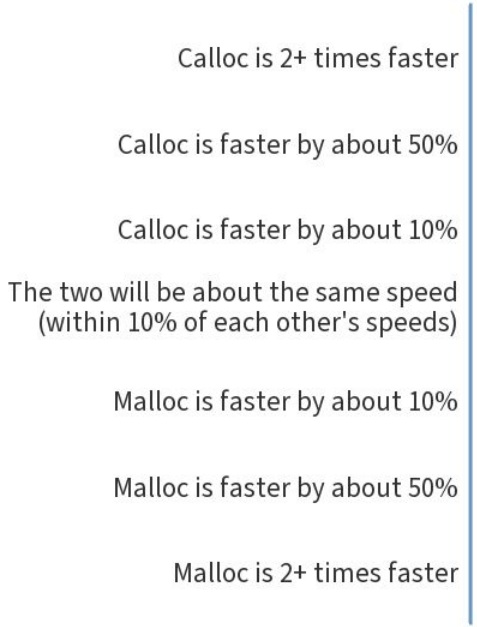
Code

- The code for these demos can be found here: https://drive.google.com/file/d/1TSjTF2uL5U4nDOgpRsaERLMljTUHz5GV/view?usp=sharing
- Contains the following demos:
  - A comparison of how long malloc and calloc take
  - A test to see how a memory leak works, and when things crash
  - A test on reading uninitialized memory locations

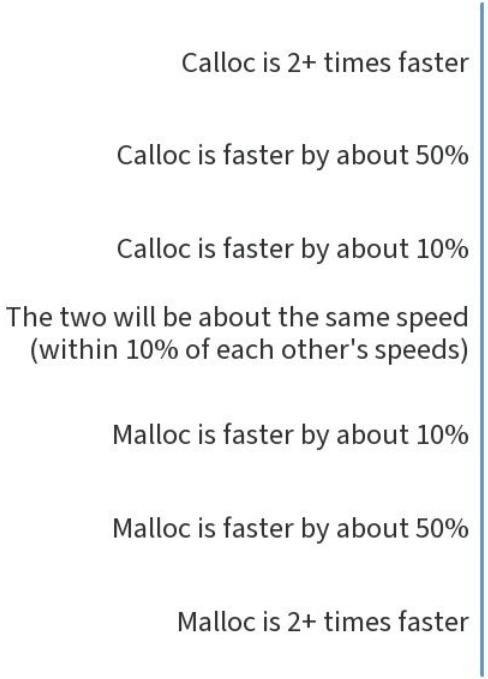
Which is faster: Malloc or Calloc?

- Calloc is 2+ times faster
- Calloc is faster by about 50%
- Calloc is faster by about 10%
- The two will be about the same speed (within 10% of each other's speeds)
- Malloc is faster by about 10%
- Malloc is faster by about 50%
- Malloc is 2+ times faster

Which is faster: Malloc or Calloc?



Which is faster: Malloc or Calloc?



What will happen when this code is run?

- The code will run correctly forever
- The code will run correctly for a long time, then suddenly crash
- A compiler error
- My computer will crash, and lecture ends early

What will happen when this code is run?

The code will run correctly forever

The code will run correctly for a long time, then suddenly crash

A compiler error

My computer will crash, and lecture ends early

What will happen when this code is run?

The code will run correctly forever

The code will run correctly for a long time, then suddenly crash

A compiler error

My computer will crash, and lecture ends early

What will happen when this code is run?

- A compiler error/warning
- The code will crash
- Almost all outputs are three characters long; "hi" followed by a random third character
- Almost all outputs are three characters or longer; "hi" followed by random nonsense characters
- Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters
- All outputs are "hi"

What will happen when this code is run?

A compiler error/warning

The code will crash

Almost all outputs are three characters long; "hi" followed by a random third character

Almost all outputs are three characters or longer; "hi" followed by random nonsense characters

Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters

All outputs are "hi"

What will happen when this code is run?

A compiler error/warning

The code will crash

Almost all outputs are three characters long; "hi" followed by a random third character

Almost all outputs are three characters or longer; "hi" followed by random nonsense characters

Almost all outputs are "hi", but some outputs are "hi" followed by random nonsense characters

All outputs are "hi"

Agenda

- Strings
- C Memory model
- Heap usage
- C Demos
- **Endianness**

Endianness

- So far, we've discussed how we store values in binary
  - Ex. We write `int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}.`
  - If we assume `&i == 0xF000 0000`, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			

- If we do `i[1]`, the compiler adds `0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`, then takes the four bytes starting from that address as an integer. This corresponds to 0x73206E65, so it works.
- What happens if we do `((char*) i)[2]`?
- Note: These slides follow a convention of using a new 0x prefix for every array element. Thus, `0x64726167 0x73206E65 0x7400646F` is an array of 32-bit values, while `0x64 0x72 0x61 0x67` is an array of 8-bit values.

Endianness

- So far, we've discussed how we store values in binary
  - Ex. We write `int[] i = {0x6472 6167, 0x7320 6E65, 0x7400 646F}`.
  - If we assume `&i == 0xF000 0000`, then our memory would look something like this:

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	?	?	?	?	?	?	?	?	?	?	?	?

- If we do **`((char*) i)[2]`**, the compiler adds 0xF000 0000 + 2 * sizeof(char) = 0xF000 0002, then takes the one byte starting from that address as a char. This yields something (since there is data there), but what it yields depends on how the subbytes of our 4-byte int get stored in memory. The way things get stored is known as the endianness of the system.

Analogy: How do we write dates?

- You want to communicate three dates to your friend:
  - October 7, 1999 (your birthday)
  - June 30, 2022 (today)
  - February 3, 2022 (When the US Formatting and Localization team will meet, per xkcd)
- Challenge: You can only write down nine numbers (no slashes or dashes).
- Option 1: Write the date in year-month-day:
  - 99 10 07 22 06 30 22 02 03
- Option 2: Write the date in day-month-year:
  - 07 10 99 30 06 22 03 02 22
- Option 3: Write the date in month-day-year:
  - 10 07 99 06 30 22 02 03 22
- Does it matter which one you use?
  - No, as long as your friend knows which one you're using

Analogy: How do we write dates?

- 99 10 07 22 06 30 22 02 03 (year-month-day) goes from largest-size to smallest-size. This is known as **big-endian** order.
- 07 10 99 30 06 22 03 02 22 (day-month-year) goes from smallest-size to largest-size. This is known as **little-endian** order.
- 10 07 99 06 30 22 02 03 22 (month-day-year) is neither big-endian nor little-endian. We generally call all such orderings **middle-endian**.
- In both versions, each group of three numbers is a date. As long as you read groups of 3 numbers together as a date (and read it the same way you wrote it), you'll extract the correct date.
- The only difference is if I ask something like "What's the third number written down"

Big-Endian

- In a big-endian system, we write the Most Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F

- If we do `((char*) i)[2]`, the compiler adds `0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`, then takes the one byte starting from that address as a char. This yields 0x61.
- If we do `printf((char*) i);` we would interpret this block of memory as a character array, so we’d get 0x64 0x72 0x61 …, which when converted to ASCII yields “drags net”

Little-Endian

- In a little-endian system, we write the Least Significant Byte “first” (that is, in the lower address.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- If we do `((char*) i)[2]`, the compiler adds `0xF000 0000 + 2 * sizeof(char) = 0xF000 0002`, then takes the one byte starting from that address as a char. This yields 0x72.
- If we do `printf((char*) i);` we would interpret this block of memory as a character array, so we’d get 0x67 0x61 0x72 …, which when converted to ASCII yields “garden sod”

Endianness

- Note: In either case, if we read `i[1]`, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Big Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0x73 0x20 0x6E 0x65. Since this is big-endian, we combine them with the first address being the MSB, so we get 0x73206E65, which is the correct result

Endianness

- Note: In either case, if we read `i[1]`, we end up with our original integer. This is a good checksum to confirm your understanding; if you write a number, you should be able to read that number to get the same thing.

Address (Last hex digit)	0	1	2	3	4	5	6	7	8	9	A	B
Value	0x64726167				0x73206E65				0x7400646F			
Value (Big Endian)	0x64	0x72	0x61	0x67	0x73	0x20	0x6E	0x65	0x74	0x00	0x64	0x6F
Value (Little Endian)	0x67	0x61	0x72	0x64	0x65	0x6E	0x20	0x73	0x6F	0x64	0x00	0x74

- In Little Endian: the compiler adds **`0xF000 0000 + 1 * sizeof(int) = 0xF000 0004`**, then takes the four bytes starting from that address: 0x65 0x6E 0x20 0x73. Since this is big-endian, we combine them with the first address being the LSB, so we get 0x73206E65, which is the correct result

Endianness

- All endiannesses work as long as you're consistent. It only affects cases where you either transfer to a new endianness (different systems can use different endiannesses), or when you split/merge a block of data into smaller/larger chunks.
  - This comes up a lot when working with unions, since a union is designed to interpret the same block of data in different ways.
- Officially, C defines this to be undefined behavior; you're not supposed to do this.
  - With unions, you're supposed to use that block as only one of the components
- In practice, undefined behavior is a great way to hack someone's program, so it ends up popping up in CS 161. It also ends up showing up when you do memory dumps (see homework).

Endianness

- Big Endian is commonly used in networks (e.g. communicating data between computers).
- Little Endian is commonly used within the computer
  - The default endianness for C, RISC-V, x86, etc. In general, you'll be working with little-endian systems when programming

Endianness

- Some systems are bi-endian (allows you to switch endianness if you want)
- Others use a completely different endianness (ex. PDP-11 does some weird order)
- There are mild benefits for either endianness (one might let you read important data faster than the other), but ultimately, we use different endiannesses because computer programmers are bad at coordinating standards, and once a standard is made, backwards compatibility is an issue.
- The **`hton`** and **`ntoh`** function sets convert from host endianness to network endianness and vice versa.