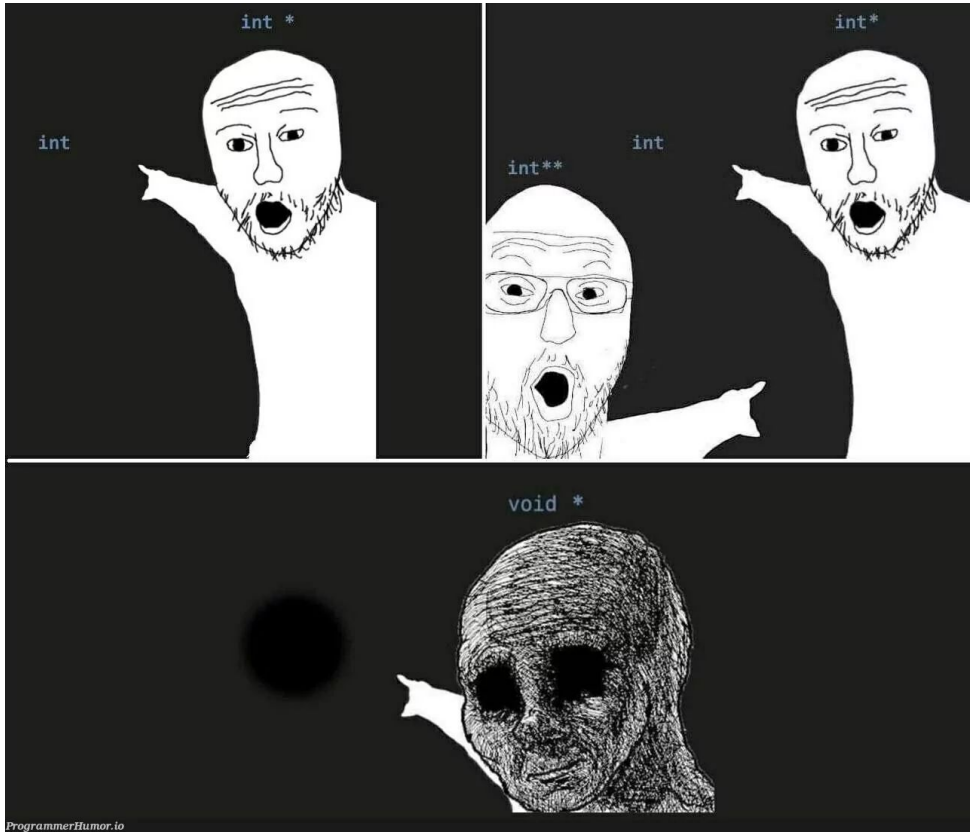
CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Last Time...



Announcements

- **TODAY**, 6/26: Lab 0
- **Wednesday,** 6/28: HW1
- **Thursday,** 6/29: Lab 1, Lab 2
- **Friday,** 6/30: Project 1
- **Rosalie's Affinity OH**
  - This is a space dedicated to those who don't come from traditionally computer science backgrounds, and needs a little more time & guidance on the material
  - Please don't abuse this space! We have normal staff office hours and regular instructor office hours
  - Also available by appointment, if you can't make the time
  - M/W 5-6PM
- **Student Incident Report Form**
  - If you have been made uncomfortable by a staff member or a fellow student, please feel free to report it here. It's also on the course website under https://cs61c.org/su23/policies/#extenuating-circumstances-and-inclusion

Today’s agenda

- Memory endianness
- Structs
- Strings
- More dynamic allocation things
    - Case study of C code!

Structs

Introduction to memory alignment

What we know about memory so far

- It's byte addressable
- Stores things according to memory addresses (any index < size of memory)

However, this can be quite inefficient! We want to optimise in two major ways

- Using as much of memory as possible effectively
- Allowing the processor to access data in memory quickly

By allowing the system to access memory at any memory address, like in arrays, we're not optimising for the processor or hardware.

- Memory alignment helps alleviate some of those issues!

What is memory alignment?

- Memory alignment: a system-dependent rule that tells us where data can be stored (usually not at every byte)
  - Doesn't this break the byte-addressable status of memory?
  - Nope! Byte addressable means memory at its smallest is formed of buckets of 8-bit quantities. It does not guarantee safe access to any memory address.
- Often times, alignment will not change across 32-bit vs 64-bit systems
  - In other words, addresses you're allowed to store data at remain the same, regardless of if your memory address is 4 bytes or 8 bytes
- To have a memory alignment to *n*-byte boundaries means:
  - Data will be stored starting at addresses divisible by *n*
- Can be accomplished with the "aligned" attribute `__attribute__((aligned(n)));`

Introduction to structs

- A "struct" is really just an instruction to C on how to arrange a bunch of bytes in a bucket
- ```
  struct foo {
      int a;          /* 4 bytes. */
      char b;         /* 1 byte. */
      struct foo *c;    /* 4 bytes. */
      char d[9];      /* 9 bytes. */
      char e;         /* 1 byte. */
  };
  ```
- How large is this struct?
  - 4 + 1 + 4 + 9 + 1 = 19 bytes? (Nope!)

System default often are aligned to 4 bytes!

NOTE: alignment does not affect struct fields, but rather where structs start and end. Field placement is affected by padding (default) and packing.
  ```

Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Point;


Point p1;
Point p2;
Point *paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

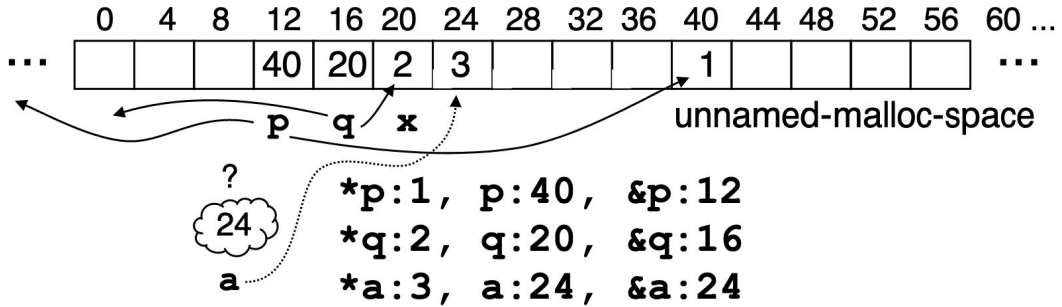
/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
```

Introduction to Unions

- A "union" is also instruction to C on how to arrange a bunch of bytes
- ```
  union foo {
      int a;
      char b;
      union foo *c;
  };
  ```
- Provides enough space for the largest element

```
union foo f;
f.a = 0xDEADB33F; /* treat f as an integer and store
                     that value */
f.c = &f; /* treat f as a pointer of type
             "union foo *" and store the
             address of f in itself */
```
```

Typedefs

- Notice for structs and unions, we have to use "`struct struct_type var_name`" or "`union union_type var_name`" to declare a variable
  - It's kind of clunky!
- Typedefs, or type definitions are ways of declaring and referencing variable types with a different name than what the explicit type is
  - `int`s are typedefed from other more explicit types
    - `typedef int32_t int;`
  - Generally they're used for externally facing types
- `typedef <original type> newtype;`
- ```
  typedef struct foo {
      // stuff
  } Foo;
  ```

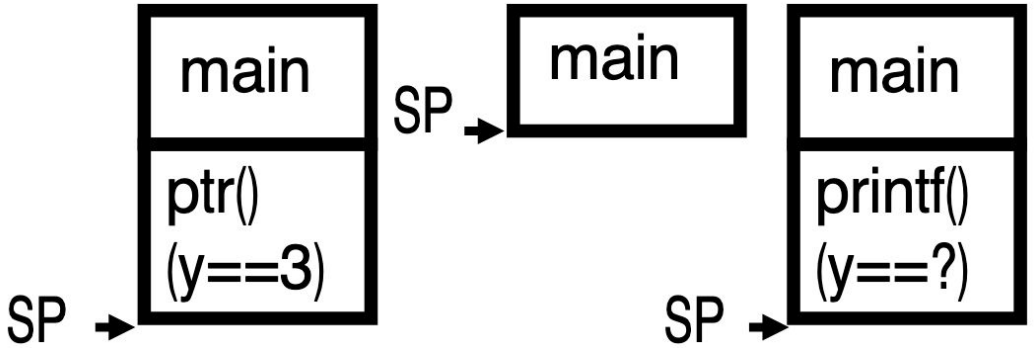
NOTE: generally typedef structs/unions start capitalised!

```
typedef union fooUnion {
    // stuff
} FooUnion;
```
```

Strings

But first... characters!

- Unlike with numbers, there's no "natural" way to assign bitstrings to characters
- This mapping is instead defined through the ASCII mapping
- Each character uses 8 bits of data (one byte)
- C does not distinguish between chars and 8-bit integers; arithmetic operations can be performed on chars, with C treating characters as their ASCII value.
    - Ex. The character '2' corresponds to ASCII value 50. If we did '2' + '2', C would interpret this as 50+50 = 100, which, if turned back into a char, would be the character 'd'. We can thus write '2'+'2'=='d'
- The next slide will have a shortened version of the ASCII table; this will be provided whenever ASCII translations are needed.

ASCII table

HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR
0x20	32	SPACE	0x30	48	0	0x40	64	@	0x50	80	P	0x60	96	`	0x70	112	p
0x21	33	!	0x31	49	1	0x41	65	A	0x51	81	Q	0x61	97	a	0x71	113	q
0x22	34	"	0x32	50	2	0x42	66	B	0x52	82	R	0x62	98	b	0x72	114	r
0x23	35	#	0x33	51	3	0x43	67	C	0x53	83	S	0x63	99	c	0x73	115	s
0x24	36	$	0x34	52	4	0x44	68	D	0x54	84	T	0x64	100	d	0x74	116	t
0x25	37	%	0x35	53	5	0x45	69	E	0x55	85	U	0x65	101	e	0x75	117	u
0x26	38	&	0x36	54	6	0x46	70	F	0x56	86	V	0x66	102	f	0x76	118	v
0x27	39	'	0x37	55	7	0x47	71	G	0x57	87	W	0x67	103	g	0x77	119	w
0x28	40	(	0x38	56	8	0x48	72	H	0x58	88	X	0x68	104	h	0x78	120	x
0x29	41	)	0x39	57	9	0x49	73	I	0x59	89	Y	0x69	105	i	0x79	121	y
0x2A	42	*	0x3A	58	:	0x4A	74	J	0x5A	90	Z	0x6A	106	j	0x7A	122	z
0x2B	43	+	0x3B	59	;	0x4B	75	K	0x5B	91	[	0x6B	107	k	0x7B	123	{
0x2C	44	,	0x3C	60	<	0x4C	76	L	0x5C	92	\	0x6C	108	l	0x7C	124	|
0x2D	45	-	0x3D	61	=	0x4D	77	M	0x5D	93	]	0x6D	109	m	0x7D	125	}
0x2E	46	.	0x3E	62	>	0x4E	78	N	0x5E	94	^	0x6E	110	n	0x7E	126	~
0x2F	47	/	0x3F	63	?	0x4F	79	O	0x5F	95	_	0x6F	111	o	0X00	0	NULL

Strings = character arrays

- 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*`".
- Remember that C arrays do not explicitly keep track of their own lengths.
  - 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'}`
- 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.

String Instantiation (1/3)

- There are a lot of ways in which strings can be created in C!
  - How it's declared and how it's initialised both affect where strings are stored in memory and what permissions we have with them
- String literals: a series of characters between double quotes! The NULL terminator is implicitly included when manipulating strings literals.
  - They're immutable!
  - Well, mostly anyways.
- `char* str_ptr = "string literal here";`
  - `str_ptr` is created on the stack (if local), static/data R(W) (if global)
  - `"string literal here"` is probably stored in read-only memory (likely static/data section)
    - Will vary based on system architecture and compiler complexity!
  - Here, we're creating a reference to a string literal, in other words pointing to where the immutable string literal is stored in memory
  - This initialisation of `str_ptr` is immutable

String Instantiation (2/3)

- `char str_arr[] = "string literal here";`
  - `str_arr` is created on the stack (if local), static/data RW (if global)
  - Arrays by default are mutable!
  - This string literal is still immutable but this is still valid:
    `str_ar[idx] = 'c';`
  - Pending compiler optimisation and system, this setup copies the string literal shown, plus a `'\0'` into a character array of length 20 (19 + 1) on the stack of RW static memory
    - Ex. if this code was compiled with gcc, the compiler will allocate 20 bytes of space on the stack for this string and set those 20 bytes equal to `"string literal here"`, without creating a corresponding static string.
- `char str_arr2[] = {'s', 't', 'r', 'i', 'n', 'g', '\0'};`
  - `str_arr2` is is on stack or static memory
  - The contents of the initialisation populates the array
- `char str_arr3[7];`
  - `str_arr3` is on stack or static memory
  - The contents are garbage because the string hasn't been initialised yet!

String Instantiation (3/3)

- `char* str_ptr2 = (char*) calloc(sizeof(char) * 7);`
  - `str_ptr2` is a pointer created on the static (if local) or in static/data memory (if global)
  - Once dynamically allocated, the value stored in `str_ptr2` is a heap memory address
  - We can do something like the following to populate it:

```
for (idx = 0; i < 6; i++) {
    *(str_ptr2 + idx) = 'a' + idx;
}
```

  - Printing `str_ptr2` will result in "abcdef"
  - Why do we use `calloc` instead of `malloc`?
    - We need a NULL terminator at the end of all strings, and `'\0' == 0b00000000`
    - `calloc` allows us to fill the entire chunk of memory `str_ptr2` is pointing to with 0s
    - Saves us time in explicitly setting `*(str_ptr + 6) = '\0'`
  - What does `strlen(str_ptr2)` return?
  - Why do we end at `strlen(str_ptr2)` instead of 7?

Dynamic Allocation in C

Dynamic Memory Allocation (1/4)

- C has operator **`sizeof()`** which gives size in bytes (of type or variable)
- Size of objects can be misleading and is bad style, so use **`sizeof(type)`**
  - Many years ago an **`int`** was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?
- "**`sizeof`**" knows the size of arrays:
  - **`int ar[3]; // Or:  int ar[] = {54, 47, 99}`**
    `sizeof(ar)` **`12`**
  - ...as well for arrays whose size is determined at run-time:
    `int n = 3;`
    `int ar[n]; // Or: int ar[fun_that_returns_3()];`
    `sizeof(ar)` **`12`**
- What's the size of a pointer? **`sizeof(int*)`**
  - It depends on the system!!! (Usually 4 or 8 bytes)

Dynamic Memory Allocation (2/4)

- To allocate room for something new to point to, use **`malloc()`** (with the help of a typecast and **`sizeof`**):

  `ptr = (int *) malloc (sizeof(int));`
  - Now, ptr points to a space somewhere in memory of size (**`sizeof(int)`**) in bytes.
  - **`(int *)`** simply tells the compiler what will go into that space (called a typecast).
- **`malloc`** is almost never used for 1 var
- **`ptr = (int *) malloc (n*sizeof(int));`**
  - This allocates an array of n integers.

Dynamic Memory Allocation (3/4)

- Once **`malloc()`** is called, the memory location contains garbage, so don’t use it until you’ve set its value.
- After dynamically allocating space, we must dynamically free it:
  - **`free(ptr);`**
- Use this command to clean up.
  - Even though the program frees all memory on exit (or when main returns), don’t be lazy!
  - You never know when your main will get transformed into a subroutine!

Dynamic Memory Allocation (4/4)

- The following two things will cause your program to crash or behave strangely later on, and cause *very very* hard to figure out bugs:
  - **`free()`**ing the same piece of memory twice
  - calling **`free()`** on something you didn't get back from **`malloc()`**
- The runtime does not check for these mistakes
  - Memory allocation is so performance-critical that there just isn't time to do this
  - The usual result is that you corrupt the memory allocator's internal structure
  - You won't find out until much later on, in a totally unrelated part of your code!

Managing the Heap: `realloc(p, size)`

- Resize a previously allocated block at **`p`** to a new size
- If **`p`** is NULL, then **`realloc`** behaves like **`malloc`**
- If size is 0, then **`realloc`** behaves like **`free`**, deallocating the block from the heap
- Returns new address of the memory block; note: it is likely to have moved!

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
... ... ...
ip = (int *) realloc(ip,20*sizeof(int));
/* always check NULL, contents of first 10 elements retained */
... ... ...
realloc(ip,0); /* identical to free(ip) */
```

Arrays not implemented as you'd think

```
void foo() {
    int *p, *q, x;
    int a[4];
    p = (int *) malloc (sizeof(int));
    q = &x;
    *p = 1; // p[0] would also work here
    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
    *q = 2; // q[0] would also work here
    printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
    *a = 3; // a[0] would also work here
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



K&R: "An array name is not a variable"

When Memory Goes Bad

Pointers in C

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier / faster / etc to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
  - Dangling reference (use ptr before malloc)
  - Memory leaks (tardy free, lose the ptr)

Writing off the end of arrays...

- ```
  int *foo = (int *) malloc(sizeof(int) * 100);
  int i;
  ....
  for(i = 0; i <= 100; ++i) {
     foo[i] = 0;
  }
  ```
- Corrupts other parts of the program...
  - Including internal C data
- May cause crashes later
  ```

Returning Pointers into the Stack

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```
int *ptr () {
     int y;
     y = 3;
     return &y;
};

main () {
     int *stackAddr, content;
     stackAddr = ptr();
     content = *stackAddr;
     printf("%d", content); /* 3 */
     content = *stackAddr;
     printf("%d", content); /*13451514
*/
};
```



Use After Free

- When you keep using a pointer..

```
struct foo *f
....
f = malloc(sizeof(struct foo));
....
free(f)
....
bar(f->a);
```

- Reads after the free may be corrupted
  - As something else takes over that memory. Your program will probably get wrong info!
- Writes corrupt other data!
  - Uh oh… Your program crashes later!

Forgetting realloc Can Move Data...

- When you realloc it can copy data...

```
struct foo *f = malloc(sizeof(struct foo) * 10);
....
struct foo *g = f;
....
f = realloc(sizeof(struct foo) * 20);
```

- Result is g may now point to invalid memory
  - So reads may be corrupted and writes may corrupt other pieces of memory

Freeing the Wrong Stuff...

- If you **`free()`** something never **`malloc`**'ed()
  - Including things like

```
struct foo *f = malloc(sizeof(struct foo) * 10)
...
f++;
...
free(f)
```

- **`malloc`** or **`free`** may get confused..
  - Corrupt its internal storage or erase other data...

Double-Free...

- E.g.,

```
struct foo *f = (struct foo *)
    malloc(sizeof(struct foo) * 10);
…
free(f);
…
free(f);
```

- May cause either a use after **`free`** (because something else called **`malloc()`** and got that data) or corrupt **`malloc`**'s data (because you are no longer freeing a pointer called by **`malloc`**)

```
double free or corruption (out)
```

Losing the initial pointer! (Memory Leak)

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  ... ... ...
   plk++;
}
```

This *may* be a memory leak
if we don't keep somewhere else
a copy of the original malloc'ed
pointer

Valgrind to the rescue...

- Valgrind slows down your program by an order of magnitude, but...
  - It adds a tons of checks designed to catch most (but not all) memory errors
    - Memory leaks
    - Misuse of free
    - Writing over the end of arrays
- Tools like Valgrind are absolutely essential for debugging C code

How does `int x = -83;` look like in memory?

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	0×64726167				0×73206E65				0×7400646F			

- 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 0×73206E65, 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	0×64726167				0×73206E65				0×7400646F			
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.

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 0×61.
- If we do **`printf((char*) i);`** we would interpret this block of memory as a character array, so we’d get 0×64 0×72 0×61 ..., 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	0×64726167				0×73206E65				0×7400646F			
Value	0×67	0×61	0×72	0×64	0×65	0×6E	0×20	0×73	0×6F	0×64	0×00	0×74

- 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 0×72.
- If we do **`printf((char*) i);`** we would interpret this block of memory as a character array, so we’d get 0×67 0×61 0×72 ..., which when converted to ASCII yields “garden sod”

Endianness: Summary

- 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: 0×73 0×20 0×6E 0×65. Since this is big-endian, we combine them with the first address being the MSB, so we get 0×73206E65, which is the correct result

Endianness: Summary

- 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: 0×65 0×6E 0×20 0×73. Since this is big-endian, we combine them with the first address being the LSB, so we get 0×73206E65, which is the correct result

Endianness: Summary

- 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: Summary

- 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

C Demo

Linked List Example

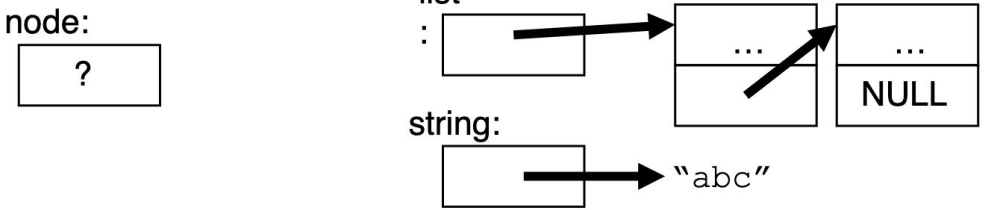
Linked List Example

- Let's look at an example of using structures, pointers, **`malloc()`**, and **`free()`** to implement a linked list of strings.

```
struct Node {
    char *value;
    struct Node *next;
};
typedef struct Node *List;

/* Create a new (empty) list */
List ListNew(void)
{ return NULL; }
```

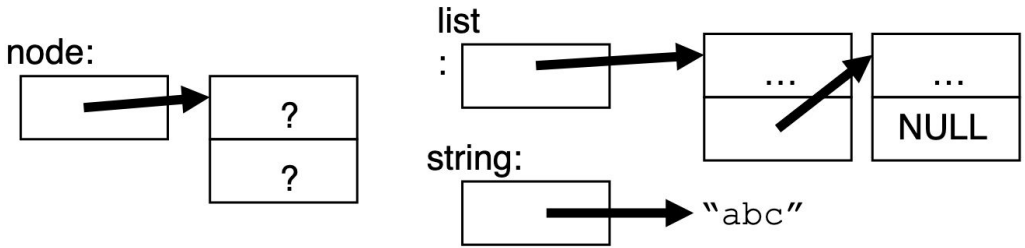
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



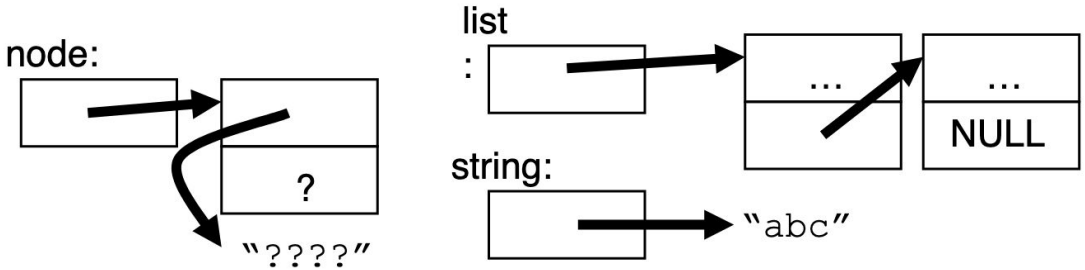
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



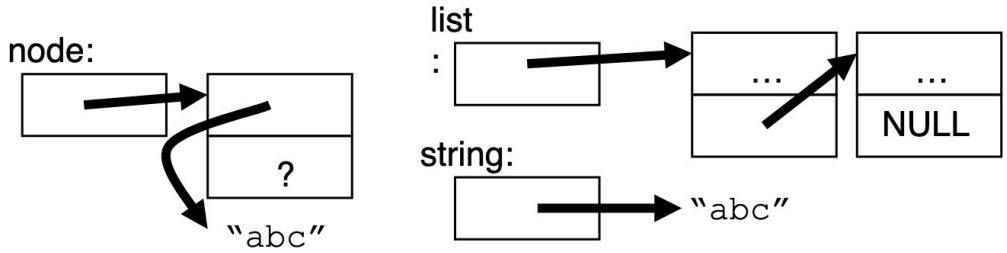
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



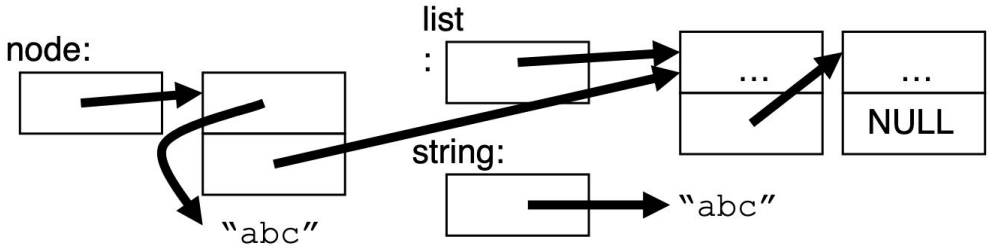
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



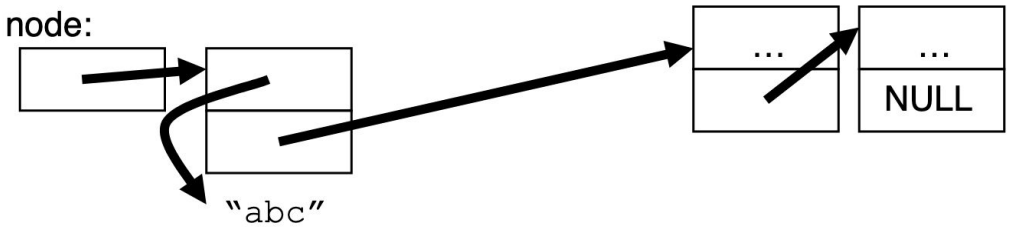
Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Linked List Example

```
/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```



Function Pointer Example

map (actually mutate_map easier)

```
#include <stdio.h>

int x10(int), x2(int);
void mutate_map(int [], int n, int(*)(int));
void print_array(int [], int n);

int x2 (int n) {   return 2*n;   }
int x10(int n) {   return 10*n;   }

void mutate_map(int A[], int n, int(*fp)(int)) {
     for (int i = 0; i < n; i++)
          A[i] = (*fp)(A[i]);
}

void print_array(int A[], int n) {
     for (int i = 0; i < n; i++)
          printf("%d ",A[i]);
     printf("\n");
}
```

```
% ./map
3 1 4
6 2 8
60 20
80
```

```
int main(void)
{
     int A[] = {3,1,4}, n = 3;
     print_array(A, n);
     mutate_map (A, n, &x2);
     print_array(A, n);
     mutate_map (A, n, &x10);
     print_array(A, n);
}
```