



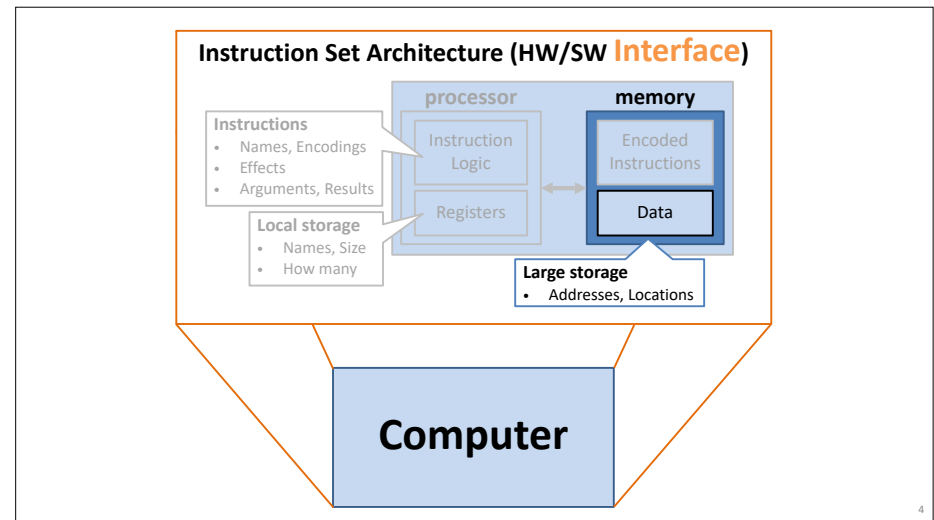
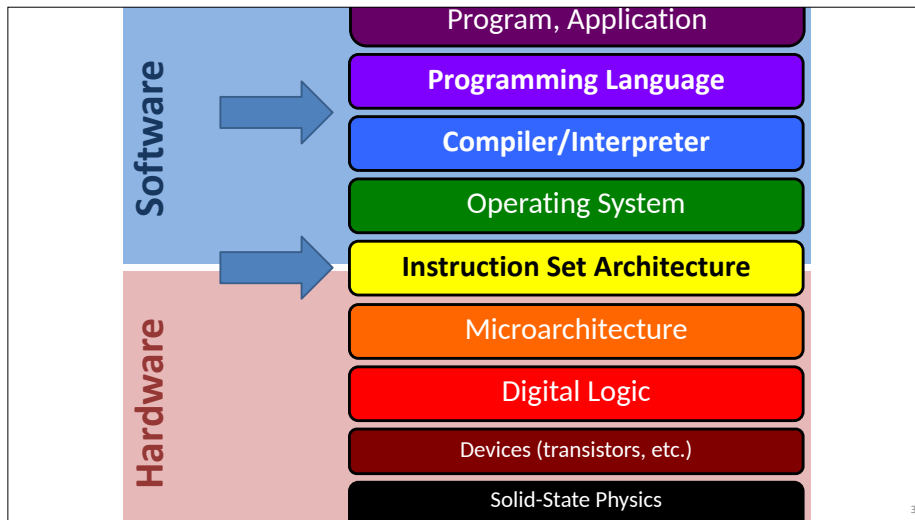
CS 240 Stage 2! Hardware-Software Interface

Memory addressing, C language, pointers
Assertions, debugging
Machine code, assembly language, program translation
Control flow
Procedures, stacks
Data layout, security, linking and loading



Programming with Memory

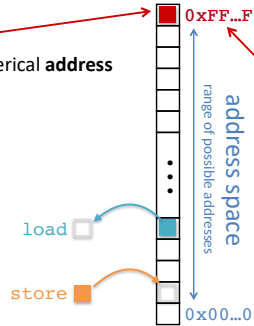
the memory model
pointers and arrays in C



Byte-addressable memory = mutable byte array

Location / cell = element

- Identified by unique numerical **address**
- Holds **one byte (8 bits)**



Address = index

- Unsigned number
- Represented by one word
- Computable and storable as a value

Operations:

- **Load:** read contents at given address
- **Store:** write contents at given address

5

Multi-byte values in memory

Store across contiguous byte locations.

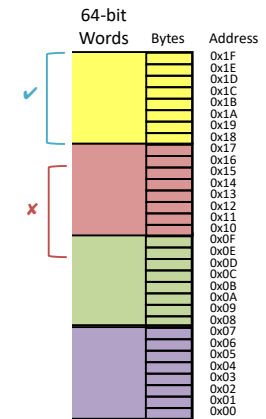
Example: 8 byte (64 bit) values

Alignment

Multi-byte values start at addresses that are multiples of their size

Bit order within byte always same.

Recall: byte ordering within larger value?



6

Is an `int` stored at address 0x00000002 aligned?

Yes

No

Maybe

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Is an `int` stored at address 0x00000002 aligned?

Yes

0%

No

0%

Maybe

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Is an `int` stored at address 0x00000002 aligned?

Yes 0%

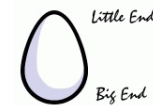
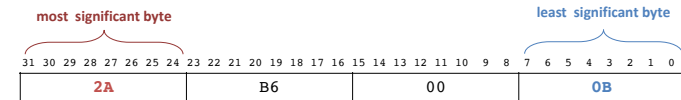
No 0%

Maybe 0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app

Endianness: details

In what order are the individual bytes of a multi-byte value stored in memory?



Address	Contents
03	2A
02	B6
01	00
00	0B

Little Endian: least significant byte first

- low order byte at low address
- high order byte at high address
- used by x86, ... and CS240!

Address	Contents
03	0B
02	00
01	B6
00	2A

Big Endian: most significant byte first

- high order byte at low address
- low order byte at high address
- used by networks, SPARC, ...

10

Data, addresses, and pointers

	0x24
	0x20
	0x1C
	0x18
	0x14
	0x10
	0x0C
	0x08
	0x04
	0x00

For these slides, we'll draw the bytes in this reverse order so that multi-byte values can be read directly

→ +3 +2 +1 +0
memory drawn as 32-bit values, little endian order

11

Data, addresses, and pointers

address = index of a location in memory

pointer = a reference to a location in memory, represented as an address stored as data

Let's store the number 240 at address 0x20.

$$240_{10} = F0_{16} = 0x00 \ 00 \ 00 \ F0$$

At address 0x08 we store a pointer to the contents at address 0x20.

At address 0x00, we store a pointer to a pointer.

The number 12 is stored at address 0x10.

Is it a pointer?

How do we know if values are pointers or not?

How do we manage use of memory?

	0x24
00 00 00 F0	0x20
	0x1C
	0x18
	0x14
00 00 00 0C	0x10
	0x0C
00 00 00 20	0x08
	0x04
00 00 00 08	0x00

→ +3 +2 +1 +0
memory drawn as 32-bit values, little endian order

12

C: Variables are locations

The compiler creates a map from variable name → location.

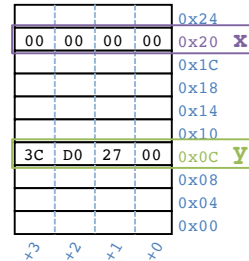
Declarations do not initialize!

```
int x; // x @ 0x20
int y; // y @ 0x0C

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;

// 1. load the contents @ 0x0C
// 2. add 3
// 3. store sum @ 0x20
x = y + 3;
```



13

C: Variables are locations

The compiler creates a map from variable name → location.

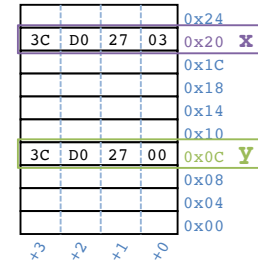
Declarations do not initialize!

```
int x; // x @ 0x20
int y; // y @ 0x0C

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;

// 1. load the contents @ 0x0C
// 2. add 3
// 3. store sum @ 0x20
x = y + 3;
```



14

C: Pointer operations and types

address = index of a location in memory

pointer = a reference to a location in memory, an address stored as data

Expressions using addresses and pointers:

&__ address of the memory location representing __
a.k.a. "reference to __"

*__ contents at the memory address given by __
a.k.a. "dereference __"

Pointer types:

__* address of a memory location holding a __
a.k.a. "a reference to a __"

15

C: Types determine sizes

Sizes of data types (in bytes)

Java Data Type	C Data Type	32-bit word	64-bit word
boolean	<i>bool</i>	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	(pointer) *	4	8

Used by CS Linux, most modern machines

address size = word size

16

C: Pointer example

& = address of
* = contents at

`int* p;`

Declare a variable, p

that will hold the address of a memory location holding an int

`int x = 5;`

`int y = 2;`

Declare two variables, x and y, that hold ints, and store 5 and 2 in them, respectively.

`p = &x;`

Take the address of the memory

representing x

... and store it in the memory location representing p. Now, "p points to x."

Add 1 to

the contents of memory at the address

`y = 1 + *p;`

given by the contents of the memory location representing p

... and store it in the memory location representing y.

17

C: Pointer example

& = address of
* = contents at

C assignment:

Left-hand-side = right-hand-side;

value

```
int* p; // p @ 0x04
int x = 5; // x @ 0x14, store 5 @ 0x14
int y = 2; // y @ 0x24, store 2 @ 0x24
p = &x; // store 0x14 @ 0x04
```

```
// 1. load the contents @ 0x04 (=0x14)
// 2. load the contents @ 0x14 (=0x5)
// 3. add 1
// 4. store sum as contents @ 0x24
y = 1 + *p;
```

```
// 1. load the contents @ 0x04 (=0x14)
// 2. store 0xF0 as contents @ 0x14
*p = 240;
```

00	00	00	0B	0x24	y
				0x20	
				0x1C	
				0x18	
00	00	00	05	0x14	x
				0x10	
				0x0C	
				0x08	
00	00	00	14	0x04	p
				0x00	
x-3	x-2	x-1	x+0		

18

What is the result of printing the decimal values of `a` and `b` at the end of this code?

```
int a = 1;
int b = 5;
int* p = &a;
*p = *p + 1;
a = a + 1;
```

```
p = &b;
*p = *p * 2;
```

2, 10

3, 5

3, 10

6, 5

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

What is the result of printing the decimal values of `a` and `b` at the end of this code?

```
int a = 1;
int b = 5;
int* p = &a;
*p = *p + 1;
a = a + 1;
```

```
p = &b;
*p = *p * 2;
```

2, 10

3, 5

3, 10

6, 5

None of the above

0%

0%

0%

0%

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

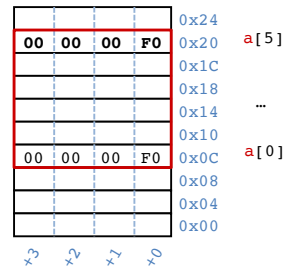
C: Arrays

Declaration: `int a[6];`
 Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



25

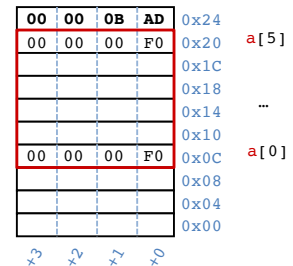
C: Arrays

Declaration: `int a[6];`
 Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`
 No bounds check: `a[6] = 0xBAD;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



26

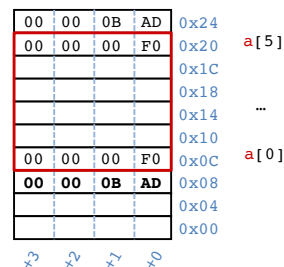
C: Arrays

Declaration: `int a[6];`
 Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`
 No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



27

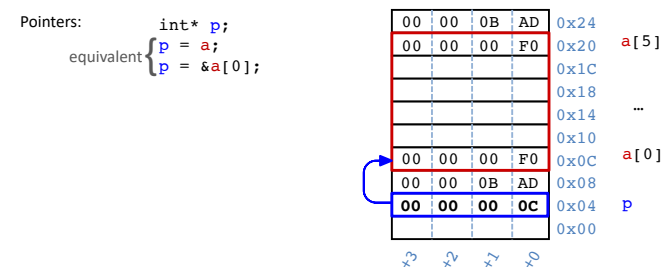
C: Arrays

Declaration: `int a[6];`
 Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`
 No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



28

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

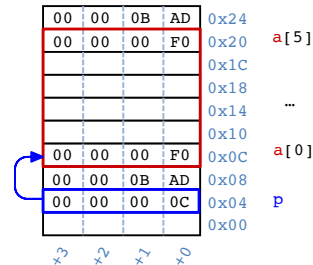
No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



29

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

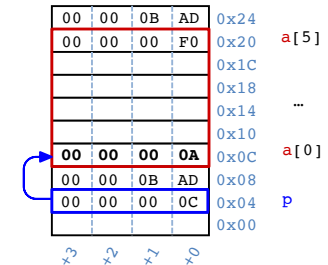
No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



30

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

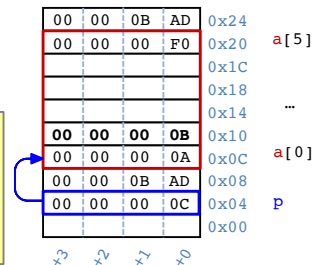
equivalent `{ p[1] = 0xB;`
`*(p + 1) = 0xB;`

array indexing = address arithmetic
 Both are scaled by the size of the type.

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



31

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

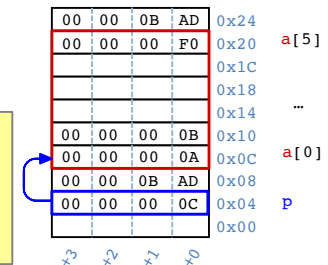
equivalent `{ p[1] = 0xB;`
`*(p + 1) = 0xB;`
`p = p + 2;`

array indexing = address arithmetic
 Both are scaled by the size of the type.

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



32

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

equivalent `{ p[1] = 0xB;`
`*(p + 1) = 0xB;`
`p = p + 2;`

array indexing = address arithmetic
 Both are scaled by the size of the type.

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

00	00	0B	AD	0x24	
00	00	00	F0	0x20	a[5]
				0x1C	
				0x18	
				0x14	...
00	00	00	0B	0x10	
00	00	00	0A	0x0C	a[0]
00	00	0B	AD	0x08	
00	00	00	14	0x04	p
				0x00	
x3	x2	x1	x0		

33

C: Arrays

Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`
`a[5] = a[0];`

No bounds check: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

Pointers: `int* p;`
 equivalent `{ p = a;`
`p = &a[0];`
`*p = 0xA;`

equivalent `{ p[1] = 0xB;`
`*(p + 1) = 0xB;`
`p = p + 2;`

array indexing = address arithmetic
 Both are scaled by the size of the type.

`*p = a[1] + 1;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

00	00	0B	AD	0x24	
00	00	00	F0	0x20	a[5]
				0x1C	
				0x18	
				0x14	...
00	00	00	0B	0x10	
00	00	00	0A	0x0C	a[0]
00	00	0B	AD	0x08	
00	00	00	14	0x04	p
				0x00	
x3	x2	x1	x0		

34

Assume `p` has type `int*`. Are `p[2] = 5` and `*(p + 2) = 5` equivalent? What about `p[2] = 5` and `*p + 2 = 5`?

No; No.

No; Yes.

Yes; No.

Yes; Yes.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Assume `p` has type `int*`. Are `p[2] = 5` and `*(p + 2) = 5` equivalent? What about `p[2] = 5` and `*p + 2 = 5`?

No; No.

0%

No; Yes.

0%

Yes; No.

0%

Yes; Yes.

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Assume `p` has type `int *`. Are `p[2] = 5` and `*(p + 2) = 5` equivalent? What about `p[2] = 5` and `*p + 2 = 5`?

No; No.

0%

No; Yes.

0%

Yes; No.

0%

Yes; Yes.

0%

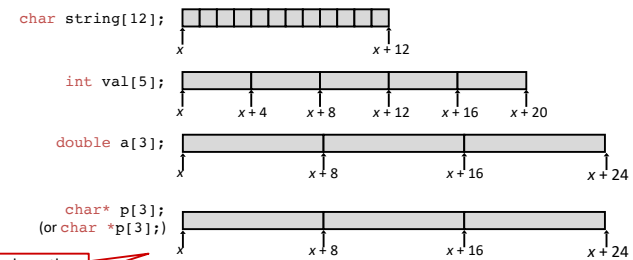
Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app

C: Array allocation

Basic Principle

`T A[N];`
Array of length N with elements of type T and name A
Contiguous block of $N * \text{sizeof}(T)$ bytes of memory

Use `sizeof` to determine proper size in C.



size depends on the machine word size

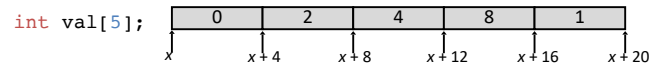
38

C: Array access

ex

Basic Principle

`T A[N];`
Array of length N with elements of type T and name A
Identifier A has type T^*



Expression	Type	Value
<code>val[4]</code>	<code>int</code>	1
<code>val</code>	<code>int *</code>	
<code>val+1</code>	<code>int *</code>	
<code>&val[2]</code>	<code>int *</code>	
<code>val[5]</code>	<code>int</code>	
<code>*(val+1)</code>	<code>int</code>	
<code>val + i</code>	<code>int *</code>	

39

Representing strings

A C-style string is represented by an array of bytes (`char`).

- Elements are one-byte **ASCII codes** for each character.
- ASCII = American Standard Code for Information Interchange

32	space	48	0	64	@	80	P	96	^	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

40

C: Null-terminated strings

ex

C strings: arrays of ASCII characters ending with *null character*.

Why?

0x57	0x65	0x6C	0x6C	0x65	0x73	0x6C	0x65	0x79	0x20	0x43	0x53	0x00
'W'	'e'	'l'	'l'	'e'	's'	'l'	'e'	'y'	' '	'C'	'S'	'\0'

Does Endianness matter for strings?

```
int string_length(char str[]) {  
  
  
  
  
  
  
}
```

41

How many bytes does it take to store the null-terminated string "hi bye"?

0

1

3

5

6

7

8

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

How many bytes does it take to store the null-terminated string "hi bye"?

0

1

0%

3

0%

5

0%

6

0%

7

0%

8

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

How many bytes does it take to store the null-terminated string "hi bye"?

0

1

0%

3

0%

5

0%

6

0%

7

0%

8

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

C: * and []

ex

C programmers often use ***** where you might expect **[]**:

e.g., **char***:

- pointer to a **char**
- pointer to the first **char** in a string of unknown length

```
int strcmp(char* a, char* b);
```

45

C: 0 vs. '\0' vs. NULL

0
Name: zero
Type: **int**
Size: 4 bytes
Value: 0x00000000
Usage: The integer zero.

'\0'
Name: null character
Type: **char**
Size: 1 byte
Value: 0x00
Usage: Terminator for C strings.

NULL
Name: null pointer / null reference / null address
Type: **void***
Size: 1 word (= 8 bytes on a 64-bit architecture)
Value: 0x0000000000000000
Usage: The absence of a pointer where one is expected.
Address 0 is inaccessible, so *NULL is invalid; it crashes.

Is it important/necessary to encode the null character or the null pointer as 0x0?

What happens if a programmer mixes up these "zeroey" values?

46

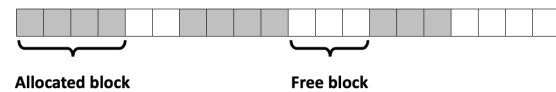
Memory address-space layout

Addr	Perm	Contents	Managed by	Initialized
2 ^N -1↑				
Stack	RW	Procedure context	Compiler	Run time
↑				
Heap	RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run time
Statics	RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
Literals	R	String literals	Compiler/ Assembler/Linker	Startup
Text	X	Instructions	Compiler/ Assembler/Linker	Startup
0				

47

C: Dynamic memory allocation in the heap

Heap:



Managed by memory allocator:

pointer to newly allocated block
of at least that size

number of contiguous bytes required

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

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

pointer to allocated block to free

48

C: standard memory allocator

```
#include <stdlib.h> // include C standard library
```

void* malloc(size_t size)
Allocates a memory block of at least size bytes and returns its address.
If memory error (e.g., allocator has no space left), returns NULL.

Rules:

- Check for error result.
- Cast result to relevant pointer type.
- Use sizeof(...) to determine size.

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

Deallocates the block referenced by ptr, making its space available for new allocations.

ptr must be a malloc result that has not yet been freed.

Rules:

- ptr must be a malloc result that has not yet been freed.
- Do not use *ptr after freeing.

49

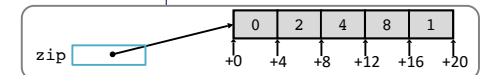
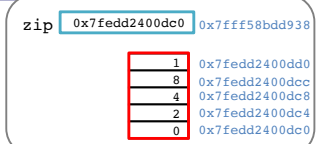
C: Dynamic array allocation

```
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) { // if error occurred
    perror("malloc"); // print error message
    exit(0); // end the program
}

zip[0] = 0;
zip[1] = 2;
zip[2] = 4;
zip[3] = 8;
zip[4] = 1;

printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
printf("\n");

free(zip);
```



50

C: Array of pointers to arrays of ints

```
int** zips = (int**)malloc(sizeof(int*) * 3);

zips[0] = (int*)malloc(sizeof(int)*5);
int* zip0 = zips[0];
zip0[0] = 0;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][4] = 1;

zips[1] = (int*)malloc(sizeof(int)*5);
zips[1][0] = 2;
zips[1][1] = 1;
zips[1][2] = 0;
zips[1][3] = 4;
zips[1][4] = 4;

zips[2] = NULL;
```

Why terminate with NULL?

Why no NULL?

zips

0x10004380	0x10008900	0x00000000		
0	2	4	8	1
2	1	0	4	4

51

Fill out the Pointers partner form (even if solo)



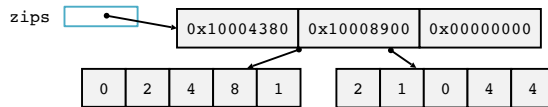
<https://forms.gle/kmL62aTcm9Nh3xjG9>

<http://xkcd.com/138/>



52

Zip code

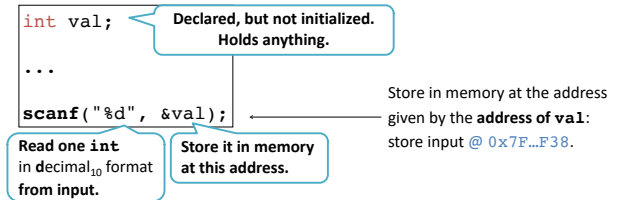


```
// return a count of all zips that end with digit endNum
int zipCount(int* zips[], int endNum) {
```

```
}
```

53

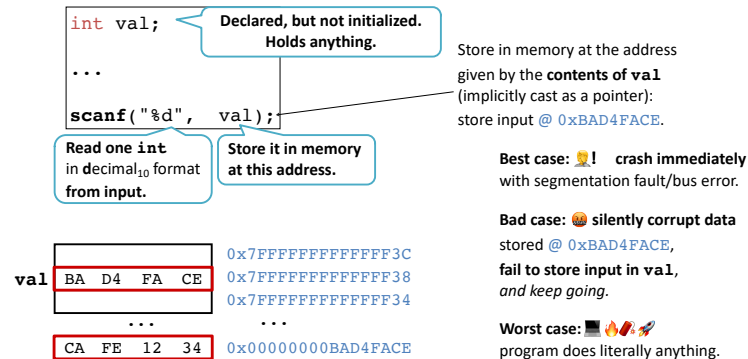
scanf reads formatted input



```
val CA FE 12 34 0x7FFFFFFF3C
0x7FFFFFFF38
0x7FFFFFFF34
```

54

C: Classic bug using scanf



55

C: Memory error messages

- 11: **segmentation fault** ("segfault", SIGSEGV)
accessing address outside legal area of memory
- 10: **bus error** (SIGBUS)
accessing misaligned or other problematic address

More to come on debugging!



<http://xkcd.com/371/>

56

C: Why?

Why learn C?

- Think like actual computer (abstraction close to machine level) without dealing with machine code.
- Understand just how much Your Favorite Language provides.
- Understand just how much Your Favorite Language might cost.
- Classic.
- Still (more) widely used (than it should be).
- Pitfalls still fuel devastating reliability and security failures today.

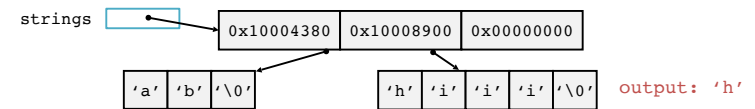
Why not use C?

- Probably not the right language for your next personal project.
- It "gets out of the programmer's way" ... even when the programmer is unwittingly running toward a cliff.
- Advances in programming language design since the 70's have produced languages that fix C's problems while keeping strengths.

57

Group example: longest string starts with

ex



```
// Return the starting character of the longest string in the
// null-terminated strings array.
// You can use: int strlen(char *str)
char longest_string_starts_with(char ** strings) {
```

58

Group example: longest string starts with

ex

```
// Return the starting character of the longest string in the
// null-terminated strings array.
// You can use: int strlen(char *str)
char longest_string_starts_with(char ** strings) {
    int longest = 0;
    char c = '\0';
    while (*strings) {
        int l = strlen(*strings);
        if (l > longest) {
            longest = l;
            c = *(*strings);
        }
        strings++;
    }
    return c;
}
```

59