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

pointers and arrays in C

https://cs.wellesley.edu/~cs240/s20/
Byte-addressable memory = mutable byte array

Location / cell = element
- Addressed by unique numerical address
- Holds one byte
- Readable and writable

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

Endianness: To store a multi-byte value in memory, which byte is stored first (at a lower address)?

Little Endian: least significant byte first
- low order byte at low address, high order byte at high address
  - used by x86, ...

Big Endian: most significant byte first
- high order byte at low address, low order byte at high address
  - used by networks, SPARC, ...

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

Multi-byte values in memory

Store across contiguous byte locations.

Alignment (Why?)

Bit order within byte always same.
Byte ordering within larger value?

Programming with Memory
C: Variables are locations

Compiler maps variable name → location.

Declarations do not initialize!

```c
int x; // x at 0x20
int y; // y at 0x0C
x = 0; // store 0 at 0x20
// store 0x3CD02700 at 0x0C
y = 0x3CD02700;
// load the contents at 0x0C,
// add 3, and store sum at 0x20
x = y + 3;
```

C: Pointer operations and types

*address* = index of a location in memory

*pointer* = a reference to a location in memory, represented as 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 ___"

C: Pointer example

```c
int* p;
int x = 5;
int y = 2;
p = &x;
y = 1 + *p;
```
C: Pointer example

**C assignment:**

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

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

// load the contents at 0x04  (0x14)
// load the contents at 0x14  (0x5)
// add 1 and store sum at 0x24
```
y = 1 + *p;
```

// load the contents at 0x04  (0x14)
// store 0xF0 (240) at 0x14
```
*p = 240;
```

**Arrays**

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

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.
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

---

C: Pointer type syntax

Spaces between base type, *, and variable name mostly do not matter.

The following are equivalent:

```c
int* ptr;
```

I see: "The variable ptr holds an address of an int in memory."

```c
int * ptr;
```

Looks like: "Dereferencing the variable ptr will yield an int."

Or "The memory location where the variable ptr points holds an int."

Caveat: do not declare multiple variables unless using the last form.

```c
int a, b; means int *a, b; means int* a; int b;
```

---

C: Pointer example

**C assignment:**

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

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

// load the contents at 0x04  (0x14)
// load the contents at 0x14  (0x5)
// add 1 and store sum at 0x24
```
y = 1 + *p;
```

// load the contents at 0x04  (0x14)
// store 0xF0 (240) at 0x14
```
*p = 240;
```

**Arrays**

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

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.
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
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.

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
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.
```

---

C: Pointer type syntax

Spaces between base type, *, and variable name mostly do not matter.

The following are equivalent:

```c
int* ptr;
```

I see: "The variable ptr holds an address of an int in memory."

```c
int * ptr;
```

Looks like: "Dereferencing the variable ptr will yield an int."

Or "The memory location where the variable ptr points holds an int."

Caveat: do not declare multiple variables unless using the last form.

```c
int a, b; means int *a, b; means int* a; int b;
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
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.
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
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.
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
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.
```

---

C: Arrays

Declaration:  

```c
int a[6];
```

- **element type**
- **name**
- **number of elements**

```
Arrays are adjacent memory locations storing the same type of data.
```
C: Array allocation
Basic Principle

T A[N];
Array of length N with elements of type T and name A

Contiguous block of N*sizeof(T) bytes of memory

char string[12];
int val[5];
double a[3];
char* p[3];
(or char *p[3];)

Use sizeof to determine proper size in C.

C: Array access
Basic Principle

T A[N];
Array of length N with elements of type T and name A

Identifier A has type

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

C: Null-terminated strings
C strings: arrays of ASCII characters ending with null character.

int string_length(char str[])
{
}

Does Endianness matter for strings?

C: * and []
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);
int string_length(char* str) {
  // Try with pointer arithmetic, but no array indexing.
}
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: 0x00000000000000
- 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?

C: Dynamic array allocation

```c
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) {
    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);
```
C: Array of pointers to arrays of ints

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

C: scanf reads formatted input

```c
int val;
...
scanf("%d", &val);
```

i.e., store it in memory at the address given by the contents of val:
store into memory at 0xBAD4FACE.
C: Classic bug using `scanf`

```
int val;
...
scanf("%d", &val);
```

i.e., store it in memory at the address given by the contents of `val`:
store into memory at 0xBAD4FACE.

- **Best case**: segmentation fault, or bus error, crash.
- **Bad case**: silently corrupt data stored at address 0xBAD4FACE, and `val` still holds 0xBAD4FACE.
- **Worst case**: arbitrary corruption

---

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.
- Many advances in programming language design since then have produced languages that fix C's problems while keeping strengths.

---

C: Memory error messages

```
OKAY, HUMAN.
HUH?
BEFORE YOU HIT COMPILERS,
LISTEN UP:

YOU KNOW WHEN YOU'RE FALLING ASLEEP AND
YOU IMAGINE YOURSELF WALKING OR
SOMETHING.

AND SUDDENLY YOU MISSTEP, STUMBLE,
AND JUST WAKE?

YEAH!
DOUBLE-CHECK YOUR DATA POINTERS, OKAY?
```

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

More to come on debugging!

http://xkcd.com/371/