Memory and Pointers in C
Why memory and pointers?

Interface

- Linux
- macOS
- Windows

CPU

Memory
Recall RISC-V from CS3410

Interface

• load / store instructions
• instruction / stack pointer registers
Physical memory

CPU under the cooling fan

Memory
Physical memory

* https://www.youtube.com/watch?v=Mjb12GCKycw

16 GB of DDR4 SDRAM

Intel i7 CPU
Physical memory

256MB of DDR3L

* https://digilent.com/shop/arty-a7-artix-7-fpga-development-board/
From physics to abstraction

• An ECE course would study voltage, current, etc.

• A CS course studies the abstraction of memory.
  • i.e., a simple math model, such as

<table>
<thead>
<tr>
<th>Content</th>
<th>1st byte</th>
<th>2nd byte</th>
<th>3rd byte</th>
<th>......</th>
<th>2^32th byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>0x0000 0000</td>
<td>0x0000 0001</td>
<td>0x0000 0002</td>
<td>......</td>
<td>0xFFFF FFFF</td>
</tr>
</tbody>
</table>
From abstraction to C variable

- A variable in C language is a range of memory.

- For example,

```c
int val = 0x19950128;
// say a RISC-V C compiler puts val at 0x60000
//     li t0, 0x60000
//     li t1, 0x19950128
//     sw t1, 0(t0)
```

<table>
<thead>
<tr>
<th>Address</th>
<th>0x0006 0000</th>
<th>0x0006 0001</th>
<th>0x0006 0002</th>
<th>0x0006 0003</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>0x 28</td>
<td>0x 01</td>
<td>0x 95</td>
<td>0x 19</td>
<td>......</td>
</tr>
</tbody>
</table>
```plaintext
int val = 0x19950128;
int* val_ptr = &val;
// say the compiler puts val_ptr at 0xC0000

<table>
<thead>
<tr>
<th>val_ptr</th>
<th>Content</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x 00</td>
<td>0x0000C 0000</td>
</tr>
<tr>
<td></td>
<td>0x 00</td>
<td>0x0000C 0001</td>
</tr>
<tr>
<td></td>
<td>0x 06</td>
<td>0x0000C 0002</td>
</tr>
<tr>
<td></td>
<td>0x 00</td>
<td>0x0000C 0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val</th>
<th>Content</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x 28</td>
<td>0x0006 0000</td>
</tr>
<tr>
<td></td>
<td>0x 01</td>
<td>0x0006 0001</td>
</tr>
<tr>
<td></td>
<td>0x 95</td>
<td>0x0006 0002</td>
</tr>
<tr>
<td></td>
<td>0x 19</td>
<td>0x0006 0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

From C variable to address
From address back to C variable

```c
int val = 0x19950128;
int* val_ptr = &val;
*val_ptr = 0x1234ABCD;
```

### Table: val_ptr and val

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0006 0000</td>
<td>0x 00</td>
<td>0x000C 0000</td>
<td>0x 00</td>
<td>0x000C 0001</td>
<td>0x 01</td>
</tr>
<tr>
<td>0x0006 0001</td>
<td>0x 06</td>
<td>0x000C 0002</td>
<td>0x 95</td>
<td>0x0006 0000</td>
<td>0x 28</td>
</tr>
<tr>
<td>0x000C 0002</td>
<td>0x 00</td>
<td>0x000C 0003</td>
<td>0x 19</td>
<td>0x0006 0001</td>
<td>0x 01</td>
</tr>
</tbody>
</table>

Step 1: read val_ptr as an address
Dereference a pointer: 2 steps

int val = 0x19950128;
int* val_ptr = &val;
*val_ptr = 0x1234ABCD;

Step 1: read val_ptr as an address
Step 2: write a value to that address
Program = variables + code

- Variables can be in
  - the read-only data section
  - the data section
  - the stack section
  - the heap section

- Machine code is in the code section
Quiz: variables in sections

int str_len = 14;

int main() {
    char* str = malloc(str_len);
    memcpy(str, "Hello World!\n", str_len);
    printf("%s", str);
    return 0;
}
int str_len = 14;

int main() {
    char* str = malloc(str_len);
    memcpy(str, "Hello World!\n", str_len);
    printf("%s", str);
    return 0;
}
Take-away #1: For each program, only code and stack are mandatory. Heap, Data, Read-only data are optional.
OS is a program
Zoom is another program

OS code & stack

Stack

Code

Zoom code & stack

Code

Stack
Add CPU into the picture

OS code & stack
- Stack
- Code

CPU
- Stack pointer register
- Instruction pointer register

Zoom code & stack
- Code
- Stack
CPU in the context of OS

OS code & stack

Stack

Code

CPU

Stack pointer register

Instruction pointer register

Zoom code & stack

Code

Stack
CPU in the context of Zoom

OS code & stack
- Stack
- Code

CPU
- Stack pointer register
- Instruction pointer register

Zoom code & stack
- Code
- Stack
Take-away #2

CPU context = memory abstraction + stack pointer + instruction pointer

Different CPUs can be in the context of different programs.
A brief summary

- Physical memory is a piece of hardware.
- Memory abstraction is a simple math model.
- A C variable is a range in the memory abstraction.
- C variables can be in stack/heap/data/rodata sections.
- Program = variables + code; And code + stack is a must.
- CPU context = memory abstraction + stack / instruction pointers
A variable is a few bytes in memory.

```c
int val = 0x19950128;
```

A variable is a few bytes in memory.
## Types tell the **size** of variables

<table>
<thead>
<tr>
<th>Type</th>
<th>sizeof(Type) (32bit CPU)</th>
<th>Type</th>
<th>sizeof(Type) (32bit CPU)</th>
<th>sizeof(Type) (64bit CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>char*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>int*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>long long*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>float*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>void</td>
<td>N/A</td>
<td>void*</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
char c = 'a';               // type is char; sizeof(c) == 1
char* c_ptr = &c;           // type is char*
// sizeof(c_ptr) == 4 or 8
*c_ptr = 'd';               // 'd' has type char
// *c_ptr has type char
assert(c == *c_ptr && c == 'd');

char** c_ptr_ptr = &c_ptr;  // type of &c_ptr is char**
// sizeof(c_ptr_ptr) == 4 or 8
assert(**c_ptr_ptr == 'd');

char another_c = 'g';
*c_ptr_ptr = &another_c;    // same as c_ptr == &another_c
// &another_c has type char*
// *c_ptr_ptr has type char*
// **c_ptr_ptr has type char
assert(**c_ptr_ptr == 'g');

Why pointer of pointer?

```c
void g() {
    int a; // a is 4 bytes on the stack of g()
    // and a is not initialized

    f(&a); // pass the address of a to f()
    assert(a == 123);
}

void f(int* ptr) {
    *ptr = 123; // modify 4 bytes on the stack of g()
}
```
void test() {
    void* item;  // item is 4 or 8 bytes on the
    // stack of test()

    queue_dequeue(..., &item);
}

int queue_dequeue(queue_t, queue, void** pitem) {
    *pitem = ...;  // modify 4 or 8 bytes on the
    // stack of test()
}

Why void*?

• A generic queue should store any type of variable.

• In C, any pointer type can cast to a void*.

• For example,

```c
int a = 0x19950128;        // a 4-byte variable
void* ptr = (void*) &a;    // cast int* to void*
char* c_ptr = (char*) ptr; // cast void* to char*
assert(*c_ptr == 0x28);    // dereferencing c_ptr will
// read 1 byte instead of 4
```
Some other C features
typedef unsigned int uint32_t;
uint32_t val;
// same as ‘unsigned int val’, but uint32_t is cleaner

// Similarly
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
// Data have **structures**
// For example,
// an IPv4 network packet header:

```c
struct header {
    uint8_t version:4;
    uint8_t ihl:4;
    uint8_t tos;
    uint16_t len;
    uint16_t id;
    uint16_t flags:3;
    uint16_t frag_offset:13;
    uint8_t ttl;
    uint8_t proto;
    uint16_t csum;
    uint32_t saddr;
    uint32_t daddr;
};
```
// Consider a function receiving network packets
// The payload size is unknown

char* recv_packet() {
    struct header header;
    net_recv_header(&header);
    char* payload = malloc(header.payload_size);
    net_recv_payload(payload);

    return payload;
}
Why `free()`?

```c
void process_packet () {
    char* payload = recv_packet();

    ..... // process the payload

    free(payload);

    // Otherwise, there is a memory leak which
    // can waste tons of memory in long-running programs
}
```
Summary

- Memory abstraction is a simple math model.
- **C variables** represent regions in the memory abstraction.
- **CPU context** = memory abstraction + stack / instruction pointers.
- **Types** specify the variables’ size in order to reduce bugs in programs.
- Reference of a variable, `&var`, returns an address as a pointer variable.
- Dereference of a pointer variable, `*ptr`, returns the variable at address ptr.
- Other useful C language features: `typedef`, `struct`, `malloc()` and `free()`