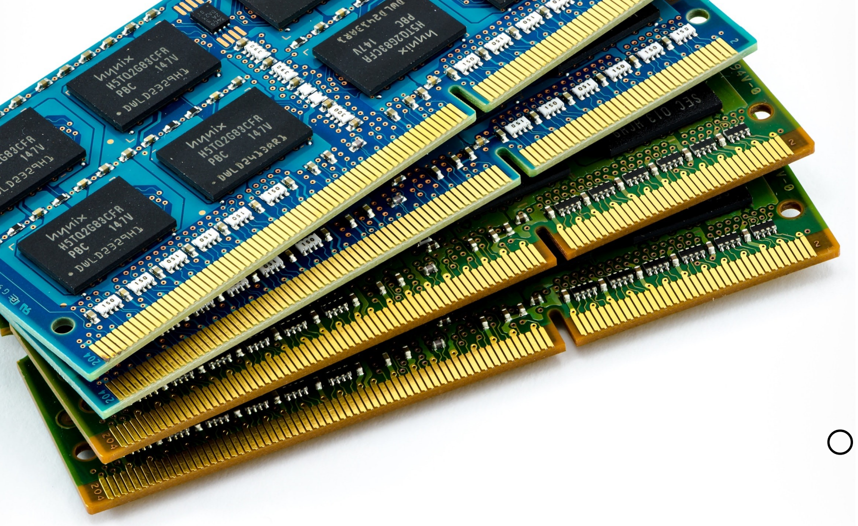
COMPSCI2030 Systems Programming

Pointers

Yehia Elkhatib

University of Glasgow



Memory

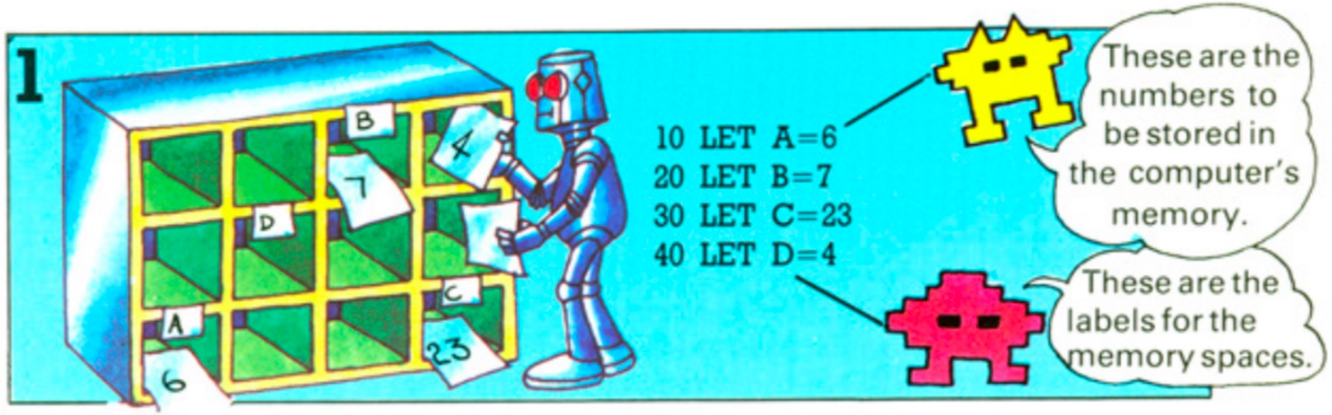
- Definition from the Cambridge Dictionary
  1. the ability to remember information, experiences, and people
  2. something that you remember from the past
  3. ***the part of a computer in which information or programs are stored either permanently or temporarily, or the amount of space available on it for storing information***

What exactly is it?
How do I use it?

How should we think about memory?

- We can think of memory as a sorting cabinet where each box stores the value of a variable
- The variable name is the <u>label</u> which allows us to remember where we stored what

Introduction to Computer Programing
(Brian Reffin Smith, (Usborne, 1982))



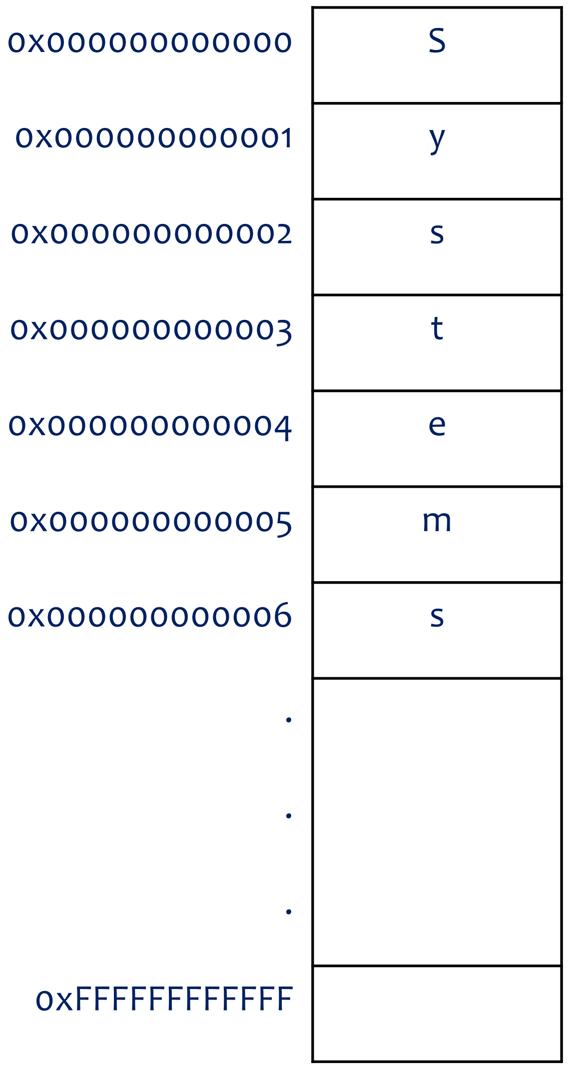
When you put a piece of data into the computer's memory you have to give it a label so you can find it again. You can use letters of the alphabet as labels. To label a memory space and put a number in it you can use the word LET, as shown above. A labelled memory space is called a variable because it can hold different data at different times in the program.

How should we think about memory?

- We can also think of memory as a single long street where each house has a unique address
- We have some notion of spatial locality
  - houses close to each other are neighbours; others are far away

Byte Addressable Memory

- Every *byte* in memory has a unique address
- On a 64-bit architecture, addresses are 64-bits (or 8 bytes) long
  - In theory, a 64-bit architecture can address up to 2^{64} bytes = 16 exabytes
  - In practice, x86-64 only uses the lower 48 bits of an address, supporting up to 2^{48} bytes = 256 TB
- An address is made up of 12 hexadecimal numbers ~ 48 bits



To manipulate values in memory

- We need 3 things:
  - get the memory address of a variable (i.e. *pointer*)
  - pass pointers (e.g. to functions) for manipulation
  - set a value at a pointer

```c
#include <stdio.h>

void set_to_zero(int x) {
    x = 0;
}

int main() {
    int y = 42;
    set_to_zero(y);
    printf("%d\n", y);
}
```

Variables in memory

- As we learned: every variable in C is stored at a memory location that does not change over its lifetime
- This location is identifiable by its address
- Depending on the size of the data type, the value of the variables will span multiple bytes in memory
- We can ask for the address of a variable in C using the address-of operator &

```c
int main() {
  int x = 42;
  int y = 23;
  printf("&x = %p\n", &x); // print the address of x
  printf("&y = %p\n", &y); // print the address of y
}
```

Pointers

- We store the address of a variable as the value of *another variable* that we call a pointer

```c
int x = 42;
int * pointer_to_x = &x; // this is a pointer referring to x
printf("value of pointer_to_x: %p\n", pointer_to_x); // prints 0x77...
```

- The *dereference operator* * allows us to access the value of the variable we are pointing to:

```c
printf("value of x: %d\n", *pointer_to_x); // prints 42
```

- A pointer to a variable of data type `t` has the data type `t *`
- Every pointer has the same size: the size of an address
  - on a 64-bit architecture, addresses are 8 bytes (or 64 bits) each
  - i.e size of a pointer is independent of the type it is pointing to

Pointers are normal variables

- A pointer is a variable like any other
- The pointer is stored at its own location

```
int x = 42; // stored at 0x7ffeedbed3dc
int * ptr = &x; // stored at 0x7ffeedbed3d0
```

- We can get the address of where the pointer is stored using &

```
printf("%p\n", &ptr); // prints 0x7ffeedbed3d0
```

- We can store the address of a pointer in another pointer

```
int * * ptr_to_ptr = &ptr; // stored at 0x7ffeed7ed3c8
```

- We can change where a pointer points to

```
int y = 23; // stored at 0x7ffeebaf23c4
ptr = &y;
```

Pointers and const

- In C every variable can be annotated with the <u>*type qualifier*</u> `const`, indicating that its value can not be changed
  - This is enforced by the compiler

```
const_error.c:4:6: error: cannot assign to variable 'pi' with const-qualified type 'const float'
  pi = 2.5;
  ~~ ^
const_error.c:3:15: note: variable 'pi' declared const here
  const float pi = 3.14;
  ~~~~~~~~~~~~^~~~~~~~~
```

- Pointers can be `const` in three ways

1. The *pointer itself*, i.e. the address, cannot be changed: `float * const ptr`
2. The *value we are pointing to* cannot be changed: `const float * ptr`
3. *Both* value and pointer cannot be changed: `const float * const ptr`

Call-by-value Revisited

- We learned last time that arguments are passed *by-value*
  - i.e. the value of the argument is copied into the function parameter
- This is also true for pointers
- Arrays are treated specially
  - a *pointer to the first element* is copied instead of the entire array
- The array is treated like a pointer
  - in fact `int param[]` and `int * param` are interchangeable

```
float average(float array[], int size) {
  float sum = 0.0f;
  for (int i = 0; i < size; i++) { sum += array[i]; }
  return sum / size;
}
```

```
float average(float array[], int size);
float average(float * array, int size);
```

Pointers and Arrays

- The name of an array refers to the *<u>address of its first element</u>*

```
int vector[6] = {1, 2, 3, 4, 5, 6};
int * ptr = vector; // this is equivalent to: int * ptr = &(vector[0]);
```

- We can use the array indexing notation on pointers

```
printf("5th element: %d\n", ptr[4]); // prints "5th element: 5"
```

- The expressions `ptr[i]` and `*(ptr + i)` are equivalent
- Two important differences:
  - `sizeof` returns different values (size of array vs. size of pointer)

```
printf("%ld\n", sizeof(vector)); // prints '24' (== 6 * 4 bytes)
printf("%ld\n", sizeof(ptr)); // prints '8' (size of a pointer)
```

  - we cannot change an array, only its elements

```
vector = another_vector; // error: array type 'int [6]' is not assignable
```

Pointers and NULL

- Sometimes there is no meaningful value for a pointer at a certain time
- We use the value `0` or the macro `NULL` to represent pointing to *nothing*
- `NULL` often represents an erroneous state
  - e.g. an element was not found in an array

```
// return pointer to value found in array; NULL otherwise
float* search(float needle, float haystack[], int haystack_size) {
        for (int i = 0; i < haystack_size; i++)
                if (needle == haystack[i])
                        return &haystack[i];
        return NULL;
}
```

- Dereferencing `NULL` **will crash your program!**
  - This has led to *many* software bugs
  - The inventor of `NULL`, Tony Hoare, called it his *billion-dollar mistake*

Pointer Arithmetic

- We can use *pointer arithmetic* to modify the value of a pointer
  1. add / subtract integer values to/from a pointer
  2. subtract two pointers from each other
  3. compare pointers

```
int vector[6] = {1, 2, 3, 4, 5, 6};
int * ptr = vector; // start at the beginning
while (ptr <= &(vector[5])) {
  printf("%d ", *ptr); // print the element in the array
  ptr++;  } // go to the next element
```

- Pointer arithmetic takes into account the size of the type the pointer is pointing to

```
int * i_ptr = &i;
char* c_ptr = &c;
i_ptr++; // this adds 4-bytes (1x sizeof(int)) to the address stored in i_ptr
c_ptr+=2; // this adds 2-bytes (2x sizeof(char)) to the address stored in c_ptr
```

Pointers and structs

- Pointers are extremely useful in building data structures
- For example, a linked list
  - each node has a value and a pointer to the next node

```
struct node {
        char value;
        struct node * next;
};

int main() {
  struct node c = {'c', NULL};
  struct node b = {'b', &c};
  struct node a = {'a', &b};
  struct node * ptr = &a;
  while (ptr) {
    printf("%d\n", (*ptr).value);
    ptr = (*ptr).next;
  }
}
```

The last node in the list has a next-pointer to NULL

We use a pointer to iterate over the linked list

Command line arguments

- This is the information entered after the program name when you start the program

```
int main(int argc, char* argv[]) { ... }
```

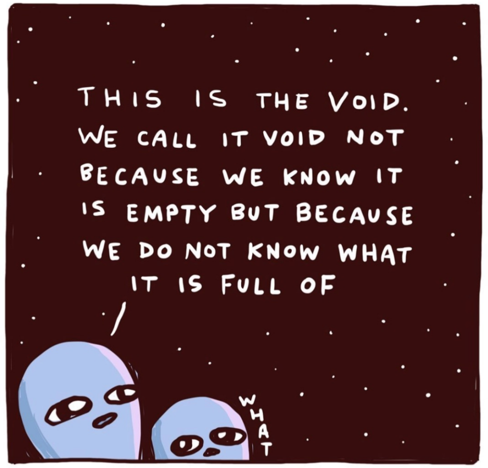
- `argc` specifies the number of command line arguments
- `argv` specifies an array of command line arguments as strings
  - A single string is represented as an array of characters: `char *`
  - The type of `argv` `char * []` can also be written `char * *`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
  // print every command line argument
  for (int i = 0; i < argc; i++)
    printf("%s\n", argv[i]);
}
```

void *

- Sometimes we want to write generic code to work with all data types
  - e.g. swapping two variables or sorting a list
  - To swap two variables x and y of arbitrary type, we copy all bytes at the location of x to y and vice versa
- For this we write a function
  - it takes two pointers and the number of bytes to be swapped
- void * is a generic pointer
  - every pointer is automatically convertible to it
  - only serves as an address pointing to *something* 🤷🏾‍♂️
- We cannot access the value we are pointing to
  - we do not know what those bits mean
  - **dereferencing a void pointer is forbidden**

```
void swap(void *x, void *y, size_t l) {
  char *a = x, *b = y, tmp;
  while(l--) {
    tmp = *a;
    *a++ = *b;
    *b++ = tmp; }
}
```



Checkpoint

- Assume that you have declared an array using Which of the following comparisons are true?

```c
int array[2][3][4];
```

```c
array[0][0] == &array[0][0][0];
array[0][1] == array[0][0][1];
array[0][1] == &array[0][1][0];
```

True

False

True

Checkpoint

- Write the prototype for a function that takes an array of pointers to type `char` as its one argument and returns `void`.

```
void func1(char *p[]);
```

```
void func1(char **p);
```

- How would such function know how many elements are in the array of pointers passed to it?
  - It has no way of knowing. This value must be passed to the function as another argument.

Lab Sheet

Tasks 4.A-B