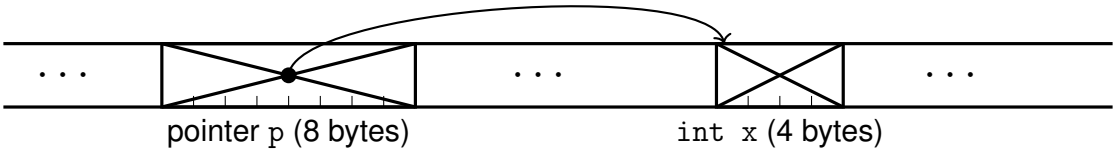
Chapter 5 – Arrays and pointers

1 Pointers and Addresses

Definition A **pointer** is simply a variable that containts the memory address of a variable (cf K&R §5.1). In all programs we've written so far, we used variables to work with numbers: loop indices, parameter values, etc. And memory addresses are not different from ordinary numbers.

At runtime, all variables are stored in memory: a `char` takes one byte, an `int` is typically 4 bytes, etc. The size needed to store a pointer depends on the hardware architecture. For instance, a so-called "32-bit CPU" works with addresses encoded on 4 bytes. However, modern devices (e.g. laptop, smartphone) are typically based on a "64-bit CPU" which means that a pointer occupies 8 bytes. In the memory diagram below, a pointer `p` contains the address of an integer `x`. We say that `p` **points to** `x`.



Syntax If "`T`" is a type name, like `int` or `float`, then a variable declared with type "`T*`" is a "pointer to some T", or just a "T pointer". It does not store a value of type T, but the *address of a value of type T*. As illustrated below, such an address can be obtained with the **address of** operator (spelled `&`) also known as the **reference to** operator. For any variable `V` of type `T`, expression `&V` evaluates to a pointer (of type `T*`) to `V`.

Conversely, we can "follow" a pointer with the **dereference** operator (spelled `*`). When `P` is a pointer of type `T*`, expression `*P` (we say "star-P") is of type `T` and denotes the variable pointed by `P`.

```
int x;    // x is a non-pointer (aka 'scalar') variable

int *p;   // p is a pointer to some int

p = &x;   // p now points to x

*p = 42;  // x now equals 42
```

Exercise 1 Go read K&R §5.1 again until you understand pointers. Ask us about anything confusing.

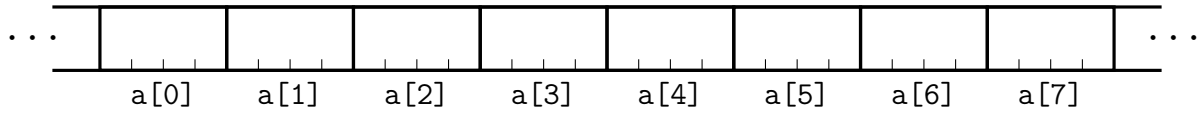
Pointers as arguments In C, function arguments are always **passed "by value"**: each argument is evaluated as an expression before the call, and only the resulting value is "copied" into the called function. However, pointer types can be used as function parameters just like any other data type ! When used correctly, this trick enables a called function to access (and modify) variables belonging to a calling function.

Exercise 2 Type in the program below, execute it, then explain in one sentence what is wrong with this code. Then fix the type signature of `swap()`, rewrite its body accordingly, and adjust the call site.

```
#include <stdio.h>

void swap(int a, int b)
{
    int temp=a;
    a=b;
    b=temp;
}

int main()
{
    int x = 5;
    int y = 7;
    printf("x=%d, y=%d\n", x, y);
    swap(x, y);
    printf("x=%d, y=%d\n", x, y);

    return 0;
}
```

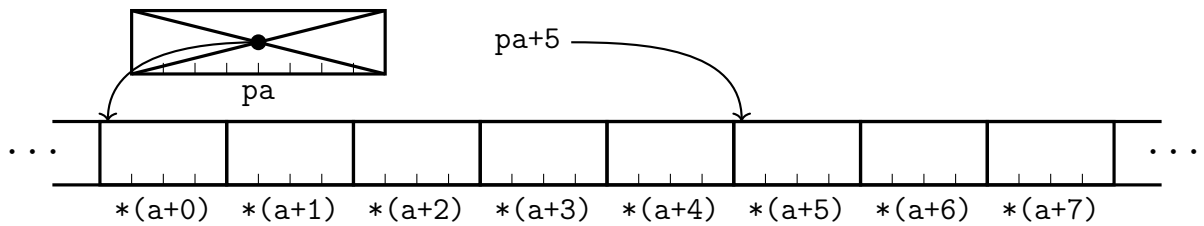
2 Arrays

Definition An **array** is a block of memory storing consecutive elements of the same data type. For instance, declaration `int a[8];` allocates an array of eight integers named `a[0]`, `a[1]`, ..., `a[7]`, like illustrated below.



In C, arrays and pointers are closely related. Any operation that can be achieved by array subscripting (i.e. using notation `a[i]`) can also be done with pointers (i.e. using the star operator). For instance, if `pa` is a pointer to an integer, declared as `int *pa;` then the assignment `pa = &a[0];` sets `pa` to point to element of `a` at index zero.[1] In other words, `pa` now contains the address of `a[0]`, so notations `a[0]` and `*pa` are synonyms.
Also, by definition, an array name without brackets evaluates to the address of the beginning of the array, i.e. the address of its first element. In other words, writing `pa=a;` is the same as writing `pa=&a[0];`

Pointer arithmetic By definition, if a pointer `pa` points to a particular element of an array, then `pa+1` points to the next element, `pa+i` points "`i` elements after `pa`", and `pa-i` points "`i` elements before `pa`". In other words, if `pa` points to `a[0]`, then `*(pa+i)` refers to element `a[i]`, as illustrated below.



[1] Yes, element at index zero really is the *first* element, but then element at address 3 really is the *fourth* element, and so on. It does get confusing.

Exercise 3 For more details on arrays, read K&R §5.3. Ask us about anything confusing.

Exercise 4 Using the code below as a template, write short programs to:

- print all values of `tab`, using array subscripting
- print all values of `tab`, using only pointer operations
- print all the addresses of elements in `tab`, using array subscripting
- print all the addresses of elements in `tab`, using pointer operations

The length of the array is a known constant (here 10).

```
int tab[] = {4,-5,8,23,-15,37,89,12,1,-42};

int main()
{
   for(int i=0; i<10; i++)
   {
        ...

   }
}
```

Exercise 5 Write a program that loops over an array of numbers and finds the maximum value. The length of the array is a known constant.

2.1 Bubble Sort

Disclaimer If you took IST-ASM earlier this semester, you already have followed the instructions below. Please move on directly to the C implementation.

In this section, you will implement a simple sorting algorithm. As stated by Wikipedia[2], **bubble sort** *"repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list. "*

Example We start with the array T initialized as follows: (6 5 3 1 8 7 2 4). During each pass, the algorithm iterates on all pairs T[i]/T[i+1] and swaps values if needed:

i=0: $6 > 5$ so we swap them: (**6 5** 3 1 8 7 2 4) $\rightarrow$ (**5 6** 3 1 8 7 2 4)
i=1: $6 > 3$ so we swap them: (5 **6 3** 1 8 7 2 4) $\rightarrow$ (5 **3 6** 1 8 7 2 4)
i=2: $6 > 1$ so we swap them: (5 3 **6 1** 8 7 2 4) $\rightarrow$ (5 3 **1 6** 8 7 2 4)
i=3: $6 \leqslant 8$ so we leave the 6 in place and we move on
i=4: $8 > 7$ so we swap them: (5 3 1 6 **8 7** 2 4) $\rightarrow$ (5 3 1 6 **7 8** 2 4)
i=5: $8 > 2$ so we swap them: (5 3 1 6 7 **8 2** 4) $\rightarrow$ (5 3 1 6 7 **2 8** 4)
i=6: $8 > 4$ so we swap them: (5 3 1 6 7 2 **8 4**) $\rightarrow$ (5 3 1 6 7 2 **4 8**)

We're now finished with the first pass, let's start again:

i=0: $5 > 3$ so we swap them: (**5 3** 1 6 7 2 4 8) $\rightarrow$ (**3 5** 1 6 7 2 4 8)
i=1: $5 > 1$ so we swap them: (3 **5 1** 6 7 2 4 8) $\rightarrow$ (3 **1 5** 6 7 2 4 8)
i=2: $5 \leqslant 6$ so we leave the 5 in place and we move on
i=3: $6 \leqslant 7$ so we leave the 6 in place and we move on
i=4: $7 > 2$ so we swap them: (3 1 5 6 **7 2** 4 8) $\rightarrow$ (3 5 1 6 **2 7** 4 8)
i=5: $7 > 4$ so we swap them: (3 1 5 6 2 **7 4** 8) $\rightarrow$ (3 5 1 6 2 **4 7** 8)
i=6: $7 \leqslant 8$ so we leave them in place, and we have reached the end of the array.

We're now finished with the second pass.

[2] `https://en.wikipedia.org/wiki/Bubble_sort`

Exercise 6 **(pen & paper)** Continue unrolling the algorithm until a whole pass does no swap.

Exercise 7 Write a C program that implements bubble sort in an array of integers. Like in all similar exercises, the length of the array is an known constant.

To make your work easier, you may want to go after simpler subgoals first, for example:

- Given an index i, swap array elements T[i] and T[i+1];
- Perform a single pass on the entire array, swapping elements as needed;
- Repeat such passes until the array is fully sorted.

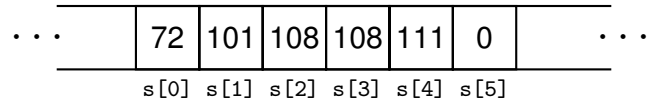
3 Strings

In C, strings are implemented as arrays of characters, with no explicit indication of length. Instead, a null byte marks the end of the string. For instance, a literal **string constant** written as `"Hello"` is an array of 6 elements of type `char`. Why 6 and not just 5 ? This is because C strings are always terminated with a byte of value zero, so that programs can find the end.

In a program, you would typically declare such a string with `char *s="Hello";` or `char s[]="Hello";` both of which are equivalent to this more verbose, less elegant syntax:

```
char s[]={'H', 'e', 'l', 'l', 'o', '\0'};
```

Note that we don't need to explicitely indicate the size of an initialized array: the compiler will count elements and allocate six bytes for us. The resulting memory layout is illustrated below:



Exercise 8 Go read the first page of K&R §5.5 for more info about strings. Ask us about anything confusing.

Exercise 9 Write a program which loops over characters in a string, and prints every value as an ascii character, then a decimal number, then a hexadecimal number:

```
% ./deascii
ASCII 'h' = dec 104 = hex 68
ASCII 'e' = dec 101 = hex 65
ASCII 'l' = dec 108 = hex 6c
ASCII 'l' = dec 108 = hex 6c
ASCII 'o' = dec 111 = hex 6f
```

Exercise 10 Write a function with signature `int my_strlen(char *s);` which returns the length of string `s`, excluding the terminal `'\0'`. Test it by comparing its behaviour with the standard function `strlen()` from `string.h`.

4 Passing command-line arguments to your programs

One first way to interact with an excecutable program is to provide it with parameters when we launch it from the command line. You've already passed parameters to shell programs such as `ls` or `mkdir`.
You can also pass command-line arguments to a C program of your own. From the program's point of view, CLI arguments become the parameters of the main function. Go and have a look at `https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html`. It says, among other interesting things: *"command line arguments are the whitespace-separated tokens given in the shell command used to invoke the program"*. Look at the program below.

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  printf("hello!\n");
  return 0;
}
```

Notice the change in signature: the `main` function now has 2 arguments:

- `argc` is an integer indicating the number of arguments given to the command line;
- `argv` is an array (of size `argc`) of null-terminated character strings. Argument `argv[0]` will always be the name of the executable.
    - Note that here we spelled the signature with both a star and brackets, to convey the idea of an array of pointers. But this is the same as saying `char **argv`.

Exercise 11 Go read K&R §5.10. Ask us questions about anything confusing.

Exercise 12 Write a new C program with a `main()` function that follows the new signature:

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

Your program should print out its number of arguments, then every argument received, one per line:

```
% ./main A B CDEF 123
nb args: 4
arguments:
0: ./main
1: A
2: B
3: CDEF
4: 123
```

5 Putting it all together: parsing integers

Exercise 13 The standard library offers function `int atoi(char *s)` (in `stdlib.h`) which is able to interpret a text string with decimal digits into an integer value. Write a C program which parses each of its command-line arguments with `atoi()` and prints the corresponding number:

```
% ./parse_atoi 1 30 -42 +7 0 00 abcd
1 -> 1
30 -> 30
-42 -> -42
+7 -> 7
0 -> 0
00 -> 0
abcd -> 0
```

Exercise 14 Now write a function `int my_atoi(char *s);` which does the same as `atoi()`. Hint: remember that consecutive digits have consecutive ASCII codes, e.g. `'7'-'0' == 55-48 == 7`. Validate your implementation with the program from the previous question.

Exercise 15 (optional) Write a function `int my_htoi(char *s);` which converts a string of hexadecimal digits. For instance, `my_htoi("2A")` will return 42.
Then use that function in an actual program:

```
%./parse_htoi 1 30 2A FFFF
1 -> 1
30 -> 48
2A -> 42
FFFF -> 65535
```