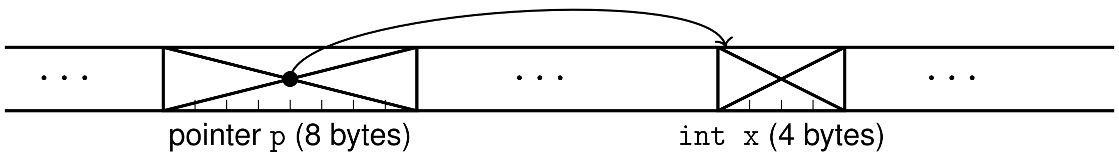
Chapter 4 – Arrays and pointers

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 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 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
```

For more info on pointers, read K&R §5.1 again.

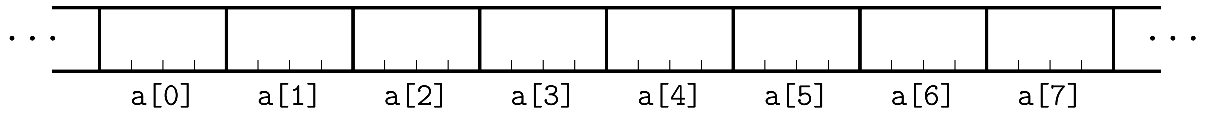
Pointers as arguments Pointer types can be used for function arguments just like any other data type. However, remember that arguments are always passed "by value": each parameter is evaluated as an expression and only its value is passed to the called function.

Exercise 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 the 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;
}
```

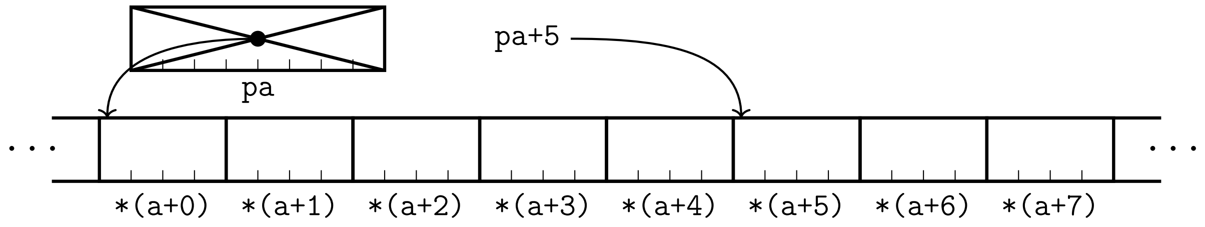
2 Arrays

Definition An **array** is a block of memory storing consecutive elements of the same data type. For instance, the declaration `int a[8];` allocates an array of ten 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 zero of `a`. 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 its element zero, so in practice the assignment above could have been written as `pa=a;`

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. In other words, if `pa` points to `a[0]`, then `*(pa+i)` refers to element `a[i]`, as illustrated below.



For more details on arrays, read K&R §5.3.

Exercise 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

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

int main()
{

}
```

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

We’re now finished with the second pass.

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

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

Exercise Write a C program that implements bubble sort in an array of integers. Like in the "find the maximum" exercise, the length of the array is an explicit parameter.

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 stored as arrays of characters. For instance, a literal **string constant** written as `"Hello"` is an array of 6 elements of type `char`. Even though we only want 6 letters, a C string is always terminated with a byte with value zero, so that programs can find the end.

In a program, you would typically declare a string with `char s[]="Hello";` which is equivalent to this more verbose, less elegant syntax[2]:

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

This array of `char`s can then be manipulated just like any other array. For example, you can print the ASCII character for the i-th element of a string `s` with `printf("%c",s[i]);` or just its numerical value with `printf("%x",str[i]);`
For more info about strings, read the first page of K&R §5.5.

Exercise Write a function with signature `int strlen(char[] s);` which returns the length of string `s`, excluding the terminal `'\0'`.

Exercise Write a function `int htoi(char[] s);` which converts a string of hexadecimal digits, including an optional `"0x"` into its equivalent integer value. For instance, `htoi("2A")` will return 42.
Hint: remember that consecutive digits have consecutive ASCII codes, e.g. `'7'-'0' == 55-48 == 7`.

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. From the program's point of view, these are accessed through the parameters of the main function. Go and have a look at . 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.

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

Contrary to the one used in the previous chapter, the `main` function now has 2 arguments[3]:

- `argc` holds the number of arguments given to the command line;
- `argv` is an array of `argc` strings of characters. Note that the first element of this array is the name of the binary program itself.

Side note Note also that we have replaced the call to `puts` by a call to the `printf` function.

[2]Character literal `'0'` denotes the null byte

[3]These are the only two forms for the main function of a C program: 1) no arguments; 2) exactly two of type `int and char **` type. For more information on types in C, see next chapter.

Exercise Write a new C program with a main function that follows the signature

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

Using the `printf` function, print the number of arguments passed to your program at the command line. To test your program, execute it with different numbers of arguments.

Exercise Write a new C program that prints all the command line arguments it receive when you launch it from the command line.