HPC1: Lecture 1

Andreas F. Borchert and Michael C. Lehn

15 October 2018

Contents

1 Required tools and skills **1**

2 Virtual memory **1**
2.1 Traditional memory layout of a UNIX process 1
2.2 Notes on variables and memory allocation in C 2
2.3 More about dynamic memory allocation and pointers 4
2.4 More on releasing allocated memory 4

3 Dealing with vectors in C **5**
3.1 Required information for vectors 5
3.2 Example without a vector struct 5

1 Required tools and skills

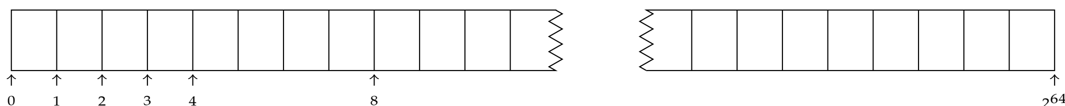
1. You should be familiar with using the UNIX terminal and an editor (e. g. *vim* which we ourself use and recommend).
2. If you want to use your own computer you need a C/C++ compiler that supports at least the standards C99 and C++17 (e. g. the GNU C++ compiler with version 7.1 or higher).
3. You should be fairly familiar with the C programming language.

2 Virtual memory

We will often use the *virtual memory* to illustrate and explain certain programming concepts. On a 64 bit computer the virtual memory is an array with 2^{64} addressable bytes. That means:

- We have 2^{64} memory cells and each cell contains a byte value.
- Each cell has an address within the range from 0 to $2^{64} - 1$.

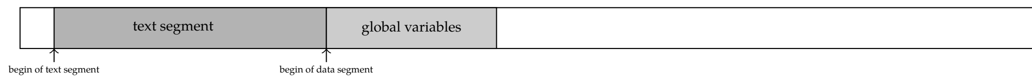
We can illustrate this as follows:



2.1 Traditional memory layout of a UNIX process

When a program gets executed the operating system loads its code into memory and initializes its global variables in memory. Furthermore it initializes some CPU registers such that the execution starts with the first statement of function *main*:

- The state of the virtual memory right after the process was created can be illustrated as follows:



The so called *text segment* contains the machine instructions generated by the compiler from our C source files. The *data segment* contains the global variables of the C program.

- When the program is running some functions will called (even in a minimalistic program at least function *main* will be called). Each function can have some local variables that will be located on the stack. When a function calls another function the stack grows (and when a function returns the stack shrinks). Typically the stack grows towards smaller addresses, i. e. in our illustration it grows to the left.

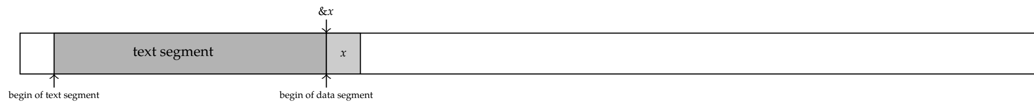
Besides locally defined variables we can use dynamically allocated memory for storing data. Such memory must be explicitly allocated and deallocated, e. g. by functions *malloc* and *free*.

2.2 Notes on variables and memory allocation in C

Here is a simple C program that exemplifies usage of global and local variables and usage of dynamically allocated memory:

```c
#include <stdlib.h> // include declarations for malloc, free

double x; // global varibale

void f()
{
    double y; // local variable (of function f)

    double *v; // local variable (of function f)

    v = malloc(sizeof(double)*10); // allocate memory block and assign of the block to v

    *v = 42; // dereferencing pointers and pointer
    *(v + 4) = 2018; // arithmetic are explained below in
    v[4] = 2018; // more detail

    free(v); // release memory block
}

int main()
{
    double y; // local variable (of function main)

    f();
}
```

- The program consists of two functions (i. e. *f* and *main*) and one global variable (i. e. *x*). So when we start the (compiled) program the memory is setup as follows:

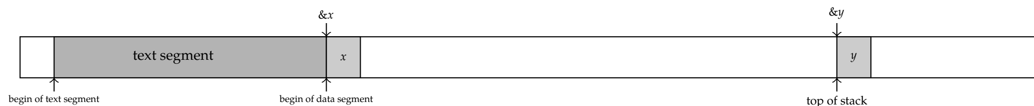


The text segment contains code for the functions *main* and *f* from our C program and (assuming static linkage) functions from the system library like *malloc* and *free*.

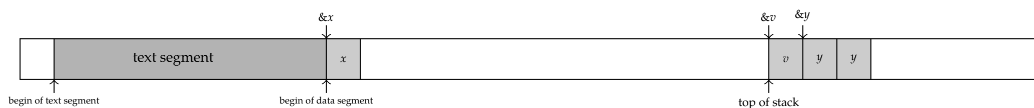
Note that in line 3 the global variable x was defined uninitialized. For global variables (and not for local variables!) that means it gets automatically initialized with zeros. Furthermore, the IEEE standard specifies that a variable of type **double** has a size of eight bytes (i. e. 64 bits). In the picture the address of variable x was denoted by $\&x$ (the common C syntax for denoting the address of a variable).

- Whenever a function gets called it can use the stack for its local variables. Besides some other details we ignore the fact that the stack also might contain things like return addresses

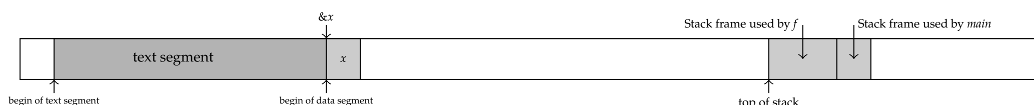
  In our example first function *main* gets called. So right after the call we have the following situation:



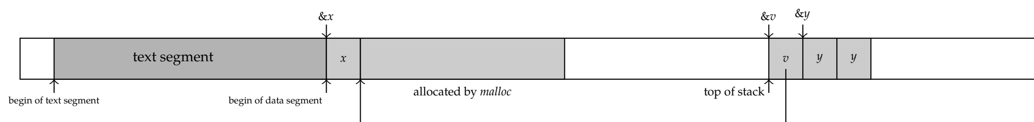
Then function *f* (which has local variables y and v) gets called from within *main* and we have



You should be aware (and not confused) that there are two variables y on the stack! Note that this moment the stack is used to store local variables of functions *main* and function *f*. As each function has its own part of the stack (*stack frame*) there is no ambiguity:



- Function *f* allocates memory from the heap in line 11. After this statement we have



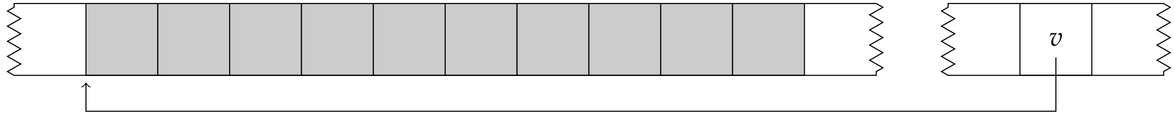
We will have a closer look at dynamic memory allocation below. For the moment be aware of the following:

  – Allocating a memory block means that this block is somehow marked as being used. That mean a subsequent call to *malloc* will allocate a distinct block (that does not interleave with any previously allocated block).
  – You can release a memory block with *free*. It is notable that this function just requires the address of the block you want to release and not the size of the block (so the later information must be stored in some hidden data structure).

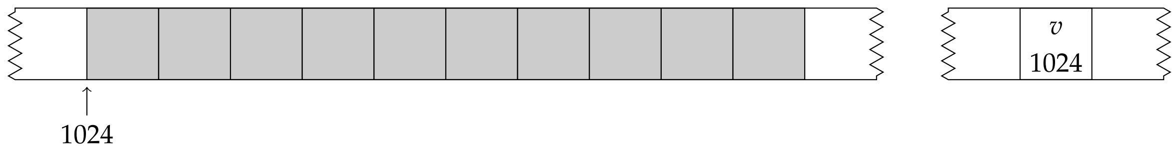
2.3 More about dynamic memory allocation and pointers

Variable v has type **double** $*$. That means it stores an address which we can regard to as a pointer. Let's have a closer look at the memory after the statement in line 11:

- Variable v has type **double** $*$. That means it stores an address which we can regard to as a pointer. The allocated memory block consists of $10 \cdot 8$ bytes (recall: eight bytes are needed to store one variable of type **double**). So we can visualize the situation after line 11 as follows:

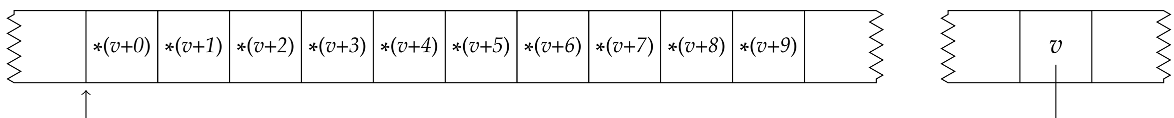


- Using an exemplary value for the address returned by *malloc* it might be more obvious that a pointer is just a more vivid illustration for interpreting an integer as an address:

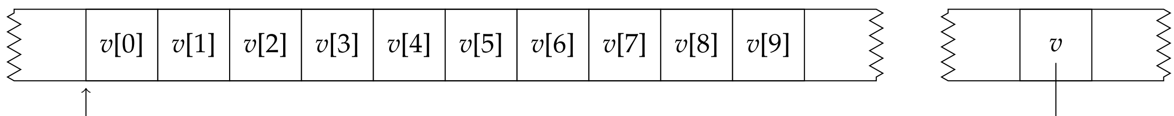


In lines 14–16 we assign values to some doubles within the allocated memory block. We use two different notations for *dereferencing* pointer v. We can illustrate the meaning of these notations as follows:

- Using pointer arithmetic and the dereference operator (denoted by $*$) we can refer to single double values by



- Equivalently we can use the element access operator (denoted by []):



2.4 More on releasing allocated memory

Whenever memory was allocated in a program it also has to be released. More precisely, it has to be release exactly once.

- If a program does not release the memory it leads to memory leaks. In our example this would happen without the *free* statement in line 18.
- If memory gets released more than once it might corrupt the internal data structures of the dynamic allocation machinery. For example, in a code like this:

```
1   double *v = malloc(10*sizeof(double));
2
3   // ...
4
5   free(v);
6   free(v);
```

In our example both cases can be avoided easily. In more complex cases it requires more discipline and planning!

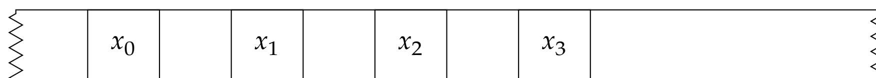
3 Dealing with vectors in C

In the mathematical description of a vector with n elements we will start indexing of elements at zero, e. g.

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

This will make it easier to translate a mathematical algorithm to C code. In typical applications the vector elements will sometimes be stored in a contiguous block:

or scattered with a constant stride. For example, with a stride of two we have following layout:



Note that using pointer arithmetic we can regard the first case as a special case with a stride equal to one. Traditionally, the stride in memory between vector elements is also called *increment*.

3.1 Required information for vectors

In order to describe how vector elements are stored in memory we need to know

- the vector length,
- the address of the first element (in our example the address of x_0) and
- the increment between vector elements.

For the vector length we will use an integer of type *size_t* (which is an unsigned integer type) and for the increment an integer of type *ptrdiff_t*.

3.2 Example without a vector struct

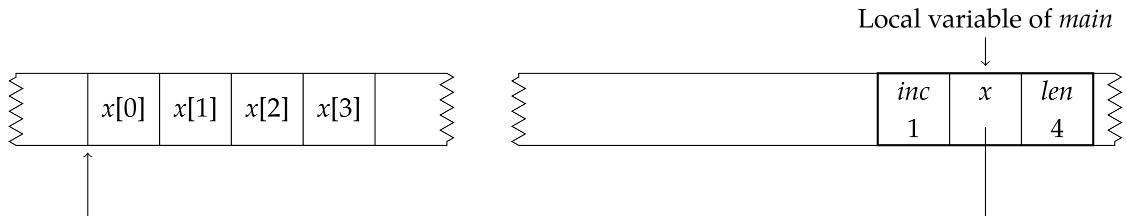
Consider the following example program:

```
1  #include <stdlib.h> // include declarations for malloc and free
2  #include <stddef.h> // include declarations for size_t and ptrdiff_t
3
4  void
5  init_vector(size_t n, double *x, ptrdiff_t incX)
6  {
7      for (size_t i=0; i<n; ++i) {
8          x[i*incX] = i + 42;
9      }
10 }
11
12 int
13 main()
14 {
15     size_t len = 4;
16     double *x = malloc(len*sizeof(double));
17     ptrdiff_t inc = 1;
18
```

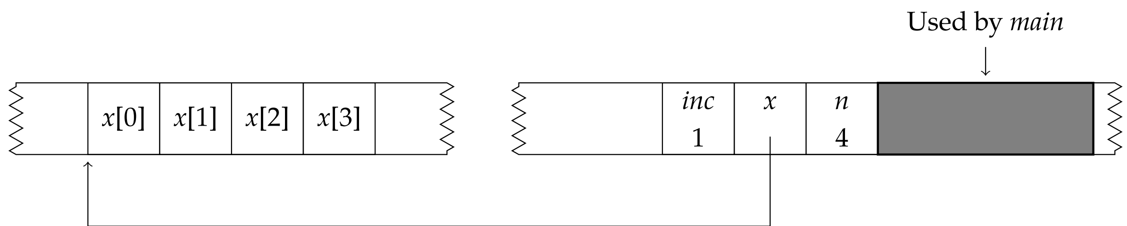
```
19    init_vector(len, x, inc);
20
21    init_vector(2, x+1, 2);
22
23    free(x);
24 }
```

When we run the compiled program the memory gets modified though statements as follows:

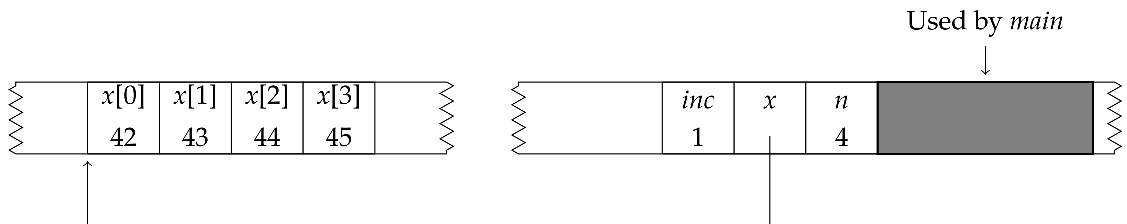
- In lines 15–17 we define and initialize the local variables *len*, *x* and *inc*. These variables are used to describe the storage of a vector. Elements are stored in a dynamically allocated memory block. After this lines we have:



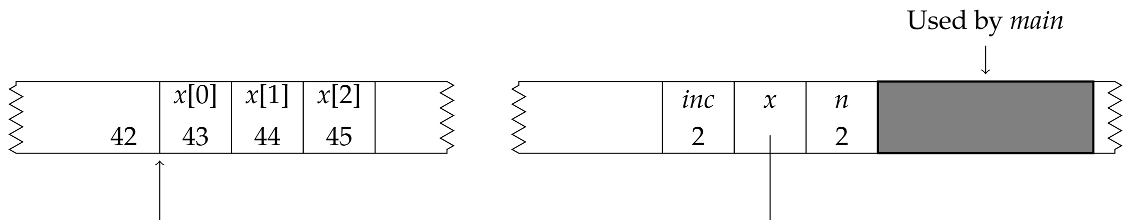
- In line 19 we call function *init_vector*. Arguments are passed to the functions as copy (*call by value*). So when the program enters the function after line 6 we have:



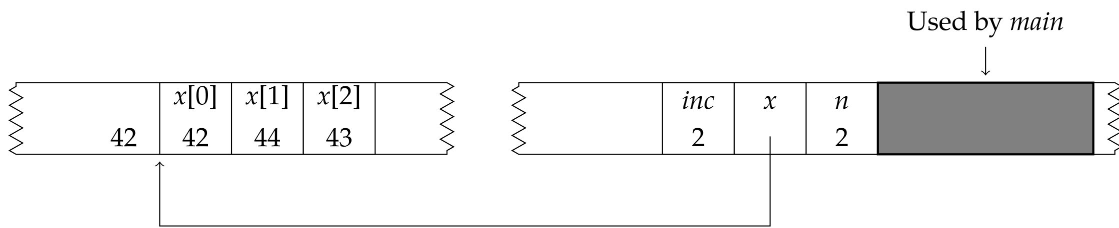
The for loop in lines 7–9 modifies the vector elements to



- In line 21 we call function *init_vector* again. However, we changed the parameters. So when we enter the function the stack looks like this:



hence the function modifies the memory content to



- After the function returns to *main* we have