Imperial College
London

Software Engineering 1: Introduction to Computing

6 – Raw memory (part a: static memory)

Max Cattafi (m.cattafi@imperial.ac.uk)

So far we have been using features of the language and of the standard library that allow developers to focus on the application rather than on implementative details such as where exactly the data is stored, how much memory is allocated and so on.

These notes are an introduction to aspects related to memory management in C++, in particular arrays, pointers, the relationship between pointers and arrays, and in *part b* (manual) dynamic memory allocation.

Please note that just because we introduce arrays it doesn't mean that therefore we switch to using arrays instead of vectors: in applications at a medium or higher level of abstraction **it is often the case that vectors** (or some other container or data structure from the standard library) **are a better choice than arrays**.

Similar remarks apply to pointers. In fact in contemporary C++ using the kind of pointers covered in these notes ("raw pointers") is not advised (there are other, more sophisticated, kinds of pointers).

However it is worth learning these topics for a variety of reasons, including:

- It allows us to gain a better understanding of what's inside the features we use, for instance how vectors are implemented. As a result we become more competent users of these features.
- There are applicative domains, such as systems programming, in which it is actually necessary to work at a lower level of abstraction.
- Aspects of memory management are also present in C++ programming at a higher level of abstraction, for instance when some features of

object oriented programming are used.

A sequence of numbers (using arrays)

Let's begin with a simple example using arrays. The following program reads a sequence of numbers, stores it in an array and prints it back in reversed order. The sequence is terminated when the value 0 is encountered (and the 0 shouldn't be included in the array).

Arrays are similar to vectors in the sense that they can be used to store a collection of elements (of the same type) which can then be accessed using their respective indices. As for vectors, the index of the first element is 0.

However allocation of memory for arrays is not automatically managed when a new element is added. The memory is allocated at the beginning in a fixed amount. This represents the **maximum** number of elements that the array can store. This is the **physical size** of the array.

Moreover there is no built-in way to know how many elements have been stored in the array, this information needs to be kept in a separate variable which keeps the count. This is the **logical size** of the array.

```
1  #include <iostream>
2
3  int main(){
4
5      int int_a[128];
6      // declaring an array of int called int_a,
7      // allocating memory for 128 integers;
8      // this will allocate 128 cells
9      // that initially have no meaningful values;
10     // 128 is the physical size of this array
11     // (expressed in terms of number of elements)
12     // and it's the maximum number of elements it can contain
13
14     int tmp;
15
16     int i = 0;
17     // declaring an index i
18     // initialized at 0
19
20     std::cout << "enter a number:" << std::endl;
21     std::cin >> tmp;
22
23     while(tmp != 0){
```

```
24          int_a[i] = tmp;
25          // at each iteration
26          // we overwrite the previous (non meaningful)
27          // content of the cell
28
29          i++;
30          // we update the index by incrementing it
31
32          std::cout << "enter a number:" << std::endl;
33          std::cin >> tmp;
34      }
35
36      int int_a_size = i;
37      // when we are out of the loop i contains the logical
38      // size of the array, that is how many elements have been stored
39      // we need to save this information somewhere
40      // (if it's overwritten it can't be retrieved)
41
42      std::cout << "you entered the following numbers:" << std::endl;
43
44      // similar to what we would do with vectors
45      // (but we don't have a size() member function);
46      // the index of the first element is 0
47      // the index of the last element is int_a_size - 1
48
49      for(int i = 0; i < int_a_size; i++){
50          std::cout << int_a[i] << std::endl;
51      }
52
53      std::cout << "that in reverse order are:" << std::endl;
54
55      for(int i = int_a_size - 1; i >= 0; i--){
56          std::cout << int_a[i] << std::endl;
57      }
58
59      return 0;
60  }
```

To improve readability and make the code more maintenable, the physical size of arrays is often specified using a preprocessor constant:

```
1   #include <iostream>
2
3   #define ASIZE 128
4
5   int main(){
6
```

```
7       int int_a[ASIZE];
8
9       // the rest of the program is unchanged
10
11      int tmp;
12
13      int i = 0;
14
15      std::cout << "enter a number:" << std::endl;
16      std::cin >> tmp;
17
18      while(tmp != 0){
19          int_a[i] = tmp;
20          i++;
21
22          std::cout << "enter a number:" << std::endl;
23          std::cin >> tmp;
24      }
25
26      int int_a_size = i;
27
28      std::cout << "you entered the following numbers:" << std::endl;
29
30      for(int i = 0; i < int_a_size; i++){
31          std::cout << int_a[i] << std::endl;
32      }
33
34      std::cout << "that in reverse order are:" << std::endl;
35
36      for(int i = int_a_size - 1; i >= 0; i--){
37          std::cout << int_a[i] << std::endl;
38      }
39
40      return 0;
41  }
```

Exercise

Note: using arrays as parameters of functions requires additional knowledge so do not attempt it for this exercise.

Write a program which reads from the user the name of a file, reads a sequence of integers from that file, stores the numbers into an array and prints its content, then reads from the user an integer and prints on the screen whether it is contained in the file or not and at which line.

As shown in the examples above, arrays allocate a fixed amount of memory at the beginning, when they are declared. The amount is left up to you,

> pick a number that can be suitable for the use case without allocating too much unused memory.

Pointers

The following example introduces pointers. The data stored by our programs is located at certain addresses in memory. **Pointers are variables that contain memory addresses**. For example an `int` variable will be allocated at a certain address when declared, and a variable of type pointer to `int` (often shortened as just "a pointer to `int`") can be used to store the memory address of the `int` variable. (Indeed also pointers are located at a certain memory address, so you can have pointers to pointers and so on.)

This example introduces also several operators: `&`, `*` and `->` as described in the comments.

```
1  #include <iostream>
2
3  // the following structured data type will be used for some of the examples below
4
5  struct point{
6      double x;
7      double y;
8
9  };
10
11 int main(){
12     int a = 15;
13     std::cout << "variable a of type int" << std::endl;
14     std::cout << "with value: " << a << std::endl;
15     std::cout << "at address: " << &a << std::endl;
16     // the operator & used in the previous line returns
17     // the memory address of a variable;
18     // _it is not_ a reference: same symbol
19     // but a different and separate meaning
20
21     int* pa;
22     // pa is declared as a variable of type pointer to int
23
24     pa = &a;
25     // the address of a is assigned to pa
26     // (it is now the value contained in pa)
27
28     std::cout << std::endl; // just to add some spacing from the previous print
29
30     std::cout << "variable pa of type pointer to int" << std::endl;
31     std::cout <<  "with value: " << pa << std::endl;
32     std:: cout << "(the value will be the address of variable a as printed above)" << std::endl;
33     std::cout << "at address: " << &pa << std::endl;
34
35     std::cout << std::endl;
36
```

```
37      std::cout << "the value of a can be accessed with *pa:" << std::endl;
38      std::cout << "a: " << a << " " << "*pa: " << *pa << std::endl;
39      // the operator * used in the previous line
40      // returns the value pointed by an address contained in a pointer;
41      // it is called dereferencing operator
42      // and _it is not_ the same * used
43      // in the pointer variable declaration above:
44      // same symbol but a different and separate meaning
45
46
47      std::cout << std::endl;
48
49      *pa = 3;
50      // the value accessed with dereferencing
51      // can also be changed, the line above
52      // can be considered equivalent to a = 3
53
54      std::cout << "after changing the value of a using pa:" << std::endl;
55      std::cout << "a: " << a << " " << "*pa: " << *pa << std::endl;
56
57      std::cout << std::endl;
58
59      int b = 10;
60      pa = &b;
61      // pa now points to b
62      std::cout << "pa now points to b:" << std::endl;
63      std::cout << "&b: " << &b << " " << "pa: " << pa << std::endl;
64      std::cout << "a: " << a << " " << "b: " << b << " " << "*pa: " << *pa << std::endl;
65
66      std::cout << std::endl;
67
68      *pa = 6;
69      std::cout << "after changing the value of b using pa:" << std::endl;
70      std::cout << "a: " << a << " " << "b: " << b << " " << "*pa: " << *pa << std::endl;
71
72      std::cout << std::endl;
73
74      point p;
75      p.x = 0.5;
76      p.y = 0.6;
77
78      std::cout << "p:" << std::endl;
79      std::cout << "value: " << p.x << " " << p.y <<std::endl;
80      std::cout << "address: " << &p << std::endl;
81
82      std::cout << std::endl;
83
84      point* ppoint = &p;
85      // pointer ppoint is assigned the memory address of p
86      // "ppoint points to p"
87
88      std::cout << "ppoint: " << std::endl;
89      std::cout << "value: " << ppoint << std::endl;
90
91      (*ppoint).x = 0.7;
92      // accessing and changing the x coordinate
93      // of p through ppoint:
94      // first dereferencing, then dot notation
95      // (parentheses needed)
96
97      ppoint->y = 0.8;
98      // accessing and changing the y coordinate
```

```
    // this is equivalent to (*pp).y = 0.8
    // just more compact

    std::cout << std::endl;

    std::cout << "p after the coordinates change through ppoint:" << std::endl;
    std::cout << p.x << " " << p.y <<std::endl;

    return 0;
}
```

> **Exercise**
>
> Compile and run the example above.

Pointers and arrays

The following program illustrates the relationship between pointers and arrays.

```
#include <iostream>
#define ASIZE 128

int main(){

    int na[ASIZE];
    int* pna;

    na[0] = 1;
    na[1] = 2;
    na[2] = 3;

    std::cout << "na[0]: " << na[0] << " &na[0]: " << &na[0] << std::endl;
    std::cout << "na[1]: " << na[1] << " &na[1]: " << &na[1] << std::endl;
    std::cout << "na[2]: " << na[2] << " &na[2]: " << &na[2] << std::endl;

    std::cout << "na: " << na << std::endl;
    std::cout << "(it will be the same value as &na[0])" << std::endl;

    std::cout << std::endl;

    pna = na;
    // now pna = na = &na[0]

    *pna = 11;
    *(pna + 1) = 22;
    *(pna + 2) = 33;

    // these additions are done using
    // _pointer arithmetics_
    // (see below)

    std::cout << "after changing the values using pna" << std::endl;

    std::cout<<"na[0]: "<<na[0]<<"&na[0]: "<<&na[0]<<"pna: "<<pna<< std::endl;
```

```
36      std::cout<<"na[1]: "<<na[1]<<"&na[1]: "<<&na[1]<<"pna+1: "<<pna+1<<std::endl;
37      std::cout<<"na[2]: "<<na[2]<<"&na[2]: "<<&na[2]<<"pna+2: "<<pna+2<<std::endl;
38
39      std::cout << std::endl;
40
41      std::cout << "sizeof(int): " << sizeof(int) << std::endl;
42
43      // sizeof is a function (available in the core language)
44      // which returns the size (in bytes) of a type
45
46      // if pna is an int*
47      // (pna + 1) in pointer arithmetics is actually
48      // (pna + 1 * sizeof(int)) (goes to next variable)
49
50      std::cout << std::endl;
51
52      pna[0] = 4;
53      pna[1] = 5;
54      pna[2] = 6;
55
56      // the square brackets (indexing) operator is the same
57      // as adding offset in pointer arithmetics + dereferencing
58      // in other words pna[1] = 5 is equivalent to *(pna + 1) = 5
59      // (this doesn't work in the same way with vectors)
60
61      std::cout << "after changing the values using pna" << std::endl;
62
63      std::cout << "na[0] = " << na[0] << std::endl;
64      std::cout << "na[1] = " << na[1] << std::endl;
65      std::cout << "na[2] = " << na[2] << std::endl;
66
67      int b;
68      pna = &b;
69      // ok
70
71      // na = &b;
72      // not ok (wouldn't compile):
73      // arrays are not pointers
74      // (they point to a fixed address and
75      // allocate memory when declared)
76
77      return 0;
78  }
```

> **Exercise**
>
> Compile and run the example above.

Pointers and functions

The following program contains a function which takes in input the memory addresses of two integer variables and, using these addresses, swaps their values.

In the C programming languages, for example, references don't exist, so

a behaviour similar to "passing by reference" is achieved by "passing by pointer" as in the example below.

```
#include <iostream>

void swapintp(int* pn1, int* pn2);

int main(){

    int a, b;

    std::cin >> a >> b;
    std::cout << "a: " << a << " " << "b: " << b << std::endl;

    swapintp(&a, &b);
    // the ampersands above are the operators which
    // return the address of a variable as in the examples above
    // (_these are not_ reference declarations)

    std::cout << "a: " << a << " " << "b: " << b << std::endl;

    return 0;
}

void swapintp(int* pn1, int* pn2){
    // the * in the line above is syntax for
    // the declaration of
    // pointer variables (or pointer parameters)
    // which is different from the * in the lines below
    // used for dereferencing
    int tmp = *pn1;
    *pn1 = *pn2;
    *pn2 = tmp;
}
```

Exercise

Compile and run the example above.

A sequence of numbers (using arrays and functions)

The following program is similar to one of the examples above, however this time instead of printing the input array in reverse order, we create a second array that contains the elements of the first array in reverse order and then we print it.

We do so by using a function which takes in input the input array (in order to work, it needs in input also its logical size) and provides in output the array with the elements in reverse order.

```
#include <iostream>

#define ASIZE 128

void reverseintar(int in[], int size,  int out[]);

int main(){

    int int_a[ASIZE], int_a2[ASIZE];

    int tmp;

    int i = 0, int_a_size = 0, int_a2_size = 0;

    std::cout << "enter a number:" << std::endl;
    std::cin >> tmp;

    while(tmp != 0){
        int_a[i] = tmp;
        i++;

        std::cout << "enter a number:" << std::endl;
        std::cin >> tmp;
    }

    int_a_size = i;

    reverseintar(int_a, int_a_size, int_a2);
    int_a2_size = int_a_size;

    // calling the function

    // only the names of the arrays are passed as arguments
    // this is equivalent to passing the address
    // of the first memory area of the arrays

    // the (logical) size of the input array
    // needs to be passed too, as there is no other way
    // for the function to know it

    // (in this case the logical size of the output array
    // is the same as the input one; when this is not the case
    // the function needs to provide in output this size too)

```

```
45      std::cout << "you entered the following points:" << std::endl;
46
47      for(int i = 0; i < int_a_size; i++){
48          std::cout << int_a[i] << std::endl;
49      }
50
51      std::cout << "that in reverse order are:" << std::endl;
52
53      for(int i = 0; i < int_a2_size; i++){
54          std::cout << int_a2[i] << std::endl;
55      }
56
57      return 0;
58  }
59
60  void reverseintar(int in[], int size, int out[]){
61
62  // the parameters in and out will contain the address of the
63  // first memory areas of the arrays passed as arguments
64  // (the empty brackets denote this)
65
66  // it could also be written as follows:
67  // void reversepointsar(point* in, int size,  point* out)
68  // however this is arguably less readable as it is not distinct
69  // from the case when we want to pass
70  // only one variable by pointer
71
72      int j = 0;
73      for(int i = size - 1; i>= 0; i--){
74          out[j] = in[i];
75
76          // the line above is equivalent to
77          // *(out + j) = *(in + i);
78
79          j++;
80          // we need to keep track of the index for out
81          // (different from the index for in)
82      }
83  }
```

Exercise

Consider the exercise above: "write a program which reads from the user the name of a file, reads a sequence of integers from that file, stores the numbers into an array and prints its content, then reads from the user an integer and prints on the screen whether it is contained in the file or not and at which line".

Write a variation of this program by defining and using a function which takes in input an array of integers and another integer (and any other needed input) and returns the index of the integer in the array if it is found and -1 otherwise.

Exercise

Write a program which reads from the user the name of a file, reads a sequence of integers from that file, stores the numbers into an array and prints its content, then reads from the user an integer and prints on the screen all the elements of the array which are less than the integer entered by the user.

For this purpose define and use a function which takes in input an array of integers and another integer (and any other needed input) and provides in output an array with all the numbers which are less than the integer given in input (the function should also provide in output any other needed output).