Exercise Class 8

Contents

1 Sorting (bubble sort and maximum sort) - *Presentation*

Two presentations showing examples of a bubble sort algorithm and a maximum sort algorithm. The bubble sort slides may be suitable to explain exercise 7.3 c). You can also show the slides of the maximum sort algorithm, but this is optional.

Questions?

2 Multidimensional Arrays/Vectors - *Blackboard OR Programming Environment*

Because pointers are not yet introduced and because the first exercise involving multidimensional arrays is part of this week's exercise sheet, this chapter only introduces some basic functionalities.

If we need a for-loop to read into, copy or output an array, we need two nested for-loops to do the same for a two-dimensional array.

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

for (int i = 0; i < 4; i++)
  for (int j = 0; j < 3; j++)
    std::cin >> array[i][j];
```

For vectors this looks similar.

```
int m = 4;
int n = 3;
std::vector< std::vector<int> > vector (m, std::vector<int>(n));

for (unsigned int i = 0; i < m; i++)
  for (unsigned int j = 0; j < n; j++)
    std::cin >> vector[i][j];
```

It is also possible to assign multidimensional vectors, for example:

```
std::vector< std::vector<int> > imposter = vector;
```

Questions?

3 Representation of a Matrix in an Array - *Presentation*

This presentation shows how a matrix (=2D array) can be stored in an array, i.e., with a single dimension only. You can tell the students that this is how eventually multi-dimensional arrays are actually stored in linear memory, but do not go into anymore details.

Questions?

4 Strings - *Programming Environment (OR Blackboard)*

For reference see the script or sites like http://de.cppreference.com/w/cpp/string/basic_string.

Strings are part of the C++ Standard Library, so they require `#include <string>` to work. Roughly speaking they are special vectors for storing text in the form of characters. They have all the functionality the vector provides, like `.push_back()` and the safer access using `.at()`. However, they also provide us with nice little features, like the `+=` operator being defined in such a way that it appends a string to another one. Also the input and output operators are adapted so that we do not need a for-loop every time we want to read or write more than one character.

```cpp
#include <iostream>
#include <string>

int main () {

  std::string text;
  std::cin >> text;              // reads in a text of arbitrary length, for example "Hello"

  text += " world!";             // appends text to the string, in this case changing it to
                                 // "Hello world!"


  std::string text2 = text;      // initialization also works with a string variable on the
                                 // right hand side, in this case text2 = text

  std::cout << text2 << "\n";     // outputs whole text stored in text2, here "Hello world!"

  return 0;
}
```

Remember that function names are simplified.

Next follow some additional useful functions on strings. Let us look at an example, in which all those functions are used on a sentence.

```
#include <iostream>
#include <string>

int main () {

    std::string str ("The quick brown fox jumps over the lazy dog.");

    std::cout << str.find("fox") << "\n";         // outputs 16
    std::cout << str.find("fox", 30) << "\n";     // outputs std::string::npos
                                                  // (substring not found)
    str.replace(10, 5, "red");
    std::cout << str << "\n";                     // outputs "The quick red fox jumps over the
                                                  // lazy dog."

    str.erase(10,4);
    std::cout << str << "\n";                     // outputs "The quick fox jumps over the lazy
                                                  // dog."

    return 0;
}
```

First, there is `find (const string& string, int pos = 0)`, which gives us the position of where a substring `const string& string` first appears in the string we are accessing with the . operator. The return value is a simple `int`. In the above example, the word `"fox"` starts at position 16 of the string. If no such substring is found, `find` returns the constant std::string::npos.

The additional input argument `int pos`, which has the default value `0`, allows us to start the search for the substring somewhere in the middle of the string. However, beware that, if the substring we are looking for appears before that, it will not be found. That is exactly what happens above, we are not able to find the word `"fox"`, if we search from the 30th character on to the end of the string.

Another useful function is `replace (int pos, int count, const string& string)`, which replaces the range $[pos, pos + count)$ with `const string& string`. In the example above we replaced `"brown"` with `"red"`.

And last there is `erase (int pos, int count)`, which erases the range $[pos, pos + count)$. In the example we erase the chars `"d fo"`.

Questions?

5 Lindenmayer Systems and Turtle graphics - *Presentation*

Two presentations: One for explaining Lindenmayer systems and their implementation, and the other one explaining the usage of the turtle library.

Questions?

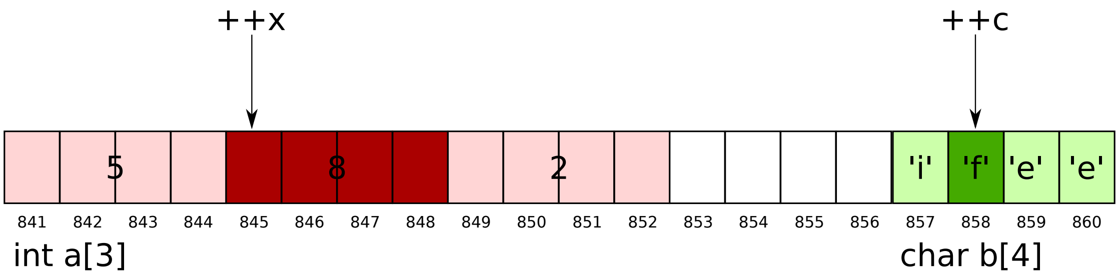
6 Pointers - *Blackboard AND Presentation*

Pointers are very similar to references, but different in the way that we can make them reference another variable, even after the initialization. Remember, once a reference is initialized, it will reference the same variable until the end of its own scope. The assignment operator `=` just changes the value of the variable that is referenced.

```
int i = 5;
int j = 7;

int& k = i;   // k is initialized to reference i
k = j;        // the value of i is changed to 7
```

With pointers we have a different situation. We will be able to change the value of the variable the pointer points to (we call this variable target), and we will be able to make the pointer have another target. To implement this idea we need the concept of addresses. We already learned that the memory in computers is divided into equally sized chunks called bytes (which consist of 8 bits each in modern computers). Each byte has an address number. So our pointer has to know which byte it points to, which is exactly what is saved inside the pointer variable, the address of the current target.

This is illustrated in the first part of the introductory pointer slides.



Questions?

So what does code which manipulates these pointers look like? We need a way to tell the program when to make the pointer point to another target and when to change the value stored in the target. We realize these by introducing two operators, the reference or address operator `&` and the dereference or value operator *. The example below illustrates how they work.

```
int a = 5;
int b = 7;

int* x = 0;               // always initialize empty pointers to zero!

x = &a;                   // the address of a is written to x; x points to a

std::cout << a << "\n";   // outputs 5
std::cout << *x << "\n";  // outputs 5 too

std::cout << x << "\n";   // outputs 0x28fef8 (address of a)
std::cout << &a << "\n";  // outputs 0x28fef8 (address of a) too
```

```
x = &b;                    // x now points to b

*x = 1;                    // changes value of b to 1
```

Questions?

7 Pointers on Arrays

An important part of this section is to give them many examples on how to use pointers; especially easier examples. Also, feel free to add your own examples and exercises if you can think of some that work better for your group. If you see that your students could benefit from a quick recap of the regular pointer theory (i.e. obtaining the address of an lvalue, dereferencing a pointer, ...) feel free to do this.

To be able to apply pointers to arrays we first need to understand array-to-pointer conversion. This allows us to get the address of the first element of an array `int arr[5]` by converting `arr` to an address implicitly.

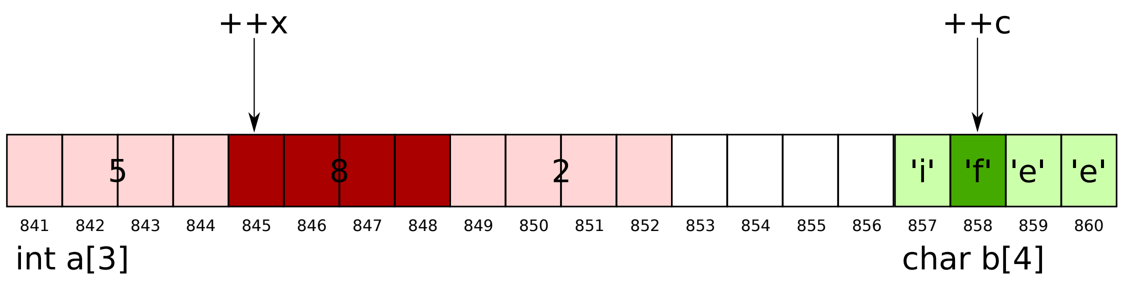
```
int arr[] = {7,1,0,2,5};

int* point = arr;    // arr gets converted to the address of the first array element a[0]

std::cout << *point << "\n";          // outputs 7

std::cout << *(point + 3) << "\n";    // outputs 2
```

One could also write `&arr[0]` to get the same address directly (here no implicit type conversion happens), but most of the time, you will see programmers simply write `arr`.

Another thing we used in the example is pointer arithmetic; by adding `3` to `point` we access the 4th member of the array and are able to output its value with the dereference operator. Pointers and hence addresses behave similarly to `unsigned int` variables, many arithmetic operations such as addition and subtraction are defined for them. However, one has to be aware that `ptr + 1` means different things for different types of targets:

You might have wondered why pointers have a type to begin with and why it is always the same type as the target. That is because the pointer needs to know the length of the target. For an `int` pointer, the length of the target is 4 bytes (32 bit), so if one increments an `int` pointer by one (for example in ++ptr), 4 is added to the address saved in the pointer variable in order to point to the next element in the array. But we can also have a `char` pointer, in which case incrementing the pointer by one (for example in ++ptr) results in a shift of just one byte in order to point to the next element in the array.

To illustrate this you can draw a picture like the following to the blackboard. There, `x` *is a pointer of type* `int*` *initially pointing to* `a[0]`*. And accordingly for* `c` *which is a* `char*`*.*



Questions?

With pointer arithmetic we can also see that `arr[expr]` and `*(arr + expr)` yield the same element and both allow access to a random element of `arr`. Let's compare!

```
int arr[] = {9,2,4,5,1,2,6};

for (int i = 0; i < 7; ++i)
  std::cout << arr[i] << "\n";

for (int* i = arr; i < arr + 7; ++i)
  std::cout << *i << "\n";
```

Notice that `arr + 7` actually points to the first element behind the array. We call a pointer to that element the past-the-end pointer. Accessing this element leads to the same errors we saw in the chapter about arrays (random crap or segmentation fault). But in our for-loop this element is never *accessed*, since the loop condition says `i < arr + 7`.

The past-the-end pointer is important for remembering the end of an array. We can always get the pointer to the first element of an array by converting the array variable to an address, as we did above. However, the program will not remember the end for us, so we have to do it ourselves. You might ask why we save the past-the-end pointer and not the pointer to the last element in the array. The answer is that, if we substract the pointer to the first element from the past-the-end pointer, we get the length of the array.

Questions?

There are slides available (`Pointers_On_Arrays (EN)`) which contain a short animation of some of the most important pointer operations. Show them to your students if you think that they can benefit from these slides. (The only purpose of the code snippet is to show how pointers work with as few distracting parts as possible and in a way as simple as possible. There is no "deeper meaning" behind this code.)