# Exercise Class 9

## Contents



## 1 Pointers: Exercises and Remaining Topics

*You may want to repeat certain aspects of pointers in this class. Make sure to ask the students if have questions.*

### Exercise 1) Finding Mistakes - *Presentation*

### Exercise 2) Reverse Copy - *Presentation*

*This exercise might be a bit hard, especially for beginners. Thus if you have many beginners in your group then you should consider leaving this exercise away completely. But then you should come up with an example of your own for passing a range of an array to a function using pointers.*
*If you cover this in your class, give the students enough time (maybe up to 5 minutes if possible) to stare at the program and think about what it does. Most of the time they will start talking to each other after a while to answer remaining questions and compare their results. That is the time when you can slowly start the discussion using the remaining slides.*

*Questions?*

### 1.1 Call by reference using pointers

Regarding the discussion about calling functions by value or by reference, pointers can be used to call by reference.

In C++ prefer to use references if possible, since they are safer and it is harder to make mistakes due to their simple syntax. However, when handing a certain part of an array, a range indicated by two pointers, to a function, we use pointers.

*Questions?*

## 2 Iterators for Vectors - *Programming Environment*

Vectors have a similar concept as pointers, called iterators. The interaction between vectors and iterators is very similar to the one between arrays and pointers, however, there are some convenient things, we want to have a look at. Since a vector knows its length, it is able to provide an iterator to the first element, and an iterator that points past-the-end. In the case of vectors there is no implicit vector to iterator conversion like there is for arrays. Instead we can use `.begin()` and `.end()` on a vector to get the iterators. These iterators make the `&` operator unnecessary for vectors, this can make the code more readable. For example it now looks like this when we iterate over a vector:

```
#include <vector>

// (for reference: the corresponding pointer example)
// int array[] = {8,3,1,4,6,9};
// for (int* p = array; p != array + 6; ++p)
//  std::cout << *p;

std::vector<int> vec = {8,3,1,4,6,9}; // C++11 syntax!

for (std::vector<int>::iterator it = vec.begin(); it < vec.end(); ++it)
  std::cout << *it;
```

Code using iterators instead of pointers is generally more readable, since the address operator `&` will usually be substituted by things like `.begin()` and `.end()`. However, the value operator * will still appear in your code.

*Questions?*

## 3 Using - *Programming Environment*

When type names become as complicated as `std::vector<int>::iterator`, a nice command called `using` can simplify your code immensely. It gives a new name to an already existing type, which can have several advantages which we will explain below. But first, let us revisit the code example from chapter 2 and add `using`s.

```
using intvec = std::vector<int>;
using intvecit = std::vector<int>::iterator;

intvec vec = {7,3,5,2,7,9};

for (intvecit it = vec.begin(); it < vec.end(); ++it)
  std::cout << *it;
```

Looking at the example, we notice that, once the new names are introduced, the code becomes shorter and more readable.

Generally, `using` is used for two reasons: The first reason, already mentioned above, is that they make code a lot more readable. The second reason is that it makes your code easier to maintain. For example, let's say we want to change the type we do all our calculations with from `float` to `double`, because we need the higher precision. If we remembered to give the type `float` a different name using `using`, we would have to change exactly one line of code. Instead of `using mytype = float;` we would have `using mytype = double;`. However, without `using` we now would have to go through all our code substituting `float` with `double`.

*Questions?*

## 4 const Pointers - *Blackboard*

We introduced pointers as references which can still point to a new target. If we turn the sentence from above around, it says that a reference is a pointer which cannot change its target anymore, in other words, a `const` pointer. But what "kind" of `const` pointer? Since we have two operators `*` and `&` that allow us to change the target or the value of the target, there are two kinds of `const`'s for pointers. One that does not let us change the value of any target, and one that forces us to always point to the same target. For this lecture (and in general), most of the time we are going to see the first kind of `const`, the one that allows us to, for example, iterate over an array, but forbids us to change the values inside.

```
int x = 7;
int y = 5;


const int* i = &x;
++(*i);                  // does not compile, value stored in x cannot
                         // be changed via this pointer access
i = &y;                  // works


int* const j = &x;
++(*j);                  // works
j = &y;                  // does not compile, address stored in j is const


const int* const k = &x; // one can also do both
++(*k);                  // does not compile, value stored in x cannot
                         // be changed via this pointer access
k = &y;                  // does not compile, address stored in k is const
```

A couple of additional points to consider:

- Since iterators are very similar to pointers, they can also be `const` in both the address and the value.
- It is not possible to have a non-`const`-pointer pointing to a `const`-target:

  ```
  const int a = 5;
  int* a_ptr = &a; // Error: must be: const int* a_ptr = a;
  ```

*Questions?*