Learning C++

Florian Weber

May 28, 2015

Contents

0.1 Draftwarning

Warning: This is an early draft. It is known to be incomplete, badly structured and formatted and shouldn't yet be used for learning-purposes.

1 Getting Started

At this point you should have a working compiler and text-editor, so that we can start out with looking into the fundamental building-blocks of the language.

This chapter may not appear to be very entertaining or of much direct use, but it is very important as everything we will encounter here is needed to procede to the more interessting topics that will build on top of it.

1.1 Hello World

```
#include <iostream>

int main() {
        std::cout << "Hello World!\n";
}
```

```
>> Hello World!
```

This program prints the text “Hello World!”, followed by a newline to the standard-output. Let’s look at it line by line:

```
#include <iostream>
```

This line tells the compiler to use a “header” with the filename “iostream” when compiling. A header is basically another C++-file that will be pasted to the place where the include-statement appears.

The “iostream”-header is part of the standard-library that should be shipped with every compiler and has therefore be available on every system. “iostream” contains lots of stuff that are related to reading and writing from and to other ressources like standard-input, standard-output, files and so on.

```
int main() {
```

This defines the main-function. Basically everything from the opening brace will now be executed in the order it appears until we reach the matching closing brace.

```
std::cout << "Hello World!\n";
```

`std::cout` is a construct from the iostream-header that we included and linked to the standard-output. Via it we can write a lot of different things by "pushing them into it" with the output-operator "<<". In this case we just push a so called string-literal into it. A string-literal is a text surrounded by two quotes ('"'); for the case that one wants to include a character that could create ambiguities, there are also some escape-sequences: A "\n" becomes a newline, a "\t" becomes a tab, a "\"" becomes a quote and a "\\" becomes a backslash.

So the string-literal in our example becomes the text "Hello World!" directly followed by a newline.

Now for the last line:

```
}
```

This is just the closing brace of the main-function an the point where the program ends.

1.2 Basic Arithmetic and Variables

Since we now know how to print stuff, we can go on to do some very basic calculations and safe their results.

Say we want to calculate the sum of two numbers and multiply the result with itself. A very simple approach would be this:

```
#include <iostream>

int main() {
        std::cout << (3 + 5) * (3 + 5) << "\n";
}
```

```
>> 64
```

What we can see here is that numbers are directly supported by the language and that we can use the usual notation to do calculations with them. The most basic operations on numbers supported by C++ directly (without help from libraries) are:

`+`	Addition
`-`	Subraction
`*`	Multiplication
`/`	Division
`%`	Modulo

Most of these work just as one would think that they do, the later two will however require some further notes:

- Dividing two integers will result in an integer; this is done via truncation: `3 / 4` will have the result 0, since 0.75 will get rounded down. This is however not the case when calculating with real numbers (called `doubles` in C++): `3.0 / 4` will have 0.75 as result.
- Modulo is only supported by integers, so `7.0 % 4` will *not* work.
- Since division by 0 doesn't make much sense, C++ strictly forbids to do this (same rule applies for modulo). Trying to do this will likely result in a crashing-program and is in fact undefined behavior (more about this and why you should really avoid it, will be explained soon).

Aside from these restrictions the usual rules known from school apply here: Multiplication and division have higher precedence than addition and subtraction and if you want to change this, you have to use parenthesis.

Back to our example: It surely works but it isn't very flexible and changing one value requires changes in two different places which is always a bad thing, since it can easily create errors. Also: There is no need to calculate the sum a second time after we have already done this. Variables solve this problem:

```
#include <iostream>

int main() {
        auto a = 3;
        auto b = 5;
        auto sum = a + b;
        std::cout << sum * sum << "\n";
}


>> 64
```

The line `auto a = 3;` creates a new variable named "a" that holds the value 3. Since "3" is an integer it has the type `int` in C++. The language infers the type of a variable from the value it is initialized with and checks that all uses of all variables are consistent after that. It is important to understand that the type of a variable will never change after it has been created; if you try to assign a value of a different type your code may sometimes compile, but this is because the compiler found a way to convert the argument to the type of your variable.

In all places where you can use a literal number it is also possible to use a variable, for that reason you shouldn't be stingy with variables since they can vastly increase readability;

Let's take a closer look at the type that we are using in this example: `int`. It is the languages default type for integers and can on most modern systems hold numbers between -2147483648 and 2147483647, which should be enough for most applications.

The third statement (`auto sum = a + b;`) demonstrates that variables can also have longer names than just one letter (which they should have almost always) and that we can initialize them from compund expressions like `a + b`.

Finally it should be mentioned that there are other ways to create in variables in C++ too and that encountering them in other peoples code is basically guaranteed, but that this style is usually recommended in the context of modern C++, because it avoids several gotchas in the language, is easy to read and relatively consistent.

1.3 Reading User-input

In order to write programs that are not entirely static, we can read things too:

```
#include <iostream>

int main() {
        auto num = 0;
        std::cout << "Please enter a number:\n";
        std::cin >> num;
        std::cout << "You entered " << num << "\n";
}


>> Please enter a number:
<< 42
>> You entered 42
```

Reading behaves somewhat similar to writing: We have a character-stream of incoming characters (usually from the keyboard) and push those into variables with a stream-operator.

To read more than one value, we can also chain these reads:

```
#include <iostream>

int main() {
        auto i1 = 0;
        auto i2 = 0;
        auto i3 = 0;
        std::cout << "Please enter three numbers:\n";
        std::cin >> i1 >> i2 >> i3;
```

```
        std::cout << "The sum of your numbers is " << i1+i2+i3 << "\n";
}
```

```
>> Please enter three numbers:
<< 23 42 5
>> The sum of your numbers is 70
```

We will see later that we can read a lot of types other than integers, including the not yet covered strings that can store simple text.

1.4 Conditional Execution

Let's write a program that prints the modulus of two numbers:

```
#include <iostream>

int main() {
        std::cout << "Please enter two numbers seperated by whitespace:\n"
        auto num1 = 0;
        auto num2 = 0;
        std::cin >> num1 >> num2;
        std::cout << num1 << " / " << num2
                  << " = " << num1 / num2 << "\n";
}
```

```
>> Please enter two numbers seperated by whitespace:
<< 8 4
>> 8 / 4 = 2
```

At this point we see a great problem: What if the second number that we enter is 0? As already noted we are not allowed to do this calculation (aside from the fact that it doesn't make any sense). The solution is an `if`-statement:

```
#include <iostream>

int main() {
        std::cout << "Please enter two numbers seperated by whitespace:\n"
        auto num1 = 0;
        auto num2 = 0;
        std::cin >> num1 >> num2;
        if (num2 != 0) {
                std::cout << num1 << " / " << num2
```

```
                        << " = " << num1 / num2 << "\n";
        }
}
```

```
>> Please enter two numbers seperated by whitespace:
<< 8 4
>> 8 / 4 = 2
```

The structure of an `if`-statement is very simple: `if`, followed by a boolean expression between parenthesis, followed by a list of statements between braces. A boolean expression is a (small) piece of code that evaluates to a value of the type `bool` (that is either true or false). Often this is achieved with a comparission like the above: `num2 != 0` is the C++-way of asking whether `num2` $\neq 0$. The available comparission-operators are these:

C++	Meaning
`a == b`	$a = b$
`a != b`	$a \neq b$
`a < b`	$a < b$
`a <= b`	$a \leq b$
`a > b`	$a > b$
`a >= b`	$a \geq b$

In addition to those an expression can be prefixed with "`!`", which negates it's value: `!true == false`. To negate bigger expressions just enclose them in parenthesis: `!(true == false)` will be evaluated to `true`.

Another thing that one should know is that integers (and some other types) can be implicitly converted to bool, if they are used as boolean expression; in that case `0` becomes `false` and every other value becomes `true`. There is no final consensus whether one should write `if (i != 0)` instead of `if (i)`, but for the beginning it is certainly a good idea to be explicit here.

Back to the `if`-statements: What if we want to do two different things for each case? For this, there is the so called `else`-statement that can follow an `if`-statement. it is basically the word `else` followed by statements between braces.

```
if (num2 != 0) {
        std::cout << num1 / num2 << std::endl;
} else {
        std::cout << "Error: division by zero!\n";
}
```

This leads us to another problem: What if we have more than two cases? Then we can just use an `else if`:

```
if (num == 0) {
} else if (num < 0) {
        std::cout << "num is negative\n";
} else {
        std::cout << "num is positive\n";
}
```

(Technically the braces around a single conditional statement are not mandatorry; they are however **strongly** recommended since ommiting them can very easily lead to bugs (misstakes), especially in the case where you nest conditionals.

1.5 Undefined Behavior

At this point it is time for the safety-instructions. C++ is a language that was designed to be very fast and portable which sometimes conflicts with ease of use. As a result the C++-standard explicitly does not always require a certain behavior for programs that contains a given construct.

These constructs are almost always very questionable to start with and disallowing them is usually a good thing. Examples include overflowing an `int` (calculating a value that is outside the representable range of int, for example by executing "`2'000'000'000 + 2'000'000'000`"), reading uninitialized variables and accessing unowned memory. Possible behavior ranges from apperently doing what the programer expected, over randomly crashing to severe security-holes. Testing what happens and trusting that everything is fine won't work too, because the next version of your compiler might decide to do something completely different. To illustrate this, let's look at a real-world example:

Postgresql, a very popular database, hat two integers `a` and `b` of type `int` that were both known to be positive. At this point they had to calculate the sum of these two but wanted to detect the case that the sum was outside of the representable range of `int`. They did it somewhat like this:

```
auto sum = a + b;
if (sum <= b) {
        // error-handling
}
```

The assumption that the result will be smaller than `b` if an overflow occurs was funded in the fact that practically every single modern cpu works that way. However: The C++-standard forbidds that kind of code and compilers started to optimize on the assumption that `a + b` would *never* overflow.

As a result of this compilers deduced that adding a positive number to another number would never result in something smaller than the second numbers. Therefore the check whether `sum <= b` would always be false and could be removed.

When the first compiler introduced this behavior the postgres-maintainers protested and refused to fix their code. Instead the used some options that GCC provided to disable this optimisation. When other compilers also added this optimisation, they tried to continue doing similar things for them too, but in the end they had to surrender and fix their code.

The lesson from this is that even if your code seems to work, it might stop doing this tomorow when the next version of your compiler will be released.

tl;dr: Avoid undefined behavior by any means necessary.

1.6 Loops

If we want to check a condition once, we can use `if`. What however if we want to execute something a previously unknown number of times? This is where loops come into play. The simplest form is the so-called `while`-loop. The syntax is basically the same as it is for the the `if`-statement.

The main-difference is that the condition will not be checked once, but again after every execution of the loop-body:

```
#include <iostream>

int main() {
        auto i = 0;

        while (i != 3) {
                std::cout << "i still isn't 3.\n";
                // '++i' is a shorthand-notation for 'i = i + 1'
                ++i;
        }
        std::cout << "i is now 3.\n";
}
```

```
>> i still isn't equal to 3.
>> i still isn't equal to 3.
>> i still isn't equal to 3.
>> i is now 3.
```

The way we used `i` here is quite commont: A counter that counts the number of times that the loop-body was executed and is compared then to the required number of executions.

We call this kind of variables loop-counters and they are conventially often named `i`, `j` and `k` (this is one of the rare cases where single-letter-names are acceptable).

Since the concept of a loop-counter is needed so often, there is also some special syntax to support it: The `for`-loop. It has the following structure:

for (*variable-declaration*; *condition*; *loop-operation*) { *loop-body* }

- *variable-declaration* Is a place to declare variables that will live during the execution of the loop. At this point this will most of the time be the loop-counter.
- *condition* is a conditional statement that works exactly like it does in the while-loop.
- *loop-operation* is a statement that is executed after every execution of the loops body. It should only be used to do things like incrementing the loop-counter.
- *loop-body* is similar to the if's conditional body: It is executed as long as the loops condition is met.

Since this is quite theoretical, let's revisit the example that we used for the `while`-loop:

```
#include <iostream>

int main() {
        // TODO: unsigned
        for(auto i = 0; i != 3; ++i) {
                std::cout << "i still isn't 3.\n";
        }
        std::cout << "i is now 3.\n";
}
```

Here `int i = 3` is the variable declaration. It creates an integer `i` and intializes it with 0. The condition of this loop is that `i` is not equal to 3 which is of course true in the beginning.

1.7 Strings

We already took a very short look at string-literals and used them when printing stuff:

```
#include <iostream>

int main() {
        std::cout << "Hello World!\n";
}
```

Of course it is also possible to safe a string in a variable. In order to achieve that we have to include the `<string>`-header and create a variable of the type `std::string`:

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

int main() {
        auto str = std::string{"some string.\n"};
        std::cout << str;
}
```

```
>> some string.
```

As with integers there are several operations that `std::string` supports: Copying, assigning and comparing all work as one would expect. In addition we can concatenate `std::string`s and string-literals using the `+`-operator:

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

int main() {
        auto str1 = std::string{"foo"};
        auto str2 = std::string{"bar"};
        if (str1 == str2) {
                std::cout << "ERROR: this should never happen!\n";
        }
        auto str3 = std::string{}; // str3 = ""
        str3 = str1 + str2;
        if (str3 == "foobar") {
                std::cout << "Everything is fine!\n";
        }
}
```

```
>> Everything is fine!
```

Conceptually a string is basically just a sequence of characters. In C++ there is a type called `char` that, as one might expect, represents a character. Technically a `char` is an integer with a width of one byte. This allows a range from either -128 to -127 or 0 to 255 (your own implementation will almost certainly use -128 to 127, but keep in mind that this is not everywhere the case). The values between 0 and 127 are the so called ASCII-characters that contain the latin alphabet(a-z in both upper and lower case), arabian numbers (0-9), basic punctuation (point, comma, semi-colon, colon, ...) and some stuff that basically noone uses any more.

To represent further characters like the “Ä”, “Ö”, “Ü” and “ß” several chars have to be combined to a sequence (this is called UTF-8 and you shouldn’t use anything else). The problem with this approach is, that the number of chars and logical characters in strings

differ. There is no reasonable solution to that problem (In case you heard about UTF-32: It isn't either, since there is something called "combining characters").

Let's take a short look at how chars can be used:

```
#include <iostream>

int main() {
        auto c1 = 'A'; // Note the single-quotes!!
        auto c2 = 'B'; // a symbol encoded in them has type char
        if (c1 != c2) {
                std::cout << "Comparission works!\n";
        }

        auto c4 = char{65}; // 'A' has the value 65, so this works, but you
                            // shouldn't really do it
        std::cout << c4 << '\n'; // newline is just one char, so this works
}
```

```
>> Comparission works!
>> A
```

The reason we are looking so deep into characters is that you can access them in a `std::string` with the "[]"-operator:

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

int main() {
        auto str = std::string{"foobar"};
        std::cout << str[0] << '\n';
        str[1] = 'O';
        std::cout << str << '\n';
}
```

```
>> f
>> fOobar
```

To access the n'th Character we write $n-1$ between the square-brackets that we attach to the variable. We will take a look into the reasons for this so-called zero-indexing later, for the meantime it is enough to know that this is how C++ (and most other programming-languages) work and just accept that.

This leaves us with two questions: How do we find out the size of the string and what happens if our indexes refer to nonexisting characters.

The first question is answered by the so called method `size()`. A later chapter will deal with the question what methods are, so for the meantime it is enough to know that if you have a variable `str` of the type `std::string`, you can find out the number of chars in it with the following code: "`str.size()`"

The answer to the second question is more terrifying: If your index is invalid, your program contains undefined behavior and is likely to crash in an uncontrollable way. The one exzeption is the value returned by `size()`: It is guaranteed to return a char with the value zero (the value, not the character '0'), but it must not be written to.

In order to somewhat reduce the dangers of this, there is another method called `at()` that mostly behaves like the square-brackets but is guaranteed to trigger C++'s error-handling-mechanisms (Since we haven't looked into those yet, this would currently mean a guaranteed controlled shut-down instead of undefined behavior).

Let's look at how we can use these things in practice:

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

int main() {
        auto str = std::string{"foo"};
        std::cout << "std.size() = " << str.size() << '\n';
        for (auto i = 0u; i < str.size(); ++i) {
                std::cout << str[i] << str.at(i);
        }
        std::cout << '\n';
}
```

```
>> 3
>> ffoooo
```

The last thing strings for now will be how to read them from standard-input. The obvious way works but has the potential disadvantage, that it reads a word (words are seperated by whitespace in this context). Further words will of course be just ignored in that case:

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

int main() {
        auto str1 = std::string{};
        auto str2 = std::string{};
        std::cin >> str1 >> str2;
        std::cout << str1 << ", " << str2 << '\n';
}
```

```
<< word1 word2 word3
>> word1, word2
```

1.8 Vectors

Strings are sequences of characters, but what if we want a sequence of something else? This is what `std::vector` is designed for. In order to use it we have to include the `<vector>`-header first and then declare what it should contain:

```
#include <vector>
#include <iostream>

int main() {
        // a vector of ints:
        auto  ints = std::vector<int>{0, 1, 2, 3, 4};

        // a vector of strings:
        auto strings = std::vector<std::string>{"Foo", "Bar"};
}
```

Aside from reading, printing and concatenating with `+` all the things that we just learned about strings can also be done with `std::vector`:

```
#include <vector>
#include <iostream>

int main() {
        auto vec1 = std::vector<int>{1, 2, 3};
        std::cout << vec[1] << ", " << vec.at(2) << '\n';
        auto vec2 = std::vector<int>{2, 3};
        if (vec1 != vec2) {
                std::cout << "the vectors contain different elements\n";
        }
}


>> 2, 3
>> the vectors contain different elements
```

As we can see above, initializing a vector with elements is as easy as writing the values in braces and assigning these during construction. Alternativly we can pass it an integer (and optionally a value) in parenthesis in order to create a vector of a certain size with all elements being defaulted to the given value or, if none is given, it's default value (empty string for strings, zero for numbers):

```
#include <vector>
#include <iostream>

int main() {
        // a vector of 1000 integers:
        auto vec1 = std::vector<int>(1000);

        // a vector of 100 strings with value "foo":
        auto vec2 = std::vector<std::string>(100, "foo");
}
```

1.9 Foreach-Loops

We already know the for-loops and we have already seen how we can use them to iterate over all elements in a `std::vector` or `std::string`:

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

int main() {
        auto vec = std::vector<std::string>{"foo", "bar"};
        for (auto i = 0u; i < vec.size(); ++i) {
                std::cout << vec[i] << '\n';
        }
}
```

```
>> foo
>> bar
```

This works, but it is both verbose and not very general: At some point we will come across data-structures that hold sequences, but don't allow access with square-brackets.

To solve these (and some other) problems, C++ has a so called range-based-for-loop, that allows us to say what we really want:

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

int main() {
        auto vec = std::vector<std::string>{"foo", "bar"};
        for (std::string str: vec) { // read: for each str in vec:
```

```
                std::cout << str << '\n';
        }
}
```

```
>> foo
>> bar
```

Nobody will deny that this code is much cleaner and easier to follow, but we will encounter problems if we try to assign a new value to `str`: `str` is a copy of the `std::string` in the vector, so changing it won't change the value in the vector. Another problem is that the copy may be expensive if the string is large. Last but not least it is tedious to repeat the type that is already stated (in the definition of vec). The solution to these problems is to just write "`auto&`" instead of the type. The ampersand (the "&") makes sure that `str` is not a copy of the element in the container, but just an alias (another name) for the element itself.

So our final version looks like this:

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

int main() {
        auto vec = std::vector<std::string>{"foo", "bar"};
        for (auto& str: vec) { // read: for each str in vec:
                std::cout << str << '\n';
        }
}
```

1.10 Summary

In this chapter we took a short look into the very basics of C++. In order to keep the complexity at managable levels we skipped over many details and simplified a few other things.

The reason is that we will need everything we learned here in basically all of the upcomming chapters, that will hopefully provide more of a red thread and make things clearer.

The topics that you should remember from now on, are:

- How to print stuff
- What a variable is
- What integers, strings and vectors are, and how we can use them

- The basic control-structures: `if`, `else`, normal `for` and range-`for`

Before you continue, you should do the training-tasks to get some basic feelings about how to program and how the language behaves. It is impossible to learn programming without writing code yourself.

1.11 Training

- Write a programm that asks the user for their name and prints "`Hello <username>`" after that.
  - Hint: Safe the name in a `std::string`
- Write a programm that will print all integral numbers between 1 and 100.
  - Hint: Use a normal `for`-loop
- Modify the above programm, so that it will print "Fizz" if the number can be cleanly divided by 3, "Buzz" if it can be cleanly divided by 5 and "Fizzbuzz" if it can be cleanly divided by 15. Otherwise keep printing the number itself.
  - Hint: `7 % 3 == 0` will tell you, if seven can be cleanly divided by three.
  - Hint: You will have to put the `if`'s and `else`'s inside the loop.

2 More Basics

In the last chapter we learned about some of the most important basics, that we absolutely need to use C++; many of them apply with little changes to other languages. This chapter now intends to deepen this knowledge and add some information that are more specific to C++ and provide more structure.

Most of what is covered here is the kind of knowledge that we should have heard about at some point, but that we don't need to worry too much about once we pass on.

2.1 Integers

So far we know `int`; it is an integer-type that is usually 4 bytes wide and can therefore hold values between about minus and plus two billion. Historically `int` has been considered a reasonable default-choice for integers. It did however have some problems from day one on:

- While `int` is **usually** four bytes wide, this is totaly plattform-dependent, making it a bad idea to use `int` for truly plattform-indipendent code.
- When 64-Bit computers became popular, compiler-vendors decided agains increasing the size of `int` to eight byte, making it a realistic problem that there may be strings or vectors with elements that **cannot** be adressed with `int` **at all**.
- `int`s can be negative, but can get super-problematic if negative values don't make any sense (indexes into vectors for example).

Since these problems are not recent discoveries, C++ has answers to all of them. Some of those are however imperfect too or have similar problems, which is why we won't cover the topic of integer-types completely for now, but will just take a large enough look at it, to equip ourselves with the subset that is actually usefull.

First we will take a look at integers of different width: Sometimes using 4 bytes for a number would be a complete waste of spaces; sometimes 4 bytes are not nearly enough to store all numbers that we might come across. To accomodate for those needs C++ also has the integer-types `short`, `long` and `long long`; additionally `char` may is used as integer as well. Since their size is however non standardized as well, we won't use those directly. Instead the header `<cstdint>` provides us with aliases that finally give us what we want:

```
#include <cstdint>

int main() {
        auto one_byte_integer   = std::int8_t{};  // value: 0
        auto two_byte_integer   = std::int16_t{23};
        auto four_byte_integer  = std::int32_t{42};
        auto eight_byte_integer = std::int64_t{-5};
}
```

Note that we have to state the type at the right side of the initializations, otherwise the default (`int`) would be used which is not what we want here.

This solves problem number one. The second problem is that we all of those integer-types can store negative values, which sometimes wouldn't make any sense. If we want to know for sure that this is not the case, we can use the unsigned-family: All the core-language-types can be prefixed with `unsigned` to annotate that they won't store negative numbers. Instead they may store somewhat bigger positive numbers (the maximum of the signed version times two plus one). Be aware that this brings some problems with it: If you subtract one from zero, the the numbers will wrap around and result in the biggest possible value of that type (unlike for signed integers, this is guaranteed by the standard instead of undefined behavior).

Let's take a look at code:

```
int main() {
        // unsigned without further type means unsigned int:
        auto u1 = unsigned{}; // value: 0

        // alternatively we may add a u after a numer
        // to make it to an unsigned int:
        auto u2 = 42u;

        // if we want to state the complete type we have
        // to use parenthesis because it consists of two
        // words:
        auto u3 = (unsigned int){}; // 0

        // Just for demonstration how you can use the
        // other unsigned integers:
        auto u2 = (unsigned short){23};
        auto u3 = (unsigned long){1234567890};
}
```

Obviously the parenthesis are ugly and we are back at the problem of the unspecified sizes. The answer is the same that we already got for the signed integers: "Use the `<cstdint>`-header!"

The unsigned integers follow the same naming-scheme that signed ones do, except for an added 'u' in front of the name:

```
#include <cstdint>

int main() {
        auto one_byte_integer   = std::uint8_t{};  // value: 0
        auto two_byte_integer   = std::uint16_t{23};
        auto four_byte_integer  = std::uint32_t{42};
        auto eight_byte_integer = std::uint64_t{5};
}
```

It should be mentioned for completeness, that there is also a `signed`-keyword which may be used instead of the `unsigned`-keyword and has the obvious meaning. Since most integer types are however guaranteed to be signed from the start, it doesn't have any effect at all most of the time. The one exception to that rule is `char`:

Unlike all other integer-types, `char` and `signed char` are not different names for the same type, but different types. In fact it is not guaranteed that `char` is signed (it will most likely be on your plattform, but there are definitely others for which this is not the case). Basically this is just another reason to use `std::int8_t` and `std::uint8_t` if you need integers that are one byte wide and only use `char` if you want a character.

The final integer-types that we need for now are those to work with containers: `std::size_t` and `std::ptrdiff_t`, where the first one is what we need most often.

`std::size_t` is an unsigned integer that is guaranteed to be able to index any value in an array and is *not* necessarily the same as `unsigned int`. In fact, on most modern system it won't be.

`std::ptrdiff_t` is basically the signed version of `std::size_t` it's mostly needed to represent the distance between two values in containers (distances can be negative if the first element comes after the second). Aside from that there aren't a lot of situations where we would reach for it.

Now, for one of those parts where c++ really shows it's age and it becomes obvious why many consider it to be a complicated language:

What is the result-type if an operation involves two integers of types I_1 and I_2? The sad answer is that this is ridicoulusly complicated, hard to predict and strongly depends on the plattform. The best thing here really is to make no assumptions and be very explicit for any mixed types and hope the best for expressions where $I_1 = I_2$.

Two provide two examples of just how retarded the situation is:

1. The common type of `unsigned int` and `signed long` may well be `unsigned int` (though admittedly only on plattforms where the size of `int` and `long` are the same).

2. The common type of `std::uint8_t` and `std::int8_t` is `std::int32_t` on most plattforms (that is because all types that are smaller than `int` are promoted to `int` before there values are used and `int` is often 32 bits wide.

One very important implication of this mess is that you should never compare signed and unsigned integers:

```
if (-1 > 1u) {std::cout << "-1 is greater than 1\n";}
```

This print statement will be executed, because it will be promoted to `unsigned int` where it will represent the largest possible value. Again: **Never** compare signed and unsigned values directly.

Now that we swallowed this, for the good news: While there are still a handfull of other places where C++ behaves very strange, we won't see anything that is really worse than integer-conversions, so basically it can only get better from here on.

To sum this section up, let's sum up what integers we actually need, and when we need them:

Type	When to use
`std::size_t`	As index for containers like `std::vector` (`vec[i]`)
`std::ptr_diff_t`	To safe the distance between two elements in a container
`int`	To safe a signed number of average size that is not an index
`unsigned`	To safe a non-negative number of average size that is not an index
`std::uintXX_t` `std::intXX_t`	Mostly needed when a certain size is strictly required or for optimisations; this is a surprisingly rare thing

2.2 Floatingpoint Numbers

Since we have now seen C++'s integers and learned a bit about why they are weird, we can now take a relaxing look at the way easier floats.

Or we could if they were easy, but sadly they too are surprisingly complex. In this case it isn't C++'s fault however: The problems we will talk about now are basically inherent properties of floating-point precission, the standard that describes what most implementations do and the sometimes weird behavior of x86/x64-CPUs.

Let's start with the easy part: C++ offers three floatingpoint-types:

- `float`
- `double`
- `long double`

The default one is `double` and unless you have good reasons to do something else, you should probably use that.

Now, what is a floating-point number? Basically it is similar to what is also known as scientific notation for numbers: Instead of "123.45" we can also write "$1.2345 \cdot 10^3$". The advantages of that approach is that we can easily keep the relative differences between numbers small no matter whether they are very big or very small. While the absolute difference between $1.23 \cdot 10^{-23}$ and $3.5 \cdot 12^{-23}$ is much smaller than the difference between 10000000000 and 10000001234, we will probably care much more about it, because the second number is actually twice as big as opposed to a difference of less than one part in a million.

Furthermore it gets much easier to save very large and very small numbers that way: We can easily write down a number with a hundred digits or a number with a hundred zeros between the decimal-point and the first non-zero number: 10^{100} and 10^{-100}.

Floating point numbers as the ones used in C++ basically do it like that, except that they don't work by saving decimal but binary numbers. We don't have to know about the details here though, since at the end of the day most things just work.

Except when they don't: The problems that we face here are with the finite precission of computers and the fact that they have to do rounding. Furthermore some numbers cannot be represented exactly, so for instance the check whether `0.1 * 10.0 == 1.0` may return false in C++.

It is important to note that this is not a problem of C++, but of basically every programming-language that supports some kind of non-integer-numbers.

Now, how do we tackle those problems? Well, the usual workaround is to never compare such numbers directly but always to use an „epsilon" which is a small number that we will accept as error. An example probably shows it best:

```
#include <iostream>

int main() {
        auto size_1 = double{}:
        auto size_2 = double{};
        std::cout << "Please enter two sizes: ";
        std::cin >> size_1 >> size_2;
        if (size_1 + 0.00001 < size_2) {
                std::cout << "The first size is smaller.\n";
        } else if (size_2 + 0.00001 < size_1) {
                std::cout << "The first size is larger.\n";
        } else {
                std::cout << "Both sizes are about the same\n";
        }
}
```

As a general rule of thumb: If you can reasonably avoid floating-point, avoid it. This doesn't say that you should never use it (in fact you should sometimes), but that you will probably have an easier time with integers.

2.3 `auto`

So far we have created all our variables like this:

```
auto var = some_value;
```

This style is also known as „Almost Allways Auto“ and is recommended by Herb Sutter (a very famous C++-guru and head of the standards-committee).

Technically it says „Create a variable named var with the value '`some_value`' and the type of '`some_value`'“. Since another style is also **very** common and in a small number of cases still needed it should be mentioned here too:

```
double var1 = 0;
int var2 = 23;
std::string foo; // We don't have to explictly assign a value

double var2; // For some type with disatrous consequences. DON'T DO THIS!
```

Aside from loosing the advantage of being able to find the definiton by looking for `auto name`, we also introduce a class of bugs that way: For some types (most notably all build-in ones, that is all integers, floats, ...) the state of the variable is not defined after creating it like that and reading it is undefined behavior.

The one exception where we will use this style is when we are forced to do so because the type disallows the creation with auto. A notable example for this is `std::random_device` which we will encounter when learning about randomness and on older plattforms that don't yet suppport C++11 completely the filestreams.

Whether or not you follow the AAA-style is your own decission in the end, but however you choose, be *consistent*. For the start it is certainly not a bad approach of using it to see whether it works for you.

2.4 Training

- You are giving classes to pupils and students. For their grading you want to to save the points that they scored on their training-excercises. What would you use as datatype for those points? You don't give anyone half points.
- You want to calculate the mean points of all your students. What should you use as datatype? What for the median?

- You are a German upper school student and want to save all your grades (those are between 0 and 15). Since you get tons of them, you don't want to use a lot of memory. What will you use as datatype?

Solutions:

- Since there are no negative points you should pick an unsigned integer. At this point there are several ways to argue for several among them. A good first approach would be to just take `unsigned` since we don't expect the number to get astronomically large and the default-width should be enough for this case.

Alternatively the same points could be made in favour of `std::uint16_t` and `std::uint32_t`. A good rule of thumb is however to pick the default until it turns out to be problematic. * Since the mean value is general not representable using integers ($\frac{1+2}{2} = 0.666\ldots$), we need floating points. In that case we just pick a `double` and are happy. The situation for the median is different however: Since it is an actual value from the set, it should have the same integer-type as you picked for your points.

Alternatively you might argue for the median to use `double` too, since it might be the mean of two values if your set has even size and means should be represented as `double`s. This is fine too. * Since the numbers between 0 and 15 are never negative we again pick an unsigned integer type. In this case we do however know a clear upper bound and see that it fits easily into a `std::uint8_t`, which is therefore what we should pick.

3 Functions

A function is a construct available in virtually all programming languages. By the simplest definition, functions are reusable snippets of code. Reducing repetition is the main purpose of functions; doing so makes the code better understandable and reduces the chance of errors.

It's perhaps best to start describing functions by a simple example: Let's say we have a vector of ints and we want to find the biggest element. The code to accomplish this is pretty simple:

```
#include <vector>
#include <iostream>

int main() {
        std::vector<int> vec {5, -3, 2, 7, 11};

        auto smallest_element = vec[0];
        for (auto x: vec) {
                if (x<smallest_element) {
                        smallest_element = x;
                }
        }
        std::cout << "smallest element is " << smallest_element << '\n';
}
```

```
>> smallest element of vec is -3
```

This solution works, but the problem is, that every time we want to do this again, we have to write those lines again, which is unpleasant at best but most likely also error-prone. Functions give us the flexibility to avoid rewriting this:

```
#include <vector>
#include <iostream>

int smallest_element(std::vector<int> vec) {
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
```

```
                        smallest_value = x;
                }
        }
        return smallest_value;
}

int main() {
        std::vector<int> vec1 {5, -3, 2, 7, 11};
        std::vector<int> vec2 {0, 1, 2, 3};

        std::cout << "smallest element of vec1 is "
                  << smallest_element(vec1) << '\n'
                  << "smallest element of vec2 is "
                  << smallest_element(vec2) << '\n';
}

>> smallest element of vec1 is -3
>> smallest element of vec2 is 0
```

The amount of saved typing is obvious. We also have the advantage that we are now able to improve the algorithm in one place so that the improvements are right away in the whole program.

3.1 The signature of a function

Now: How does this work? Let's look at the first line of the function:

```
int smallest_element(std::vector<int> vec)
```

This is called the functions signature. It consists of three parts: The name of the function (`smallest_element`), it's returntype (`int`) and a comma-separated list of it's arguments.

The name is probably the easiest part to understand: It identifies the function. The same restrictions that exist for variable names (may not start with a digit, may not contain whitespace or special characters other than underscore, ...) apply also for names of functions.

The returntype is the type of the thing, that a function returns. We gave it a vector of ints and request the smallest element, so the returntype is of course int. If there is nothing to return, the returntype is `void`.

The arguments are the data on which the function should operate. Since we want to inspect a vector of ints it has to get into the function somehow, so we pass it as argument. The list may be empty, in which case we just write `()`; otherwise we write the type of the argument, followed by the name with which we refer to it in the function. If their are further arguments they have to be written in the same way, divided by commas.

3.2 The body of a function

Since we should now have a basic idea of how the signature works we can examine the rest of it, the so called body:

```
{
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
                        smallest_value = x;
                }
        }
        return smallest_value;
}
```

The first thing that should be noted are the braces that start and end the function. They also create a new scope so that every variable in it only exists in the function.

The first five lines inside that scope are pretty normal C++ without any real surprises.

The last line however contains a so called return-statement. It consists of the word `return` followed by whitespace, followed by a statement that has a type, followed by a semicolon. „statement followed by a type“ mainly refers to either a literal, a variable, a call to a function that returns a value or some term that involves operators (which can be seen as functions). So all of these are valid return-statements:

```
// literal 1:
return 1;

// variable:
int returnvalue = 2;
return returnvalue;

// the result of some calculations involving operators:
return 2*3 + 4*5;

// the returnvalue of some function:
return some_function(1, 3);
```

It is important that the type of the used expression is either identical to the returntype of the function or can be trivially converted to it:

```
double fun_1() {
        return 1.0;
        // fine: 1.0 is double, as is the returntype of
        // the function.
```

```
}

double fun_2() {
        return 1;
        // fine too: while 1 has type int, it can be trivially
        // converted to double.
}

double fun_3() {
        return "some string";
        // error: a string-literal cannot be trivially converted
        // to double.
}
```

3.3 Calling a Function

Calling a function is probably the easiest part in this chapter: Just write the name of the function followed by parenthesis that contain the arguments. Note that the arguments are copied into the function, so any changes that are made to them there wont change the value of the argument at the callside.

```
#include <iostream>

void print_string(std::string str) {
        std::cout << str << std::endl;
}

void print_ints(int i1, int i2) {
        std::cout << i1 << ", " << i2 << std::endl;
}

void print_hello_world() {
        std::cout << "Hello World!" << std::endl;
}

int main() {
        print_string("some string");
        print_ints(1, 3);
        print_hello_world();
}
```

```
>> some string
>> 1, 3
>> Hello World!
```

If we are interested in the returnvalue, we can just use the call as if it would be a value:

```
#include <iostream>

int add(int i1, int i2) {
        return i1 + i2;
}

// implemented according to http://xkcd.com/221/
int get_random_number() {
        //chosen by fair dice-roll:
        return 4;
}

int main() {
        int a = add(1,2);
        std::cout << "The value of a is " << a << ".\n"
                  << "A truly random number is: "
                  << get_random_number() << '\n';
}
```

```
>> The value of a is 3.
>> A truly random number is: 4.
```

3.4 Some examples

```
#include <iostream>

// a function that takes no arguments and returns
// an int:
int function_1() {
        return 1;
}

// a function that prints an int that is passed to it and
// returns nothing:
void print_int(int n) {
        std::cout << n << std::endl;
```

```
}

// a function that returns nothing, takes no arguments and
// does nothing:
void do_nothing() {}
```

3.5 Function Overloading

TODO: rewrite this text

After having learned about both `const` and references, we now know enough about passing arguments into functions to get pretty far. While there are some (relatively strange) other ways of doing that, the importance for most programmers to know about them is very small, as they are mainly relevant to those people who implement C++ itself. Since we are currently far away from doing this, we'll move that topic to the distant future and instead take a look at something that is useful for everyone instead: Overloading functions.

Careful readers may have noticed in the introduction to functions, that the naming-requirements did not explicitly include, that the name has to be unique. That is because it isn't. Functions are identified not only by their names, but also by their arguments. If the arguments of two functions differ in number or type it is valid for them to share a name.

```
#include <iostream>

void function(int n) {
        std::cout << "function(int " << n << ");\n";
}

void function(double d) {
        std::cout << "function(double " << d << ");\n";
}

int main() {
        function(3);
        function(2.718);
}

>> function(int 3);
>> function(double 2.718);
```

In order to get used to this technique, we'll revisit our old companion `smallest_element()`. While the current version is already pretty good, we might be interested in a related

but not identical function for strings: Find the smallest character. In order to keep this somewhat interesting, we'll define that a lowercase character is always smaller than an uppercase one and characters that come earlier in the alphabet are smaller than those that come later.

To keep this task manageable, we'll restrict ourselves to the characters of the English alphabet and ignore all other ones. Let's take a look at the code:

```
#include <iostream>
#include <string>
#include <vector>
#include <locale> // required for isupper and islower

// the old version for vectors of ints
int smallest_element(const std::vector<int>& vec) {
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
                        smallest_value = x;
                }
        }
        return smallest_value;
}

// the new version for strings
// return 0 if no character is found
char smallest_element(const std::string& str) {
        std::locale l{}; // required for isupper and islower
        char smallest_char = 0;
        bool result_is_lowercase = false;
        for (char c: str) {
                if (std::islower(c, l)) {
                        if(smallest_char == 0 || !result_is_lowercase || c < smallest_char) {
                                smallest_char = c;
                                result_is_lowercase = true;
                        }
                }
                else if (!result_is_lowercase && std::isupper(c, l)) {
                        if(smallest_char == 0 or c < smallest_char) {
                                smallest_char = c;
                        }
                }
        }
        return smallest_char;
}
```

```
int main() {
        std::cout << "the smallest character of 'Foobar' is '"
                  << smallest_element("Foobar") << ''\n'
                  << "the smallest number of 1, 3, 6, -3, 4 and 2 is: "
                  << smallest_element(std::vector<int>{1, 3, 6, -3, 4, 2})
                  << '\n';
}
```

```
>> the smallest character of 'Foobar' is 'a'
>> the smallest number of 1, 3, 6, -3, 4 and 2 is: -3
```

While the new code may not be very beautiful, it shows that there is no problem with creating two different functions that share a name and an abstract behavior (finding the smallest element in some kind of list) but differ in implementation.

4 References

Functions are a great thing, but if we look closer there are problems with how the arguments are passed. Let's look at our `smallest_element`-function again:

```
#include <vector>
#include <iostream>

int smallest_element(std::vector<int> vec)
{
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
                        smallest_value = x;
                }
        }
        return smallest_value;
}

int main()
{
        std::vector<int> vec;
        for(size_t i=0; i < 10000000; ++i) {
                vec.push_back(i);
        }

        std::cout << "smallest element of vec is "
                  << smallest_element(vec) << '\n';
}
```

```
>> smallest element of vec is 0
```

The problem that we face here is somewhat non-obvious: Remember that the arguments to a function are copied. While this is no problem for a single integer, we do not want to copy a big vector if we can avoid it. If we examine the code we see, that there wouldn't be any problem if `smallest_element` operated on the original vector instead of a copy.

This is where references come into play.

A reference is an alias for another variable. It must therefore be initialized with one the moment it is constructed. It is not possible to make it alias another variable after that. The easiest way to understand them is probably some code:

```
#include <iostream>

int main()
{
        int x = 0; // a normal integer

        int& ref = x;
        // a reference to x. Note that the type is
        // written almost identical with the exception
        // of the '&' that makes ref a reference.

        std::cout << "x=" << x << ", ref=" << ref << '\n';

        x = 3;
        std::cout << "x=" << x << ", ref=" << ref << '\n';

        ref = 4;
        std::cout << "x=" << x << ", ref=" << ref << '\n';

        int y = ref;
        std::cout << "x=" << x << ", y=" << y
                  << ", ref=" << ref << '\n';

        y = 1;
        std::cout << "x=" << x << ", y=" << y
                  << ", ref=" << ref << '\n';

        ref = y;
        std::cout << "x=" << x << ", y=" << y
                  << ", ref=" << ref << '\n';

        y = 0;
        std::cout << "x=" << x << ", y=" << y
                  << ", ref=" << ref << '\n';

}
```

```
>> x=0, ref=0
>> x=3, ref=3
>> x=4, ref=4
```

```
>> x=4, y=4, ref=4
>> x=4, y=1, ref=4
>> x=1, y=1, ref=1
>> x=1, y=0, ref=1
```

A reference to a certain type is itself a type. If the referenced type is `T`, then a reference to it is written as `T&`. Note that it is impossible to create a reference to a reference; while the syntax `T&&` exists and is related to references, it does something completely different and should not be covered at this point.

Now that we know references we can go back to our initial problem: Passing a big vector into a function. The solution is now very straightforward:

```
#include <vector>
#include <iostream>

// pass a reference instead of a value:
//                                    ↓
int smallest_element(std::vector<int>& vec)
{
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
                        smallest_value = x;
                }
        }
        return smallest_value;
}

int main()
{
        std::vector<int> vec;
        for(size_t i=0; i < 10000000; ++i) {
                vec.push_back(i);
        }
        // vec is automatically passed as reference:
        std::cout << "smallest element of vec is "
                  << smallest_element(vec) << '\n';
}
```

```
>> smallest element of vec is 0
```

We see that it doesn't make any difference from the callside, whether a function copies it's arguments (this is called 'pass by value') or just uses the original ('pass by reference').

Aside from the fact that passing by reference is often faster than passing by value, it allows us to change the value of the original too:

```
#include <iostream>

void increase(int& n)
{
        n += 10;
}

int main()
{
        int x = 0;
        std::cout << "the value of x is " << x << '\n';
        increase(x);
        std::cout << "the value of x is " << x << '\n';
}
```

```
>> the value of x is 0
>> the value of x is 10
```

Note however, that while this is possible, it is often a bad idea, since it makes reasoning about where a variable is changed much harder. On the other hand there are situations where this really is the best alternative. As a general advice: If you are unsure, don't do it.

5 Const

As we have seen in the last chapter, there are mainly two reasons to pass an argument to a function by reference: It may be faster and we are able to change the original value.

We also saw that it is often unnecessary inside the function to be able to change the value of the argument and noted that it is often a bad idea because it makes reasoning about the code harder. The problem boils down to the fact, that we are currently unable to see from the signature of a function whether it will change the value of it's arguments. The solution to this problem is called `const`.

5.1 Immutable values

`const` behaves somewhat similar to references: It is an annotation to an arbitrary type that ensures, that it will not be changed. Let's start by looking at const variables, also called constants:

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

int main()
{
        const int zero = 0;
        const int one = 1;
        const std::string str = "some const string";

        // reading and printing constants is perfectly fine:
        std::cout << "zero=" << zero << ", one=" << one
                  << ", str='" << str << '\n';

        // even operations that do not change the values are ok:
        std::cout << "the third letter in str is '"
                  << str[2] << '\n';

        // doing calculations is no problem:
        std::cout << "one + one + zero = " << one + one + zero
                  << '\n';
```

```
        // trying to change the value results in a compiler-error:
        //zero = 2;
        //one += 1;
}
```

```
>> zero=0, one=1, str='some const string'
>> the third letter in str is 'm'
>> one + one + zero = 2
```

Aside from the possibility that the purpose of restricting what can be done with variables may be unclear at this point, it is probably relatively easy to understand what the above code does and how `const` works so far.

So, why should we use constants instead of variables and literals? The answer has to be split into two parts, concerning both alternatives:

A constant may be more suitable than a variable if the value will never change, because it may both enable the compiler to produce better code (knowing that a certain multiplication is always by two instead of an arbitrary value will almost certainly result in faster code) and programmers to understand it faster as they don't have to watch for possible changes.

On the other hand constants are almost always better then literal constants. Consider the following examples:

```
#include <iostream>

int main()
{
        for(double m = 0.0; m <= 2.0; m+=0.5) {
                std::cout << m << "kg create " << m * 9.81
                          << " newton of force.\n";
        }
}
```

```
>> 0kg create 0 newton of force.
>> 0.5kg create 4.905 newton of force.
>> 1kg create 9.81 newton of force.
>> 1.5kg create 14.715 newton of force.
>> 2kg create 19.62 newton of force.
```

```
#include <iostream>

//gravitational_acceleration of earth
const double GRAVITATIONAL_ACCELERATION = 9.81;
```

```
int main()
{
        for(double m = 0.0; m <= 2.0; m+=0.5) {
                std::cout << m << "kg create "
                          << m * GRAVITATIONAL_ACCELERATION
                          << " newton of force.\n";
        }
}
```

```
>> 0kg create 0 newton of force.
>> 0.5kg create 4.905 newton of force.
>> 1kg create 9.81 newton of force.
>> 1.5kg create 14.715 newton of force.
>> 2kg create 19.62 newton of force.
```

Even this pretty small example gets easier to understand, once we give names to constant values. It should also be obvious that the advantage in readability increases even further if we need the value multiple times. In this case there is even another advantage: Should we be interested to change the value (for example because we want to be more precise about it), we just have to change one line in the whole program.

5.2 Constant References

At this point we understand how constant values work. The next step are constant references. We recall that a reference is an alias for a variable. If we add constness to it, we annotate, that the aliased variable may not be changed through this handle:

```
#include <iostream>

int main()
{
        int x = 0;
        const int y = 1;

        int& z = x;

        const int& cref1 = x;
        const int& cref2 = y;
        const int& cref3 = z;

        // int& illegal_ref1 = y; // error
```

```
        // int& illegal_ref3 = cref1; // error

        std::cout << "x=" << x << ", y=" << y << ", z=" << z
                << ", cref1=" << cref1 << ", cref2=" << cref2
                << ", cref3=" << cref3 << '\n';

        x = 10;

        std::cout << "x=" << x << ", y=" << y << ", z=" << z
                << ", cref1=" << cref1 << ", cref2=" << cref2
                << ", cref3=" << cref3 << '\n';

        // ++ref1 // error
        // ++ref2 // error
}

>> x=0, y=1, z=0, cref1=0, cref2=1, cref3=0
>> x=10, y=1, z=10, cref1=10, cref2=1, cref3=10
```

We note several things:

- It is allowed to create const references to non-const values, but we may not change them through this reference.
- References may be constructed from other references.
- We may add constness when we create a reference, but we may not remove it.

5.3 Functions and Constants

With this knowledge it is pretty easy to solve our initial problem of passing arguments to functions by reference: We just pass them by `const` reference which unites the performance-advantage with the ease of reasoning about possible changes to variables.

```
#include <iostream>
#include <vector>

//pass by const-reference
int smallest_element(const std::vector<int>& vec)
{
        auto smallest_value = vec[0];
        for (auto x: vec) {
                if (x<smallest_value) {
                        smallest_value = x;
                }
```

```
        }
        return smallest_value;
}

int main()
{
        std::vector<int> vec;
        for(size_t i=0; i < 10000000; ++i) {
                vec.push_back(i);
        }
        // getting a const reference to any variable is trivial,
        // therefore it is done implicitly:
        std::cout << "smallest element of vec is "
                  << smallest_element(vec) << '\n';
}


>> smallest element of vec is 0
```

This leaves us with the question of how to pass arguments into a function. While they may not be entirely perfect, the following two rules should apply in most cases:

- If you just need to look at the argument: Pass by const reference.
- If you need to make a copy anyways, pass by value and work on the argument.

The rationale for this rule is simple: Big copies are very expensive, so you should avoid them. But if you need to make one anyways, passing by value enables the language to create much faster code if the argument is just a temporary value like in the following code:

```
#include <iostream>
#include <locale> // for toupper()
#include <string>

std::string get_some_string()
{
        return "some very long string";
}

std::string make_loud(std::string str)
{
        for(char& c: str){
                // toupper converts every character to it's equivalent
                // uppercase-character
                c = std::toupper(c, std::locale{});
```

```
        }
        return str;
}

int main()
{
        std::cout << make_loud(get_some_string()) << '\n';
}
```

```
>> SOME VERY LONG STRING
```

Let's ignore the details of the function `toupper()` for a moment and look at the other parts of `make_loud`. It is pretty obvious that we need to create a complete copy of the argument if we don't want to change the original (often the only reasonable thing). On the other hand: In this special instance changing the original would not be a problem, since it is only a temporary value. The great thing at this point is, that our compiler knows this and will in fact not create a copy for this but just "move" the string in and tell the function: "This is as good as a copy; change it as you want."

6 Function Templates

Sometimes we have several almost identical functions, the only difference being that they operate on different types. Function templates are a feature of the C++ language that allows to have a single implementation that works for multiple types instead of duplicating the code. During compilation the compiler will duplicate the code for us as many times as needed.

6.1 Basic principles

Function templates are defined by adding `template<`*type list*`>` before the declaration of the function. For example,

```
template<class Type>
void foo(int x)
{
    /* put code here */
}
```

Now we can use `Type` within the function body as any other type such as `char` or `double`. The template parameter list may contain multiple parameters. Each of them must be prepended with either of `class` or `typename` keywords.

The above function can be called as e.g. `foo<char>(2)`. Each time the function is called, the compiler "pastes" `char` into each location where `Type` is used and checks whether the resulting code is valid. If it's not valid, an error is raised. Otherwise, the function behaves in the same way as if `Type` was `char` or any other given type. See the example below:

```
#include <iostream>
#include <iomanip>

// Converts integer to different types and prints it
template<class Type>
void foo(int x)
{
    std::cout << Type(x) << "\n";
}
```

```
int main()
{
    std::cout << std::fixed << std::setprecision(3); // setup formatting
    foo<int>(67);    // print 67 as an int
    foo<double>(67); // print 67 as double
    foo<char>(67);   // print 67 as a character
}

>> 67
>> 67.000
>> C
```

6.2 Deduction

Template parameters can be used anywhere in the function, including the parameter list. For example, `template<class T> void bar(T a, T b) { ... }`. If all template parameters appear in the function parameter list, then the compiler can deduce the actual type of the parameter automatically, so the function template can be called in the same way as any other function, e.g. `bar(2, 3)`. See the example below:

```
#include <iostream>
#include <iomanip>

// Prints the given type
template<class T>
void print(T x)
{
    std::cout << x << "\n";
}

int main()
{
    std::cout << std::fixed << std::setprecision(3); // setup formatting
    print(64);          // prints 64 as an int
    print(64.2);        // prints 64.2 as a double
    print(double(64)); // prints 64 as a double
    print<char>(64);   // override the automatic deduction -- force T to be char
    print('c');         // print 'c' as a char
    print("bar");       // prints "bar" as const char*
}

>> 64
```

```
>> 64.200
>> 64.000
>> D
>> c
>> bar
```

In general one should never pass template-arguments that can be inferred, since the compiler knows better anyways and the function-template may do unexpected things.

6.3 Non-type parameters

7 Structs and Classes

Let's assume we want to calculate the distance between two points in space; the formula for this is quite simple: Sum the squares of the distances in every dimension and take the square-root:

$$\text{distance} = \sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2 + |z_1 - z_2|^2}$$

Since this is somewhat heavy to write every time, we'll use a function for that:

```
#include <iostream>
#include <cmath> // needed for sqrt(), abs() and pow()

double squared_distance(double p1, double p2) {
        return std::pow(std::abs(p1 - p2), 2);
}

double distance(double x1, double y1, double z1, double x2,
                double y2, double z2) {
        auto x = squared_distance(x1, x2);
        auto y = squared_distance(y1, y2);
        auto z = squared_distance(z1, z2);
        return std::sqrt(x + y + z);
}

int main() {
        std::cout << "The points (0,1,2) and (4,1,0) have the distance "
                  << distance(0,1,2,4,1,0) << '\n';
}

>> The points (0,1,2) and (4,1,0) have the distance 4.47214
```

The solution is working, but if we are honest, it isn't really nice: Passing the points into the function by throwing in six arguments is not only ugly, but also error-prone. Luckily C++ has solutions for this: Structs and classes. The biggest difference between these two is conventional, not technical, so we can look into them together.

A struct is basically a collection of values. In our example a point is represented by three doubles which even got implicit names: `x`, `y`, and `z`. So let's create a new type that is exactly that:

```
#include <iostream>
#include <cmath>

struct point {
        double x;
        double y;
        double z;
};

double squared_distance(double p1, double p2) {
        return std::pow(std::abs(p1 - p2), 2);
}

double distance(const point& p1, const point& p2) {
        auto x = squared_distance(p1.x, p2.x);
        auto y = squared_distance(p1.y, p2.y);
        auto z = squared_distance(p1.z, p2.z);
        return std::sqrt(x + y + z);
}

int main() {
        std::cout << "The points (0,1,2) and (4,1,0) have the distance "
                << distance(point{0,1,2}, point{4,1,0}) << '\n';
}
```

```
>> The points (0,1,2) and (4,1,0) have the distance 4.47214
```

Reducing six arguments to two, which in addition share semantics is clearly an improvement. It is obvious that the code got way cleaner.

7.1 Construction

Above we created our points by writing `point{0,1,2}`. This worked because point is an extremely simple structure. In general (we'll discuss the exact circumstances later) we need to implement the initialization ourself though.

Considering our current struct: Leaving variables uninitialized is evil and there is no exception for variables in structs and later on classes. So let's make sure, that they are zero, unless explicitly changed:

```
#include <iostream>

struct point {
```

```
        // this makes sure that x, y and z get zero-initialized
        // at the construction of a new point:
        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
};

int main() {
        // no longer possible:
        // auto p = point{1,2,3};

        // this has always been possible, but dangerous
        // now it's safe thanks to zero-initialization:
        point p1;

        // this is exactly the same as above:
        point p2{};

        std::cout << "p1: " << p1.x << '/' << p1.y << '/' << p1.z << '\n';
        std::cout << "p2: " << p2.x << '/' << p2.y << '/' << p2.z << '\n';
}


>> p1: 0/0/0
>> p2: 0/0/0
```

This works but we lose the great advantage of initializing a point with the values we want in a comfortable way. The solution to this is called a constructor. It is a special function that is part of a struct and is called when the object is created.

Let's create one that behaves like the one we had in the beginning:

```
#include <iostream>

struct point {
        // a constructor has neither returntype nor is it
        // possible to return a value from it. Aside from that,
        // it's name is identical with that of it's class:
        point(double x_arg, double y_arg, double z_arg) {
                // we can access all member-variables of the struct
                // inside the constructor:
                x = x_arg;
                y = y_arg;
                z = z_arg;
        }
```

```
        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
};

int main() {
        // now these constructions work again:
        point p1{1,2,3};
        auto p2 = point{4,5,6};

        std::cout << "p1: " << p1.x << '/' << p1.y << '/' << p1.z << '\n';
        std::cout << "p2: " << p2.x << '/' << p2.y << '/' << p2.z << '\n';
}

>> p1: 1/2/3
>> p2: 4/5/6
```

If we look at the code, we see a very common situation: We have several data-members in our struct, one argument for each of them, and we directly assign the value of the argument to the member. This is fine, if the members are just doubles or ints, but it can create quite an overhead, if the default-construction of the member (which must be completed upon entry of the constructor) is expensive, like for `std::vector`. To solve this problem, C++ provides a way to initialize data-members before the actual constructor-body is entered:

```
#include <iostream>

struct point {
        // the members x, y and z are intialized with the arguments x, y, and z:
        point(double x, double y, double z) : x{x}, y{y}, z{z} {}
        //                          the actual body is now empty ↑↑

        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
};

int main() {
        point p{1,2,3};
        std::cout << "p: " << p.x << '/' << p.y << '/' << p.z << '\n';
}

>> p: 1/2/3
```

This way of initializing members is almost always preferable if it is reasonably possible. It should however be noted, that there is one danger using it: The member-variables are initialized in the order of declaration *in the class*, not in the order of the initialization, that the constructor seems to apply. As a result the following code is wrong:

```
struct dangerous_struct {

        // undefined behavior: var1 gets initialized before var2.
        // -> var2 is read before initialized
        dangerous_struct(int arg) : var2{arg}, var1{var2} {}

        int var1;
        int var2;
};
```

Note however, that it *is* allowed to initialize data-member from already initialized other data-members.

7.2 Methods

OK, so we are now able to read and write member-variables and initialize them via constructors. If we think about it, a constructor is just a special function that is part of the struct and their is no real reason to disallow other functions being part of structs.

These functions are called "member-functions" by the C++-standard, but are often referred to as "methods" by programmers. One advantage of using methods over free functions is that methods are tightly associated with a certain object and may therefore state the intent in a clearer way (we will learn more advantages as we will learn more about structs and classes).

So, how do we create them and how do we use them? Let's say we want to have a convenient way of getting a string that represents our point:

```
#include <iostream>

struct point {
        point(double x, double y, double z) : x{x}, y{y}, z{z} {}

        // note that the instance of point is passed implicitly
        std::string to_string() const {
                // as in the constructor we can access all data-members without
                // qualifying the instance of point:
                return '(' + std::to_string(x) + ", " + std::to_string(y)
                                + ", " + std::to_string(z) + ')';
        }
```

```
        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
};

int main() {
        point p{1,2,3};
        std::cout << "p: " << p.to_string() << '\n';
}


>> p: (1.000000, 2.000000, 3.000000)
```

So we just write a function inside the class and call it by picking an instance of the class and append the function-call with a '.' to it:

```
object.method(arguments)
```

The overall effect of this is somewhat similar to a free function that takes a reference to the object as first argument and is called like this:

```
function(object, arguments)
```

At this point we face a problem: We learned earlier that we should usually pass arguments as const references if reasonably possible. But since the instance of the methods class is passed implicitly we cannot annotate it directly. This is why the `to_string` method in our point-class has "const" at the end of it's signature: This annotates publicly that the method won't change anything in the class. If we really *want* to change the class, we just don't add it.

So, when should we use a method instead of a free function?

- If you mutate the internals of a struct or class, use a method.
- If the whole point of the operation is accessing internals of a struct, use a method.
- If the operation involves multiple objects and none of them is clearly the dominant subject, use a function.
- If the operation is not an important part of the struct or class, a function is often the better way: If you implement a class for numbers, make sinus a function, not a method.

Note that these are guidelines, not fixed rules, and that we will learn about further reasons to decide one way or the other as we go on.

7.3 Classes

Let's say that at this point we decide, that cartesian coordinates (x, y z) are boring and decide to use polar-coordinates instead. Polar coordinates consist of two angles that point into a direction and a distance:

```
#include <iostream>

struct polar_point {
        polar_point(double h, double v, double dist): h_angle{h}, v_angle{v}, distance{dist} {

        double h_angle = 0.0;
        double v_angle = 0.0;
        double distance = 0.0;
};

int main()
{
        polar_point p{0.0, 0.0, 123};
        std::cout << "distance to origin: " << p.distance << '\n';
}

>> distance to origin: 123
```

Let's assume that the angles are represented as radians. Also we want the distance to never be negative (in that case we would adjust the angles). This creates problems: A careless user of our point could easily create an invalid state. The solution for this is to restrict the access to the members: Only methods should be allowed to touch them directly. Everyone else should only be allowed to interact with them via methods. This can achieved by making them *private*:

```
#include <cmath> // for M_PI
#include <exception> // for termiante()
#include <iostream>

struct polar_point {
        polar_point(double h, double v, double dist):
                        h_angle{h}, v_angle{v}, distance{dist} {}

        double get_h_angle() const {return h_angle;}
        double get_v_angle() const {return v_angle;}
        double get_distance() const {return distance;}

        void set_distance(double dist);
```

```
        void set_h_angle(double angle);
        void set_v_angle(double angle);

private:
        double h_angle = 0.0;
        double v_angle = 0.0;
        double distance = 0.0;
};

void polar_point::set_distance(double dist) {
        if(dist >= 0) {
                distance = dist;
        } else {
                std::terminate();
        }
}

void polar_point::set_h_angle(double angle) {
        if(angle >= 0 && angle < 2* M_PI) {
                h_angle = angle;
        } else {
                std::terminate();
        }
}

void polar_point::set_v_angle(double angle) {
        if(angle >= 0 && angle < 2* M_PI) {
                v_angle = angle;
        } else {
                std::terminate();
        }
}

int main()
{
        polar_point p{0.0, 0.0, 123};
        p.set_h_angle(3.5);
        p.set_v_angle(2.7);
        std::cout << "distance to origin: " << p.get_distance()
                  << ", angles: " << p.get_h_angle() << ", " << p.get_v_angle() <<

        // this would make the program crash safely, before worse things could happ
        //p.set_h_angle(42);
}
```

```
>> distance to origin: 123, angles: 3.5, 2.7
```

While terminate is still a harsh way of handling errors (later on exceptions will make this cleaner), we can now be sure that nobody will touch our privates and bring them into a bad state.

To reiterate: Everything in a struct that comes after `private:` cannot be accessed from outside of the struct. In order to get back to the initial behavior, we can put a `public:` in the same way into the struct. There are some further details to this, but none that are currently important.

At this point we can introduce classes. Basically a class is the same as a struct with the single exception that everything in it is by default private instead of public. While this is the only technical difference the important difference lies in the usage-conventions. Basically all existing coding-standards agree that everything that consists of more than a few trivial public members and maybe some simple methods should be a class. Since it is generally considered a good idea to put the public interface first this ends up with the somewhat ironic situation that most classes start with `public:`.

Let's look at a simple example:

```
#include <iostream>

class some_class {
public:
        some_class(int val): mem{val} {}

        int get_mem() const {return mem;}
        void set_mem(int val) {mem = val;}
private:
        int mem;
};

int main() {
        some_class foo{4};
        std::cout << foo.get_mem() << '\n';
}
```

```
>> 4
```

We see that there really isn't much special about it. Nevertheless we'll use `class` instead of `struct` for most of our types from now on (with the exception of types that basically are only a collection of some values without fancy stuff).

7.4 Destructors

A constructor is a function that is called upon the construction of an object to initialize it's state correctly. Many languages have this feature. C++ is however one of the relatively few languages that also have the opposite: A destructor.

A destructor is a function that will run whenever an object ceases to exist. It's main purpose is to clean up any resources that the object might own in. Consider std::vector: It is a class that manages an arbitrary amount of integers; these have to be stored somewhere in memory and when the vector gets destroyed, the memory has to be returned to the system. The later is done in the destructor:

```
#include<vector>

int main() {
        std::vector<int> vec; // empty vector, uses little memory
        for(int i = 0; i < 10000; ++i) {
                // fill with 10000 integers, using increasing amounts of memory
                vec.push_back(i);
        }
        // at this point we have 40KB of memory in use. However: Once we leave the
        // std::vectors destructor will free this memory implicitly
}
```

Now, how is a destructor created? Basically it is just a method of the class that has no returntype and the name "~classname", for instance:

```
#include <iostream>

class myclass {
public:
        myclass(int i): i{i} {std::cout << "Hello from #" << i << '\n';}
        ~myclass() {std::cout << "Goodbye from #" << i << '\n';}
private:
        int i;
};

int main() {
        myclass object1{1};
        myclass object2{2};
}


>> Hello from #1
>> Hello from #2
>> Goodbye from #2
```

```
>> Goodbye from #1
```

As we see the objects that are constructed first get destructed last. This is guaranteed by the standard and quite important: Assume we want acquiring multiple resources, where some cannot exist without others already existing; thanks to the guaranteed order of destruction no object will cease to exist while other, later constructed ones, might still need it.

We will learn more about this technique in later chapters, for now it should be enough to know, that it is called "*'Resource Acquisition Is Initialization*"' (RAII) and that it is one the most important techniques of C++. Some people call it C++'s greatest feature.

7.5 Summary

In this chapter we learned how to create custom types. For a simple collection of values, we can use simple structs, if we need something more advanced, a class with private members and methods is usually a better solution.

Classes and structs can have member-functions (so called methods) of which constructors and destructors take a special role since they create/destroy the object.

8 Class-Templates

In the last chapter we created a struct for Cartesian points:

```
#include <iostream>

struct point {
        point(double x, double y, double z) : x{x}, y{y}, z{z}
        {}

        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
};

int main() {
        point p{1,2,3};
        std::cout << "p: " << p.x << '/' << p.y << '/' << p.z << '\n';
}


>> p: 1/2/3
```

This still works great, but their is one problem: What should we do, if we want a point that only consists of integral coordinates? Or with floats instead of doubles? Or with complex numbers?

The obvious and bad solution would be to create one struct for each and give them different names like "`point_f`", "`point_i`" and so on. This is repetitive, boring, error-prone and therefore hard to maintain. Especially since the only difference in these structs will be type of the values.

To solve this problem C++ has so called class-templates (they work with structs too). So if we want a point-class for every type T, we can just write this:

```
#include <iostream>

template<typename T>
struct point {
        // use const-references because T might be a 'heavy' type:
```

```
        point(const T& x, const T& y, const T& z) : x{x}, y{y}, z{z} {}

        T x = 0;
        T y = 0;
        T z = 0;
};

int main()
{
        point<int> p_int{1,2,3};
        std::cout << "p_int: " << p_int.x << '/' << p_int.y << '/' << p_int.z << '\

        point<float> p_float{1.5,2.3,3.2};
        std::cout << "p_float: " << p_float.x << '/' << p_float.y << '/' << p_float
}


>> p_int: 1/2/3
>> p_float: 1.5/2.3/3.2
```

This is not really different from writing normal functions so far, so let's see how we can create methods.

If we implement the method directly in the class, there is no difference at all. If we want to implement it outside of the class, there are two small changes:

- Instead of `foo::bar()` we have to write `template<typename T> foo<T>::bar()` in the signature, since there is now more than one class called foo.
- The implementation must be available to every user (and since not be put into another file). Since we haven't yet grown out of just using one file, this is currently no big deal.

Let's see an example:

```
#include <iostream>

template<typename T>
struct point {
        point(const T& x, const T& y, const T& z) : x{x}, y{y}, z{z} {}

        T x = 0;
        T y = 0;
        T z = 0;

        //definition in class-template:
        void print() {
```

```
                std::cout << x << '/' << y << '/' << z << '\n';
        }

        // definiton outside
        void reset();
};

template<typename T>
void point<T>::reset() {
        x = 0;
        y = 0;
        z = 0;
}

int main() {
        point<int> p{1,2,3};
        p.print();
        p.reset();
        p.print();
}

>> 1/2/3
>> 0/0/0
```

That is basically it. There really shouldn't be any surprises so far. We will learn more about this mechanism in the future, for example how we can create special versions for certain instantiations and why this mechanism is way more powerfull than it currently appears.

9 Inheritance

Let's say we want to write a simple game: We have different types of fighting units with different strength and abilities. Among them are a knight and a guard, where the knight is an offensive unit while the guard clearly has a focus on defense:

```
#include <iostream>

class knight {
public:
        knight(): health_level{100}, defense_level{15}, attack_level{35} {}

        bool alive() const {return health_level > 0;}

        unsigned defense() const {return defense_level;}
        unsigned attack() const {return attack_level;}
        void train() {++attack_level;}

        template<typename Defender>
        void attack(Defender& defender) const {
                defender.defend(attack());
        }

        void defend(unsigned attack_strength) {
                if (attack_strength <= defense_level) {
                        return; // no damage done
                }
                const auto actual_attack_strength = attack_level - defense_level;
                if (actual_attack_strength >= health_level) {
                        health_level = 0; // warrior is dead
                } else {
                        health_level -= actual_attack_strength;
                }
        }
private:
        unsigned health_level;
        unsigned defense_level;
        unsigned attack_level;
};
```

```
class guard {
public:
        guard(): health_level{100}, defense_level{30}, attack_level{20} {}

        bool alive() const {return health_level > 0;}

        unsigned defense() const {return defense_level;}
        unsigned attack() const {return attack_level;}
        void train() {++defense_level;}

        template<typename Defender>
        void attack(Defender& defender) const {
                defender.defend(attack());
        }

        void defend(unsigned attack_strength) {
                if (attack_strength <= defense_level) {
                        return; // no damage done
                }
                const auto actual_attack_strength = attack_level - defense_level;
                if (actual_attack_strength >= health_level) {
                        health_level = 0; // warrior is dead
                } else {
                        health_level -= actual_attack_strength;
                }
        }
private:
        unsigned health_level;
        unsigned defense_level;
        unsigned attack_level;
};



int main() {
        knight black_knight{};
        guard castle_guard{};
        while (true) {
                black_knight.attack(castle_guard);
                if (!castle_guard.alive()) {
                        std::cout << "The castle has fallen!\n";
                        return 0;
                }
```

```
                castle_guard.attack(black_knight);
                if (!black_knight.alive()) {
                        std::cout << "The castle has been defended.\n";
                        return 0;
                }
        }
}
```

```
>> The castle has fallen!
```

This is quite a lot of duplicate code to create two warrior-classes that are almost identical; we also require a member-template to implement the attack-method, which isn't actually that bad but we'll see that there is a better solution for this one too.

If we look into the code we see that most of the properties it has are actually things that are shared among all kinds of warriors. This is where *inheritance* comes into play.

9.1 The Basics

Inheritance is a (probably badly named) technique to describe a very general thing that has a certain set of properties and use this to implement more specialized versions. Let's look at a simple first version:

```
#include <iostream>

class warrior {
public:
        warrior(unsigned health_level, unsigned defense_level, unsigned attack_level):
                health_level{health_level}, defense_level{defense_level}, attack_level{attack_

        bool alive() const {return health_level > 0;}

        unsigned defense() const {return defense_level;}
        unsigned attack() const {return attack_level;}

        template<typename Defender>
        void attack(Defender& defender) const {
                defender.defend(attack());
        }

        void defend(unsigned attack_strength) {
                if (attack_strength <= defense_level) {
                        return; // no damage done
```

```
                }
                const auto actual_attack_strength = attack_level - defense_level;
                if (actual_attack_strength >= health_level) {
                        health_level = 0; // warrior is dead
                } else {
                        health_level -= actual_attack_strength;
                }
        }

protected:
        unsigned health_level;
        unsigned defense_level;
        unsigned attack_level;
};

class knight: public warrior {
public:
        knight(): warrior{100, 15, 35} {}
        void train() {++attack_level;}
};

class guard: public warrior {
public:
        guard(): warrior{100, 30, 20} {}
        void train() {++defense_level;}
};

int main() {
        knight black_knight{};
        guard castle_guard{};
        while (true) {
                black_knight.attack(castle_guard);
                if (!castle_guard.alive()) {
                        std::cout << "The castle has fallen!\n";
                        return 0;
                }
                castle_guard.attack(black_knight);
                if (!black_knight.alive()) {
                        std::cout << "The castle has been defended.\n";
                        return 0;
                }
        }
}
```

```
>> The castle has fallen!
```

No matter how we look at it, this is definitely an improvement.

Some things to note at this point:

- Instead of private, the attributes of our warrior-class are `protected`; this is a mixture of `public` and `private` that allows inheriting classes to access these members as if they were public but seals the access to everyone else (private members cannot be accessed in inheriting classes).
- To create a class knight that is a special form of a warrior we write `class knight: public warrior{};`. This will copy all the properties of a warrior into our knight. The `public` is very important here, since there is also a thing called private-inheritance (which is the default here), that has only a very limited number of applications and protected-inheritance that only exists for completeness (it is completely unheard of any situation in which it would solve a problem). Don't worry about these two here, they really should be considered experts-only features.
- In the constructor of knight we call the constructor of the base-class before everything else; if we don't do this, the default constructor will be called.

10 Containers

At this point we understand enough about the core-language in order to take a look at the standard-library. Until now we have already made use of things like the io-facilities, `std::vector` and `std::string`. This chapter will now try to get a somewhat more systematic view of what is already offered and how it can be used to it's best effect.

While the standard-library of C++ is much smaller than the ones of some other languages, it is still to big to learn completeley in one attempt. It is therefore strongly recommended to occasionaly take some time and go reading references; sometimes one will come across real gems where there were none expected.

10.1 Iterators

Whenever we needed an arbitrary amount of elements so far, we resorted to `std::vector` without looking to deep into it's concepts and features. We already saw that we can copy them, request an element at a certain index, iterate over all elements and insert at the end:

```
#include <iostream>
#include <vector>

int main() {
        std::vector<int> vec = {1, 2, 3};
        vec.push_back(4);
        for (auto&& elem: vec) {
                std::cout << elem << '\n';
        }
        auto vec2 = vec; // copy
}

>> 1
>> 2
>> 3
>> 4
```

Usually `std::vector` is exactly what we want, but sometimes we would like to use different data-structures, because they are faster or have otherwise different semantics.

The most important container aside from `std::vector` is probably `std::array` from the `<array>`-header. It is basically like `std::vector` except that the size is fixed at compile-time which implies that methods like `push_back` or `pop_back` do not exist for it. In exchange it can be faster, especially for very small element-counts.

Another container from the stdlib is `std::list`. It is a so called doubly-linked-list and allows relatively cheap insertion in the middle of the sequence in exchange for much slower iteration and insertion at the end. In many real-world problems it is a much worse alternative compared to `std::vector` so you really shouldn't use it, unless **measuring(!)** shows that `std::vector` is the bottleneck for some problem and `std::list` vastly improves this.

The reason for looking into `std::list` is at this point however not to actually use it, but because it points us to a problem that we didn't have so far: Since we cannot access an element in the middle of a list efficiently, the classtemplate doesn't offer us `operator[]` or `at()`. Instead it forces us to use the way more general *iterators*.

Iterators are C++'s answer to the question how to represent a generic range. An iterator is basically an object that points to a certain object in a sequence. We can retrive that object or move the iterator on. For starters, let's look at the iterators of `std::vector`:

```
#include <iostream>
#include <vector>

int main() {
        std::vector<std::string> vec = {"foo", "bar", "baz"};

        // get an iterator that points to the first element
        // in the vector (1):
        std::vector<std::string>::iterator it = vec.begin();

        // retrive the element that the iterator points to
        // using the dereferencing-operator '*':
        std::cout << *it << '\n';

        // since the iterator is not constant we can assign
        // new values through it too:
        *it = "meow";
        std::cout << *it << '\n';

        // finally we can move the iterator to the next element
        // using the increment-operator:
        ++it;
        std::cout << *it << '\n';

        // To call a method on the referenced object, we can
```

```
        // either dereference it  first or just use the
        // arrow-operator:
        std::cout << (*it).size() << ", " << it->size() << '\n';

        // finally we can compare iterators:
        if (it != vec.begin()) {
                std::cout << "it doesn't point to the first "
                             "element anymore.\n";
        }
        if (it == ++(vec.begin())) {
                std::cout << "it does instead point to the second.\n";
        }
}
```

```
>> foo
>> meow
>> bar
>> 3, 3
>> it doesn't point to the first element anymore.
>> it does instead point to the second.
```

Most of the above operations are supported by all iterators. The exceptions for this rule are comparission and ironically the methods to access the value: Some iterators only allow reading, but not writing those, others only allow writing, but not reading. Most however allow both.

Another possible limitation is that we are not allowed to make a copy of the iterator, increment the original and then use the copy:

```
// this may be illegal for some iterators:

auto original_iterator = get_output_iterator();
auto copy = original_iterator;

++original_iterator;
++copy;
```

Since this is quite complicated but doesn't help with understanding the basic ideas and concepts, let's put these iterators aside for a moment and focus on those that do allow these things.

We call these *forward-iterators* since they allow us to traverse a sequence in a "forward order".

Forward-iterators are split into two categories: Const and mutable iterators. As the names suggest, you are not allowed to change a value through a const-iterator and the

compiler enforces this. Obviously it is a good idea to prefer them to mutable iterators, *if* this is possible.

Now, sometimes we want to do more than just access all elements of a sequence in ordern. A very simple thing that we might wish in addition is to move backwards. This is where *bidirectional-iterators* come into play.

A bidirectional-iterator allows us to move backwards by using the decrement-operator; this is by the way the kind of iterator that `std::list` offers:

```
std::list<int> my_list = {1, 2, 3, 4, 5};
// to get an iterator to the first element, we call .begin():
auto it = list.begin(); // it -> 1
++it; // it -> 2
++it; // it -> 3
--it; // it -> 2
--it; // it -> 1
```

So far, so unsurprising. It should also be noted, that we are always allowed to use the post-increment/decrement-operator, when we are allowed to use the pre-versions.

Before we move on to the most powerfull iterator-category, let's take a look at how we can use iterators to represent a range.

Obviously we need two iterators to do this: One to indicate the begining of the range and one to indicate the end. The obvious choice for this would be to use an iterator to the first element as start, and an iterator to the last element as end. This would hovever lead to several problems, one being that we could not represent an empty range.

The solution to this problem is that we use a so called half-open range: The first iterator points to a valid element, but the last one points one element past the last valid one. If the range is empty, both iterators are equal. Let's take a look at this:

```
#include <list>

int main() {
        std::list<int> lst;

        // .end() returns the past-the-end-iterator:
        if (lst.begin() == lst.end()) {
                std::cout << "The list is emtpy.\n";
        }

        lst.push_back(23);

        if (lst.begin() != lst.end()) {
                std::cout << "The list is not empty.\n";
        }
```

```
        // this is how we can use iterators to iterate over
        // all elements of a sequence:
        for (auto it = lst.begin(); it != lst.end(); ++it){
                std::cout << *it;
        }
}
```

```
>> The list is empty.
>> The list is not empty.
>> 23
```

Now that we have seen that it is time to look at the most powerfull iterators of all: *Random-access-iterators*.

Random-access-iterators are iterators that allow moving more then one element in either direction and have the ordering-comparission-operators in addition to the equality-operators. Let's take a look at all these things in order.

In order to move a random-access-iterator more than one element, we use the normal arithmetic operators (plus and minus):

```
// the iterators of std::vector provide random-access,
// which is one of many reasons why std::vector is so
// awsome:
std::vector<int> vec = {0, 1, 2, 3, 4, 5, 6};

auto it = vec.begin(); // it -> 0
it += 3; // it -> 3
it -= 1; // it -> 2

// note that we can add negative offsets:
it += -2; // it -> 1

auto it2 = it + 3; // it2 -> 4
it2 += 3; // it2 = past-the-end-iterator
```

Concerning the comparission: In general we are only allowed to compare iterators that are part of the same range for anythying but equality. As one might expect, the iterators to elements that come earlier in a sequence compare smaller to iterators to later elements:

```
std::vector<int> vec = {0, 1, 2, 3, 4, 5, 6};

auto it1 = vec.begin();
auto it2 = it1 + 3;
```

```
assert(it1 < it2);
```

Before you continue, you should make sure, that you have understood at least the basic concepts of iterators, since large parts of the standard-library are basically built around them. Let's take a look at some examples for that:

- If we want to concatenate the contents of a container to a `std::vector`, we use iterators for that: `vec.insert(vec.end(), other_container.begin(), other_container.end());`
- If we want to write our own container and want it to be usable with range-`for`, all we need to do is implement iterators and the appropriate `.begin()` and `.end()` member-functions.
- Whenever the standard-library provides functionality that is intended to be used with sequences, it works on iterators.

10.2 Algorithms

10.3 Tuples and Pairs

11 Memory

In this chapter we will take a first look into the memory-model of C++. It should be noted beforehand that while these things are often very useful in order to understand the behavior or design-decissions of the language, things like naked pointers are usually **not** language-features that should be used in regular code.

It should also be noted, that the compiler is usually very good in making highlevel-code fast, so using these things won't make your code faster. In fact the chances are good, that using the stdlib-facilities will usually be faster due to more optimization there.

Finally, these things are dangerous. In this chapter we will come around undefined behavior more often then in any other chapter so far (only multithreading will rival us in that regard). The things here are like the nuclear weapons of C++ and you should treat them with the proper respect.

11.1 What is memory?