Learning C++

Florian Weber

Contents

Chapter 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 Variables and basic Arithmetic

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
```

This 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() {
```

```
    int a = 3;
    int b = 5;
    int sum = a + b;
    std::cout << sum * sum << "\n";
}
```

```
>> 64
```

`int` is the default type for integers. On most modern systems it can hold numbers between -2147483648 and 2147483647, which should be enough for most applications. The statement `int a = 3;` now creates a new integer with the name `a` and the value 3. `a` is called a variable since it holds a value that can be changed. Almost everything that can be done with literal numbers in the code can also be done with variables.

The third statement (`int 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`.

1.3 Conditional Execution

1.4 Undefined Behaviour

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 behaviour 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 signed-integer-overflow, reading uninitialized variables and accessing unowned memory. Possible behaviour 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:

```
int 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 protest 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.

1.5 Vectors

1.6 Loops

1.7 Summary

Chapter 2

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.

2.1 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.

2.2 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.
}
```

2.3 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.
```

2.4 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() {}
```

2.5 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
```

```cpp
// 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.

Chapter 3

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) << std::endl;
}
```

```
>> 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) << std::endl;
}

>> 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.

Chapter 4

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`.

4.1 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.

4.2 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.

4.3 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) << std::endl;
}
```

```
>> 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()) << std::endl;
}
```

```
>> 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."

Chapter 5

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.

5.1 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
```

5.2 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.

5.3 Non-type parameters

Chapter 6

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.

6.1 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.

6.2 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.

6.3 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{di

    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() << '\n';

    // this would make the program crash safely, before worse things could happen:
    //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).

6.4 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 function,
    // 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.

6.5 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.

Chapter 7

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 << '\n';

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

```
>> 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.

Chapter 8

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.

8.1 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_level} {}

    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.