Contents

The first C++ program

This section first shows how a simple C++ program looks like with details. Let's compare the following two programs written in Python and C++, the functionality of which is equivalent to each other:

```
#
#  Version 1: Python
#

def main():
    print("Hello world!")
    print("How are you?")

main()
```

```
//
//  Version 2: C++
//

#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    cout << "How are you?"
         << endl;
}
```

What similarities and differences do the programs have:

- C++ program will automatically start from the function named `main`.
- In Python it was just an agreement to name the starting function as `main`. We did this because that is how many programming languages do it anyway. Python doesn't require this, but C++ does.
- Most of the C++ commands end with semicolon (;). Unfortunately this rules is not universal and one can't just stick semicolon everywhere. This is a major source of angst and programming errors for an inexperienced C++ programmer who hasn't yet quite figured out where semicolon belongs and where it doesn't.
- C++ commands generally end with semicolon, and this makes the language very flexible when it comes to dividing single command in multiple lines without having to use any special notation. Python on the other hand usually requires the character \ at the end of the continued line.
- Program lines belonging together (**blocks**) are written inside **curly braces** { } in C++, when Python used indentation for the same purpose. However, it is a good practice to use indentation in C++, too, because it makes programs more readable and understandable.
- Comments in C++ start with `//`. Python used # sign instead.
- Output command is not part of C++ language itself, but it has been implemented in a library instead. If you want to print something on the screen or read user input from the keyboard, you have to use `iostream` library:

```
#include <iostream>
```

The `#include <...>` structure is analogous to `import` command in Python. If you need program code written in another file, you can use `include` command. If you want the compiler to search for the other file from C++ library files, you will write the name of the library between angle brackets, as above. If you want the compiler to search for the file from the files written by the programmer, you will write the name of the file between quote marks (`""`).

Lines beginning with `#` are directives for the preprocessor. Preprocessor is a part of the compiler, and its task is to prepare source code files for the actual compilation. Preprocessor makes textual substitution, for example, directive`#include <iostream>` copies the content of `iostream` library in the place of the directive.

- When `#include <iostream>` has been added in the beginning of a source code file, information can be printed on the screen using `cout` command with `<<`operator. One `cout` command can be used to print as many values as required just by adding `<<` operator in front of each value. If a special value named `endl` is printed, the cursor moves in the beginning of the next line on the screen.
- C++ code has one extra line that Python version seemingly does not have:

```
using namespace std;
```

This line makes it possible to use names from the `iostream` library as such, like `cout` and `endl`, instead of `std::cout` and `std::endl`.

Python actually has exactly the same mechanism working when using libraries (the example above did not). When using libraries in Python, one choice is to write:

```
import math
...
math.sqrt(2.0)
```

One must use the library name as a prefix when using the service of the library. This can be avoided by writing:

```
from math import *
...
sqrt(2.0)
```

Command `using namespace` makes it possible to refer to identifiers declared in a named **namespace** without the name of the namespace as a prefix. On the other hand, using such prefixes can be useful, too. In a way, a namespace defines a set of identifiers and makes them invisible outside the namespace. In this way, namespaces enable use of same identifiers, if the identifiers have been defined in different namespaces, when it is clear, which of these identifiers with the same name is in question. For example, a student register program could use identifier `name` to refer to the name of a student as well as to that of a course. If both of these names have been defined a namespace of its own, we could refer to them as `Student::name` and `Course::name`.

At this phase of the course, you need not worry about namespaces very much. It is enough to realize that if an identifier belongs to some other namespace than the global one, the compiler does not find it, if you have not written command `using namespace` or written the name of the namespace as a prefix as:

```
name_of_the_namespace::identifier
```

Huom

The larger programs you write, the more important it is to avoid ”useless” names. If you use large libraries with all idenfiers of them, the number of identifiers may grow so big that it becomes difficult to invent new names.

Because of that, it is usually better to write with prefixes as `std::cout` and `std::endl` than using the whole namespace with command `using namespace std;`, which enables writing `cout` and `endl`, although the latter form is shorter and simpler.

Without command `using namespace`, the program above is as follows:

```
//
//  Version 3: C++ with better programming style
//

#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
    std::cout << "How are you?"
              << std::endl;
}
```

Interpretation vs compilation

Python is an **interpreted** programming language and C++ a **compiled** one.

Executing programs written in an interpreted programming language requires a utility program every single time we want to run the program. This program is called an **interpreter** (e.g. Python interpreter). If the interpreter program is uninstalled or it somehow gets deleted, it is not possible to run programs written in the programming language the lost interpreter was for. The interpreter's job is to transform the commands in the source code to the machine language as the program's execution progresses. This process is required since the CPU of the computer doesn't know how to directly run programs written in any other language than its own machine language.

In the case of compiled programming languages a utility program is still required if we wish to execute our program. This program is called a **compiler** (e.g. C++ compiler). Compiler processes the source code completely before it is ever executed.

Compilation process includes error checkings, and it produces a whole machine language program that is usually stored in the hard drive to a separate file (e.g. with `.exe` suffix in Windows). Once the compilation process is completed, the compiler program is not needed anymore since the fully compiled machine language commands have been stored in the executable file, which can be run over and over again without any utility program (i.e. the compiler). If the source code is ever modified, it must, of course, be compiled again if we want the modifications to be compiled into the machine language so that we can run the new version of the program.

There are many programming languages which are somewhere between the two extremes described above. Programs written in these languages are translated into some kind of simpler form (bytecode) which is then interpreted. In a way, this is a hybrid form between interpreting and compiling.

Strictly speaking, compilation consists of (at least) two phases. The program is first compiled into so called object file. In this form, the program is in machine code, but it cannot be executed yet. The reason is that the final code still needs parts from the library, and thus, the program must be linked.

Linking means that the separately compiled parts are joined together as a single *executable*program. Compilers typically hide the linking phase, but large programs require that compilation must be done in two phases to avoid recompilation after each little modification. This benefit is significant, if the program is greater than 100000 or a million of lines, but it can be observable even with much smaller programs, too.

Variables and typing

In processing variables, C++ has more details than Python. However, you need not worry, although getting started feels slow to you.

Python vs. C++: Variables and input

Let's compare the following two programs (which work exactly the same):

```
def main():
    name = input("Input your name: ")
    age = int(input("Input your age: "))

    print("Nice to meet you", name, age)
    print("After", 50 - age, "years you'll be 50.")

main()
```

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

using namespace std;

int main() {
    string name = "";
    cout << "Input your name: ";
    getline(cin, name);

    int age = 0;
    cout << "Input your age: ";
    cin >> age;

    cout << "Nice to meet you " << name << " "
         << age << endl;
    cout << "After " << 50 - age
         << " years you'll be 50."
         << endl;
}
```

Remarks and conclusions:

- C++ program is often longer that similarly functioning Python program. One example is not a proof (obviously), but experience in general seems to support this idea.
- In Python one can start using a variable just by assigning it a value with an assignment command (=). Python is not very strict about the type of information you store in a variable and during the program execution a variable can contain values that are of different type.

  In C++ a variable *must always be declared* before it can be used. A declaration tells the C++ compiler the name of the variable we want to use in the future and as importantly what type of information can we store in it.
- In Python variable and function names can contain any characters that are considered letters. For example scandinavian letters are accepted. In C++ variable and function names must consist of ASCII-letters, digits, and underscore (_). The first character must be a letter.

- In strings, scandinavian letters are accepted also in C++. However, in some situations they can be problematic, so it is good to avoid them.
- Unlike in Python, the data type that we use to manipulate text (`string`) is not part of the language: it is implemented as a library. Therefore if we need to use strings in C++ programs, we must include the string library, i.e. write `#include <string>` in the beginning of the program.
- Unlike `print` function in Python, `cout` in C++ doesn't automatically add spaces between the values it prints. It doesn't add newline at the end either, unless `endl`is manually added.
- Variables `cin` and `cout` are C++'s mechanism to handle keyboard and screen in C++ code (input and output operations).
- The primitive ways to read text from input stream `cin` are function `getline` for reading strings and input operator `>>` for reading a value of a suitable type for a variable.
- Note that `input` function in Python and `getline` function in C++ work in different ways. The former one takes the printable string as a parameter and it returns the user-given string. The latter takes two parameters, the first of which is an input stream (that is always `cin` in the beginning of the course). The second parameter is a variable, which the input value will be assigned to. Note that the return value of `getline` cannot be assigned to a string.
- Input operator `>>` skips all empty characters, also line breaks. This means that, the input is read until the user gives visible characters. Test this by inputting a couple of line feeds before giving a number as the age. This may feel strange when compared to Python, where we have used to read input line by line with `input` function.
- This is enough to let us start. We will later study IO stream in more detail, and we will also learn how to check the correctness of user input (when the user input with `>>` is of an invalid type).

Variables in C++

In C++ a variable must be defined (or at least declared) before it can be used. The definition of a varible contains two parts: **declaration** and **initialization**. Declaration consists of two parts, and thus, there are totally three parts:

1. the type of the variable
2. the name of the variable
3. initialization of the variable (to set an initial value).

Parts 1 and 2 (declaration) are mandatory. Some concrete examples of variable definitions:

```
// Variable definitions without initialization (i.e. declaration)
int age;
double temperature;
string name;

// Variable definitions with initialization
int age = 21;
double temperature = 16.7;
string name = "John";
```

A declared but uninitialized variable has an undetermined value until it is set somehow (e.g. with `cin >>` or = operator):

```
int age;
double temperature;
// At this point both age and temperature have undetermined values

age = 21;
cin >> temperature;
// Now both variables have determined values
```

Varoitus

Uninitialized variables may produce programming errors that are very difficult to trace. You follow a good programming style, when you always initialize variables in definition.

C++ is strict about the type of a variable: *one can only store into a variable a value of the same type as the variable's type is.*

Scope of a variable

Local variables are defined inside a block (a pair of curly brackets {}) and it can be used from the point of its definition all the way to the closing bracket of the block it was defined in:

```
int main() {
    int guests = 0;
    while ( guests < 100 ) {
       int free_seats = 100 - guests;
       ... lines hidden ...
       // This is the last point where variable free_seats can be used.
    }
    ... lines hidden ...
    // This is the last point where variable guests can be used.
}
```

Data types

The data types in C++ that most of the simple programs can be implemented with are `int`, `double`, `bool`, and `string`. C++'s `double` is basically the same what was called `float` in Python (decimal number).

In addition to these types that were pretty much familiar from Python, C++ has a data type called `char` which is used to manipulate single characters. A simple example of using `char` type is:

```
char gender = 'F'; // F for female
```

Literals and constants

Term **literal** means a nameless constant value, for example: `42`, `1.4142`, `'x'`, `"text"`, or `true`.

Notice the following differences between Python and C++:

- Python has no separate data type for manipulating single characters, instead they are handled as strings which contain only one character.
- In Python it makes no difference whether you use single quotes or double quotes to express string literal. In C++ `string` type literals are written inside double quotes as `"abc"`, and `char` literals inside single qoutes as `'x'`.
- Therefore, for example, `´X´` is a character, and `"X"` is a string of a single character. String `"X"` cannot be assigned to a variable, the type of which is `char`.

There is one extra feature concerning C++ variables that doesn't have a counterpart in Python: **constants**. They are basically variables whose value can't be changed after it is set in the initialization.

```
const double PI = 3.14;

... lines hidden ...

// Changing the value of constant yields an error message
PI = 3.141592653589793; // ERROR
```

Constants are meant for naming such values that always remain the same. Natural constants as *pi* are straightforward examples, but there are other uses, too:

```
const int LOTTO_BALLS = 7;
```

Because you cannot change the value of a constant after definition, it is quite logical that it must be initialized with declaration.

It is a common practise to name constants with ALL_CAPITAL_LETTERS, and if the name of the constant contains multiple words, the spaces are replaced with underscore. Descriptive constant names make the source code easier to understand.

Dynamic vs. static typing

How does the following Python program behave?

```
def main():
    print("Start")
    var_1 = 5.3
    var_2 = 7.1
    print(var_1 + var_2)
    var_2 = "Hello"  # Recycling var_2 for a new purpose
    print("Midpoint")
    print(var_1 + var_2)
    print("End")

main()
```

When the program is executed it will fail on line 8 with the following printout:

```
Start
12.399999999999999
Midpoint
Traceback (most recent call last):
  File ``code/part-01-04.py'', line 11, in <module>
    main()
  File ``code/part-01-04.py'', line 8, in main
    print(var_1 + var_2)
TypeError: unsupported operand type(s) for +:
  'float' and 'str'
```

This behaviour is a result of **dynamic typing** which is used in Python. It means that the validity of the data is only checked when we try to use the data.

In practise dynamic typing in a programming language results:

- One variable can contain different type values during the execution of the program.
- The program will run fine until we reach the point where we try to do something with values that are not compatible.

Dynamic typing is very common in interpreted programming languages.

The same example in C++ would look something like this:

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

using namespace std;

int main() {
    cout << "Start" << endl;
    double var_1 = 5.3;
    double var_2 = 7.1;
    cout << var_1 + var_2 << endl;
    string var_3 = "Hello";  // New variable
    cout << "Midpoint" << endl;
    cout << var_1 + var_3 << endl;
    cout << "End" << endl;
}
```

If we try to compile this program we will get a compilation error (simplified here to make a point):

```
main.cpp:13: error: no match for 'operator+'
             cout << var_1 + var_3 << endl;
```

C++ uses **static typing**: if we try to manipulate data in our program in a way that C++ language doesn't allow/understand, we will get a compilation error.

In practise when we use statically typed language the following are true:

- Variables must be defined before they can be used.

- Variables have a permanent type and only the data that is of the same type can be stored in the variable.
- If there is an error in the program related to types, the compiler produces an error message and the program will not be compiled. Therefore it can't be executed, tested, or enjoyed in any other way.

Static typing is very common in compiled programming languages.

Notice how on line 11 in our C++ test program we had to define a new `string` variable `var_3`, since we cannot assign a string to `var_2`, although this is possible in Python.

Good side of the dynamically typed languages is that programmer is saved from micromanaging the types of variables i.e. not having to define every single variable manually. Many programmers also like the possibility of using some variables to store different type values.

Negative sides of dynamically typed languages include the very real problem of producing a program that seems to work but actually has some hidden errors lurking somewhere. This happened in the example program, when we tried to add a real number with a string. In addition, using the same variable for different purposes is not a good idea from the point of view of code understandability.

Type conversion

In type conversion, the compiler considers the data of a variable as a different type, from that given in the definition of the variable. Type conversion is either *implicit* or *explicit*.

Compiler manages implicit type conversion automatically. For example, it can convert single precision `float` to double precision `double` automatically:

```
float f = 0.123;
double d = f;
```

Or it can convert `int` to `double`:

```
double d = 3;
```

Moreover, it can convert integer to a character:

```
char ch = 113;  // ch is 'q', the ASCII value of which is 113.
```

Explicit type conversion happens, when the programmer wants to define how to consider the data of a variable as of a different type. In such case, the programmer uses operator `static_cast`.

For example, comparison between a signed and unsigned int is suspicious, and thus, the compiler gives a warning:

```
int i = 0;
unsigned int ui = 0;
...
if( ui == i ) {  // This line yields a warning
   ...
}
```

If the programmer is absolutely sure that he/she wants the compiler to see a signed int as an unsigned int, he/she can do the type conversion explicitly, and the warning vanishes:

```
if( ui == static_cast< unsigned int >( i ) ) {
   ...
}
```

The following example emphasizes for the reader that a character is considered as an integer:

```
cout << "Input a character: ";
char ch = ' ';
cin >> ch;
int ascii_value_of_ch = static_cast< int >( ch );
cout << "ASCII value of " << ch << " is " << ascii_value_of_ch << endl;
```

Control structures

Like Python, also C++ has control structures for selection and iteration. We will first compare `if`and `while` structures of these languages. After that we see one way to use `for` loop in C++.

Python vs. C++: selection and loops (if and while)

Let's compare the selection and loop structures of Python and C++ by considering the following example programs:

```
def main():
    secret_number = int(input("Input a secret number: "))
    guessed_number = -1
    guesses = 0

    while secret_number != guessed_number:
        guessed_number = int(input("Give a guess: "))

        guesses += 1

        if guessed_number < secret_number:
            print("Your guess is too small!")
        elif guessed_number > secret_number:
            print("Your guess is too great!")
        else:
            print("Correct!")

    print("Number of guesses: ", guesses)

main()
```

```
#include <iostream>

using namespace std;

int main() {
    int secret_number = 0;
    cout << "Input a secret number: ";
    cin >> secret_number;

    int guessed_number = -1;
    int guesses = 0;

    while ( secret_number != guessed_number ) {
        cout << "Give a guess: ";
        cin >> guessed_number;

        guesses += 1;

        if ( guessed_number < secret_number ) {
            cout << "Your guess is too small!" << endl;
        } else if (guessed_number > secret_number ) {
            cout << "Your guess is too great!" << endl;
        } else {
            cout << "Correct!" << endl;
        }
    }

    cout << "Number of guesses: "
         << guesses << endl;
}
```

Without syntactic variance (semicolons, parenthesis etc.) the examples have no differencies. Based on this, we can see what is meant by the claim that programming is a language-

independent skill. For example, iteration is a concept that keeps the same from programming language to another.

Let's pay some attention to use of parenthesis. In C++, you can drop the curly brackets away, if the block has only a single command. For example, the following code:

```
int = 0;
while(i < 10)
{
    ++i;
}
cout << i << endl;
```

can be shortened as:

```
int = 0;
while(i < 10)
    ++i;
cout << i << endl;
```

This may seem to be natty, but before you get used to write code following the latest example, you should note that such a programming style can lead to programming errors that are hard to detect. When writing Python code, you have got used to think that indented text belongs inside a block, and thus, you may easily forget the parenthesis. C++ compiler does not mind about missing parenthesis in the following kind of example:

```
int = 0;
while(i < 10)
    ++i;
    cout << i << endl;
cout << "Job continues after the loop" << endl;
```

The first `cout` statement will not be repeated, because it does not belong to the while block. Since the curly brackets are missing, the block consist of a single statement, which in this case is `++i;`. The first `cout` statement is executed only once: at the moment when the iteration ends and the next statement after the iteration is executed.

C++ compiler does not care how you have divided the program text into lines. Two most used ways to position the curly brackets are as follows. The first one of them leads to smaller number of lines:

```
if( x ) {
    ...
}
```

The other way has the same indentation for the beginning and ending brackets:

```
if( x )
{
    ...
}
```

You can use either of these ways, but it is better to be consistent, especially when concerning control structures. Do not mix the style in the same program.

Varoitus

In Python programming you may have used exception mechanism (`try`structure) for example in the place of `if` statement. Unless you are absolutely sure that you can, then do not try to use exceptions of C++ on this course. Their proper use will be explained only on the next programming course.

For statement in C++ (one way of use)

C++ also has a `for` loop that has several ways to use. It is especially handful for going through the elements of a data structure. Therefore, we will introduce such usages only when we have learned to use data structures of C++.

To enable you to use `for` loop already in the exercises of this round, we show how to use it in going through a certain interval of integers:

```
for ( int number = 5; number < 10; ++number ) {
    cout << 9 * number << endl;
}
```

The above code corresponds to the following Python code:

```
for number in range(5, 10):
    print(9 * number)
```

About C++ operators

The conditions (truth-valued expressions) of previous examples showed some relational operators of C++. In all, C++ has the following relational operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`, the meaning of which is the same as Python has for the corresponding operators. Like Python, C++ has assignment operators: `+=`, `-=`, `*=`, and `/=`.

As logical operators, C++ has the following:

- logical not: `not a`, which is the same as `!a`
- logical and: `a and b`, which is the same as `a && b`
- logical or: `a or b`, which is the same as `a || b`.

The two last ones are so called short-circuit operators, which means that the second operand will not be evaluated, if the result of the whole expression can be concluded based on the first operand. For example, `false && a` is false, regardless of the value of `a`. Similarly, `true || a` is true, regardless of the value of `a`.

Moreover, the two last logical operators have their bitwise counterparts, i.e. the operators `&` and `|`. These operators compare the operands bitwise, i.e. the corresponding binary numbers, bit by bit.

C++ also has a bitwise operator ^ for logical xor, i.e. exclusive or.

Characteristics of basic data types

At first glance, the following two programs will look identical to you:

```
def main():
    number = int(input("Input a number: "))
    if number < 0:
        print("Must be non-negative!")
    else:
        result = 1;
        while number > 0:
            result = result * number
            number -= 1
        print("Factorial is:", result)

main()
```

```
#include <iostream>

using namespace std;

int main() {
    cout << "Input a number: ";
    int number = 0;
    cin >> number;

    if ( number < 0 ) {
        cout << "Must be non-negative!" << endl;
    } else {
        int result = 1;
        while ( number > 0 ) {
            result = result * number;
            --number;
```

```
        }
        cout << "Factorial is: " << result << endl;
    }
}
```

Then again, if you run the programs and set the initial value at, for example, `17`, Python program will print `355687428096000`, the C++ program `-288522240`. The result of the C++ is undoubtedly wrong, because the value of a factorial (the multiplication of non-negative numbers) may not be negative. What causes this problem?

In a way, Python is a rare programming language, because the value of integers is not limited. Most of the other programming languages present integers (and other types of numbers) on machine language level as a certain, fixed amount of bits. This amount of bits usually depends on the processor architecture that is used, but sometimes it also depends on the chosen compiler. In the case of integers, the typical amount of bits is 32, but in some cases, it is 16 or 64.

If the processor of the computer presents integers with 32 bits, it will not be able to process other integers than the ones between -2147483648 and 2147483647 (altogether 2^{32} different integers). If the result of a calculation on that processor is an integer that may not be presented with 32 bits, it happens an **overflow**. The overflow will naturally make the result incorrect.

One could compare overflow to the moment when the odometer (the distance gauge) of your car reaches its highest number, and ”spins around” back to the smallest number it is capable of showing. (Kuva: Ant75)

A similar situation may arise when calculating with real numbers, despite the fact that bit amounts and numerical values are not the same, because real numbers are presented in a different form than integers. **Underflow** means the situation where you count with real

numbers and the absolute value of the result is too small for the processor to differentiate between result and zero.

When a programmer works with C++, they have some power over how numbers are processed. For example, there are several data types meant for presenting integers in C++, while there was only one in Python.

Since the aforementioned trivia is not vital at this point, the data types we will be using on this course are as follows:

- `int` - the normal integer, fit for dealing with positive and negative values
- `unsigned int` - integer type fit for non-negative numbers (natural numbers) only
- `long int` - type that may allow a larger area of presentation than the usual `int`type, but only if the processor architecture of the computer supports it
- `unsigned long int` - a rather logical combination of the two types above.

It is vital to note that C++ language does not define the amount of bits used to present numeric data types. Therefore, you should not trust all of the integers of the `int` type always to be presented by 32 bits, even though this applies to the working environment of our course.

Values and references

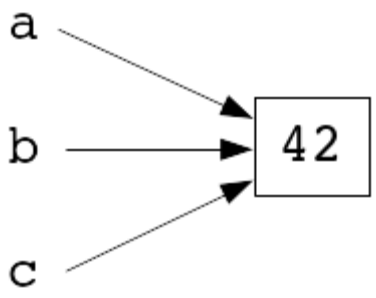
In Python, there are no variables, at least not in their traditional sense.

As well as names and data elements (objects), Python has a mechanism for tying a name to a data element. Basically this means that you can assign a name to a certain data element with the Python command `=`.

On the previous programming course (Programming 1: Introduction), the relationship between a name and a data element was illustrated by presenting the following code:

```
a = 42
b = a
c = 42
```

in a figure more or less like this one:

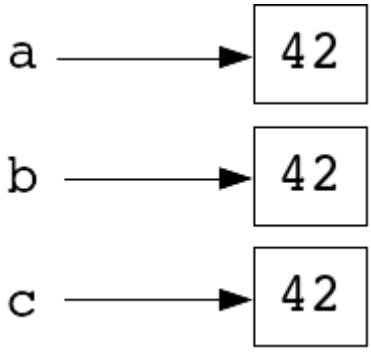


This means that there is only one data element, `42`, which can have one or more names (`a`, `b`, and `c`). The names are used to manage the data element. This way of naming of the data elements is called **reference semantics**.

The mechanism (by default) is different in C++ - the operator (=) creates a separate copy of the data element:

```
int a = 42;
int b = a;
int c;
c = 42;
```

and visually we can present it like this:



So, now the program processes several different data elements which all present the same value, the integer 42.

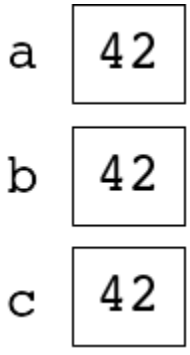
This mechanism, where the processed data element is copied during initialization and assignment, is called **value semantics**.

The visualisation above is not as helpful as you might hope, which is why we dig a little deeper into how value semantics works:

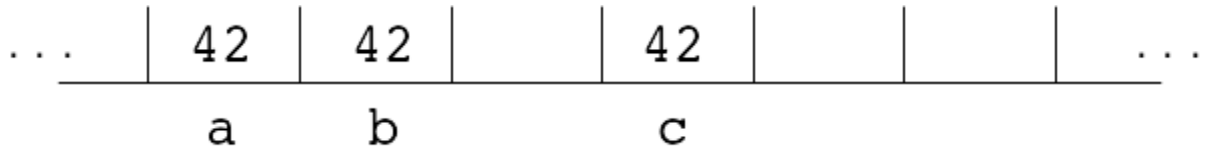
- In programming languages using value semantics, the variable does not mean the name given to a data element, but it should rather be interpreted as *the name given to a certain memory cell*.
- Also, assignment and initialization are interpreted as the replacement of an existing piece of information with a new one within the memory area in question.

The term **variable** is derived from the following fact: the contents of the memory to which the variable gives a name (the value saved in the variable) may change. Considering all that has been said, we might also claim that Python does not really have proper variables, because the only thing that changes during a program's execution in Python programming is which name is linked to which value.

Depending on the situation, a better example of presenting the variables in C++ is one of the following:



or, on a more technical level



in which the slots represent the memory cells.

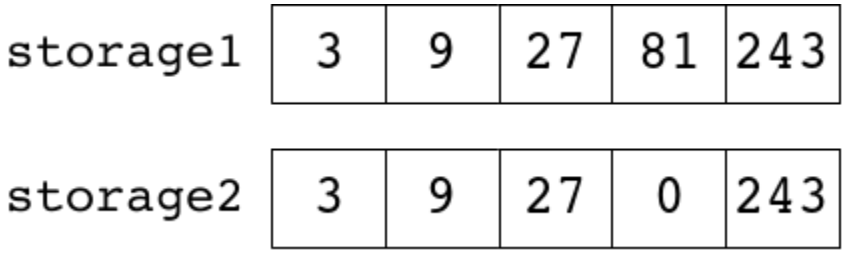
At this point, the question is: Why did we have to think so hard about the difference between reference semantics and value semantics? An example will answer the question. Let's have a look at two programs using a slightly more complex data type. In the Python program, the data type is `list`, and the almost identical corresponding feature of C++ is `deque`, a type from C++ library.

```
def main():
    storage1 = [ 3, 9, 27, 81, 243 ]
    storage2 = storage1
    storage2[3] = 0
    print(storage1[3], storage2[3])

main()
#include <iostream>
#include <deque>

using namespace std;

int main() {
    deque<int> storage1 = { 3, 9, 27, 81, 243 };
    deque<int> storage2;
    storage2 = storage1;   // Copying storage1 to storage2
    storage2.at(3) = 0;  // Indexing with at method (storage2.at(3)) corresponds to storage2
[3] in Python
    cout << storage1.at(3) << " "
         << storage2.at(3) << endl;
}
```

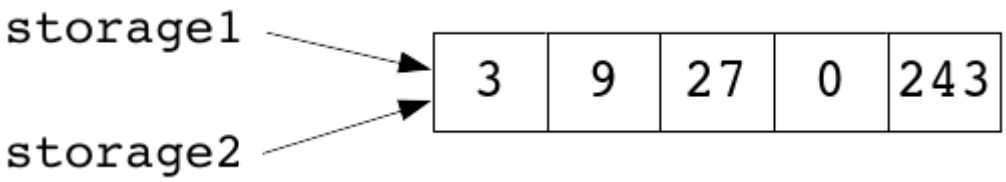
The previous example teaches us that when you create a new `deque` in C++, you need to express the kinds of elements it includes. You tell this in angle brackets. So, in the previous example, the each `deque` contains integers.

Based on everything you now know about value and reference semantics, can you tell how the operation of Python differs from C++? Python prints: `0 0` and C++: `81 0`.

Why? The explanation is that in C++, the assignment `storage2 = storage1` creates a new copy of the `deque` structure. As you touch the copy (`storage2`) and change one of the elements, the structure of the original (`storage1`) is preserved with no changes.



In Python, both `storage1` and `storage2` are just two different names for the same list, and changes made in one of them will be visible in the other as well.



References in C++

C++ also offers a chance to create **references**, that is, to assign different names to the same variable.

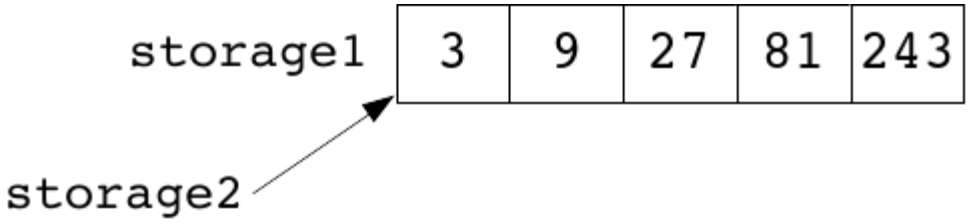
In C++, you can create a reference by inserting the sign & between the name of the data type and that of the variable in the variable definition. You also need to initialize the reference with the variable you wish to be referenced (i.e. the variable you want to have an additional name).

```
target_type& reference_name = target_variable;
```

Another way to implement the previous `deque` structure program is like this:

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
    deque<int> storage1 = { 3, 9, 27, 81, 243 };
    deque<int>& storage2 = storage1;
    storage2[3] = 0;
    cout << storage1[3] << " "
         << storage2[3] << endl;
}
```

This implementation works similarly to the previous Python program, which means it prints `0 0`.



The references of C++ are not very useful in the previous example, but soon we will be using references as the parameter type for functions. This will allow us to use them more purposefully, because the help of a reference enables the programmer to use the original data element instead of a copy.

(Nice to know: The reference operand & may change places: it does not matter whether it is joined to the operator or the operand or divided from each of them by spaces:

```
int integer = 42;
```

```
int& reference1 = integer;
int &reference2 = integer;
int & reference3 = integer;
```

Logically, though, the operand belongs together with the data type, and for clarity's sake, it is worth using the practice presented first.)

Functions

Let us have a look at the following code:

```
def middle_value(number1, number2, number3):
    if number2 <= number1 <= number3 \
            or number3 <= number1 <= number2:
        return number1
    elif number1 <= number2 <= number3 \
             or number3 <= number2 <= number1:
        return number2
    else:
        return number3

def main():
    print(middle_value(4, 5, 1))

main()
```

```
#include <iostream>

using namespace std;

int middle_value(int number1, int number2, int number3) {
    if ( (number2 <= number1 and number1 <= number3)
            or (number3 <= number1 and number1 <= number2) ) {
        return number1;
    } else if ( (number1 <= number2 and number2 <= number3)
            or (number3 <= number2 and number2 <= number1) ) {
        return number2;
    } else {
        return number3;
    }
}

int main() {
    cout << middle_value(4, 5, 1) << endl;
}
```

- The most fundamental differences of the example code snippets are related to the demands that dynamic and static typing place: A C++ programmer needs to explicitly (separately) define the parameter and return value types of a function, so that the compiler may check whether the parameters used to call for the function are of the proper type, and that the return value is used in an acceptable way.
- The actual call notation of a function is identical in both languages:

  ```
  function_name(comma_separated_parameters)
  ```

- Also, the return value of a function is given identically in the two languages, with a `return` command.
- The examples also demonstrate a detail of the relational operators: Python allows you to join relational operators in a chain:

  ```
  if a <= b <= c:
      ...
  ```

C++ requires you to write logical expressions with the and operator:

```
if ( a <= b and b <= c ) {
    ...
}
```

Details of C++ functions

Before calling a function in C++, the compiler must be aware of its existence, the types and amounts of parameters, and the type of the function's return value.

A function can be completely **defined** before you attempt to call it:

```
#include <iostream>

using namespace std;

double average(double number1, double number2) {
    return (number1 + number2) / 2;
}

int main() {
    cout << average(2.0, 5.0) << endl;
}
```

Defining a function means writing the code of a function in its entirety.

You can also **declare** the function before the point of calling it, and define it only after the call:

```
#include <iostream>

using namespace std;

double average(double number1, double number2);

int main() {
    cout << average(2.0, 5.0) << endl;
}

double average(double number1, double number2) {
    return (number1 + number2) / 2;
}
```

If you declare a function, you will write a minimal amount of information about it so that the compiler can check whether or not the function is called correctly. Declaring a function in C++ happens like this:

```
return_value_type function_name(comma_separated_definitions_of_parameter);
```

Of course, you will need to define the function at some point, because function declaration does not include the body of a function, i.e. the commands you want to perform when calling that function.

In C++, if you need to define a function that does not return any value (it is often called **subroutine** or **subprogram**), you can do it by using a special data type called `void`:

```
#include <iostream>

using namespace std;

void print_multiplication_table(int number) {
   if ( number <= 0 ) {
      cout << "Error: only positive numbers are valid!" << endl;
      return;
   }
   int multiplier = 1;
   while ( multiplier <= 9 ) {
      cout << multiplier * number << endl;
      ++multiplier;
   }
}

int main() {
   print_multiplication_table(7);
}
```

A subroutine does not have to have a `return` command at all. Instead, the return happens automatically when the execution reaches the end of the function's body.

Python's dynamic typing means that a single function may basically use parameters of different types, and act in a different way depending on the parameter type. It goes something like this:

```
def funcion(param):
    if type(param) is int:
        # What if the parameter has the type int?
    elif type(param) is float:
        # What if the parameter has the type float?
    else:
        # What to do otherwise?
```

Because C++ uses static typing, an arrangement like the one above will not work, since the compiler must already know the parameter types during compilation.

Then again, there are several C++ mechanisms you can use to create **generic/polymorphic**functions like the one above. **Overloading** is the easiest of these mechanisms to understand. It means the programmer may define several functions with the same name, as long as there are differences in their parameter types and/or amounts. Once an overloaded function is called, the compiler is able to use the given parameters at the calling point to decide which version of the function it should call:

```
#include <iostream>

using namespace std;

double square_of_the_sum(double a, double b) {
    return (a + b) * (a +b);
}

int square_of_the_sum(int a, int b) {
    return (a + b) * (a +b);
}

int main() {
    // Calling the first version (since parameters are double)
    cout << square_of_the_sum(1.2, 3.4) << endl;

    // Calling the second version (since parameters are int)
    cout << square_of_the_sum(3, 4) << endl;
}
```

As we hinted earlier, there are other C++ mechanisms to use at similar situations. You will get to know them in the advanced programming courses (Programming 3: Techniques and Software Design).

Specialities of the `main` function

In C++, the return value type of the `main` function is always `int`. It is customary for the program to end with the return value `EXIT_SUCCESS` if there are no errors. If there are errors in the execution, the return value will be `EXIT_FAILURE`. They are constants defined in the library `cstdlib`. `EXIT_SUCCESS` has the value 0 and `EXIT_FAILURE` 1. Sometimes you see programs where `main` will simply end with the command `return 0`. For the sake of creating readable program code, it is always better to use fixed constants than magic numbers.

In theory, the `main` function should end with the `return` command, because it is of the `int` type. However, the `main` function is an exception. If there is no return statement, the compiler will automatically set `EXIT_SUCCESS` as the return value of the `main` function.

Parameter passing

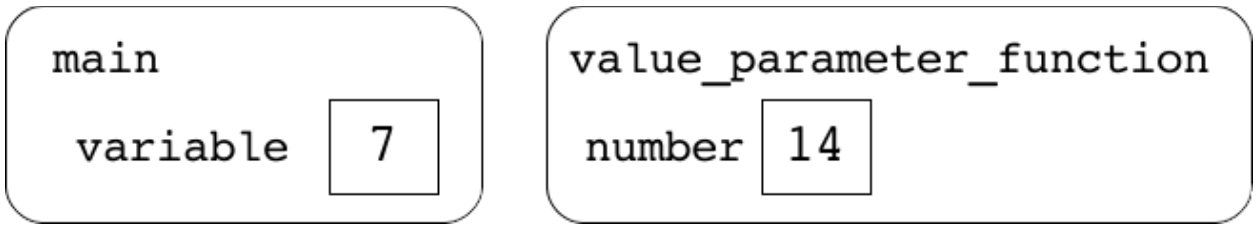
When creating a variable, a C++ programmer has the option of deciding whether they will create a genuine variable or only a reference to an existing variable. Because of this, they have new possibilities also with formal parameters of a function.

Before giving a definition for formal parameters, we need to know what is meant by **actual parameters**. They parameters that are used in function call. **Formal parameters**, instead, are used in function definition. Therefore, they are variables that help access the actual parameters or their values in function body.

Unless the programmer decides otherwise, the function parameters of C++ are **value parameters** by default.

```
#include <iostream>

using namespace std;

void value_parameter_function(int number) {
    number = 2 * number;
}

int main() {
    int variable = 7;
    cout << variable << endl;  // Prints 7
    value_parameter_function(variable);
    cout << variable << endl;  // Prints again 7
}
```

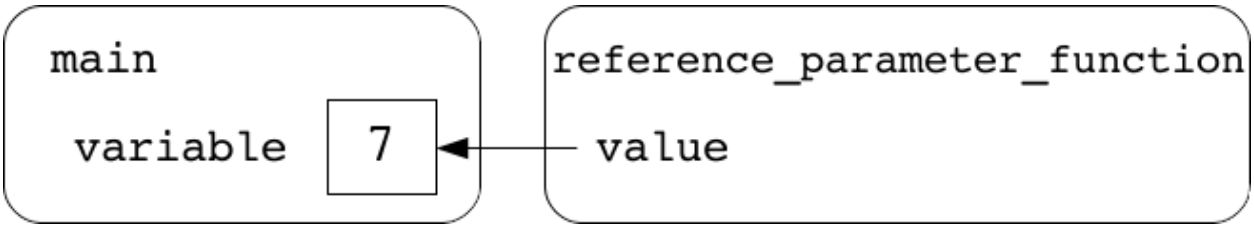
In the code above, the formal parameter called `number` is a new variable. It is initialized to the value of the actual parameter `variable`, which is 7. Although the function above will double the value of its formal parameter, it does not have an effect on the actual parameter, since this is a separate variable.



If desired, a programmer can define some or all of the formal parameters of a function as **reference parameters**.

```
#include <iostream>

using namespace std;

// The only differences in this new version are
// - the name of the function has been changed for clarity's sake
// - ampersand character (&) has been added in the parameter definition.
void reference_parameter_function(int& number) {
    number = 2 * number;
}

int main() {
    int variable = 7;
    cout << variable << endl;  // Prints 7
    reference_parameter_function(variable);
    cout << variable << endl;  // Prints 14
}
```

In the new version, the parameter `number` is a reference parameter, and the changes made to it are immediately reflected in the actual parameter `variable`.



Because of the fact that in C++, a reference gives an additional name to an existing variable (or, more precisely, to a memory area that is already in use), the following code snippet is not correct:

```
void reference_parameter_function(int& number) {
    number = 2 * number;
}

int main() {
    reference_parameter_function(7); // ERROR
}
```

The reason: a literal cannot have references in C++, because it is not a variable.

Without a closer look, one might think that Python's function parameters are value parameters. In reality, all of the "variables" in Python follow reference semantics (which means they are references), including the functions' parameters.

The mistaken conclusion might, in part, be based on the fact that a Python programmer must also remember the concepts of a **mutable** and **immutable data type**, which are Python's mechanism to produce reference semantics for some of its data types. A value of the immutable data type as a function's parameter will seemingly act as if it was a value parameter.

Constant parameters

We had a situation where we wanted to make changes inside a function to the variable passed by as a parameter. Some situations can be the opposite: Within a function, you want to leave the variable passed by as a parameter unchanged (or make it unchangeable). In this case you can define the parameter as constant with the keyword `const`. (We used this keyword already in section 2.3 to define the named constants.)

The keyword `const` will be written before the formal parameter, as demonstrated below:

```
void constant_parameter_function(int const& number) {
    cout << number << endl;
    int variable = number;
}
```

The use of parameters defined as constant is restricted, because the C++ compiler makes sure that the parameter will not be changed. For example, the code snippet you see above prints the parameter and its value is assigned to the local variable, and both of these actions are legal. Below you can see an example of illegal use of a constant parameter:

```
void constant_parameter_function(int const& number) {
    number = 2 * number;  // ERROR
}

int main() {
    constant_parameter_function(7);  // CORRECT
}
```

In the examples above the parameter `number` is a reference parameter. What is the point of it? Just earlier the reference parameters made it possible to change the formal parameter such that the change also affects the actual parameter? When we defined the parameter as constant, wasn't it supposed to prevent this from happening?

Let us first see if the constant parameter should be a value parameter. It is entirely possible, but the only extra benefit would be the compiler reporting an error if we tried to change the value of the parameter. It would not change the situation much, because as we noticed above, changing the value of the formal parameter does not affect the actual parameter.

In that case, what is the point of using a reference parameter as the constant parameter? On this course we have already discussed value semantics and reference semantics. When you assign a value using value semantics, you copy the value, but in reference semantics you simply change the target of the reference to be a different data element. The difference between value and reference parameters is similar. Let us imagine that the parameter is a very large data structure or container (which we will discuss further in the next two rounds). Copying the whole data structure, element by element, is a hard task if you compare it to just taking the reference that targets the beginning of the said data structure, and moving it to take a new target.

Therefore if we want to pass a large data structure by as a parameter, the most reasonable way to do this is to pass it as a reference. If we also wish not to change the parameter in question (even by accident) within the function, the best choice is to define the parameter as constant.

For the reasons mentioned above, the constant parameters are often references. (They can also be pointers, which we will study later as the course advances.)

Strings and characters

In Python, the character string (`str`) is one of the immutable data types. It means that you cannot change a string-type value:

```
def main():
    text = "abcdefg"
    print(text[3])  # Prints d
    text[3] = "X"   # Error!
    print(text)

main()
```

The problem was avoided by creating a new string-type value every time they wanted to edit the text of the string. The new, edited text was inserted to the new string, and the string was named after the original string:

```
def main():
    text = "abcdefg"
    print(text[3])  # Prints d
    text = text[:3] + "X" + text[4:]
    print(text)     # Prints abcXefg

main()
```

In C++, you can change an existing string (except if it is declared as constant, i.e. `const string`):

```
#include <iostream>
#include <string>   // Note: string is a library type in C++

using namespace std;

int main() {
     string text = "abcdefg";
     cout << text.at(3) << endl;  // Prints d
     text.at(3) = 'X';            // Note: char type literal 'X'
     cout << text << endl;        // Prints abcXefg
}
```

Both Python and C++ start indexing the characters in a string at zero.

In C++, the single characters of a string are of the `char` type, which means that the indexing operation `text.at(indeksi)` will always produce a `char` type value, or if the target of the assignment is the indexing operation itself, the value assigned must be `char`.

Operations that target strings are called **methods** in C++, just like in Python:

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

using namespace std;

int main() {
     string text = "One string to rule them all...";
     cout << text.length() << endl;
     cout << text.substr(4, 14) << endl;
     text.replace(4, 6, "thing");
     cout << text << endl;
}
```

Some operations that target strings will return numerical values, such as:

```
text.length();     // Number of characters
text.find("abc");  // At which point can you find the first "abc"?
```

If these numerical values connected to strings (e.g. the amount of characters, indexes) are to be saved in a variable, the type of the variable must be `string::size_type`:

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

using namespace std;

int main() {
     string name = "";
     string::size_type temp = 0;

     cout << "Input your name: ";
     getline(cin, name);

     temp = name.length();
     cout << "Your name has " << temp << " characters" << endl;

     temp = name.find("nen");
     if ( temp == string::npos ) {
          cout << "Your name doesn't contain \"nen\"." << endl;
     } else {
          cout << "Combination \"nen\" is located starting from "
               << temp << endl;
     }
}
```

Please note that `find` returns the value `string::npos` if it cannot find the combination of characters it was looking for.

Some useful string operations in C++

Methods

Methods are called with the notation `variable.method(parameters)`.

- `string::size_type length()`

  The return value tells how many characters there are in a string.

  ```
  string::size_type len = 0;
  len = text.length();
  ```

- `char at(index)`

  Returns the *index-th* character of the string. If calling the `at` function is the object of an assignment, it replaces *index-th* character.

  ```
  char letter;
  letter = text.at(5);
  text.at(2) = 'k';  // Please note the type char
  ```

- `string erase(index)`

  `string erase(index, len)`

If there is only one parameter, it will destroy all the characters in the string, starting at *index*. If the call also has the parameter *len* (length), it will destroy as many characters onward as the parameter *len* describes.

```
text.erase(4);
text.erase(2, 5);
```

- `string::size_type find(target)`

  `string::size_type find(target, index)`

  Starting at the beginning of the string (or at the point *index*), the above function will find the first occurrence of the string *target*, and return the index in question. If *target* cannot be found, the return value is the constant `string::npos`.

```
string::size_type location = 0;
location = text.find("abc", 5);
if ( location == string::npos ) {
  // Could not find "abc" from index 5 onwards
} else {
  // "abc" was found.
}
```

- `string::size_type rfind(target)`

  `string::size_type rfind(target, index)`

  This one is similar to the function `find` earlier, but this will start the search at the end of the string and proceed towards the beginning.

- `string substr(index)`

  `string substr(index, len)`

  Returns a part of the string. If you only have one parameter, the return value will be the substring that starts at *index* and finishes at the end of the string. If you also give the parameter *len* (length), the function will return as many characters as *len* describes onwards from *index*.

```
string text = "abcdefg";
string result;
result = text.substr(3);     // "defg"
result = text.substr(4, 2);  // "ef"
```

- `string insert(index, addition)`

  Adds the text *addition* before position *index*.

```
string text = "abcd";
text.insert(2, "xy");  // "abxycd"
```

- `string replace(index, len, replacement)`

  Replaces the contents of the string with the text *replacement*, starting at *index*for the length of *len* characters.

```
string text = "ABCDEF";
text.replace(1, 3, "xy");  // "AxyEF"
```

Operators

- `text1 == text2`

  `text1 != text2`

  `text1 < text2`

  `text1 > text2`

  `text1 <= text2`

  `text1 >= text2`

  You can compare different strings to each other with relational operators. The operators that include a ”smaller than” or ”larger than” compare the so-called**lexical order**, which for strings means something close to their alphabetical order.

- `text1 + text2`

  Glues (concatenates) two strings into one.

```
string test1 = "Qt";
string test2 = "Creator";
string result;
result = test1 + " " + test2;  // "Qt Creator"
```

- `str += addition`

  Glues (concatenates) an *addition* to the end of the string. The *addition* may be either a string or a single character (`char` type value).

```
string result = "Qt";
result += ' ';        // "Qt "
result += "Creator";  // "Qt Creator"
```

Functions

- `getline(stream, line)`

  Reads one line of text from a file or the keypad (`cin`) and save it in the string-type reference parameter *line*.

- `int stoi(text)`

  Changes the parameter *text* to the corresponding integer.

```
string numeric_text = "123";
int number;
number = stoi(numeric_text);  // 123
```

- `double stod(text)`

  Just like `stoi` above, but this changes the string into a real number.

```
string numeric_text = "123.456";
double number;
number = stod(numeric_text);  // 123.456
```

The problem with the functions `stoi` and `stod` is that they do not recognise a faulty input if the beginning just looks like a value of the right type. For example, the `stoi` function accepts `"123abc"` and returns 123. With the knowledge you have gathered so far, it is not easy to spot this mistake.

Characters in C++

In Python, we processed characters (letters) the same way we do a one character string.

In C++, the type `char` is actually an integer. A `char` variable includes an ASCII value:

```
cout << "Enter a character: ";
char ch = ' ';
cin >> ch;
int ascii_value_of_ch = static_cast< int >( ch );
cout << "ASCII value of " << ch << " is " << ascii_value_of_ch << endl;
```

The fact that the character is actually an integer enables you to perform calculations on the char type:

```
for( char letter = 'a'; letter < 'z'; ++letter ){
    cout << letter;
}
cout << endl;
```

This might be useful if you, for some reason, wish your program to go through all the letters in alphabetical order.

Library `cctype`

In C++, you can use the library called `cctype` (don't forget `include`), which contains the following functions, among others:

- `islower(character)` checks if you have a lowercase letter (return value `bool`)
- `isupper(character)` checks if you have an uppercase letter
- `isdigit(character)` checks if you have a digit
- `tolower(character)` converts an uppercase letter into a corresponding lowercase letter
- `toupper(character)` converts a lowercase letter into a corresponding uppercase letter.

You will find more functions included in the `cctype` library on the Internet in the many C++ library references there are.

Interfaces and object-oriented programming

In programming, the term **interface** means an arrangement that limits the programmer's direct access a part of a program. The programmer may utilize the hidden/limited parts only by means of certain operations that have been defined beforehand. In practice, interface means that the programmer does not need to know exactly how something has been implemented, yet they are still able to use its services.

We can use `string` type variables and the data type `string` as an example here. The average programmer does not have the information or knowledge about the way the `string` type has been implemented in a C++ library. They do not understand the problems that have to be solved in order to create a structure in which you can save a yet-undetermined length of text.

However, anyone who has understood the material of the previous section is able to utilize the `string` type in their program as it contains a group of functions and operators that can be used to complete all the necessary operations. These ready-to-use operations are the (**public**) **interface** of the `string` type.

You can think of the public interface as the kind of interface that defines what can and cannot be done. What could be the meaning of **private interface**?

Interfaces are an essential part of object-oriented programming, and object languages are handy in defining interfaces. C++, for example, offers mechanisms for defining public and private interfaces. Object-oriented programming operates on objects communicating with each other via their public interfaces, and the control flow of the entire program is essentially based on passing messages between objects, and the objects reacting to these messages.

The idea of object-oriented programming will be explained more clearly in the subsections of this section, the first of which will take a look at classes and objects, and compare them to corresponding elements in Python. Next, we will take up pointers, because they are the most sensible way of creating many of the properties that go with object-oriented programming in C++. This will become apparent at a later stage of this course. Only after the abovementioned two theory sections will we be ready to implement our first actual object-oriented program.

Classes and objects

Let us remind ourselves how to define a simple class and how to use it in Python:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def celebrate_birthday(self, next_age):
        self.__age = next_age

    def print(self):
        print(self.__name, ":", self.__age)

def main():
    pal = Person("Matt", 18)
    print(pal.get_name())
    pal.print()
    pal.celebrate_birthday(19)
    pal.print()

main()
```

What is the difference between a **class** and an **object**? A class is a data type, and object is a value or a variable whose data type is a class. Sometimes objects are also called **instances of a class**.

A similar class can be written in C++ like this:

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

using namespace std;

class Person {
  public:
    Person(string name, int age);
    string get_name() const;
    void celebrate_birthday(int next_age);
    void print() const;
  private:
    string name_;
    int    age_;
};  // Note semicolon!

int main() {
    Person pal("Matt", 18);
    cout << pal.get_name() << endl;
    pal.print();
    pal.celebrate_birthday(19);
    pal.print();
}

Person::Person(string name, int age):
```

```
    name_(name), age_(age)  {
}

string Person::get_name() const {
    return name_;
}

void Person::celebrate_birthday(int next_age) {
    age_ = next_age;
}

void Person::print() const {
    cout << name_ << " : " << age_ << endl;
}
```

There are two clear parts in the definition of a class in C++:

- `public`, where you present the public interface of the class, i.e. the **methods**(**member functions**), by which the objects of the class can be operated
- `private`, where you hide the variables (**member variables**, **attributes**) you use to describe the concept implemented as the class.

The member variables within the `private` part cannot be directly accessed from outside the class. If and when you want to use their values, you need to add a method to the public interface of the class, and that method will allow you access to member variables.

The member variables are used in the body of the method just like normal variables, but you do not have to define them separately, because each object has a member variable copy of their own.

Methods can be divided into four categories:

constructor

The constructor function is always named after the class, and is given no return value type.

The constructor is always automatically called when you create a new object. The constructor's job is to initialize the created object.

selector

Selectors are methods that you use to examine but not change the **state of the object** (the values of the member variables).

You can make a method as a selector by adding the reserved word `const` after the final parenthesis of the parameter list. This will prevent any attempt to change the state of the object in the method in question.

mutator

Mutators are such methods that let you change the state of an object, i.e. the values of the member variables.

destructor

The destructor is called automatically when an object comes to the end of its life time. The example program does not have a destructor.

With the exception of the constructor and the destructor, methods are called usually with the notation:

```
object.method(parameteters);
```

The call of a constructor takes place automatically behind the scenes every time you need to initialize a new object. The parameters of the constructor are written in parentheses after the name of the object you want to define. As you define a constructor function, you will initialize the member variables of the object using the **initialization list**:

```
AClass::AClass(parameter1, parameter2):
  attribute1_(parameter1), attribute2_(parameter2) {
}
```

The initialization list is written in the definition of the constructor after the colon and before the constructor's body (curly brackets, {}). You should list all of the attributes of the class in the same order they are in at the class interface, and after that, you should initialize them with initial values.

If a class has a destructor, it will automatically be called at the end of the object's life time. The life time of the local objects ends at the end of their scope (i.e. a program block where an object is visible).

The class is a tool for creating **abstracts** (**concepts**) in a program. The details of how a class has been implemented are hidden from its user, who can only use the class via the public interface.

There will be concepts in the program that are defined by their possible uses: "A person is what you can do to them." Experience has shown that the program becomes clearer if you use concepts defined by their functionality like the one above.

Operator functions

It was earlier listed four categories for methods. In addition to them, there is a special case: operators.

With classes you can define your own types (abstract data types), and for them it is natural to define functions, the meaning of which is similar to some existing operators such as +, +=, ==, etc.

For example, consider a class called `Fraction` for describing fractions. If the class has two integer attributes `numerator_` and `denominator_`, then we can define a function called `operator==` for comparing the equivalence to another fraction as follows:

```
bool Fraction::operator==(const Fraction& other) const
{
    return numerator_ == other.numerator_ && denominator_ == other.denominator_;
}
```

when assuming that the fractions are as reduced as possible.

After that we can compare two fractions in the same way as any numeric types:

```
Fraction f1(2, 3);
Fraction f2(3, 4);
```

```
if(f1 == f2) ...
```

when assuming that the constuctor of `Fraction` takes two parameters: one for the numerator and the other for the denominator.

How to implement a class in a separate file

In the example above, we implemented the class in the same file as the main program, just like we always did with Python programs on the previous course. Let us now create a new version and separate the class into two files. Let us first have a look at the renewed contents of `main.cpp`:

```
#include <iostream>
#include "person.hh"

using namespace std;

int main() {
    Person pal("Matt", 18);
    cout << pal.get_name() << endl;
    pal.print();
    pal.celebrate_birthday(19);
    pal.print();
}
```

We will notice that when the class definition was removed, the `include` directive was added to the beginning of the file:

```
#include "person.hh"
```

This is the way how a class that was implemented elsewhere can be used in the main program. Now, let us have a look at the contents of the file in question, i.e. the file `person.hh`:

```
#include <string>

using namespace std;

class Person {
  public:
    Person(string name, int age);
    string get_name() const;
    void celebrate_birthday(int next_age);
    void print() const;
  private:
    string name_;
    int    age_;
};  // Note semicolon!
```

The file with the extension `.hh` is a so-called **header file** or a definition part. Here we only tell what kind of a class we are talking about. As you can see, the file does not have the implementations of class methods.

The method implementations are in the corresponding implementation file `person.cpp`:

```
#include <iostream>

#include <string>
```

```
#include "person.hh"  // Obs! Implementation file includes the corresponding header (defin
ition) file!


using namespace std;


Person::Person(string name, int age):
    name_(name), age_(age)  {
}


string Person::get_name() const {
    return name_;
}


void Person::celebrate_birthday(int next_age) {
    age_ = next_age;
}


void Person::print() const {
    cout << name_ << " : " << age_ << endl;
}
```

At this point, you might ask how the method implementations will be a part of the program, considering the file `main.cpp` uses the directive `include` to only include the header file `person.hh`. The file `person.cpp` needs to be added to the program at the compiling phase. We will get into this in the next part called ”Creating and compiling a class in Qt Creator”.

In C++, it is customary to divide a class into two parts: the definition and the implementation. They are written in their separate files. Therefore, for each class you will write two files:

- The definition part includes the definition of the class and is saved in a file with the extension `.hh`, named after the class but with a lowercase first letter. See the file `person.hh` in the example above.
- The implementation part includes the implementations of the methods of the class and is saved in a file with the extension `.cpp`, named after the class but with a lowercase first letter. See the file `person.cpp` in the example above. (The terminology can be a bit confusing. We mentioned above ”the implementations of the methods”. Referring to earlier discussion, these are actually (almost) the same as ”function definitions”, as opposed to ”function declaration”.)

When you use the C++ libraries with the directive `include`, you use angle brackets (<>). When you use your own files with the directive `include`, you write the filename within quotation marks (””).

Creating and compiling a class in Qt Creator

When you want to add a new class to an already existing project in Qt Creator, as you choose ”New File or Project,” you can click on ”C++” and ”C++ Class” in the next window. This makes Qt Creator automatically create both of the files you need (`.cpp` and `.hh`). In addition, Qt Creator will automatically add the new implementation file among the files that will be compiled in the project.

If you wish, you can have a look at the project file (with the extension `.pro`) in Qt Creator and see the point `SOURCES`, which tells the compiler which files to include in compilation. This will update automatically when you create a class in the abovementioned way. As you can see, the point `SOURCES` contains only implementation files, **not** header files.

We will consider compilation more precisely in the context of modularity. It will also be the time to become more acquainted with the phases of compilation. For the beginning of the course, you can happily let Qt Creator worry about the compiling automatization.

The class Clock

Let us implement a second example class `Clock` in two different versions. The first version contains nothing drastically new compared to what you have learned earlier. It is supposed to serve as an introduction to the next example. We will compare the first and the second version with each other.

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

using namespace std;

class Clock {
  public:
    Clock(int hour, int minute);
    void tick_tock();  // Time increases with one minute
    void print() const;

  private:
    int hours_;
    int minutes_;
};

int main() {
    Clock time(23, 59);
    time.print();
    time.tick_tock();
    time.print();
}


Clock::Clock(int hour, int minute):
    hours_(hour), minutes_(minute) {
}
```

```
void Clock::tick_tock() {
    ++minutes_;
    if ( minutes_ >= 60 ) {
        minutes_ = 0;
        ++hours_;
    }
    if ( hours_ >= 24 ) {
        hours_ = 0;
    }
}

void Clock::print() const {
    cout << setw(2) << setfill('0') << hours_
         << ":"
         << setw(2) << minutes_
         << endl;
}
```

The most important thing to remember from the first version is how the method `print` uses the operations from the library `iomanip` to format the print layout. Let us not worry about them.

In the second version, we will change the class `Clock` a little. First, please note the attributes.

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

using namespace std;

class Clock {
  public:
    Clock(int hour, int minute);
    void tick_tock();
    int fetch_hour() const;
    int fetch_minutes() const;
    void print() const;

  private:
    // Minutes since the previous midnight
    int minutes_since_midnight__;
};

int main() {
    Clock time(23, 59);
    time.print();
    time.tick_tock();
    time.print();
}

Clock::Clock(int hour, int minute):
    minutes_since_midnight__(60 * hour + minute) {
}

void Clock::tick_tock() {
    ++minutes_since_midnight__;
    if ( minutes_since_midnight__ >= 24 * 60 ) {
        minutes_since_midnight__ = 0;
    }
```

```
}

int Clock::fetch_hour() const {
    // When you divide an integer by an integer,
    // the result is a rounded down integer.
    return minutes_since_midnight__ / 60;
}

int Clock::fetch_minutes() const {
    return minutes_since_midnight__ % 60;
}

void Clock::print() const {
    cout << setfill('0') << setw(2) << fetch_hour()
         << "."
         << setw(2) << fetch_minutes()
         << endl;
}
```

The new version has some interesting changes when compared to the original version.

- The example shows one of the good characteristics of classes (or rather, interfaces): The private interface (implementation) of the class has been changed altogether, but because the public interface was kept compatible with the original, there is no need to change the code that utilizes the class (`main`).
- The method `print` no longer has a direct reference to the member variable of the class; instead, the time to be printed is found with the help of two new methods, `fetch_hour` and `fetch_minute`. This means that the method `print` no longer needs to know the format of the time in the `private` part.

  Therefore, a new (informal) interface has been created within the class: Some of the methods of the class do not operate with the member variables directly but instead through the methods `fetch_hour` and `fetch_minute`.

  The benefit of this solution is that if you change the implementation of the class (meaning the `private` part and the methods operating on that part), you do not have to touch the implementation of the `print` method as long as you take care that `fetch_hour` and `fetch_minute` work the same way as previously.
- Please note that you can use the method to call another method of the same class: The notation of the call is the same as when you call a normal (non-method) function:

  ```
  method(parameters);
  ```

  The above kind of method call will target to the same object you used before the full stop when you called the original method.

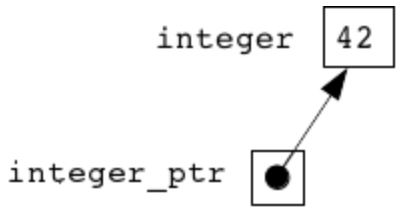
The benefits of using classes

When you use classes to implement concepts in a program, you will nearly always achieve some benefits:

- It becomes possible to change your implementation in the private interface, while the public interface will stay compatible with the earlier implementation.

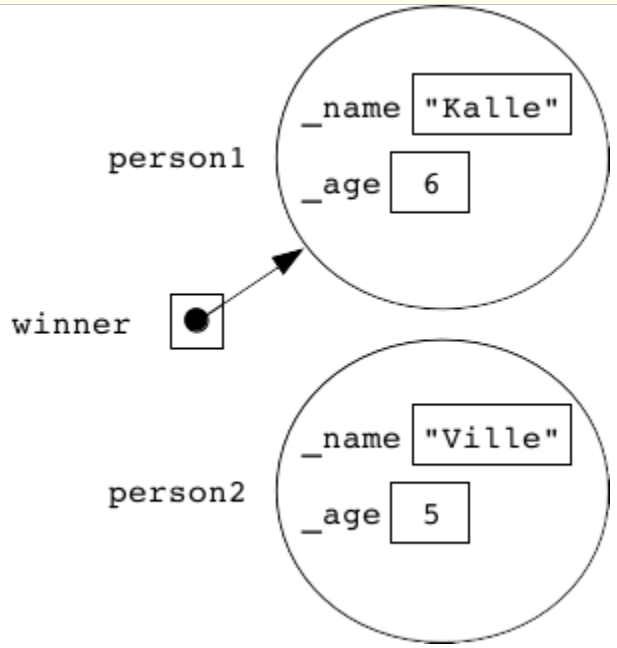
- You can be sure about the **integrity of data**. Constructors and mutators will take care that the object does not receive erroneous values.
- The classes make the program clearer, more understandable and easier to maintain.
- Often, classes are **reusable**.
- Classes help you to control the complexity of a program, because they allow you to combine logical parts of the program.

In fact, these benefits do not only come with classes; all the other mechanisms that allow you to create clear interfaces in your program provide the same benefits as well. For example, a function also forms an interface. As long as you understand what parameters you are supposed to give to the function and which value it will return, you do not need to know anything about the implementation. Therefore, all of the benefits mentioned above are also true of functions.



So, **a pointer is a variable that stores a reference**. Because a pointer is a variable, you can also change the value it points to:

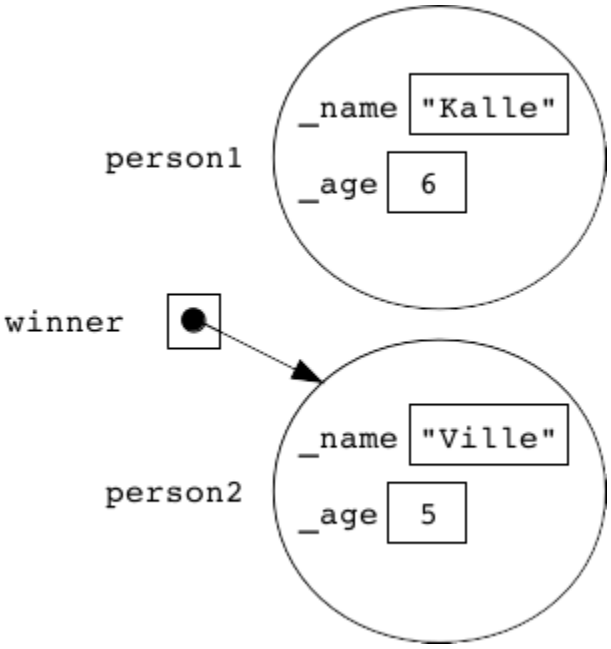
```
Person person1("Kalle", 6);
Person person2("Ville", 5);
Person* winner = &person1;
```



The pointer `winner` now points to the object `person1`. But after executing the next line of code:

```
winner = &person2;
```

the same pointer has turned to point to the object `person2`.



If we want the pointer to point nowhere, we can also set it to `nullptr`.

This is only an introduction to the concept of pointers, and this is not meant to be a comprehensive set of learning material on this matter. The next exercise includes a finished main program that contains one pointer-type variable. Even though you do not need to make any changes to the main program, please go and examine the main program as an example of pointer use.

Pointers will be needed for several different purposes on this course, and later on, we will delve deeper into the details of using them.

From the list, choose branch master, just below the newly added remote repository (the line with master, but do not confuse it with another master line). Click ”Merge” (if this takes you to a menu, choose ”Fast Forward”). If your repository is in a state, in which it should be, you should now see the updates in the structure of the directory. In this case, you should see new rounds in the directory `templates`.

If you have, for example, changed the content of the directory `templates`, it is possible that the above operation does not succeed. In such a case, contact course personnel.

The above action is needed whenever course staff has delivered new material in `template_project`, typically at the beginning of each round.

Choice 2: command line

Setting a remote repository in command line

Open command line and move to your local repository. Execute the following command:

```
git remote add <shortname> <url>
```

where <shortname> is a short, descriptive name for the new remote repository and <url> is the address of if. For example:

```
git remote add course https://course-gitlab.tut.fi/prog2-2019-SPRING/template_project
```

To test how the above command succeeded, write:

```
git remote –v
```

You can see a list of all remote repositories, for example:

```
course https://course-gitlab.tut.fi/prog2-2019-SPRING/template_project (fetch)
course https://course-gitlab.tut.fi/prog2-2019-SPRING/template_project (push)
origin https://course-gitlab.tut.fi/prog2-2019-SPRING/123456.git (fetch)
origin https://course-gitlab.tut.fi/prog2-2019-SPRING/123456.git (push)
```

If everything goes well, you need to do the above kind of remote addition only once during the course.

Retrieving new materials from the remote repository in command line

After setting a remote repository, you can retrieve updates from it to your local repository by writing:

```
git pull <remote> <branch>
```

where <remote> is the name of the remote repository that you gave above, and <branch> is the branch of the remote repository, from which you want to retrieve the updates.

Note that the above command starts a text editor set for Git (vi by default). In the text editor, you will write a merge message describing why you have merged. This happens in the same way as writing commit messages, which we have learned earlier. (If you happen to end up in vi editor, you can exit from there by writing ”:wq”.)

For example, the command described above can be executed as:

```
git pull course master
```

We have not considered Git branches in this course, they are mainly material of the next programming course. The only branch so far is master.

The above action is needed whenever course staff has delivered new material in `template_project`, typically at the beginning of each round.

Alternatively (if you do not want to write a merge message in an editor) you can, instead of pull, do the same action in two phases as:

```
git fetch <remote> <branch>
git merge <remote>/<branch> -m "Write a message here"
```

From above, you can conclude that git pull is actually the same as git fetch + git merge.

Towards C++ data structures

C++ does not include the more complicated data structures and their operations in the language itself. Instead, they are implemented in libraries. The combination of these libraries is usually called **Standard Template Library (STL)**.

STL is a combination of three logical parts:

- **containers**, which means data structures similar to Python's structures `list`, `set`and `dict`, among others that are not included in Python
- **iterators**, which you can think of as certain kind of bookmarks where you save the location of a single data element within a container
- **algorithms**, which provide ready-to-use mechanisms for executing basic operations to containers (e.g. using `sort`, you can sort the elements of a container in the order of your choice, be it smallest to largest or something else).

STL is an extremely large collection of libraries. This course will only provide an all-round glance at it. The course TIE-20100 Data structures and algorithm studies STL more in-depth.

For example:

```
vector<int>::size_type number_of_scores = scores.size();

...

number_of_scores = scores.size();
```

You can access the first and the last value of an element:

```
cout << "first: " << scores.front() << endl
     << "last:  " << scores.back() << endl;
```

The individual values of a vector can be accessed the same way as the single characters of a string:

```
cout << scores.at(3) << endl;
...
scores.at(index) = new_score;
```

Indexing starts at zero, and the index of the last element is the number of the elements minus one.

If you try using the method `at` to index an element that does not exist from the vector, you will get an exception that will crash the program if it is not handled.

If you use the vector as a parameter of a function, it can be either a value or a reference parameter as usual:

```
void function1(vector<double> measurements);
void function2(vector<double>& measurements);
```

Large data structures should usually not be passed to the function as a value parameter if it can be avoided. (Think about what happens in passing a value. You can draw a picture, or you can go back to the section 3.4 of this material.) For efficiency reasons, you should use `const` references instead of value parameters:

```
void function3(const vector<double>& measurements);
```

The keyword `const` together with the reference makes it impossible to change the target of the reference. This makes passing by a reference parameter safer: The caller of the function knows that the function will not make any changes to the variable passed as a parameter.

If you want, you can set the amount of elements in the vector during the vector definition:

```
// A vector with room for 20 integers at the start.
// Integers themselves not initialized.
vector<int> numbersA(20);

// A vector with room for 10 real numbers,
// each of them initialized to have the value 5.3.
vector<double> numbersB(10, 5.3);
```

Please note that there are multiple different initializing notations for vectors (and other STL structures). Below you can see some examples:

```
vector<string> namesA(5);
vector<string> namesB(10, "unknown");
vector<string> namesC = { "Matti", "Maija" };
```

The STL vector is a data structure that keeps the elements in the order they were stored. In STL terminology, structures like these are called **sequences**. There are several different sequences in STL:

- The vector is great for storing multiple data elements for later processing, especially if you are not going to search elements from the stored data, or if the speed of the search is not significant. Naturally, vector is a good choice when you have to be able to process the elements in the same order they were added into it.
- We have earlier mentioned `deque` that is very similar to `vector`, but less efficient. However, `deque` has the benefit that you can add elements to its beginning with the method `push_front` and remove them from the beginning with the method `pop_front`.
- There is also `list` which, despite the name, has nothing in common with the Python list structure, and you should not use `list` if you are not familiar with dynamic memory management.

You will learn more about the differences of containers on the course TIE-20100 Data structures and algorithms, where their efficiency, among other things, will be discussed.

For troubleshooting

When using STL containers in the program code, you should add the following line into your Qt Creator project file:

```
QMAKE_CXXFLAGS += -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC
```

Having added it, you will get more warnings from the compiler about suspicious code on STL containers (such as out-of-bounds indexing), which usually helps in detecting errors.

The personnel of the course have added the line in question into the project files of the assignments that use STL containers.

C++'s `for` loop

There are basically three ways to use the `for` loop in C++:

1. Going through the elements in a container

```
vector<int> numeric_vector;
...
for ( int vector_element : numeric_vector ) {
    cout << vector_element << endl;
}
```

You should basically know this usage already. It is an equivalent of the Python structure

```
for element in list:
    print(element)
```

7. Going through the interval of chosen integer values:

```
8. for ( int number = 5; number < 10; ++number ) {
9.     cout << 9 * number << endl;
10.}
```

which is a more complex equivalent of the Python structure:

```
for number in range(5, 10):
    print(9 * number)
```

11. Creating an infinite loop:

```
12.for ( ;; ) {
13.    ...
14.}
```

that works precisely like this structure

```
while ( true ) {
    ...
}
```

Going through the elements in a container

Let us have a look at the first use of the `for` loop:

```
for ( int vector_element : numeric_vector ) {
    cout << vector_element << endl;
}
```

Here, the `vector_element` is a kind of ”parameter.” Passing by value is the default, and changing the `vector_element` does not change the value stored in the container. If you want to modify the elements in the container, you should write:

```
for ( int& vector_element : numerical_vector ) {
    vector_element = vector_element * 2;
}
```

that is, to pass by reference.

Please note that you cannot add elements to or remove them from the above kind of container in the body of the `for` loop.

At this point, it is useful for you to get to know the C++ keyword `auto`, which deducts the data type. For example, the following line:

```
int i = 42;
```

could also be written like this:

```
auto i = 42;
```

because the compiler can deduce from the assigned value that the variable type is `int`. Although in this case, it is better to use the word `int` because it is more informative for a human reader, and the word `auto` would not even shorten the code.

There is a simpler way to express the earlier `for` loop:

```
vector<int> numerical_vector;
...
for ( auto vector_element : numerical_vector ) {
    cout << vector_element << endl;
}
```

It is handy, especially when the elements in the container have a complex data type.

The interval between chosen integer values

The second usage type shows us that the `for` loop looks much more complicated in C++ than in Python:

```
for ( int number = 5; number < 10; ++number ) {
    cout << 9 * number << endl;
}
```

There are three parts within brackets, each separated with a semicolon, and you can name them, for example, in the following way:

```
for ( initialization; condition; increment ) {
    body
}
```

Of the three parts, initialization is executed only once - when you enter the loop. The value of the condition is always checked before executing the body of the loop. It decides whether the body will be executed again. Increment is always executed last after executing the body.

Nice to know: If you want to go over a sequence, you can write:

```
for( int i : { 2, 3, 5, 7, 11, 13, 17 } ) {
    ...
}
```

where `{ 2, 3, 5, 7, 11, 13, 17 }` is the initialization list.

ntence like this:

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

enum PostalAbbreviation {AL, AK, AZ, AR, CA, CO, ERROR_CODE};  // Excluded the rest 44 eleme
nts

struct StateInfo {
    std::string name;
    PostalAbbreviation abbreviation;
};

const std::vector<StateInfo> STATES = {
    { "Alabama", AL },
    { "Alaska", AK },
    { "Arizona", AZ },
    { "Arkansas", AR },
    { "California", CA },
    { "Colorado", CO }  // Excluded 44 lines
};

// Version 2: Better solution
PostalAbbreviation name_to_abbreviation(std::string name)
{
    for(auto s : STATES) {
        if(name == s.name){
            return s.abbreviation;
        }
    }
    return ERROR_CODE;
}
```

The difference between these two examples is clear: After going through a little bit of trouble to design and initialize a suitable data structure, the actual function `name_to_abbreviation` was decreased significantly. The technique of data driven programming is always - in one way or another – the use of pre-processed information comprised of initial values and the results from them.

The benefits of data driven programming include:

1. The programs created are usually shorter, but still clear.
2. The possibility of making errors is smaller.
3. The programs are easier to expand and maintain.

A clear sign of a need for data driven programming within a program is an `if` structure that grows out of proportion (or a `switch` structure, which we have not studied so far.)

Waterdrop game, the first version

Let's study a bit larger program comprising both objects and use of vectors. We will implement a game working like this:

http://media3.gamesbox.com/games/files/splash-back.swf

The game has a game board with squares, some of which are empty and the other ones have waterdrops of different sizes. The goal is to clean the board out of water. The player has a water tank, and they can drip water to some of the squares in order to increase the size of the waterdrop. The maximum size of a waterdrop in a square is 4 droplets. If you drip water more than that to a square, the drop will pop into four parts and splashes go into four directions. The splashes proceed until they reach the border of the game board and fall away or hit another drop and increase the amount of water of that drop with one droplet. The goal, i.e. an empty game board, can be reached by adding water to suitable squares to make drops to splash out of the game board.

Pull program code from your Git central repository. You can find code of the game implemented with simple ASCII graphics from the directory `examples/04/waterdrop_game_v1`. Let's study the implementation.

Varoitus

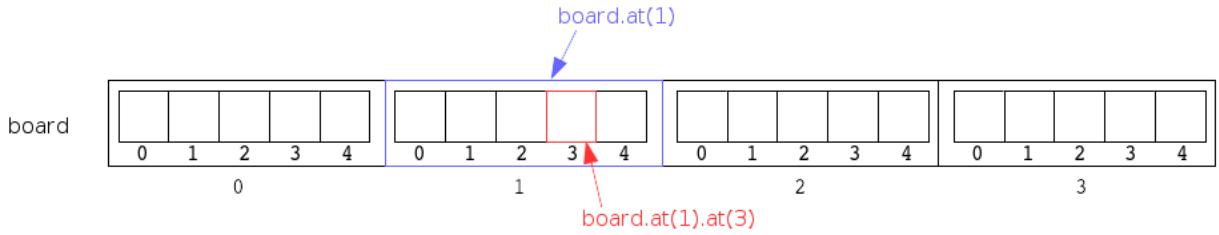
During exploring the code you may want to edit it. Experimenting is usually a more efficient way to learn than just reading. Recall that you must not change the code in the `examples` directory. Otherwise you cannot pull other examples from the repository in sequel.

Instead, for editing, copy the directory `examples/04/waterdrop_game_v1` to the directory called `student/04/waterdrop_game_v1`, and open the latter one in Qt Creator.

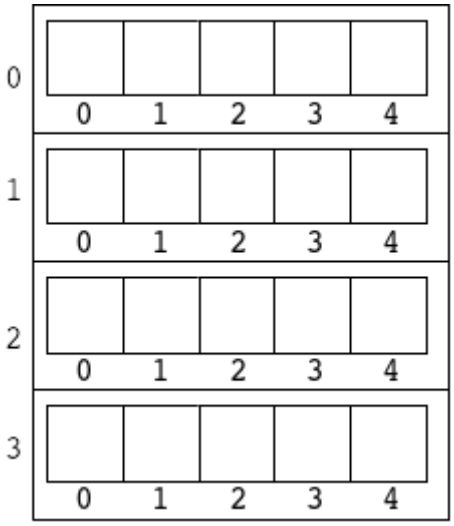
It is perhaps a good idea to run the program once before reading and exploring the code more precisely. When the program prints the prompt `x, y>`, the user is expected to give the coordinates of the square, in which they want to drip water. This is not so fluent as with a graphical interface, but we start in this way. The origin of the game board is on the left top corner, and you can see X coordinates on the top and Y coordinates on the left.

Data structures: game board and water drops

The game board is a two-dimensional grid that can be implemented as a two-dimensional `vector`.



Recall from the previous programming course how to unify data structures and how to refer to the elements of such structures.



If desired the same figure can be drawn in another order that can be easier to understand (it prevents to confuse X and Y coordinates). Each time you feel it difficult to understand a data structure, take a pen a draw a picture about it.

The squares of the game board contain waterdrops. A waterdrop includes data, e.g. how many droplets of water it has. A waterdrop has actions as well, you can, for example, add water to it when it grows or pops. Since a waterdop has both data and actions, we can implement it as an object.

Our game board is a two-dimensional `vector`, the elements of which are objects. However, the game board has squares that are empty of water. Is it possible to have a waterdrop object that has no water? For this reason, it is more natural to name this object as `Square` instead of `Waterdrop`. We have now a two-dimensional `vector`, the elements of which are squares that can be either empty or contain water:

```
class Square { ... };
std::vector< std::vector< Square > > board;
```

Since we need to refer to this data structure several times in the program, we define:

```
using Board = std::vector< std::vector< Square > >;
```

With this definition we can avoid repeating `std::vector< std::vector< Square > >`, which makes the program clearer and shorter. But where to write the above `using` clause? We will answer to this question by exploring the file division of the program.

Program files

The program is divided into three files as follows:

- `main.cpp` comprises the main function and utility functions used by the main function.
- `square.hh` comprises the interface of the class and program-wide definitions.
- `square.cpp` comprises the implementations of the methods of the class.

At this point, let's consider, if it were more clear to put the program-wide definitions to a `.hh` file of their own instead of `square.hh`. In a way, this would be a better solution, because the main task of `square.hh` is to give the definition of the class.

On the other hand, recall that a square is a part of the game board, and the game board consists of squares, and thus, these two concepts are very closely related to each other. Due to this, aforementioned `using` clause can be put in `square.hh`.

In addition, the program is rather small, and thus, there need not be very many files. In larger programs with longer files, a separate file for the above purpose would be a good solution.

Explore also the `#include` directives of the files. You can notice the same practice, used also in previous examples, that `square.hh` has been included in both `.cpp` files, since its definitions are used these files.

Forward declation of a class

When reading the files, you can notice that before the `using` clause, `square.hh` has the line:

```
class Square;
```

This is a forward declaration of class `Square`. It is needed, because the line:

```
using Board = std::vector< std::vector< Square > >;
```

cannot be compiled, if the compiler does not know what `Square` is. On the other hand, the interface of `Square` class cannot be compiled, if the compiler does not know what `Board` is. In other words, `Square` should have been defined before `Board`, and also vice versa.

This problem can be solved by using forward declarations. Forward declaration of `Square` tells to the compiler that `Square` is a class. This is enough information for the compiler to enable it to compile the `using` clause.

Class `Square`

Attributes

Clearly, a square object must know, how many droplets of water it contains.

In addition, each square object should have access to other square objects, because it adds water to other squares when popping. For this reason, each square object has a pointer to the game board, in which it lies. Through the pointer, the square object has access to anything lying on the board.

Square object needs to deal with those squares that are either vertically or horizontally adjacent to it. Therefore, the square object must know its own location. With its own indices, it can index the game board, to which it has access through the aforementioned pointer.

Search for the definitions of the attributes declared above from program code.

Methods

In addition to the constructor and destructor, the square object comprises the following actions:

- A square must know how to print itself, when the whole game board will be printed.
- It must be possible to add water to a square.
- The waterdrop in a square pops, when the amount of water grows over four droplets.
- It must be possible to ask from a square, if it contains water. This information is needed, if a waterdop pops and splashes its water to other squares.

Find out the definitions of these methods from the interface of `Square`.

Printing, adding water, and asking the amount of water are clearly such methods that process a square object from outside the class. Therefore, these methods can be found from the `public`interface. For example, when the main function drips water from the tank to the game board, it calls `addWater` method of the square object with certain coordinates. As an other example, when a square object about to pop looks for the nearest square having water, it uses `hasWater` method of other square objects.

However, `pop` is not a method that can be called by anyone. Popping happens only when the amount of water of a waterdrop exceeds four droplets. Its execution is based on the internal state of the square object. Therefore, `pop` method can be put in the `private` interface, and the object itself can call it at the right time from another method.

Referring to methods with operators `.` and `->`

When exploring the program more precisely, you can notice that the methods of the game board vector is sometimes called in a good old way as:

```
object.method(parameters);
```

Such a call can be found e.g. in the loop structure in `main` function:

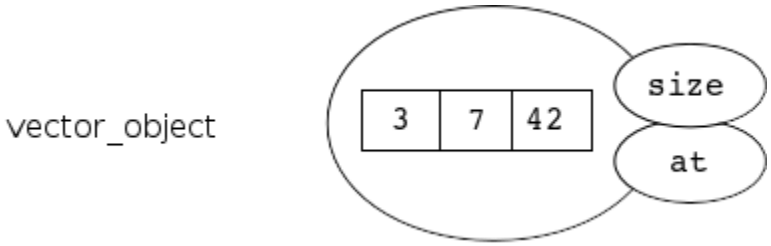
```
board.at(y-1).at(x-1).addWater();
```

But sometimes a different way of calling is used:

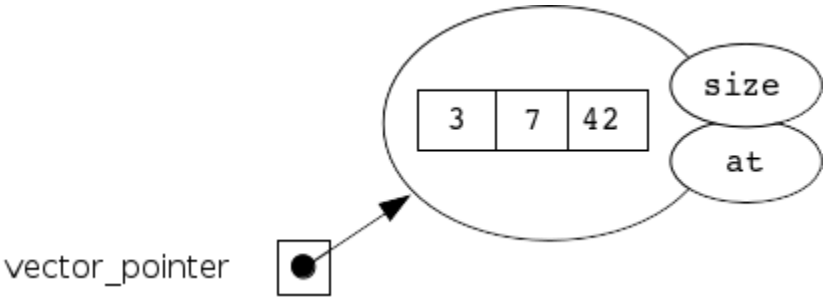
```
object->method(parameters);
```

Such calls can be found e.g. in pop method.

The reason for the differences is that in some parts of the program we handle an object directly, and in some other parts we handle an object through a pointer. The main function has a vector object, but square objects have pointer to a vector object.



When we use a named object itself, we call methods with the notation `object.method(parameters)`, e.g. `vector_object.size()`.



When we use an object through a pointer, we call methods with the notation `object->method(parameters)`, e.g. `vector_pointer->size()`.

Qt Creator guides you to use the right notation. Try this by moving the cursor to one of the method definitions of `Square` writing to a new line `board_.` and see what happens.

Main function and other functions

To avoid the main function to grow too long, file `main.cpp` has utility functions

- to initialize the game board with waterdrops of random sizes
- to print the game board
- to check, if the game board is empty of water, i.e. if the game is over
- to read user input command, i.e. the coordinates, to which to drip water.

By using (calling) the above functions, we can keep the main function rather short and easy to read. Explore the main function. Is the code easy to understand?

The most difficult section in main function is probably the condition of `while`, the first part of which is the value of integer variable `waterTank`. What happens, if we use an integer variable in the place of a boolean value? Compiler makes an *implicit type conversion (coersion)* from boolean to integer. As well, we could write `waterTank != 0` as the condition.

Next explore function `initBoard`

The library `random` comprises the class `default_random_engine`, which is one of the random number generators provided by the class. When you call the constructor of the random number generator class (i.e. create a random number generator object), you can give so called seed value as a parameter. If you do not give a seed value at this phase, you can give it later by calling the function `seed`, as has been done in the code of the waterdrop game. After giving a seed value, you must tell, which distribution you want to use. The library `random` provides several distributions, and we will use one of them called `uniform_int_distribution`. In addition, we must tell to the distribution the interval, from which the random numbers will be taken.

Random numbers generated by the above kind of random number generator are so called *pseudorandom numbers*. It means that the same seed value generates always the same series of random numbers. You can test this by executing the following lines of code several times:

```
default_random_engine gen(42);
uniform_int_distribution<int> distr(1, 100);
std::cout << distr(gen) << std::endl;
std::cout << distr(gen) << std::endl;
std::cout << distr(gen) << std::endl;
std::cout << distr(gen) << std::endl;
std::cout << distr(gen) << std::endl;
```

Pseudorandom numbers are useful for testing purposes, because they allow you to repeat the same execution for several times. This requires only to know the seed value used.

The function `initBoard` asks for the seed value. If the user gives no seed value (but presses enter instead, when an empty string is the input), the random number generator will be initialized with the number read from computer clock. If the user gives a seed value, then it will be used. When the submission system of the course tests automatically such a program that uses random numbers, the program must be implemented such that automatic tester can choose the seed value.

The function `stoi` converts a string to an integer. We have discussed this earlier.

The game board is initialized such that new square objects are created in two nested `for` loops. The amount of water of a square object is a random number between 0-4. The created square objects are first stored to vector `row` (inner `for` loop). When `row` vector is full, it is stored to the vector describing the whole game board (outer `for` loop).

Initializing the game board by using random numbers is not necessarily the best solution, from the point of view of the usability of the game. It is possible to get such a game board that you can win by dripping just one droplet of water to the right position of the game board. If you like challenges, you can develop a better algorithm for initializing the game board with the possibility to choose the level of difficulty. (For example, the total amount of water must always be the same. Winning the game becomes more difficult at higher levels.)

Next explore function `printBoard`

Function printBoard prints the numbers in X and Y axels, empty spaces, and new lines. It calls printing function of square objects, when each square prints itself, i.e. prints the correct character based on the state of the square.

As the numbers of X and Y axels, only the remainder after division by ten is printed. In this way, we can use the same space (one character) for all numbers in the ASCII user interface. You can try to increase the size of the board, and see what happens.

Most error-prone property in the program is that the printable coordinates start from 1, but indexing a vector starts from 0. You can see the effect of the property in this function.

There would be a little less details in this function, if it were implemented without printing empty spaces. However, such a solution would make the ASCII user interface even more uncomfortable to use.

Why on earth the implementor of the function has used output stream as a parameter? Maybe the reason is that in some situation printing could be done to another output stream than `std::cout`. We will consider streams more precisely later.

Next explore function `readCommandSuccesfully`

This function makes only minimal checks for the validity of input values. It checks if the given coordinates belong to the interval defined previously.

Control-C quits the program.

Conclusions

The program is a good example both of storing objects to a vector and of object-oriented programming. The most essential logic of the game is implemented such that square objects communicate with each other, for example, by adding water to other squares.

While exploring the program we have learnt new concepts such as forward declaration of a class and `using` clause in defining a new type. In addition, we have seen how to handle objects through pointers.

The game does not act exactly in the same way as the original one (the game behind the link). Namely, in our game, when a waterdrop pops, it adds water to other drops immediately. In the original game, a popped waterdrop produces splashes that moves the faster the closer the other waterdrops lie. We will return to such implementation, after we have gained more knowledge.

Code layout

Although C++ does not require indentations, use them to improve the readability of the code.

The body of control structures must be enclosed with curly brackets, i.e. `{ }`, even if the body consists of a single statement. Curly brackets can be written in the lines of their own, or the

starting curly bracket, i.e. `{`, can be written at the end of the preceding line. You can use either of the above ways, but use the chosen way consistently in the whole program.

Object-oriented programming

The style issues of this section are related to classes and object-oriented programming.

Style issues in the current and following sections have been taken from the Finnish book ”Olioiden ohjelmointi C++:lla” written by Matti Rintala and Jyke Jokinen (pp. 341-353). Most of the points have been mentioned in other parts of the material with larger explanation. Therefore they are listed here shortly.

- Each header file (`.hh`) contains a single public interface (class). Header files contain only definitions (no implementations).
- Header files must be prevented from including more than once:

```
#ifndef CLASS_A_HH
#define CLASS_A_HH
...
#endif // CLASS_A_HH
```

- `include` directives can be used to include only header files. If you include several files, you must list your own files first and after that the library files.
- The implementation of an interface is written in the implementation file (`.cpp`) in the same order as they are in the corresponding header file (`.hh`).
- Classes have been defined with a good public interface (as small as possible), and all the operations are conducted via this interface.
- Write the `public` part of a class before the `private` one.
- The `public` part of a class contains no member variables.
- The `private` part of a class contains no useless variables.
- No operations of a class are implemented outside the member functions.
- The member variables of a class must be initialized in their definition or in the initialization list of the constructor in the same order as they are defined in the header file.
- When possible, define a member function as a `const` function.

Other style issues

Finally, some general style rules are listed shortly below.

- The scope of a variable must be as small as possible (e.g block, function, class).
- Each variable must be defined separately, and thus, avoid writing definitions as `int a, b;`.
- Variables must be initialized.

- The characters for pointers and references (`*` and `&`) are written immediately after the type without an empty space. For example:

```
Date* ptr = nullptr;
void printDate( Date& dateObject );
```

- The return value type of `main` is always `int`.
- The names of the parameters of a function must be given in both the declaration and definition of a class, and they must be identical in both places.
- If you want to pass an object as a parameter for a function, use a reference to the object instead the object itself, whenever possible.
- The default value parameters of a function must be given in the declaration of the function, and you must not give more of them in the definition of the function.
- A function must not return a reference or a pointer to its local data.
- Records (`struct`) must not have member functions, but constructors, an (empty) destructor, and operator functions are allowed.
- Do not index vectors with brackets, but use `at` operation instead. (The same holds also for other containers that have these operations defined.)
- Unless you are absolutely sure that you can use C++ exception mechanism, then do not try to use it. Exception handling in C++ will be considered only on the next programming course. Especially, C++ exception mechanism must not be used in the place of control structures (as you may have done in Python programs).

C++ has other style guidelines e.g. those related to dynamic memory management and inheritance of classes. We will return to them in the context of these topics.

Huom

We have tried to follow the above guidelines in the template codes and examples of the course. However, we have not always succeeded in doing so, since there have been many authors writing the codes and most often they have worked in a hurry. We hope that you inform us if you notice any violations of the guidelines.

Debugging

Debugging is a programming term used to describe the locating and correcting of errors in a program. Often, but not always, the most demanding part of it is locating the error.

In order to try and find the error, you must be aware of its existence. You can achieve that only by testing the program with different inputs and comparing the results with the results you know are correct. The basic ideas of testing were shortly discussed on the previous programming course.

Here, we are going to introduce you to the general principles of a couple of useful mechanisms for searching for the reason of the error once you have noticed your program works incorrectly.

The first mechanism is **test printing**. It was used especially on the previous programming course. Test printing means that you add print commands (`print`, `cout`) into the code around the expected error, purely with the purpose of testing the code. They show you how the program proceeds and how the values of interesting variables change. The idea is, of

course, that you can determine where the error is by reasoning, after you have looked at the printed variable values and the printing order of the test prints. There is no reason to discuss test printing further, because you are already familiar with it as a debugging mechanism.

Another error tracing mechanism, and often much more user-friendly, is the use of a so-called **debugger**. A debugger is a separate program that provides an environment in which to execute the program you want to test.

The use and functionality of a debugger depend on the programming language and the compiler or interpreter you use, but typically, it supports at least the following operations, in one way or another:

- setting breakpoints (with either arrival on the line, calling a certain function, or handling a variable as the criteria)
- executing the program one command or function at a time (stepping)
- observing the state (value) of the variables
- observing the hierarchy of function calls (stack trace).

Usually, you locate the error following steps like these:

1. Get a rough estimate of the error's location (in other words, make an educated guess on where the error could be located). You can do this either by testing or, if the program crashes during execution, with a debugger.
2. Set the breakpoint before the assumed error, and run the program within the debugger.
3. Execute the program from the breakpoint onwards, one command at a time, and observe how e.g. the variable values and return values of the functions change, until you can, hopefully, deduce the reason of the error.
4. If you cannot locate the error on your first try, it is usually a good idea to try and find the error location by moving the breakpoint forward or back.

File management

Let's compare file management in Python and C++. Below you can see a simple program (written in both languages) that counts the sum of integers in a file. Each of the numbers should be on their own line in the file. There must be no empty lines.

```
def main():
     filename = input("Input file name: ")
     try:
          file_object = open(filename, "r")
          sum = 0
          for line in file_object:
               sum += int(line)
          file_object.close()
          print("Sum: ", sum)
     except IOError:
          print("Error: failed opening the file.")
     except ValueError:
          print("Error: bad line in the file.")

main()
#include <iostream>
```

```
#include <fstream>  // Notice the required library for file handling
#include <string>

using namespace std;

int main() {
    string filename = "";
    cout << "Input file name: ";
    getline(cin, filename);

    ifstream file_object(filename);
    if ( not file_object ) {
        cout << "Error: failed opening the file." << endl;
    } else {
        int sum = 0;
        string line;
        while ( getline(file_object, line) ) {
            sum += stoi(line);
        }
        file_object.close();
        cout << "Sum: " << sum << endl;
    }
}
```

The programs do not, in fact, work identically, because the C++ program uses the function `stoi`to change a line read from the file into an integer, and the function does not notice if there is additional characters after the numerical ones.

If there is something at the beginning of the line that does not qualify as an integer, `stoi` will exit with an exception, and the program terminates. We will get back to the exceptions in C++ at a later stage (not until the next programming course).

If you wish to read files in C++, you follow the steps below:

- At the beginning of the program, include the library `fstream`. It includes data types used in file processing.
- Depending on whether you want to read the file or write in it, you define an object of either the type `ifstream` or `ofstream`, and you pass to its constructor the name of the file as a parameter.
- A failure in opening a file can be recognized by using the file variable as the condition of the `if` statement: it will be evaluated as `true` if opening the file was successful, and if not, `false`.
- Just as in Python, the easiest way to read a file is to handle the reading in a loop, one line at a time, into a string variable, and then handle that string with string operations.
- After you have finished reading from or writing in a file, it is good practice to close it by calling the method `close`. For example, doing this when writing in files ensures that all written data will be saved onto the hard drive. Also, the method `close` releases the file from the use of the program.

You can read data from the file variable `ifstream` with the input operator `>>`, the same way you are used to reading from the stream variable `cin`. You write into a file (which means storing onto the hard drive) when you target the output operator `<<` at the `ofstream` type file variable, and this action is similar to what you are used to doing with `cout`.

Therefore, **stream** is a general name to a variable that can be used for handling the peripheral devices of the computer (reading and writing). All the actions with the peripheral devices have been implemented in C++ with data streams. In addition, the mechanism is

sophisticated enough that you can handle all the input streams in an identical way. It is the same with output streams.

In practice, it means that both `cin` and all the `ifstream` type stream variables are handled with the same operations, and they act in a similar way to each other. On the other hand, also `cout`and the `ofstream` type streams are handled the same way. This is good, because then the programmer only needs to learn one mechanism, and they can use it for several purposes.

Some output and input operations

A short list of useful stream handling operations:

- `cout << output`

  `output_stream << output`

  You can print or save data into the *output stream* using the output operator `<<`.

- `cin >> variable`

  `input_stream >> variable`

  You can read a data element of a known type from the *input stream* directly into a variable of the same type with the input operator `>>`.

- `getline(input_stream, line)`

  `getline(input_stream, line, separator)`

  Read one line of text in the *input stream* and save it into the string variable *line*. If the call includes a third parameter - a *separator* of the type `char` - reading and saving into the variable *line* will be continued until the first *separator* comes up from the stream.

```
string text_line = "";

// One line of text from the keyboard
getline(cin, text_line);

// Until the next colon
// (might read several lines at a time)
getline(file_object, text_line, ':');
```

- `input_stream.get(ch)`

  Read one character (`char`) from the input stream into the variable *ch*:

```
// You can read the file one character at a time:
char input_char;
while ( file_object.get(input_char) ) {
    cout << "Character: " << input_char << endl;
}
```

Checking the success of stream operations

A stream variable can be used in C++ as a condition of the `if` structure. The compiler interprets the stream as `bool` type (executes an implicit type conversion), and sets the value as `true` if the stream is valid, and as `false` if it is not. For example:

```
ifstream file_object("file.txt");
if (file_object) {
    // Opening was successful, next we will do something to the file
} else {
    // Opening was unsuccessful, the stream is invalid
    cout << "Error! ..." << endl;
}
```

You can see the successfulness of all the stream-targeted operations in C++ by writing the operations as the condition of an `if` or `while` structure. For example:

```
while(getline(file_object, line)){
    // You will enter the loop structure if reading a line from the stream succeeded
}
// The loop structure ends when you cannot read any more lines,
// i.e. when the whole file has been read
```

Here is a more detailed explanation of what just happened. Every time you execute an operation targeting a stream in your program, the return value of the operation will be the same target stream. If your operation is included as a condition of an `if` or `while` structure, the compiler will perform an implicit type conversion for the stream into a `bool` type. As we explained above, the value will be `true` or `false` depending on whether the last operation targeted at that stream was successful.

You will get the simplest solutions to the assignments of this course by always writing the stream-targeted operations as conditions of the aforementioned structures (`if`, `while`) **, because then you simultaneously test the success of your stream operation.**

Trivia: the stream also has the method `input_stream.eof()`, and you can use it to check if **the latest attempt to read from the stream was unsuccessful because the file has been read through**.

```
// First, you need to try to read something from the stream
file_object.get(ch);

// After doing that, you can try to examine if
// the reading attempt was a failure because of the fact
// that there was nothing left to be read.
if ( file_object.eof() ) {
    // The file has been read from start to finish: the character
    // in the variable is now an undefined and useless value.
} else {
    // The variable holds a char type value,
    // successfully read from the file.
}
```

You often see incorrect use of the method `eof`. For example, the following solutions are almost always incorrect:

```
while ( not file_object.eof() ) {  // ERROR!
    getline(file_object, line);
    ...
}
```

The mistake is that reading the file is set as a success when the last line has been read. It means that `eof` returns `false`. (If you do not understand, look above to see what `eof` does again.) This is the reason why the condition of `while` will be `true` after reading the last line, and the program will execute the loop structure once more, even though the file does not contain any lines to read.

For example, you can use the method `eof` to examine whether your loop structure finished handling a file because it reached the end of the file, or because of an error. On this course, we will not practice the special error situations arising from file management in as detailed a way as here.

STL iterators

Iterators are data types (or variables, depending on the context) you can use to examine and modify the elements stored in a container. You can think of the iterator as a bookmark that remembers the location of one element in a container.

Their idea is to offer a uniform interface that allows you to use the same operations on all the elements within containers, despite the exact type of the container. In other words, the operations are created to be able to handle the elements that the iterator(s) is(/are) pointing to, despite the type of the container. (To be precise, the iterator does not point to the element, it points to the space between two elements.)

A frequent problem you will encounter is how to go through all the elements within a container. In case of a vector, it is not difficult, because the elements of a vector can be indexed through with a loop. This is not the case with all containers, because elements in certain containers do not have an index number.

If one iterator allows you to mark the location of one element, two iterators can be used to show a range of elements in the container: every element between two elements. You will later see how STL algorithms utilize this characteristic.

All the container types in C++ offer the programmer a group of iterator types to be used to handle the containers. The most useful of them are:

```
container_type<element_type>::iterator
container_type<element_type>::const_iterator
```

The `const_iterator` type allows the programmer to examine the elements in the container, not edit them. You need this kind of iterators if the container variable has been defined as a constant with the keyword `const`. For example:

```
const vector<string> MONTHS = { "January", "February", ... };
```

We will now implement a very simple program that examines the elements of a vector with an iterator and then prints their values to the screen:

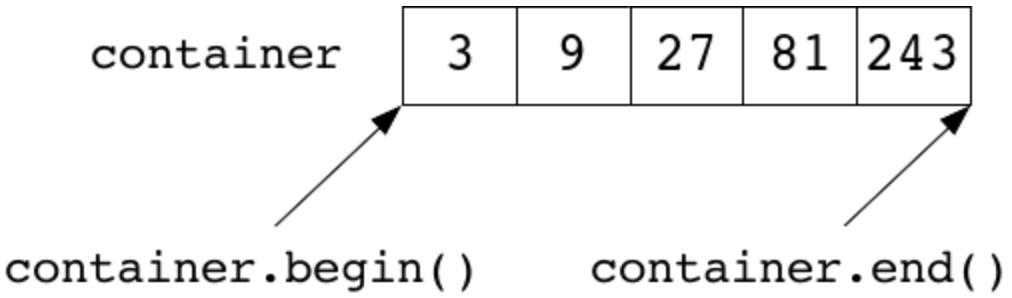
```
#include <iostream>
```

```
#include <vector>

using namespace std;

int main() {
    vector<double> numbers = {1.0, 1.5, 2.0, 2.75};
    vector<double>::iterator veciter;
    veciter = numbers.begin();
    while ( veciter != numbers.end() ) {
        cout << *veciter << " ";
        ++veciter;
    }
    cout << endl;
}
```

The return value of the method `begin` shows the location of the first element in the container. The method `end` returns the iterator pointing to the end of the container: `end` does not point to any specific value in the container. It is a kind of ending symbol.

Often, the iterator `end` is used as a return value depicting an error or a failure of the function returning that iterator.

The following picture illustrates the iterators `begin` and `end`:



The element of a container, pointed to by the iterator, can be handled by targeting the operator * at it. For example:

```
cout << *veciter << endl;
*veciter = 0.0;
*veciter += 2.5;
function(*veciter);
```

If you want to target with methods at the object the iterator is pointing to, you should use the `->`operator.

You can use the ++ operator to make the iterator point to the element that is next in line, or the `--` operator to point it to the previous element. You can use the == and `!=` operators to test whether two iterators are pointing to the same element or to different ones.

If all you want to do is go through all the elements within the container (it can be used with other containers than vector as well), you can use the `for` structure:

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

using namespace std;
```

```
int main() {
    vector<double> numbers = {1.0, 1.5, 2.0, 2.75};

    for ( auto element: numbers ) {
        cout << element << " ";
    }
    cout << endl;
}
```

It works analogically to the Python structure:

```
numbers = [ 1.0 1.5 2.0 2.75 ]
for element in numbers:
    print(element, end=" ")
print()
```

A version of `for` that goes through the C++ container elements behind the scenes uses iterators, but that is not visible to the programmer.

The iterating variable of the abovementioned `for` structure (in our example, `element`) is, by default, a copy of the element next in turn in the container being worked on (copy semantics).

However, if we want to be able to change the container's elements in the body of the `for`structure, we should define the iterating variable as a reference:

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

using namespace std;

int main() {
    vector<double> numbers = {1.0, 1.5, 2.0, 2.75};

    for ( auto& element: numbers ) {
        element *= 2;
    }

    // At this point the elements of numbers vector
    // have been doubled: 2.0, 3.0, 4.0 5.5
}
```

An important detail you should keep in mind: if you add elements to a container or delete them from it, the values of the iterators that were set to point to its original elements will not be valid anymore.

The word `auto` used in the examples is a reserved word in C++. You can use it as the type of the variable you are defining in cases where the compiler is able to deduce the type from the initializing value:

```
auto amount = 0;              // int
auto temperature = 12.1;      // double
auto iter = numbers.begin();  // vector<double>::iterator
```

We are, however, not using `auto` in all the examples where it would be possible, because it is good for you to learn the type defining mechanisms of a statically typed language yourself.

STL algorithms

- The STL library `algorithm` provides you with the most common operations you need to conduct on elements within a container.
- In order to use the algorithm library, here is an addition that you need to insert into the code:

```
#include <algorithm>
```

- The ready-made algorithms are generic: They do not care about the type of the container, since they are able to operate with any container.
- You relay the part of the container's elements that you want to process to the algorithm function by using the iterator range. Every function has at least two iterator parameters.
- For example, the algorithm (function) `sort` can sort the elements of the `vector`into an ascending order, as long as you use the iterators to tell to it the range that you want to sort:

```
vector<double> double_vector;
...
sort(double_vector.begin(), double_vector.end());
```

- You should remember that you can replace the iterators `begin` and `end` with any pair of iterators representing the range:

```
vector<double>::iterator range_begin = double_vector.begin();
vector<double>::iterator range_end = double_vector.end();
++range_begin;   // Pointing now to the second element
--range_end;     // Pointing now to the last element

// Sorting all the elements within the vector
// excluding the first and the last one.
// The first and the last one stay in the same place.
sort(range_begin, range_end);
```

- Important: the operations in the algorithm library do not have any effect on the target element of the second iterator parameter. Therefore, in the previous example, the last element of the vector was not sorted, because `range_end` was pointing to it.
- Below you can find a short list of useful functions from the algorithm library. We use the data structure `vector` in most of them because at this point, it is the only STL container type you know. However, unless otherwise mentioned, the functions will operate with other container types and strings as well.
- `count` calculates the amount of elements of a certain value within the container:

```
vector<string> enemies;
...
// How many people by the name of Aki are there in the enemies vector?
cout << count(enemies.begin(), enemies.end(), "Aki")
     << " enemies by the name of Aki!" << endl;
```

The function `count` does not work with the `map` structure quite as straightforwardly.

- `min_element` and `max_element` search the container for the element with the smallest or the greatest value, and return the iterator pointing at the element in question:

```
vector<int> amounts;
...
vector<int>::iterator smallest_it;
smallest_it = min_element(amounts.begin(), amounts.end());
```

```
cout << "Smallest amount: " << *smallest_it << endl;
```

The functions `min_element` and `max_element` do not work with the `map` structure quite as straightforwardly.

- `find` searches the container for the chosen value and returns the iterator into the first element it finds or, if it cannot find the value, it returns the `end` iterator of the container.

```
vector<string> patients;
...
vector<string>::iterator iter;
iter = find(patients.begin(), patients.end(), "Kai");
if ( iter == patients.end() ) {
    // There is no Kai in the patient queue.
    ...
} else {
    // Kai was found and is located where iter points at
    // print it and delete it from the queue.
    cout << *iter << endl;

    // The container vector has the method erase. You can use it to
    // remove the element targeted by the iterator.
    patients.erase(iter);
}
```

You should remember that the `string` type and the types you will learn later, `set`and `map`, have a method function called `find`. It is worth using, because it is several times as fast as the `find` in the algorithm library, especially when used by the latter two types.

- `replace` replaces all chosen values with a new value:

```
replace(text_vector.begin(), text_vector.end(),
        "TUT", "TUNI");
```

  in which all elements with the value ”`TUT`” within the vector are replaced with the value ”`TUNI`”.

  `replace` does not work with the `set` or `map` structures.

- `fill` changes all elements within the iterator range into the given value:

```
string my_string = "";
...
// After the following fill operation, the
// string will be filled with question marks.
fill(my_string.begin(), my_string.end(), '?');
```

  `fill` does not work with the `set` or `map` structures.

- `reverse` reverses the order of the given range

```
vector<Player> order_of_turns;
...
// On the next round of the game,
// the players take their turns in the opposite order.
reverse(order_of_turns.begin(), order_of_turns.end());
```

  `reverse` does not work with the `set` or `map` structures at all.

- `shuffle` shuffles the elements and puts them in a random order:

```
vector<Card> deck;
minstd_rand gen; // A random number generator called gen is created
...
shuffle(deck.begin(), deck.end(), gen);
```

  `shuffle` does not work with the `set` or `map` structures at all.

  C++ has several different random number generators, and you can find them in the `random` library. Using the data type `minstd_rand`, you can create a random number generator of a certain kind, and with the `default_random_engine` we saw above, a random number generator of another kind. At this point, it is enough for you to know how to create one (or both) of them. If you want, you can search the Internet to find out more about the different random number generators in C++.

- copy copies the elements of the container into another container:

```
string word = "";
...
// Initialize the char_vector to have
// as many elements as there are characters in
// the word.  Please note that this is, again, an initialization
// where you must use parentheses.
vector<char> char_vector(word.length());

// All the characters in the word are copied
// into the char_vector as elements.
copy(word.begin(), word.end(),
     char_vector.begin());
```

  You must have free space at the target location of the copy action. At the start, the char_vector above is initialized to include as many elements as the wordincludes characters.

  This concerns all STL algorithms storing results in a container: The target container must have space for all the elements you are going to store.

  The element types of the source container and target container must be the same.

- Several algorithm library functions have a version that you can adjust by using a function parameter. Let us take the sort function as an example.

  Before getting into the details, it is important for you to understand that the sort function can sort data into order by magnitude only if the data has a defined magnitude relation (less/greater). It does not concern, for example, a vector with colors as its elements, because it is not known whether purple is less or greater than, say, lilac since you cannot sort colors by their magnitude.

  This does not present a problem when the elements in the container to be sorted contain ordinary data for that language, such as strings or other predefined types of C++. However, as soon as you try to use classes implemented by yourself that cannot be compared automatically, you have a problem.

- Here is a small example (irrelevant parts of the code are left out to save room):

```
class Student {
public:
    Student(string name, int student_id);
    string fetch_name() const;
    int fetch_student_id() const;
    void print() const;
private:
    string name_;
    int    id_;
};

bool compare_ids(const Student& stud1, const Student& stud2) {
    if ( stud1.fetch_student_id() < stud2.fetch_student_id() ) {
        return true;
    } else {
        return false;
    }
}

bool compare_names(const Student& stud1, const Student& stud2) {
```

```
    if ( stud1.fetch_name() < stud2.fetch_name() ) {
        return true;
    } else {
        return false;
    }
}

int main() {
    vector<Student> students = {
        { "Teekkari, Tiina", 121121 },
        { "Arkkari, Antti",  111222 },
        { "Fyysikko, Ville", 212121 },
        { "Teekkari, Teemu", 100011 },
        { "Kone, Kimmo",     233233 },
    };

    // Let's order and print in the increasing order of student ids.
    sort(students.begin(), students.end(), compare_ids);
    for ( auto stud : students ) {
        stud.print();
    }

    cout << string(30, '-') << endl;

    // Let's order and print in the increasing order of names.
    sort(students.begin(), students.end(), compare_names);
    for ( auto stud : students ) {
        stud.print();
    }
}
```

- The functions `compare_ids` and `compare_names` of the example are so-called comparison functions:
  - The parameters (2 pcs) are constant references to the elements of the data type your function is supposed to compare.
  - The type of the return value is `bool` and the return value is `true` only if the first parameter is strictly smaller than the second one.
- You can give the comparison function as an additional parameter to such functions of the algorithm library that work differently depending on the size of the elements in the processed container.

  Therefore, you could search and print the student with the greatest student number from the function above:

  ```
  vector<Student>::iterator iter;
  iter = max_element(students.begin(),
                     students.end(),
                     compare_ids);

  // Because the target of the next print method is
  // the object the iterator is pointing to, you will use
  // the operator -> instead of a normal period.
  iter->print();
  ```

- When you give the comparison function as a parameter to a function in the algorithm library, you use only the name of the function. If there were parentheses after the function, it would try to call

the function and give the return value of the comparison function as the parameter, and this is not supposed to happen.

- Functions with other functions as their parameter are called **higher-order functions**.

More STL containers

In addition to series (a container that preserves the order of the elements), STL has so called **associative containers**. In their internal implementation, they store the elements in a way that is as fast as possible to search. You cannot think of the elements within associative containers as arranged in a queue, and therefore they do not have an index number. The data is stored in associative containers on the basis of a **(search) key**, and the searches are conducted with that key.

First, we will get to know two associative container types provided by C++:

- `set` - completely analoguous with the Python `set` type, which is identical with the mathematical set where each value can exist no more than once.
- `map` - works similarly to `dict` in Python. It includes pairs of a key and a mapped value, and it is quick to find the mapped value you need, if you know the key attached to it.

Similarly to the container `vector`, the `set` and `map` need their `include` line at the beginning of your code:

```
#include <set>  // for using the set structure
#include <map>  // for using the map structure
```

After these two containers we will get to know pairs, and after that unordered containers will be briefly introduced.

Examples on STL `set`

We will now define a set where you can store strings:

```
set<string> ship_has_been_loaded;
```

After the definition, the set is automatically empty.

The set can be initialized with another set of the same type, or another set of the same type can be assigned into it:

```
set<int> lottery_numbers_1;
...
set<int> lottery_numbers_2(lottery_numbers_1);
...
lottery_numbers_1 = lottery_numbers_2;
```

You can also initialize and assign your set with a curly bracket list:

```
set<int> prime_numbers = {2, 3, 5, 7, 11, 13, 17};
```

```
...

set<string> friends;
...
friends = { "Matti", "Maija", "Teppo" };
```

You can add a new element into the set with the method `insert`:

```
ship_has_been_loaded.insert("dogs");
```

The addition works also if the element is already included in the set but will not do anything.

The amount of elements in the set can be found out with the method `size`:

```
// Amount of friends
cout << friends.size() << endl;
```

You can examine whether an element is included in the set by using the method `find`:

```
// The word is not included in the set, let's add it.
if ( ship_has_been_loaded.find(word)
          == ship_has_been_loaded.end() ) {
    ship_has_been_loaded.insert(word);
    cout << "Ok!" << endl;

// The word was already included in the set.
} else {
    cout << "You lost, " << word << " has already been loaded!" << endl;
}
```

A single element can be removed from the set with the method `erase`:

```
// Teppo is not a friend anymore.
friends.erase("Teppo");
```

The method `clear` empties the set (removes all the elements):

```
ship_has_been_loaded.clear();
```

The relational operators can be used on sets with the same element type:

```
if ( lottery_numbers_1 == lottery_numbers_2 ) {
  // Both of the sets include exactly the same elements.
  ...
} else {
  // The contents of the sets differ.
  ...
}
```

The method `empty` returns the value `true` only if the set is empty:

```
if ( not friends.empty() ) {
    // There is at least one friend.
}
```

Examples on STL map

Unlike other STL structures, the definition of the map includes both the types of the key and the mapped value:

```
map<string, double> prices;
```

After the definition, the uninitialized map container is empty.

As with other containers, the map can be initialized from another map of the same type, and you can assign a map of a similar type into it:

```
map<string, string> dictionary_1;
...
map<string, string> dictionary_2(dictionary_1);
...
dictionary_1 = dictionary_2;
```

You can also use the curly bracket list for initializing or assigning:

```
map<string, double> prices_2 = {
  { "milk",   1.05 },
  { "cheese", 4.95 },
  { "glue",   3.65 },
};
...
prices_2 = {
  { "pepper",    2.10 },
  { "cream",     1.65 },
  { "chocolate", 1.95 },
};
```

The data related to a search key can be accessed with the method at:

```
cout << prices_2.at("cream") << endl;  // 1.65
prices_2.at("cream") = 1.79;
```

However, you should remember a possible problem when using the at method: If the key cannot be found in the map, the result is an exception and the program will terminate. So please note: you cannot add new elements to map with the method at.

You can avoid the problematic situation by checking if the search key is located in map before calling at, just like the code below shows you.

```
if ( dictionary_3.find(word) != dictionary_3.end() ) {
    // The word was found in the map.
    cout << dictionary_3.at(word) << endl;
} else {
    // The word was not found in the map.
    cout << "Error, an unknown keyword!" << endl;
}
```

It uses the previously known fact that `find` returns the iterator `end()` if it does not find what it was searching for.

You can remove a key-value pair from the `map` by giving the search key you want to be removed as a parameter to the method `erase`:

```
if ( prices_2.erase("chocolate") ) {
    // The erasing was successful, "chocolate"
    // is not in the pricelist anymore.
    ...
} else {
    // The erasing was not successful, "chocolate"
    // was not in the pricelist to start with.
    ...
}
```

Even though it is not an error to remove a key-value pair that does not exist, you can check their existence with `find` before the removal if you want.

You can add a new key-value pair with the `insert` method:

```
dictionary_1.insert( { word, word_in_finnish } );
```

If `word` is already in `dictionary_1` as a search key, `insert` will have no effect.

The method `size` returns the amount of key-value pairs stored in `map`:

```
cout << "Dictionary contains " << dictionary_2.size()
     << " pairs of words." << endl;
```

The action of going through the elements of `map` has also been implemented with iterators. It is, however, not as straightforward as with other STL containers, because each element now contains two parts: a key value and a mapped value. It has been implemented by making the elements of `map` structs (see the material on data driven programming on the previous round). Therefore, suppose a `map` structure defined like this:

```
map<int, string> students = {
    // id, name
    { 123456, "Teekkari, Teemu" },
    ...
};
```

then the elements stored in the structure would be of the following struct:

```
struct {
    int    first;
    string second;
};
```

where the field of the search key is actually named `first`, and the field of the mapped value, `second`.

Therefore you can go through the elements of `map` with an iterator like this:

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

using namespace std;

int main() {
    map<int, string> students = {
        { 200001, "Teekkari, Tiina" },
        { 123456, "Teekkari, Teemu" },
        // ...
    };

    map<int, string>::iterator iter;
    iter = students.begin();
    while ( iter != students.end() ) {
        cout << iter->first << " "
             << iter->second << endl;
        ++iter;
    }
}
```

It is worth noting in the code that the struct fields pointed by the iterator are handled with the operator `->`, instead of `.`, used usually.

If you want to avoid direct use of iterators, you can use `for` to go through the elements in `map`:

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

using namespace std;

int main() {
    map<int, string> students = {
        { 200001, "Teekkari, Tiina" },
        { 123456, "Teekkari, Teemu" },
        // ...
    };

    for ( auto data_pair : students ) {
        cout << data_pair.first << " "
             << data_pair.second << endl;
    }
}
```

Because `data_pair` in the previous loop is not an iterator but a real struct located in `map`, we have used the usual `.` operator to handle the fields `first` and `second`.

STL pair

Previously we considered `map`, the elements of which are actually pairs. It is possible to use pairs as such, and they can found from the library `utility`, and thus, you need to write at the beginning of the code the text:

```
#include <utility>
```

As can be concluded from its name, a pair consists of two elements. You can access the first one of them by writing `first` and the second one by writing `second`, as told in the context of `map`.

Below you can see different ways to declare and initialize pairs:

```
pair <int, char> pair1;
pair1.first = 1;
pair1.second = 'a';

pair <int, char> pair2 (1, 'a');

pair <int, char> pair3;
pair3 = make_pair(1, 'a');

auto pair4 = make_pair(1, 'a');
```

Unordered containers

The containers `map` and `set` introduced earlier have the property that their elements are ordered according to the key values. Therefore, a new element is inserted to such a location that the order is preserved.

Above you saw the example that traverses a `map` with an iterator or in a `for` loop. The output of the example code would print the elements in ascending order. This is useful in some situations. Most of the exercises (related to STL containers) on this course, the output is required to be in a certain order. Such a requirement makes it easier to implement automated tests.

In general, preserving the order makes some operations inefficient. Because of that, STL provides counterparts for `map` and `set`, called `unordered_map` and `unordered_set`. As you can conclude from the names, the elements in the latter containers are not in any specific order.

Since there is no need to keep the elements in order, it is possible to make searching, inserting, and removing elements more efficient on average than these operations are with ordered containers. You should use unordered containers, if the order of the elements does not matter, and you need not traverse the whole container element by element, but you only search for a certain element at a time.

The later course ”Data structures and algorithms” considers efficiencies of different containers more precisely, and especially for efficiency reasons it most often uses `unordered_map` and `unordered_set`, instead of `map` and `set`.

Recursion in general

Recursion means defining something in terms of the subject itself.

For example, the mathematical concept of a natural number’s factorial can be defined like this:

$$n!=n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot3\cdot2\cdot1n!=n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot3\cdot2\cdot1$$

Also, we define 0 ! to be 1.

Therefore, the factorial of a natural number (larger-than-zero) is the product of it and all the positive integers smaller than it. For example, $5!5!$ is $5\cdot4\cdot3\cdot2\cdot15\cdot4\cdot3\cdot2\cdot1$, i.e. 120.

On the other hand, we can also define a factorial recursively:

$$n!=\{1 \text{ if n is } 0n\cdot(n-1)! \text{ if } n>0n!=\{1 \text{ if n is } 0n\cdot(n-1)! \text{ if } n>0$$

When we apply the recursive definition to calculate $5!5!$, the result is:

$$5!=5\cdot4!=5\cdot4\cdot3!=5\cdot4\cdot3\cdot2!=5\cdot4\cdot3\cdot2\cdot1!=5\cdot4\cdot3\cdot2\cdot1\cdot0!=5\cdot4\cdot3\cdot2\cdot1\cdot1=1205!=5\cdot4!=5\cdot4\cdot3!=5\cdot4\cdot3\cdot2!=5\cdot4\cdot3\cdot2\cdot1!=5\cdot4\cdot3\cdot2\cdot1\cdot0!=5\cdot4\cdot3\cdot2\cdot1\cdot1=120$$

The idea of recursion is that you present the solution in the *same format as the original problem but as its sub-problems with a simpler solution*. After that, you can apply the recursive rule over and over again onto the received sub-problems, until the problem is simple enough for you to see the solution directly.

The goal of recursion is to divide and conquer.

Recursion in programming

Usually, recursion in the context of programming is related to functions. [1] A recursive function is self-defined, meaning it calls itself.

Let us implement the aforementioned factorial calculation with a recursive C++ function:

```
unsigned int factorial(unsigned int n) {
    if ( n == 0 ) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

which calculates and returns the factorial of its parameter `n`.

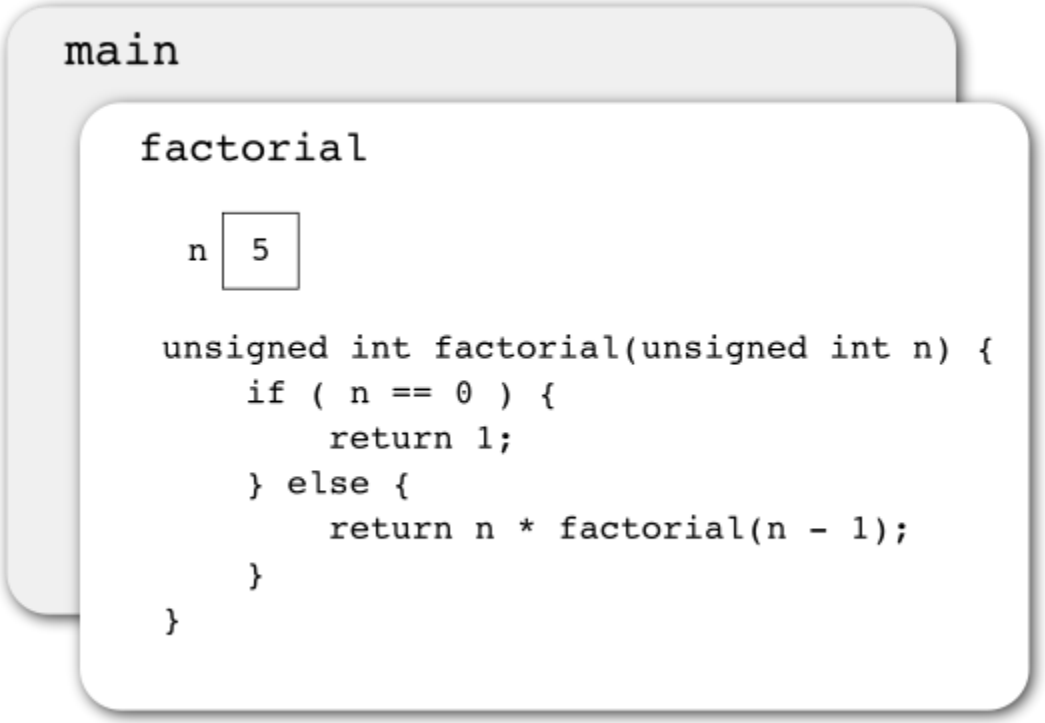
Basically, you can write the function straight from the recursive definition of factorial, but it is far more useful to understand how the function works, and why it produces a correct result.

Let us presume you call the function `factorial` like this:

```
int main() {
    cout << factorial(5) << endl;
}
```

and examine, stage by stage, how the execution proceeds. After the main program has called the function, the program executes the body of the function `factorial`:

```
unsigned int factorial(unsigned int n) {
    if ( n == 0 ) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

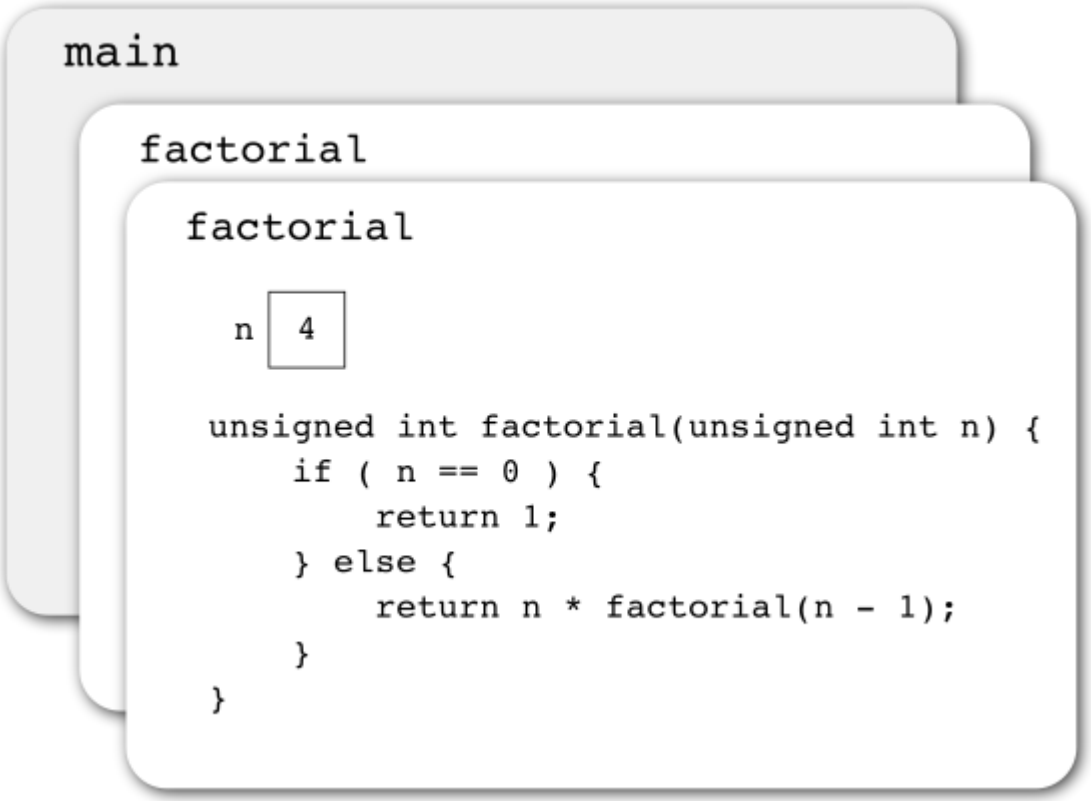


The covered `main` box in the picture means that the execution of the `main` function is not finished. First, the `factorial` function covering `main` has to execute the `return` command, and only after that, the execution of `main` can continue from the point where `factorial` was called.

Because the value of `n` is 5, the program executes the `else` branch.

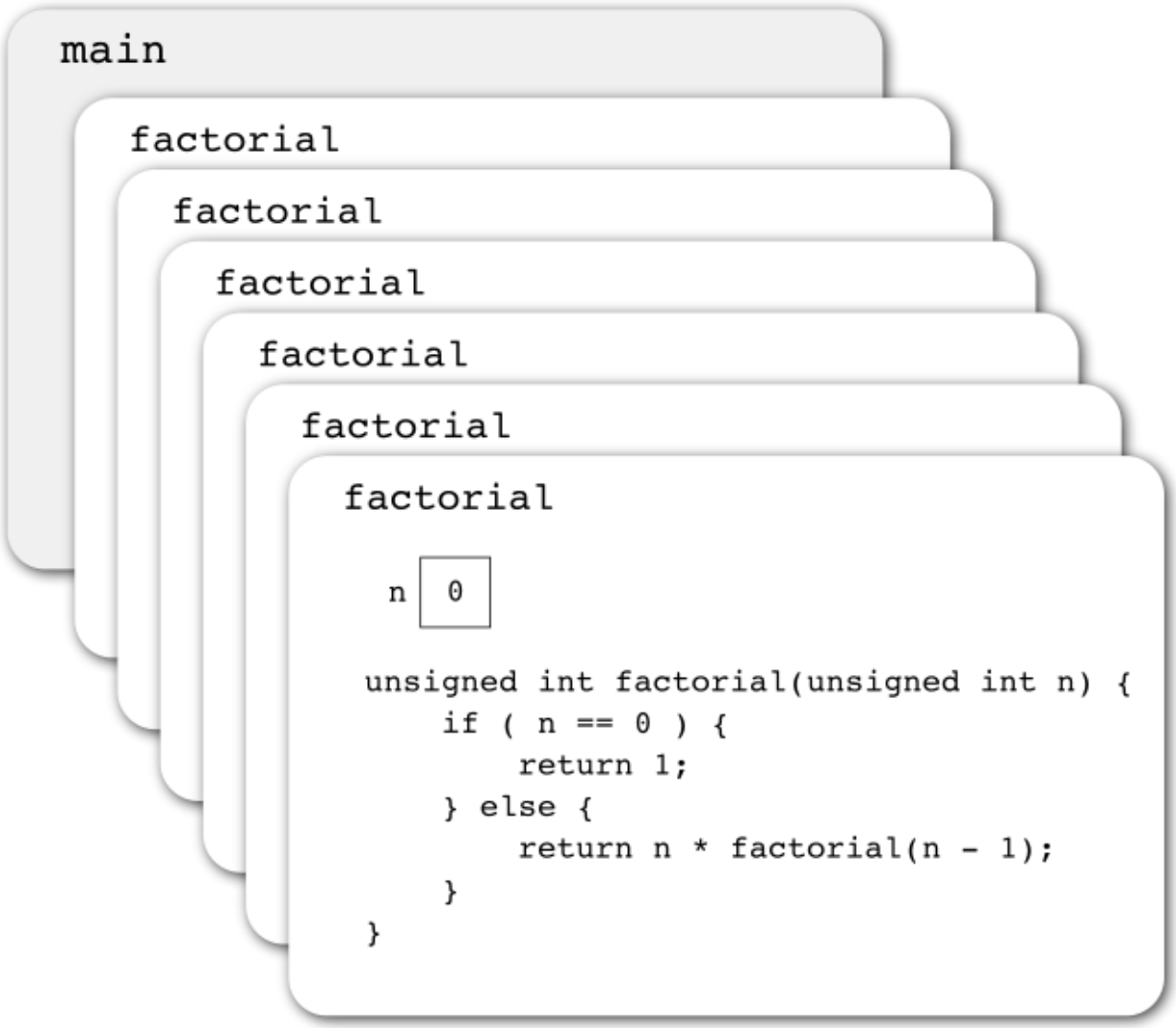
Before `factorial` is able to return a value, it must calculate a value to the expression `n * factorial(n - 1)`. Therefore, the execution of `factorial` remains unfinished until the recursive call `factorial(n - 1)` finishes calculating the factorial of 4.

In fact, the execution of this recursion round is identical to the one before, except that the value of the parameter `n` is now 4:



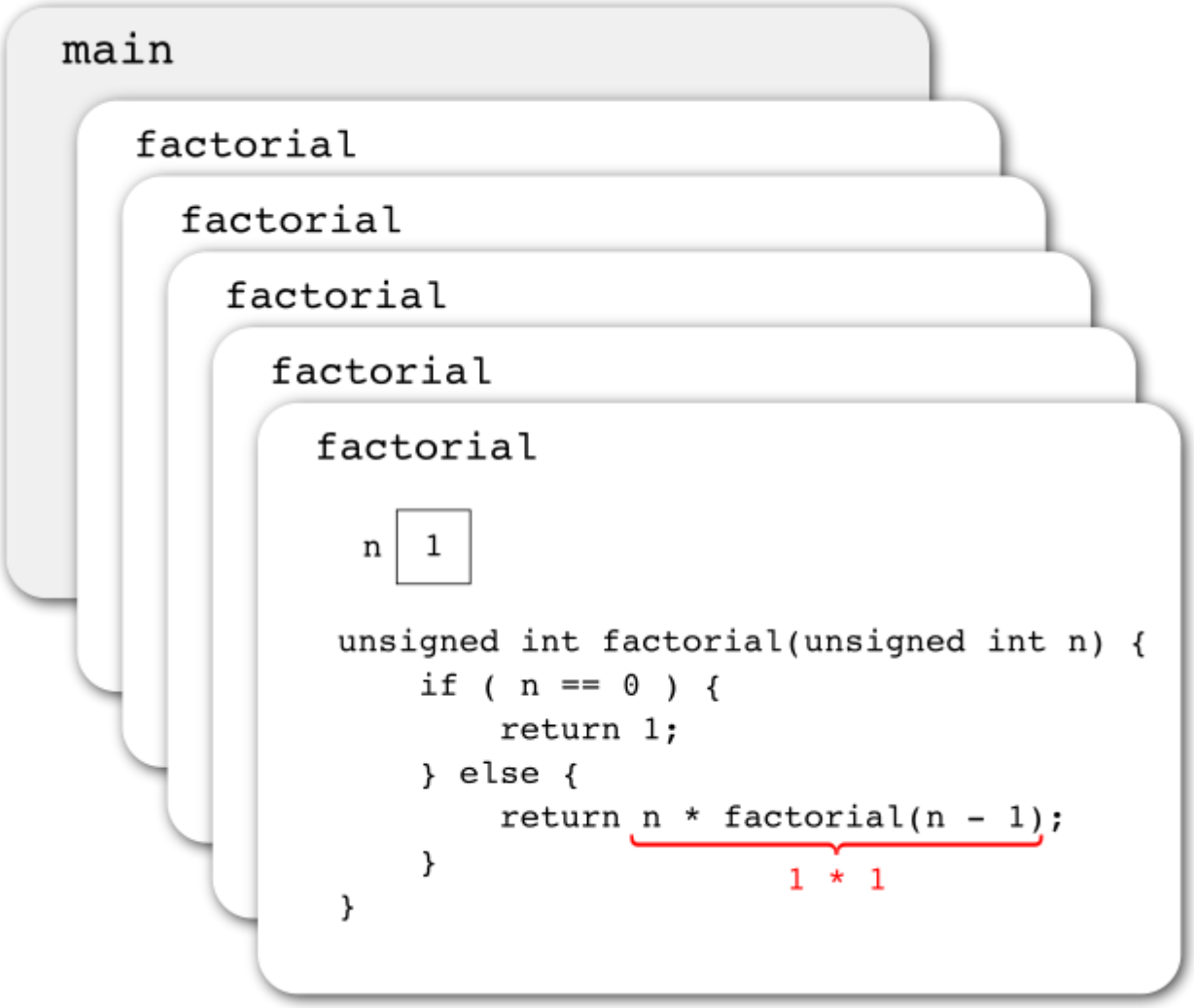
The execution, again, goes to the `else` branch, and the recursion advances deeper and deeper. Take note of the unfinished executions of the `factorial` function increasing in the background.

The program will continue like this, until the value of the parameter `n` reaches 0:



Now, instead of the recursive call, what happens is that `factorial` returns the value 1, and the stack of unfinished function calls starts to be released.

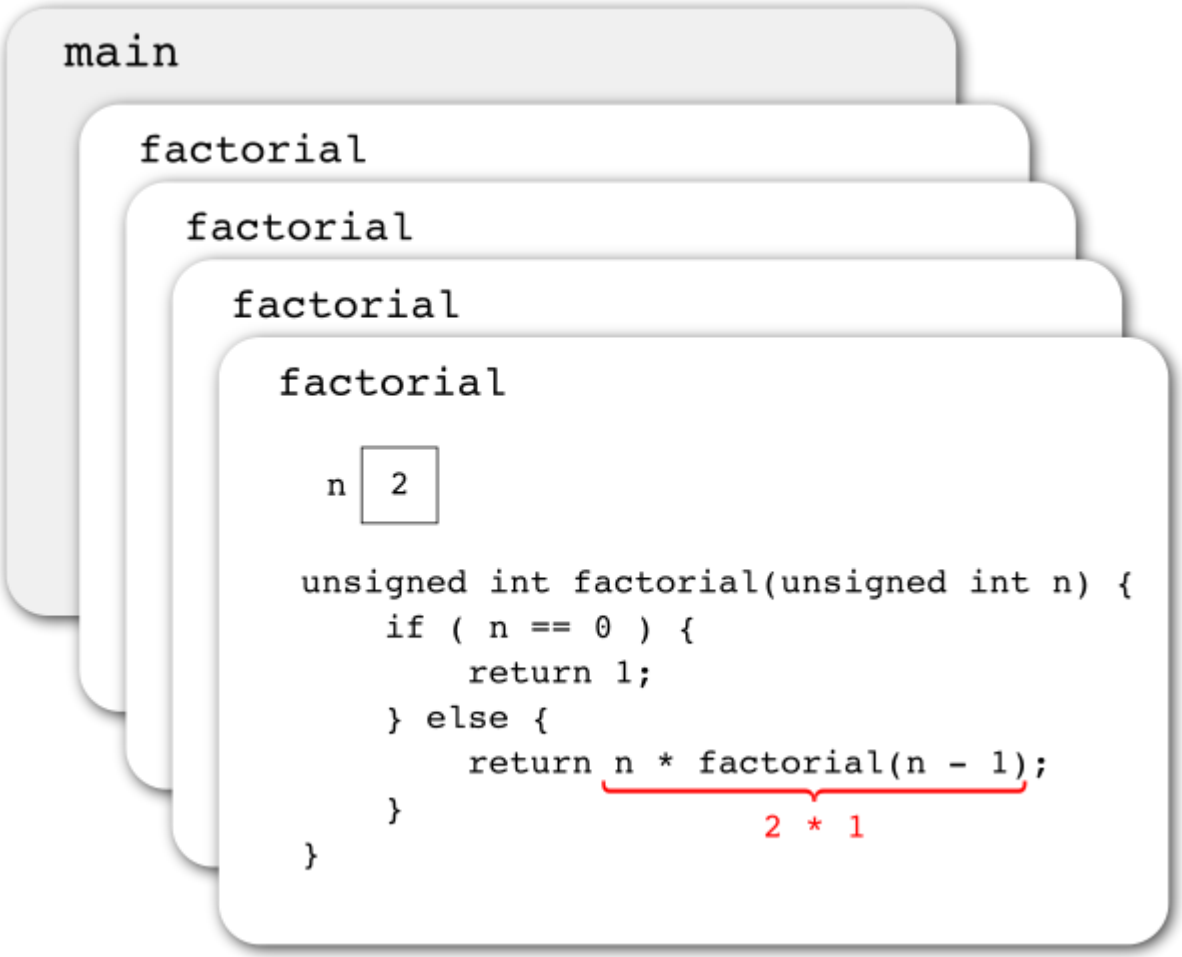
The last recursive call returns the value 1 to the second last call of `factorial`, and we multiply it by the value of parameter `n` that is also 1:



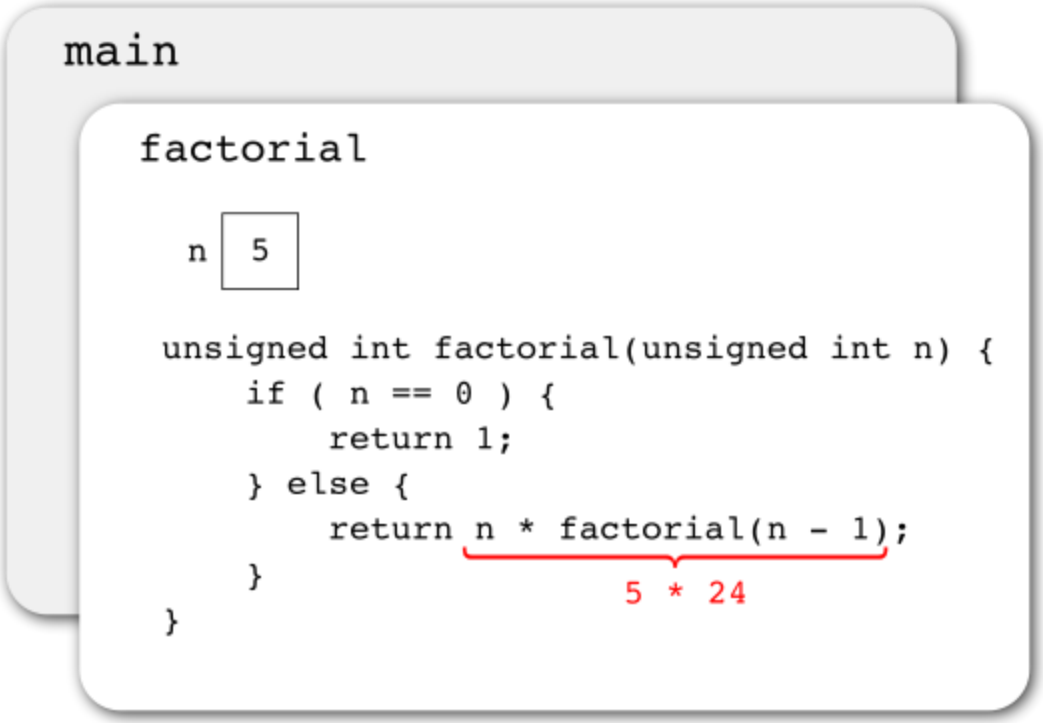
Please note that each call (instance) of the `factorial` function has its *own* version of the parameter variable `n`. The value of `n` is still the same when we return from the recursively called function to execute the commands of the calling instance. Why?

It is the same with all local variables.

The next unfinished `factorial` function continues in the same way:



In this way, recursive calls unwind and the execution returns back to the first instance of the `factorial` function:



It returns the value of the factorial of 5 to the main program:

```
int main() {
    cout << factorial(5) << endl;
}
```
120

Recursion is nothing special from the programming language's viewpoint: recursive functions are just functions, just like all the rest. Recursion is difficult to understand because it demands the programmer to perceive the problem and its solution in a possibly nonintuitive (?) way.

It is easier to design and understand recursive functions when you remember the following facts:

- A function is recursive if it calls itself (directly or indirectly).
- During the execution, as many instances of the recursive function are "on" as there are unfinished recursive calls.
- Each instance has its own values for the parameters and local variables.

Recursive function has two essential characteristics you should remember:

1. It has to have a **terminating condition** (or several) that recognizes the trivial cases of the problem and reacts to them without having to make a new recursive call.
2. Each recursive call has to simplify the problem in question in order to finally reach the trivial case.

The previous was represented in the `factorial` function like so:

1. The condition of the `if` structure (`n == 0`) is a trivial case: We know the value of zero's factorial right away.
2. Each recursion round was considering the factorial of a number smaller by one: The call parameter decreased by one.

If you forget either of these, the whole program will crash when you call the function (segmentation fault in Unix). The reason for that is that the function ends up calling itself indefinitely (an infinite recursion): The variables of the earlier calling instances [2] use memory until all the memory is used.

Recursion creates repetition. In principle, any loop structure can be replaced with a recursion. The best use of recursion is solving problems that are, by nature, recursive. You can simplify them "naturally" in a way that the reduced sub-problems are simplified cases of the original problem.

Often, a recursive solution is shorter and cleaner than a corresponding loop structure solution. However, usually the recursive solution is at least slightly slower and uses more memory than the loop structure.

---

[1] Data structures can also be recursive.

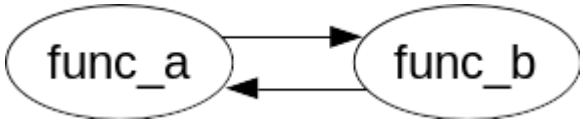
---

[2] And the other data concerning the earlier recursion rounds.

Different kinds of recursion

We can divide recursive functions in groups based on their characteristics and, for your general education, it is good to know them.

- **Direct recursion** happens when a function calls itself in its own body. For instance, `factorial` in the previous example uses direct recursion.
- **Indirect recursion** or **mutual recursion** means that the function `func_a` calls the function `func_b` which, in turn, calls `func_a`, et cetera:



  The knot that is created can include even more functions, i.e. the ”circle of calls” can be longer.

  You should avoid using indirect recursion, because it is often difficult to understand.

- **Tail recursion**, which we will get into this below.

Tail recursion

Tail recursion is the name of the situation where the return value of a recursive call becomes the return value of the calling instance (function) without any additional operations. It means that as the recursions ”unwind”, there are no actions left in the called function. In other words, the recursive call is located in such a point in the function that after the call, there are no statements to execute or expressions to evaluate. The function `factorial` of the previous example *is not* tail-recursive, because you first need to multiply the return value of the recursive call by `n` in order to receive a result that can be used as a return value:

```
return n * factorial(n - 1);
```

However, `factorial` can be created as tail-recursive:

```
unsigned int factorial(unsigned int n, unsigned int result) {
    if ( n == 0 ) {
        return result;
    } else {
        return factorial(n - 1, n * result);
    }
}
```

and you should give `1` as its last parameter in the call:

```
cout << factorial(5, 1) << endl;
```

It is typical for tail-recursive functions that the result is gathered bit by bit in an additional parameter (`result`), and the gathered value can be returned as such when the finishing condition is fulfilled. The consequence of this is that the initial value of the additional parameter must be the final result of the trivial case. Why?

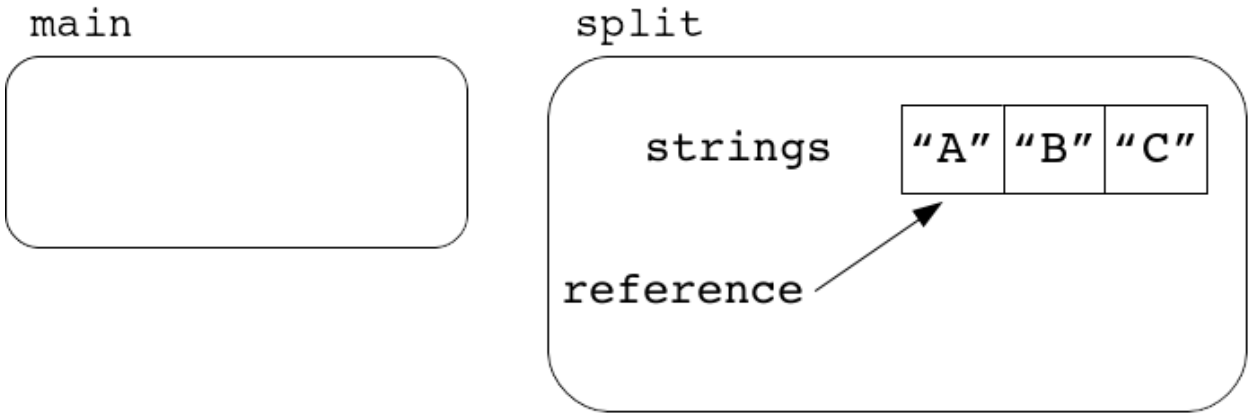
Tail recursion is an important concept, because you can mechanically move it to a loop structure. That way, you achieve the benefits of recursion but simultaneously get rid of its downsides.

Try it out yourself: In the previous assignment, you calculated the sum of integers, which is also easy to implement tail-recursively. If you want to try it, add a new function `sum_tail_recursive`into your file. This function has the same definition as the function `sum_recursive`, except that it also has an additional `int` type parameter, into which you begin calculating the sum.

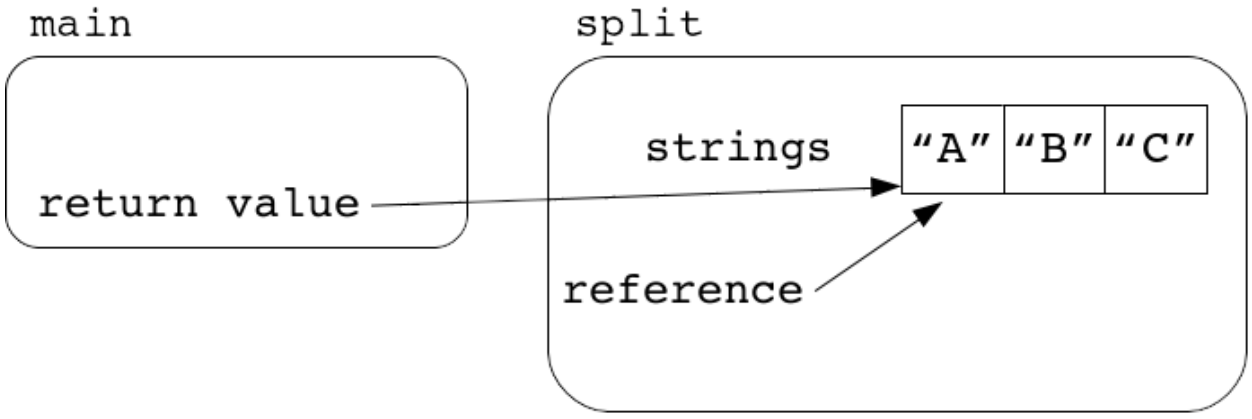
Introduction to memory management

Let us remind ourselves of the assignment (from round 4) where we created the `split` function. The return value of that function was `vector`. The assignment gave you a tip: You were not supposed to return a reference to `vector`, but the `vector` itself. The reason for this tip was that because we have learned to pass large data structures (such as `vector`) as constant reference parameters instead of value parameters, you could easily think that you can do the same with return values.

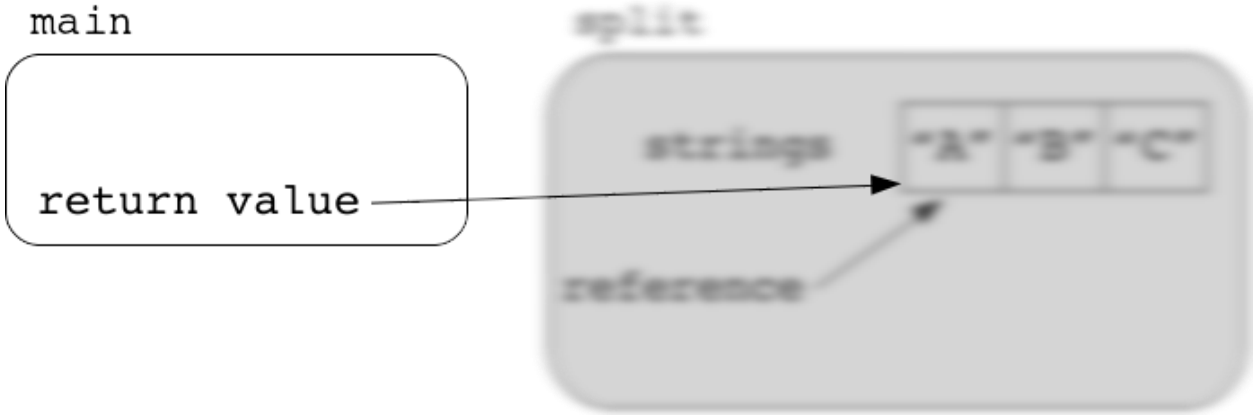
Even though at first, it might sound smart to increase your efficiency by passing a reference to `vector` as a return value instead of passing the `vector` itself, it does not work. We learned on the previous programming course already that when the execution of a function is finished, all the local variables of the function are removed from the memory of the computer. We could present the abovementioned in this way, for example:



We form a local `vector` variable and a reference to it in the function `split`.



If the function `split` returns the reference to its local variable for the main program…



...the reference used by the main program becomes a so-called **dangling pointer** because, when the execution of `split` is finished, all its local variables are removed from the memory of the computer. A dangling pointer is a little distracting as a term, because it can mean *either a reference or a pointer* targeting a variable that no longer exists.

Returning a reference that points to a local variable is a good example of a mistake you can make as a programmer if you do not understand how the memory management of programming languages works.

Until now, all the variables we have used have been so-called automatic variables, which means that the compiler would find a storage place for them in the memory and take care of destroying them. As we proceed to more complicated programs, we will face situations where it would be better if the programmer was able to determine exactly when a variable was destroyed. For example, in the above illustrations, it would be useful if the programmer was able to decide how long it is possible to use the contents of the `vector`.

On this round, we will study manual memory management in C++. The simplest way to practice this is for you to create a data structure with a dynamic size (= a data structure that can change in size). This means that we will implement something that resembles the `list` of Python or the STL `vector` of C++.

The reason for this practice is not a programmer having to implement dynamic containers themself. Actually, this is not even possible in most modern programming languages. Instead the reason is for you to learn the common principles of memory management and to be able to use dynamic memory management for other means which we will encounter on the next few rounds. Also, this practice will help us understand, for example, the functioning of the STL containers. An in-depth understanding of memory management helps the programmer to avoid mistakes even if they use pre-made dynamic data structures in their code.

While practicing the use of manual memory management, we will also take a much more detailed look at pointers than the first view given at the beginning of the course, when we compared value and reference semantics. The pointer is an important tool in C++ for implementing the higher-level structure of a program (indirect pointing and sharing access/ownership).

Memory and memory addresses

In this section, we will give you a rather non-detailed and simplified introduction to the concept of the (main) memory of a computer.

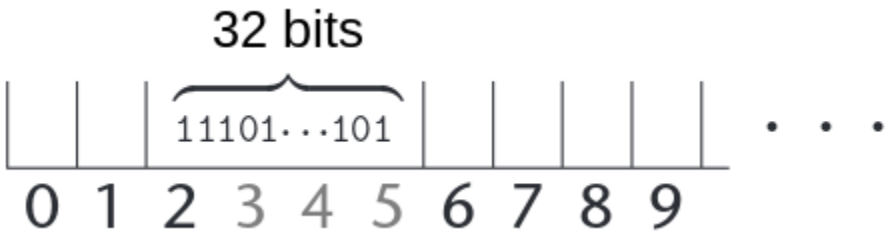
All the data handled by a computer is stored in the main memory of the computer. [*] You can think of the memory as a group of numbered cells:

and the processor can store a small amount of data into each of them. One cell is called a **memory location**, and its index number is called a **memory address**.

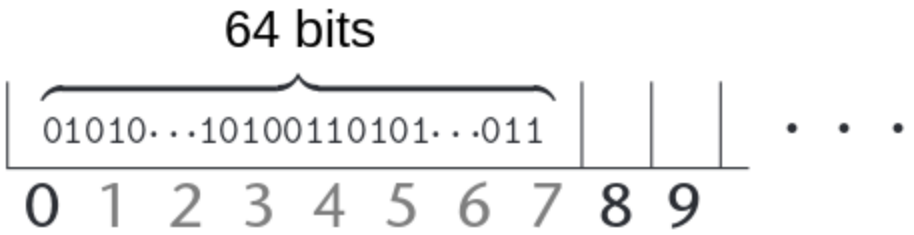
A memory location can contain one **byte** of data, which is comprised of 8 **bits**. Usually, however, the memory is handled in parts that are the size of multiples of a byte.

One memory location can store a data element that can be presented with 8 bits (for example, `char`). In case you need to store data that is presented with a larger number of bits, you have to use a suitable amount of consecutive memory locations.

For example, if the elements of the data type `int` have 32 bits, you could save one integer (i.e. its binary form) into four consecutive memory locations:



Correspondingly, a 64-bit `double` could be stored like this:



The examples show you that the same memory locations can store different types of information at different times.

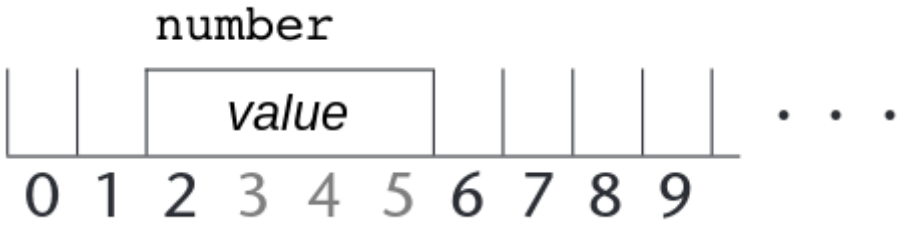
Because all the data is presented as bits in the depths of the computer, you cannot find out what type of data is stored in a memory location only by looking at its contents. This leads us to an old truth: All data needs to have a type in order for us to handle it correctly.

In C++, the definition of a variable is (according to value semantics) a way to name the contents of a memory location. The compiler chooses a location in the memory for a variable, and keeps track of the information on its type and the chosen memory address. Based on the type information, the compiler is able to use the correct machine commands to handle the value of the variable (i.e., the contents of the memory location).

If you are, for example, declaring an integer variable:

```
int number;
```

the compiler finds the necessary amount of free, consequent memory locations, and allocates them as the storage of the variable `number`:



After this, every time the program code uses the variable `number`, it is interpreted to mean the *value* located in the memory locations allocated to it.

For the future, it is important for you to remember these points:

- each memory location is identified with a unique memory address
- the memory location stores the actual data
- the variable is the name for the value stored in the memory location,

and, resulting from that:

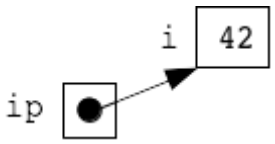
- each variable is connected to a memory address (meaning the one storing the value of the variable).

This is not the whole truth, but at least all the initial data is, at some point, stored in the main memory.
[*]

The example assumes that the pointer is 4 bytes, or 32 bits. In reality, in most modern computers, the pointers are 64 bits.

We can see in the example that when we use `*ip` in an expression, it works like an `int`-type variable: Its value is evaluated from the storage location to which `ip` points, and we can assign data into it with the operator `=`, which stores the value into the storage location in question. Pointers of other types work the same way.

The value of a pointer shows **where** a piece of data is located. Because the number of the storage location is not relevant in itself, the contents of the previous illustration are often presented in the following, simplified way:



By depicting the pointer with an arrow, we can show where it points to without repeating the index of the storage location, which is irrelevant. In other words, this picture is the same as the previous one, except that this one is more abstract as the storage location index has been left out and the pointing is depicted with an arrow.

With the use of the * operator, you can find out the **actual pointed data** from the pointer. It goes without saying (or does it?) that you can store a variable address only in a pointer with the same *target type* as the address type.

In the example, we initialize `nullptr` to the pointer `ip`. This is a small but important detail. A pointer like this is called the **null** or **nil** pointer. The special value `nullptr` can be assigned to all of the pointer variables. In this way you can set a pointer **to point nowhere**. You can compare the equality or inequality of a pointer to the `nullptr` value, and you can use `nullptr` as a pointer-type parameter. Often, the functions that have pointer as their return value type return `nullptr`when errors occur.

If you try to access [*] data through the null pointer during the execution of a program, it always results in an execution error and terminates the program. It is always good to use the null-pointer as the initializing value of pointer variables, and as the error-return value of functions with pointer as their return value type.

If you have stored the memory address of a `class`-type or `struct`-type value, the methods and fields of the pointed data are handled with the operator `->`.

---

[*] This means examining the data to which the pointer points. In other words, to target the unary operator * or the operator `->` at the pointer.

C++ arrays

On this course, it is common to use the STL structure `vector` when several values of the same type need to be stored and the elements then accessed by their index number. However, there is a more primitive type in C++ that acts partly like a vector. This structure is called an **array**.

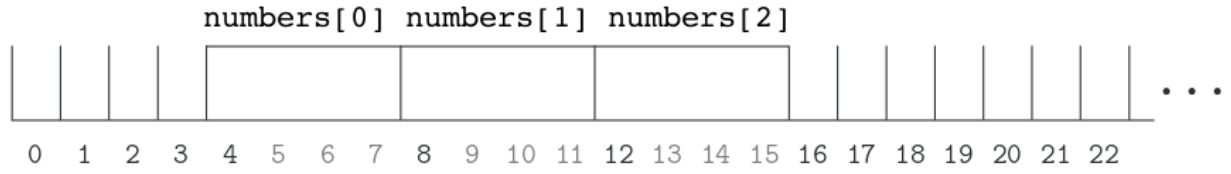
An array is a very simple **static data structure** (meaning it has a fixed number of elements), and you can access its separate elements with the [ ] operator. C++ inherits this structure directly from C, the ancestor of C++.

On this course, you will not need arrays themselves, because all the necessary things are easier to do with `vector`, but for your general education, it is good to know something about how the array works. You will need this information if you ever need to write programs with C.

The array type variable is defined like this:

```
int numbers[3];  // The array has space for three integers
```

As a result of the definition above, the compiler allocates enough consequent memory locations to store three `int` values, one after another.

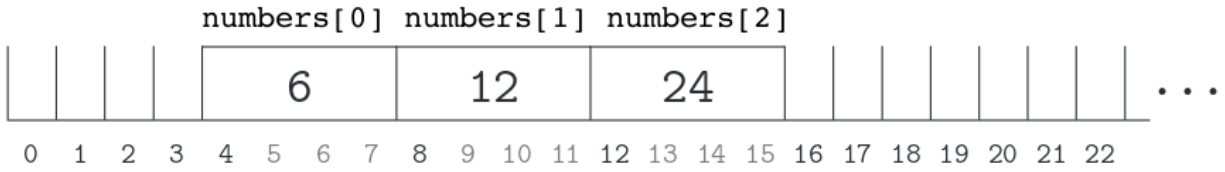


You can now use the [ ] operator to access the elements of the array:

```
numbers[0] = 6;
```

```
numbers[1] = 12;
numbers[2] = 24;
cout << numbers[0] + numbers[2] << endl;  // 30
```

and after that, the memory looks like this:



For efficiency reasons, the array type has been implemented so that it can be presented as a constant pointer to its first element.

Therefore, this:

```
cout << numbers << endl;
```

will print the address `0x000004` in the case of the exemplary array above.

Since the array is always understood as an address to its first element, it is a rather primitive data type:

- An array variable cannot be assigned into another array variable with the operator =. You cannot initialize an array with another array either.
- If you give an array as a parameter to a function, the array will always act like a reference parameter, because the function receives the memory address of the array's first element as the formal parameter. When this address is handled with the [] operator, we naturally end up operating with the elements of the original array.
- An array cannot directly be an element in an STL container.

Instead, you can go through the array with a pointer:

```
int* array_ptr = nullptr;
array_ptr = numbers;
while ( array_ptr < numbers + 3 ) {
    cout << *array_ptr << endl;
    ++array_ptr;
}
```

Above, we applied the so-called **pointer arithmetic**: You can add an integer into the starting address of the array, and the result is a pointer to an element whose index is the mentioned integer.

If the added value is the number of elements in the array, the result is a pointer that points to the memory location immediately after the last element:

```
cout << numbers + 3 << endl;  // Prints memory address 16
```

It is handy to use this address in the condition of a loop that uses a pointer to go through an array, just like we did above.

In practice, pointer arithmetic is also the reason why the code:

```
cout << numbers[2] << endl;
numbers[1] = 99;
```

means the same as the code:

```
cout << *(numbers + 2) << endl;
*(numbers + 1) = 99;
```

As we said earlier, arrays and their functionalities are not among the most essential content of this course. However, it is worth knowing them for the future. It is not completely unexpected that you end up writing some lower lever C code on the later courses (e.g. Microprocessors, Machine-Level Programming), where you will encounter arrays.

Dynamic data structures

Dynamic data structures are such structures, for which memory is allocated (and deallocated) based on the commands written by the programmer. In C++, these commands are `new` and `delete`, and they will be introduced more precisely on the next subsection.

Dynamic data structures are typically containers that contain several elements of the same type. Such an element can be defined as a C++ `struct` with several fields. The elements are linked together such that each element has a link field (pointer) to another element(s) of the same structure.

Examples of dynamic data structures are linked lists, stacks, queues, trees, networks etc. These structures are typically defined as classes, and they can be operated by the member functions specific for the structure in question. As with any class, the programmer is responsible for implementing the member functions.

From these structures, we will introduce a linked list more precisely on the upcoming subsection.

Dynamic memory allocation

Until now, we have only used automatic variables, which means that allocating (reserving) memory for them and afterwards deallocating (releasing) it was automated:

- At variable definition, the compiler has taken care of finding the necessary amount of memory somewhere and allocated it.
- When a variable reached the end of its **lifetime**, [1] the compiler, again, has deallocated the memory that was no longer needed.

However, there are many situations when a programmer has to be able to control the **lifetimes**of variables (starting from the allocation of memory for a variable and ending with the deallocation of the reserved memory) in order to create dynamic data structures. When the control over the lifetime of a variable is completely the programmer's responsibility, the variable is called a **dynamic variable**. The mechanisms and tools used for controlling dynamic variables are called **dynamic memory management**.

C++ has two basic commands you can use to allocate and deallocate dynamic memory: `new` and `delete`.

If there is not enough free memory space when the operation `new` is executed, an exception occurs and the program execution terminates. However, handling that exception is, even on a conceptual level, so challenging that, on this course, we will pretend that we never run out of memory.

The programmer can allocate a new dynamic variable with the command `new`:

- The lifetime of a variable begins at the moment when `new` succeeds to allocate memory for the variable.
- A dynamic variable does not have a name, but the operation `new` returns a pointer, the value of which reveals where the new dynamic variable is located within the main memory.
- To make the dynamic variable useful, its address (the return value of `new`) needs to be stored in a pointer variable.
- If the variable is a class-type one, `new` will take care of calling the constructor.

When you no longer need the dynamic variable, and you want to get rid of it, you deallocate the memory with the command `delete`:

- The variable reaches the end of its lifetime, and the memory that was allocated to it is deallocated for other uses.
- If a class-type variable has a destructor defined, `delete` will call the destructor.

Because the compiler does not automate anything when handling a dynamic variable, it is important for you to remember that each variable allocated with `new` must also be deallocated with `delete` when you no longer need it.

If you forget to do this or you mess things up and end up with a situation where [2] not all the memory allocated with `new` can be deallocated, what you then have is called a **memory leak**: The program continues to keep memory locations allocated even though it no longer needs them.

A memory leak is a bad thing, especially in programs with long execution times. If you constantly allocate memory but never deallocate some of it, the amount of main memory used by the program increases, and the whole system is burdened.

The first example about dynamic memory management:

```
int main() {
    int* dyn_variable_address = nullptr;
    dyn_variable_address = new int(7);
    cout << "Address: " << dyn_variable_address << endl;
    cout << "Start:   " << *dyn_variable_address << endl;
    *dyn_variable_address = *dyn_variable_address * 4;
    cout << "End:     " << *dyn_variable_address << endl;
    delete dyn_variable_address;
}
```

Creating the pointer variable `dyn_variable_address`, so that we have a storage place for the address of the variable we are going to create on the next line.

Allocating a new dynamic variable, and initializing its value to 7 and store its address for later use.

Using the dynamic variable with its memory address and the * operator.

When you no longer need the dynamic variable, you must deallocate the memory reserved for it with the `delete` operator.

When you compile and execute the code, you receive the following print:

```
Address: 0x1476010
Start:   7
End:     28
```

Here and in the other execution examples that print pointer values to the screen, please remember that the value of the memory address can be different at different executions.

---

[1] The local variables reach the end of their lifetime and deallocate the memory when the execution of the program leaves the block where the variable was defined. When the lifetime of an object ends, also the lifetimes of its member variables end, and memory is deallocated.

---

[2] You can, for example, accidentally lose the value of the pointer variable that stores the memory address of a dynamically allocated variable.

Linked list

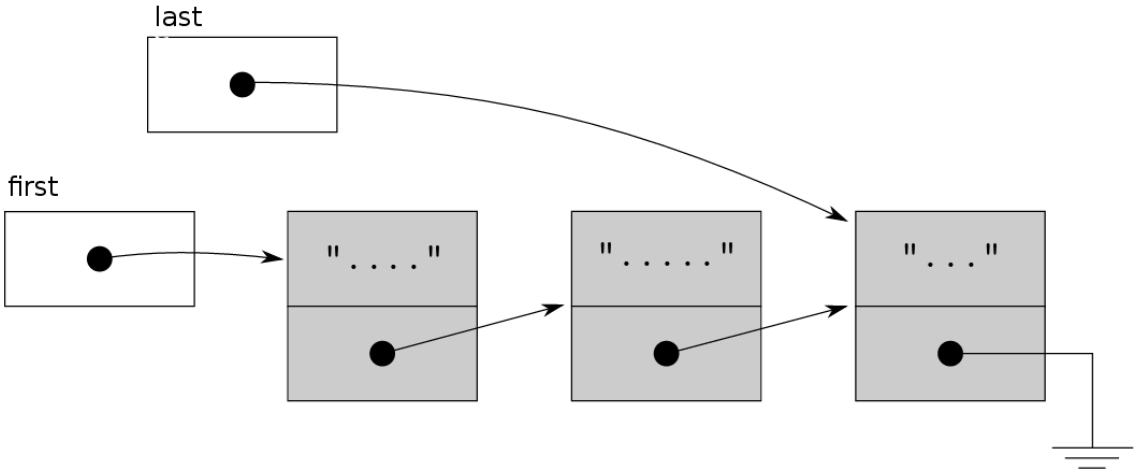
One basic mechanism for implementing dynamic data structures is the so-called **linked list**.

Linked lists consist of `struct`-type elements, each separately allocated with the command `new`. In addition to the actual value stored in the list, each element includes a pointer, by which the next list element can be found. We also need a separate pointer-type variable where we store the location of the list's first element. Sometimes, it is efficient to also have a second pointer variable to store the address of the last element in the list.

For instance, if we want to store text in a list so that each line of an email is a separate element of the list, we need the following `struct`-type and the "log" pointers:

```
struct List_item {
    string  text_line;
    List_item*  next;
};

List_item*  first;
List_item*  last;
```

When you add three elements into a list like this, the result is as:



In the illustration above, we emphasized the dynamically allocated `List_item` structs with a grey background. Also, the value `nullptr` is usually depicted as "grounding" in the illustrations.

Every time we insert a new element into the list, we use `new` to allocate space for the struct that includes the actual value we want to insert, as well as log data related to the implementation of this structure (that is, the pointer pointing to the next element of the list). Every time we remove an element from the list, we use the `delete` command to deallocate the space reserved for it.

Also, when inserting and removing elements, we have to update the pointer values so that they store the correct memory addresses.

A good way to implement list structures is hiding the abovementioned struct type and pointers pointing to the first and last elements of the list in the `private` part of the class, and implementing all the operations needed for handling the list as class methods.

Naturally, you are allowed to use even more complex structures (such as trees and graphs). Only the sky is the limit once you have understood the basics of handling link fields of struct type.

Task list with C++ pointers

Explore the project `examples/08/task_list` in Qt Creator. If you are also interested in executing and editing the program code, copy it under your `student` directory.

The project includes three files of program code:

- `list.hh`, the public interface of the list class
- `main.cpp`, the user of the class
- `list.cpp`, the implementation of the list class.

The main program includes nothing particularly interesting about dynamic memory management , which is why we will review the most essential parts of the list-class next.

list.hh

lines 12-13:

The purpose of the list is to insert new elements at the end and remove elements at the beginning. When an element is removed, its value is stored in the reference parameter`removed_task`.

line 17:

For the first time, we have to implement a destructor for a class. If the lifetime of a `List`-type object comes to an end and the list is not empty, the destructor deallocates the reserved cells in the list.

In C++, the destructor method shares its name with the class, but there is an additional tilde character at the beginning of its name. The destructor will be automatically called behind the scenes when an object comes to the end of its lifetime.

lines 20-23:

Define a struct-type that allows us to present the individual elements of the list. Because the type `List_item` has been defined in the `private` part of the class, we can only use it in the methods of the class.

lines 25-26:

The logging member variables that are supposed to store the location of the first and last elements of the list.

In this example, we keep track also of the address of the list's last element. This is sensible if you are going to add the elements at the end of the list. Why?

list.cpp

Constructor:

The constructor initializes both log pointer member variables with the value `nullptr`, which will later be interpreted to mean that there are no elements in the list (the list is empty).

Destructor:

When the `List` object reaches the end of its lifetime, the destructor is called automatically. It deallocates all the dynamically allocated elements (that are of the type `List_item`) still remaining in the list.

As the list was implemented in this example with the normal C++ pointers, deallocating the dynamic memory is the programmer's responsibility. Therefore, if we did not have the destructor, some memory would remain allocated.

The way the destructor works is that on each round of the loop, one element is deleted at the beginning of the list, and the memory allocated to it is deallocated.

Method `print`:

List printing has been implemented with a "standard mechanism" that works every time we need to go through the list's elements from the beginning to the end.

The temporary pointer variable is set to point to the first element of the list. For as long as its value is not `nullptr` (why will its value eventually be `nullptr`?), we print the contents of the pointed element. Finally, we move the temporary pointer to point to the next element on the list.

Method `add_back`:

It is easiest to understand the procedure of adding an element into the list if you draw a picture animating the steps you need to take.

The basic idea is as follows. Allocate dynamically a new struct of the type `List_item` and save its address. Then, update all the relevant pointers so that the list is preserved in the desired form. Please note that inserting needs to be done in different ways depending on whether we want to add the first element into an empty list, or whether the list already has elements. (Why?)

Method `remove_front`:

In this implementation, we always remove the first element of the list, the one where `first_` points to. Before removal, we must be sure that the list includes some elements. If it does not, we inform about an error (return value `false`).

The actual removing goes as follows. You remove the first element from the list, which means that you update the pointers to make the member variable `first_` point to the next element after the element you want to remove (or to `nullptr`, if we removed the last element of the list).

After updating the pointers, we deallocate the memory reserved for the element we want to delete.

Please note that removing an element is done differently when it is the only element of the list to when it is not.

Method `is_empty`:

This method is a simple one. The list is empty if `first_` is `nullptr`, just like it was with the constructor.

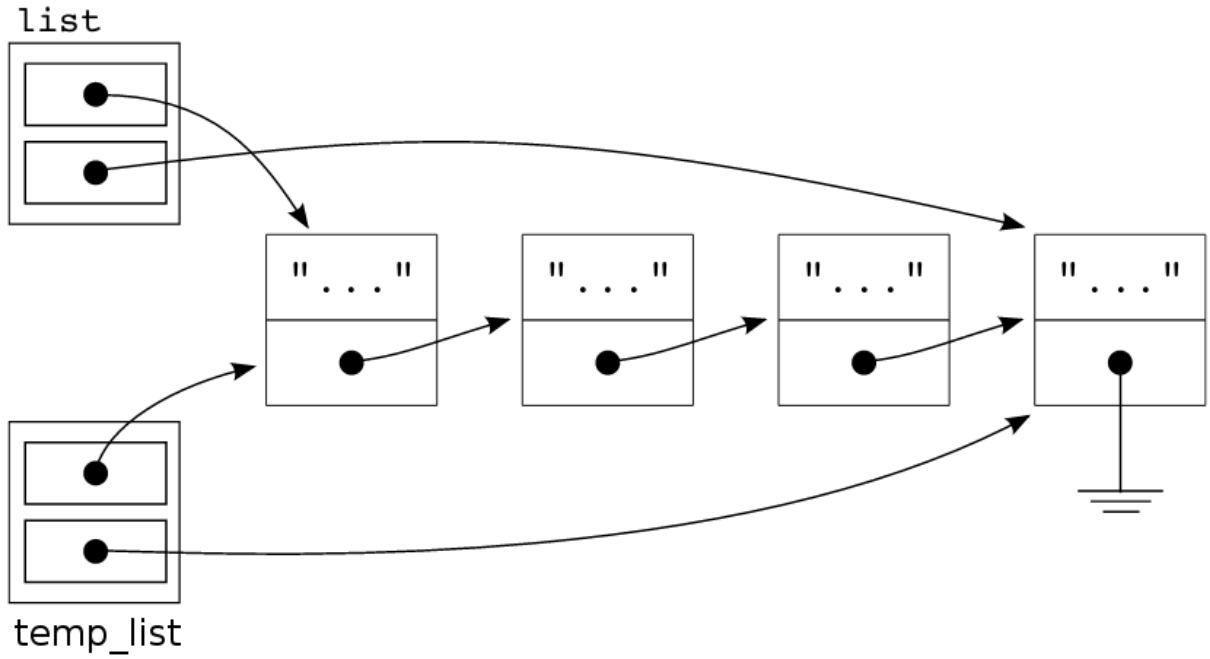
Summary

When implementing linked data structures, you must always keep in mind all the special situations you can encounter with data structures. In this example, inserting and removing must be done differently in different situations.

Inserting data into an empty list requires different actions to inserting into a list that is not empty. Consequently, removing the last remaining element of the list is different to removing the elements when there are several of them left.

In order for you to be successful with the most complicated list situations, it is good to pause while implementing your list and think about the situations you might have to consider:

- inserting the first element into an empty list
- removing the only remaining element from a list
- inserting an element to the beginning of a list
- removing the first element of a list
- inserting an element in the middle of a list
- removing an element from the middle of a list
- inserting an element at the end of a list
- removing an element from the end of a list.



The problem comes to life the instant we leave the block where we defined `temp_list` as a local variable: The lifetime of `temp_list` comes to an end, and then its destructor is called.

The destructor deallocates all the memory that was allocated to `temp_list`, which happens to be the same memory that we allocated to `list`. The member variables of the variable `list` remain pointing to the memory that we already deallocated (dangling pointers).

In the end, the point of the example above is this: It is easy to get into trouble if you use the default copy constructor and assignment operators of C++ when the value you want to copy includes pointers.

By the way, it is worth noting that we will mess up in the following example as well (why?):

```
void function(My_class formal_parameter) {
```

```
    ...
}
...
int main() {
  My_class actual_parameter;
  ...
  function(actual_parameter);
  ...
}
```

The possible solutions to our problem about assignment and initializing are:

- We disable the assigning and initializing of data types with self-made pointers altogether. This can be done such that the result of trying either of them will cause a compilation error.
- For these data types, we implement versions of the assignment operator and copy constructors that really copy the elements from one structure to another. That is, they allocate new memory for each copied element with the command `new`, and by doing so, construct a new copy of the original structure from scratch. This approach is called **deep copy**.

  (A mechanism where you only copy the memory address (which often leads to trouble), is called **shallow copy**.)

On this course, we will only get to know the easier way where you disable the copying altogether. The deep copy will be postponed for the later courses, because it will require explaining and understanding parts of C++ that are, for now, irrelevant.

Disabling the default copy constructor and assignment

You can easily disable the use of C++'s default copy constructor and assignment operation by writing the following additions into the public interface of a class:

```
class List {
  public:
    List(const List& initial_value) = delete;
    List& operator=(const List& assignable_value) = delete;
    ...
};
```

The types of the parameters must be `const`-references to the object of the class, and the assignment operator has to pass a reference to the object of the class.

On this course, it is good practice to do the disabling we discussed above every time you create a data structure that includes a dynamic data structure in its `private` part. When you do that, you avoid many errors and difficult debugging sessions.

Does this mean that it is not possible to define functions with self-made dynamic data structures given as their parameters? No, because it is still possible to define functions that do not need a copy constructor in their call: if the parameter type is reference or a `const`-reference. For example:

```
bool read_task_file(List& tasks);
bool save_task_file(const List& tasks);
```

Neither of them has any need for using a disabled copy constructor.

You need a copy constructor only when you are working with a value parameter, because the value parameter is initialized from an actual parameter by using a copy constructor.

(Q) Valgrind, the memory management analyser

Goal: I will learn to use `valgrind` for tracking the problems with memory management, and to interpret the output of `valgrind`.

This is important because, in the remaining projects, one requirement is that the submitted program has no problems with memory management. You are supposed to make sure that there are none by using `valgrind` during the test phase.

Instructions: Retrieve the code template: `templates/09/valgrind/` -> `student/09/valgrind/`.

Let us examine this small program. You can see several problems to do with memory management right away:

```
#include <iostream>

using namespace std;

int main() {
    int number1;
    int number2 = 111;
    int *ptr1 = new int;
    int *ptr2 = new int(222);

    cout << number1 << " "
         << number2 << " "
         << *ptr1 << " "
         << *ptr2 << endl;

    delete ptr1;

    *ptr1 = 333;
}
```

In the program consisting of a few lines of code, you can find errors with memory management just by looking at the code. When the size of the program grows, finding errors becomes more difficult.

You can trace the memory management errors with a program called `valgrind` that must be executed separately. It analyzes the code and prints an error message if, for example,

- your program allocates dynamic memory with `new` but does not deallocate it with `delete`
- your program uses a variable (allocated dynamically or automatically) without a value set to it
- you try to use dynamically allocated memory that has been deallocated already.

The tool `valgrind` is installed on linux-desktop. If you want, you also can install it on your own computer. You can execute `valgrind` in either Qt Creator or via the command line. For the sake of practice, let us now execute `valgrind` with both ways, so that we learn more than one way of using it.

Executing `valgrind` in Qt Creator

In Qt Creator, open the project you moved into the directory `student/09/valgrind/`.

If you want, you can execute the program. In the Application Output window, there should be the output ”exited with code 0”, that is, the execution of the program finished with the return value EXIT_SUCCESS, which means that everything went well (despite the program containing basically nothing but errors). Please note that when a program has as many problems with memory management as does our example program, it is not guaranteed that it will execute without hiccups. When the personnel created the assignment, it was possible to execute the program all the way to the end with the compiler installed on linux-desktop, but we cannot guarantee that it works similarly in all environments.

Execute `valgrind` by selecting `Analyze > Valgrind Memory Analyzer`. The program starts, but this time, the Application Output window will contain the output ”Analyzing finished”.

Also, Qt Creator opens the window Memcheck, and in its black top bar, it is supposed to say ”Memory Analyzer Tool finished, 8 issues were found”. We will examine the contents of that window in this assignment. If your Qt Creator shows nothing below the titles Issue and Location, click the filter icon located in the black top bar of the window and put a check on ”External Errors”. This will make the 8 problems `valgrind` found appear on the list.

Executing `valgrind` on the command line

Start the command line user interface and navigate into the directory where you saved the abovementioned program code (the project directory of the previous section).

1. Compile your source code manually by writing the compilation command into the Linux command line:

```
2. g++ -std=c++11 -Wall -g <cpp files>
```

   If your source code contained no errors, the result will be the executable program (a file in machine language) called `a.out`.

3. If you want, you can try executing the program by writing this command on the Linux command line:

```
4. ./a.out
```

5. Then execute `valgrind` command for the compiled program:

```
6. valgrind --quiet --leak-check=full ./a.out
```

If there are no memory problems in that program, there will be no extra outputs on your screen aside from the outputs of your program.

Because this example program did contain problems with memory management, `valgrind` prints an enormous amount of information, and we will now take a good look at it. You need to see the output from the very beginning, so scroll the terminal upwards far enough for you to see the `valgrind` command you wrote. (Tip: you also could stretch the terminal window as long as you can for easier working.)

Finally, try executing `Analyze > Valgrind Memory Analyzer with GDB` in Qt Creator. You will see that it starts up `valgrind` in debugging mode, which allows you to step through the program code line by line, like debuggers usually do. Now, the window Application Output has a print that looks the same as the one in the terminal window.

When quitting, please make sure you have closed the debugger! (We practiced this at the end of assignment 2.2.2.)

Usually, Qt Creator goes from Edit mode to Debug mode when you execute `valgrind`. Sometimes, Qt Creator does not do so, and to see the Memcheck window, you need to go from Edit mode to Debug mode yourself. When you encounter this ”characteristic” (?), it is good to understand that `valgrind` is always executed in Debug mode in Qt Creator.

Function pointers

When talking about programming languages, we often mention that the programs include data and functions. However, the division is not quite this straightforward. The functions can be data as well.

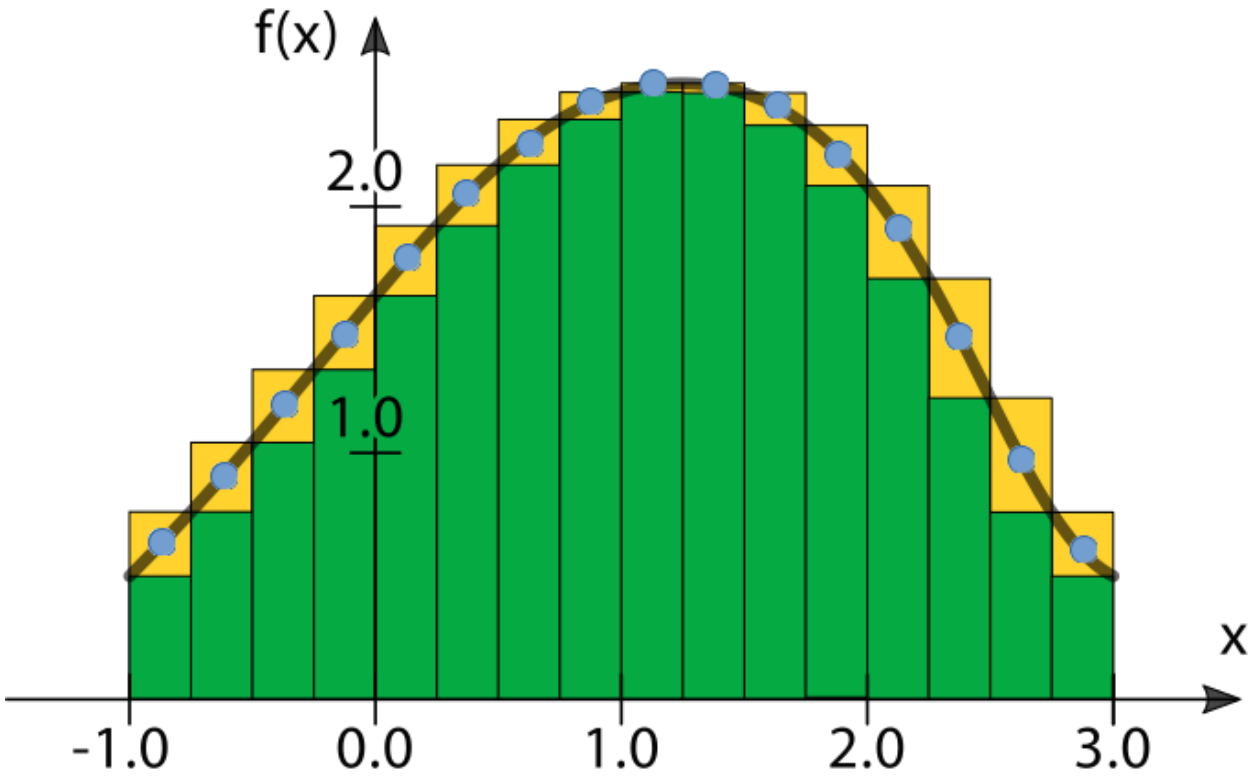
You might hear the phrase ”functions are first class citizens”. This means that you can operated with functions in the same way you do with the other data elements. You can, for example, pass a function as a parameter to another function, pass it as a return value of a function, or store it into a variable. In C++, you can do all this with the so-called **function pointer**.

A function taking another function as its parameter or returning another function is called a **higher-order function**. Functions that are not higher-order ones can be called as first-order functions.

Numeric integration

Let us study an example that you can find in the directory `examples/09/integral`.

The idea of numeric integration is to approximate the area between the graph of a function and the X-axis by dividing the area into narrow, rectangular areas, and calculating the sum of these rectangular areas.



In the illustration above, you can see two different ways of approximating the area between the graph, drawn in black, and the X-axis. The result of calculating the sum of the areas of the green rectangles is a lower sum of the approximate value of the area. The yellow rectangles are partly hidden behind the green ones, but by summing their areas we can get the upper sum for that same approximate value of the area. We could get a closer approximate value by increasing the amount of rectangles (i.e. by decreasing their widths).

In our example program, we do not calculate the lower sum or the upper sum. Instead, the height of a rectangle is defined by the value of the function at the center of the rectangle. In the illustration, these points are marked with blue dots.

The example program implements the function `integrate`, the first parameter of which is the function `f` that we want to integrate:

```
double integrate(Func f, double left, double right, int number_of_partitions = 500);
```

The implementation of the function uses the following definition:

```
using Func = decltype(&polynomial);
```

The command `decltype` reveals the data type of any variable or function. We also could use the type definition syntax of C++ for the definition:

```
using Func = double(*)(double);
```

meaning, `Func` is a pointer to a function that returns a double and also gets a double as a parameter. The first double in the definition is the return value, `(*)` means the function pointer, and within the parentheses, there is the list of parameters.

Certainly, it would be possible to leave out the statement `using`, and to define the function `integrate` in the following way:

```
double integrate(double(*f)(double), double left, double right, int number_of_partitions = 5
00);
```

Please note that we do not simply replace the identifier `Func` with the contents of the right side of the assigning operator in the statement `using`. Pay attention to where the name of the function pointer `f` is written.

When we call the function `integrate`, we must give it a pointer to a function as a parameter, like below:

```
cout << integrate(sin,        0,  2) << endl;
cout << integrate(cos,        0,  2) << endl;
cout << integrate(sqrt,       0,  2) << endl;
cout << integrate(polynomial, 0,  2) << endl;
```

Please note that the examples above do not have parentheses after the first parameter, because they are not function calls, they are pointers to a function.

Purposes of function pointers

We will get back to function pointers during the next rounds. Function pointers are useful, for example, in the following situations:

- We want to choose the functionality to be executed in a program, but we do not want to include a very long `if` structure, with each block containing one of the possible functionalities (see the example `examples/04/datadrivenprogramming`).
- We are implementing graphical user interfaces and we want to bind a functionality to a component of the user interface.

Smart pointers

Although all the actions with the dynamic memory can be implemented with the C++ pointers and the commands `new` and `delete` we presented to you earlier, using them is often pretty messy. Especially when handling complicated dynamic structures, we often end up in a situation where we allocate memory with `new` but do not remember to use `delete` to deallocate it. To make this duty easier, C++ includes so-called **smart pointers** among its tools.

The smart pointers of C++ are library data types that automate the deallocating of memory when nothing points to it anymore. In plain language: the allocated memory will be automatically deallocated when there are no (smart) pointer variables pointing to that memory location in the program.

Smart pointer types are great because you use them mostly like you use normal (raw) pointers, and on top of that, you do not have to worry about deallocating memory.

To use smart pointers, you must include this line in the beginning of your program:

```
#include <memory>
```

which lets us use the types:

```
shared_ptr
unique_ptr
weak_ptr
```

On this course, we will only get to know the type `shared_ptr`.

`shared_ptr` pointers

A simple example on the use of `shared_ptr`:

```
#include <iostream>
#include <memory>  // You must remember this.

using namespace std;

int main() {
     shared_ptr<int> int_ptr_1( new int(1) );
     shared_ptr<int> int_ptr_2( make_shared<int>(9) );

     cout << *int_ptr_1 << " " << *int_ptr_2 << endl;
     cout <<  int_ptr_1 << " " <<  int_ptr_2 << endl;
     cout <<  int_ptr_1.use_count() << " " <<  int_ptr_2.use_count() << endl << endl;

     *int_ptr_2 = *int_ptr_2 - 4;
     int_ptr_1 = int_ptr_2;

     cout << *int_ptr_1 << " " << *int_ptr_2 << endl;
     cout <<  int_ptr_1 << " " <<  int_ptr_2 << endl;
     cout <<  int_ptr_1.use_count() << " " <<  int_ptr_2.use_count() << endl;
}
```

Defining a pointer variable of the type `shared_ptr`, into which we can store the memory address of a `int`-type variable. Initializing it to point to the variable that is reserved dynamically with `new` and that has the value of 1.

The same action as on the line above, but we use the pointer formed by the function `make_shared` as the initial value of the `shared_ptr` pointer. The difference to the previous one is that this way is faster.

`shared_ptr` pointers are used mostly in the same way as normal pointers.

Calling the method `use_count` to print the value of the reference counter for both pointers. The method `use_count` returns the number of `shared_ptr` pointers pointing to the same memory location as the target object of the `use_count` method.

When the value of the reference counter reaches zero, the allocated memory will be deallocated automatically.

The most relevant part about the example: we set `int_ptr_1` to point to the same memory address as `int_ptr_2`. Now, there are no `shared_ptr`-type pointers pointing to the allocated memory where `int_ptr_1` originally pointed to: **the allocated memory is automatically deallocated**.

The lifetime of the variables `int_ptr_1` and `int_ptr_2` ends, which means that the memory area they pointed to has no `shared_ptr` pointers: **the allocated memory is automatically deallocated**.

The execution of the program creates the following prints:

```
1 9
0x2589010 0x2589060
1 1

5 5
0x2589060 0x2589060
2 2
```

Please note that the example does not have any `delete` commands, even though dynamic memory is allocated both with `new` and the function `make_shared`. This is precisely the idea of the smart pointers, i.e. moving the responsibility for deallocating dynamically allocated memory to `shared_ptr` objects. We often use the term **owner**, which is just a fancy term to describe whose responsibility memory deallocation is.

In the example above, we compared using `shared_ptr` pointers to using normal pointers. Almost all the operations (the unary `*`, `->`, comparison, and printing) that work with normal pointers work with `shared_ptr` pointers as well. The greatest difference is that the operators `++` and `--` do not work with `shared_ptr`. Also, assignment is possible if you do it from another `shared_ptr` of the same type.

Below we list a couple of useful characteristics of `shared_ptr` that you might have use for later:

- If you want to get the memory address from a `shared_ptr` pointer as a normal C++ pointer, you can do it with the method `get`:

  ```
  shared_ptr<double> shared_double_ptr( new double );
  ...
  double *raw_double_ptr = nullptr;
  ...
  raw_double_ptr = shared_double_ptr.get();
  ```

- You cannot use the operator `=` to assign a normal pointer to a `shared_ptr`pointer.
- However, you can assign a `nullptr` to a `shared_ptr` pointer.
- A `shared_ptr` pointer cannot be directly compared to a normal pointer. The comparison is possible if the `shared_ptr` is changed into a normal pointer with the method `get`, for example:

  ```
  if(raw_pointer == shared_pointer.get()) {
      ...
  }
  ```

- A `shared_ptr` pointer can be compared with a `nullptr`.

The type `shared_ptr` has a troublesome characteristic: If you create a ”loop” of them, the memory will never be deallocated:

```
#include <iostream>
#include <memory>

using namespace std;
```

```
struct Test {
    // Other fields
    // ...
    shared_ptr<Test> s_ptr;
};

int main() {
    shared_ptr<Test> ptr1(new Test);
    shared_ptr<Test> ptr2(new Test);

    ptr1->s_ptr = ptr2;
    ptr2->s_ptr = ptr1;
}
```

It is useful to draw a picture of the situation above so that you understand the kind of ”egg or chicken” problem we have here. The memory that `ptr1` points to cannot be deallocated because the pointer `ptr2->s_ptr` points to it. Then again, the memory that `ptr2` points to cannot be deallocated either because `ptr1->s_ptr` points to it.

Task list with `shared_ptr` pointers

See the project `examples/09/task_list_v2` in Qt Creator. If you are also interested in executing and editing the program code, copy it under your `student` directory.

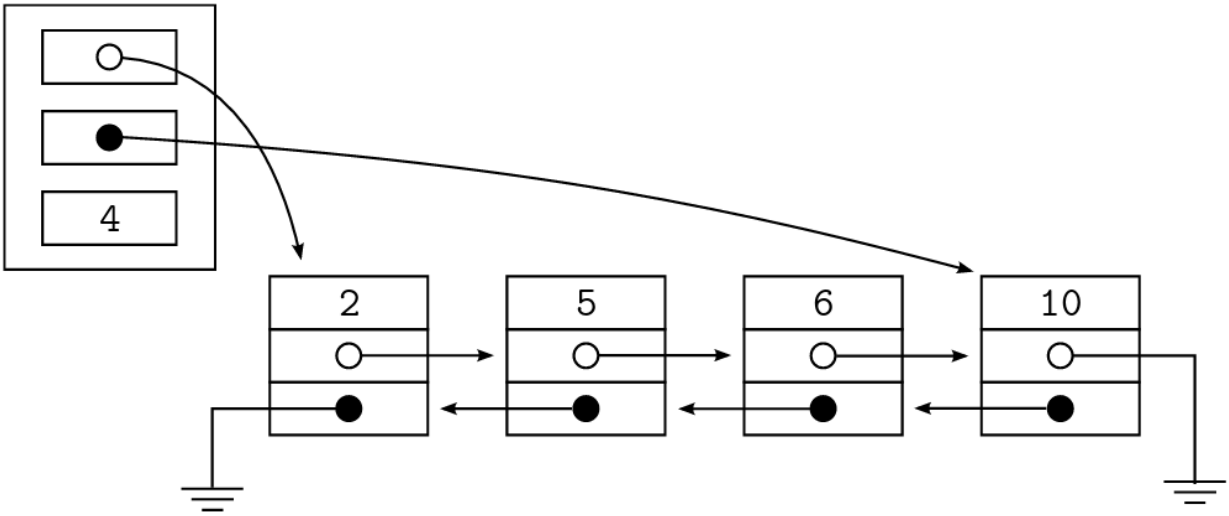
The project includes an implementation of a list structure that uses `shared_ptr` pointers, equivalent to an example on the previous round. The only algorithmically new thing in the modified example is that there is no need to implement a destructor for the `List` class (or to use `delete` commands on any other occasion) because the `shared_ptr` pointers deallocate memory when nothing points to it anymore.

Please note that all the different situations in handling a linked list must be taken into account, just like when using normal pointers (inserting the first element into an empty list, removing the last element from the list, etc).

Doubly-linked list

Let us implement a set of integers as a doubly-linked list in which the elements are stored in ascending order. Also, if you attempt to add an integer that is already in the set, the addition fails (this means that each integer can be in the set only once).

For example, a set consisting of the elements 2, 5, 6, and 10, looks like this:



In the illustration above, `shared_ptr` pointers are depicted as white circles, and normal pointers as black circles.

In the program code, the member variable pointing to the first element of the list as well as the field in each element that points to the next element are implemented with the type `shared_ptr`. The reason is that this way, you do not have to deallocate the elements of the list with `delete`.

Then why has it been smart to implement the member variable pointing to the last element and the field pointing to the previous element in the list as a normal pointer?

Examine the example program that you can find in the directory `examples/09/two_way_list`. The implementation of the list is in the files `two_way_list.hh` and `two_way_list.cpp`.

The code is quite long, and it is not reasonable to go through it line by line here. The functions dealing with addition and deletion have comments on most of their important parts, which means that you are able to study them by yourself. However, here are some overall notions about the implementation.

- It has been necessary to implement one of the link fields (i.e. pointers) in the list elements as `shared_ptr`, and one as a normal pointer, because of the following reasons.
  - By using `shared_ptr`, the programmer does not have to remember to deallocate dynamic memory.
  - On the other hand, if both of the pointers in the list element were `shared_ptr`-type, the memory would never be deallocated, because consequent elements would point to each other.

    Overall, this is the weakness of `shared_ptr`: Using it alone, you cannot implement structures that include pointer loops (A points to B and B points to A, or a longer chain).
  - Then again, the member variable pointing to the last element of the list is a normal pointer, because when you remove the last element (lines 103-104 in `two_way_list.cpp`) from a non-empty list, that member variable must be set to point to the originally second-to-last element (i.e. the previous element of the element under removal):

    ```
    last_ = last_->prev;
    ```

    If `last_` were not a normal pointer, we would end up in a situation where we would have to assign a normal pointer into a `shared_ptr` pointer somewhere else than at initialization. This is dangerous, and almost always leads to problems.

- Every time you need to handle `shared_ptr`-type pointers in the same expression as normal pointers, using the method `get`, you can get the memory address object that `shared_ptr` is pointing to as a normal pointer. The value you receive can be assigned into a normal pointer variable, and you can compare it with a normal pointer.

  You must not deallocate the normal pointer you receive from the method `get`with `delete`, because that will mess up the internal log of `shared_ptr`.

- It is worth noting that with adding as well as with removing, we had to examine several very special cases which we talked about at the end of the material section 8.4.2.

- Every time we handle a pointer structure that is even a little bit more complicated than a singly-linked list, we almost always end up with expressions of the following form:

  ```
  a->b->c
  ```

  In other words, the operator `->` must be written several times in the same expression.

  In fact, there is nothing odd about it. The C++ rules on the order of operation dictate that the expression is interpreted from left to right.

  If it will make this easier, you can think of replacing the arrow operator with the text: ”pointing to a field of a struct called”. The example you just saw would be interpreted as: the value of *a*, pointing to a field of a struct called *b*, pointing to a field of a struct called *c*.

Modularity

Modularity is a mechanism that divides a large program into small, more easily manageable parts when designing and implementing a program. If a program is modular, it has been divided into distinctly considered parts, and the result of their combined functioning is the final program.

Module is a whole consisting of cohesive programming structures. In most programming languages each module is implemented as a separate source code file, or a pair of source code files.

To some extent, modules are similar to classes: Each module has a public and a private interface. Classes and modules, however, are not the same thing. The only similarities are that both are used through their public interface. Also, if a class forms a distinct part of the program, it is often reasonable to implement it as a module, just like in the examples we are going to show you.

Mini example: geometry calculations

Let us study an example that shows us how to implement the basic, mechanical details of modules. The program of this example is a simple prototype of geometry calculator. It consists of two modules: a main program, and a module containing geometric calculations.

The main program module, located in the file `calculator.cpp`, looks like this:

```
// Module: calculator / file: calculator.cpp
// Provides the main function for a geometry calculator.
#include "geometry.hh"
```

```
#include <iostream>

using namespace std;

int main() {
    double dimension = 0.0;

    cout << "Input the length of the side of a square: ";
    cin >> dimension;
    cout << "Perimeter: " << square_perimeter(dimension) << endl
         << "Area:      " << square_area(dimension) << endl;

    cout << "Input the radius of a circle: ";
    cin >> dimension;
    cout << "Perimeter: " << circle_perimeter(dimension) << endl
         << "Area:      " << circle_area(dimension) << endl;
}
```

In C++, each module providing services to other modules in its public interface needs a **header file** that describes all the services included in the public interface of the module.

In our example, the module `geometry` offers functions to the main program for calculating the perimeter and the area of certain geometric shapes. The header file of the module then consists of the declarations of the functions required by the main program.

```
// Module: geometry / file: geometry.hh
// Header file of module geometry: provides the declarations for functions
// needed in calculations concerning geometric shapes
#ifndef GEOMETRY_HH
#define GEOMETRY_HH

double square_perimeter(double side);
double square_area(double side);

double circle_perimeter(double radius);
double circle_area(double radius);

#endif // GEOMETRY_HH
```

Now, at the latest, is the time to notice that the main program module (`calculator.cpp`) includes the following line:

```
#include "geometry.hh"
```

This is the directive of the preprocessor that tells the C++ compiler that a module is using services from the public interface of another module. Having added the line `#include` at the beginning of the file `calculator.cpp`, you can call the functions declared in `geometry.hh` even if their definitions are located elsewhere.

A preprocessor is the part of the compiler that prepares the files before the actual compiling. Basically, the preprocessor only replaces text with other text, such as replacing the line `#include "geometry.hh"` with the contents of the file `geometry.hh`.

Because `#include` merely joins the files as part of a file that will be compiled, it is possible for a file to be included several times, and the overlapping definitions prevent compilation. The problem can be solved by writing the directives `#ifndef`, `#define`, and `#endif` in the

header file as shown in the code above. They let the preprocessor know that the contents of a file will only be included in the compiled file once, regardless of how many times it is attempted to include them.

According to the example, after the lines `#ifndef` and `#define`, there is the name of the header file in question, written in capital letters, and with the period replaced with an underscore character. This is a common stylistic habit in programming, and you should follow it as well.

There is usually no need to write a separate header file for the main program module because it does not offer services to other modules.

The implementation file `geometry.cpp` includes the definitions of the functions declared in the module's public interface, and the assisting services of the private interface they need.

```
// Module: geometry / file: geometry.cpp
// Implementation file of module geometry: provides the implementations
// for functions needed in calculations concerning geometric shapes
const double PI = 3.141593;

double square_perimeter(double side) {
    return 4 * side;
}

double square_area(double side) {
    return side * side;
}

double circle_perimeter(double radius) {
    return 2 * PI * radius;
}

double circle_area(double radius) {
    return PI * radius * radius;
}
```

Compilation phases of the geometry calculator

Qt Creator will make sure the programs are compiled correctly. We will now study the phases of compiling a program. We will compile the geometry calculator you saw above using the command line, because this way, you can see the phases better.

Copy the directory `examples/10/geometry` into the directory `student` for compilation. You can see that this directory does not include the Qt Creator file `.pro`.

The simplest way to compile the program is to go to the directory containing the files and write the command:

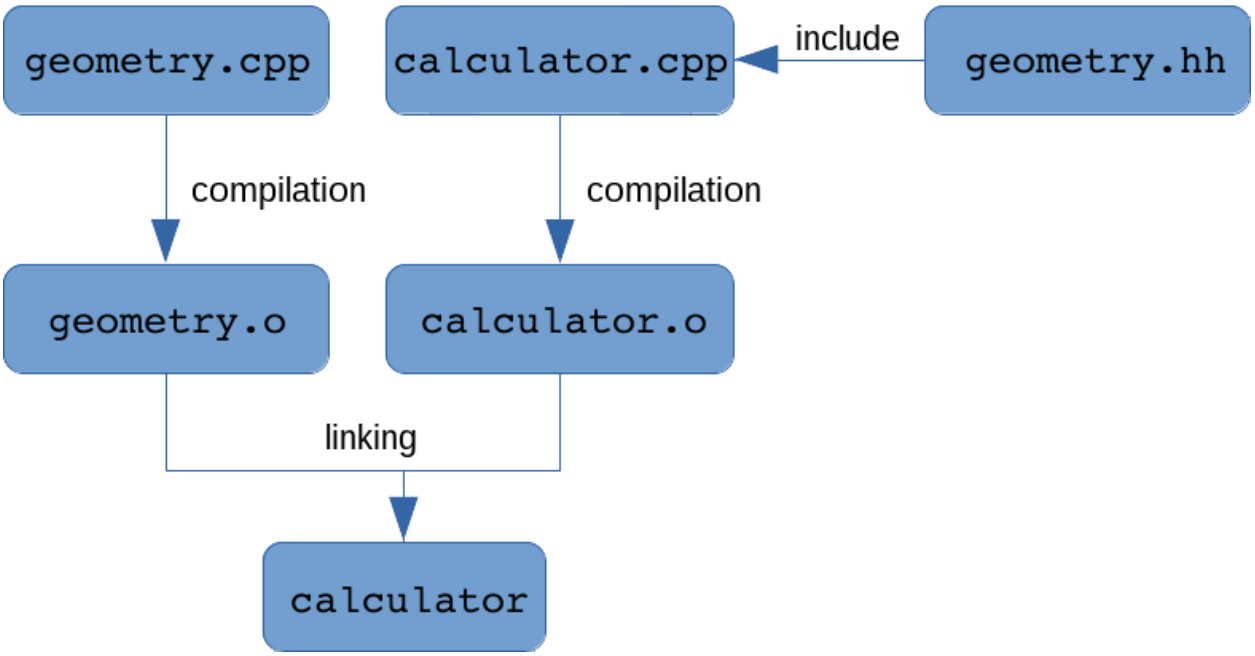
```
g++ calculator.cpp geometry.cpp
```

Please note that you have not the file `geometry.hh` in your compiling command. Why not? This file will be taken in compilation because the preprocessor includes its contents in the file `calculator.cpp`.

If you list all the files after the command, you can see the file `a.out` in the directory. This is an executable program (*binary*). You can execute it on the command line by writing the command:

```
./a.out
```

This compilation process, while being the simplest possible, does not reveal more about compilation phases than if you did it in Qt Creator. Delete the binary file (the command you need is `rm`) and start over to compile phase by phase.



Compilation phases of the geometry calculator

First, we only compile the file `geometry.cpp`. You can do it with the command:

```
g++ -c geometry.cpp
```

Here, the command line parameter `-c` (*compile*) lets the compiler know that we are only going to compile, not link. If you list the files after that, you will see that the file `geometry.o` has appeared in the directory. This file includes machine code, but you cannot execute it alone because it does not contain a complete program. As you might know, the file `geometry.cpp` does not include the `main` function at all, for example.

Next, we will compile only the file `calculator.cpp`, with a similar command:

```
g++ -c calculator.cpp
```

After that, you have two object files with the extension `.o` in the directory.

The last phase of compilation is that you combine the object files into an executable file. You can do it using the following command:

```
g++ calculator.o geometry.o
```

If you want to name the binary something other than `a.out`, you can give the parameter `-o name` to the compilation command, for example:

```
g++ -o calculator calculator.o geometry.o
```

When compiling large programs, it can make things easier if you separate the compiling and linking phases. This way, the programmer will always know which stage gave the error message when executing the compilation.

Qt Creator uses a program called `make` to automate the compilation. It is also possible to use `make` from the command line. If you end up working on a project that is compiled and executed from the command line, you should familiarize yourself with `make`.

More complex example: bus timetables

See the directory `examples/10/bus_timetables` and find the program located in it that asks for the time and bus number, and then prints the three closest departure times of that bus. (This directory does not include the Qt Creator file `.pro` either, which means you have a chance to practice compiling a program from the command line.)

The program consists of three modules:

`timetable.cpp`

The main program module includes the initialization of the `Timetable` data structure, a very simple user interface, and the search of bus departures, based on the time and bus number the user has given.

It uses the services in the public interfaces of both the module `time` and the module `utilities`.

`time.hh` + `time.cpp`

The module defines the class `Time`, making it possible to handle times: initializing, setting, reading from the keyboard, printing, and comparing (less or equal) two objects of the type `Time`.

It uses the services in the public interface of the module `utilities`.

`utilities.hh` + `utilities.cpp`

The module provides the functions to transform a string into an integer, and to read an integer from the keyboard. The services offered by the module are somewhat miscellaneous functions that cannot really be included in a more suitable module.

It does not use the services offered by the other modules.

The program might seem complicated at first, but apart from a couple of sections, it does not contain anything new. The sections containing new things are marked with the comment `//***`, and we will explain them here:

Line 52 in the file `timetable.cpp`

You should remember that the easiest interpretation of a reference is giving a nickname to an existing variable. From this point of view, the definition:

```
const vector<Time>& timevec = iter->second;
```

simply means that from now on, the name `timevec` can be used to mean the same thing as `iter->second`. The definition makes it possible to simplify the code, and diminish the trouble of writing.

The word `const` is included here because `iter` points to the `const` container `timetable`, which you can only handle as constant.

Line 7 in the file `utilities.cpp`

The reserved word `namespace` is used to create a private interface for the module, whose services can only be accessed from within the same module (source code file).

All the declarations and definitions enclosed by the curly brackets after the reserved word `namespace` can only be used in the file `utilities.cpp`.

An unnamed namespace of a module is equivalent to the `private` part of a class.

Line 11 in the file `time.cpp`

This is the first example of a class constructor where the initialization of an object does not happen in the initializing list but in the commands located in the body of the constructor. There is nothing odd to this in itself, since the constructor is a function, and its body can contain commands when necessary.

However, with the know-how you have at this point of the course, there is a problem with the implementation of the constructor of the class `Time`: What should be done if the parameter `time` is not correct? The function `set_value` does return the value `false` in that kind of a situation, but the constructor cannot use that value because it does not have a return value itself.

The correct solution is to create an exception, as we did in Python in similar situations. We just cannot yet do it in C++ (but you will learn it only on the next programming course).

> Everything else in the examples should already be familiar to you. We may have used some of the mechanisms you learned earlier in some creative ways, so you should pay attention when you read about them.

Module's public interface (.hh file)

> The public interface of a module (this means the services one module provides for other modules) is written in the header file of the module which is traditionally named with the extension `.hh` in C++.
>
> A public interface can contain:
>
> - declarations of functions
> - definitions of constants
> - definitions of new data types (also classes)
> - any combination of the above things.
>
> A public interface **must not include**:
>
> - definitions of variables
> - definitions of functions or class methods.

Each source code file that needs a service from the public interface of another module has to have the following line:

```
#include "module_proving_services.hh"
```

Even if it was not necessary to do this in the example code, it might be necessary to put this `#include` line into the `.hh` file of a module as well. In what kind of a situation would you have to do that?

Also, it is common that the `.cpp` file of a module has to include its own `.hh` file (see the `time.cpp`file in the example). Why is this necessary?

You **must not use** the directive `#include` for including `.cpp` files. Even if that might work in some situations, it is a sign that the programmer is largely ignorant of how to use module mechanism.

With regard to programming style, it is considered good policy to include your own modules first when a module includes both the modules written by the programmer and the standard libraries of the system, as below:

```
#include "my_module_1.hh"
...
#include "my_module_n.hh"
#include <standard_library_1>
...
#include <standard_library_m>
```

This way you will ensure that the compiler checks whether the self-written modules have all the `include` files you need. In other words, you make sure that your own modules form a compilation unit that can be compiled independently.

When working with complex programs, you often face the situation where one `.hh` file is included several times because different source code files execute the `#include` command to it to fulfill their own needs. In certain situations, that leads to problems. You can work around this problem as explained at the beginning of this material section when the mini example on geometry calculations was considered.

Module’s private interface (.cpp file)

The implementation file, or the `.cpp` file, defines all the functions and methods that were declared in the module’s public interface in the header file. The implementation file can, for certain, include any code that is necessary for implementing the functions of the public interface.

If you want to implement some assisting functions of the module itself in the `.cpp` file, and want to make sure you cannot find a way to call them from other modules, you must declare and define them within the unnamed namespace:

```
namespace {  // Declaration part
    void private_function_of_the_module();
    ...
}
```

```
...
the function definitions of the public interface are here
...

namespace {  // Definition part
    void private_function_of_the_module() {
        ...
    }
    ...
}
```

As you know, the declaration part can be omitted if you arrange the function definitions in such an order that enables the compiler to see the function definitions before you try to call them.

The way to do this is to move the definition into the beginning of the file, as we did in the file `utilities.cpp`, starting from line 7.

The mechanism of the unnamed namespace presented above only works with the regular functions. Instead, if you define methods of a class in the `.cpp` file, the `namespace` mechanism has no meaning, because a class uses the `public` and `private` mechanisms to manage its interfaces and scopes.

In the module's public interface (the `.hh` file), if there are definitions of constants or data types that you need in the `.cpp` file of the module as well, you have to include the module's own `.hh`file.

The programmer can also define named namespaces:

```
namespace my_space {
    void my_function() {
        ...
    }
}
```

You can call functions like these in two ways; either:

```
my_space::my_function();
```

or by adding the command `using namespace` into the code file:

```
using namespace my_space;
...
my_function();
```

You will probably not need to implement a named namespace on this course, but it is a nice explanation to the frequent code snippet:

```
using namespace std;
```

You can avoid name conflicts by using named namespaces. For example, the identifier `name` could be used in a student database both in student information and in course

information. After implementing both in their own namespaces, you could access them as `Student::name` and `Course::name`.

How to design modules

In a real programming project, you divide the program into modules at design phase. Basically, you can find the modules when you consider what the logical sub-parts of the program you are implementing are.

The bus timetable program we used as an example is almost laughably simple and so small that the module division barely benefits us at all, but off the top of our heads (before coding anything), we thought of these:

- The program must be able to handle and compare clock times.
- Because it must be able to read numbers presented in different forms from the keyboard and conduct changes between strings and numbers, it seemed pretty clear that we need a set of functions that will manage the numbers put in by the user.
- We also need a very simple user interface.
- Also, we need an algorithm that is able to get the suitable bus departures from the data structure the program is using.

Finally, mostly for technical reasons, we decided to implement the user interface and the search algorithm as the main program module, and the two others as modules of their own, which leads us to the final module division of the program:

- module for the main program (`timetable`)
- module for time management (`time`)
- an assisting module for number management (`utilities`).

The goal in finding the modules is to divide the program into parts, each of which:

- implements a distinctly divided part of the whole, and
- is simple enough (what if it is not?).

If the problem is on the larger side, you can look for the modules by splitting the problem and splitting these sub-problems into even smaller parts, until you have sub-problems that are small enough to be easily managed. This approach is called the **top-down** method.

Here are some common guidelines for entities you should usually implement as modules when working with programs of the same size as the ones on this course:

- class
- main program
- user interface or handling a complex input
- reading and analyzing a file
- common algorithms (searching, sorting).

Imagination helps a lot.

Note that we could have implemented the modules of the bus timetable program as classes with their public and private interfaces. Classes are the most typical examples of modules in C++ programming.

Designing the public interfaces of the modules is more challenging than designing the modular division.

Especially after gaining some experience, you are usually able to see the modular division almost immediately when working with programs of the size of the ones on these basic courses.

When designing a public interface, you need to take a much firmer stand on the services a module is going to provide for others, and how it will do it. In order to do that and succeed in some way, you will need to consider the structure of the program and its implementation quite carefully.

If an interface has been designed the ideal way, each module provides services to other modules in its public interface, which you do not need to edit while the project advances. Such a successful result is rarely seen. In practice, as the project proceeds, you always encounter situations where you understand that something was done badly in the interface, or that something was forgotten from it completely. In these situations, you must change the interfaces, and it can get expensive if someone has already been busy writing a lot of code based on the imperfect interface. The changes on the interface are reflected everywhere it has been used.

Benefits of modularity

The benefits of modularity are mostly the same as the benefits of classes, because they mostly stem from the use of interfaces:

- The implementation of a module (that is, the `.cpp` file i.e. the private interface) can be modified while the public interface stays intact.
- The parts of the program that logically belong together can be combined in the same package, which simplifies the program and makes its testing and managing easier.
- The modules can be developed in the project side by side after agreeing on the public interface.
- Modularity is a good tool in managing large programming projects.
- Often, you can reuse whole or partial modules.
- Most programming languages that support modularity allow you to compile modules separately. This speeds up the developing, and uses less resources, since after completing the changes, you only need to re-compile the modules that were modified.
- From the examples of this round, you can find the directory `examples/10/fraction` containing the class `Fraction`.
- The fraction class is an example on abstract data type. At the same time, it acts as an example on overloading operators.
- The class `Fraction` has two attributes: `numerator_` and `denominator_`. It has overloaded operators for equality comparison, addition, and multiplication.

- When we overload addition and multiplication operators, we can use them in the same way as using the corresponding operators for primitive types. For example:

```
Fraction frac1(2, 3); // creating the fraction 2/3
Fraction frac2(3, 4); // creating the fraction 3/4
Fraction frac3 = frac1 + frac2;
```

- Of course, it would be possible to use the operator function in the same way as any function:

```
frac3 = frac1.operator+(frac2);
```

- From the above kind of calling convention, we can see that `frac3` corresponds to the return value of the function, `frac1` is the target object of the call, and `frac2` is the parameter. It also reveals that the first operand of addition must be an object, not a fraction literal. We could implement operator overloading without such restriction, but we can do with our solution.
- Note that we can combine overloaded operators in usual way, which means that we can have several additions and multiplications in the same expression.
- Also the output operator `<<` has been overloaded. Overloading this operator has the benefit that we can print the instances of programmer-defined abstract data types in the same way as other printable values. For example:

```
Fraction frac(2, 3); // creating the fraction 2/3
cout << "The value of frac is " << frac << endl;
```

- Overloading the operator `<<` is different from overloading arithmetic operators. The operator `<<`is a method of the class `ostream` (and `cout` is an instance of that class). Now, the target object is not an object of the `Fraction` class.
- To make the above kind of natural printing way possible, it pays off to implement the overloaded output function as an independent function that is not a method of any class. However, the output function needs the values of the private attributes `numerator_` and `denominator_` of the `Fraction` class. For this reason, we need the methods `getNumerator` and `getDenominator`.
- The first parameter of the output function is the output stream. The second one is an instance of the programmer-defined class, which in this case is an instance of `Fraction`. Note also that the output function returns the output stream that it has received as a parameter. This enables chaining of the operator `<<`, which can be seen in the latest code fragment above.
- In the same way, we could also overload the input operator `>>`. However, we have not done so, instead the input is read in the main program. The reason is that in this way, we can check that the user does not give zero as a denominator.
- The `Fraction` class has two private methods: `gcd` and `reduce`. The method `reduce` transforms a fraction into as reduced shape as possible. Such is needed, for example, in equality comparison. In addition, the result of both addition and multiplication is given in the most reduced form.

- Reducing a fraction is implemented by first counting the greatest common divisor (gcd) between the numerator and denominator, and then both of these are divided by the counted divisor. Here we need the method `gcd` that has been implemented recursively.
- All in all, the fraction example shows how a programmer-defined class and instances of that class can be used very much the same way as ready-made primitive types.
- The implemented `Fraction` is not perfect. As we have overloaded the addition operator, when it is possible to use `+` symbol in a usual way, the user of the class can easily assume that they can also use the operator `+=` as usual. However, this is not the case, since that operator has not been overloaded. As an extra exercise, you can implement this operator function.

objects

When you examined the example above, you noticed that the structure of the program is quite complicated. It makes us wonder if we really did gain something from the multitude of objects. Would it not have been better if the gameboard had simply included the data on how much water the drops had and where the splashes went and the gameboard object had controlled everything?

Colliding splashes

An example of how it can be useful to have objects that are responsible for small things is the previous execution, where two splashes proceed to opposite directions and meet each other in one square. Looking at the class `Splash`, you might wonder how the meeting of two (or more) splashes could complicate the movement algorithm of a splash, but now you see it does not. The class `Splash` controls the movement of a single splash object. Even if two objects met in the same square, it would not make the functioning of one of them any more complicated. If we, instead, had a gameboard object controlling all the details about drops and splashes, this could affect the splash movement algorithm in a different way. This teaches us that the idea of object-oriented programming is to create small parts that are independently responsible for their own functioning.

Destructing objects

Earlier, we already talked about the gameboard object destructing the splash objects in the method `moveSplashes`. Destructing drop objects, however, is very different. If the drop method `addWater` exceeds the maximum size and it pops, the drop object will destruct itself after creating the splashes. It is done by the drop object calling the the gameboard method named `removeDrop`.

It is important that the self-destruction is the last thing happening in the method, because the functioning of an object is undefined after its memory has been deallocated.

This difference in *destructing* splash objects and *self-destructing* drop objects is a good depiction of what the **responsibility of a module** means. The drop object self-destructs independently. This makes it easier to implement the gameboard object. Instead, the method `moveSplashes` of the gameboard object does many actions we could just as well dedicate to the splash object. If the implementation was different, we could have the splash object add the water to the drop object it hits, and then destruct itself. Since the gameboard object performs some of the actions of the splashes, it has grown larger than it perhaps should. On the other hand, if we gave more responsibility to the splash object, it would need

access to the gameboard, and that would make the data structures of the program even more complicated. There are several viewpoints to each of the different solutions. A single right solution does not exist. As a programmer, you must be able to weigh the pros and cons of different solutions.

Summary

While being far from perfect, the implementation of this program offers you a great opportunity for philosophical thought about the good and the bad of the module division in this implementation. Did we make wise decisions when choosing which functionality is the responsibility of which module? Does each module respond the things it is responsible for? If we changed the division of responsibilities, would it make the program more or less complicated?

These issues will be discussed further on the programming courses following this one (e.g. Programming 3: Techniques and Software Design). However, if you are interested in this topic, it is worth thinking about it when you create other projects of your own.

More about programming style

At 4.6 in the course material, we discussed programming style rather extensively. However, at that point, we had not considered dynamic memory management at all, and thus, it was not possible to pay attention to programming style related to that topic, either.

Rules for programming style concerning dynamic memory management that are used on this course, are listed above:

- The object or module that have allocated memory, is principally responsible to deallocate the memory, too.
- If a `delete` command is targeted to an assignable pointer variable, it will be assigned to the value `nullptr` immediately after the `delete` command.
- A destructor must deallocate all the resources that have been allocated by the object in question.
- Unnecessary copy constructor and assignment operator must (on this course) be disabled by introducing them in the `public` part of the class and using the word `delete`, as explained in the course material at the end of round 8.
- If there is no good reason for using normal and smart pointers mixed, then do not do so.

The next round introduces graphical user interfaces. Memory management related to them are considered in the section about widgets and especially in the context of parent-child mechanism. This mechanism makes memory management simpler and decreases the need for `delete` command. That command is still needed if an object has been created with the `new`command and if the object has no parent.

Above, we call the constructor of the base class in the initializing list of the subclass. That call must be the first thing in the initializing list.

Inheritance in its basic form, shown here, is usable if there is a class that is almost what you need but lacks some operations in its public interface.

For example, we can imagine that we want a version of the C++ class `string` that has the operation `is_palindrome` in its public interface:

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

using namespace std;
```

```
class MyString: public string {
  public:
    MyString();
    MyString(const string& init_value);
    bool is_palindrome() const;
  private:
    // No private attributes needed in this example
};

MyString::MyString(): string("") {
}

MyString::MyString(const string& init_value): string(init_value) {
}

bool MyString::is_palindrome() const {
    string::size_type left = 0;
    while ( left < length() / 2 ) {
        if ( at(left) != at(length() - left - 1) ) {
            return false;
        }
        ++left;
    }

    return true;
}

int main() {
    MyString s1;
    MyString s2( string("abcba") );
    MyString s3(s2);

    cout << s1.is_palindrome() << endl;
    cout << s2.is_palindrome() << endl;
    cout << s3.is_palindrome() << endl;

    s1.append("ab");
    s2.append("z");
    s3.append("z");

    cout << s1.is_palindrome() << endl;
    cout << s2.is_palindrome() << endl;
    cout << s3.is_palindrome() << endl;

}
```

Unfortunately, it is not always that easy to do inheritance from complicated base classes, but the idea is well represented here.

Because all the objects of the subclass are also members of the base class, you can basically use them in all the same places as you use the base class objects. In practice, C++ is not that flexible. However, you can use a subclass object as a parameter to a function if the type of its formal parameter is a reference or a pointer to a base class object.

```
void process_vehicle(Vehicle& v) {
    v.report_travel(15.0);
}

int main() {
```

```
    Vehicle veh(20.0, "red");
    Car subaru(75.0, "blue-grey", "ABC-123");
    process_vehicle(veh);
    process_vehicle(subaru);
}
```

When you get down to it, this is also a mechanism for trying to achieve the benefits of dynamic typing in a language that uses static typing.

Notes on inheritance

Inheritance is an interesting mechanism, which means that you might want to use it just because it is new. This can easily lead to silly solutions. Use inheritance sparingly and only when you can say with reason that it is a good solution.

This is a good base rule:

Only inherit the subclass Y from the base class X if you can explain to yourself how each Y is also an X.

For example, it is reasonable to inherit a car from a vehicle because each car is also a vehicle. It is not, however, reasonable to inherit a car from an engine, because a car is not an engine. Here, we have a so-called **has-a** relationship: a car has an engine. Therefore, the way the implementation mechanism should go is that the `private` part of the class `Car` has a member variable of the type `Engine`.

The example above is almost laughably obvious. However, the fact is that an inexperienced class designer will spring this trap in any situation that is slightly more confusing, even if a closer look at the situation might reveal it to be exactly the same as an is-a vs. a has-a situation.

The examples on inheritance given in this section are only a superficial take on all the inheritance mechanisms offered by C++. Every one of you should now have a basic understanding of what inheritance means.

GUI application with Qt

Teksti: Essi Isohanni, Esko Pekkarinen and Ari Suntioinen

On the previous programming course we already created graphical user interfaces with the Tkinter library of Python. Just as Python, C++ also has several libraries for creating user interfaces. We selected the Qt library of C++ for implementing graphical user interfaces on this course. (Qt is a library, and Qt Creator is a programming environment where the mentioned library can be easily used.)

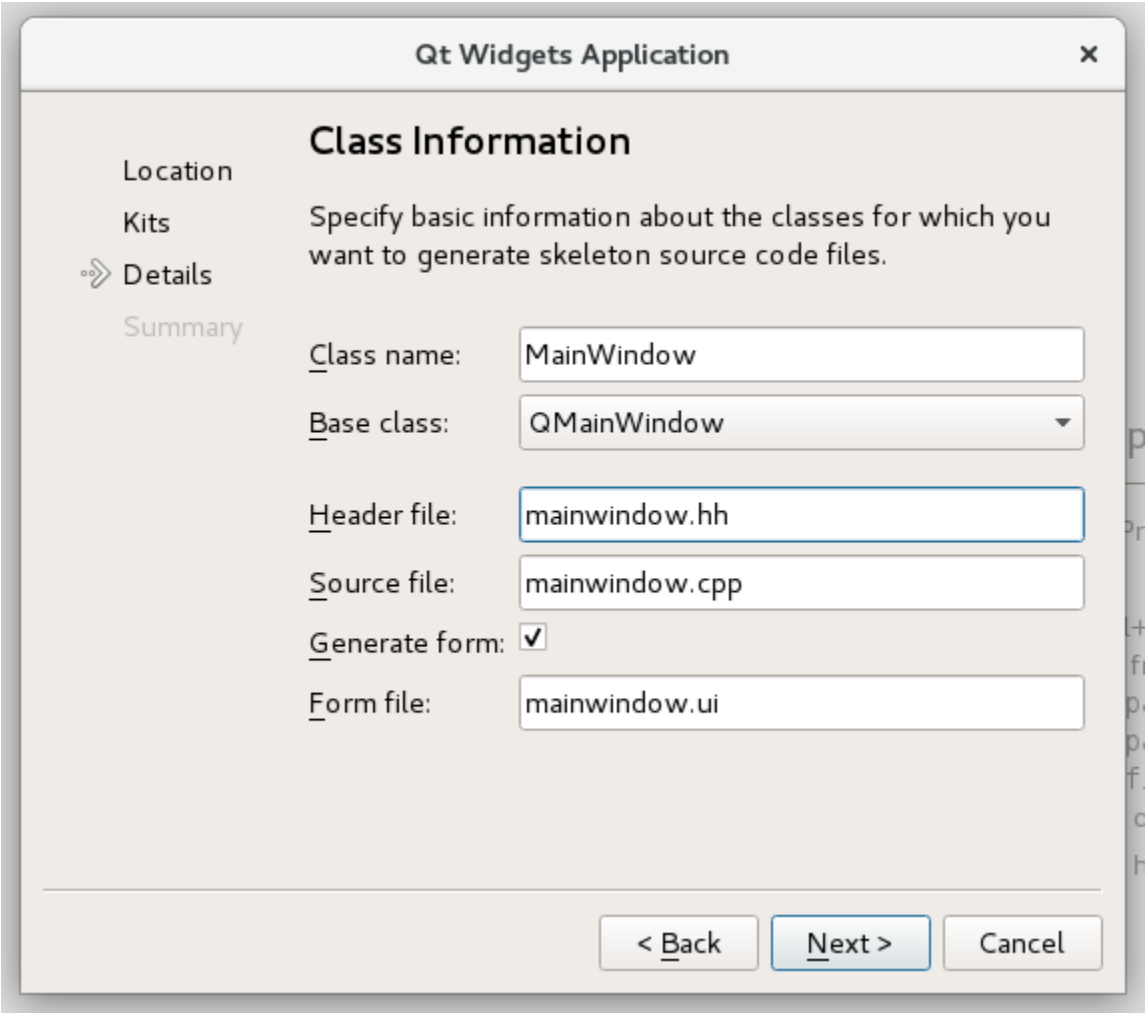
On the previous programming course we learned that all the components of a user interface, such as the user interface window itself, are objects. We also learned that the graphical user interfaces have a so-called event handler that follows, for example, the mouse clicks in a user interface. Qt follows the same principles. That is why we now just need to learn which objects are already implemented in Qt, and how to use them.

We will get to know Qt by way of you first creating a new project in Qt Creator. Qt Creator generates an empty user interface window, from which you can start. We will see the code inside that base. In the two next material sections, our example will be the template code of the exercise `templates/11/colorpicker_designer`, which includes some simple functionalities. (Remember to copy it into `student/11/colorpicker_designer`.) The most important mechanisms of Qt will be explained with the help of these two programs.

Creating a project with a graphical user interface

Create a new project in Qt Creator, and make it include a graphical user interface:

1. As usual, start Qt Creator in the menu at the top bar of the virtual desktop: `Applications > Programming > Qt Creator`.
2. Creating a new project has the usual first steps: `File > New File or Project...`, or choose the button New Project in the main window of Qt Creator.
3. From now on, creating the project is somewhat different to what we are used to. In the window New Project, you should this time select the project type like this: `Applications > Qt Widgets Application` (until now, we always selected `Non-Qt Project > Plain C++ Project`).
4. As usual, select your project file in the Introduction and Project Location window. You can name the program e.g. as `first_gui` (our purpose it just to create an empty user interface for this project).
5. Complete Kit Selection as usual, because aside from what is offered, no other developing environments are provided for virtual desktops. If you have installed Qt Creator on your own computer, you might have other options here. In that case, do as you have done earlier.
6. Now, the Class Information window that opens up is the biggest difference to the earlier projects. Complete your selections according to the following illustration:



You will see the filenames updating automatically when you choose a name for your user interface class at ”Class name.”

7. In the Project Management window, you should select the Add to Version Control field to have the value ”<None>”. After this, you can click the Finish button.

8. You have now created the project, and Qt Creator has automatically formed a project file, the bodies of source code files, and a file with the suffix `.ui` that describes the user interface formed in Qt Designer. In case you named your user interface class `MainWindow`, Qt Creator has created these files:

```
9. main.cpp
10. mainwindow.cpp
11. mainwindow.hh
12. mainwindow.ui
13. first_gui.pro
```

   More about these in a while.

14. Before starting to code, remember to add the following line in your `.pro` file:

```
15. CONFIG += c++14
```

   for avoiding problems later.

16. You can compile and execute the body of the user interface generated by Qt Creator normally, by clicking the run button (green triangle).

At this point, the program will not do anything particularly interesting, because the automatically generated code only includes the empty main window where the programmer can form the user interface they want.

Files created by Qt

Let us have a look at the files created by Qt Creator in order to get an idea about the structure of the program on a general level. We will proceed in this order: `main.cpp`, `mainwindow.hh`,`mainwindow.cpp`, and finally, `mainwindow.ui`.

`main.cpp`

The file includes automatically generated code that creates the object of the type `MainWindow` and launches the user interface.

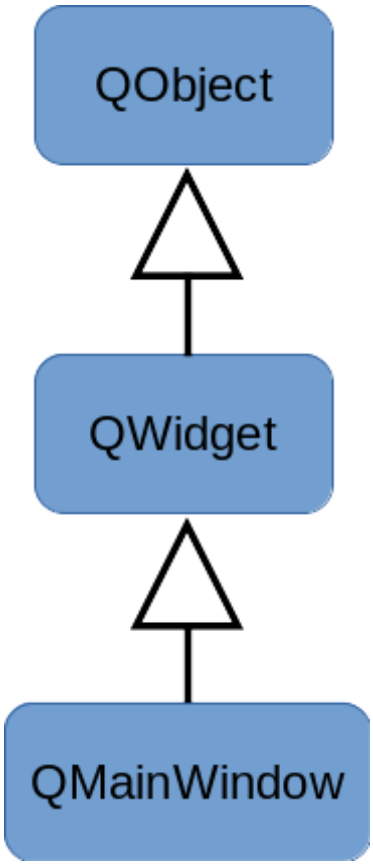
In programs that are as simple as the ones we are creating on this course, there is no need to edit `main.cpp`.

On the last line of the function, the call of `exec` targeted to the object `QApplication` is Qt's way to start an event loop (or main loop) that we talked about already when we learned about Tkinter on the previous programming course.

Otherwise, there is nothing particularly interesting in the main program module.

`mainwindow.hh`

All the user interface elements in Qt have been inherited from the class `QObject`, whose public interface provides some very-low-level tools for handling user interface elements (widgets) (e.g. the communication between different user interface elements with the signal-slot mechanism).



An ordinary programmer rarely ends up using the class `QObject` directly, and you can barely see it in the code produced by Qt Creator, excluding the line `Q_OBJECT` in `mainwindow.hh` which you do not have to care about now.

The class `QObject` is the base class for the class `QWidget`, which again works as a base class for all of the user interface elements of Qt (sliders, buttons, menus, et cetera).

When a programmer wants to create their own (simple) user interface, they implement a class of their own that has been inherited from the class `QWindow`, and add to it new features that must be included in the user interface element they are implementing. The class `QWindow` has also been inherited from `QWidget`. The file `mainwindow.hh`, created by Qt Creator, shows us how the main window of this exercise has been implemented as the class `MainWindow`, inherited from the class `QWindow`. The programmer can modify the automatically created class `MainWindow` by adding methods, member variables, and so on, to it.

`mainwindow.cpp`

This file does not (yet) include anything particularly exciting. You can find the definitions of the methods of the class that was defined in the file `mainwindow.hh`.

`mainwindow.ui`

The file contains the description of the user interface in so-called XML format. This file is not supposed to be edited manually. When a file is chosen from either the Projects list or from the project file menu, Qt Creator goes into Designer mode in which the user interface elements, the layout, and some of the functionalities can be designed interactively using the mouse. We will get to know this in the later section of the learning material.

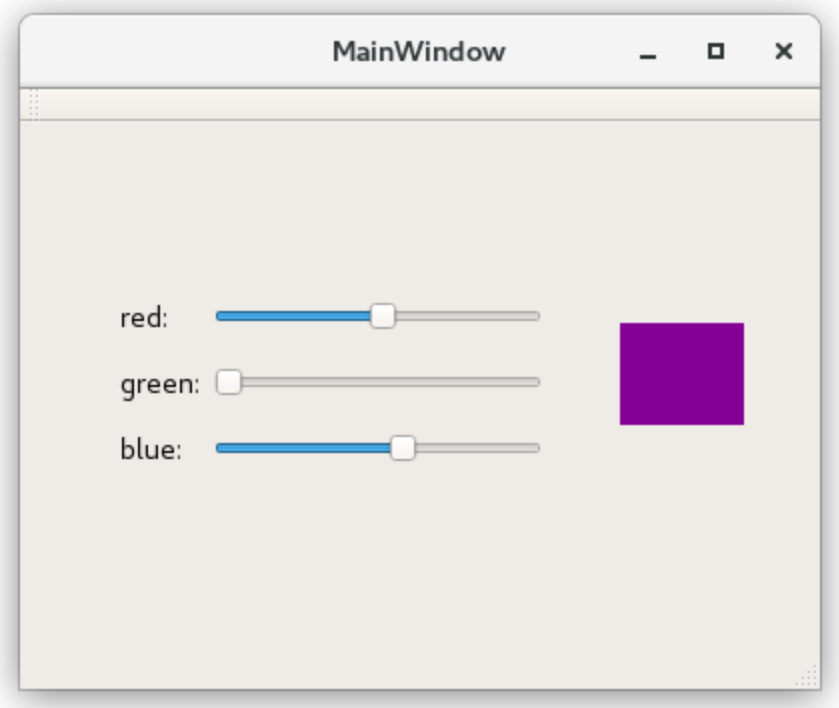
If you have already opened the file `mainwindow.ui` and moved to Qt Designer, you can return to the familiar view by clicking Edit mode on the dark left side menu. In the Edit mode, you can see what the XML description of a user interface looks like.

Signals and slots

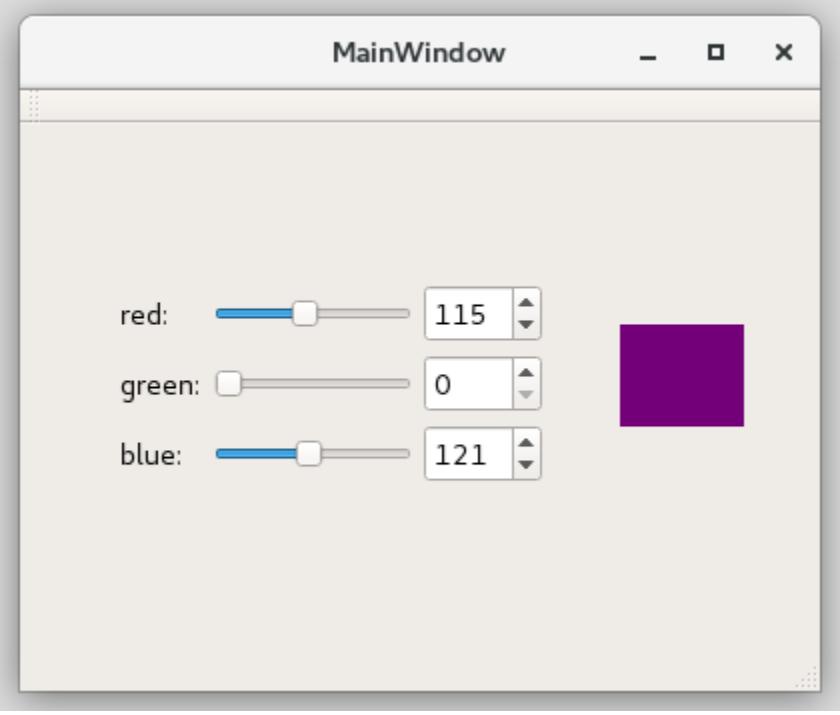
Teksti: Essi Isohanni, Esko Pekkarinen and Ari Suntioinen

On this round we will use Colorpicker program both as an example and as a template for an exercise. Here will study its user interface components and functionalities. Copy the project from `template/11/colorpicker_designer` to `student/11/colorpicker_designer`, and execute the program within it.

Below you can see the user interface of the colorpicker template



and that of the completed version after doing the exercise given in the next Plussa section.

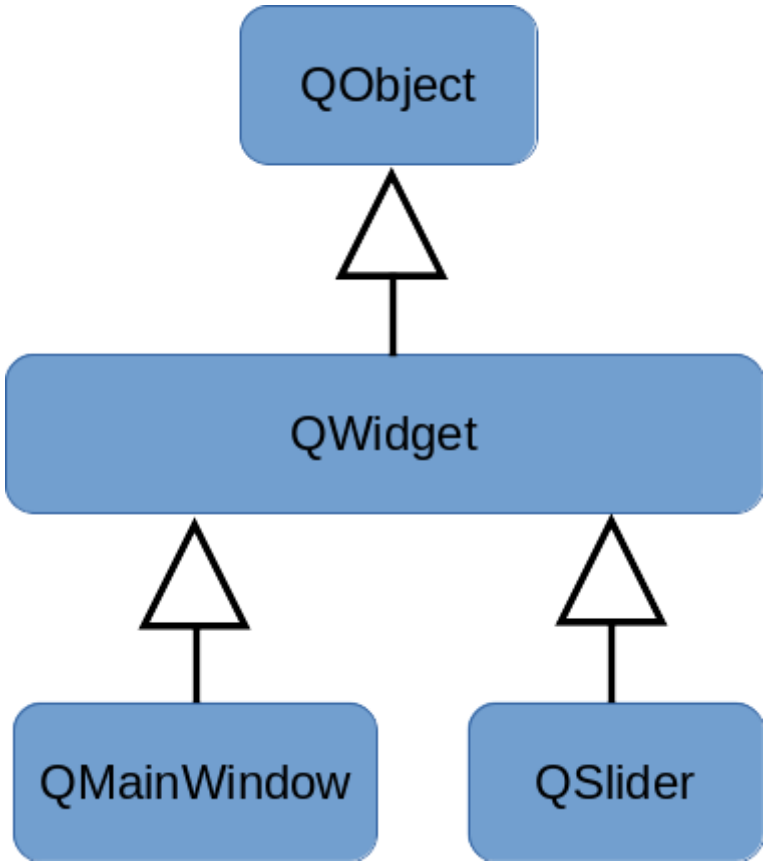


The final user interface of the Colorpicker program has three *sliders* and three *spin boxes* that you can use to choose the color components of a color's RGB value, meaning the amount of red, green, and blue colors. On the screen, there is also a *label* showing the chosen color, which is formed from the values selected in the user interface components at a certain time. To study the functionality of the program, change the values of the color components and see how it changes the color presented on the screen.

You can conclude that the color label is connected to the values of all sliders, because it changes color when the value of any of the components mentioned above changes.

Signal & slot mechanism

In Qt, all the components are implemented by inheriting them from the class `QWidget`, just like `QMainWindow`, mentioned in the previous material section.

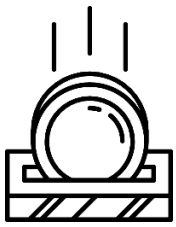


Joining user interface components to each other like described above is done with the so-called **signal-slot** mechanism. The simple way to convey this idea is that all the classes inherited from the class `QObject` can send signals and receive them using slots. Sending a signal is done with the keyword `emit`. For example, a slider could have this:

```
emit valueChanged(value_);
```

or:

```
emit valueChanged(42);
```

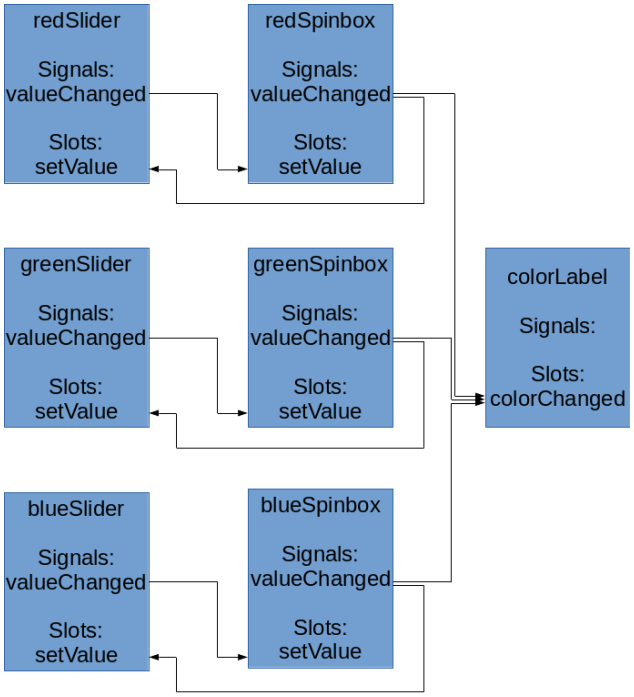


In the Finnish version of this text, we did not bother translating the word *slot*. You can think of it as a hole of a ”one-armed bandit”, or a slot machine, where inserting a coin activates a function (such as starting a fruit game or the music of a jukebox).

Looking at program code level, the signal looks like a function. If you wish, you can think of sending a signal as a function call. The difference between it and an ordinary function call is that the component sending the signal does not know who/what receives the signal. Also, there can be several receivers.

If you change the values of the sliders in the Colorpicker program, it causes emitting the signal `valueChanged`. The way Qt works is that sending a signal activates the slots connected to it. In the final Colorpicker version, the slider emits the signal `valueChanged`, and that activates the `setValue` slot of the ”corresponding” spin box, which in turn changes the value of the spin box. Similarly, editing the value of a spin box changes *both* the value of the ”corresponding” slider *and* the color of the color label.

These relationships can be represented as:



This material is a very simplified presentation of how Qt signals work. In case you wish to get deeper into the matter, we recommend you read the Qt documentation on signals and slots. The Qt documentation explains everything on a general level, whereas this course material uses a single program as the example.

Program code of Colorpicker

Let us examine the program code of Colorpicker. The file `main.cpp` is not different to the empty window we examined on the previous material section. Therefore, we will start with the file `MainWindow.hh`.

As with the previous section, the class `MainWindow` is inherited from `QMainWindow`, and the first thing written in its interface is `Q_OBJECT`. Also, the definitions of the constructor and destructor are in the public interface of the class.

In the section `private slots`, you can find those slots that are implemented in the main window just for the use of the class. This means that only the class itself can connect signals to them. If the class additionally had the section `public slots`, it would include signals that you can use from outside the class as well. The visibility of slot sections is similar to the class interfaces we learned earlier.

The class `MainWindow` has the implementation of the slot `onColorChanged` that defines how a window is supposed to work when the user edits the value of the color.

The `private` part of a class contains definitions of the member variables, just like with other classes. As we learned on the previous programming course, a user interface class can have user interface components as its member variables. You should have as member variables only those user interface components that you must be able to handle in the program code. This is why we here have no widgets. In this case, all widgets have been defined in Qt Designer that you will learn to use in the next exercise section. In program code, we can refer to them via the user interface (`ui`).

The class interface also includes the constant definition for the maximum value of RGB values.

Next, open the file `MainWindow.cpp`. The initializing list of the constructor initializes all the user interface components that are member variables, just as we are used to do with member variables. We will complete some settings (`setMinimum` and `setMaximum`) for the user interface components in the body of the constructor, because we cannot do that directly in their constructor.

The most important thing to do at the end of the constructor's body is connecting each changing signal of a horizontal slider to the slot `onColorChanged` of the `MainWindow` object. We already mentioned the purpose of `onColorChanged`. Your task is to connect the changing signals of the sliders and the spin boxes in pairs.

The final situation of signals and slots is shown in the previous figure. By comparing the code and the figure, try to find out which connections are already implemented and which connections are left for you to be completed.

At the end of the constructor, we call the slot `onColorChanged()` like we call any function, so that the color of the color label is changed to the initial value of the color selectors.

The implementation of the slot `onColorChanged` is short. It creates an object of the type `QColor`that can be shown in the object `QLabel`. The RGB value, obtained from the horizontal slider objects, is passed to the constructor of the object to be created.

At the end, it is good to stop and think about why the slot `onColorChanged` was implemented in the class `MainWindow`. However, it is a `colorLabel` object that changes color. The slot must be able to handle horizontal slider objects, which means it is easiest to implement it in the main window. The slider objects are children of the main window, which is why they can be handled in the main window. (Of course, it would be possible to pass slider objects or references to them as parameters outside of the class as well, but we will not consider that solution here.)

Final notes

In order to use Qt, the programmer must understand many of the characteristics of the C++ programming language, because all the user interface components are implemented as inherited classes, and the objects created from them are stored using pointers. Now we

understand why graphical user interfaces were not the starting topic of this course, even if it would have potentially been nice to study them right after the introduction to graphical user interfaces, studied at the end of the previous programming course.