This little text should give you a quick refresher about the most important aspects of C++. If you don't understand some part of the code or have trouble with the exercises, don't hesitate to ask your TA, he/she is glad to help you!

1 Basic example

The following small program demonstrates the most basic language features in C++: Printing and formatting output, looping, how to write a `main()` function and how to report errors during program execution.
To compile this code, use the provided Makefile by running:
`$ make`

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

// in case you want to get arguments passed to your program
// (such as "./basic a b c"), use the following definition:
//     int main(int argc, char** argv) { ... }
int main()
{
    const int N = 9;
    // print the header
    std::cout << "Calculating the factorial for n = 0, ..., N:" << std::endl;
    std::cout << "n" << std::setw(12) << "n!" << std::endl;

    int product = 1;
    for(int n = 0; n <= N; ++n)
    {
        if (n != 0)
        {
            // calculate the next next factorial
            product *= n;
        }

        // output the factorial
        std::cout << n << std::setw(12) << product << std::endl;
    }

    // tell the caller program that no error has occurred
    return 0;
}
```

Listing 1: A basic C++ program.

```makefile
CC=g++
CFLAGS=-O3 -Wall -Wextra -Wpedantic

all: basic

```

```
basic.o: basic.cpp
	$(CC) -c -o basic.o basic.cpp $(CFLAGS)

basic: basic.o
	$(CC) -o basic basic.o

clean:
	rm -f *.o *~ basic

.PHONY: all
.PHONY: clean
```

Listing 2: A basic Makefile.

Exercise 1
Adapt the code so that the number N is not hardcoded, but read from command line.

2 Reusing code: Functions

The above code does a good job: It calculates the correct factorials for all integers up to and including N.
However, it would be nice to be a bit more flexible. For example, in most cases we just want to calculate 5!, but not 1!, 2!, 3! and 4!. What we need now is a *function*. A function is just a piece of code that performs one task. It can be used (*called*) by other parts of code. The above code can be adapted to use functions:

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

// POST: returns n!
unsigned int factorial(const unsigned int n)
{
    unsigned int fac = 1;

    for (int i = 2; i <= n; ++i)
    {
        fac *= i;
    }

    return fac;
}

int main()
{
    int N;
    std::cout << "Please tell me the number N: " << std::flush;
    std::cin >> N;
    std::cout << std::endl;
    // print the header
    std::cout << "Calculating the factorial for n = 0, ..., N:" << std::endl;
    std::cout << "n" << std::setw(12) << "n!" << std::endl;

    for(int n = 0; n <= N; ++n)
    {
        // output the factorial
        std::cout << n << std::setw(12) << factorial(n) << std::endl;
    }

    // indicate that no error has occurred
    return 0;
}
```

Listing 3: The basic example, this time using functions.

How nice! Now we can use factorials wherever we want without having to copy the code that actually performs the calculation.

Writing all function definitions into one file is a bad idea because this leads to very ugly code that is difficult to read and understand. In C++, to separate the code into multiple `.cpp` files, it is not enough only to move the function codes, as each C++ file is compiled separately, and it would be impossible to call one function from another file. Instead, for each `.cpp` file, a `.hpp` *header file* is created, listing all function declarations (prototypes). Remember to put the include guards in the header files in order to prevent duplicate declarations of functions: when your code includes libraries A, B, and library A includes library B, the content of library B would be declared twice! An example of a simple header file looks as follows:

```cpp
#ifndef FACTORIAL_HPP
#define FACTORIAL_HPP

// POST: returns n!
unsigned int factorial(const unsigned int n);

#endif // FACTORIAL_HPP
```

Listing 4: The header file for our factorial function.

The implementation can be found in the file `factorial.cpp`. Finally, the header file can be included in our program as any other library would be, with one exception: Our own library is included with quotation marks (") instead of lesser-greater signs (<>). This is because the library cannot be found where the system libraries are.

3 Templates

Our `factorial` function is now a working piece of code, nicely separated from the rest of the program. Let's assume that we need to calculate the factorial of big numbers, say 100. The result of this is of the order of magnitude of 10^{157}, which is not representable by an `unsigned int` (which is represented by 4 bytes on common hardware)! We would need a big-number data type, let's call it `bigint` to calculate this result![1]

[1]It is a really inefficient idea to calculate the factorial of big numbers using our algorithm. A better approach would be to use the Γ function.

Do we have to rewrite our code to use the new type `bigint`? Perhaps there are people that just need to calculate factorials of ordinary `unsigned int` variables. They don't want to buy and install the expensive `BigNumbers`^{TM} library. Therefore, we cannot just change our code.
Another idea would be to just write and maintain two functions that are identical, except that different types are used. But we don't want to copy code every time a change is made to one of the two functions!
The solution to this dilemma is straightforward: We let the compiler do the copying when needed. This can be done by using *templates*.
Example:
We stick to the factorial example. We do not have to change our `main.cpp` file! However, we cannot compile our library separately anymore because the compiler needs to know the used data type at compile time. Therefore, the implementation of a template function must be given in the header file, which is shown in the listing below:

```cpp
#ifndef FACTORIAL_HPP
#define FACTORIAL_HPP

// POST: returns n!
template <typename T>
T factorial(const T n)
{
    T fac = 1;

    for (T i = 2; i <= n; ++i)
    {
        fac *= i;
    }

    return fac;
}

#endif // FACTORIAL_HPP
```

Listing 5: Template version of the factorial function.

Template metaprogramming is one of the most powerful features of C++. If used in a clever way, it is possible to get zero-cost abstractions: Classes and functions that provide high-level functionality while performing only the necessary operations at runtime. The most prominent library that makes use of template metaprogramming is the C++ STL (Standard Template Library) itself. Another good example in the field of linear algebra is Eigen.
The foundation of template metaprogramming, also called *generic programming*, are concepts: A concept is a collection of properties a type must provide so that it can be used with in a generic function. A simple example for a concept would be:

"Provide a method with signature `bool is_integer()`."

In C++20, it will hopefully be possible to represent concepts in the source code. But until then, the programmer has to program by contract. This means that concepts have to be declared in a PRE statement.
The STL also heavily utilizes the *iterable* concept, which allows to combine many container types with many algorithms (see STL `algorithm`). One can also define containers of one's own.

Exercise 2

What concepts needs a type `T` from our factorial code to fulfill? (Which operations must it provide?)

4 Call-by-value, call-by-reference and pointers

There are many possibilities to pass values to a function. Make yourself familiar again by reading the following use cases:

1. **Call-by-value:**
   This is the most basic option. It means the function receives a copy of the argument. There are two cases in which call-by-value should be used:

   - First, use it for builtin data types like `int`, `double`, `char` or other small arguments. For those, passing a value by reference needs about as much copying as passing the value directly, so it doesn't matter[2].
   - Second, if you need a copy anyway.

   In all other cases, copying the parameters unnecessarily is bad for performance: The copying of data costs time, but isn't necessary if the function assigns something new to the parameter.

   It is not possible to modify a passed-by-value variable so that the change takes effect outside of the function:

```
void increment(int a)
{
    ++a;
}

int main()
{
    int x = 0;
    increment(x);
    // x is still 0

    return 0;
}
```

2. **Pointers:**
   Pointers are the C-style way of passing data without copying it. Instead of sending the value, only its memory location (address) is sent. The pointer is sent to the function, which then *dereferences* it to access the underlying value:

```
void increment(int* a)
{
    ++(*a);
}

```

[2]For example, a reference needs 8 bytes of memory (it's the memory location of the data). If your data is at the order of 8 bytes or less, why bother sending the address instead of the value itself.

```
int main()
{
    int x = 0;
    int* p = &x;
    increment(p);
    // x is now 1

    // alternative formulation without helper variable:
    int y = 0;
    increment(&y);
    // y is now 1

    return 0;
}
```

3. **Call-by-reference**
   Pointers are inherited from C, but they are a bit cumbersome to handle: One always has to use the dereference operator. References are a good way to get rid of this disadvantage without having to copy data[3]. The above code looks as follows when using references:

```
void increment(int& a)
{
    ++a;
}

int main()
{
    int x = 0;
    increment(p);
    // x is now 1

    return 0;
}
```

   This looks quite a bit more readable.

Exercise 3

Which one(s) of the following function definitions is/are not valid, and why?

1.
```
int add(int a, int b)
{
    return a + b;
}
```

2.
```
int add(int& a, int& b)
{
```

[3] Although it looks like a syntax simplification, it was originally introduced to allow operator overloading, i.e. defining custom +, - and other operators for custom data types.

```
    return a + b;
}
```

3.
   ```
   int& add(int& a, int& b)
   {
       return a + b;
   }
   ```

4.
   ```
   int* add(int* a, int* b)
   {
       return a + b;
   }
   ```

5.
   ```
   int* add(int* a, int* b)
   {
       return (*a) + (*b);
   }
   ```

6.
   ```
   int* add(int* a, int* b)
   {
       return &(*a) + (*b);
   }
   ```

7.
   ```
   int* add(int* a, int* b)
   {
       return &((*a) + (*b));
   }
   ```

Exercise 4

Among all the valid function calls from the last exercise, which one(s) correspond(s) to the usual semantics one would expect for the addition of two numbers?

5 Classes and objects

So far, we have seen how to write functions, generalise them using template metaprogramming and how to pass values to them. Sometimes, it is desired to perform many operations on the same data. By using only what was discussed until now, it would be necessary to pass this data to all functions. This is unnecessarily verbose. Classes and structs provide a better and more natural way to work on the same data.
Classes describe a collection of variables (*member variables* and *static variables*) and functions (*member functions* and static functions). Objects are instances of classes: Each object of class

A has its own set of member variables and member functions can be called on the object itself. Static variables, however, are associated with the class A and shared among all objects of type A. The same is true for static functions: They can only access static or global variables, but not member variables because they're not associated with an object, but with the class.
The following listing shows how to define a class:

```cpp
#ifndef SIMPLECLASS_HPP
#define SIMPLECLASS_HPP

class SimpleClass
{
public:
    using simple_type = int;
    using counter_type = unsigned int;

    // constructors
    SimpleClass(simple_type value);
    SimpleClass();

    // destructor
    ~SimpleClass();

    // member functions
    void set_member(simple_type value);
    simple_type get_member();

    // static functions
    static counter_type get_object_counter();

private:
    // a member variable
    simple_type x_;

    // a static variable
    static counter_type obj_counter_;
};  // don't forget the semicolon!

#endif // SIMPLECLASS_HPP
```

The implementation of this class is given in a separate file:

```cpp
#include "simpleclass.hpp"

// static member variables should be initialised in the implementation file:
SimpleClass::counter_type SimpleClass::obj_counter_ = 0;

SimpleClass::SimpleClass(simple_type value) : x_(value)
{
    // increase the object counter to keep track of how many objects we have created so far
    ++obj_counter_;
}
```

```cpp

// don't copy paste code, but call the constructor we have already implemented!
SimpleClass::SimpleClass() : SimpleClass(0) {}

SimpleClass::~SimpleClass()
{
    // clean up actions should be here:
    // most importantly, free all dynamically allocated memory, unregister MPI types
    // (see later in this course or next semester for details about MPI)

    // in this example, decrease the object counter when an object is destroyed
    --obj_counter_;
}

void SimpleClass::set_member(simple_type value)
{
    x_ = value;
}

SimpleClass::simple_type SimpleClass::get_member()
{
    return x_;
}

SimpleClass::counter_type SimpleClass::get_object_counter()
{
    return obj_counter_;
}
```

Finally, one also wants to use a class. In our example, this would look as follows:

```cpp
#include <iostream>
#include "simpleclass.hpp"

int main()
{
    std::cout << "No. objects: " << SimpleClass::get_object_counter() << std::endl; // 0 objects
    SimpleClass obj;
    std::cout << "No. objects: " << SimpleClass::get_object_counter() << std::endl; // 1 object

    // create a new scope to demonstrate the destructor:
    {
        SimpleClass tmp_obj;
        std::cout << "No. objects: " << SimpleClass::get_object_counter() << std::endl; // 2 obje
    }   // tmp_obj is destroyed when it is out of scope

    std::cout << "No. objects: " << SimpleClass::get_object_counter() << std::endl; // 1 object

    // how to access member functions:
    std::cout << "Member of obj: " << obj.get_member() << std::endl;    // 0
    obj.set_member(5);
    std::cout << "Member of obj: " << obj.get_member() << std::endl;    // 5

    return 0;
}
```

6 Inheritance

Often, classes can be organised in a hierarchical way: One starts with the most general example (e. g. a class `Animal`). Then, one can create *child classes* (`Fish`, for example) that have all the properties of the *parent class* `Animal`, plus some more properties. This pattern is known as *inheritance*.

Exercise 5

(Optional) Fill out the `//TODO` parts in the provided calculator skeleton files!

Solutions

Exercise 1

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

int main()
{
    int N;
    std::cout << "Please tell me the number N: " << std::flush;
    std::cin >> N;
    std::cout << std::endl;
    // print the header
    std::cout << "Calculating the factorial for n = 0, ..., N:" << std::endl;
    std::cout << "n" << std::setw(12) << "n!" << std::endl;

    int product = 1;
    for(int n = 0; n <= N; ++n)
    {
        if (n != 0)
        {
            // calculate the next next factorial
            product *= n;
        }

        // output the factorial
        std::cout << n << std::setw(12) << product << std::endl;
    }

    // indicate that no error has occurred
    return 0;
}
```

Exercise 2

`T` must be copy-constructible in order to allow the initializations in lines 8 and 10. Further, it must support the pre-increment operator (`++i`) as well as the multiplication operator (needed in line 12).
Finally, for large data types `T`, it would be good for performance if it provides a move constructor (because this prevents unnecessary copying of data), but this isn't mandatory.

Exercise 3

The functions in numbers 1 and 2 are valid, there is no problem.
Number 3 is not valid: The result of the addition is stored in a temporary variable that is only guaranteed to exists in the scope of the function, but not outside. Therefore, a reference to a non-existing location in memory would be returned. The compiler recognizes this and the code will not compile.

From a technical point of view, a pointer is an address of some data in memory. Therefore, pointers are `int`s. They can be used in all kinds of arithmetical operations, also for additions. For these reasons and because a dereference pointer `*a` to an int is an `int` as well, the functions in numbers 4-7 are valid. However, one has to be careful: Even though the function definitions are valid, the pointers must not be dereferenced! They do not point to valid regions in memory, in general.

Exercise 4

Function definitions 1 and 2 are not only valid, but also fulfill the correct semantics for addition. Number 3 does not have to be discussed because the code is invalid.
Number 4 can be considered correct as well, if the desired behaviour is to add up addresses, for whatever reason that might be. A better signature, however, would then be to return an `int` instead of an `int*` in order to prevent dereferencing the invalid pointer.
Function definitions 5-7 do not make much sense.
Note that "semantically correct" depends on the definition of the correct semantics. This definition has not been given in the exercise text, so other solutions might be possible. The solution given above tries to keep semantics as simple as possible.

Exercise 5

See solution code:

```
#ifndef CALCULATOR_HPP
#define CALCULATOR_HPP
#include <vector>
#include <cmath>

template <typename T>
class Function
{
public:
    virtual ~Function(){};

    // calculates the value of the function represented by this class at the point x
    virtual T operator()(const T& x) = 0;
    // calculates the value of the function represented by this class at the point x (assuming th
    virtual T derivative(const T& x) = 0;
};

// an example child class
template <typename T>
class ConstantFunction : public Function<T>
{
public:
    ConstantFunction(const T& c) : c_(c) {}

    ConstantFunction() : ConstantFunction(0) {}

    // comment out the parameter to silent -Wunused-parameter warning
    T operator()(const T& /* x */)
    {
        return c_;
    }

    T derivative(const T& /* x */)
    {
        return 0;
    }

private:
    T c_;
};

template <typename T, int D>
class Polynomial : public Function<T>
{
public:
    // PRE: coeffs must be of size D+1 and contain the coefficients:
    // coeffs[0] + coeffs[1]*x + coeffs[2]*x^2 + ... + coeffs[D]*x^D
    Polynomial(const std::vector<T>& coeffs) : coeffs_(coeffs) {};

    T operator()(const T& x)
    {
        // An alternative way to evaluate polynomials is Horner's method
        // (without pow() function calls).
        T polyval = 0;
        for(int i = 0; i <= D; ++i)
        {
            polyval += coeffs_[i] * std::pow(x, i);
        }

        return polyval;
    }

    T derivative(const T& x)
    {
        // An alternative way to evaluate polynomials is Horner's method
        // (without pow() function calls).
        T deriv_val = 0;
        for(int i = 1; i <= D; ++i)
        {
            deriv_val += i * coeffs_[i] * std::pow(x, i-1);
        }

        return deriv_val;
    }

private:
    const std::vector<T> coeffs_;
};
#endif // CALCULATOR_HPP
```