**Lambdas** (certain kind of unnamed functions) are a mechanism for implementing functionality for temporary use inside program code. They are related to algorithms discussed in Section 2.

**Exceptions** are a way of handling unexpected situations, such as errors.

**Namespaces** allow hierarchy in naming types and objects, and are particularly useful in larger software.

# 1. Lambda expressions

Lambda is an expression that defines a nameless callable function, with some additional properties that are not possible with the traditional functions. Lambda expressions were introduced in C++11, and they are useful with algorithms, particularly, if the specified behavior is not needed in more than one or two places in the code.

Lambda expression includes a **capture list** in brackets ([], in this case empty), **parameter list** (as in functions), and the body of the lambda function. Like normal functions, also a lambda function can have a return value.

sort(starks.begin(), starks.end(),

[](const std::string& a, const std::string& b) -> bool

{ return a.size() < b.size(); });

 in straight-forward cases the compiler will be able to determine the correct return value type (as with the auto type).

auto it = find\_if(starks.begin(), starks.end(),

[len](const std::string& a)

{ return len <= a.size(); });

The find\_if() algorithm will return iterator to the first element in a container that meets the given criteria. Here we want to return an element that is at least 'len' characters long. The capture list provides a way to transfer variables from outside scope to within the lambda function, as done here for variable 'len'. We could not use an additional function argument for this, because then the function interface would not be compatible with what find\_if() expects (an unary function with one argument). Also multiple variables can be included in capture list, as separated by comma.

## 1.1 Different types of captures

Like function arguments, the capture variables in lambdas can work based on value or by reference. By default the variable works by value, i.e., any modifications on the variable done inside the lambda function do not affect the outside function. The variable can also be defined to work by reference: [&variable] { code };, in which case the changes to variable inside the lambda function will affect also outside the lambda.

Also implicit capture lists can be defined. In this case the variables are not explicitly defined in code, but the compiler determines which variables need to be imported inside the lambda function. Format such as [=] { code };tells that "code" will use the needed outside variables following *by value* semantic. [&] { code }; is an implicit capture list using *by reference* semantic.

# 2. Exception handling

Exceptions are a run-time failure handling mechanism that is quite different to anything that C has (but other languages have similar mechanisms). When a function **throws** an exception for some reason (e.g., failure to allocate memory or open file), the **function exits immediately**.

Same happens to the next-level function that had called the function throwing an exception, and for the whole call stack until somewhere the exception is properly caught and handled. This is called **stack unwinding**. If there is no proper handler for the exception up the call stack, the program will eventually terminate because of the exception. Typically this is not the desired behavior, but the exception should be properly handled by the program.

Because the exceptions significantly affect the program control by prematurely terminating functions, there are challenges with resource management. Particularly, we should ensure that dynamically allocated memory is properly released also when exception occurs. On exception, the destructors of the known objects are run automatically, but if we have allocated memory using the raw mechanisms, the memory will not be released. Smart pointers are automatically released, which gives another good reason to use them.

When exception is thrown, it is provided with an **Exception object** that tells more information about the reason of the exception. The occurrence of exceptions is tested with a **try/catch** blocks that are intended to cover a critical part where we expect an exception is possible. The **exception** base class is defined in **exception** header.

 While not technically mandatory, we derive the standard exception class, that has initially defined the what() function as virtual, i.e., its implementation can be overridden by subclasses, as we do here. It would be possible to add additional members and function to the exception class, for example to deliver additional information about the error. We could also have parameters in the constructor, for example to present a and b, and have an additional function to query them when catching the expression.

When throw expression is used, the function terminates immediately. This line constructs a new exception object.

The main() function calls the divide function inside a **try** block. The parameters in call on line 34 will cause the exception to be raised. This causes the try block to terminate, and the execution to jump to the **catch** block. the catch expression defines which kinds of exceptions we want to handle. In this case we just give std::exception base class, which matches various built-in exceptions, and the our own which we derived from it.

The stack is unwound to the point where the appropriate exception is caught. If there is no catch block for matching type of exception, the program execution is terminated entirely.

# 3. Namespaces

Unlike C, that had a single global namespace, C++ allows defining namespaces that can be nested. This is helpful in large programs, to avoid name collisions between two parts of code that often may be developed by different independent parties. Namespace can contain any kind of declaration: classes, variables, templates, other namespaces and so on.

Namespace should be used, for example, for a library that can be used by multiple different programs. We have extensively used one such namespace already, **std**, which stands for the C++ standard library. Developing a that uses global names should be avoided.

Because any program or software library typically consists of multiple files, it is possible to specify a namespace block for the same name in multiple discontinuous locations and multiple files. Even though the type name for GeomVector block is under MyMath namespace, the variables are local to the function and in local naming scope.

he global namespace is implicitly declared for names defined in global scope. If one wants to explicitly refer to such global names, the notation ::one\_name refers to one\_name under global scope.

## 3.1 Using declaration

A lazy writer can use the using declaration to bring a name from other namespace to the current namespace. using std::string; would allow to use just string in further places where the string type from C++ standard library is used.

It is also possible to import entire namespace, for example by using namespace std; for the standard library. Although this may save in typing effort, it is rarely a good idea, because it opens the possibility of name collisions between the standard namespace and the current name space.

A virtual function whose declaration ends with =0 is called a pure virtual function.

It is important to note that class becomes abstract class when it contains pure virtual destructor.

to make a class abstract, at least one of its methods must be made pure virtual, which means the derived classes must override it. declare the destructor pure virtual. This will make your class abstract without forcing you to declare any other method pure virtual.