Course
ELEC-A7151
Course materials
Your points
Code
Code Vault
Course Pages
MyCourses
Teams Channel

This course has already ended.
The latest instance of the course can be found at: Object oriented programming with C++: 2023 Autumn

« 1 Introduction Course materials 3 Lambda expressions »
ELEC-A7151 / Module 5: Advanced Topics / 2 Exception handling

2 Exception handling¶

- **You can download the template for the programming tasks of the module as a `zip` file from `this link`.**

Contents

C++ Primer Chapter 18.1 (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 function that had called the function throwing an exception, and to the function that called that function, and so on, the exception is properly caught and handled somewhere or, if that doesn't happen, until the whole call stack is exhausted (i.e. the program crashes). This is called **stack unwinding**. If there is no proper handler for the exception at any point in the call stack, the program will eventually terminate because it has exited every function (including main). Typically this is not the desired behavior, and 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 an exception, the destructors of the stack allocated objects are run automatically, but, if memory or other resources have been allocated on the heap (i.e. using the *new* keyword or *malloc*), they have to be freed manually or a memory leak occurs.

When an 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** block that is intended to cover a critical part where we expect exceptions to be possible. Here is an example of a simple division function that covers different parts related to exception handling. In reality, the definitions would really be defined in header file, and the implementations in .cpp file. The **exception** base class is defined in **exception** header.

```
1/** Header file **/
2
3#include <iostream>
4#include <exception>
5
6// Define new exception class that derives from standard library
7// exception class.
8class DivByZeroException : public std::exception {
9   // the what function is virtual function defined in the
10   // exception class. It returns a string description of the exception.
11   // Here we define our own implementation that overrides the
12   // base class implementation.
13   virtual const char* what() const noexcept {
14       return "Division by zero";
15   }
16};
17
18
19/** Source file **/
20
21// function that calculates a / b
22// if b == 0, throw our newly implemented exception
23double divide(double a, double b) {
24   if (b == 0)
25       throw DivByZeroException(); // function terminates here
26
27   return a / b;
28}
29
30
31int main() {
32   double a;
33
34   // start a block where we test for exceptions
35   // catch expression considers all exceptions, not only DivByZero
36   // (however, we are not expecting any other exception to occur)
37   try {
38       a = divide(5, 0);
39       // here we jump to the catch block, the below is not printed
40       std::cout << "result is " << a << std::endl;
41   } catch (std::exception &e) {
42       std::cout << "Error: " << e.what() << std::endl;
43   }
44}
```

We start by defining our new **exception class** for division by zero exceptions, and implement the *what* function which returns a string about the reason for the exception. 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.

The *divide* function checks if the caller is trying to divide by zero, and in such case the function will throw an exception object using the *DivByZeroException* class we created. When the **throw** expression is called, the function terminates immediately. Together with *throw*, the code creates a new *DivByZeroException* object that will be provided to the calling function. We could also have parameters in the constructor, for example to pass *a* and *b*, and have an additional function to query them when catching the expression.

The *main* function calls the *divide* function inside a **try** block. Because there will be a division by zero, our exception will be be raised. This causes the try block to immediately terminate, and the execution will jump to the **catch** block. The argument in the catch expression defines which kind of exception we want to handle. In this case we just give the *std::exception* base class, which matches all exception of type *std::exception* and its subclasses, these include various built-in exceptions, along with our own *DivByZeroException*.

There can be multiple specific catch blocks following a try block, when we want to handle different kinds of exceptions differently. In this example only one kind of exception is thrown, but if our divide function had been more complicated, for example calling other functions that throw different exceptions, we may end up with different alternatives for exceptions. The stack is unwound to the point where the appropriate exception is caught, and the program execution moves there. If there is no catch block for matching type of exception, the program execution is terminated entirely.

Here is an example of multiple catch blocks:

```
try {
    /* code */
} catch(first_type e) {
    /* do what ever you need to for
     exceptions of type 'first_type' */
} catch(second_type e) {
    /* do what ever you need to for
     exceptions of type 'second_type' */
}
```

2.1 Programming tasks¶

Points 30 / 30 My submissions 9 Deadline Friday, 22 October 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

StrPrinter

Objective: Operator overloading, resource management, inheritance and virtual interfaces, exceptions, etc.

StringPrinter is an abstract base class for printing strings in different styles. There are two different classes that implement the *StringPrinter* interface: **DiagonalPrinter**, which outputs text in a diagonal format, and **StandardPrinter**, which just prints the text normally.

Printing is implemented overloading the *function call operator* (`operator()`). See this link for examples how it is used. Again, see the detailed descriptions in the header files.

In addition, there is **Printers** container for different kind of printers, which you need to implement.

diagonal_printer.cpp
Choose File No file chosen

diagonal_printer.hpp
Choose File No file chosen

printers.cpp
Choose File No file chosen

printers.hpp
Choose File No file chosen

standard_printer.cpp
Choose File No file chosen

standard_printer.hpp
Choose File No file chosen

string_printer.hpp
Choose File No file chosen

Submit

Points 25 / 25 My submissions 42 Deadline Friday, 22 October 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Custom exception

Objective: learn about exception handling and namespaces.

In this exercise we are going to extend the RestrictedPtr from last round. Here, you need to use your own implementation from the last round.

In the *restricted_ptr.hpp* you should implement your RestrictedPtr with exceptions and an use. This includes expanding the original implementation so that:

- it throws an **RestrictedCopyException** if you try to copy the RestrictedPtr over 3 times.
- it throws an **RestrictedNullException** if you try to call *GetData* method with a RestrictedPtr that points to a *nullptr*.

These 2 error classes inherit from the base exception class **RestrictedPtrException**. *RestrictedPtrException* inherits from *std::exception* (See image). For more details, see the header files.

You need to implement these 3 classes to the *restricted_ptr_ex.hpp* header file .

On top of this, the whole implementation in the *restricted_ptr_ex.hpp* and *restricted_ptr.hpp* files should be wrapped in a *WeirdMemoryAllocator* namespace. After this, the written namespace calls to those functions in *main.cpp* should work.

You also need to implement a wrapper/interface for this pointer class in *restricted_interface.hpp*. For more information, see the header file.

Exceptions class hierarchy, on the red you can see the exception classes created in this task1

restricted_interface.hpp
Choose File No file chosen

restricted_ptr_ex.hpp
Choose File No file chosen

restricted_ptr.hpp
Choose File No file chosen

restricted_ref_counter.hpp
Choose File No file chosen

Submit

Points 15 / 15 My submissions 7 Deadline Friday, 22 October 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

Triple

Objective: Practice creating a template class from scratch

Lastly, we have a template exercise relating to the last module.

In this exercise your task is to implement a template class from scratch. As some might have already noticed templates can be very useful when writing container-like classes. Implementing the containers with templates ensures that the container is type independent, i.e. supports different types of objects.

Your task is to implement a template container class, Triple. Triple is much like a pair, but with three values instead of two. A triple can either be homogenous or heterogenous, meaning that it can either have three values, all of which are the same type, or it can contain different types.

The Triple class that you have to implement has been declared in more detail in the *triple.hpp* file.

Note

Template implementations such as this usually fail in the compile stage. Test your implementation carefully. Certain things can only be initialised in an initialiser list.

Hint

C++ Primer Chapter 16.5 (Template specializations)

triple.hpp
Choose File No file chosen

Submit

« 1 Introduction Course materials 3 Lambda expressions »