Exception Handling: A Deeper Look

Chapter 17

Today we will discuss

- Exceptions: throwing and catching
- Exception class definition
- `unique_ptr`

Exceptions

C++ provides a mechanism to help deal with errors: *exceptions*.

Detection of an error and its handling are separated.

Exceptions

C++ provides a mechanism to help deal with errors: *exceptions*.

Detection of an error and its handling are separated.

Detection of an error: if a function finds an error that it cannot handle, it should not return normally. Instead, it `throws` an exception indicating what went wrong.

Handling: the `try`-block is used to `catch` the exception.

Exceptions

The header `<stdexcept>` defines a set of standard exceptions that both the library and programs can use to report common errors.

They are divided in two sets:

Exceptions

The header `<stdexcept>` defines a set of standard exceptions that both the library and programs can use to report common errors.

They are divided in two sets:

Logic errors

`logic_error`	logic error exception
`domain_error`	domain error exception
`invalid_argument`	invalid argument exception
`length_error`	length error exception
`out_of_range`	out-of-range exception

Exceptions

The header `<stdexcept>` defines a set of standard exceptions that both the library and programs can use to report common errors.

They are divided in two sets:

Logic errors

`logic_error`	logic error exception
`domain_error`	domain error exception
`invalid_argument`	invalid argument exception
`length_error`	length error exception
`out_of_range`	out-of-range exception

Runtime errors

`runtime_error`	runtime error exception
`range_error`	range error exception
`overflow_error`	overflow error exception
`underflow_error`	underflow error exception

Exceptions: standard streams

By default, *standard streams* (`iostream`) don't throw exceptions, but they have *stream error states* we covered in Section 13.8.

Boost.org provides a library that supports exceptions.

See examples in catchingAndThrowingExceptions.cpp

Defining and Exception Class

Let’s see how can we define an exception class:

- we can inherit from the existing exception classes, or
- we can avoid using the existing exception class

See these two examples:
definingExceptionClass.cpp
definingExceptionClass2.cpp

Re-throwing the exception

In some situations we might need to *re-throw the exceptions*.

For example: When working with a file, an exception occurred. Upon this, we want to close the file (by the handler) and notify the caller that there was an issue by re-throwing the exception.

Syntax:

```
throw;
```

Re-throwing the exception

In some situations we might need to *re-throw the exceptions*.

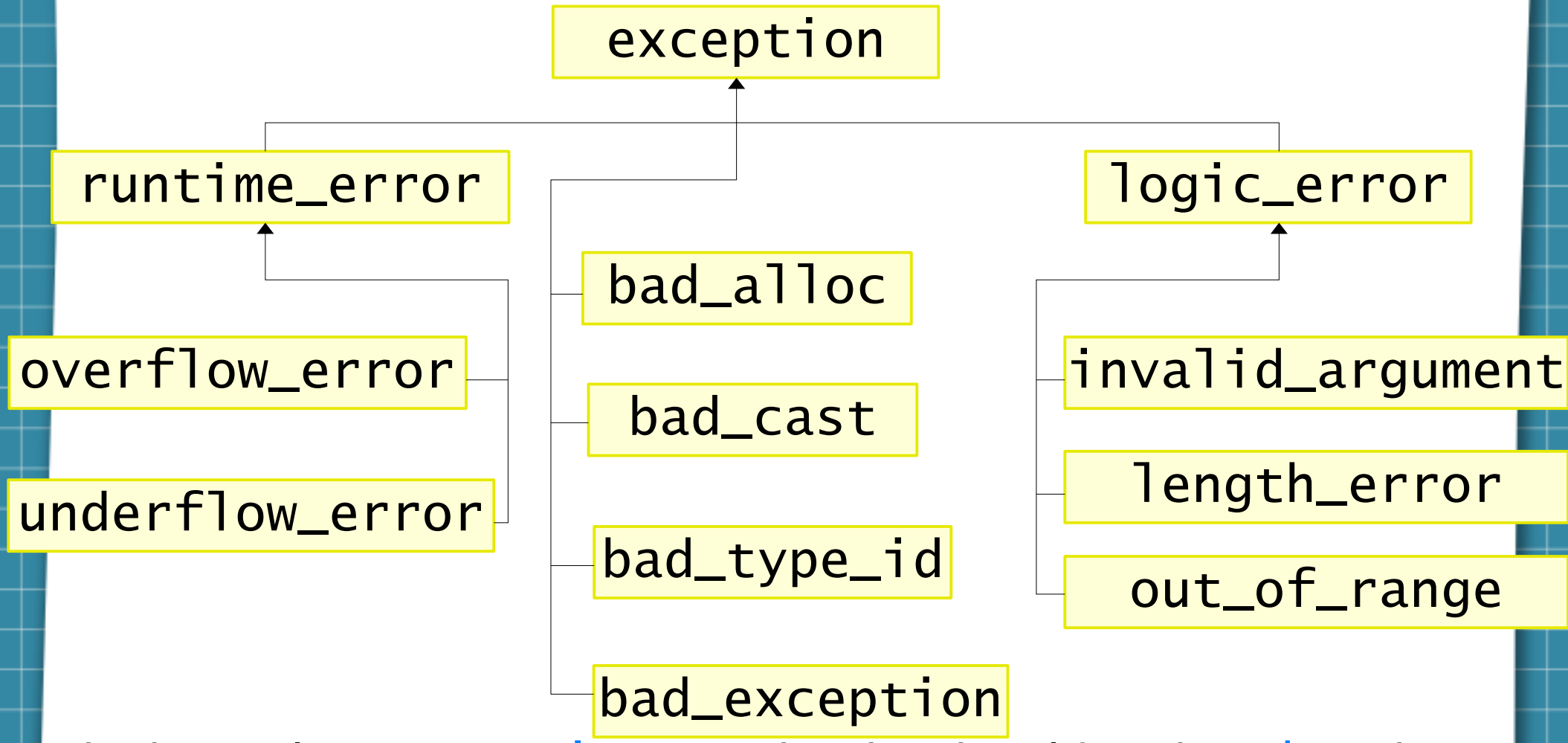
For example: When working with a file, an exception occurred. Upon this, we want to close the file (by the handler) and notify the caller that there was an issue by re-throwing the exception.

Syntax:
```
throw;
```

See an example in rethrowingException.cpp
Note: for this example, we use `class exception`, the C++ standard base exception class. `runtime_error`, `logic_error`, `invalid_argument` classes and many others are its derived classes.

Standard Library Exception Hierarchy

The C++ Standard Library includes a hierarchy of exception classes, some of them are:



The base-class `exception` contains the virtual function `what` that derived class can override to issue an appropriate error message.

Stack Unwinding

When an exception is thrown, but not caught in a particular scope, the function-call stack is *“unwound”*, and an attempt is made to catch the exception in the next outer `try-catch` block.

It means that the function, in which the exception was not caught, terminates: all local variables that have completed initialization are destroyed and the control returns to the statement that invoked the function originally.

If a `try-catch` block is located, the attempt is made to catch the exception.
If not, *stack unwinding* occurs again.

And so forth, up to the program termination.

see stackUnwinding.cpp

When to Use Exception Handling

Exception handling is designed to process *synchronous errors* that occur when a statement executes, such as *invalid function parameters* and *unsuccessful memory allocation*.

Exception handling is not designed to process errors associated with *asynchronous events* that occur in parallel with, and independent of, the program's flow of control. Examples: disk I/O completions, network message arrivals, mouse clicks and keyboard keys pressed. They occur in parallel

When to Use Exception Handling

Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects to understand each other's error-processing code.

It also enables predefined software components (like Standard Library classes) to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Functions That Do Not Throw Exceptions

Starting from C++ 11, if a function does not throw any exceptions and does not call any functions that throw exceptions, we can explicitly state it:

```
bool func(int a, double b) noexcept;

bool f2(int a) const noexcept;
```

* use in both, the prototype and the definition

Constructors, Destructors and Exception Handling

Constructors do not return a value, hence we can

- return an improperly constructed object and expect that anyone using it would determine that it is in incomplete state,

Constructors, Destructors and Exception Handling

Constructors do not return a value, hence we can

- return an improperly constructed object and expect that anyone using it would determine that it is in incomplete state,

  or

- set some variable outside the constructor to indicate that something went wrong

Constructors, Destructors and Exception Handling

Constructors do not return a value, hence we can

- return an improperly constructed object and expect that anyone using it would determine that it is in incomplete state,

  or

- set some variable outside the constructor to indicate that something went wrong

  or

- require the constructor to throw an exception that contains the error information, which allows the program to handle the failure.

Constructors, Destructors and Exception Handling

If an exception is thrown before the object is fully constructed, destructors will be called for any member objects that have been constructed so far.

Constructors, Destructors and Exception Handling

If an exception is thrown before the object is fully constructed, destructors will be called for any member objects that have been constructed so far.

If an array of objects is partially constructed, and an exception occurs, only the destructors for the array's constructed objects will be called.

Constructors, Destructors and Exception Handling

If an exception is thrown before the object is fully constructed, destructors will be called for any member objects that have been constructed so far.

If an array of objects is partially constructed, and an exception occurs, only the destructors for the array's constructed objects will be called.

Also, destructors are called for every automatic object constructed by the `try` block before an exception that occurred in that block is caught.

Constructors, Destructors and Exception Handling

Do not throw exception from the constructor of a *global object* or a *static local object*. Such exception cannot be caught, because they are *constructed before the main function executes*.

Do not forget to release resource, such as dynamically allocated memory, files, etc.

`unique_ptr` and Dynamic Memory Allocation

If an exception occurs after successful memory allocation, but before the `delete` statement executes, the *memory leak* could occur.

`unique_ptr` and Dynamic Memory Allocation

If an exception occurs after successful memory allocation, but before the `delete` statement executes, the *memory leak* could occur.

C++ 11 provides *class template* `unique_ptr` in header `<memory>`.

`unique_ptr` and Dynamic Memory Allocation

If an exception occurs after successful memory allocation, but before the `delete` statement executes, the *memory leak* could occur.

C++ 11 provides *class template* `unique_ptr` in header `<memory>`.

A `unique_ptr` maintains a pointer to *dynamically allocated memory*. When the `unique_ptr` object goes out of scope, its destructor is called, which performs `delete` or `delete[]` operation on the `unique_ptr`'s pointer data member.

`unique_ptr` and Dynamic Memory Allocation

If an exception occurs after successful memory allocation, but before the `delete` statement executes, the *memory leak* could occur.

C++ 11 provides *class template* `unique_ptr` in header `<memory>`.

A `unique_ptr` maintains a pointer to *dynamically allocated memory*. When the `unique_ptr` object goes out of scope, its destructor is called, which performs `delete` or `delete[]` operation on the `unique_ptr`'s pointer data member.

Class template `unique_ptr` provides overloaded operators `*` and `→` so that a `unique_ptr` object can be used just like a regular pointer object.

`unique_ptr` and Dynamic Memory Allocation

<u>Only one</u> `unique_ptr` at time can own a *dynamically allocated object*.

`unique_ptr` and Dynamic Memory Allocation

<u>Only one</u> `unique_ptr` at time can own a *dynamically allocated object*.

When assigning one `unique_ptr` to another, using `move`, the one on the right *transfers ownership of the dynamic memory* in manages to the one on the left of the assignment.

When passing a `unique_ptr` as an argument to another `unique_ptr` constructor, the ownership is transferred as well.

`unique_ptr` and Dynamic Memory Allocation

<u>Only one</u> `unique_ptr` at time can own a *dynamically allocated object*.

When assigning one `unique_ptr` to another, using `move`, the one on the right *transfers ownership of the dynamic memory* in manages to the one on the left of the assignment.

When passing a `unique_ptr` as an argument to another `unique_ptr` constructor, the ownership is transferred as well.

The “last” `unique_ptr` object that maintains the pointer to the dynamic memory will delete the memory.

See someClass.h and unique_ptrExample.cpp

HW assignment

(1) Consider the following program:

```
#include <iostream>
using namespace std;
int main()
{
    int donuts, milk;
    double dpg;
    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
            throw donuts;
        dpg = donuts / static_cast<double>(milk);
        cout << donuts << " donuts.\n"
            << milk << " glasses of milk.\n"
            << "You have " << dpg
            << " donuts for each glass of milk.\n";
    }
    catch (int e)
    {
        cout << e << " donuts, and No Milk!\n"
            << "Go buy some milk.\n";
    }
    cout << "End of program.\n";
    return 0;
}
```

Without running the program, what will be the output if 4 and 0 are entered when the program is run?

HW assignment

(2) Consider the following code fragment:

```
#include <iostream>
#include <memory>

class Task
{
public:
    int mId;
    Task(int id ) :mId(id)
    { std::cout<<"Task::Constructor"<<std::endl; }
    ~Task()
    { std::cout<<"Task::Destructor"<<std::endl; }
};

int main()
{
    // Create a unique_ptr object through raw pointer
    std::unique_ptr<Task> taskPtr{ std::make_unique<Task>(23) };
    //Access the element through unique_ptr
    int id = taskPtr->mId;
    std::cout<<id<<std::endl;
    return 0;
}
```

Without running the program, what will be the output when it is run ?

HW assignment

Self-Study:
Section 17.8

Suggested Practice:
Chapter 17 Summary and Self-Review Exercises