Advanced Programming Concepts with C++

Exception Handling

Fall 2021

CSI2372[A]

Amir EAMAN
Postdoc Researcher
Faculty of Engineering
uOttawa

Exception — error

can throw value. or object.

→ using → "throw (....)"

→ catch (↓) argument type need to be match

catch by reference

Catch (....) can catch anything for all exception

→ Dynamic_cast.

std: :bad_cast → build in exception class.

Exception

- An **exception** happens in an abnormal run of programs.
- **Exceptions** are anomalous or extraordinary conditions **during the execution** of programs
- **Exceptions** require special processing.
- The term **exception** represents a *data structure* (an object or a data type) storing information about the extraordinary condition which happens during abnormal, unpredictable situations. The special object which compiler used is known as the **exception object**.

Examples of **exceptions**)

- Incorrect format of input argument
- A file is missing, out-of-memory errors (resources are unavailable)
- Stack overflows
- No memory for dynamic allocation
- A network connection has been lost.

If a system does not have any **exceptions**, routines should return some special **error code**.

Exception Handling

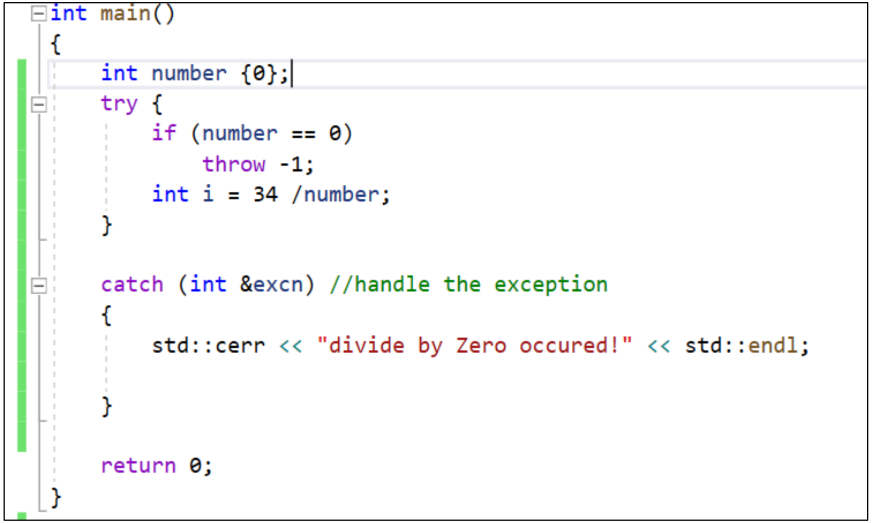
- If an **exception** occurs, the program's normal flow of execution changes and a pre-registered **exception handler** is executed.
- **Exception Handling** is the process of taking care of the occurrence of **exceptions**.
- Exception Handling recovers a program from exceptions.
- One mechanism to handle exceptions is to transfer control (i.e., raise an exception) to a ***catch***.
- Raising an exception (i.e., **throwing** an exception) signals that an exception in normal execution of the program has happened.
- One part of the program can detect a abnormal condition and can pass the job of handling that problem to another part of the program.

- If there is no exception handling for an occurred exception, the program **terminates** abruptly.

Exception Handling

- When an exception raised through executing a **throw** statement, the statement(s) after the **throw** statements are not executed. This is because control is transferred from the throw to the matching catch. In this regard, throw works like a *return* statement.

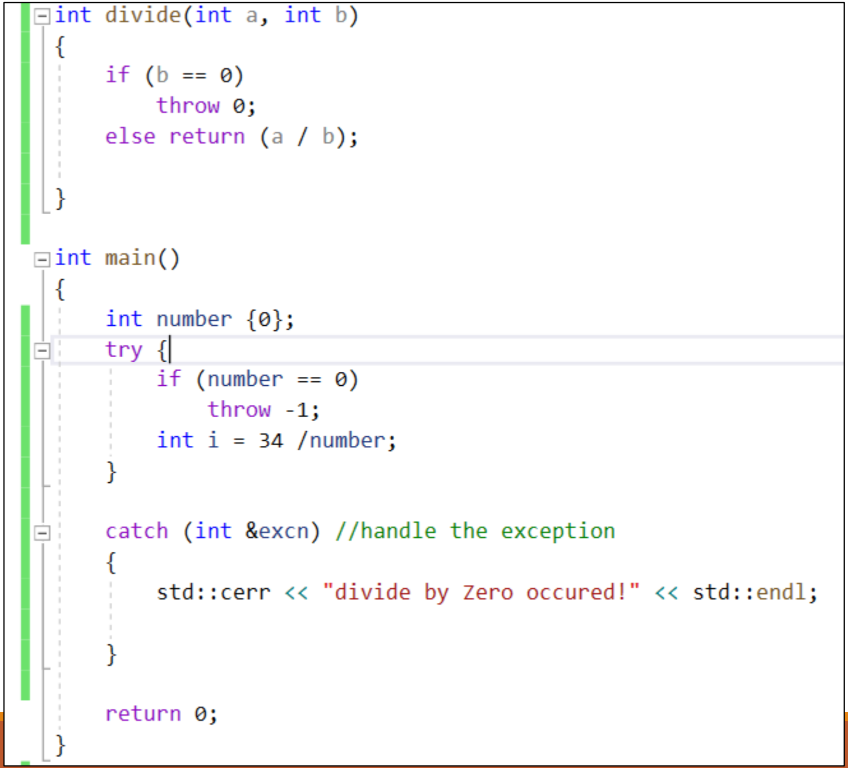
Throw and Catch of Exceptions



```
int main()
{
    int number {0};
    try {
        if (number == 0)
            throw -1;
        int i = 34 /number;
    }

    catch (int &excn) //handle the exception
    {
        std::cerr << "divide by Zero occured!" << std::endl;

    }

    return 0;
}
```

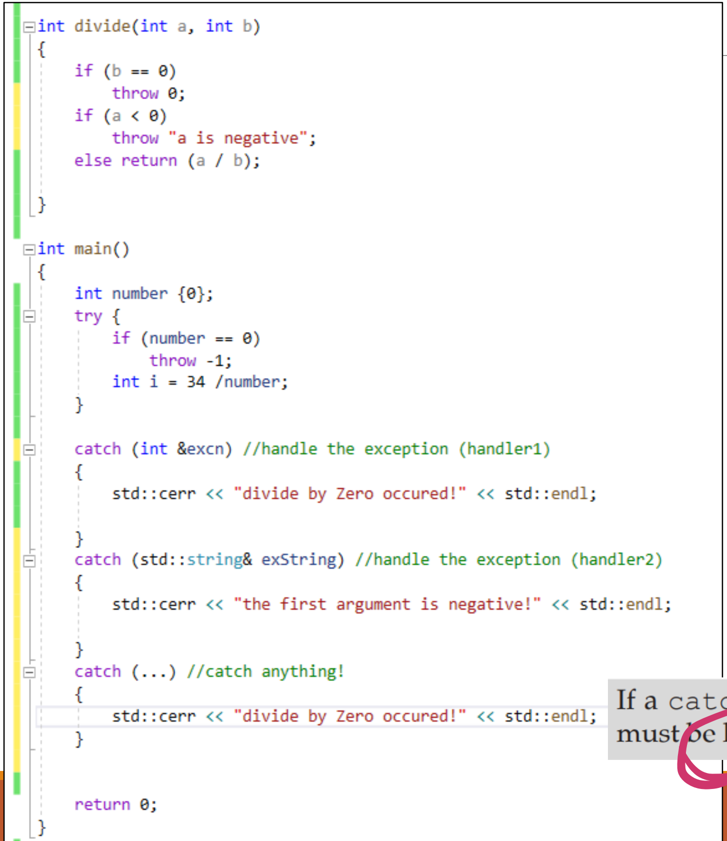
- Remember to: **throw** **by** **value** **and** **catch** **by** **reference.**

Throw and Catch of Exceptions- from functions



```
int divide(int a, int b)
{
    if (b == 0)
        throw 0;
    else return (a / b);

}

int main()
{
    int number {0};
    try {
        if (number == 0)
            throw -1;
        int i = 34 /number;
    }

    catch (int &excn) //handle the exception
    {
        std::cerr << "divide by Zero occured!" << std::endl;

    }

    return 0;
}
```

Throw and Catch of Exceptions- multiple exceptions



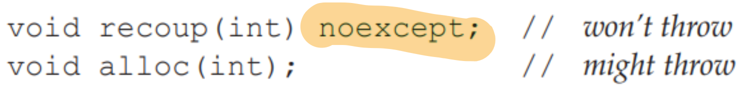
```
int divide(int a, int b)
{
    if (b == 0)
        throw 0;
    if (a < 0)
        throw "a is negative";
    else return (a / b);

}

int main()
{
    int number {0};
    try {
        if (number == 0)
            throw -1;
        int i = 34 /number;
    }

    catch (int &excn) //handle the exception (handler1)
    {
        std::cerr << "divide by Zero occured!" << std::endl;

    }
    catch (std::string& exString) //handle the exception (handler2)
    {
        std::cerr << "the first argument is negative!" << std::endl;

    }
    catch (...) //catch anything!
    {
        std::cerr << "divide by Zero occured!" << std::endl;
    }


    return 0;
}
```

The Catch-All Handler

If a `catch(...)` is used in combination with other `catch` clauses, it must be last. Any `catch` that follows a catch-all can never be matched.

The noexcept Exception Specification

- **noexcept**: we know at compile-time that a function can throw an exception. So, it denotes that whether or not a function can throw exception.
- We should mark functions that cannot possible throws exception **noexcept.**



```
void recoup(int) noexcept;   // won't throw
void alloc(int);             // might throw
```

- If a noexcept function throws an exception, the function **std::terminate()** is called, therefore, the program terminates.

The noexcept Exception Specification – not throwing functions!

- **noexcept specifier**: we know at compile-time that a function can throw an exception. So, it denotes that whether or not a function can throw exception.
- We should mark functions that cannot possible throws exception **noexcept.**

```
void recoup(int) noexcept;    //  won't throw
void alloc(int);              //  might throw
```

- If a noexcept function throws an exception, the function **std::terminate()** is called, therefore, the program terminates.
- Some functions which are **noexcept** by default:
  - default constructors
  - copy constructors
  - copy assignment operators
  - destructors
  - move constructors
  - move assignment operators

3rule
CC
CAO
D

Never
throw
exception

→ If class has a pointer of any kind. we must implement 3 methods.

Stack unwinding – the search for a matching **catch** clause!

- **Stack unwinding** is the process of destroying all the local objects (automatic variables) of a function, which are allocated on stack. During **Stack unwinding**, the function call stack is destroyed, which causes the destructors of the local variables get called. → potential danger situation of memory leak: Solution: Remember to delete allocated resources in destructors of classes (resource relinquishing in destructors).
- **Stack unwinding** searches the chain of nested function calls to find the catch clause for the exception.

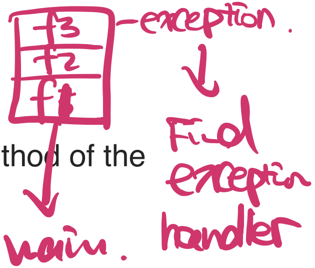
f1 → f2 → f3

Chain of function

- When **Stack unwinding** happens?
  - When an *exception* has been raised but is not yet handled.
  - Or when a function's scope is exited *normally* by reaching the end of the scope.

- C++ finds the handler of an exception by using **Stack unwinding**.

- If **Stack unwinding** cannot find any handler for the exception and it reaches the main() method of the program, the program terminates abruptly (an not handled exception has occurred).

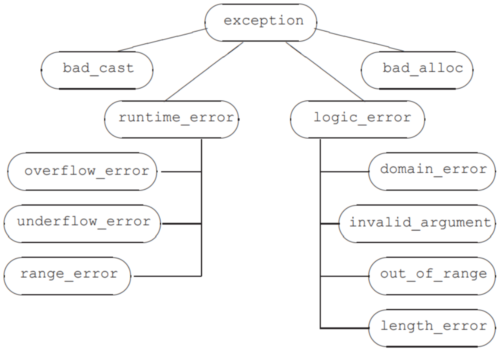


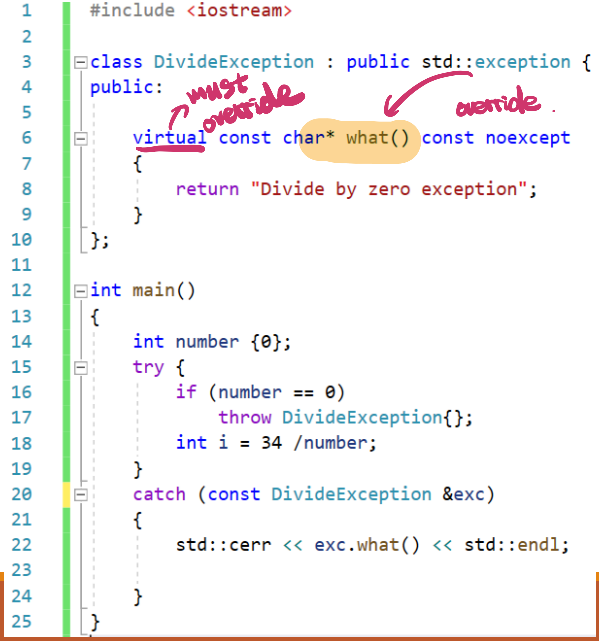
Exception Class Hierarchy

- **std::exception** is the base class of standard exception hierarchy.
- All Standard exception subclasses implement the **what()** virtual function of the base class **std::exception**.
- **what()** virtual function provides the description of the exception, which is used to identify the exception.

```
virtual const char* what() const noexcept;
```

Figure 18.1: Standard exception Class Hierarchy





```
#include <iostream>

class DivideException : public std::exception {
public:

    virtual const char* what() const noexcept
    {
        return "Divide by zero exception";
    }
};

int main()
{
    int number {0};
    try {
        if (number == 0)
            throw DivideException{};
        int i = 34 /number;
    }
    catch (const DivideException &exc)
    {
        std::cerr << exc.what() << std::endl;

    }
}
```