Lecture 11

C++ Programming

Arne Kutzner
Hanyang University / Seoul Korea

C++
Exception Handling

Basic Concepts

- An ***exception*** is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called ***exception handling***
- The exception handling code unit is called an ***exception handler***

Exception Handling

- In a language without exception handling
  - Programmers are responsible for implementing some form of error detection and recovery mechanism
- In a language with exception handling
  - Programs are allowed to trap specific errors by using exceptions, thereby providing the possibility of fixing the problem and continuing
  - Exception handling separates error-handling code from other programming tasks, thus making programs easier to read and to modify.

`try-catch` blocks

- Basic Syntax:

```
try {
   statements;
}
catch (formal parameter) {
   handler for exception e
}
```

- **`catch`**-block must immediately follow the **`try`**-block
- The Catch block is only called if there is some trouble in the try block
  - When processing of a **`catch`**-block is complete, execution continues with the statement following the **`catch`**-block. (We don't go back into the try block!)

Catch block - Formal Parameter

- The formal parameter can consists of
  - Simply a Type-name
  - Variable plus Type-name
    Then the formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis (…), in which case it handles all exceptions

Throwing Exceptions

- Exceptions are **raised** explicitly by the statement:

  `throw` *expression*`;`

- The type of the expression disambiguates the intended handler

Example for try, catch, throw

```
void foo() {
      try {
            if (error_condition) {
                  throw 1;
            }
      }
      catch (int i) {
            some_statments;
      }
}
```

try some code

throw exception

catch exception

Multiple Exception Handlers

- ```
  try {
  -- code that is expected to raise an exception
  }
  catch (formal parameter1) {
  -- handler code 1
  }
  catch (formal parameter2) {
  -- handler code 2
  }
  ...
  ```
- C++ distinguishes the handler by the type of the formal parameter (Same technique as for overloading of functions)
  - Prototype of each formal parameter must be unique
  ```

Unhandled Exceptions

- An **unhandled exception is propagated** to the caller of the function in which it is raised
- This propagation can continue until reaching the main function
- If no handler is found, some default handler is called

Example for Exception Propagation

```
void foo_1() {
      if (1) {
            throw 1;
      }
}

void foo() {
      try {
            foo_1();
      }
      catch (int i) {
            statements;
      }
}
```

No handler in `foo_1`!

In this example the exception is catched in function `foo`

try block

catch block

Nested try-catch

- Try-catches can be nested like if-statements or loops
- If none of the inner catch-blocks matches, we continue our search in the outer blocks

Example Nested try-catch

```
void foo() {
        try {
                try {
                        throw (double)1;
                }
                catch (int i) {
                        std::cout << "inner catch. i = " << i;
                }
        }
        catch (double d) {
                std::cout << "outer catch. d = " << d;
        }
}
```

inner block

outer block

Rethrowing Exceptions

- In a handler some **`throw`** without an operand can appear
- When it appears, it simply re-raises the exception, which is then handled elsewhere

Example 1 for Rethrowing

```
void foo() {
	try {
		try {
			throw (double)1;
		}
		catch (double d) {
			std::cout << "inner catch. d = " << d;
			throw;
		}
	}
	catch (double d) {
		std::cout << "outer catch. d = " << d;
	}
}
```

Here we rethrow the exception d (the double value 1)

Example 2 for Rethrowing

```
void foo() {
        try {
                throw (double)1;
        }
        catch (double d) {
                std::cout << "catch 1 d = " << d;
                throw;
        }
}
void main() {
        try {
                foo();
        }
        catch (double d) {
                std::cout << "catch 2 d = " << d;
        }
}
```

Here we rethrow the exception d (the double value 1)

Creation of Exception Classes

- So far we used primitive values as exceptions
- A better style is the development of a special Exception class.
  - Objects of this class should contain precise information about the problem

Exception Class – catch by reference

```
class Exception {
public:
        int code;
        Exception(int i) {
                code = i;
        }
};
```

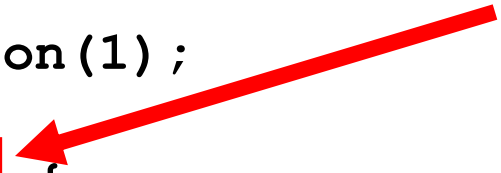
```
void foo() {
        try {
                throw Exception(1);
        }
        catch (Exception &e) {
                std::cout << e.code;
        }
}
```

No copy of the exception object

Exception Class – catch by value

```
class Exception {
public:
        int code;
        Exception(int i) {
                code = i;
        }
};
```

Creates copy of exception object



```
void foo() {
        try {
                throw Exception(1);
        }
        catch (Exception e) {
                std::cout << e.code;
        }
}
```

Exception Class use with new

```
class Exception {
public:
        int code;
        Exception(int i) {
                code = i;
        }
};
void foo() {
        try {
                throw new Exception(1);
        }
        catch (Exception *e) {
                std::cout << e -> code;

                delete e;
        }
}
```

Some special class for exceptions

Creation of exception object

Don't forget deleting your object if it is no longer required

Important!

- Note that as some exception propagates, it causes functions and blocks it passes up through to terminate, and hence to call the destructors of any local objects
- This behavior is known as "stack unwinding"

Cautions When Using Exceptions

- Be aware, that exception handling requires additional time and resources because it requires
  - rolling back the call stack, and
  - propagating the errors to the calling methods
  - temporary allocation of additional memory

Differences to the approach in Java

- In C++ we can’t catch system errors as e.g.
  - Null-Pointer Exceptions
  - Division by Zero
- There is no finally clause In C++
- C++ doesn’t know the throws-keyword (claiming of exceptions)
- In the Java exceptions have to be instances of the class Exception
  - Complete class hierarchy for several forms of exceptions is already part of the Java-specification