Föreläsning 4
Exceptions

Why and where should you use exceptions?

- Different in different language
- In C++ exceptions are used to signal “Exceptional Circumstances”
- If you use them to signal “not found” when writing a search algorithm you are not doing it right

Basic

- An exception is raised by the statement “throw”.
- The exception is catch by the closest (matching) handler:

```
try {
    std::string s;
    char * temp1 = new char[10]();
    char * temp2 = new char[10]();
    //Som code that may raise an exception
    throw MyException("Test");
}
catch (MyException & e) {
    delete temp2;
    delete temp1;
    throw;
}
```

Why exception?

- Assume that we are loading a saved state for e.g. a game from a file. Suddenly in the middle of reading there is an error (end of file, no file exist, the completely wrong data)

- This is exceptional! Should we raise/throw an exception or just give a “not succeded” return code?

Where do we want to do anything about the problem? Local or “global”

- If the right action is to try to read again we should do this locally.
- But if we are unable to load the saved state there is no way to handle this locally
  - throw an exception!

- The alternative is to have a lot of error code checking traversing slowly up in the call chain until reaching the right level.

- Exceptions jumps right out of it!
  No need for checking error conditions etc.

Exception cost

- The modern mechanics for handling exception. Match exception place with a prepared set of error handlers
- Negligibly cost when no exceptions
- But a throw/catch will always cost more than normally program execution

Exception safety alternatives: (from Wikipedia)

- No-throw guarantee:
  - Operations are guaranteed to succeed and satisfy all requirements even in exceptional situations. If an exception occurs, it will be handled internally and not observed by clients.
- Strong exception safety, also known as commit or rollback semantics:
  - Operations can fail, but failed operations are guaranteed to have no side effects, so all data retain their original values.
- Basic exception safety, also known as a no-leak guarantee:
  - Partial execution of failed operations can cause side effects, but all invariants are preserved and there are no resource leaks (including memory leaks). Any stored data will contain valid values, even if they differ from what they were before the exception.
- No exception safety:
  - No guarantees are made.

In reality

- Basic exception safety everywhere
- Stronger for some cases!
- But a general rollback police cost to much!

A small example

```
try {
      std::string s;
      char * temp1 = new char[10]();
      char * temp2 = new char[10]();
      //Som code that may raise an exception
      throw MyException("Test");
 }
 catch (MyException & e) {
      delete temp2;
      delete temp1;
      throw;
 }
```

Problem!
Every new can give exception!

```
char * temp1 = new char[10]();
```

Correct code!

```
void ExceptionExampelFixad() {
    char * temp1 = nullptr;
    char * temp2 = nullptr;

    try {
        std::string s;
        temp1 = new char[10]();
        temp2 = new char[10]();
        //Som code that may raise an exception
        throw MyException("Test");
    }
    catch (e) {
        delete temp2;
        delete temp1;
        throw;
    }
    catch (...) {
        //Error handling code here
        //The last statement is often “throw; ” that raise the
same exception again.
        throw;
    }
}
```

Let the Compiler take care of the problems!

```
class CharPtr {
    char * ptr;
public:
    CharPtr(int len): ptr(new char[len]) {}
    ~CharPtr() {
        delete[] ptr;
    }
};
void ExceptionByCompiler() {
    CharPtr temp1(10);
    CharPtr temp2(10);
    //Som code that may raise an exception
    throw MyException("Test");
}
```