

## Reporting errors

## Lecture 15

### Errors and exceptions

We have mentioned the word **exception** quite a few times during the course.

In this lecture we will understand what is an exception, what will happen when an exception occurs, and how to use it to deal with errors.

References:

- Textbook: Chapter 10, until page 538.

When we write a program that contains multiple functions, classes and even libraries, it is common that something **unexpected** can occur. E.g.,

- The program can **run out of memory** when allocating a large structure. This happens more often if you have a more primitive computer (e.g., a Palm).
- The library needs to open a file, but the file is not there.
- The library tries to establish a connection with a remote computer through the network, but the **network is down**.
- Your library needs to dereference a pointer, but the pointer is found to be a **NULL pointer**.
- The user code supply some **invalid data** to the function. E.g., ask the function to create an array which number of elements is negative.

PMOOP(0396A)

PMOOP(0396A)-15.1

## The need for error reporting

The problem of error handling:

- In general, implementation code **can detect** an error if they want to. But they **don't know what to do with the error**
- On the other hand, user code usually **knows what to do against errors**. But due to abstraction, they **don't know the implementation details**, so they **don't have a way to detect them!**
- The user might want to **ignore** the error, to **correct** the problem, to **emit an error message**, or to **exit** the program. The implementer **shouldn't unconditionally use one** of these possible strategies.
- Instead, the library writers need a way to **communicate the fact of failure** to the library user.

But there is one difficulty: errors are **not common**, so we want to **avoid complicated code** to deal with them.

PMOOP(0396A)-15.2

## Strategies for error handling

Upon an error, an implementation can do one of the followings:

- **Correct** the error. If this is possible, it should be done. E.g., if the user adds an element to an array, and the array is found to be too small, the implementation should **allocate a larger array** and copy old element over to the new array.
- **Return** an error code, to indicate that something wrong happened. This should be used **if the error usually occurs**, so it is reasonable to expect users to always deal with them.  
In case of member functions, a variant is to mark the object as erratic.
- **Exit** the program immediately. This makes it impossible for any recovery by the user code, so it should only be used **if the program state is known to be badly corrupted**.
- **Throw an exception**. This should be used whenever the error is rarely seen.

PMOOP(0396A)-15.3

## Problem of error-code

Before trying to understand what is an exception, let's first see why returning error code is not good enough in some situations.

- One problem of using error-code is that **programmers keep forgetting to check every operation for errors**.
- The operation that causes the error goes **completely undetected**. The program ends up **creating problems that are even more serious**.
- Even if the code actually gets written, it is **unlikely to be well tested** because the error situation **usually doesn't happen**.
- And if the code actually gets written and tested, it makes the user program **very difficult to read**, when the logical flow of the program is obscured by the need to check for errors.

We want not to need **writing code that check for exceptional conditions unless we know how to handle an error**.

PMOOP(0396A)-15.4

## Exceptions: concepts

**Exception throwing** is the mechanism that support this idea. Three keywords are introduced: **throw**, **try** and **catch**.

- When an **exceptional condition** occurs, gets **detected** but no **recovery** is known, an exception can be **throw'n**.
- The C++ statement that throws an exception looks like this:  

```
throw "my_error";
```
- **What can be placed** after the **throw** keyword? It is an "exception object". The type of the exception object controls which exception handling code gets executed.
- Any object can be an exception object, including built-in types (here, **const char\***) and user defined classes.

We will see later that in larger project, it is better to have a group of classes reserved for such "exception objects".

PMOOP(0396A)-15.5

## Example exception throwing code

```
#include <iostream>
class Complex {
public:
    class ZeroDivide {}; // A class just for throwing
    Complex(double r=0, double i=0): re(r), im(i) {};
    divide(Complex n) {
        if (n.re == 0 && n.im == 0) throw ZeroDivide();
        ...
    }
private:
    double re, im;
};

void f(Complex &a, Complex &b) {
    a.divide(b);
    cout << "The result of division is " << a << endl;
}
```

PMOOP(0396A)-15.6

## Exception handling

What will happen when an exception occurs?

- If **no code can handle** the exception, then **the program is terminated** by executing a built-in function *terminate()*.  
The g++ terminate function will send an Abort signal to itself, causing a core-dump. This is nice, because then you can use gdb to see what happened.
- So, **what if the caller knows better** about how to deal with the problem? How to prevent the program from being terminated this way?
- A **try-catch** block can be used to specify what to do when an exception is thrown.
- It consists of two parts:
  - The **try** block specify a group of code **sharing the same exception handling strategy**.
  - Each **catch** block specifies code to **handle one type of exceptions**.

PMOOP(0396A)-15.7

## Format of try-catch blocks

A **try-catch** block looks like this:

```
try {
    ...; // Statements to be executed
    ...; // There can be many statements in a try block
    ...; // All sharing the same error handling strategy
} catch (Type1 t) { // This handle exceptions of Type1
    ...; // Again, can contain many statements
} catch (Type2 t) { // There can be many handlers, each of a different type
    ...;
} catch (...) { // Optionally, there can be a "catch-all" handler
    ...; // If error is not caught before, it is caught here.
}
...; // If there is no exception thrown in the try block,
...; // or if the error is caught, the program continues here.
```

PMOOP(0396A)-15.8

## Example catch block

Here is a *main()* function which deal with an error by printing a message.

```
int main() {
    try {
        double r1, i1, r2, i2;
        cin >> r1 >> i1 >> r2 >> i2;
        Complex a(r1, i1), b(r2, i2);
        f(a, b);
    } catch (Complex::ZeroDivide e) {
        cerr << "Division by zero!" << endl;
    } catch (...) { // All other types of exceptions
        ...
    }
    // Program continue here
}
```

One important thing to note: the exception **needs not be handled in the same function** where the exception is first thrown.

Usually it is not: If you can handle an error locally, you don't need exception.

PMOOP(0396A)-15.9

## The glory details

- When an exception is thrown, the exception object is assigned as the **current exception object**.
- The block currently running is assigned as the **current block**.  
A block is a group of statements enclosed by braces.
- The program repeatedly does the following:
  - All **objects** of the current block are **destroyed**.
  - If the current block is **not a try block**, or if the try block **does not have a catch block** of a type of the current exception object, then:
    - If the block is a function, the current block becomes the **block of the caller**. Otherwise the current block becomes the enclosing block.
    - The process is repeated.
  - Otherwise, the correct catch block is executed. Execution resumes after the try-catch block.

PMOOP(0396A)-15.10

## Understanding our example

What happens when our example code gets executed?

- The main program creates two *Complex* objects, and calls *f* on it. It does this in a try block, **anticipating errors**.
- The *f* function does not anticipate error. It calls the *divide* member function for *a*, giving it an argument *b*.
- The *divide* member function of *Complex* **don't know how to deal with error**, but it can detect it by finding that *n* is 0. So it throws a *ZeroDivide* exception.
- The *divide* function has no error handler, so the exception **propagates** to *f*. The same happens to *f*, so the exception propagates to *main*.
- The exception handler of the *main* function **catches the error**, prints an error message, and ignore the remainder of the try-catch block.  
If a good error handler is not found, *terminate()* is called.

PMOOP(0396A)-15.11

### A few common questions

- Q1:** What will happen to the **remaining statements within the try block**, after the exception is thrown?
- A1:** It will not be executed. Instead, execution continues **after the try-catch block** that handles the error.
- Q2:** After I catch the error, will the code triggering the error be **executed again** automatically?
- A2:** No. If you want to **retry**, you have to do it yourselves, usually by **making a loop enclosing the try block**.
- Q3:** What happen if there is **no catch clause of correct type**?
- A3:** The run-time system continue to search for an exception in the **enclosing block**, or in the **caller function**.
- Q4:** What happen if no exception is thrown in a try block?
- A4:** The catch blocks will not be executed.

PMOOP(0396A)-15.12

### How exception compares to other error handling mechanisms?

Note how exceptions differ from other ways of error reporting:

- By default, an exception is **propagated** rather than get ignored.
- Exception **does not dictate the strategy** to handle the error. The user will determine whether the default is acceptable, and if not catching it.
- The user can decide that **only some of the exceptions are to be caught**, and apply the default for others.
- Many statements can **share the same exception handlers**. For example, if we do a lot of Complex operation and want to return a special value in case of numeric error, we enclose the whole thing within one large try block.
- No statement explicitly tells **where the exception comes from**, so it is more difficult to find out why a handler gets executed.

PMOOP(0396A)-15.13

### Why iostream don't use exception?

If exceptions are this nice, why doesn't iostream throw exceptions?

- **Many I/O failures should be "expected"**. E.g., any reasonable program that reads a stream of integers should know what to do in case something fails.
- And all should know what to do in case **end of file** is reached.
- If a failure is frequent enough that **it usually must be dealt with immediately**, there is little to be earned by using exceptions.  
The code that uses exception would then be more complicated and less intuitive than the code that doesn't use exception.
- On the other hand, **some I/O failures can be treated as unexpected**.  
E.g., if you read characters from a stream, you probably won't expect that the stream suddenly becomes bad.
- So C++ designers allow the programmer to **specify some error conditions to be exceptional** and thus should trigger an exception.

PMOOP(0396A)-15.14

### Example: Exceptions in iostream (need g++ 3.0)

Under the C++ standard, *ios* has an "exception state" associated with it. What will trigger an exception depends on this state.

It can be the bitwise-or of any combination of the values **ios::badbit**, **ios::failbit** and **ios::eofbit**, indicating that exceptions should be thrown when the stream becomes bad, fails or reaches EOF.

You can set and get the exception state by using the member function *exceptions* of the *ios*:

- *st.exceptions(flag)*: set the exception flag of the stream *st* to *flag*.
- *st.exceptions()*: get the current exception flags.

**Example:** *cin.exceptions(ios::badbit | ios::failbit)* asks *cin* to throw exception on both bad and fail, but not on eof.

PMOOP(0396A)-15.15

### Example

```
int main() {
    cin.exceptions(ios::badbit);
    try {
        int a, b, c;
        cin >> a;
        while (!cin.eof()) {
            cin >> b >> c;
            if (cin) do_work(a, b, c);
            else if (cin.eof()) cerr << "Incomplete last line.\n";
            else {
                cerr << "Improper format, skipping line.\n";
                cin.clear(); cin.ignore(INT_MAX, '\n');
            }
            cin >> a;
        }
    } catch (ios_base::failure) {
        cerr << "Unexpected error!\n";
    }
}
```

PMOOP(0396A)-15.16

### Where exceptions are bad

Note that in the code, we do expect that EOF will be reached at the end. In this case, it is not really nice to have it throwing exceptions:

```
int main() { ... // very ugly use of exception
    try {
        cin >> a;
        while (true) {
            try { cin >> b >> c; }
            catch (...) {
                if (cin.eof()) { cerr << "Incomplete last line.\n"; exit(1); }
                else throw;
            } ...
            cin >> a;
        }
    } catch (...) {
        if (!cin.eof()) cerr << "Improper format, skipping line.\n";
    }
}
```

PMOOP(0396A)-15.17

## Re-throwing exceptions

- There are **3 different situations** in which you want to throw an exception within the catch block:
  - There might be a type of exception in which **sometimes you know how to handle it but sometimes you don't**.  
E.g., you catch an error telling that connection can't be established, but you only know how to deal with it if it reports a temporary error.
  - You might want to do **part of the error handling**, but the **handling is not complete**. E.g., you might want to free up some memory.
  - An error occurs to the error handling code itself.
- In first 2 cases, we want to rethrow the current exception object. This can be done by the statement “**throw**”.
- In the third case, we create a **new exception object** to throw.

PMOOP(0396A)-15.18

## Example

One possible use of re-throw:

```
int f() {
    int *v = 0;           // Be careful about this!!
    try {
        v = new int [10]; // No such problem if we use vector
                          // something that may throw
        delete[] v;
    } catch(...) {       // Clean up the memory
        if (v)           // Guard against allocation error
            delete[] v;
        throw;          // Rethrow the exception
    }
}
```

However, this is **considered to be poor style**. If we use `vector<int>(10)` instead of an `int*`, the memory is automatically freed up.

What if it is not an array, but is a simple object? We will see it in the next lecture.

PMOOP(0396A)-15.19

## Scope of exception handler

- In the code of last page, note that we **write `int* v` before the try block**.
- It is necessary. Once an exception is thrown, all variables of the try block becomes **out-of-scope**.
- They are thus cleaned up **before the catch block executes**.
- That means the exception handler **cannot use the variables within it**: they no longer exists.
- But the **memory pointed-to by the pointer** is still there. This is the problem that the try-catch block try to solve.
- We try to make sure that the value of `v` is well-defined, even in case of an exception when `new` is executed.
- On memory exhaustion, an exception of type `bad_alloc` is thrown by `new`. The code made sure that in such case, `v = 0`.

PMOOP(0396A)-15.20

## Standard exceptions

Apart from `bad_alloc`, many other exception types can be thrown by the language or by standard library:

- `bad_alloc` (<new>): no more memory in `new`.
- `bad_cast` (<typeinfo>): failure in `dynamic_cast` of reference.
- `bad_typeid` (<typeinfo>): trying to get the type-id of a NULL pointer.
- `bad_exception` (<exception>): if a function says it won't throw a type of exception but do so anyway (next lecture).
- `out_of_range` (<stdexcept>): if out of range is detected by a container.
- `invalid_argument` and `overflow_error` (<stdexcept>): due to bit-sets (we didn't learnt it).
- `ios_base::failure` (`ios`): I/O failure of `iostream`.

PMOOP(0396A)-15.21

## The exception class

All the types are derived from an exception class, defined like this in <exception>:

```
class exception {
public:
    exception() throw ();           // we will see what the throw() means soon
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
private:
    //...
};
```

In other words, if you `catch (exception&e)`, then you can catch all standard errors, and print the exception by printing `e.what()`.

If you define your own exceptions in a large project, it is a **good idea to derive new exception classes from `exception`**.

PMOOP(0396A)-15.22