

15. C++ advanced (IV): Exceptions

Some operations that can fail

- Opening files for reading and writing

```
std::ifstream input("myfile.txt");
```

- Parsing

```
int value = std::stoi("12-8");
```

- Memory allocation

```
std::vector<double> data(ManyMillions);
```

- Invalid data

```
int a = b/x; // what if x is zero?
```

Possibilities of Error Handling

- None (inacceptable)
- Global error variable (flags)
- Functions returning Error Codes
- Objects that keep error status
- Exceptions

Global error variables

- Common in older C-Code
- Concurrency is a problem.
- Error handling at good will. Requires extreme discipline, documentation and litters the code with seemingly unrelated checks.

Functions Returning Error Codes

- Every call to a function yields a result.
- Typical for large APIs (e.g. OS level). Often combined with global error code.¹⁹
- Caller can check the return value of a function in order to check the correct execution.

¹⁹Global error code thread-safety provided via thread-local storage.

Functions Returning Error Codes

Example

```
#include <errno.h>
...

pf = fopen ("notexisting.txt", "r+");
if (pf == NULL) {
    fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
}
else { // ...
    fclose (pf);
}
```

Error state Stored in Object

- Error state of an object stored internally in the object.

Example

```
int i;  
std::cin >> i;  
if (std::cin.good()){// success, continue  
    ...  
}
```

Exceptions

- Exceptions break the normal control flow
- Exceptions can be thrown (throw) and caught (catch)
- Exceptions can become effective accross function boundaries.

Example: throw exception

```
class MyException{};

void f(int i){
    if (i==0) throw MyException();
    f(i-1);
}

int main()
{
    f(4);
    return 0;
}
```

Example: throw exception

```
class MyException{}
```

```
void f(int i){  
    if (i==0) throw MyException();  
    f(i-1);  
}
```

```
int main()  
{  
    f(4);  
    return 0;  
}
```

terminate called after throwing an instance of 'MyException'
Aborted

Example: catch exception

```
class MyException{};

void f(int i){
    if (i==0) throw MyException();
    f(i-1);
}

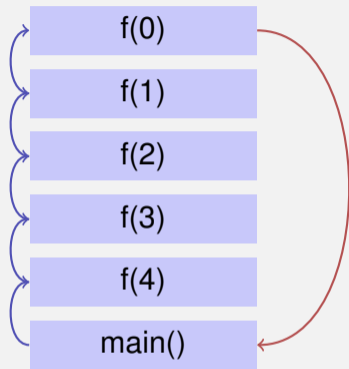
int main(){
    try{
        f(4);
    }
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```

Example: catch exception

```
class MyException{};

void f(int i){
    if (i==0) throw MyException();
    f(i-1);
}

int main(){
    try{
        f(4);
    }
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```



exception caught

Resources get closed

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
    if (i==0) throw MyException();
    SomeResource x;
    f(i-1);
}
int main(){
    try{f(5);}
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```

Resources get closed

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
    if (i==0) throw MyException();
    SomeResource x;
    f(i-1);
}
int main(){
    try{f(5);}
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```

```
closed resource
closed resource
closed resource
closed resource
closed resource
exception caught
```

When Exceptions?

Exceptions are used for *error handling* exclusively.

- Use `throw` only in order to identify an error that violates the post-condition of a function or that makes the continued execution of the code impossible in an other way.
- Use `catch` only when it is clear how to handle the error (potentially re-throwing the exception)
- Do *not* use `throw` in order to show a programming error or a violation of invariants, use `assert` instead.
- Do *not* use exceptions in order to change the control flow. Throw is *not* a better return.

Why Exceptions?

This:

```
int ret = f();  
if (ret == 0) {  
    // ...  
} else {  
    // ...code that handles the error...  
}
```

may look better than this on a first sight:

```
try {  
    f();  
    // ...  
} catch (std::exception& e) {  
    // ...code that handles the error...  
}
```


Why exceptions?

Truth is that toy examples do not necessarily hit the point.

Using return-codes for error handling either pollutes the code with checks or the error handling is not done right in the first place.

That's why

Example 1: Expression evaluation (expression parser from Introduction to programming), cf.

<http://codeboard.io/projects/46131>

Input: $1 + (3 * 6 / (/ 7))$

Error is deep in the recursion hierarchy. How to produce a meaningful error message (and continue execution)? Would have to pass error code over recursion steps.

Second Example

Value type with guarantee: values in range provided.

```
template <typename T, T min, T max>
class Range{
public:
    Range(){}
    Range (const T& v) : value (v) {
        if (value < min) throw Underflow ();
        if (value > max) throw Overflow ();
    }
    operator const T& () const {return value;}
private:
    T value;
};
```

Error handling in the constructor.

Types of Exceptions, Hierarchical

```
class RangeException {};  
class Overflow : public RangeException {};  
class Underflow : public RangeException {};  
class DivisionByZero: public RangeException {};  
class FormatError: public RangeException {};
```

Operators

```
template <typename T, T min, T max>
Range<T, min, max> operator/ (const Range<T, min, max>& a,
                             const Range<T, min, max>& b){
    if (b == 0) throw DivisionByZero();
    return T (a) * T(b);
}
```

```
template <typename T, T min, T max>
std::istream& operator >> (std::istream& is, Range<T, min, max>& a){
    T value;
    if (!(is >> value)) throw FormatError();
    a = value;
    return is;
}
```

Error handling in the operator.

Error handling (central)

```
Range<int, -10, 10> a, b, c;
try{
    std::cin >> a;
    std::cin >> b;
    std::cin >> c;
    a = a / b + 4 * (b - c);
    std::cout << a;
}
catch(FormatError& e){ std::cout << "Format error\n"; }
catch(Underflow& e){ std::cout << "Underflow\n"; }
catch(Overflow& e){ std::cout << "Overflow\n"; }
catch(DivisionByZero& e){ std::cout << "Divison By Zero\n"; }
```