

Exceptions

Introduction

Use *assert* during development process:

- to detect (and correct) bugs early
- to ease and speed up process

In *release* process

- a program should be robust
- so **handling errors is not the assert-way**
- so you write specific code for that

In C, error handling can be tricky but in C++, an **exception** is an object representing an error. A routine detecting an error *throws* an exception.

An exception is an object so you can define the error that happened.

Syntax

With error handling code in foo:

```
1 void foo()
2 {
3     try {
4         //
5         bar()
6         //
7     }
8     catch(...) {
9         //
10        throw;
11    }
12 }
```

the **catch** code block is run when an exception has been thrown

When error handling is not completed, the caught exception must be thrown again, with the instruction **throw**;

The **catch** clauses are inspected in the order in which they are listed, the appropriate **catch** clause is selected from the error type, running the corresponding code.

Hint : **FILE** => file and **LINE** => line, replaced at pre-processing

A “real” Class as an Exception

We can instrumentalize our error classes.

Misc

Procedural Lookup

my::sample is a module with method and procedures

```
1 namespace my {
2     class sample
3     {
4     public:
5         void meth();
6     }
7     std::ostream& operator<<(std::ostream& ostr, const sample& s);
8     void foo(sample& s);
9 } //end of my
10
11 void foo(int i);
```

```
1 my::sample s;
2
3 // these 3 lines are equivalent
4 std::cout << s << std::endl;
5 operator<<(std::cout, s) << std::endl;
6 my::operator<<(std::cout, s) << std::endl;
7
8 // these lines are equivalent
9 my::foo(s);
10 foo(s);
```

ADL: procedures are looked-up in the namespace(s) of their argument(s) !

Enum class vs enum

In C and C++, we can mix, due to weak typing, :

- int and bool
- enum values and int values
- pointers (auto p1 = new soccerplayer; auto p2 = (cowboy*)p1; //explicit cast)

Plain enum:

```
1 enum month { january, february, /*...*/ }
```

Class enum:

```
1 enum class month { january, february, /*...*/ }
```

In class enum, different variables of different enum types cannot be compared, and the code don't compile.

Optimizations

RVO = Return Value Optimization

NRVO = Named Return Value Optimization

NRVO and RVO are now guaranteed, there is no magic, the compiler just transform the code.

When the classical writing

```
1 return_type foo(args);
```

is better written

```
1 auto foo(args) -> return_type
```

We can write:

```
1 template<typename T1, typename T2>
2 auto foo(T1& t1, T2& t2) -> decltype(t1 + t2)
3 {
4     return t1 + t2;
5 }
```

Function object

We can use object like functions with operator overloading.

```
1  class Foo
2  {
3  public:
4      Foo(int i)
5          : i_(i)
6      { };
7      int operator()(int i) const { return i + i_; };
8
9  private:
10     int i_;
11 };
12
13 int foo(int i) { return -i; }
14
15 template<typename F>
16 int invoke(F f, int i) { return f(i); }
17
18 int main()
19 {
20     auto f = Foo{1};
21     std::cout << invoke(f, 2) << " " << invoke(foo, 2) << std::endl;
22 }
```

ouputs

```
1  3 -2
```

We can create a date with `date{day, month, year}`.

We have lambdas in C++.

Some Design Patterns

Adapter

Design pattern: an element of architecture, reusable solution to a common problem

Adapter: allows the interface of an existing class to be reused as the interface of an other class.

Chain of Responsibility

queue shall not feature operator[], neither append, *queue* shall feature push and pop, yet *queue* can rely on *array*

It is very classic to adapt the interface of an existing class.

Notes

```
1  template<unsigned n>
2  struct fact {
3      enum { ret = n * fact<n-1>::ret }
4  };
5
6  template<>
7  struct fact<1> {
8      enum { ret = 1 }
9  };
10
11 template<>
12 struct fact<0>;
13
14 int main()
15 {
16     unsigned five = 5;
17     std::cout << fact<5> << std::endl; //Works
18 }
19 int main()
20 {
21     const unsigned five = 5;
22     std::cout << fact<five> << std::endl; //Works
23 }
24 int main()
25 {
26     unsigned five = 5;
27     std::cout << fact<five> << std::endl; //Don't compile
28 }
```