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 *throw*s 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

```
namespace my {
  class sample
  {
  public:
    void meth();
  }
  std::ostream& operator<<(std:ostream& ostr, const sample& s);
  void foo(sample& s);
}//end of my

void foo(int i);
```

```
my::sample s;

// these 3 lines are equivalent
std::cout << s << std::endl;
operator<<(std::cout, s) << std::endl;
my::operator<<(std::cout, s) << std::endl;

// these lines are equivalent
my::foo(s);
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:

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

Class enum:

```
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

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

is better written

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

We can write:

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

Function object

We can use object like functions with operator overloading.

```cpp
class Foo
{
public:
  Foo(int i)
    : i_(i)
  { };
  int operator()(int i) const { return i + i_; };

private:
  int i_;
};

int foo(int i) { return -i; }

template<typename F>
int invoke(F f, int i) { return f(i); }

int main()
{
  auto f = Foo{1};
  std::cout << invoke(f, 2) << " " << invoke(foo, 2) << std::endl;
}
```

ouputs

```
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

```
template<unsigned n>
struct fact {
  enum { ret = n * fact<n-1>::ret }
};

template<>
struct fact<1> {
  enum { ret = 1 }
};

template<>
struct fact<0>;

int main()
{
  unsigned five = 5;
  std::cout << fact<5> << std::endl; //Works
}
int main()
{
  const unsigned five = 5;
  std::cout << fact<five> << std::endl; //Works
}
int main()
{
  unsigned five = 5;
  std::cout << fact<five> << std::endl; //Don't compile
}
```