C/C++: Lecture 7

Vorobev D.V

16.10.2020

Dynamic exception specification

Dynamic exception specification

Dynamic exception specification

It is a syntax to list a set of types that could be throw by function.
It was deprecated in C++11 and removed in C++17,C++20.

```
void foo() throw(// Here we list a set of types) {
    ...
}
```

Dynamic exception specification

Type **from** the list

```
class MyException {
    // implementation
};

void foo() throw(MyException) {
    // ok
    throw MyException();
}
```

Type **is not from** the list

```
class MyException {
    // implementation
};

void foo() throw(MyException) {
    // std::unexpected
    throw 1;
}
```

std::unexpected

Quick facts

- It is called when the type not specified in the list was thrown
- It calls std::unexpected_handler
- The default std::unexpected_handler calls std::terminate

```
class MyException {
    // implementation
};

void foo() throw(MyException) {
    // std::unexpected
    throw 1;
}
```

Custom handler

```
void handler() {
  std::cout << 1;
}

void foo() throw (double) {
  throw 1;
}

int main() {
  std::set_unexpected(handler);
  // 1
  foo();
  return 0;
}
```

noexcept operator and specifier

noexcept specifier

Note

noexcept does not guarantee that a function does not throw exceptions

```
// There is no CE
void bar() noexcept {
    throw 1;
}

int main() {
    bar();
}
```

noexcept specifier

Quick facts

- **noexcept** gives an additional knowledge to a compiler to perform optimizations
- an absence of exceptions on the conscience of a developer
- **noexcept** is your **promise** to the user of your function

noexcept specifier

We cannot overload functions that differ only in an exception specification

```
// CE
void foo() noexcept;

void foo();
```

noexcept specifier

Weaken non-throwing guarantee by overrides is not allowed.

```
struct Base {
    virtual void foo() noexcept;
};

struct Derived: Base {
    void foo(); // CE
};
```

noexcept specifier

Weaken non-throwing guarantee by overrides is not allowed.

```
struct Base {
    virtual void foo();
};

struct Derived: Base {
    void foo() noexcept; // ok
};
```

noexcept operator

```
void foo();
void bar() noexcept;
struct X {
      ~X(){}
};

int main() {
    std::cout << noexcept(foo()) << std::endl;
    std::cout << noexcept(bar()) << std::endl;
    std::cout << noexcept(std::declval<X>().~X());
}
```

specifier + operator

```
void bar() {
    throw 1;
}

void foo() noexcept( noexcept(bar(c)) ) {}
```

Exception that leaves constructor

```
void foo() { throw 1; }

struct MyClass {
    int* x;

    MyClass() {
        x = new int(1);
        foo();
    }

    ~MyClass() { delete x; }
};

int main() {
// Destructor is not called => memory leak
    MyClass a;
}
```

Manual resource release is error-prone

Problem: ptr is not deleted

```
void foo() { throw 1; }

void Action() {
    int* ptr = new int(1);
    // some actions
    // some actions
    foo(); // <- exception
    // some actions
    // some actions
    delete ptr;
}
```

Manual resource release is error-prone

Solution: RAII idiom
(CoreGuidelines E.6: Use RAII to prevent leaks)

```
void foo() {
    throw 1;
}

void Action() {
    std::shared_ptr<int> ptr(new int(1));

// 1. exception
// 2. shared_ptr's destructor
// 3. resource release
    foo();
}
```

Exception that leaves destructor

Never throw an exception inside the destructor

Reason

Stack unwinding $\Rightarrow$ calling the destructor of previously created objects $\Rightarrow$ throwing an exception $\Rightarrow$ 2 unhandled exceptions $\Rightarrow$ abort

Exception that leaves destructor

```
struct X {
    std::string name_;

    X(std::string name)  :
        name(std::move(name)) {
        std::cout << "X" << name_;
    }

    // we mark destructor
    // noexcept(false)
    // on purpose
    ~X() noexcept(false) {
        std::cout << "~X:" << name_;
        throw 1;
    }
};
```

```
void bar() {
    X a("a");
// 1. destructor of "a"
// 2. exception
// 3. stack unwinding
// 4. destructor of "b"
// 5. 2 uncaught exceptions
// 6. terminate
}
void foo() {
    X b("b");
    bar();
}
int main() {
    foo();
    return 0;
}
```

Since C++11 destructor is marked as noexcept(true)

```
struct X {
    ~X(){}
};

int main() {
    // 1
    std::cout << noexcept(std::declval<X>().~X());
}
```

std::terminate

It is called in the following cases:

1. an exception is thrown and not caught
2. dynamic exception specification is violated
3. noexcept specification is violated

std::terminate : uncaught exception

```
class MyException {
    // implementation
};

void foo() {
    throw MyException();
}

int main() {
    foo(); // <- uncaught exception
    return 0;
}
```

std::terminate : dynamic exception specification

```
 class MyException {
    // implementation
};
void foo() throw() {
    // 1. foo is non-throwing
    // 2. foo throws MyException
    // 3. std::terminate
    throw MyException();
}

int main() {
    foo();
    return 0;
}
```

std::terminate : dynamic exception specification

```
class MyException {
    // implementation
};

void foo() noexcept {
    // 1. foo is non-throwing
    // 2. foo throws MyException
    // 3. std::terminate
    throw MyException();
}

int main() {
    foo();
    return 0;
}
```

std::set_terminate

```
// before C++11
std::terminate_handler set_terminate(std::terminate_handler f) throw();

// since C++11
std::terminate_handler set_terminate(std::terminate_handler f) noexcept;
```