4.15 Исключения. Преимущества и недостатки использования исключений. Особенности копирования исключений при их создании и при поимке. Разница между throw без параметров и throw с параметром. Правила приведения типов при поимке исключений. Правила выбора подходящей секции catch

Преимущества и недостатки

- ✓ Исключения отделяют код обработки ошибок от нормального алгоритма программы, тем самым повышая разборчивость, надежность и расширяемость кода.
- ✓ Исключения легко передаются из глубоко вложенных функций.
- ✓ Генерация исключения – единственный чистый способ сообщить об ошибке из конструктора
- ✓ Исключения могут быть, и часто являются, определяемыми пользователем типами, несущими гораздо больше информации, чем код ошибки.
- × Исключения нарушают структуру кода, создавая множество скрытых точек выхода, что затрудняет чтение и изучение кода.
- × Исключения тяжело ввести в устаревший код.
- × Исключения неверно используются для выполнения задач, относящихся к нормальному алгоритму программы.

Особенности копирования исключений приих создании и при поимке

Рассмотрим вот такой код:

```
struct Noisy {
    Noisy() {
        std::cout << "created";
    }
    Noisy(const Noisy&) {
        std::cout << "copy";
    }
    ~Noisy() {
        std::cout << "destroyed";
    }
};

void f() {
    Noisy x;

    throw x;
}


int main() {
    try {
        f();
    } catch (const Noisy& x) {
        std::cout << "caught";
    }
}
```

Что тут произойдет при запуске? Обратим внимание на функцию f. В ней создается локальный объект x, который нам предстоит бросить. Но при выходе из функции все локальные объекты уничтожаются, поэтому мы бросаем не x, а делаем копию x, а потом уже бросаем ее. Таким

образом, программа выведет: "created copy destroyed caught destroyed"
Если мы бы принимали не по константной ссылке, а просто по значению, создалась бы еще одна копия после того, как мы скопировали объект для того чтобы бросить

А без конструктора копирования не получится бросить :(

Разница между throw без параметров и throw с параметром.

throw с параметром используется для того, чтобы бросить какой-то конкретный объект, а throw без параметров используется только внутри catch, если необходимо пробросить уже пойманное исключение наверх.
Существенное отличие заключается в том, что **throw без параметров не создает копию**
Рассмотрим пример

```
void g() {
    try {
        throw std::out_of_range("aaa");
    } catch (std::exception& ex) {
        std::cout << "caught";
        throw;
    }
}

int main() {
    try {
        g();
    } catch (std::out_of_range& oor) {

    }
```

Сейчас исключение поймается нормально, потому что в main прилетело std::out_of_range

Но если заметить throw на throw ex, то мы уже в main его не поймаем.

```
void g() {
    try {
        throw std::out_of_range("aaa");
    } catch (std::exception& ex) {
        std::cout << "caught";
        throw ex;
    }
}

int main() {
    try {
        g();
    } catch (std::out_of_range& oor) {
        std::cout << "caught2";
    }
```

std::out_of_range уничтожилось, и дальше полетело то, что мы поймали, но поймали мы родителя. Тем самым мы создали копию родителя, и дальше полетел родитель, а не std::out_of_range

Правила при поимке исключений

1. В catch не работает приведение типов за исключением ситуаций, когда это родитель и наследник. Поймать наследника по ссылке или копии на родителя можно. А вот такой код не поймает ошибку:

```
int main() {
    try {
        throw 1;
    } catch (unsigned int x) {

    }
```

2. Работает приведение типов между const и не const
3. Если у нас есть несколько catch, то компилятор выбирает первый подходящий

```
try {
    throw std::out_of_range("aaaa");
} catch (std::exception& ex) {
    std::cout << 1;
} catch (std::out_of_range& oor) {
    std::cout << 2;
} catch (...) {
    std::cout << 3;
}
```

Здесь выведется 1

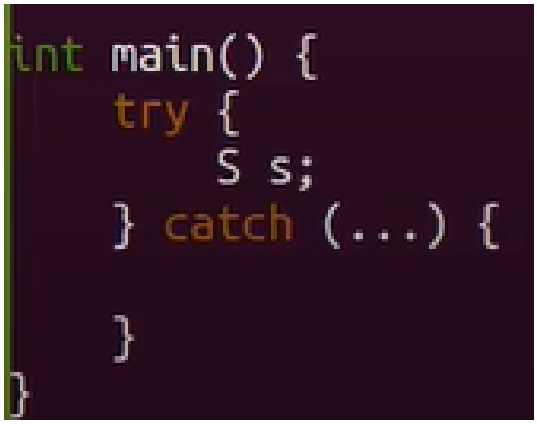
4.16 Проблема исключений в конструкторах. Поведение программы при выбросе исключения из конструктора. Идиома RAII. Проблема исключений в деструкторах. Функция uncaught_exception, ее предназначение. Спецификации исключений: спецификатор noexcept и оператор noexcept, синтаксис и пример применения. Исключения в списках инициализации конструкторов, function-try блоки

Проблема исключений в конструкторах.Поведение программы при выбросе исключения из конструктора. Идиома RAII.

Пусть у нас есть вот такой класс

```
struct S {
    int* p = nullptr;

    S(): p(new int (5))
        throw 1;
    }

    ~S() {
        delete p;
    }
};
```

Хотим выполнить такой код



Произойдет утечка памяти. Вопрос. Должен ли вызваться деструктор, если исключение вылетело из конструктора? Если исключение вылетает из конструктора, объект еще не до конца создан, и значит, комплятор не может вызать деструктор для него. Как решать такую проблему?

Идиома **RAII - Resourse Acquisition Is Inizialization** - захват ресурса есть инициализация некоторого объекта. Всякий раз, когда нужно захватить какой-то ресурс, это делаем не в лоб, а с помощью объекта, который явно это делает. Тогда каждый раз когда будет выбрасываться исключение, деструкторы локальных объектов гарантированно вызовутся и, следовательно, ресурсы будут освобождены вовремя.

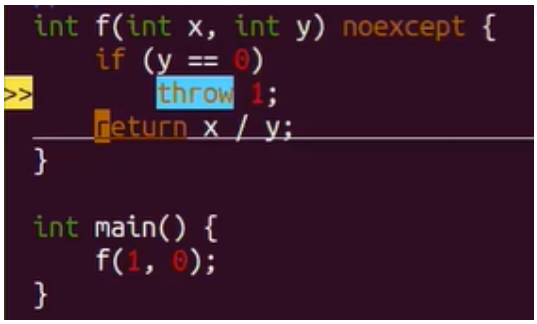
```
template <typename T>
class SmartPtr {
private:
    T* ptr;
public:
    SmartPtr(T* ptr): ptr(ptr) {}
    ~SmartPtr() {
        delete ptr;
    }
};


struct S {
    SmartPtr<int> p = nullptr;

    S(): p(new int (5)) {
        throw 1;
    }

    ~S() {}
};


void f() {
    SmartPtr<int> p = new int(5);

    throw 1;
}
```

Мы обернули указаль в класс SmartPtr, который создается в теле функции. Поэтому, будучи локальным объектом, он удалится в случае выхода из функции и утечки памяти не произойдет

Спецификации исключений: спецификатор noexcept и оператор noexcept, синтаксис и пример применения.

Подобно тому, как мы пишем, что функция может работать с константными объектами (используя ключевое слово const), можно также помечать, безопасна ли функция с точки зрения исключений с помощью слова "noexcept".

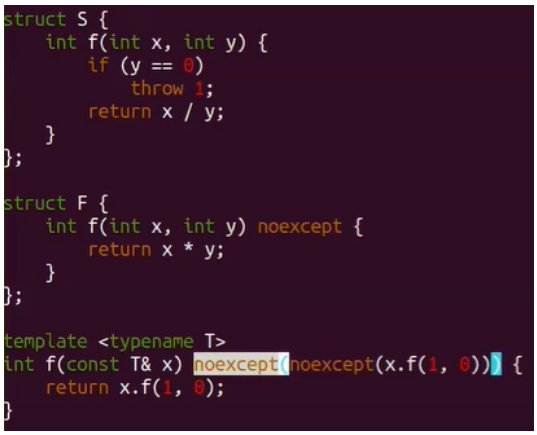
Если все-таки запихнуть throw в noexcept функцию, будет RE



Спецификатор noexcept

Теперь посмотрим вот на такой код(картинка ниже). Здесь есть функция f, которая бывает то не noexcept, то noexcept в зависимости от шаблонного параметра. По хорошему, эту функцию нужно бы либо пометить, либо не пометить noexcept в зависимости от того, является ли noexcept вызов внутри функции f. Для этого существует условный noexcept.

На картинке ниже выделенный noexcept и невыделенный рядом с ним - это разные по смыслу noexcept. Выделенный является ключевым словом, невыделенный - оператором.

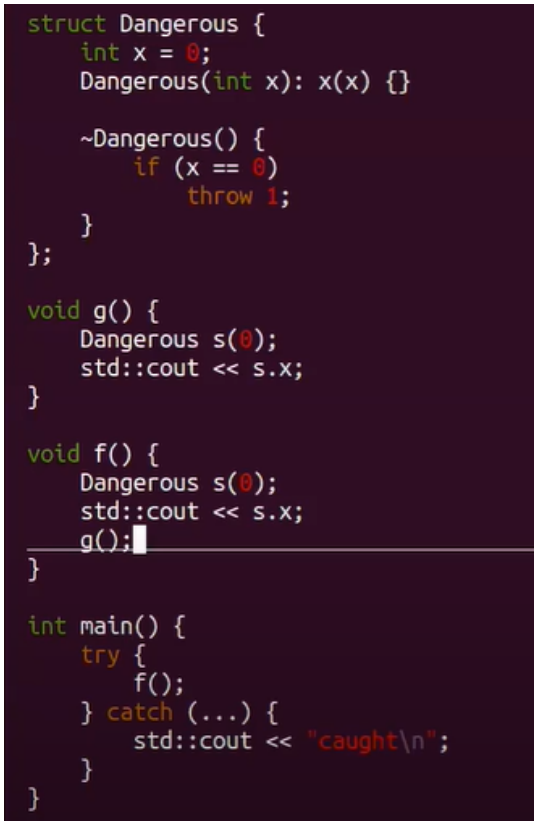


Оператор noexcept(...) возвращает true, если выражение в скобках безопасно относительно исключений и false в ином случае.

Замечание Аналогично оператору sizeof noexcept не вычисляет значение внутри него, а просто осуществляет проверку на наличие выражений, которые потенциально могут бросить исключение. Среди них:

- throw
- new
- dynamic_cast
- noexcept function call

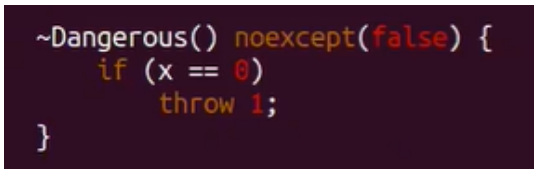
Проблема исключений в деструкторах.



Давайте проанализируем, что будет, если мы запустим этот код. Вот мы вызвали f, она создала опасный объект s, вызвала g, которая создала опасный объект s. При завершении g уничтожаются все локальные объекты, в том числе и s. Деструктор Dangerous кидает исключение, которое завершает работу g, и мы вылетаем в f вместе с этим исключением. f тоже должна завершиться, удаляются все локальные объекты, мы переходим в деструктор Dangerous... и ловим еще одно исключение! В момент, когда исключение уже летело. И программа падает
Что в этом случае делать?

Не бросать исключения из деструкторов

По умолчанию, начиная с C++11 все деструкторы считаются noexcept функциями
Если очень надо бросить исключение из деструктора, можно сделать так (но не нужно):



Исключения в списках инициализации конструкторов, function-try блоки

Пусть у нас есть какой-то класс, у которого мы инициализируем поля списком инициализации.
И внезапно один из вызовов в списке инициализации бросает исключение

```
class A {
public:
  A(int n) {
    throw 0;   // Конструктор класса A бросает исключение int
  }
};

class B {
public:
  B(int n);
private:
  A __a;
};

B::B(int n)
  : __a(n)     // Данный вызов бросает исключение
{}
```

Для решения этой проблемы используют try-catch блоки

Function-try block

Когда нам нужно обернуть всё тело функции в try, то можно использовать спецификатор **try**:

```
1     void f() try{
2     } catch() {}
```

Тогда наш код со списком инициализации переписывается следующим образом:

```
class B {
public:
  B();
private:
  P* __p;
  A __a;
};

B::B()
try
  : __p(new P), __a(0) {
} catch (int& e) {
  std::cout << "B(), exception " << e << std::endl;
  delete __p;
};
```

Функция uncaught_exception, ее предназначение.

https://en.cppreference.com/w/cpp/error/uncaught_exception

1 Определяет, есть ли у текущего потока активный объект исключения. Было исключение брошено, брошено лететь дальше или еще не поймано, std::terminate or std::unexpected. Другими словами, std :: uncaught _exception определяет, выполняется ли в настоящий момент раскрутка стека.

2 Отслеживает, сколько ошибок было брошено/летит в текущем потоке или еще не поймано