Исключения I

Обработка ошибок

Рассмотрим функцию, у которой в какой-то момент может "что-то пойти не так".

```
template <class T>
T Divide(T x, T y) {
  if (y == 0) { /* ... */ }
  return x / y;
}
```

Как эта функция должна сообщить пользователю об ошибке?

Обработка ошибок

1. Никак

```
template <class T>
T Divide(T x, T y) {
  if (y == 0) {}
  return x / y;
}
```

Очевидно, не всегда хороший способ (но многие пользуются).

Обработка ошибок

2. Вернуть специальное значение.

```
T Divide(T x, T y) {
  if (y == 0) { return kSpecialValue<T>; }
  return x / y;
}
```

Но как отличить специальное значение от результата? Что, например, вернуть при делении `int`'ов?

Обработка ошибок

3. Вернуть код ошибки, а значение записать в выходной аргумент функции.

```
template <class T>
error_t Divide(T x, T y, T* out) {
  if (y == 0) { return 1; }  // код ошибки - 1
  *out = x / y;
  return 0;  // нет ошибки
}
```

Насколько это удобно?

Обработка ошибок

4. Записывать код ошибки в специальную переменную.

```
template <class T>
T Divide(T x, T y) {
  if (y == 0) { errno = 1; }  // код ошибки - 1
  errno = 0;  // нет ошибки
  return x / y;
}
```

Получаем весь пакет проблем, связанных с глобальными переменными.

Обработка ошибок в C

В языке C чаще всего используется способ с возвратом кода ошибки.

```
error_t f() {
  int err = g();
  if (err) { /* do something */ return err;}
  err = h();
  if (err) { /* do something */ return err;}
  return 0;
}
```

- Код раздувается из-за большого числа проверок и ветвлений.
- Для получения результата необходимо передавать указатель/ссылку.
- Легко проигнорировать возвращаемое значение.

Обработка ошибок в C++

В C++ используется механизм исключений.

Исключение - объект, который генерируется при возникновении исключительной ситуации (ошибки). По идее, должен содержать необходимую информацию о природе ошибки.

Генерация ошибки происходит с помощью оператора `throw`:

```
template <class T>
T Divide(T x, T y) {
  if (y == kZero<T>) { throw 1; }  // Исключение типа int, со значением 1
  return x / y;
}
```

Оператор `throw`

После выполнения `throw` работа функции прекращается, для объектов на стеке вызываются деструкторы. Та же участь постигает остальные функции (раскручивание стека), то есть всю последовательность вызовов, приведшую к ошибке.

```
void Cancer()  { throw 1;   std::cout << "Cancer()::here\n"; }
void Smoking() { Cancer();  std::cout << "Smoking()::here\n"; }
int main()     { Smoking(); std::cout << "main()::here\n"; return 0; }
```

В данном примере `main` вызывает `Smoking`, `Smoking` вызывает `Cancer`, `Cancer` генерирует исключение `1` и тут же завершает работу. `Smoking` не обрабатывает это исключение, поэтому тоже завершает работу. Аналогично погибает `main`.

```
terminate called after throwing an instance of 'int'
```

Оператор `throw`

Подробнее про

```
throw <obj>;
```

Объект исключения хранится в специальном месте памяти и создается путем копирования или перемещения переданного объекта.

```
A a;
throw a;  // Создается исключение A как копия оъекта a
throw std::move(a);  // Объект исключения создается с помощью перемещения
```

При бросании *prvalue* происходит copy elision (C++17):

```
throw A();  // вызывается конструктор по умолчанию, без copy или move
```

`try-catch` блок

Как обработать ошибку?

Можно ли как-то перехватить ошибку и отложить (возможно, отменить) падение программы?

Да, для этого нужно воспользоваться конструкцией `try-catch`

```
template <class T>
T Divide(T x, T y) {
  if (y == 0) { throw 1; }
  return x / y;
}

int main() {
  try {
    Divide(1, 0);
  } catch (int err) {
    std::cout << "DivisionError: error code " << err << '\n';
  }
  return 0;
}
```

Как обработать ошибку?

```
int main() {
  try {
    Divide(1, 0);
    std::cout << "Java is better than C++\n";  // <- это не выведется
  } catch (int err) {
    std::cout << "DivisionError: error code " << err << '\n';
  }
  return 0;
}
```

Возникающее в блоке `try` исключение может быть поймано в `catch` блоке соответствующего типа.

По завершении блока `catch` исключение считается успешно обработанным и выполнение программы продолжается в нормальном режиме.

```
DivisionError: error code 1
```

`catch`

Одному блоку `try` может соответствовать несколько блоков `catch`

```
int main() {
  try {
    Divide(1, 0);  // бросает int
  } catch (double err) {
    std::cout << "double\n";
  } catch (int err) {
    std::cout << "int\n";
  }
  return 0;
}
```

В этом случае отработает только 1 блок, соответствующий типу брошенного исключения

```
int
```

`catch`

Если нужный блок `catch` не найден, то исключение поймано не будет. Вызов функции завершится и исключение полетит дальше.

```
int main() {
  try {
    Divide(1, 0);  // бросает int
  } catch (double err) {
    std::cout << "double\n";
  } catch (const char* err) {
    std::cout << "const char*\n";
  }
  return 0;
}
```

Исключение не обработано:

```
terminate called after throwing an instance of 'int'
```

Выбор блока `catch`

Блок `catch` выбирается только по *точному соответствию*. То есть приведений типов НЕ происходит.

```
int main() {
  try {
    Divide(1, 0);  // бросает int
  } catch (unsigned int err) {
    std::cout << "unsigned int\n";
  } catch (char err) {
    std::cout << "char\n";
  }
  return 0;
}
```

Исключение не обработано:

```
terminate called after throwing an instance of 'int'
```

Выбор блока `catch`

Из предыдущего правила есть 2 исключения (ну вы поняли, в каком смысле) :

1. `void*` может поймать любой указатель.

```
void f() {
  int x;
  try { throw &x; }
  catch (void* ptr) { std::cout << "catched\n"; }
}
```

2. Приведения по иерархии наследования вверх (к родителям) работают.

```
class B : public A {};

void f() {
  try { throw B(); }
  catch (A a) { std::cout << "catched\n"; }
}
```

Выбор блока `catch`

Срабатывает всегда первый подходящий `catch`.

```
void f() {
  int x;
  try { throw &x; }
  catch (void*) { std::cout << "catched\n"; }  // будет поймано здесь
  catch (int*)  { std::cout << "no way\n"; }   // этот блок будет проигнорирован
}
```

```
class B : public A {};

void f() {
  try { throw B(); }
  catch (A) { std::cout << "catched\n"; }  // будет поймано здесь
  catch (B)  { std::cout << "no way\n"; }  // этот блок будет проигнорирован
}
```

Ловля исключений по ссылке

Если ловить исключение по значению, то исходное исключение скопируется.

Чтобы избежать копирования можно ловить исключения по ссылке (или по константной ссылке).

```
void f() {
  try { throw std::vector<int>(1'000'000); }
  catch (const std::vector<int>& v) {}
}
```

Более того, при ловле по ссылке становится доступно полиморфное поведение (позже рассмотрим, как это помогает).

`catch(...)`

Существует особый синтаксис `catch`, позволяющий поймать любое исключение.

```
int main() {
  try {
    f();
    g();
  } catch (...) {
    std::cout << "catched\n";
  }
}
```

Что бы и откуда бы (из `f()` или `g()`) не было брошено, оно будет поймано в `catch(...)`.

Но в таком блоке нельзя определить тип исключения и поработать с объектом исключения.

Что делать в `catch` блоке?

Не нужно ловить исключения просто для того, чтобы поймать.

Если непонятно как решить проблему, лучше не трогать его (пусть летит дальше).

Но иногда необходимо перехватить исключение, чтобы, например, освободить выделенные ресурсы (естественно, при раскручивании стека компилятор не догадается вызвать `delete`):

```
void f() {
  auto ptr = new int(11);
  try {
    g(ptr);  // потенциально бросает исключение типа A
  } catch (A& a) {
    delete ptr;
    ptr = nullptr;
  }
  // ...
  delete ptr;
}
```

Что делать в `catch` блоке?

А что, если хочется освободить ресурсы и бросить исключение лететь дальше (например, мы не знаем, как решить проблему на нашем уровне)?

```
void f() {
  auto ptr = new int(11);
  try {
    g(ptr);  // потенциально бросает исключение типа A
  } catch (A& a) {
    delete ptr;
    throw a;  // бросаем исключение снова
  }
  // ...
  delete ptr;
}
```

Но:

1. `throw a;` создает новое исключение (копирование)
2. Непонятно, что написать в блоке `catch(...)`

`throw;`

Существует особая форма оператора `throw` без аргумента - `throw;`.

Она дословно означает: "снова бросить пойманное исключение". При этом копии исключения не создается - будет "лететь" тот же объект, что и раньше.

```
void f() {
  auto ptr = new int(11);
  try {
    g(ptr);  // потенциально бросает исключение типа A
  } catch (...) {
    delete ptr;
    throw;  // бросаем старое исключение
  }
    // ...
  delete ptr;
}
```

RAII и исключения

Главный довод в пользу использования RAII классов (`std::vector`, `std::unique_ptr` и т.д.)

```
void f() {
    auto ptr = new int(11);
    try {
        g(ptr);  // потенциально бросает исключение типа A
    } catch (...) {
        delete ptr;
        throw;  // бросаем старое исключение
    }
    // ...
    delete ptr;
}
```

```
void f() {
  auto ptr = std::make_unique<int>(11);  // память освободится автоматически!
  g(ptr.get());  // потенциально бросает исключение типа A
  // ...
}
```

Статическая спецификация исключений (C++11)

Спецификатор `noexcept` (C++11)

Чтобы пообещать компилятору, что функция не будет бросать исключений можно воспользоваться спецификатором `noexcept`

```
void f() noexcept {
  // ...
}
```

Если обещание будет нарушено, и исключение вылетит из `noexcept` функции, то программа завершится аварийно (без возможности перехватить исключение)

```
void f() noexcept {
  g();  // если из g() вылетит исключение, то - хана
}
```

Спецификатор `noexcept` (C++11)

То есть в `noexcept` функциях нужно использовать только `noexcept` операции, либо перехватывать все исключения (так себе вариант):

```
void f() noexcept {
  try {
    g();  // потенциально бросает исключения
  } catch (...) {
    // решаем проблему
  }
}
```

Компиляторы не выдают предупреждения при небезопасных вызовах в `noexcept` функциях, максимум - замечание.

Условный спецификатор `noexcept` (C++11)

```
void f() noexcept;
// <=>
void f() noexcept(true);
```

```
void g();
// <=>
void g() noexcept(false);
```

Это может быть использовано для маркирования функций, спецификатор которых зависит от некоторого условия (проверяемого на этапе компиляции):

```
template <class T>
void h(T x) noexcept(sizeof(T) > 1);
```

Операция `noexcept`

Ключевое слово `noexcept` еще служит для обозначения операции, которая определяет является ли выражение `noexcept` или нет:

```
void f() noexcept;
void g();
noexcept(f());  // true
noexcept(g());  // false
```

```
std::vector<int> v;
noexcept(1 / 0);  // true - деление целых чисел не генерирует исключений!
noexcept(v.push_back(1 / 0));  // false - push_back может бросить out_of_memory
```

Важно понимать, что `noexcept` (ровно как и `sizeof`) **не вычисляет** результат выражения, а просто анализирует его на предмет возможности/невозможности получить исключение.

Финальный пример

Комбинируя условный спецификатор `noexcept` и операцию `noexcept`, можно устанавливать `noexcept`, зависящий от того, являются ли вызываемые внутри функции `noexcept` или нет.

```
template <class T>
auto Sum(const T& x, const T& y) noexcept(noexcept(x + y)) {
  return x + y;
}
```

В данном примере, для `T=int` `Sum` будет `noexcept`, а, например, для `T=std::string` не будет (выделение памяти для строки может завершиться неудачно).

Небросающий `new`

Использование `new` небезопасно с точки зрения возможных исключений - если недостаточно памяти, то вылетает исключение типа `std::out_of_memory`

```
auto ptr = new int[1'000'000];  // потенциально может бросить исключение
```

Это поведение можно изменить - можно попросить вместо исключения возвращать `nullptr` с помощью следующего синтаксиса:

```
auto ptr = new(std::nothrow) int[1'000'000];
if (!ptr) { /* памяти не хватило */ }
```

Это бывает полезно, если не хочется возиться с обработкой исключений (писать `try-catch`)