Exceptions

Варианты обработок ошибок в C:

1. Код возврата
2. В указатель (аргумент)
3. В `errno`

Недостатки:

- Невозможность обрабатывать ошибки в конструкторах
- Константы ошибок лежат в глобальной памяти, что не очень хорошо при большом числе их видов; мы должны будем как минимум хранить их в разных namespace.
- Перегрузка дружественных функций-операторов (бинарных)
  Можно проблему с конструкторами решить через состояние класса с ошибкой. Но это не очень красиво, ибо ошибку нужно будет проверять в нескольких местах: деструкторе и main как минимум.

Использовать assert - тоже плохая идея, ибо в `cmake build type = RELEASE` определен `-DNDEBUG` который отключает его (макрос).

Локальные и нелокальные переходы

Локальный переход - тот, который соответствует логике программы (логике if/else/call/return)
Нелокальный переход - тот, при котором происходит передача управления на стороние исполняемые участки (переключение на контекст инспектора/исключения C++/brk)

Почему goto, setjmp, longjmp плохо

Они не зависят от контекста, приводят к наличию внешних дуг, превращающих дерево алгоритма в произвольных граф. **На практике это может, например, привести к пропуску вызова деструктора/конструктора.**

Логика и ипримененение исключений C++

Исключения используются там, где:

- Ошибка может быть восстановлена
- Там, где ошибка трудна обратываема в текущем стэковом фрейме (и требуется раскрутка стека)
- Там, где вероятность ошибки низка; частый выброс исключений хоронит производительность. А так платим лишь код сайзом.

Для исключений нужен стартовый стэковый фрейм (`try {}`) - последнее место, до которого нужно ловить брошенное исключение (если в нем не поймаем - получим аборт), **в нем же** блоки ловушек `catch (Exception_class& obj) {}` (лучше кидать класс ошибок и ловить его в виде lvalue). Бросать исключения `throw exception_class{};` можно только в следующих за данным стековым фреймам (т.е. в вызовах будущих функций).

При ловле исключений запрещены даже самые простые преобразования типов, поэтому, например:

```
int foo() {
    throw 1;
}
/* ... */
try {
    foo();
}
catch(long) { /* ... */ }
```

Не поймает исключение

Однако в исключениях разрешена деградация классов:

```
try
{
    throw new Derived{};
}
catch (Base* obj) { /* ... */ } // ok
```

```
try
{
    throw Derived{};
}
catch (Base& obj) { /* ... */ } // ok
```

> Лучше бросать и ловить исключения классов или наследников `std::esception-->std::runtime_error`.

Гарантии безопасности исключений

> `new` бросает исключение `std::bad_alloc`

```
template <typename T> class MyVector {
    T *arr_ = nullptr;
        size_t size_{0}, used_{0};

public:
    MyVector(const MyVector &r) {
        arr_ = new T[r.size_]; // 1
        size_ = r.size_; used_ = r.used;
        for (size_t i = 0; i != r.size_; ++i) {
            arr_[i] = r.arr_[i]; // 2 (throw on 6th)
        }
    }
```

Здесь копия MyVector начнет жизнь после того, как отработает `}` конструктора копирования. Пусть если `new` выбросит исключение, то он был внутри `try` блока. Пусть он отработал нормально, но что насчет оператора присваивания? Если он выбрасывает исключение на шестом элементе, и оно обрабатывается, то что тогда? Даже при ловле исключения, но без явного вызова чистки в нем, **деструктор сам не вызовется**.

Это можно исправить так:

```
template <typename T> class MyVector {
```

```
    T *arr_ = nullptr;
        size_t size_{0}, used_{0};

public:
    MyVector(const MyVector &r) {
        arr_ = new T[r.size_]; // 1
        size_ = r.size_; used_ = r.used;
        try {
            for (size_t i = 0; i != r.size_; ++i) {
                arr_[i] = r.arr_[i]; // 2 (throw on 6th)
            }
        }
        catch (...) { // лучше не использовать, но здесь оправдано
            delete[];
            throw;
        }
    }
```

Линия Калба

Идея в том, что в коде любого метода проектировать код так, чтобы его можно было разделить чертой на две части:
* Выше линии: логика, бросающая исключения
* Ниже линии: логика, изменяющее внутреннее состояние объекта/функции (например, выделение/отчистка ресурсов)

> Неявные преобразования лучше блокировать словом `explicit` в тех методах, где, например, `int` служит аргументом размера выделяемой памяти.

Гарантии безопасности исключений

Базовая гарантия

Исключение при выполнении операции может изменить состояние программы, но не вызывает утечек памяти и оставляет все объекты в согласованном (но необязательно предсказуемом состоянии)

Строгая гарантия

При исключении гарантируется неизменность состояния программы относительно задействованных в операции объектов (commit/rollback)

Гарантия бесбойности

Функция не генерирует исключений