ОБРАБОТКА ОШИБОК C-STYLE

так вот зачем в subvector везде возвращался bool

Коды возврата

```
// Считаем, что 0 - код успешного завершения
int func1()
{
    // Тут происходит какая-то работа

    // Делаем вид, что всё сломалось
    return -1;
}
```

```
int main()
{
    // Вызов функции
    int res = func1();
    // Обработка возможной проблемы
    if(res != 0) {
        cerr << "Calling func1() failed!" << endl;
        return -1;
    }

    // Тут идёт какая-то работа дальше
```

В C++ можно возвращать bool, но это менее информативно.

По коду ошибки можно понять, какая именно ошибка произошла.

Коды возврата

- Нужно помнить и соблюдать коды ошибок.
- Из функции ничего больше не вернешь
  – надо возиться с указателями.
- В реальном коде занимает много места
  – снижается читаемость;
  – легко упустить какой-то частный случай.
- Промежуточные функции тоже должны помнить и соблюдать коды.

Коды возврата

Some list of sysexits on both Linux and BSD/OS X with preferable exit codes for programs (64-78) can be found in `/usr/include/sysexits.h` (or: `man sysexits` on BSD):

```
0   /* successful termination */
64  /* base value for error messages */
64  /* command line usage error */
65  /* data format error */
66  /* cannot open input */
67  /* addressee unknown */
68  /* host name unknown */
69  /* service unavailable */
70  /* internal software error */
71  /* system error (e.g., can't fork) */
72  /* critical OS file missing */
73  /* can't create (user) output file */
74  /* input/output error */
75  /* temp failure; user is invited to retry */
76  /* remote error in protocol */
77  /* permission denied */
78  /* configuration error */
/* maximum listed value */
```

Коды возврата

```
int func1()
{
    // Тут происходит какая-то работа

    // Делаем вид, что всё сломалось
    return -1;

    // Дальше какая-то работа

    return 0;
}
```

```
int func2()
{
    // Что-то делаем

    // Вызов func1 и обработка возможной ошибки
    int f1_res = func1();
    if(f1_res != 0) {
        return f1_res;
    }

    // Что-то делаем дальше

    return 0;
}
```

```
int func3()
{
    // Что-то делаем

    // Вызов func2 и обработка возможной ошибки
    int f2_res = func2();
    if(f2_res != 0) {
        return f2_res;
    }

    // Что-то делаем дальше

    return 0;
}
```

```
int main()
{
    // Вызов func3 и обработка возможной ошибки
    int res = func3();
    if(res != 0) {
        cerr << "Calling func3() failed!" << endl;
        return -1;
    }

    // Тут идёт какая-то работа дальше
```

ИСКЛЮЧЕНИЯ

эксепшоны

Что такое exception

Exception – логический «сигнал»,
который можно сгенерировать («бросить»)
в одном произвольном месте кода
и обработать («поймать»)
в произвольном другом месте,
находящемся выше по цепочке вызовов

Как выглядит exception

```cpp
// Функция теперь void, ей не нужно возвращать статус
void func1()
{
    // Тут происходит какая-то работа

    // Делаем вид, что всё сломалось
    throw runtime_error("I'm func1. I can not do my work. I'm just too lazy today.");

    // Если требуется, работаем дальше
}
```


```cpp
int main()
{
    try {
        // Вызов функции
        func1();

        // Тут идёт какая-то работа дальше

    // Обработка возможной проблемы
    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }

    return 0;
}
```

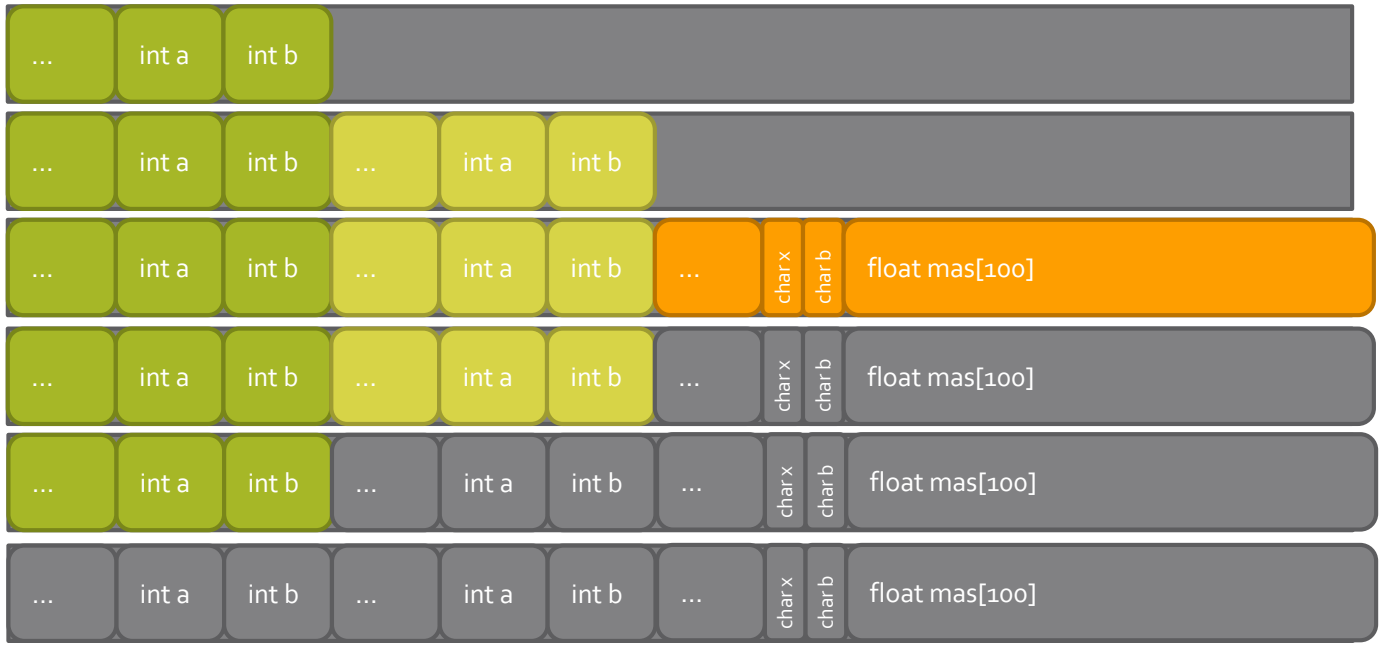
Как exception помогает

```
void func1()
{
    // Тут происходит какая-то работа

    // Делаем вид, что всё сломалось
    throw runtime_error("I'm func1. I can not do my work. I'm just too lazy today.");

    // Дальше какая-то работа
}
```

```
void func2()
{
    // Что-то делаем

    func1();

    // Что-то делаем дальше
}

void func3()
{
    // Что-то делаем

    func2();

    // Что-то делаем дальше
}
```

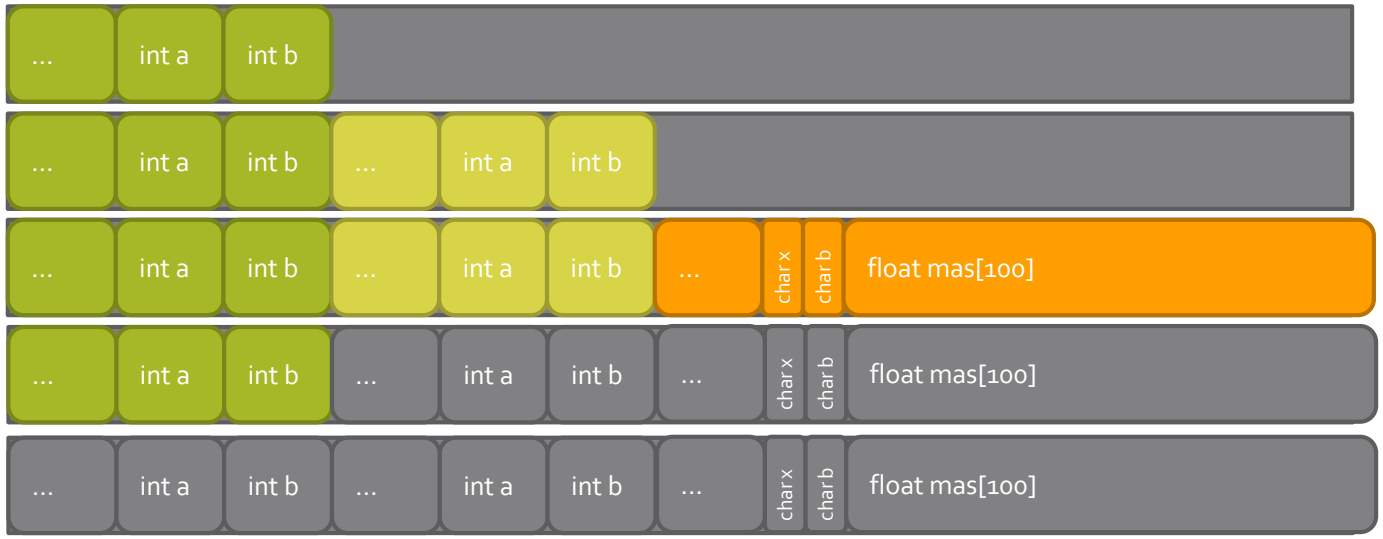
```
int main()
{
    try {
        func3();

        // Тут идёт какая-то работа дальше

    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }
```

Штатная работа стека




```
void g(char x, char b) {
    float mas[100];
}

void f() {
    int a, b;
    g('a', 48);
}

int main() {
    int a, b;
    f();
    return 0;
}
```

Исключение




```
void g(char x, char b) {
    float mas[100];
    throw 100500;
}

void f() {
    int a, b;
    g('a', 48);
}

int main() {
    int a, b;
    try {
        f();
    }
    catch(int a)
    {
        cout << a << endl;
    }
    return 0;
}
```

Зачем нужен exception

Exception позволяет отдать обработку ошибок тому,
кто обнаружил проблему (бросил exception),
и тому, кто может её обработать (поймал exception).

А всем слоям между не нужно делать ничего.

Что такое exception

Кинуть можно что угодно.

Удобнее всего кидать объект стандартного класса.

Их много, все отнаследованы от std::exception.

Или самим от него отнаследоваться.

std::range_error

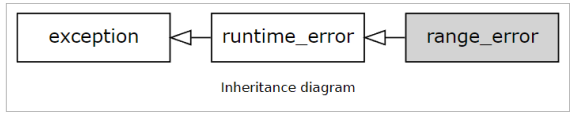
Defined in header <stdexcept>

```
class range_error;
```

Defines a type of object to be thrown as exception. It can be used to rep
result of a computation cannot be represented by the destination type).

The only standard library components that throw this exception are std
std::wstring_convert::to_bytes.

The mathematical functions in the standard library components do not t
report range errors as specified in math_errhandling).

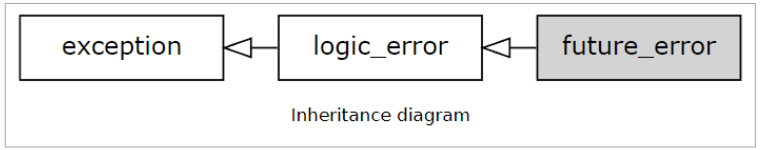


std::future_error

Defined in header <future>

```
class future_error;    (since C++11)
```

The class **std::future_error** defines an exception object that is throw
that deal with asynchronous execution and shared states (std::future
std::system_error, this exception carries an error code compatible wi



Standard exceptions

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (since C++11)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (since C++11)
  - system_error (since C++11)
    - ios_base::failure (since C++11)
    - filesystem::filesystem_error (since C++17)
  - tx_exception (TM TS)
  - nonexistent_local_time (since C++20)
  - ambiguous_local_time (since C++20)
  - format_error (since C++20)
- bad_typeid
- bad_cast
  - bad_any_cast (since C++17)
- bad_optional_access (since C++17)
- bad_expected_access (since C++23)
- bad_weak_ptr (since C++11)
- bad_function_call (since C++11)
- bad_alloc
  - bad_array_new_length (since C++11)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access (since C++17)

ИСПОЛЬЗОВАНИЕ EXCEPTION

по делу и не по делу

Использование exception

Не надо использовать exception как замену return!

- Exception – фатальная проблема, с которой нельзя справиться, а не вариант штатной работы.
- Если условный 1% вызовов приводит к exception, то это уже повод внимательно проверить логическую архитектуру.
- Обработка реально выброшенного exception – достаточно дорогая операция.

Использование exception

Не надо использовать exception в ситуациях, когда ваш код явно технически некорректно вызван.

- Если ваш код некорректно использован – надо падать, чтобы заставить вызвавшего разобраться и починить код на своей стороне.
- Exception – это когда корректный код столкнулся с фатально проблемными условиями работы.

Использование exception

- Можно и нужно задавать классы exception-ов.
- Общее правило: класс описывает логическую ситуацию, а не место возникновения проблемы.
- Ловить exception-ы нужно только в том случае, если знаете, что делать с пойманным.
- Стоит помнить, что вообще любая строчка кода потенциально выбрасывает exception.

Свои классы exception

```
// Опишем свой новый логический exception
class LazyException : public std::runtime_error
{
public:
    LazyException() : std::runtime_error("I'm just too lazy today") { }
};
```

```
// Happy path -- штатный режим работы
try {
    // Вызов func3
    func3();

    // Ещё какие-то операции дальше

// Обработка известной нам возможной логической проблемы
} catch (const LazyException& e) {
    cerr << "Damned function is lazy now. Requires special handling." << endl;
    return -1;
// Реакция по умолчанию на все прочие мыслимые проблемы
} catch (const exception& e) {
    cerr << "We failed! And something real happened this time!" << endl;
    cerr << "Failure reason: " << e.what() << endl;
    return -1;
}
```

Использование exception

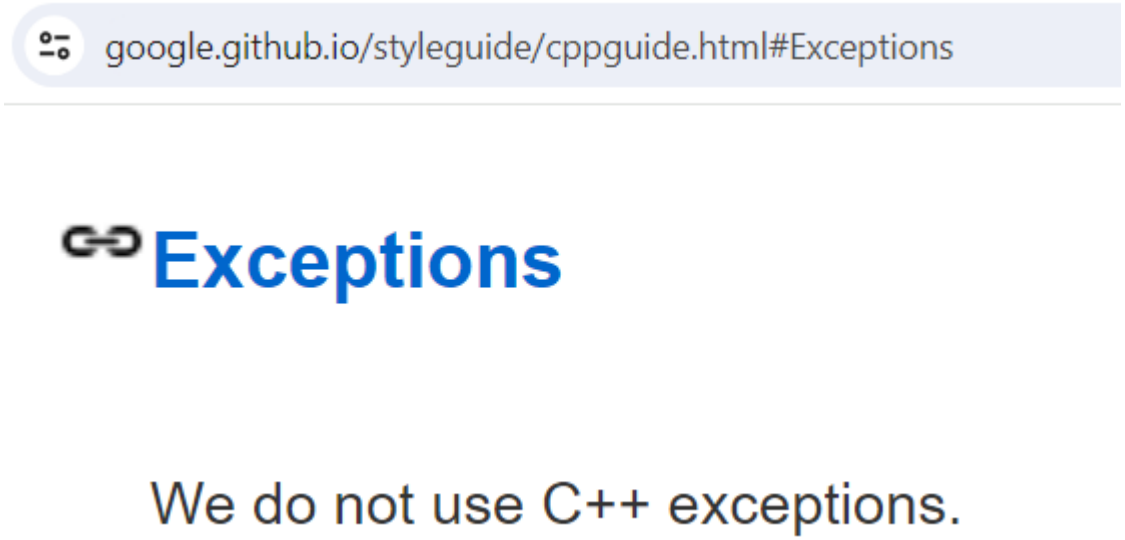
- Позволяют «локализовать» обработку проблем – проблема касается только того, кто обнаружил ошибку, и того, кто может её обработать (и не касается всех слоёв между ними).
- Позволяют разделить код штатной работы («happy path») и код обработки ошибок («error path»), сделать программу в целом читаемее.
- Для использования exception-ов нужно перестроить мышление. Проектирование обработки исключений – очень сложная часть реальной разработки.

ПРОБЛЕМЫ С ИСКЛЮЧЕНИЯМИ

как сделать плохое зло

Google code style

https://google.github.io/styleguide/cppguide.html



Плохое зло #1

```
class MyClass
{
protected:
    int* data;

public:
    MyClass()
    {
        cout << "MyClass constructor called" << endl;
        data = new int[1000];
    }

    ~MyClass()
    {
        cout << "MyClass destructor called" << endl;
        delete[] data;
    }

    void doSomeWork()
    {
        cout << "Ooops happened" << endl;
        throw runtime_error("Oops");
    }
};
```

```
void imposter()
{
    MyClass a;
    a.doSomeWork();
}

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

```
==444== Using Valgrind-3.21.0.RC2 and LibVEX; rerun
==444== Command: ./a.out
==444==
MyClass constructor called
Ooops happened
terminate called after throwing an instance of 'std
  what():  Oops
==444==
==444== Process terminating with default action of s
==444==    at 0x4B1900B: raise (raise.c:51)
==444==    by 0x4AF8858: abort (abort.c:79)
==444==    by 0x48F7A48: ??? (in /usr/local/gcc-12.2
==444==    by 0x4903079: ??? (in /usr/local/gcc-12.2
==444==    by 0x49030E4: std::terminate() (in /usr/l
==444==    by 0x4903336: __cxa_throw (in /usr/local/
==444==    by 0x1094B5: MyClass::doSomeWork() (in /h
==444==    by 0x1092DC: imposter() (in /home/amisto/
==444==    by 0x109331: main (in /home/amisto/oop-2r
==444==
==444== HEAP SUMMARY:
==444==     in use at exit: 77,901 bytes in 5 blocks
==444==   total heap usage: 6 allocs, 1 frees, 77,93
==444==
==444== LEAK SUMMARY:
==444==    definitely lost: 0 bytes in 0 blocks
==444==    indirectly lost: 0 bytes in 0 blocks
==444==      possibly lost: 144 bytes in 1 blocks
==444==    still reachable: 77,757 bytes in 4 blocks
==444==                       of which reachable via
==444==                         stdstring
==444==         suppressed: 0 bytes in 0 blocks
```

~~Плохое зло #1~~

```
class MyClass
{
protected:
    int* data;

public:
    MyClass()
    {
        cout << "MyClass constructor called" << endl;
        data = new int[1000];
    }

    ~MyClass()
    {
        cout << "MyClass destructor called" << endl;
        delete[] data;
    }

    void doSomeWork()
    {
        cout << "Ooops happened" << endl;
        throw runtime_error("Oops");
    }
};
```

```
int main()
{
    try {
        imposter();
    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }
    return 0;
}
```

```
==486== Using Valgrind-3.21.0.RC2 and LibVEX; rerun with -h for copyright info
==486== Command: ./a.out
==486==
MyClass constructor called
Ooops happened
MyClass destructor called
We failed!
Failure reason: Oops
==486==
==486== HEAP SUMMARY:
==486==     in use at exit: 0 bytes in 0 blocks
==486==   total heap usage: 5 allocs, 5 frees, 77,901 bytes allocated
```

Плохое зло #2

```
class MyClass
{
protected:
    int* data;

public:
    MyClass()
    {
        cout << "MyClass constructor called" << endl;
        data = new int[1000];
        throw runtime_error("Oops in constructor");
    }

    ~MyClass()
    {
        cout << "MyClass destructor called" << endl;
        delete[] data;
    }

    void doSomeWork()
    {
        cout << "Ooops happened" << endl;
        throw runtime_error("Oops");
    }
};
```

```
int main()
{
    try {
        imposter();
    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }
    return 0;
}
```

```
==512== Using Valgrind-3.21.0.RC2 and LibVEX; rerun with -h for copyright info
==512== Command: ./a.out
==512==
MyClass constructor called
We failed!
Failure reason: Oops in constructor
==512==
==512== HEAP SUMMARY:
==512==     in use at exit: 4,000 bytes in 1 blocks
==512==   total heap usage: 5 allocs, 4 frees, 77,916 bytes allocated
==512==
==512== LEAK SUMMARY:
==512==    definitely lost: 4,000 bytes in 1 blocks
==512==    indirectly lost: 0 bytes in 0 blocks
==512==      possibly lost: 0 bytes in 0 blocks
==512==    still reachable: 0 bytes in 0 blocks
==512==         suppressed: 0 bytes in 0 blocks
```

УМНЫЕ УКАЗАТЕЛИ

умнее программистов

Что такое smart pointer

Smart pointer:

- Специальная «тонкая» и «лёгкая» обёртка над «обычным» указателем.
- Следит, насколько ещё жив владелец указателя. Если владелец уже мёртв, освобождает память.

Плохое зло #3

```
int* data = new int[10];

for(unsigned int i = 0; i < 10; i++)
    data[i] = i;

for(unsigned int i = 0; i < 10; i++)
    cout << data[i] << endl;

// Ой, забыли
//delete[] data;
```

```
HEAP SUMMARY:
    in use at exit: 40 bytes in 1 blocks
  total heap usage: 3 allocs, 2 frees, 73,768 bytes allocated

LEAK SUMMARY:
   definitely lost: 40 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

~~Плохое зло #3~~

```
unique_ptr<int[]> data = unique_ptr<int[]>(new int[10]);

for(unsigned int i = 0; i < 10; i++)
    data[i] = i;

for(unsigned int i = 0; i < 10; i++)
    cout << data[i] << endl;

// Теперь и не надо. За очистку памяти отвечает smart pointer.
//delete[] data;
```

Какие бывают умные указатели

unique_ptr

- ровно один владелец указателя
- нельзя копировать, можно сменить владельца
- используйте по умолчанию его, если не уверены, что именно требуется в конкретном случае

shared_ptr

- несколько владельцев указателя
- можно копировать (это добавляет владельца)
- считает количество ещё живых владельцев
- освобождает память при смерти последнего

Разница между shared и unique

```
// Допустим, эта функция должна откуда-то считать пользователей,
// перебрать их, выбрать и вернуть нужного.
unique_ptr<User> getUser()
{
    // Массив указателей (smart pointer-ов) на пользователей
    // (сделаем вид, что динамически их читаем откуда-то)
    unique_ptr<User> users[2];
    users[0] = unique_ptr<User>(new User("Alice"));
    users[1] = unique_ptr<User>(new User("Bob"));

    // Работа с массивом
    cout << "The users:" << endl;
    for (int i = 0; i < 2; i++)
        cout << *users[i] << endl;

    // Очень хочется выбрать один элемент и работать с ним отдельно
    // Например, хочется вернуть его из функции
    unique_ptr<User> selected;
    selected = users[0];
    cout << "Selected user: " << *selected << endl;

    return selected;
}
```

Разница между shared и unique

```
shared_ptr<User> getUser()
{
    // Массив указателей (smart pointer-ов) на пользователей
    // (сделаем вид, что динамически их читаем откуда-то)
    shared_ptr<User> users[2];
    users[0] = shared_ptr<User>(new User("Alice"));
    users[1] = shared_ptr<User>(new User("Bob"));

    // Работа с массивом
    cout << "The users:" << endl;
    for (int i = 0; i < 2; i++)
        cout << *users[i] << endl;

    // Очень хочется выбрать один элемент и работать с ним отдельно
    // Например, хочется вернуть его из функции
    shared_ptr<User> selected;
    selected = users[0];
    cout << "Selected user: " << *selected << endl;

    return selected;
}
```

```
Calling getUser()
Constructor called for Alice
Constructor called for Bob
The users:
Alice
Bob
Selected user: Alice
Destructor called for Bob
Returned from getUser()
Selected user: Alice
Destructor called for Alice
```

Плохое зло #4

```
int main()
{
    cout << "Calling getUser()" << endl;
    shared_ptr<User> user = getUser();
    cout << "Returned from getUser()" << endl;
    cout << "Selected user: " << *user << endl;
    delete &*user;
    return 0;
}
```


```
Calling getUser()
Constructor called for Alice
Constructor called for Bob
The users:
Alice
Bob
Selected user: Alice
Destructor called for Bob
Returned from getUser()
Selected user: Alice
Destructor called for Alice
Destructor called for ▯UBob!▯U▯▯Ufree(): double free detected in tcache 2
Aborted
```

Какие ещё бывают умные указатели

weak_ptr

- используется только совместно с shared_ptr
- даёт доступ к объекту, на который указывает shared_prt, но не участвует в подсчёте ссылок
- уместен для observer-ов
- используется для борьбы с кольцевыми ссылками между shared_ptr

auto_ptr <- не особо умный указатель

- исторически «первый блин комом»
- опасное поведение при копировании
- считается устаревшим, не используйте его в новом коде, заменяйте на другие виды

Плохое зло #5

```
int main()
{
    auto_ptr<int> a = auto_ptr<int>(new int(100));
    cout << *a << endl;
    return 0;
}
```

```
==796== Using Valgrind-3.21.0.RC2 and LibVEX; rerun with -h
==796== Command: ./a.out
==796==
100
==796==
==796== HEAP SUMMARY:
==796==     in use at exit: 0 bytes in 0 blocks
==796==   total heap usage: 3 allocs, 3 frees, 73,732 bytes
==796==
==796== All heap blocks were freed -- no leaks are possible
```

```
void f(auto_ptr<int> x)
{
    cout << *x << endl;
}

int main()
{
    auto_ptr<int> a = auto_ptr<int>(new int(100));
    f(a);
    cout << *a << endl;
    return 0;
}
```

```
==806== Using Valgrind-3.21.0.RC2 and LibVEX; rerun with -h for
==806== Command: ./a.out
==806==
100
==806== Invalid read of size 4
==806==    at 0x109309: main (in /home/amisto/oop-2nd-term/2024/
==806==  Address 0x0 is not stack'd, malloc'd or (recently) free
==806==
==806==
==806== Process terminating with default action of signal 11 (SI
==806==  Access not within mapped region at address 0x0
==806==    at 0x109309: main (in /home/amisto/oop-2nd-term/2024/
==806==  If you believe this happened as a result of a stack
==806==  overflow in your program's main thread (unlikely but
==806==  possible), you can try to increase the size of the
==806==  main thread stack using the --main-stacksize= flag.
==806==  The main thread stack size used in this run was 8388608
```

УМНЫЕ УКАЗАТЕЛИ И ИСКЛЮЧЕНИЯ

в сочетании

Плохое зло #2

```
class MyClass
{
protected:
    int* data;

public:
    MyClass()
    {
        cout << "MyClass constructor called" << endl;
        data = new int[1000];
        throw runtime_error("Oops in constructor");
    }

    ~MyClass()
    {
        cout << "MyClass destructor called" << endl;
        delete[] data;
    }

    void doSomeWork()
    {
        cout << "Ooops happened" << endl;
        throw runtime_error("Oops");
    }
};
```

```
int main()
{
    try {
        imposter();
    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }
    return 0;
}
```

```
==512== Using Valgrind-3.21.0.RC2 and LibVEX; rerun with -h for copyright info
==512== Command: ./a.out
==512==
MyClass constructor called
We failed!
Failure reason: Oops in constructor
==512==
==512== HEAP SUMMARY:
==512==     in use at exit: 4,000 bytes in 1 blocks
==512==   total heap usage: 5 allocs, 4 frees, 77,916 bytes allocated
==512==
==512== LEAK SUMMARY:
==512==    definitely lost: 4,000 bytes in 1 blocks
==512==    indirectly lost: 0 bytes in 0 blocks
==512==      possibly lost: 0 bytes in 0 blocks
==512==    still reachable: 0 bytes in 0 blocks
==512==         suppressed: 0 bytes in 0 blocks
```

~~Плохое зло #2~~

```
class MyClass
{
protected:
    unique_ptr<int[]> data;
public:
    MyClass()
    {
        cout << "MyClass constructor called" << endl;
        data = unique_ptr<int[]>(new int[1000]);
        throw runtime_error("Oops in constructor");
    }

    ~MyClass()
    {
        cout << "MyClass destructor called" << endl;
        // Этот вызов больше не нужен
        // delete[] data;
    }

    void doSomeWork()
    {
        cout << "Ooops happened" << endl;
        throw runtime_error("Oops");
    }
};
```

```
int main()
{
    try {
        imposter();
    } catch (const exception& e) {
        cerr << "We failed!" << endl;
        cerr << "Failure reason: " << e.what() << endl;
        return -1;
    }
    return 0;
}
```

```
MyClass constructor called
We failed!
Failure reason: Oops in constructor
==750==
==750== HEAP SUMMARY:
==750==     in use at exit: 0 bytes in 0 blocks
==750==   total heap usage: 5 allocs, 5 frees, 77,916 bytes allocated
==750==
==750== All heap blocks were freed -- no leaks are possible
```

Финальные ремарки

- Коды возврата – C/C-style способ обрабатывать ошибки
- Исключения – способ из C++11 и новее
- У исключений есть свои минусы, но в целом их советуют использовать
- **Используйте исключения в исключительных ситуациях** – bad path медленный
- Умные указатели делают жизнь лучше
- Умные указатели работают быстро
- **Используйте умные указатели**
- Не используйте умные указатели в сочетании с сырыми
- Не используйте auto_ptr