Исключения и обработка ошибок

Максим Бакиров
C++ - Разработчик в Яндекс

Проверка связи

Поставьте “+”, если меня видно и слышно

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включен звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Максим Бакиров

О спикере:

- В C++ разработке с 2017 года
- С 2019 года работает в команде разработки Яндекс Браузера

Вспоминаем прошлое занятие

Вопрос: что такое препроцессор и чем он занимается?

Вспоминаем прошлое занятие

Вопрос: что такое препроцессор и чем он занимается?

Ответ: препроцессор - это текстовый преобразователь, который преобразует файлы исходного кода перед компиляцией

Вспоминаем прошлое занятие

Вопрос: что такое препроцессорная директива?

Вспоминаем прошлое занятие

Вопрос: что такое препроцессорная директива?

Ответ: это указание препроцессору - как ему преобразовать текст

Вспоминаем прошлое занятие

Вопрос: какие директивы отвечают за условную компиляцию?

Вспоминаем прошлое занятие

Вопрос: какие директивы отвечают за условную компиляцию?

Ответ: #ifdef, #ifndef, #if, #elif, #else, #endif

Вспоминаем прошлое занятие

Вопрос: что такое макросы и как их определить?

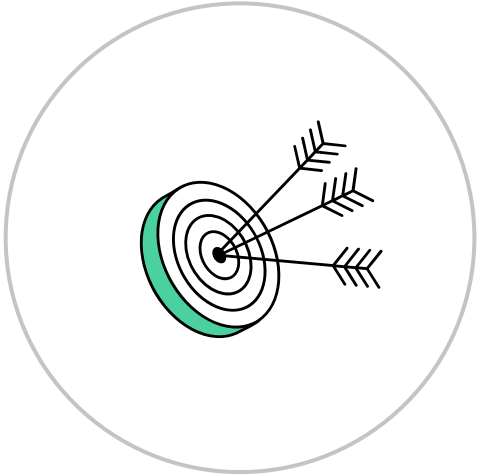
Вспоминаем прошлое занятие

Вопрос: что такое макросы и как их определить?

Ответ: макросы - это идентификаторы с параметрами, определённые с помощью директивы #define

Цели занятия

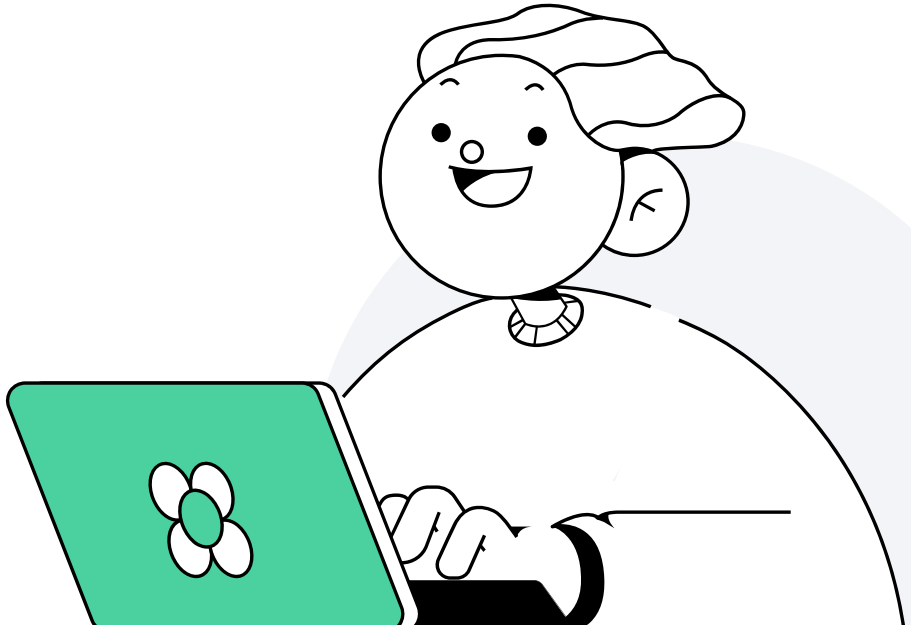
- Разберёмся, какие бывают ошибки
- Познакомимся с методами поиска, предотвращения и обработки ошибок
- Узнаем, что такое коды возврата
- Выясним, что такое исключения



План занятия

1. Ошибки и поиск ошибок
2. Коды возврата
3. Исключения
4. Итоги
5. Домашнее задание

*Нажми на нужный раздел для перехода



Ошибки и поиск ошибок

1

Ошибки

Вы уже заметили, что при написании программ возникают ошибки

Какие они бывают?

Какие бывают ошибки

Ошибки времени компиляции - это синтаксические ошибки, которые препятствуют сборке программы

Мы не раз с ними сталкивались - например, если забыли где-то поставить ; (точку с запятой) в конце строки

Синтаксические ошибки легко найти - программа не сможет собраться, а компилятор укажет, в чём ошибка

Ошибки времени выполнения возникают в ходе работы программы

Например, если вы пытаетесь поделить на ноль, обратиться к элементу массива по индексу, выходящему за его пределы, открыть несуществующий файл, выделить памяти больше, чем её доступно

При таких ошибках программа сама завершится из-за невозможности продолжать свою работу

Ошибки времени выполнения

Бывают ещё другие ошибки - они связаны с тем, как должна работать программа по задумке программиста

Например, в программе “Калькулятор” вы можете вместо перемножения чисел складывать их - программа от этого не сломается, но результат будет не тот, что задумывался изначально

Такие ошибки отловить сложнее всего - для этого существует тестирование

В этой лекции мы не будем рассматривать подобные ошибки

Ищем ошибку

Представим, что мы запускаем написанную нами программу “Калькулятор”, а она в какой-то момент прекращает работу. Как нам понять, что произошло?

Вопрос: Что нужно сделать чтобы найти ошибку? Напишите в чат

Ищем ошибку

Чтобы найти ошибку, нам нужно проследить, как работает наш код.
Для этого мы можем:

- войти в режим отладки и последовательно её выполнить
- заставить программу сообщать нам о том, на каком этапе выполнения она находится - это называется **логирование**

Как вы думаете,

какой из способов лучше?
Каковы преимущества
и недостатки каждого?

Напишите в чат

Способы поиска ошибок

Идеального решения нет - у каждого свои преимущества и недостатки

Отладка:

+ не нужно изменять исходный код
+ наглядно, можно посмотреть состояние любой переменной

– требует доступа к исходному коду
– требует наличия отладчика и возможности его запуска
– если программа большая, то поиск может затянуться

Логирование:

+ не требуется доступа к исходному коду, работает вместе с программой
+ все этапы выполнения программы доступны сразу, их надо только проанализировать

– логирование требует дополнительного кода в программе
– при маленьких объёмах кода время, потраченное на логирование, может быть больше, чем вся возможная отладка

Способы поиска ошибок. Вывод

У каждого способа своя сфера применения:

- **отладка** подходит для маленьких программ или поиска ошибки в маленьком кусочке большой программы
- **логирование** стоит использовать в больших и сложных программах

После того, как мы нашли ошибку, нам нужно понять, как её исправить или предотвратить

Что делать с найденной ошибкой?

Предположим, мы обнаружили, что наша программа “Калькулятор” падает, если мы пытаемся вторым аргументов в функцию divide передать число 0

```
float divide(float dividend, float divisor)
{
    return dividend / divisor; // Ошибка
}

int main(int argc, char** argv)
{
    float a = 5.6;
    float b = 0;
    std::cout << a << " / " << b << " = " << divide(a, b) << std::endl;
}
```

Как вы думаете, как нам исправить такой код? **Напишите в чат**

Что делать с найденной ошибкой?

Мы должны проверять, является ли переменная divisor равной 0 прежде, чем производить деление

Мы можем это сделать в двух местах:

- в функции main перед вызовом функции divide
- в функции divide перед выполнением операции деления

Как вы думаете,

в каком месте лучше это сделать
и почему?

Напишите в чат

Где проверять значение?

Это нужно делать в функции divide, потому что именно она обладает знаниями о том, какие ограничения накладываются на аргументы

Функция main не может знать всех деталей, которые надо проверять

К тому же, если функция main будет проверять аргументы всех вызываемых ею функций, то код функции main станет очень большим и несфокусированным. Хороший пример нарушения принципа инкапсуляции

Вопрос

Мы выяснили, что нужно проверять аргумент в функции divide. Но что мы должны сделать, если значение параметра divisor равняется 0?

Как вы думаете, что нужно сделать?
Напишите в чат

Что будем делать?

Если divisor равняется нулю, то это значит, что кто-то пытается воспользоваться нашей функцией неправильно. Это значит, что нужно сообщить тому, кто вызвал функцию, что она не может работать с такими аргументами и свою задачу выполнить не в состоянии

Перерыв

Коды возврата

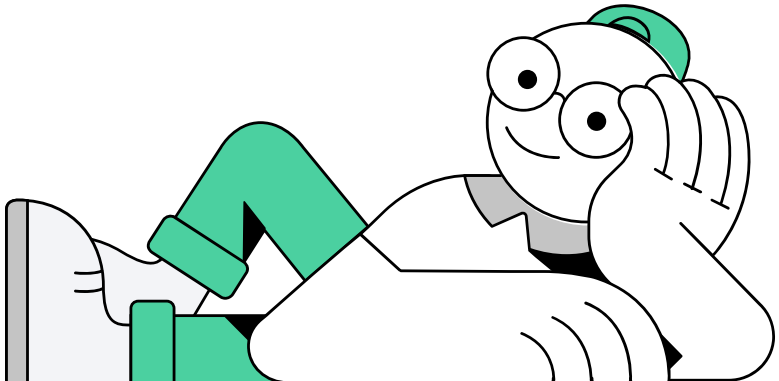
2

Код возврата

Коды возврата - это самый первый способ сообщения о произошедшей ошибке

Идея в том, что функция вместо результата, который она должна посчитать, возвращает специальный код, который сообщает о том, правильно отработала функция или нет

Обычно в качестве кодов возврата используются целые числа



Как вы думаете,

как тогда получить
результат функции?

Напишите в чат

Результат функции

Для того, чтобы вернуть результат, в функцию передаётся ещё один аргумент - указатель на переменную, в которую будет помещён результат функции. Вместо указателя может выступать переменная, переданная по ссылке

Обычно делают так:
код 0 (ноль) означает, что функция отработала корректно. Коды больше 0 означают ошибку. Давайте вернём код 1, если divisor равен 0 и функция не может произвести деление

В функции main обработаем код возврата - если он не 0, то функция отработала некорректно. Сообщим об этом пользователю

Код возврата. Пример

```
int divide(float dividend, float divisor, float* quotient)
{
    if (divisor == 0)
        return 1;
    *quotient = dividend / divisor;
    return 0;
}

int main(int argc, char** argv)
{
    float a = 5.6;
    float b = 0;
    float c = 0;
    int divide_result = divide(a, b, &c);
    if (divide_result == 0)
            std::cout << a << " / " << b << " = " << c << std::endl;
    else if (divide_result == 1)
            std::cout << "Не могу поделить на 0" << std::endl;
}
```

Новая ошибка

Теперь, когда мы изменили нашу функцию divide - появилось новое место для потенциальной ошибки - нам могут в качестве аргумента quotient прислать нулевой указатель (nullptr). Это указатель, который не указывает ни на какой участок памяти, адрес, хранящийся в нём, равен нулю

Если мы попытаемся записать значение по такому указателю - мы получим ошибку и наша программа аварийно завершится

Давайте проверим и эту возможность - если указатель нулевой, вернём код 2, а в функции main выведем сообщение пользователю

Код возврата. Пример

```
int divide(float dividend, float divisor, float* quotient)
{
    if (divisor == 0)
        return 1;
    if (quotient == nullptr)
        return 2;
    *quotient = dividend / divisor;
    return 0;
}

int main(int argc, char** argv)
{
    float a = 5.6, b = 0, c = 0;
    int divide_result = divide(a, b, &c);
    if (divide_result == 0)
            std::cout << a << " / " << b << " = " << c << std::endl;
    else if (divide_result == 1)
            std::cout << "Не могу поделить на 0" << std::endl;
    else if (divide_result == 2)
            std::cout << "Не могу записать результат в nullptr" << std::endl;
}
```

Улучшаем код возврата

Теперь представим, что вы вызвали функцию чтения из файла, а она вам вернула число 5

Что это значит? Что файла не существует? Или что он существует, но к нему нет доступа? Или доступ есть, но вы открываете его в неправильном режиме? Или всё хорошо, но не хватает оперативной памяти?

Очевидно, что просто число без документации - это не слишком информативно. Хорошо, если документация есть и её можно прочитать, но ещё лучше - если код документирует сам себя

Как вы думаете,

как нам сделать коды возврата более понятными?

Напишите в чат

Код возврата. Пример

Можно использовать перечисление (enum)!

```
enum class DivisionResult { Success, DivisionByZero, ArgumentIsNull };
DivisionResult divide(float dividend, float divisor, float* quotient)
{
    if (divisor == 0)
        return DivisionResult::DivisionByZero;
    if (quotient == nullptr)
        return DivisionResult::ArgumentIsNull;
    *quotient = dividend / divisor;
    return DivisionResult::Success;
}
int main(int argc, char** argv)
{
    float a = 5.6, b = 0, c = 0;
    DivisionResult divide_result = divide(a, b, &c);
    if (divide_result == DivisionResult::Success)
            std::cout << a << " / " << b << " = " << c << std::endl;
    else if (divide_result == DivisionResult::DivisionByZero)
            std::cout << "Не могу поделить на 0" << std::endl;
    else if (divide_result == DivisionResult::ArgumentIsNull)
            std::cout << "Не могу записать результат в nullptr" << std::endl;
}
```

Ещё один способ вернуть код ошибки (плохой)

Ещё один способ сообщить пользователю функции о результате её работы - записать его в глобальную переменную, которую пользователь функции после её вызова должен будет прочитать, чтобы узнать, как отработала функция

Этот способ плох по двум причинам:

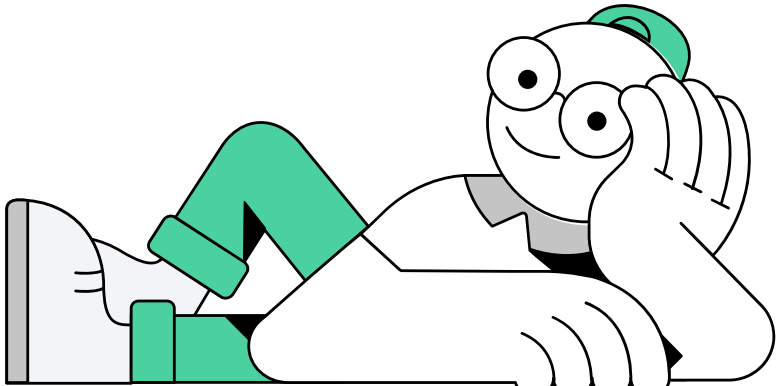
- Во-первых, он предполагает использование глобальной переменной, это само по себе плохо. Но этот момент при желании можно обойти
- Во-вторых, пользователю функции совершенно не очевидно, что где-то есть переменная с каким-то специальным именем, в которой хранится результат функции. Он может (и будет) её просто игнорировать

Код возврата. Итоги

Код возврата тоже не является идеальным решением - ведь пользователь функции может проигнорировать и его, но это уже труднее сделать

Тем не менее, использование кода возврата является наилучшим решением с точки зрения производительности. Однако не является лучшим с точки зрения безопасности и удобства использования

Если вам придётся использовать коды возврата внутри программы - используйте enum class



Исключения

Исключения (англ. exceptions) - это специальный набор инструментов для создания, нахождения и обработки ошибок

Общий смысл такой: исключения - это специальные значения, которые “выбрасываются” кодом при попытке выполнить какое-то неправильное действие (напр. деление на ноль). Выброшенное исключение содержит в себе информацию о произошедшей ошибке и проходит по стеку вызовов наверх до тех пор, пока его кто-то не обработает

Если исключение никто не обработал, то программа аварийно завершается

Исключения. Синтаксис

Для “выбрасывания” исключения используется ключевое слово **throw** (бросить)

throw <исключение>;

<исключение> - это значение любого типа. Тем не менее, в C++ создан специальный тип для исключений - **std::exception**

Исключения. Синтаксис

Этот класс символизирует общее исключение и объявляет публичный виртуальный метод **char* what()**, вызвав который, можно получить текстовое описание ошибки. От этого класса наследуются более конкретные стандартные исключения, и вы можете создать свой класс исключения

Рекомендуется пользоваться std::exception и его наследниками. Также у std::exception есть конструктор, принимающий строку - для описания ошибки

Вызов оператора throw прерывает работу программы и возвращает его наверх по стеку вызовов в ожидании обработки

Исключения. Синтаксис

Не любой код может обработать выброшенное исключение. Для этого код должен “ждать” его с помощью оператора **try** (попробовать). Если такой код дождался исключения, то обработка его производится с помощью оператора **catch** (поймать).

```
try
{
    <код, который может выбросить исключение>
}
catch(<исключение>)
{
    <код, обрабатывающий исключение>
}
```

Хорошей практикой является “бросать” исключения по значению, а “ловить” - по ссылке

Исключения. Синтаксис

Параметром для catch выступает объявление исключения - почти как аргумент в функции, но имя необязательно

Здесь мы должны указать тип исключения, которое мы ждём

Мы можем написать сразу несколько блоков catch, чтобы обрабатывать исключения разных типов - но надо помнить о том, что компилятор пытается привести выброшенное исключение к типу каждого блока catch сверху вниз. А это значит, что исключения более общего типа надо писать ниже, чем исключения более конкретного.

Исключения. Синтаксис

Наконец, чтобы обработать исключение любого типа, надо написать ... (троеточие). Такой блок всегда должен идти последним, потому что является наиболее общим

```
try { <код, который может выбросить исключение> }
catch(<тип_1> [<имя_1>]) { <код, обрабатывающий исключение типа тип_1> }
catch(<тип_2> [<имя_2>]) { <код, обрабатывающий исключение типа тип_2> }
...
catch(...) { <код, обрабатывающий любое исключение> }
```

Исключения. Пример

```
class DivisionByZeroException : public std::exception
{
public:
    const char* what() const override { return "Попытка деления на 0"; }
};
float divide(float dividend, float divisor)
{
    if (divisor == 0) throw DivisionByZeroException();
    return dividend / divisor;
}
int main(int argc, char** argv)
{
    float a = 5.6, b = 0, c = 0;
    try
    {
        c = divide(a, b);
        std::cout << a << " / " << b << " = " << c << std::endl;
    }
    catch (const DivisionByZeroException& ex) { std::cout << ex.what() << std::endl; }
    catch (...) { std::cout << "Неизвестная ошибка" << std::endl; }
}
```

Стандартные исключения

В C++ уже существуют классы исключений для разных случаев.
Вот некоторые из них:

- **logic_error** - для логических ошибок предметной области
  - **invalid_argument** - неприемлемое значение аргумента
  - **domain_error** - для разных ошибок предметной области
  - **length_error** - попытка превысить какую-либо длину
  - **out_of_range** - попытка получить доступ к элементу за пределами
- **bad_cast** - неудачная попытка применить dynamic_cast
- **bad_alloc** - неудачная попытка получить память из кучи
  - **bad_array_new_length** (C++ 11) - неправильная длина при создании динамического массива

Стандартные исключения 2

- **runtime_error** - для внешних труднопредсказуемых ошибок
  - **overflow_error** - ошибка переполнения типа
  - **system_error** - для ошибок взаимодействия с ОС
    - **ios_base::failure** (C++ 11) - ошибки библиотеки Input/Output
    - **filesystem::filesystem_error** (C++ 17) - ошибки библиотеки filesystem

Преимущества исключений

У механизма исключений есть несколько преимуществ:

- Обнаружение ошибки (throw) и обработка (catch) чётко разделены
- Выброшенное исключение может быть обработано на разных уровнях вызова - в зависимости от типа исключения. Промежуточные функции, не обладающие компетенциями для обработки исключения, могут пропустить его дальше наверх
- Исключение заставляет программиста отреагировать на ошибку - обработать исключение либо аварийно завершить работу программы. Игнорировать не удастся
- Исключение - единственный способ остановить конструирование объекта до его завершения. Это полезно, если конструктору были переданы неправильные аргументы

Недостатком исключений является производительность - она меньше, чем у кодов возврата

Почему бы не поступить проще

Зачем всё это делать?
Можно же просто при обнаружении ошибки напечатать на экран сообщение

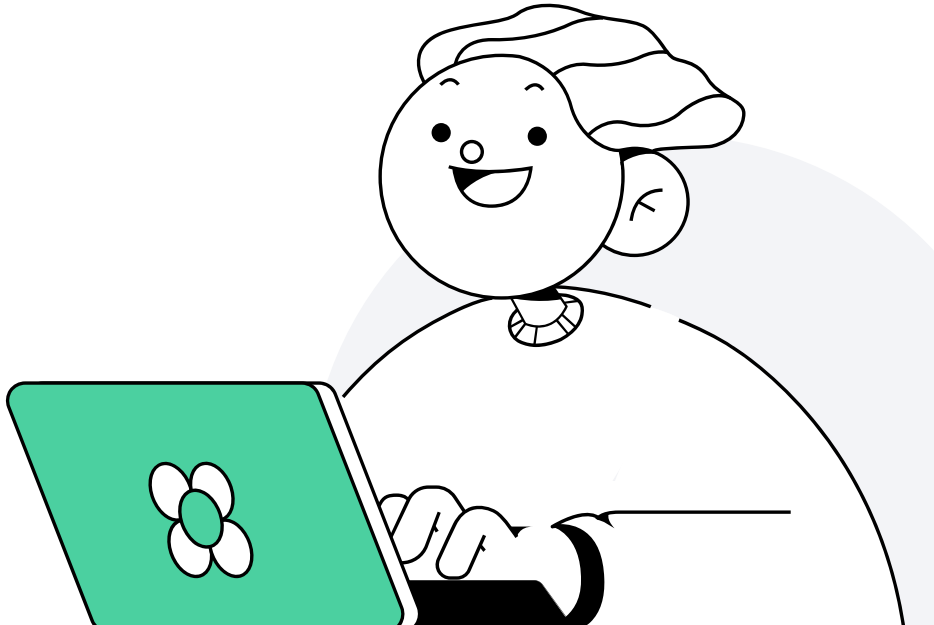
Но это плохой способ, потому что потом вы можете осознать необходимость не выводить ошибки на экран, а печатать их в файл, например - чтобы потом изучить

И тогда вам придётся бегать по всему своему коду и менять вывод на консоль на вывод в файл

Почему бы не поступить проще

Остаётся проблема возврата какого-то значения из функции
- **исключения решают эту проблему, прерывая выполнение**

Печатание на экран не решает проблему выполнения каких-то особенных действий в случае ошибки



Итоги

Итоги занятия

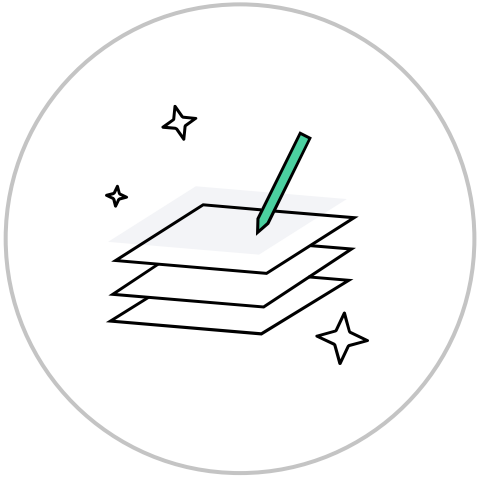
Сегодня мы

1. Разобрались, какие бывают ошибки
2. Познакомились с методами поиска, предотвращения и обработки ошибок
3. Узнали, что такое коды возврата
4. Выяснили, что такое исключения

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Обработка исключений
- Тип exception

Задавайте вопросы и пишите отзыв о лекции

Максим Бакиров
C++ - разработчик в Яндекс