Семинар #3: Инициализация. Обработка ошибок.

Инициализация

Инициализация – это процесс присвоения начального значения переменной или объекту при их создании.

Инициализация в языке C

Прежде чем углубляться в инициализацию в языке C++, давайте рассмотрим, как она реализована в языке C.

- По умолчанию объекты, созданные в сегментах стек и куча, не инициализируются. После создания объекта в них могут находиться произвольные значения:

```
int a;      // a может иметь произвольное значение
int b[5];   // Элементы массива b могут иметь произвольные значения
Book c;     // Элементы структуры c могут иметь произвольные значения
```

- Обычные переменные инициализируются с помощью знака равенства (=).

```
int a = 10;
```

- Массивы и структуры инициализируются с помощью знака равенства и фигурных скобок:

```
int b[5] = {10, 20, 30, 40, 50};
Book c = {"Harry potter", 1000};
```

- Инициализация происходит путём простого задания байтов объекта соответствующими значениями.

Инициализация в языке C++

В C++ инициализация объектов усложняется из-за того, что в языке появляются классы, а объекты классов обычно инициализируются с вызовом конструктора. Для того чтобы разобраться как происходит инициализация объектов разных типов разделим типы C++ на следующие группы:

1. **Скалярные типы** – это все числовые типы (как целочисленные, так и типы с плавающей точкой), булевый тип, указатели, а также перечисления.
2. **Агрегатные типы** – к этой группе относятся:
   - Массивы
   - Агрегатные классы. Это классы у которых нет ничего из перечисленного:
     (a) конструкторов (за исключением тех, что создаются автоматически)
     (b) приватных полей
     (c) виртуальных функций (виртуальных функций будут пройдены в курсе позже)
3. **Нормальные классы** – к этой группе относятся классы, которые не являются агрегатными.

Такое разделение типов на группы позволяет лучше понять, что происходит при инициализации объектов, так как типы из разных групп инициализируются по-разному.

- Скаляры инициализируются посредством побайтового задания соответствующих значений.
- Нормальные классы инициализируются посредством вызова соответствующего конструктора.
- Агрегаты инициализируются поэлементно. Причём элементом объекта агрегатного типа может быть как скаляр, так и класс или другой агрегат. Для инициализации элементов агрегата применяются те же правила. То есть, если элемент агрегата является скаляром, то он инициализируются побайтово, если элемент агрегата является нормальным классом – то через конструктор, а если элемент агрегата является другим агрегатом, то поэлементно.

Можно выделить 4 разных вида инициализации: *default-*, *value-*, *direct-* и *copy-инициализация*. Разные виды инициализации различаются синтаксисом, который вызывает ту или иную инициализацию, а так же тем, что происходит во время этой инициализации. Инициализация может отличаться в зависимости от вида инициализации и инициализируемого типа. Можно составить следующую таблицу:

	Синтаксис	Скаляры	Агрегаты	Норм. Классы
Default initialization	`T x;`	не инициализируются	*Default initialization* для всех элементов	Конструктор по умолчанию
Value initialization	`T();` `T x{};`	инициализируются нулями	*Value initialization* для всех элементов	Конструктор по умолчанию

Direct initialization	`T x(a);`
	`T x{a};`
	`T x(a, b, ...);`
	`T x{a, b, ...};`
		инициализируются соответствующими значениями	*Copy initialization* для всех элементов	Соответствующий конструктор

Copy initialization	`T x = a;`
	`T x = {a};`
	`T x = {a, b, ...};`
	`func(a);`
	`return a;`
	`func({a, b, ...});`
	`return {a, b, ...};`
		инициализируются соответствующими значениями	*Copy initialization* для всех элементов	Соответствующий не `explicit` конструктор

Следует отметить, что эта таблица является неполной, существует ещё варианты синтаксиса когда вызывается та или иная инициализация. Но, тем не менее, эта таблица охватывает большинство случаев инициализации объектов в C++. Также, нужно отметить, что всё вышеперечисленное не относится к ссылкам. Ссылки инициализируются по другим правилам, которые, правда, являются интуитивно понятными.

К сожалению, в некоторых случаях инициализация с использованием круглых скобок не будет работать, поскольку компилятор может испытывать трудности в различении между инициализацией и объявлением функции.

Примеры инициализации объектов в C++

Примеры инициализации объекта скалярного типа `int`:

```
int a;          // Default инициализация - a будет иметь произвольное значение
int b{};        // Value инициализация - b будет равно нулю
int c(10);      // Direct инициализация - c будет равно 10
int d{10};      // Direct инициализация - d будет равно 10
int e = 10;     // Copy инициализация - e будет равно 10
int f = int();  // int() - создаём временный объект типа int с помощью value инициализации,
                // затем инициализируем объект a, используя copy инициализацию
                // f будет равен нулю
```

Примеры инициализации нормального класса `Cat`. Предположим, что у класса `Cat` есть конструктор по умолчанию и конструктор, принимающий один аргумент типа `int`.

```
Cat a;          // Default инициализация - вызовется конструктор по умолчанию
Cat b{};        // Value инициализация - вызовется конструктор по умолчанию
Cat c(10);      // Direct инициализация - вызовется соответствующий конструктор
Cat d{10};      // Direct инициализация - вызовется соответствующий конструктор
Cat e = 10;     // Copy инициализация - вызовется соответствующий конструктор, но только если
                // этот конструктор НЕ является explicit. Если же соответствующий конструктор
                // является explicit, то будет ошибка компиляции
```

`explicit` конструкторы

Любой конструктор можно пометить как `explicit` (`explicit` можно перевести как *явный*). Такие конструкторы нельзя использовать для copy-инициализации.

```
#include <iostream>
#include <string>

struct Cat
{
    int x;
    explicit Cat(int x)
    {
        std::cout << "Cat Constructor" << std::endl;
        this->x = x;
    }
};

void func(Cat c) {}

int main()
{
    Cat a(10);  // OK, explicit конструктор можно использовать для direct-инициализации
    Cat b = 10; // Ошибка, explicit конструктор нельзя использовать для copy-инициализации
    func(10);   // Ошибка, explicit конструктор нельзя использовать для copy-инициализации
}
```

Инициализация полей классов

Объекты нормальных классов инициализируются с использованием одного из конструкторов. Но нужно уточнить, что перед вызовом непосредственно конструктора класса, должны быть инициализированы все его поля. Это должно быть понятно, ведь поля можно использовать внутри конструктора класса. Использовать неинициализированные объекты нельзя, так что все поля класса должны быть инициализировани ещё *до входа в конструктор*.

```
#include <iostream>
#include <string>

class Cat
{
private:
    std::string name;
public:
    Cat()
    {
        // Тут уже можно использовать поле name. Значит name уже инициализировано.
        name.push_back('A');
        std::cout << name << std::endl;
    }
};

int main()
{
    Cat c;  // Сначала будет инициализировано поле name, а потом вызовется конструктор
}
```

Инициализация полей при объявлении внутри класса

Как инициализируются поля класса? По умолчанию поля класса инициализируются с помощью default инициализации. Но вид инициализации можно задать прямо внутри класса. Правда, это как раз один из тех случаев, когда компилятор не может отличить инициализацию с использованием круглых скобок и объявление функции. То есть в этом случаи можно инициализировать только через знак равенства или фигурные скобки.

```
class Cat
{
private:
    int x;      // Default инициализация
    int y{};    // Value инициализация
    int z{10};  // Direct инициализация
    int w = 10; // Copy инициализация
public:
    Cat()
    {
        std::cout << x << std::endl;  // Напечатает произвольное значение
        std::cout << y << std::endl;  // Напечатает 0
        std::cout << z << std::endl;  // Напечатает 10
        std::cout << w << std::endl;  // Напечатает 10
    }
};
```

Списки инициализации полей класса

Другой способ инициализировать поля класса – это использование так называемых списков инициализации. В этом случае, список инициализации следует за двоеточием после объявления конструктора:

```
class Cat
{
private:
    int age;
    std::string name;
public:
    Cat() : name("Alice"), age(10)
    {
        std::cout << name << " " << age << std::endl;  // Напечатает Alice 10
    }

    Cat(std::string x, int y) : name(x), age(y)
    {
        std::cout << name << " " << age << std::endl;  // Напечатает значения,
                                                       // переданные в конструктор
    }
};
```

Списки инициализации – это единственный способ инициализировать поле, если его значение должно зависить от аргументов конструктора. Могут возникнуть следующие вопросы:

- Что если поле инициализировано при объявлении внутри класса и в списке инициализации? Какое из значений будет выбрано для инициализации класса? Ответ – будет выбрано значение из списка инициализации.
- В каком порядке происходит инициализация полей класса: в том порядке, в котором они объявлены в классе, или в порядке, заданном в списке инициализации? Ответ – в том порядке, в котором они объявлены в классе. Например, в примере выше, всегда сначала будет инициализироваться поле `age`, а потом поле `name`, даже несмотря но то, что в списках инициализации они идут в обратном порядке.

Особые методы

Особые методы класса (англ. special member functions) – это методы, которые будут автоматически созданы компилятором, если они не были объявлены программистом. То есть, даже если вы не написали у класса такой метод, он всё-равно будет сгенерирован автоматически.
В C++ есть всего 6 особых методов. Рассмотрим их на примере класса `Cat`.

1. Конструктор по умолчанию: `Cat()`
   Если вы не напишите ни один конструктор в классе, то компилятор сам сгенирирует конструктор по умолчанию. Этот сгенерированный конструктор по умолчанию будет эквивалентен пустому конструктору по умолчанию.

2. Конструктор копирования: `Cat(const Cat& x)`
   Если вы не напишите конструктор копирования, то компилятор сам сгенирирует конструктор копирования. Этот сгенерированный конструктор копирования будет поэлементно копировать все поля класса.

3. Оператор присваивания: `Cat& operator=(const Cat& x)`
   Если вы не напишите оператор присваивания, то компилятор сам сгенирирует оператор присваивания. Этот сгенерированный оператор присваивания будет поэлементно присваивать все поля класса.

4. Деструктор: `~Cat()`
   Если вы не напишите деструктор, то компилятор сам сгенирирует деструктор. Этот сгенерированный деструктор будет эквивалентен пустому деструктору.

Если вы написали такой класс:

```
class Cat
{
private:
    std::string name;
    int age;

public:
    Cat(std::string name, int age)
        : name(name), age(age) {}
};
```

Компилятор будет считать, что класс выглядит так:

```
class Cat
{
private:
    std::string name;
    int age;

public:
    Cat(std::string name, int age)
        : name(name), age(age) {}

    // Следующие методы будут
    // сгенерированы автоматически:
    Cat(const Cat& x)
        : name(x.name), age(x.age) {}

    Cat& operator=(const Cat& x)
    {
        name = x.name;
        age = x.age;
    }

    ~Cat() {}
};
```

Обратите внимание, что в данном примере конструктор по умолчанию не был сгенерирован, так как у класса изначально был написан конструктор. Конструктор по умолчанию генерируется только тогда, когда не был написан *ни один* конструктор.

Пример создания особых методов компилятором

Рассмотрим пример, когда компилятор автоматически создаёт особые методы. Предположим, что мы создали класс и написали в нём один (не особый) конструктор. В этом случае компилятор автоматически сгенерирует конструктор копирования, оператор присваивания и деструктор. Конструктор по умолчанию автоматически сгенерирован не будет, так как в классе был написан один конструктор.

```
#include <iostream>
#include <string>

class Cat
{
private:
    std::string name;
    int age;

public:
    Cat(std::string name, int age)
        : name(name), age(age) {}
};

int main()
{
    Cat a{"Alice", 10};  // OK, вызовется написанный нами конструктор
    Cat b;               // Ошибка, конструктора по умолчанию нет и он не был сгенерирован
    Cat c{a};            // OK, конструктор копирования мы не написали, но он был сгенерирован
                         // компилятором автоматически.

    Cat d{"Bob", 20};    // OK, вызовется написанный нами конструктор
    d = a;               // OK, оператор присваивания мы не написали, но он был сгенерирован,
                         // компилятором автоматически.
}
```

Удалённые функции

Любую функцию или метод можно сделать удалёнными с помощью ключевого слова `delete`. При попытке вызова такой функции произойдёт ошибка компиляции. Такие функции можно, например, применять, чтобы запретить один из вариантов перегрузки:

```
#include <iostream>

void func(int x)  {std::cout << "Int"  << std::endl;}
void func(double x) = delete;

int main()
{
    func(10);   // OK, напечатает Int
    func(1.5);  // Ошибка компиляции, если бы удалённой функции с параметром double не было бы,
                // то напечатал бы Int
}
```

Удалённые функции можно использовать, чтобы запретить компилятору создавать те или иные особые методы. Например, можно удалить конструктор копирования, что предотвратит автоматическую генерацию этого конструктора компилятором. В результате получится класс, объекты которого нельзя будет копировать. Может показаться, что такая возможность никогда не пригодиться, но это не так. На самом деле, в стандартной библиотеке C++ и в других библиотеках есть множество классов, объекты которых нельзя копировать.

```
#include <iostream>
#include <string>

class Cat
{
private:
    std::string name;
    int age;

public:
    Cat(std::string name, int age)
        : name(name), age(age) {}

    Cat(const Cat& c) = delete;
};

void func(Cat c) {}

int main()
{
    Cat a{"Alice", 10};  // OK, вызовется написанный нами конструктор
    Cat b{a};            // Ошибка, конструктор копирования удалён
    Cat c = a;           // Ошибка, конструктор копирования удалён
    func(a);             // Ошибка, конструктор копирования удалён
}
```

default-методы

В противоположность удаления особых методов, можно указать компилятору, что мы хотим генерировать те или иные особые методы автоматически. Для этого используется ключевое слово `default`.

```
class Cat
{
private:
    std::string name;
    int age;

public:
    Cat(std::string name, int age)
        : name(name), age(age) {}

    Cat() = default;              // Просим компилятор сгенерировать этот конструктор
    Cat(const Cat& c) = default;  // Просим компилятор сгенерировать этот конструктор.
                                  // Хотя он и так бы сгенерировал его.
};
```

В примере выше строка `Cat(const Cat& c) = default;` не изменяет поведение программы, так как конструктор копирования всё равно бы был сгенерирован автоматически, даже без этой строки. Но всё-равно прописывать такие `default` особые методы полезно, так как это делает код более ясным.

Создание объектов в куче

Операторы `new`/`delete`

Для выделения/освобождения памяти в куче в языке C используются функции `malloc` и `free`. Эти функции можно использовать и в языке C++ для выделения памяти. Но эти функции не очень подходят для создания и удаления объектов в куче. Ведь `malloc` просто выделяет память, но не инициализирует объект, и, следовательно, не вызывает конструктор объекта. А использование объекта нормального класса без вызова конструктора приведёт к UB. Аналогично, функция `free` просто освобождает память, но не вызывает деструктор.

```
#include <cstdlib>
#include <string>

int main()
{
    std::string* p = (std::string*)std::malloc(sizeof(std::string));
    *p = "Cat";    // UB, так как объект std::string в куче не был инициализован
    std::free(p);
}
```

Вместо использования функций `malloc`/`free` для создания объектов в куче следует использовать операторы `new`/`delete`. С помощью оператора `new` можно создать объект в куче и корректно его инициализировать. Тогда как с помощью оператора `delete` можно корректно удалить объект в куче.

- Оператор `new` делает следующее:
    1. Выделяет необходимое для объекта количество памяти в куче. Для этого внутри себя `new` вызывает функцию `malloc`.
    2. Инициализирует объект. Если объект является нормальным классом, то вызывает соответствующий конструктор.
- Оператор `delete` делает следующее:
    1. Деинициализирует объект. Вызывает деструктор, если объект является классом.
    2. Освобождает выделенную память. Для этого внутри себя `delete` вызывает функцию `free`.

Пример выше с использованием `new`/`delete` перепишется так:

```
#include <string>
int main()
{
    std::string* p = new std::string;  // Выделит память и вызовет конструктор по умолчанию
    *p = "Cat";                        // OK, объект std::string в куче инициализован
    delete p;                          // Вызовет деструктор и освободит память.
}
```

Синтаксис разных видов инициализации работает и при использовании оператора `new`. За исключением того, что в этом случае не используется copy инициализация.

```
int main()
{
    int* pa = new int;      // Default инициализация, *pa будет иметь произвольное значение
    int* pb = new int{};    // Value инициализация, *pb будет равно нулю
    int* pc = new int(10);  // Direct инициализация, *pc будет равно 10
    int* pd = new int{10};  // Direct инициализация, *pd будет равно 10

    delete pa; delete pb; delete pc; delete pd;
}
```

Операторы `new[]/delete[]`

Помимо операторов `new/delete` в языке также существуют операторы `new[]/delete[]`, которые нужны для создания массива объектов в куче.

```
int main()
{
    int* pa = new int[5];     // Выделяем 5 элементов в куче и default-инициализируем их
                              // Элементы будут иметь произвольные значения

    int* pb = new int[5]{};   // Выделяем 5 элементов в куче и value-инициализируем их
                              // Элементы будут равны нулю


    int* pc = new int[5]{10, 20, 30, 40, 50}; // Выделяем 5 элементов в куче и
                                              // direct-инициализируем их значениями


    delete[] pa;
    delete[] pb;
    delete[] pc;
}
```

Интересно отметить, что так как default инициализация не инициализирует объекты скалярных типов, то операторы `new` или `new[]` в случаи скалярных типов с инициализацией по умолчанию будут просто выделять память, то есть делать то же самое, что и `malloc`.

```
#include <cstdlib>
int main()
{
    // В данном случае операторы new/delete делают то же самое, что и аналогичные malloc/free
    int* pa = new int[5];
    delete[] pa;

    int* pb = (int*)std::malloc(5 * sizeof(int));
    std::free(pb);
}
```

Важно не путать пары операторов `new/delete` и `new[]/delete[]`. Первая пара используется для создания/удаления одного объекта в куче, а вторая пара - для создания/удаления массива объектов. Для удаления объекта, созданного с помощью `new` нужно использовать только `delete`, а для удаления массива объектов нужно использовать только `delete[]`. Если перепутаете, то в программе будет UB.

```
int main()
{
    int* pa = new int[5];  // Создаём массив объектов, используя оператор new[]
    delete pa;             // UB. Так как используем оператор delete,
                           // а нужно было использовать delete[].

}
```

Оператор `placement new`

Статические поля и методы

Статическое поле – это поле класса, которое принадлежит самому классу, а не конкретному объекту этого класса. Такие поля не хранятся в объектах класса. Вместо этого хранится единственная копия этого поля в сегменте памяти данные. Получить доступ к этому полю можно либо через имя класса с использованием оператора `::` либо через объект с использованием оператора точка.

```
#include <iostream>
struct Cat
{
    static int x;

    Cat()
    {
        x += 1;
    }
};

int Cat::x = 10;  // Инициализируем статическое поле.

int main()
{
    Cat a;
    Cat b;
    Cat c;

    std::cout << Cat::x << std::endl;  // Напечатает 13
    std::cout << a.x << std::endl;     // Напечатает 13
}
```

Инициализация статических полей класса должна проводиться вне самого класса. Почему статические поля нельзя инициализировать внутри класса? Это более сложный вопрос который мы оставим на следующие семинары. Пока только можно отметить, что если нужно константное статическое поле, то такое поле можно инициализировать внутри класса если добавить ключевое слово `constexpr`.

```
#include <iostream>
struct Cat
{
    static constexpr int x = 10;  // Сразу инициализируем
};

int main()
{
    Cat a;

    std::cout << Cat::x << std::endl;  // Напечатает 10
    std::cout << a.x << std::endl;     // Напечатает 10
}
```

Статические методы – это методы, которые принадлежат самому классу, а не конкретному объекту этого класса. Такие методы не могут использовать нестатические поля и методы класса. Вызвать такой метод можно либо через имя класса с использованием оператора `::` либо через объект с использованием оператора точка.

```
#include <iostream>
struct Cat
{
    static void say()
    {
        std::cout << "Hello" << std::endl;
    }
};
int main()
{
    Cat a;

    Cat::say();  // Напечатает Hello
    a.say();     // Напечатает Hello
}
```

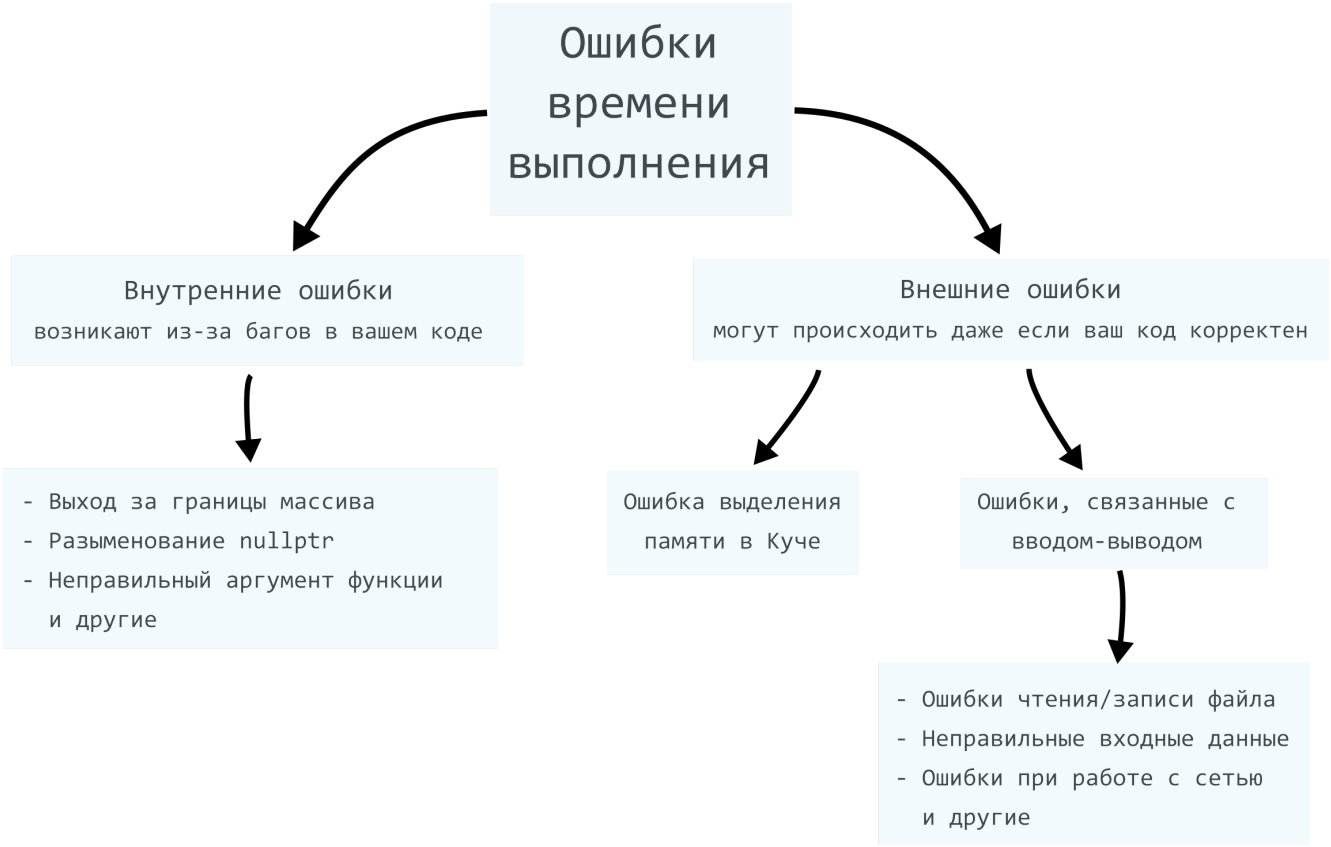
Copy elision

Обработка ошибок

Есть множество различных классификаций ошибок, здесь представлена одна из них, которая может отличаться от классификаций в других источниках. Ошибки делятся на:

- Ошибки времени компиляции
- Ошибки линковки
- Ошибки времени выполнения
- Логические ошибки (ошибки в логике работы программы; могут привести или не привести к ошибке времени выполнения)

Под обработкой ошибок обычно понимается обработка ошибок времени выполнения. Их, в свою очередь, можно тоже разделить на два типа: внутренние и внешние.



Эти два типа ошибок времени выполнения сильно различаются друг от друга и могут обрабатываться различными способами.

- От внутренних ошибок, как правило, нельзя восстановиться. Лучшее решение в этом случае – это завершить выполнение программы и вывести сообщение об ошибке.
- Внешние ошибки часто можно предусмотреть и продумать действия программы при возникновении такой ошибки. Иногда можно даже восстановить работу программы.

Методы обработки ошибок

assert

`assert` обычно используется для отладки в Debug режиме, но иногда его применяют и для примитивной отладки ошибок. В случае, если условие в `assert` не выполняется, вызывает функцию `std::abort` и завершает программу. Нет возможности как-то обработать ошибку и продолжить выполнение программы.

```
#include <iostream>
#include <cmath>
#include <cassert>
float geometricMean(float a, float b)
{
    assert(a >= 0 && b >= 0);
    return std::sqrt(a * b);
}

int main()
{
    std::cout << geometricMean(-2, 2) << std::endl;
    std::cout << "After" << std::endl;
}
```

Выполнение данной программы напечатает на экран:

```
Assertion failed: a >= 0 && b >= 0, file test2.cpp, line 7
```

Использование глобальной переменной

Обработка ошибок через глобальную переменную `errno` — простой метод, широко применявшийся в языке C. При возникновении ошибки ей присваивается код (например, `EDOM`, `EPERM` и другие), который затем проверяется после вызова функции. Сегодня такой подход считается устаревшим.

```
#include <iostream>
#include <cmath>
#include <cerrno>
float geometricMean(float a, float b)
{
    if (a < 0 || b < 0)
    {
        errno = EDOM;
        return 0;
    }
    return std::sqrt(a * b);
}

int main()
{
    errno = 0;
    float result = geometricMean(-2, 2);
    if (errno == EDOM)
    {
        std::cout << "Error. Domain error!" << std::endl;
        std::exit(1);
    }
    std::cout << "Result = " << result << std::endl;
}
```

Коды возврата

Простой и широко используемый способ обработки ошибок – возврат заранее оговоренного значения из функции при возникновении ошибки. После вызова функцию, которая использует коды возврата, необходимо проверить на возвращаемое значение. Сложность может возникнуть, если функция уже возвращает значение. Тогда придётся передавать значение или код возврата каким-то другим способом, например через аргументы по ссылке.

```
#include <iostream>
#include <cmath>

int geometricMean(float a, float b, float& result)
{
    if (a < 0 || b < 0)
        return -1;
    result = std::sqrt(a * b);
    return 0;
}

int main()
{
    float result;
    int code = geometricMean(-2, 2, result);
    if (code == -1)
        std::cout << "Error!" << std::endl;
    else
        std::cout << "Result = " << result << std::endl;
}
```

Другой способ, это использование специального объекта для хранения и результата и кода ошибки:

```
#include <iostream>
#include <cmath>

struct Result
{
    float value;
    int code;
};

Result geometricMean(float a, float b)
{
    if (a < 0 || b < 0)
        return {0.0, -1};
    return {std::sqrt(a * b), 0};
}

int main()
{
    Result res = geometricMean(-2, 2);
    if (res.code == -1)
        std::cout << "Error!" << std::endl;
    else
        std::cout << "Result = " << res.value << std::endl;
}
```

Коды возврата часто применяются как в языке C (можно вспомнить, например, функцию `malloc`), так и в языке C++. Но в C++ для работы с кодами возврата часто используются вспомогательные классы, такие как `std::optional` и `std::expected`.

Недостатки кодов возврата

В целом, коды возврата это хороший способ обработки ошибок, но у него есть несколько недостатков:

- Самый главный недостаток кодов возврата заключается в том, что для корректной обработки ошибок необходимо делать проверку кода возврата *после каждого вызова функции*. В результате весь код превращается в "лапшу" и проверок. Такой код сложно писать и поддерживать.
- Если ошибка возникает глубоко в коде, а обрабатываться должна выше, её приходится "прокидывать" через все вложенные функции. Что также усложняет написание и поддержку кода.
- Нельзя обработать ошибки в некоторых функциях, таких как конструкторы, деструкторы или перегруженные операторы.

Исключения

Исключения – это основной механизм обработки ошибок в C++. Использование исключений позволяет отделить код обнаружения ошибки от её обработки.

```
#include <iostream>
#include <cmath>
#include <stdexcept>

float geometricMean(float a, float b)
{
    if (a < 0 || b < 0)
        throw std::runtime_error{"Domain error"};
    return std::sqrt(a * b);
}

int main()
{
    try
    {
        std::cout << geometricMean(-2, 2) << std::endl;
    }
    catch (std::runtime_error& e)
    {
        std::cout << "Error. " << e.what() << std::endl;
    }
}
```

- При обнаружении исключения "бросаем" объект, используя ключевое слово `throw`. В качестве бросаемого объекта может быть объект практически любого типа, но обычно используются специальные типы из библиотеки `stdexcept`, такие как, например, `std::runtime_error`.
- Код, в котором могут возникнуть ошибки помещаем в блок `try`.
- Если внутри `try` (включая вызванные из него функции) будет выброшено исключение, и существует блок `catch` с совместимым типом, управление передаётся в этот `catch`. При переходе объекты, созданные в `try`, корректно уничтожаются.
- Одному блоку `try` могут соответствовать несколько блоков `catch`, для обработки нескольких типов ошибок.
- Метод `what` класса `std::runtime_error` просто возвращает C-строку (`const char*`), содержащую описание ошибки, заданное при создании объекта. В данном случае это строка `"Domain error"`.

Основы обработки ошибок с помощью исключений

Несколько `catch`-блоков

Бросать можно объект практически любого типа. При этом тип объекта в блоке `catch` должен быть совместимым с типом бросаемого объекта, а именно:

- Тип в `catch` должен полностью совпадать с типом бросаемого объекта.
- Тип в `catch` должен быть ссылкой/константной ссылкой на тип бросаемого объекта.
- Тип в `catch` должен быть ссылкой на базовый класс для типа бросаемого объекта. (более подробно этот случай будет разобран в теме "Наследование")

Одному блоку `try` могут соответствовать несколько блоков `catch`. Для обработки ошибки будет выбран блок `catch` с совместимым типом. Если подходят несколько блоков, то будет выбран первый `catch` с совместимым типом.

```
#include <iostream>
int main()
{
    try
    {
        throw 123;
    }
    catch (float x)
    {
        std::cout << "Float: " << x << std::endl;
    }
    catch (long long x)
    {
        std::cout << "Long Long: " << x << std::endl;
    }
    catch (const int& x)
    {
        std::cout << "Const Int Ref: " << x << std::endl;
    }
    catch (int x)
    {
        std::cout << "Int: " << x << std::endl;
    }
}
```

В данной программе мы бросаем объект типа `int` со значением `123`. Блоки `catch` с типами `float` и `long long` не подойдут. Блоки с типами `const int&` и `int` подходят, но среди них будет выбран первый. В результате программа напечатает:

```
Const Int Ref: 123
```

Обработчик `catch` для любого типа

Есть специальный обработчик исключений `catch(...)`, который может перехватывать абсолютно любые типы исключений. Перехватывает исключения любого типа, но с его помощью нельзя получить сам бросаемый объект.

```
catch (...)
{
    std::cout << "Any Type" << std::endl;
}
```

Что происходит при бросании объекта

Класс `Verbose` просто печатает сообщение при вызове особого метода. Его можно найти в `seminar3_initialization/theory/code/verbose.hpp`.

```
#include <iostream>
#include "verbose.hpp"
int main()
{
    try
    {
        Verbose a{"Alice"};
        throw a;
    }
    catch (Verbose x)
    {
        std::cout << x.getName() << std::endl;
    }
}
```

Что напечатает данная программа? А если ловить не по значению, а по ссылке?

Раскрутка стека

Исключения в деструкторах

Библиотечные классы для исключений

`std::exception`, `std::runtime_error` и другие.

Исключения в языке C++. Оператор `new`.

Далеко не для всех типов ошибок в C++ используются исключения. Деление на 0. Выход за границы массива, вектора. Но используется для `new`. `std::bad_alloc`. `std::nothrow`.

Исключения в стандартной библиотеке C++

Исключения в `std::string` и `std::vector`.

- `std::bad_alloc`
- `std::out_of_range`
- `std::invalid_argument`
- `std::domain_error`

Недостатки исключений

- Производительность
- Усложнение программы
- Возможность утечки памяти/ресурсов

Кратко об использовании исключений при написании своих классов

Гарантии безопасности. Copy and swap.