



Lesson 7

Перегрузка операторов C++.



Введение.

Перегрузка операторов в C++ позволяет создавать новые смыслы для стандартных операторов или создавать пользовательские операторы для типов данных. Это мощный инструмент, который улучшает читаемость и понимание кода, делает его более выразительным и интуитивно понятным. В этой лекции мы рассмотрим основные концепции перегрузки операторов, правила перегрузки, а также примеры использования.



Плюсы.

- **Увеличение выразительности кода:** Позволяет использовать стандартные операторы с пользовательскими типами данных, что делает код более лаконичным и понятным.
- **Удобство использования:** Пользователи вашего класса смогут использовать привычные операторы для работы с объектами этого класса, что упрощает их взаимодействие с кодом.
- **Создание интуитивных интерфейсов:** Перегрузка операторов позволяет создавать интуитивно понятные интерфейсы для пользовательских типов данных, делая код более читаемым и легким для использования.
- **Увеличение производительности:** Перегрузка операторов может улучшить производительность кода, поскольку нативные операторы могут быть эффективно оптимизированы компилятором.



Минусы.

- **Потеря ясности:** Перегрузка операторов может привести к потере ясности кода, если перегруженные операторы не используются там, где ожидается их стандартное поведение.
- **Нарушение привычной семантики:** Некоторые перегруженные операторы могут создавать неочевидное поведение, что может привести к ошибкам в использовании или недопониманию.
- **Увеличение сложности кода:** Перегрузка операторов может привести к увеличению сложности кода, особенно если операторы перегружаются с неочевидным поведением или если перегружено большое количество операторов.
- **Ограниченная возможность перегрузки:** Некоторые операторы не могут быть перегружены или могут быть перегружены только в определенном контексте, что может ограничить гибкость использования перегрузки операторов.



Концепции.

Перегрузка операторов позволяет использовать стандартные операторы для пользовательских типов данных или изменить поведение стандартных операторов для встроенных типов. Она выполняется путем создания специальных функций для определенных операторов, которые называются функциями перегрузки операторов.



Правила перегрузки.


- **Синтаксис функции перегрузки оператора:** Функции перегрузки операторов имеют особый синтаксис. Например, оператор сложения для пользовательского типа MyClass будет иметь вид **MyClass operator+(const MyClass& obj).**
- **Типы аргументов:** В зависимости от оператора, который вы перегружаете, количество и типы его аргументов могут варьироваться.
- **Возвращаемое значение:** Обычно функции перегрузки операторов возвращают значение, результат операции.
- **Права доступа:** Определенные операторы могут быть перегружены только как методы класса или только как глобальные функции. Например, операторы << и >> для потоков перегружаются как глобальные функции, в то время как операторы присваивания и индексирования могут быть перегружены как методы класса.



Пример класса.

```
class MyNumber {  
    private:  
        int value;  
  
    public:  
        MyNumber(int val) : value(val) {}  
        ...  
}
```

код определяет перегрузку оператора + и -, который позволяет складывать два объекта типа MyNumber. Функция принимает в качестве аргумента ссылку на другой объект MyNumber (const MyNumber& other).



```
// Перегрузка оператора сложения
MyNumber operator+(const MyNumber& other) const {
    return MyNumber(value + other.value);
}

// Перегрузка оператора вычитания
MyNumber operator-(const MyNumber& other) const {
    return MyNumber(value - other.value);
}
```





```
// Перегрузка оператора умножения
```

```
MyNumber operator*(const MyNumber& other) const {  
    return MyNumber(value * other.value);  
}
```

```
// Перегрузка оператора деления
```

```
MyNumber operator/(const MyNumber& other) const {  
    if (other.value == 0) {  
        std::cerr << "Error: Division by zero" << std::endl;  
        return MyNumber(0);  
    }  
    return MyNumber(value / other.value);  
}
```



```
// Перегрузка оператора инкремента (префиксный) ++a
MyNumber& operator++() {
    ++value;
    return *this;
}
```

```
// Перегрузка оператора декремента (префиксный) --a
MyNumber& operator--() {
    --value;
    return *this;
}
```



```
// Перегрузка оператора инкремента (постфиксный)
```

```
MyNumber operator++(int) {  
    MyNumber temp(value);  
    ++value;  
    return temp;  
}
```

```
// Перегрузка оператора декремента (постфиксный)
```

```
MyNumber operator--(int) {  
    MyNumber temp(value);  
    --value;  
    return temp;  
}
```



```
// Перегрузка оператора вывода в поток
```

```
friend std::ostream& operator<<(std::ostream& os, const MyNumber& num) {  
    os << num.value;  
    return os;  
}
```

```
// Перегрузка оператора ввода из потока
```

```
friend std::istream& operator>>(std::istream& is, MyNumber& num) {  
    is >> num.value;  
    return is;  
}
```



```
// Перегрузка оператора равенства
```

```
bool operator==(const MyNumber& other) const {  
    return value == other.value;  
}
```

```
// Перегрузка оператора неравенства
```

```
bool operator!=(const MyNumber& other) const {  
    return !(*this == other);  
}
```



Исключения и обработка ошибок.

исключения (exceptions) - это механизм, который позволяет обрабатывать ошибки и необычные ситуации в программе. Когда происходит ошибка, например, деление на ноль или ошибка ввода-вывода, программа может сгенерировать исключение. Это исключение затем может быть перехвачено (обработано) в коде программы. **Механизм исключений в C++** позволяет разделить код обработки ошибок от основной логики программы. Это делает код более чистым и понятным, так как основной код программы не будет перегружен проверками на ошибки.

Исключения создаются с использованием ключевого слова **throw**, а обработка исключений выполняется с помощью блока **try-catch**.

*нет ограничений на количество блоков **catch**, которые можно использовать для одного блока **try**. Вы можете писать столько блоков **catch**, сколько необходимо для обработки различных типов исключений. И **catch (...)** с тремя точками - способен перехватить все исключения выше!*



спецификатор noexcept

спецификатор используется для указания, что функция не выбрасывает исключений. Если функция, помеченная как noexcept, выбросит исключение, программа завершится с ошибкой.

```
void foo() noexcept {  
    // Функция, которая не выбрасывает исключений  
}
```

std::terminate - функция вызывается, когда выбрасывается исключение из функции, объявленной с noexcept, или если происходит исключение, которое не перехватывается ни одним блоком catch. То есть если исключение не перехватывается ни одним блоком catch, то программа завершится вызовом std::terminate. Также может быть использована в конструкторе и деструкторе класса.



```
#include <iostream>
```


```
void processInput(int x) {  
    if (x == 0) {  
        throw "Division by zero error";  
    }  
    int result = 100 / x;  
    std::cout << "Result: " << result << std::endl;  
}
```

```
int main() {  
    try {  
        processInput(0);  
    } catch (const char* errorMessage) {  
        std::cerr << "Error: " << errorMessage << std::endl;  
    }  
    return 0;  
}
```




Классы исключений (встроенные)

- **std::logic_error**: класс используется для исключений, связанных с ошибками логики программы, которые могут быть обнаружены на этапе выполнения, но не являются ошибками времени выполнения. К ним относятся, например, ошибки проверки предусловий и инвариантов.
- **std::runtime_error**: класс используется для исключений, связанных с ошибками времени выполнения, которые невозможно обнаружить на этапе компиляции. К ним относятся, например, ошибки ввода-вывода, деление на ноль и другие.
- **std::invalid_argument**: Исключение, которое выбрасывается при передаче функции недопустимого аргумента.
- **std::out_of_range**: Исключение, которое указывает на то, что значение находится за пределами допустимого диапазона.
- **std::bad_alloc**: Исключение, которое возникает при нехватке памяти при выделении оперативной памяти с помощью `new`.
- и др



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

double divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero error");
    }
    return static_cast<double>(a) / b;
}

int main() {
    try {
        std::cout << "Result: " << divide(10, 0) << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```



Собственный класс исключений

Собственный класс исключений — это пользовательский класс, предназначенный для обработки специфических ошибок, которые могут возникнуть в программе. Обычно такой класс наследуется от одного из стандартных классов исключений, таких как `std::exception` или его подклассы (например, `std::runtime_error` или `std::logic_error`). Собственный класс исключений позволяет разработчику создавать исключения с дополнительной информацией или функциональностью, специфичной для данной программы или библиотеки. Создание собственных классов исключений позволяет более точно определить и обрабатывать ошибки в определенном коде.

Основные компоненты собственного класса исключений:

1. **Наследование:** Собственный класс исключений наследует от стандартного класса исключений, чтобы обеспечить совместимость с существующей инфраструктурой обработки исключений в C++.
2. **Конструктор:** Конструктор собственного класса исключений обычно принимает сообщение об ошибке и передает его базовому классу, чтобы это сообщение могло быть получено методом `what()`.
3. **Дополнительная информация:** В собственный класс исключений могут быть добавлены дополнительные данные или методы, необходимые для более детальной диагностики и обработки ошибок.



Недостатки - Собственного класса исключений

- **Избыточность кода:** Создание собственного класса исключений для каждой возможной ошибки может привести к избыточности кода, особенно если ошибки относительно редки или их обработка не требует дополнительной информации.
- **Наследование от стандартных исключений:** Если собственный класс исключений наследуется от стандартного класса, это ограничивает возможности использования других стандартных исключений, поскольку множественное наследование исключений может усложнить логику обработки исключений.
- **Перехват и обработка (затруднение отладки):** Не всегда ясно, каким образом собственные классы исключений будут перехватываться и обрабатываться в различных частях программы, что может затруднить отладку и поддержку.
- **Усложнение интерфейса/кода:** Использование собственных классов исключений может привести к усложнению интерфейсов функций и методов, если они вынуждены поддерживать возможность генерации различных исключений.
- **Необходимость чёткой документации:** Каждый собственный класс исключений требует четкой документации, чтобы разработчики понимали, какие ошибки он предназначен для обработки и как правильно их использовать.



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

// собственный класс исключения
class MyException : public std::exception {
private:
    std::string errorMessage; // Сообщение об ошибке

public:
    // Конструктор с сообщением об ошибке
    MyException(const std::string& message) : errorMessage(message) {}

    // Метод, возвращающий описание ошибки
    const char* what() const noexcept override {
        return errorMessage.c_str();
    }
};
```



// Функция, которая может выбросить исключение

```
void processInput(int x) {  
    if (x == 0) {  
        throw MyException("Custom exception: Division by zero error");  
    }  
    int result = 100 / x;  
    std::cout << "Result: " << result << std::endl;  
}
```

```
int main() {  
    try {  
        processInput(0);  
    } catch (const MyException& e) {  
        std::cerr << "Error: " << e.what() << std::endl;  
    }  
    return 0;  
}
```



Задачи#1

- Напишите программу, которая запрашивает у пользователя два числа и выводит результат их деления. Обработайте исключение деления на ноль.
- Создайте класс, представляющий стек целых чисел. Реализуйте методы добавления элемента в стек и извлечения элемента из стека. Обработайте исключение при извлечении элемента из пустого стека.
- Напишите программу, которая читает данные из файла и выполняет какие-то операции с этими данными. Обработайте исключение при открытии файла.
- Реализуйте функцию, которая принимает на вход строку и пытается преобразовать ее в целое число. Обработайте исключение, если строка не может быть преобразована в число.
- Создайте класс для работы с матрицами. Реализуйте оператор умножения матриц. Обработайте исключение при попытке умножения матриц разных размеров.



Задачи#2


- Напишите программу, которая пытается открыть соединение с удаленным сервером. Обработайте исключение, если соединение не удалось установить.
- Разработайте функцию, которая считает факториал числа. Обработайте исключение, если переданное число отрицательное.
- Создайте класс для работы с геометрическими фигурами. Реализуйте метод вычисления площади фигуры. Обработайте исключение, если переданы некорректные данные (например, отрицательные значения длин сторон).
- Напишите программу, которая пытается открыть и прочитать содержимое несуществующего файла. Обработайте исключение, возникающее при попытке чтения файла, который не существует.
- Реализуйте функцию, которая делит одно число на другое. Обработайте исключение, если второе число равно нулю, и выведите сообщение об ошибке.



Вторичные типы данных;

- **size_t** - это тип данных, используемый для представления размеров объектов в памяти, обычно используемый для индексации массивов и вычисления размеров контейнеров. Он является беззнаковым целым типом данных, который обеспечивает портбельность кода между различными платформами, поскольку его размер зависит от архитектуры и компилятора, но всегда гарантировано, что этот тип данных может представлять размер любого объекта в памяти.
- **ptrdiff_t**: тип данных используется для представления разницы между двумя указателями. Он обеспечивает портбельность кода, так как размер этого типа данных зависит от архитектуры и компилятора, и обеспечивает возможность корректно представлять разницу между указателями на различных платформах.
- **intptr_t и uintptr_t**: типы данных представляют целые числа, достаточно большие для представления указателей на объекты в памяти и используются для выполнения арифметических операций с указателями. **intptr_t** - это знаковый тип данных, который способен хранить в себе значение указателя, а **uintptr_t** - беззнаковый тип данных, который может хранить адрес объекта в памяти.

size_t ПРИМЕР



```
#include <iostream>

int main() {
    const size_t arraySize = 5; // определение размера массива
    int myArray[arraySize]; // создание массива целых чисел

    // Заполнение массива
    for (size_t i = 0; i < arraySize; ++i) {
        myArray[i] = i * 2; // заполнение элементов массива удвоенными значениями индекса
    }

    // Вывод элементов массива
    std::cout << "Элементы массива: ";
    for (size_t i = 0; i < arraySize; ++i) {
        std::cout << myArray[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

ptrdiff_t ПРИМЕР

```
#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr1 = &arr[1]; // Указатель на второй элемент массива
    int *ptr2 = &arr[4]; // Указатель на последний элемент массива

    // Вычисляем разницу между указателями
    ptrdiff_t diff = ptr2 - ptr1;

    std::cout << "Разница между указателями: " << diff << std::endl;

    return 0;
}
```

В примере **ptrdiff_t** используется для хранения разницы между указателями **ptr1** и **ptr2**. Так как **ptrdiff_t** может хранить **отрицательные значения**, он идеально подходит для представления разницы между указателями, которые находятся в разных частях массива.

Разница между указателями **ptr1** и **ptr2** вычисляется как разность адресов, на которые указывают указатели.



Thank you for your attention!