Семинар #2: Классы и перегрузка операторов.

Классы

Объектно-ориентированное программирование (ООП) – это парадигма разработки программного обеспечения, основанная на использовании объектов, которые представляют собой экземпляры классов.

Класс – это пользовательский тип данных, который объединяет данные и методы. Классы в C++ объявляются с помощью ключевого слова `struct` или с помощью ключевого слова `class`.

Поля класса – это переменные, которые объявлены внутри класса.

Методы класса – это функции, которые определены внутри класса и могут оперировать с его полями.

Члены класса – это сущности, которые определены внутри класса. Поля класса и методы класса являются членами класса.

Свободные функции – это функции, которые не являются методами, то есть просто обычные функции.

Рассмотрим пример простого класса `Book`:

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

struct Book
{
    std::string title;
    int price;

    void makeDiscount(int discount)
    {
        if (price > discount)
            price -= discount;
        else
            price = 0;
    }
};

int main()
{
    Book a = {"Harry Potter", 500};
    a.makeDiscount(100);
    std::cout << a.price << std::endl;  // Напечатает 400
}
```

Данный класс обладает двумя полями (`title` и `price`) и одним методом (`makeDiscount`). В отличии от структур языка C, классы языка C++ могут содержать внутри себя функции, называемые методами. В функции `main` был создан объект класса `Book`, под названием `a`. Вызов метода у объекта осуществляется через специальный оператор точка:

```
a.makeDiscount(100);
```

Внутри метода `makeDiscount` можно пользоваться полями класса. И если мы вызываем метод от объекта `a`, то будут использоваться поля объекта `a`.

Инкапсуляция

Инкапсуляция – это размещение в классе полей и функций, которые работают с этими полями. Рассмотрим два разных класса `Book`. Первый класс не будет включать функцию для работы с его полями; такая функция будет находиться вне класса. Это подход, при котором класс используется без инкапсуляции. Второй класс, напротив, будет применять инкапсуляцию, и функция для работы с его полями будет реализована внутри класса.

Без инкапсуляции:

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

struct Book
{
    std::string title;
    int price;
};

void makeDiscount(Book& b, int discount)
{
    if (b.price > discount)
        b.price -= discount;
    else
        b.price = 0;
}

int main()
{
    Book a = {"Harry Potter", 500};
    makeDiscount(a, 100);
    std::cout << a.price << std::endl;
}
```

С инкапсуляцией:

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

struct Book
{
    std::string title;
    int price;

    void makeDiscount(int discount)
    {
        if (price > discount)
            price -= discount;
        else
            price = 0;
    }
};

int main()
{
    Book a = {"Harry Potter", 500};
    a.makeDiscount(100);
    std::cout << a.price << std::endl;
}
```

Константные методы класса

Константный метод класса – это метод, который не может изменить поля объекта, на котором он вызывается. Константные методы помечаются словом `const` при объявлении метода класса. Например, в следующем примере класс `Book` имеет константный метод `printTitle`:

```
struct Book
{
    std::string title;
    int price;

    void printTitle() const
    {
        std::cout << title << std::endl;
    }
};
```

Внутри класса `printTitle` нельзя изменять методы `title` и `price`. Попытка изменить поля внутри константного метода приведет к ошибке компиляции. Очень важно всегда указывать то, что метод является константным, если внутри него поля не изменяются.

Скрытие данных

Модификаторы доступа `public` и `private`

Модификаторы доступа служат для ограничения доступа к полям и методам класса.

- `public` – поля и методы могут использоваться где угодно
- `private` – поля и методы могут использовать только методы этого класса и друзья (особые функции и классы, объявленные с использованием ключевого слова `friend`)

Создадим класс под название `Alice`, который будет содержать приватное поле `x`, приватный метод `cat`, публичное поле `y` и публичный метод `dog`. Попытаемся использовать эти члены класса из другого метода класса и вне этого класса. Внутри класса можно использовать любые члены класса, а вне класса можно использовать только публичные члены класса.

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

struct Alice
{
private:
    int x;
    void cat() const
    {
        std::cout << "cat" << std::endl;
    }

public:
    int y;
    void dog() const
    {
        std::cout << "dog" << std::endl;
    }

    void test()
    {
        x = 10;  // OK, приватные поля можно использовать в самом классе
        cat();   // OK, приватные методы можно использовать в самом классе

        y = 10;  // OK, публичные поля можно использовать где угодно
        dog();   // OK, публичные методы можно использовать где угодно
    }
};

int main()
{
    Alice a;

    a.x = 10;  // Ошибка, нельзя использовать приватные поля вне класса
    a.fx();    // Ошибка, нельзя использовать приватные методы вне класса

    a.y = 20;  // OK, поле y публичное
    a.dog();   // OK, метод dog публичный
    a.test();  // OK, метод test публичный
}
```

Друзья

Друзья класса – это такие функции или классы, которые имеют доступ к приватным членам класса. Для обозначения того, что функция или класс является дружественной, используется ключевое слово `friend`:
В следующем примере был создан класс `Alice` у которого есть два друга: функция `func` и класс `Bob`. Мы можем вызывать приватный метод `cat` класса `Alice` из функции `func` и класса `Bob`, так как это друзья класса `Alice`.

```
#include <iostream>
struct Alice
{
private:
    void cat() const {std::cout << "cat" << std::endl;}
public:
    friend void func();  // Указываем, что функция func является другом класса Alice
    friend struct Bob;   // Указываем, что класс Bob является другом класса Alice
};

void func()
{
    Alice a;
    a.cat();  // Несмотря на то, что метод cat приватный, функция func имеет к нему доступ
}

struct Bob
{
    void test()
    {
        Alice a;
        a.cat();  // Несмотря на то, что метод cat приватный, класс Bob имеет к нему доступ
    }
};

int main()
{
    func();
    Bob b;
    b.test();
}
```

Дружественность не является симметричной. Если один класс является другом второго, то это не означает, что второй класс является другом первого. Например, в примере выше, `Bob` является другом `Alice`, но `Alice` не является другом `Bob`, поэтому `Alice` не имеет доступа к приватным полям и методам класса `Bob`.

Различие между определение класса с помощью `class` и `struct`

Классы в языке C++ можно создавать как с помощью ключевого слова `struct`, так и с помощью ключевого слова `class`. Есть ровно два отличия между этими двумя вариантами определения классов. Первое различие заключается в том, что у классов, созданных с использованием `struct`, все члены по умолчанию публичны, в то время как у классов, созданных с помощью `class`, члены по умолчанию являются приватными. Второе различие, связано с типом наследования по умолчанию, мы пройдём его позже.

```
struct Alice
{
    int x; // Это публичное поле
};
```

```
class Alice
{
    int x; // Это приватное поле
};
```

Конструкторы и деструкторы

Конструктор – это метод класса, который вызывается автоматически при создании объекта класса. Конструкторы используются для инициализации объектов классов. Для того, чтобы создать конструктор, нужно в классе прописать метод без возвращаемого значение и с именем, совпадающим с названием класса. Перепишем класс `Book` из прошлых примеров так, чтобы его поля были приватными, но у класса был бы конструктор, который бы задавал эти поля:

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

class Book
{
private:
    std::string title;
    int price;

public:
    Book(const std::string& x, int y)
    {
        std::cout << "Constructor" << std::endl;
        title = x;
        price = y;
    }

    void print() const
    {
        std::cout << title << " " << price << std::endl;
    }
};

int main()
{
    Book b("Dune", 500);  // Напечатает Constructor
    b.print();            // Напечатает Dune 500
}
```

В данном примере нельзя задать значения полей объекта вне класса, так как они приватные. Но можно вызвать конструктор, который будет инициализировать эти поля. В строке:

```
Book b("Dune", 500);
```

создаётся объект `a` типа `Book`. При создания этого объекта и вызывается написанный нами конструктор, который задаёт поля соответствующими значениями.

Различные синтаксисы вызова конструктора

Язык C++ обладает богатой историей, в течение которой в него были добавлены множество новых возможностей. В результате, многие задачи в C++ можно решать различными способами. В частности, существует как минимум три способа создания объекта класса: с использование знака равенства, с использованием круглых скобок и с использованием фигурных скобок.

```
Book a = {"Dune", 500}; // Похоже на инициализацию структуры из C, но вызовется конструктор
Book b("Dune", 500);    // Изначальный синтаксис C++.
Book c{"Dune", 500};    // Синтаксис, введённый в стандарте C++11.
```

Все эти способы делают одно и то же, а именно, создают объект класса с вызовом конструктора этого класса.

Перегрузка конструкторов

Как и другие функции и методы, конструкторы можно перегружать. В этом примере создадим у класса `Book` два конструктора. Первый конструктор будет иметь один параметр – цену книги, а название книги этот конструктор будет задавать значением `"Harry Potter"`. Второй конструктор будет, как и раньше, иметь два параметра.

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

class Book
{
private:
    std::string title;
    int price;

public:
    Book(int y)
    {
        std::cout << "One" << std::endl;
        title = "Harry Potter";
        price = y;
    }

    Book(const std::string& x, int y)
    {
        std::cout << "Two" << std::endl;
        title = x;
        price = y;
    }

    void print() const
    {
        std::cout << title << " " << price << std::endl;
    }
};

int main()
{
    Book a = {100};          // Напечатает One, способ вызова через символ =
    Book b = 100;            // Напечатает One, способ вызова через символ =
    Book c(100);             // Напечатает One, способ вызова через круглые скобки
    Book d{100};             // Напечатает One, способ вызова через фигурные скобки
    a.print();               // Напечатает Harry Potter 100

    Book e = {"Dune", 500};  // Напечатает Two, способ вызова через символ =
    Book f("Dune", 500);     // Напечатает Two, способ вызова через круглые скобки
    Book g{"Dune", 500};     // Напечатает Two, способ вызова через фигурные скобки
    e.print();               // Напечатает Dune 500
}
```

В функции `main` были созданы объекты класса `Book` различными способами, используя два разных конструктора. Обратите внимание, что если у конструктора один параметр, то при использовании синтаксиса с знаком равенства, фигурные скобки можно опустить.

Конструктор по умолчанию и конструктор копирования

Конструктор по умолчанию (англ. *default constructor*) – это конструктор, который не принимает аргументов.

Конструктор копирования (англ. *copy constructor*) – это конструктор, который принимает по ссылке (обычно константной) один аргумент такого же типа, что и данный класс.

Создадим класс с конструктором по умолчанию и конструктором копирования:

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

class Book
{
private:
    std::string title;
    int price;

public:
    Book()
    {
        std::cout << "Default" << std::endl;
        title = "Harry Potter";
        price = 1000;
    }

    Book(const Book& other)
    {
        std::cout << "Copy" << std::endl;
        title = other.title;
        price = other.price;
    }

    void print() const
    {
        std::cout << title << " " << price << std::endl;
    }
};

int main()
{
    Book a;       // Напечатает Default, ещё один способ
    Book b = {};  // Напечатает Default, способ вызова через символ =
    Book c();     // Ошибка, такой способ не работает для конструктора по умолчанию, так как
                  // невозможно отличить от прототипа функции.
    Book d{};     // Напечатает Default, способ вызова через фигурные скобки
    a.print();    // Напечатает Harry Potter 1000

    Book e = a;   // Напечатает Copy, способ вызова через символ =
    Book f(a);    // Напечатает Copy, способ вызова через круглые скобки
    Book g{a};    // Напечатает Copy, способ вызова через фигурные скобки
    e.print();    // Напечатает Harry Potter 1000
}
```

Конструкторы и передача в функции/возврат из функций

Конструкторы также вызываются при передаче в функции, которые принимают по значению, и при возврате из функций. В данном примере есть функция `cat`, которая принимает объект типа `Book` по значению. Можно передавать в эту функцию аргументы конструктора класса `Book` в фигурных скобках. В этом случае внутри функции будет создан объект с помощью соответствующего конструктора. Если у конструктора один параметр, то фигурные скобки можно опустить. Аналогичным образом всё работает при возврате из функции.

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

class Book
{
private:
    std::string title;
    int price;
public:
    Book(int y)
    {
        std::cout << "One ";
        title = "Harry Potter";
        price = y;
    }
    Book(const std::string& x, int y)
    {
        std::cout << "Two ";
        title = x;
        price = y;
    }
    void print() const
    {
        std::cout << title << " " << price << std::endl;
    }
};

void cat(Book x)
{
    x.print();
}

Book dog()
{
    return {"Brave New World", 1000};
}

int main()
{
    cat({"Dune", 500});  // Напечатает Two Dune 500
    cat({500});          // Напечатает One Harry Potter 500
    cat(500);            // Напечатает One Harry Potter 500

    Book b = dog();      // Напечатает Two
    b.print();           // Напечатает Brave New World 1000
}
```

Конструкторы копирования и передача в функции/возврат из функций

Чаще всего в фунции передаются уже созданные объекты. Что если в функцию, которая принимает по значению, передать уж созданный объект? В этом случае работают такие же правила, что и прежде. Вызовется конструктор, который сможет принять объект такого типа, то есть вызовется конструктор копирования. Однако, в похожем случае при возврате из функции, конструктор копирования почему-то не вызывается...

```
#include <iostream>
#include <string>
class Book
{
private:
    std::string title;
    int price;
public:
    Book(const std::string& x, int y)
    {
        title = x;
        price = y;
    }

    Book(const Book& other)
    {
        std::cout << "Copy ";
        title = other.title;
        price = other.price;
    }

    void print() const
    {
        std::cout << title << " " << price << std::endl;
    }
};

void cat(Book x)
{
    x.print();
}

Book dog()
{
    Book b("Brave New World", 1000);
    return b;
}

int main()
{
    Book a("Dune", 500);
    cat({a});        // Напечатает Copy Dune 500
    cat(a);          // Напечатает Copy Dune 500

    Book b = dog();  // НЕ напечатает Copy (почему? это более сложный вопрос - Copy Elision)
    b.print();       // Напечатает Brave New World 1000
}
```

Деструктор

Деструктор – это метод класса, который вызывается автоматически при уничтожении объекта класса. Для переменных в сегменте стек, это происходит при выходе из области видимости в которой находится объект. Название деструктора состоит из символа тильда (~) и названия класса. Деструктор не принимает никаких аргументов.

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

class Book
{
private:
    std::string title;
    int price;
public:
    Book(const std::string& x, int y)
    {
        title = x;
        price = y;
        std::cout << "Constructor " << title << std::endl;
    }

    ~Book()
    {
        std::cout << "Destructor  " << title << std::endl;
    }
};

int main()
{
    Book a("Cat", 500);
    Book b("Dog", 1000);

    if (true)
    {
        Book c("Mouse", 2000);
    }
    std::cout << "End of main" << std::endl;
}
```

Данная программа напечатает:

```
Constructor Cat
Constructor Dog
Constructor Mouse
Destructor  Mouse
End of main
Destructor  Dog
Destructor  Cat
```

Порядок вызова деструкторов для объектов в одной области видимости является обратным к вызову конструкторов. Данный порядок вызова деструкторов имеет смысл, так как есть значительная вероятность, что объект созданный позже будет зависеть от предыдущих объектов. Если бы деструкторы вызывались в другом порядке, и объекты созданные раньше удалялись бы раньше, то мы не смогли бы удалить следующие объекты, так как они бы зависели от уже удалённых объектов.

RAII

RAII (Resource Acquisition Is Initialization) – это идиома программирования, которая обеспечивает управление ресурсами с помощью автоматического освобождения ресурсов при выходе из области видимости.

Дословное *Resource Acquisition Is Initialization* можно перевести как *Получение ресурса это инициализация*, хотя более точно эту идиому следовало бы назвать как *Получение ресурса это инициализация, а освобождение ресурса это деинициализация*. Основной смысл этой идиомы заключается в том, что следует получать ресурс в конструкторах, а освобождать ресурс в деструкторах. Ресурсом в данном контексте чаще всего является память в куче, но может быть и нечто другое, например, файловый дескриптор или объект устанавливающий соединение по сети.

Приведём пример использования идиомы RAII. Предположим, что по какой-то причине мы не можем использовать `std::string` в классе `Book`, а вместо этого должны сами выделять память в куче, в которой будем хранить название книги. Выделять память будем в конструкторе, а освобождать память, очевидно, следует в деструкторе. Это и есть смысл идиомы RAII.

```
#include <iostream>
#include <cstdlib>
#include <cstring>

class Book
{
private:
    char* title;
    int price;
public:
    Book(const char* x, int y)
    {
        title = static_cast<char*>(std::malloc((strlen(x) + 1) * sizeof(char)));
        std::strcpy(title, x);
        price = y;
    }

    ~Book()
    {
        std::free(title);
    }

    void print() const
    {
        std::cout << title << std::endl;
    }
};

int main()
{
    Book a("Dune", 500); // Память под строку "Dune" выделится в конструкторе
                         // и освободится в деструкторе
}
```

Таким образом, если мы будем использовать эту идиому, то память будет освобождаться автоматически при уничтожении объекта и нам больше не нужно будет думать о корректномы выделении и освобождении памяти. В C++ есть большое количество классов, которые используют эту идиому. Например, классы `std::string` и `std::vector` используют идиому RAII.

Указатель this

Специальный скрытый указатель **this** можно использовать внутри класса для получения адреса объекта, у которого был вызван данный метод.

```
#include <iostream>
#include <string>
using std::cout, std::endl;
struct Book
{
private:
    std::string title;
    float price;

public:
    Book (const std::string& title, float price) { this->title = title; this->price = price; }

    void printThis() const
    {
        cout << this << endl;
    }

    void printTitle() const
    {
        cout << title << endl;
        cout << this->title << endl;
    }

    int getPrice() const { return price; }
    void setPrice(float price) { this->price = price; }
};

int main()
{
    Book a = {"War and Peace", 1700};
    cout << &a << endl;
    a.printThis();
    a.printTitle();
}
```

Геттеры и сеттеры

Геттеры и сеттеры – это методы класса, которые предоставляют контролируемый доступ к приватным полям объекта. В классе выше геттером/сеттером являются методы **getPrice** и **setPrice**. Такие методы используются по следующим причинам:

- Возможность добавить нетривиальную логику при чтении или изменении поля. Например, можно добавить проверку на допустимые значения задаваемого поля, добавить логирование, добавить в геттер/сеттер код для дебагга программы и т. д.
- Даже если такая логика не нужна сейчас, она может понадобиться в будущем. Если в классе использовались геттеры/сеттеры, то добавить новый код, выполняемый при чтении/изменении поля будет очень просто.
- Использование геттеров/сеттеров упрощает изменение внутренней структуры класса. Если в будущем поле будет заменено другими полями или вычисляться иначе, внешний код останется работоспособным – ведь интерфейс через геттеры и сеттеры не изменится.

Перегрузка операторов

Перегрузка операторов – это возможность определить, как стандартные операторы должны работать для объектов классов (не путайте данное понятие с перегрузкой функций).

Перегруженный оператор, как свободная функций

В простейшем случае перегруженный оператор можно реализовать как свободную функцию. Для этого нужно просто написать функцию с особым названием `operator?`, где вместо `?` нужно подставить соответствующий оператор. Рассмотрим перегрузку операторов на следующем примере, в котором написан пустой класс `Cat` и несколько перегруженных операторов для этого класса.

```
#include <iostream>
struct Cat {};

void operator+(Cat a, int b)
{
    std::cout << "This is Cat plus int" << std::endl;
}

void operator*(Cat a, Cat b)
{
    std::cout << "This is Cat multiplies Cat" << std::endl;
}

void operator*(Cat a)
{
    std::cout << "Unary operator: dereferencing Cat" << std::endl;
}

int main()
{
    Cat c;
    c + 10;     // Напечатает This is Cat plus int
    c * c;      // Напечатает This is Cat multiplies int
    *c;         // Напечатает Unary operator: dereferencing Cat
}
```

Когда компилятор видит выражение `c + 10` он знает, что тип объекта `c` это `Cat`, а тип объекта `10` это `int`. Поэтом он начинает искать соответствующую перегрузку функции с названием `operator+` и допустимыми типами параметров. В итоге, компилятор воспринимает строку `c + 10` как строку:

```
operator+(c, 10);
```

На самом деле можно даже использовать полные имена функций перегруженных операторов:

```
int main()
{
    Cat c;
    c + 10;             // Напечатает This is Cat plus int
    operator+(c, 10);   // Напечатает This is Cat plus int

    *c;                 // Напечатает Unary operator: dereferencing Cat
    operator*(c)        // Напечатает Unary operator: dereferencing Cat
}
```

Основные правила перегрузки операторов

- Нельзя перегружать оператор, для объектов не являющихся классами (или `enum`-ами). Хотя бы один из параметров перегруженного оператора должен иметь тип класса (или `enum`-а). Например, нельзя перегрузить сумму двух `int`-ов, перегрузить арифметику указателей и прочее.
- Так как `std::string` и `std::vector` и т. д. – это классы, то для них создавать перегруженные операторы можно, только нужно следить за тем, чтобы ваш оператор не совпал с уже существующим оператором в стандартной библиотеке.
- Для перегрузки можно использовать только уже существующие операторы C++ (не все). Нельзя создать новый оператор, например `operator@` или `operator+-` и использовать его.
- Приоритет и ассоциативность перегруженных операторов соответствует соответствующим настоящим оператором. То есть бинарный оператор `*` будет всегда иметь приоритет выше чем бинарный оператор `+`, даже если это перегруженные операторы. Изменить приоритет операторов нельзя.
- Перегруженный оператор может принимать возвращая объекты по ссылке/константной ссылке также как это делают обычные функции.
- Можно перегружать большинство операторов, используемых в C++, но некоторые операторы всё-же нельзя перегружать. А именно нельзя перегружать:
  - Оператор точка (`.`) – доступ к члену класса.
  - Оператор `::` – разрешения видимости пространств имён.
  - `sizeof`, `alignof`
  - Тернарный оператор `?:`.
  - Оператор `.*` – доступ по указателю на член класса. Этот оператор C++ мы ещё не проходили.
  - И еще несколько операторов, которые мы еще не проходили.
- Оператор присваивания `=` нельзя перегружать как свободную функцию, только как метод класса.
- Оператор стрелочка `->` перегружается особым образом.

Перегруженный оператор, как метод класса

Перегруженный оператор присваивания

Ключевое слово `this` - это указатель на экземпляр класса, который можно использовать в методах этого класса. Оператор присваивания – это просто перегруженный оператор `=`. Оператор присваивания должен вернуть ссылку на текущий объект, то есть `*this`.

Нужно различать оператор присваивания и вызов конструктора:

```
Point a = Point(7, 3);  // Конструктор ( оператор присваивания не вызывается )
Point b = a;            // Конструктор копирования ( оператор присваивания не вызывается )
Point c;                // Конструктор по умолчанию
c = a;                  // Оператор присваивания
```

Оператор присваивания должен возвращать ссылку на левый аргумент.

Перегруженный оператор стрелка `->`

Перегрузка оператора `<<` с объектом `std::ostream`

Создаём свой класс комплексного числа

Используем полученные знания для создания класса комплесных чисел. В данной реализации комплексных чисел все перегруженные операторы являются свободными функциями, хотя большинство из них можно было бы сделать методами.

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

namespace mipt {
struct Complex
{
    float re;
    float im;
};

// Арифметические операторы, два комплексных числа
Complex operator+(Complex a, Complex b)
{
    Complex result = {a.re + b.re, a.im + b.im};
    return result;
}

Complex operator-(Complex a, Complex b)
{
    Complex result = {a.re - b.re, a.im - b.im};
    return result;
}

Complex operator*(Complex a, Complex b)
{
    Complex result = {a.re * b.re - a.im * b.im, a.re * b.im + a.im * b.re};
    return result;
}

Complex operator/(Complex a, Complex b)
{
    float bSquared = a.re * a.re + a.im * a.im;

    Complex result;
    result.re = (a.re * b.re + a.im * b.im) / bSquared;
    result.im = (a.im * b.re - a.re * b.im) / bSquared;
    return result;
}

// Арифметические операторы, комплексное число и вещественное
Complex operator+(Complex a, float b)
{
    Complex result = {a.re + b, a.im};
    return result;
}
```

```
Complex operator+(float a, Complex b)
{
    return b + a;
}

Complex operator-(Complex a, float b)
{
    Complex result = {a.re - b, a.im};
    return result;
}

Complex operator-(float a, Complex b)
{
    return b - a;
}

Complex operator*(Complex a, float b)
{
    Complex result = {a.re * b, a.im * b};
    return result;
}

Complex operator*(float a, Complex b)
{
    return b * a;
}

Complex operator/(Complex a, float b)
{
    Complex result = {a.re / b, a.im / b};
    return result;
}

Complex operator/(float a, Complex b)
{
    Complex ac = {a, 0.0f};
    return ac / b;
}

// Операторы присваиваний сложения и т. п.
Complex& operator+=(Complex& a, Complex b)
{
    a.re += b.re;
    a.im += b.im;
    return a;
}
Complex& operator-=(Complex& a, Complex b)
{
    a.re -= b.re;
    a.im -= b.im;
    return a;
}
```

```
Complex& operator+=(Complex& a, float b)
{
    a.re += b;
    return a;
}
Complex& operator-=(Complex& a, float b)
{
    a.re -= b;
    return a;
}

Complex& operator*=(Complex& a, float b)
{
    a.re *= b;
    a.im *= b;
    return a;
}

Complex& operator/=(Complex& a, float b)
{
    a.re /= b;
    a.im /= b;
    return a;
}


// Унарные операторы
Complex operator+(Complex a)
{
    return a;
}

Complex operator-(Complex a)
{
    Complex result;
    result.re = -a.re;
    result.im = -a.im;
    return result;
}

Complex operator*(Complex a)
{
    Complex result = {};
    result.re =  a.re;
    result.im = -a.im;
    return result;
}
```

```
/*
    Перегружаем оператор<< между типами std::ostream такой( тип имеет std::cout) и Complex
    для удобного вывода комплексных чисел на экран.

    Обратите внимание, что мы возвращаем ссылку на ostream.
    Таким образом результатом выражения  cout << a  будет cout.
    Поэтому можно делать так: cout << a << b << c ...
*/

std::ostream& operator<<(std::ostream& out, Complex a)
{
    if (a.re != 0)
        out << a.re;

    if (a.im > 0)
    {
        if (a.im != 1.0)
            out << " + " << a.im << "i";
        else
            out << " + i";
    }
    else if (a.im < 0)
    {
        if (a.im != -1.0)
            out << " - " << -a.im << "i";
        else
            out << " - i";
    }
    return out;
}
// Перегружаем оператор » с объектом типа std::istream (такой тип имеет std::cin)
std::istream& operator>>(std::istream& in, Complex& c)
{
    in >> c.re >> c.im;
    return in;
}
}

int main()
{
    mipt::Complex a = {1, 2};
    mipt::Complex b = {3, -1};

    std::cout << a + b << std::endl;  // Напечатает 4 + i
    std::cout << a * b << std::endl;  // Напечатает 5 + 5i
    std::cout << a / b << std::endl;  // Напечатает 0.2 + 1.4i
}
```

Создаём свой класс строки

Используя классы и перегрузку операторов можно написать свой удобный класс строки, аналог класса `std::string`. Можете найти код такой строки в `module2_classes_and_templates/seminar2_encapsulation/theory/code/2string`.

Раздельная компиляция класса

Методы можно вынести из определения класса следующим образом:

Определение методов в теле класса:

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


struct Point
{
    float x, y;

    float norm() const
    {
        return std::sqrt(x *x + y * y);
    }

    void normalize()
    {
        float pnorm = norm();
        x /= pnorm;
        y /= pnorm;
    }

    Point operator+(const Point& r) const
    {
        Point result = {x + r.x, y + r.y};
        return result;
    }
};




int main()
{
    Point p = {1, 2};
    p.normalize();
    std::cout << p.x << " "
              << p.y << std::endl;
}
```

Определение методов вне тела класса:

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

struct Point
{
    float x, y;

    float norm() const;
    void normalize();
    Point operator+(const Point& r) const;
};

float Point::norm() const
{
    return sqrt(x*x + y*y);
}

void Point::normalize()
{
    float pnorm = norm();
    x /= pnorm;
    y /= pnorm;
}

Point Point::operator+(const Point& r) const
{
    Point result = {x + r.x, y + r.y};
    return result;
}

int main()
{
    Point p = {1, 2};
    p.normalize();
    std::cout << p.x << " "
              << p.y << std::endl;
}
```

Теперь эти методы можно скомпилировать отдельно. Для этого их нужно вынести в отдельный компилируемый файл `point.cpp`, а определение класса в отдельный файл `point.h`. Так называемый заголовочный файл `point.h` нужен, так как определение класса нужно и файле `point.cpp` и в файле `main.cpp`. Для компиляции используем:

```
g++ main.cpp point.cpp
```

point.h

```
struct Point {
    float x, y;

    float norm() const;
    void normalize();
    Point operator+(const Point& r) const;
};
```

point.cpp

```
#include <cmath>
#include "point.h"

float Point::norm() const {
    return sqrt(x*x + y*y);
}

void Point::normalize() {
    float pnorm = norm();
    x /= pnorm;
    y /= pnorm;
}

Point Point::operator+(const Point& r) const{
    Point result = {x + r.x, y + r.y};
    return result;
}
```

main.cpp

```
#include <iostream>
#include "point.h"

int main() {
    Point p = {1, 2};
    p.normalize();
    std::cout << p.x << " " << p.y << std::endl;
}
```