МИНИСТЕРСТВО РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

«КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ»

РЕФЕРАТ

по дисциплине: Основы алгоритмизации и программирования

на тему: Перегрузка операторов в C++

Выполнил:

студент 1 курса 7 группы

факультет ИТ

Кишко Иван Петрович

Проверил:

доцент, к.т.н.

Белодед Николай Иванович

Минск 2020

Оглавление

[Предисловие 3](#_Toc35262253)

[Конструктор и деструктор 3](#_Toc35262254)

[Указатель this 6](#_Toc35262255)

[} 7](#_Toc35262256)

[Цели 8](#_Toc35262257)

[Введение 8](#_Toc35262258)

[**Какие операторы можно перегружать?** 8](#_Toc35262259)

[Перегрузка основных операций 9](#_Toc35262260)

[Оператор + 9](#_Toc35262261)

[Оператор == 9](#_Toc35262262)

[Оператор [] 10](#_Toc35262263)

[Перегрузка постфиксного инкремента 10](#_Toc35262264)

[Перегрузка префиксного инкремента 11](#_Toc35262265)

[Перегрузка операторов > и < 12](#_Toc35262266)

[Выводы 12](#_Toc35262267)

# Предисловие

## Конструктор и деструктор

*Конструктор* (от construct – создавать) – это особый метод (функция, принадлежащая объекту), который выполняется автоматически в момент создания объекта.

Общий вид описания конструктора:

Название\_объекта(передаваемые параметры)

{

тело консутрктора

}

#### Конструктор по умолчанию

Существует несколько типов конструкторов. Один из них - конструктор по умолчанию.

Это конструктор, который не принимает никаких параметров

Если в объекте не определён конструктор по умолчанию, компилятор неявно создаст его. Он будет аналогичен явно объявленному конструктору с пустым телом. Например:

#include <iostream>

using namespace std;

struct point {

int x;

int y;

};

int main()

{

point point1;// ошибки во время работы программы нет, вызывается неявный конструктор

return 0;

}

Это то же самое, что и

#include <iostream>

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point()

{}

};

int main()

{

point point1;// ошибки во время работы программы нет, вызывается явный конструктор

return 0;

}

ПРИМЕЧАНИЕ!Конструктор не возвращает никакого значения и не имеет типа возвращаемого значения.

#### Конструктор с параметрами

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point(int xCoord,int yCoord)

{

x = xCoord;

y = yCoord;

}

};

int main()

{

point point1(5,3);//конструктор с параметрами

cout << "x = " << point1.x << " y = " << point1.y;

return 0;

}



ОДНАКО! Теперь если возникнет необходимость создать объект с конструктором по умолчанию, используется синтаксис point point1();

#### Деструктор

Прежде чем говорить о третьем типе конструкторов ,необходимо упомянуть, что такое деструктор.

*Деструктор* – это особый метод (функция, принадлежащая объекту), который выполняется автоматически при удалении объекта.

Общий синтаксис

~название\_объекта()

{

тело деструктора

}

ПРИМЕЧАНИЕ! Деструктор не имеет возвращаемого значения и не принимает параметров.

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point()

{

cout << "Вызвался конструктор\n";

}

~point()

{

cout << "Вызвался деструктор\n"; //после вызова деструктора объект будет удален

}

};

int main()

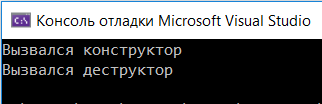
{

setlocale(LC\_ALL, "ru");

point point1;

return 0;

}



#### Конструктор копирования

Конструктор копирования применяется для создания нового объекта как  уже существующего. Такой конструктор принимает как минимум один аргумент: ссылку на копируемый объект.

Если конструктор копирования не описан явно, это не значит, что его не существует. Компилятор определяет его неявно.

#include <iostream>

using namespace std;

struct point {

int size;

int\* coord;

point(int size)

{

point::size = size;

coord = new int[size];

}

~point()

{

delete[]coord;

}

};

int main()

{

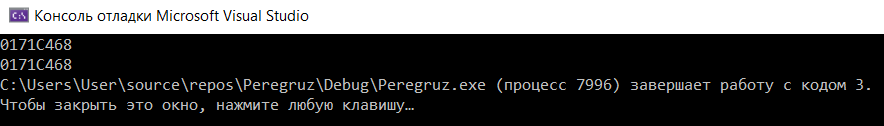
point point1(5);

point point2 = point1; //эквивалентно point point2(point1);

cout << point1.coord << endl << point2.coord;

return 0;

}



Здесь конструктор копирования задан неявно. Он выглядит примерно так

point(point const& copy)

{

size = copy.size;

coord = copy.coord;

}

В результате выполнения программы мы получим ошибку. Из консоли видно, что указатель coord объекта point1 и point2 хранят один и тот же адрес, поэтому при вызове деструктора память очищается дважды, отсюда и ошибка. Исправим её, задав конструктор копирования явно:

#include <iostream>

using namespace std;

struct point {

int size;

int\* coord;

point(int size)//конструктор с параметрами

{

point::size = size;

coord = new int[size];

}

point()//конструктор по умолчанию

{

}

~point()//деструктор

{

delete[]coord;

}

point(point const& copy)//конструктор копирования

{

size = copy.size;

coord = new int[size];

for (int i = 0; i < size; i++)

coord[i] = copy.coord[i];

}

};

int main()

{

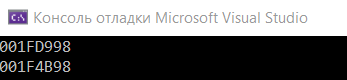
point point1(5);

point point2(point1);

cout << point1.coord << endl << point2.coord;

return 0;

}



## Указатель this

**this** — это указатель на адрес объекта , при этом он  является скрытым первым параметром любого метода (кроме статических методов), а типом указателя выступает имя объекта. И еще кое-что из теории:

1. this является зарезервированным словом С++;
2. мы можем использовать this явно в методах-элементах класса;
3. а вот явно объявить,  инициализировать  либо изменить указатель this, возможности нет;

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point(int x,int y)//конструктор с параметрами

{

cout << "constructor " << this << endl;//this хранит адрес текущего объекта,поэтому выведется адрес объекта, у которого сработал конструктор

this->x = x;//обращение к полю x текущего объекта через указатель this и присваивание ему значения

this->y = y;//обращение к полю у текущего объекта через указатель this и присваивание ему значения

}

};

int main()

{

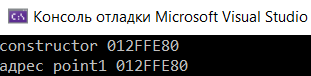
setlocale(LC\_ALL, "ru");

point point1(5,4);

cout <<"адрес point1"<< &point1;

return 0;

}



# Цели

Узнать ,что такое перегрузка операторов . Какие операторы можно перегружать ,а какие – нельзя. Зачем необходима перегрузка операторов и как она реализуется

# Введение

Оператор в С++ - это некоторое действие или функция, обозначенная специальным символом. Для того что бы распространять эти действия на новые типы данных, при этом сохраняя естественный синтаксис, в С++ была введена возможность перегрузки операторов.(+, -, <<, >>, == и так далее)

Перегрузка операторов в программировании дает возможность одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Перегрузка операторов, это всего лишь более удобный способ вызова функций, поэтому не стоит увлекаться перегрузкой операторов. Использовать её следует только тогда, когда это упростит написание кода. Но, не настолько, чтобы это затрудняло чтение. И не забывайте, возможность перегрузки есть только для пользовательских типов/классов.

**Какие операторы можно перегружать?**

Мы можем перегрузить почти любой оператор C++, учитывая следующие исключения и ограничения:

•Нельзя определить новый оператор, например, operator\*\*.

•Следующие операторы перегружать нельзя:

1. ?: (тернарный оператор);
2. :: (доступ к вложенным именам);
3. . (доступ к полям);
4. ->(доступ к полям по указателю);
5. sizeof, typeid и операторы приведения типов.

• Следующие операторы можно перегрузить только в качестве методов:

* 1. = (присваивание);
  2. -> (доступ к полям по указателю);
  3. () (вызов функции);
  4. [] (доступ по индексу);
  5. ->\* (доступ к указателю-на-поле по указателю);
  6. операторы конверсии и управления памятью.
* Количество операндов, порядок выполнения и ассоциативность операторов определяется стандартной версией.
* Как минимум один операнд должен быть пользовательского типа. Typedef не считается.

# Перегрузка основных операций

## Общий синтаксис

тип\_возвращаемого\_значения ключевое слово operator сам\_оператор(передаваемые параемтры)

{

Тело оператора

}

## Оператор +

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point operator+(const point& other)

{

point temp;//объект для возврата результата

temp.x = this->x + other.x; //this - указатель на текущий объект(На

temp.y = this->y + other.y; //объект,стоящий слева от оператора)

return temp;

}

point(int x, int y)//конструктор,принимающий начальные значения объекта

{

this->x = x;

this->y = y;

}point() {} //конструктор по умолчанию

};

int main()

{

point B3(10, 15), B4(1, 3), B5;

B5 = B3 + B4;

cout << "X = " << B5.x << " Y = " << B5.y << endl;

return 0;

}



## Оператор ==

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point(int x, int y)//конструктор,принимающий начальные значения объекта

{

this->x = x;

this->y = y;

}

bool operator==(const point other)

{

return this->x == other.x && this->y == other.y;//Вернуть true если соответствующие координаты равны

}

};

int main()

{

point B1(3, 5), P(5, 12);

bool result = B1 == P;

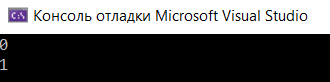
cout << result << endl; //result = 0

point H(4, 2), J(4, 2);

result = H == J;

cout << result << endl; //result = 1

}



## Оператор []

#include <iostream>

using namespace std;

class Point {

int coord[3];

public:

int& operator[](int index) //возврат по ссылке для возможности изменения данных

{

return coord[index];

}

Point(int x, int y, int z)

{

coord[0] = x;

coord[1] = y;

coord[2] = z;

}

};

int main()

{

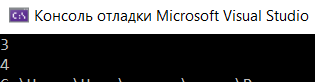
Point M(3, 4, 5);

cout<<M[0]<<endl;

M[0] = 4;

cout << M[0];

}



## Перегрузка постфиксного инкремента

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point operator++(int value)//передача параметра необходима

{ //для отличия потсфиксной записи от префиксной

point temp(\*this);//конструктор копирования

this->x++;

this->y++;

return temp;

}

point(int x, int y)//конструктор,принимающий начальные значения объекта

{

this->x = x;

this->y = y;

}point() {}//конструктор по умолчанию

};

int main()

{

point A1(1, 3), B1;

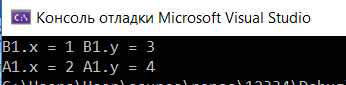
B1 = A1++;//сначала выполняется присваивание , а затем увеличение полей структуры А1

cout << "B1.x = " << B1.x << " B1.y = " << B1.y << endl;

cout << "A1.x = " << A1.x << " A1.y = " << A1.y;

point B3(10, 15), B4(1, 3), B5;

}



## Перегрузка префиксного инкремента

#include <iostream>

using namespace std;

struct point {

int x;

int y;

point operator++()//перегрузка постфиксного оператора

{

this->x++;

this->y++;

return \*this;

}

point(int x, int y)//конструктор,принимающий начальные значения объекта

{

this->x = x;

this->y = y;

}point() {}//конструктор по умолчанию

};

int main()

{

point A1(1, 3), B1;

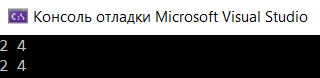
B1 = ++A1//сначала выполняется увеличение полей структуры А1,а затем присваивание

cout << B1.x << " " << B1.y << endl;

cout << A1.x << " " << A1.y;

point B3(10, 15), B4(1, 3), B5;

}



## Перегрузка операторов > и <

#include <iostream>

typedef unsigned short day;//создаем тип данных день на основе беззнакового типа short

typedef unsigned short month;//создаем тип данных месяц на основе беззнакового типа short

typedef unsigned short year;//создаем тип данных год на основе беззнакового типа short

struct Data

{

day dd;

month mm;

year yyyy;

bool operator>(Data temp)

{

return this->yyyy > temp.yyyy ||

this->yyyy == temp.yyyy && this->mm > temp.mm ||

this->yyyy == temp.yyyy && this->mm == temp.mm && this->dd > temp.dd;

}

bool operator<(Data temp)

{

return this->yyyy < temp.yyyy ||

this->yyyy == temp.yyyy && this->mm < temp.mm ||

this->yyyy == temp.yyyy && this->mm == temp.mm && this->dd < temp.dd;

}

};

int main()

{

setlocale(LC\_ALL, "ru");

Data date1 = { 7,1,1980 };

Data date2 = { 7,2,1993 };

Data date3 = { 7,1,1980 };

if (date1 < date2)std::cout << "Истина" << std::endl;

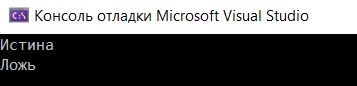
else std::cout << "Ложь" << std::endl;

if (date1 > date2)std::cout << "Истина" << std::endl;

else std::cout << "Ложь" << std::endl;

return 0;

}



# Выводы

Перегрузка - это возможность поддерживать несколько функций с одним названием, но разными сигнатурами вызова. Проследил на примерах выше синтаксис реализации перегрузок. Уяснил ,какие операторы перегружать можно ,а какие нельзя. Таким образом благодаря перегрузке операторов мы можем выполнять со своими типами данных те же операции, что и со встроенными.