Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра Электронных вычислительных машин

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

К ЛАБОРАТОРНЫМ РАБОТАМ

по курсу

«Объектно-ориентированное программирование»

(часть 1)

для студентов факультета ФРЭ, ФКП

Минск – 2018

# **Введение**

Основная цель настоящих методических указаний – дать материал для лабораторных занятий по изучаемым темам дисциплины. В указаниях по каждой изучаемой теме приводится список практических заданий для выполнения.

**Темы лабораторных работ:**

1. Организация ввода/вывода, динамическое выделение памяти.
2. Классы и объекты.
3. Дружественные функции и классы, перегрузка операторов.
4. Наследование, простое наследование.
5. Принцип полиморфизма, виртуальные функции, абстрактные классы.
6. Множественное наследование, виртуальное наследование.
7. Реализация шаблонов классов.
8. Практические приемы использования шаблонов классов.

# **Лабораторная работа №1**

**Тема работы**: Организация ввода/вывода, динамическое выделение памяти

**Цель работы:** Изучить организацию ввода/вывода и работу с динамической памятью при программировании алгоритмов в С++.

**Теоретические материалы.**

**Ввод/вывод.** В С++ ввод и вывод данных производится потоками байт. Поток (последовательность байт) − это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода/вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот.

Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.

В файле iostream перегружаются два оператора побитового сдвига

<< // поместить в выходной поток

>> // считать со входного потока

и объявляются три стандартных потока:

cout // стандартный поток вывода (экран)

cin // стандартный поток ввода (клавиатура)

cerr // стандартный поток диагностики (ошибки)

**Объект cin.** Для ввода информации с клавиатуры используется объект cin. Формат записи cin имеет следующий вид:

cin [>>имя\_переменной];

При вводе необходимо, чтобы данные вводились в соответствии с форматом переменных.

**Объект cout.** Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид:

сout << data [ << data];

где data − это переменные, константы, выражения или комбинации всех трех типов. Пример использования объектов cin и cout.

#include<iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL,"Russian");

int i;

double x;

cout << "Введите число с двойной точностью" << endl;;

cin >> x; // ввод числа с плавающей точкой

cout << "Введите положительное число" << endl;

cin >> i; // ввод целого числа

cout << "i \* x=" << i\*x << endl; // вывод результата

return 0;

}

Идентификатор endl называется манипулятором. Он очищает поток cerr и добавляет новую строку.

Для управления выводом информации в языке С++ используются манипуляторы. Для их использования необходим заголовочный файл iomanip. Манипуляторы hex и oct используются для вывода числовой информации в шестнадцатеричном или восьмеричном представлении. Применение их можно видеть на примере следующей простой программы:

#include<iostream>

using namespace std;

void main()

{

setlocale(LC\_ALL,"Russian");

int a=0x11, b=4, // целые числа: шестнадцатеричное и десятичное

c=051, d=8, // восьмеричное и десятичное

i, j;

i=a+b;

j=c+d;

cout << i << ' ' << hex << i << ' ' << oct << i << ' ' << dec << i << endl;

cout << hex << j << ' ' << j << ' ' << dec << j << ' ' << oct << j << endl;

}

Манипуляторы изменяют значение некоторых переменных в объекте cout. Эти переменные называются флагами состояния. Когда объект посылает данные на экран, он проверяет эти флаги.

Пример использования манипуляторов форматирования информации.

#include<iostream>

#include <iomanip>

using namespace std;

void main()

{

int a=0x11;

double d=12.362;

cout << setw(4) << a << endl;

cout << setw(10) << setfill('\*') << a << endl;

cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl;

}

Манипуляторы setw(), setfill(' ') и setprecision() позволяют изменять флаги состояния объекта cout. Синтаксис их показывает, что это функции, позволяющие изменять флаги состояния объекта cout. Функции имеют следующий формат:

setw(количество\_позиций\_для\_вывода\_числа)

setfil(символ\_для\_заполнения\_пустых\_позиций)

setprecision (точность\_при\_выводе\_дробного\_числа)

Наряду с перечисленными выше манипуляторами в С++ используются также манипуляторы **setiosflags**() и **resetiosflags**() для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на *переменные состояния*. Функция setiosflags() устанавливает указанные в ней флаги, а resetiosflags() сбрасывает (очищает) их. Для того чтобы установить или сбросить некоторый флаг, могут быть использованы функции **setf**() или **unsetf**(). Флаги формата объявлены в классе **ios**. Рассмотрим пример, демонстрирующий использование манипуляторов.

#include<iostream>

#include <iomanip>

using namespace std;

int main()

{

setlocale(LC\_ALL,"Russian");

char s[]="Минск БГУИР ";

cout << setw(30) << setiosflags(ios::right) << s << endl;

cout << resetiosflags(ios::right);

cout << setw(30) << setiosflags(ios::left) << s << endl;

return 0;

}

**Динамическое распределение памяти.** Память под массивы можно выделять динамически, т. е. размещать в свободной памяти (free store). Свободная память – это предоставляемая системой область памяти для объектов, время жизни которых устанавливается программистом. В С++ для операций выделения и освобождения памяти используются встроенные операторы **new** и **delete**.

**Оператор new** имеет один операнд. Оператор имеет две формы записи:

[::] new [(список\_аргументов)] имя\_типа [(инициализирующее\_значение)]

[::] new [(список\_аргументов)] (имя\_типа) [(инициализирующее\_значение)]

В простейшем виде оператор new можно записать следующим образом:

new имя\_типа;

new имя\_типа(выражение);

new имя\_типа[выражение];

Оператор new выделяет надлежащий объем свободной памяти для хранения указанного типа и возвращает базовый адрес объекта. Когда память недоступна, оператор new возвращает NULL либо возбуждает соответствующее исключение, например:

#include<iostream>

#include <iomanip>

using namespace std;

void main()

{

setlocale(LC\_ALL,"Russian");

int \*p, \*q, size,i;

p=new int(5); // выделение памяти и инициализация

cout << "Введите размер массива" << endl;

cin >> size;

q=new int[size]; // выделение памяти под массив

cout << "Введите элементы массива" << endl;

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

cin >> q[i];

cout << " Массив q" << endl;

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

cout << setw(4)<< q[i];

cout << endl;

delete p;

delete [] q;

}

Можно динамически выделить память под двухмерный массив, используя «указатель на указатель». В языке С++ допустимо объявлять переменные, имеющие тип «указатель на указатель». Объявляется «указатель на указатель» следующим образом:

int \*\*mas;

Фактически «указатель на указатель» – это адрес ячейки памяти, хранящей адрес указателя. Например:

#include<iostream>

#include <iomanip>

using namespace std;

void main()

{

setlocale(LC\_ALL,"Russian");

int \*\*mas; // указатель на указатель на массив

int n, m; // количество строк и столбцов массива

cout << "Введите количество строк и столбцов" << endl;

cin >> n >> m; // ввод количества строк и столбцов

mas=new int\*[n]; // выделение памяти под массив

указателей

for(int j=0; j < n; j++)

mas[j]=new int[m];

cout << "Введите элементы в массив" << endl;

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

for(int j=0; j < m; j++)

cin >> mas[i][j];

cout << "Двухмерный массив" << endl;

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

{

cout << endl;

for(int j=0; j < m; j++)

cout << setw(4) << mas[i][j];

}

cout << endl;

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

delete [] mas;

}

Оператор delete уничтожает объект, созданный с помощью new.

**Оператор delete** имеет две формы записи:

[::] delete переменная\_указатель // для указателя на один элемент

[::] delete [] переменная\_указатель // для указателя на массив

Первая форма используется, если соответствующее выражение new размещало не массив. Во второй форме присутствуют пустые квадратные скобки, показывающие, что изначально размещался массив объектов. Оператор delete не возвращает значения.

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Осуществить демонстрацию работы функций по вариантам. Все возможные данные вводятся с клавиатуры с проверкой на корректность. Ввод/вывод данных осуществлять только в main. В конце программы осуществить вывод результатов работы функций. Память выделять динамически. В случае отсутствия надобности в дальнейшем использовании выделенной памяти осуществлять освобождение памяти.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | Реализовать 2 функции для формирования массива строк из 2 заданных строк: первая строка будет являться первой строкой массива строк, а каждый символ второй строки будет являться первым символом остальных строк в массиве, при этом каждая такая строка будет формироваться путем сложения ASCII-кода соответствующего символа первой строки с разницей между ASCII-кодом первого символа текущей строки с ASCII-кодом первого символа первой строки. В одном случае результирующий массив строк является возвращаемым значением, а аргументы не изменяются; в другом - в качестве одного из аргументов передается матрица (память под нее выделяется в функции), которая примет результирующее значение работы функции и функция ничего не возвращает. |
| **2** | Реализовать 2 функции для математического перемножения 2 матриц, учитывая то, что заданные матрицы могут и не иметь возможности для перемножения. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **3** | Реализовать 2 функции транспонирования матрицы и 2 функции сложения 2 матриц, учитывая то, что заданные матрицы могут и не иметь возможности для сложения. Перед сложением первая матрица транспонируется. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **4** | Реализовать 2 функции для транспонирования вектора и 2 функции для построения матрицы из 2 векторов: один вектор транспонируется и становится первым столбцом, а второй становится первой строкой (будет ли первая строка начинаться со второго столбца или же первый столбец со второй строки остается на усмотрение студента). Элемент матрицы является результатом произведения соответствующего элемента первой строки на соответствующий элемент первого столбца (естественно, что это справедливо для всех элементов кроме элементов первой строки и столбца). В одном случае результирующая значение является возвращаемым значением, а вектора переданные в качестве аргументов не изменяются; в другом - в качестве одного из аргументов передается вектор/матрица (память под него/нее выделяется в функции), которая примет результирующее значение работы функции и функция ничего не возвращает. |
| **5** | Реализовать 2 функции для перемножения 2 матриц, учитывая то, что заданные матрицы могут и не иметь возможности для перемножения. Перемножение для первого элемента результирующей матрицы осуществлять путем умножения первого элемента первой матрицы на последний элемент второй матриц, для остальных элементов аналогично, при этом проход по матрицам для перемножения осуществляется в виде улитки. Заполнение и вывод матриц осуществляется стандартно. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **6** | Реализовать функцию транспонирования матрицы и функцию для перемножения 2 матриц, учитывая то, что заданные матрицы могут и не иметь возможности для перемножения. Перемножение осуществляется путем умножения элемента первой матрицы на элемент второй матрицы. При самом перемножении необходимо идти построчно по матрицам, при этом первой строкой первой матрицы будет являться строка с наименьшей суммой элементов, а для второй матрицы - с наибольшей разницей элементов, остальные строки матриц определяются аналогично. Перед умножением матриц производится транспонирование первой матрицы. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **7** | Реализовать 2 функции для математического перемножения подматриц 2 введенных матриц, учитывая то, что заданные матрицы могут и не иметь возможности для перемножения. Подматрицы определяются путем разделения исходной матрицы столбцом и строкой в которых находится элемент, находящийся в главной и побочной диагоналях. При этом должна проводиться проверка на возможность соответствующего разделения. Подматрицы для перемножения выбираются крест накрест, то есть верхняя левая подматрица первой матрицы умножается на нижнюю правую подматрицу второй матрицы и так далее. Строка и столбец с помощью которых матрицы делятся на подматрицы просто перемножаются между собой (строка со строкой, столбец со столбцом), то есть первый элемент строки первой матрицы умножается на первый элемент строки второй матрицы и так далее. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **8** | Реализовать 2 функции для сложения 2 матриц, учитывая то, что заданные матрицы могут и не иметь возможности для сложения. Сложение осуществляется путем сложения элемента первой матрицы на элемент второй матрицы. При самом сложении элементы главной диагонали первой матрицы складываются с соответствующими элементами побочной диагонали второй матрицы (подразумеваются элементы, который находятся в строке с одним и тем же номером, то есть складывается элемент главной диагонали первой матрицы находящийся в первой строке матрицы с элементом побочной диагонали находящимся в первой строке второй матрицы) и наоборот, а части, отделенные главной и побочной диагоналями, складываются верхняя с нижней, левая с правой и наоборот с учетом нахождения в одном столбце и соответствующей строки (для верхней и нижней) или в одной строке и соответствующем столбце (для левой и правой) и наоборот. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **9** | Реализовать 2 функции для сложения 2 матриц, элементами которых являются строки, учитывая то, что заданные матрицы могут и не иметь возможности для сложения. При сложении строки со строкой должна осуществляться конкатенация (объединение) строк. В одном случае результирующая матрица является возвращаемым значением, а матрицы переданные в качестве аргументов не изменяются; в другом - первая передаваемая матрица становится результирующим значением работы функции и функция ничего не возвращает. |
| **10** | Реализовать 2 функции для разности (элементы первой строки присутствующие во второй строке удаляются) 2 строк: в одном случае результирующая строка является возвращаемым значением, а аргументы не изменяются; в другом - первая передаваемая строка становится результирующим значением работы функции и функция ничего не возвращает. |

# 

# **Лабораторная работа №2**

**Тема работы**: Классы и объекты.

**Цель работы:** Изучить структуру класса, атрибуты доступа к компонентам класса; рассмотреть принцип работы конструкторов (с параметрами, с параметрами по умолчанию), конструктора копирования и деструктора при работе с объектом, статические и константные данные и методы.

**Теоретические материалы.**

Основная идея введения классов заключается в том, чтобы предоставить программисту средства для создания новых типов данных, которые могут использоваться так же, как и встроенные типы. Класс – это абстракция, точнее, это тип, определяемый пользователем. Например, мы можем задать структуру с именем point, которая содержит координаты ***x*** и ***y*** точки на экране дисплея:

struct point

{

int x,y;

};

Пусть нам нужны две функции, которые позволяют нарисовать точку set\_pixel() и прочитать ее координаты get\_pixel():

void set\_pixel(int, int);

void get\_pixel(int \*, int \*);

В нашем примере данные и функции, работающие с этими данными, отделены друг от друга. Связь между ними можно установить, если задать структуру в виде

struct point

{

int x,y;

void set\_pixel(int, int);

void get\_pixel(int \*, int \*);

};

Данные x и y, объявленные в приведенной структуре, называются компонентами-данными, а функции – компонентами-функциями или методами. Объект объявляется следующим образом:

имя\_класса имя\_объекта;

Теперь мы можем обратиться к данным и вызвать функции, только указав имя объекта, к которому они принадлежат. Для этих целей можно использовать те же операции точка «.» и «->». Рассмотрим пример.

#include<iostream>

#include <iomanip>

using namespace std;

struct point // объявление структуры

{

int x,y;

void set\_pixel(int, int); // установить значения переменных x и y

void get\_pixel(int \*, int \*); // получить значения x и y

};

int main()

{

setlocale(LC\_ALL,"Russian");

int a,b;

point my\_point, \*pointer;

pointer = &my\_point;

my\_point.set\_pixel(50,100);

pointer -> get\_pixel(&a,&b);

cout << "a= " << a << " b= " << b << endl;

return 0;

}

void point::set\_pixel(int a, int b)

{ x=a;

y=b;

}

void point::get\_pixel(int \*a, int \*b)

{

\*a=x;

\*b=y;

}

Поскольку различные структуры могут иметь функции с одинаковыми именами, при описании функции необходимо указывать, для какой структуры она описывается:

void point::set\_pixel(int x, int y)

{

тело функции

}

Синтаксис описания функции, принадлежащей структуре, имеет следующий вид (рис. 1).

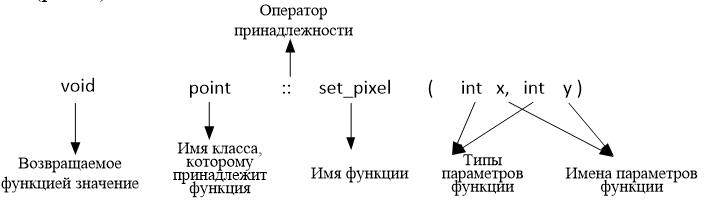


Рис.1. Синтаксис описания функции, принадлежащей структуре

Оператор принадлежности «::» иначе называется оператором разрешения области видимости. Этот оператор самого высокого приоритета.

::i // унарный оператор :: указывает на внешнюю область видимости

point::i // бинарный оператор :: указывает на область видимости класса

В языке С++ структура – это тоже класс. С другой стороны, мы можем записать вместо ключевого слова struct ключевое слово class, например:

class point

{

int x,y;

void set\_pixel(int, int);

void get\_pixel(int \*, int \*);

};

В языке С++ класс, определяемый посредством ключевых слов struct, class, union, включает в себя функции и данные, создавая новый тип объектов. Компоненты класса имеют ограничения на доступ. Эти ограничения определяются ключевыми словами private, protected, public. Для ключевого слова class по умолчанию все компоненты будут private. Это означает, что они (их имена) будут недоступны для использования вне компонентов класса. Ограничения доступа для некоторого компонента можно изменить, записав перед ним атрибут модификации доступа – ключевое слово public или protected и двоеточие. Таким образом, упрощенную форму описания класса можно записать в виде:

class имя\_класса

{

данные и функции с атрибутом private (по умолчанию)

protected:

данные и функции с атрибутом protected

public:

данные и функции с атрибутом public

} объекты этого класса через запятую.

Обычно ограничения на уровень доступа касаются элементов данных: данные имеют атрибут private или protected, а методы – public.

Смысл атрибутов доступа следующий:

− **private** – член класса с атрибутом private может использоваться только методами собственного класса и функциями-«друзьями» этого же класса; по умолчанию все члены класса, объявленного с ключевым словом class, имеют атрибут доступа private;

− **protected** – то же, что и private, но дополнительно член класса может использоваться методами и функциями-«друзьями» производного класса, для которого данный класс является базовым;

− **public** – член класса может использоваться любой функцией программы, т. е. защита на доступ снимается.

Явно ограничения на доступ могут переопределяться записью атрибута перед компонентами класса. Элементы класса типа структуры (struct) и объединения (union) по умолчанию принимаются как public. Для ключевого слова struct атрибут можно явно переопределить на private или protected. Для ключевого слова union явное переопределение атрибута доступа невозможно. Класс или структура может содержать любое количество секций с заданными атрибутами. Секция начинается с ключевого слова и двоеточия после него. Секция заканчивается в конце описания структуры (класса) или началом другой секции.

Пример использования атрибутов доступа к элементам класса.

#include<iostream>

#include <iomanip>

using namespace std;

class String

{

char str[25]; // атрибут доступа private

public:

void set\_string(char \*); // функция инициализации строки str

void display\_string(); // функция вывода строки str на экран

char \* return\_string(); // функция возвращения строки

};

void String::set\_string(char \*s)

{

strcpy(str,s); // копирование s в str

}

void String::display\_string()

{

cout << str << endl;

}

char \* String::return\_string()

{

return str;

}

int main()

{

setlocale(LC\_ALL,"Russian");

String str1; // объявление объекта

str1.set\_string("Минск");

str1.display\_string();

cout << str1.return\_string() << endl;

return 0;

}

Использование функций для установки начальных значений данных объекта часто приводит к ошибкам. В связи с этим введена специальная функция, позволяющая инициализировать объект в процессе его декларирования (определения). Эта функция называется конструктором. Функция-конструктор имеет то же имя, что и соответствующий класс.

class String

{

char str[25]; // атрибут доступа private

public:

String(char \*s) // конструктор

{

strcpy(str,s);

}

};

Конструктор может иметь и не иметь аргументы и он никогда не возвращает значение (даже типа void). Класс может иметь несколько конструкторов, что позволяет использовать несколько различных способов инициализации соответствующих объектов. Иначе можно сказать – конструктор является функцией, а значит он может быть перегружен. Конструктор вызывается, когда связанный с ним тип используется в определении. Например:

#include<iostream>

using namespace std;

class Over

{

int i;

char \*str;

public:

Over()

{

str="Первый конструктор";

i=0;

}

Over(char \*s) // Второй конструктор

{

str=s;

i=50;

}

Over(char \*s, int x) // Третий конструктор

{

str=s;

i=x;

}

Over(int \*y)

{

str="Четвертый конструктор\n";

i=\*y;

}

void print();

};

void Over::print()

{

cout << "i= " << i << " str= " << str << endl;

}

int main()

{ setlocale(LC\_ALL,"Russian");

int a=10, \*b;

b=&a;

Over my\_over; // Активен конструктор Over()

Over my\_over1("Для конструктора с одним параметром");

Over my\_over2("Для конструктора с двумя параметрами", 100);

Over my\_over3(b); // Для четвертого конструктора

my\_over.print();

my\_over1.print();

my\_over2.print();

my\_over3.print();

return 0;

}

Результаты выполнения программы представляются в следующем виде:

i=0; str= Первый конструктор

i=50; str= Для конструктора с одним параметром

i=100; str= Для конструктора с двумя параметрами

i=10; str= Четвертый конструктор

Конструктор может содержать значения аргументов по умолчанию. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Созданный программистом конструктор, у которого все аргументы по умолчанию, называется конструктором с умолчанием, т. е. конструктором, который можно вызывать без указания каких-либо аргументов. Для каждого класса может существовать только один конструктор с умолчанием.

Пример использования конструктора по умолчанию.

#include<iostream>

using namespace std;

class Time

{

public:

Time(int = 0, int = 0, int = 0); // конструктор с параметрами по умолчанию

void setTime(int, int, int); // установка часов, минут, секунд

void printStandart(); // печать времени в стандартном формате

private:

int hour; // 0–23

int minute; // 0–59

int second; // 0–59

}

// Конструктор Time инициализирует члены класса нулевыми значениями

Time::Time(int hr, int min, int sec)

{

setTime(hr, min, sec);

}

// Установка нового значения времени. Выполнение проверки корректности

// значений данных. Установка неправильных значений на 0.

void Time::setTime(int h, int m, int s)

{

hour = (h >= 0 && h < 24) ? h : 0;

minute = (m >= 0 && m < 60) ? m : 0;

second = (s >= 0 && s < 60) ? s : 0;

}

// Печать времени в стандартном формате

void Time::printStandart()

{

cout << (( hour == 0 || hour == 12) ? 12 : hour % 12)

<< ":" << ( minute < 10 ? "0" : "" ) << minute

<< ":" << ( second < 10 ? "0" : "" ) << second

<< ( hour < 12 ? "AM" : "PM" );

}

int main()

{

setlocale(LC\_ALL,"Russian");

Time t1, // все аргументы являются умалчиваемыми

t2(2), // минуты и секунды являются умалчиваемыми

t3(21, 34), // секунды являются умалчиваемыми

t4(12, 25, 42), // все значения указаны

t5(27, 74, 99); // все неправильные значения указаны

cout << "\nВсе аргументы по умолчанию ";

t1.printStandart();

cout << "\nЧасы заданы; минуты и секунды по умолчанию ";

t2.printStandart();

cout << "\nЧасы и минуты заданы; секунды по умолчанию ";

t3.printStandart();

cout << "\nЧасы, минуты и секунды заданы: ";

t4.printStandart();

cout << "\nВсе значения заданы неверно: ";

t5.printStandart();

cout << endl;

return 0;

}

**Конструктор имеет следующие отличительные особенности**:

− всегда выполняется при создании нового объекта, т.е. когда под объект отводится память и когда он инициализируется;

− может определяться пользователем или создаваться по умолчанию;

− не может быть вызван явно из пределов программы (не может быть вызван как обычный метод). Он вызывается явно компилятором при создании объекта и неявно при выполнении оператора new для выделения памяти объекту;

− всегда имеет то же имя, что и класс, в котором он определен;

− никогда не должен возвращать значения;

− не наследуется.

**Копирующий конструктор.** Рассмотрим следующую программу, демонстрирующую использование копирующего конструктора.

#include<iostream>

#include<iomanip>

using namespace std;

class Massiv

{

int \*mas; // указатель на массив

int n; // количество элементов массива

public:

Massiv(int n1=0); // конструктор с параметрами по умолчанию

void vvod(); // функция ввода значений массива

void display(); // функция вывода значений массива

int fun(Massiv ob); // функция вычисления суммы элементов массива

~Massiv(); // деструктор

};

Massiv::Massiv(int n1) // определение конструктора

{

n=n1;

mas=new int[n];

}

Massiv::~Massiv() // определение деструктора

{

delete [] mas;

}

void Massiv::vvod() // определение функции ввода значений массива

{

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

cin >> mas[i];

}

void Massiv::display() // определение функции вывода значений массива

{

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

cout << setw(4) << mas[i];

cout << endl;

}

int Massiv:: fun(Massiv ob) // функция вычисления суммы элементов массива

{

int sum=0;

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

sum+=ob.mas[i];

return sum;

}

int main() // головная функция

{

setlocale(LC\_ALL,"Russian");

int summa;

int n; // размер массива

cout << "Введите размер массива" << endl;

cin >> n;

Massiv ob(n); // объявление объекта

cout << "Введите элементы массива" << endl;

ob.vvod(); // вызов функции vvod

cout << "Исходный массив" << endl;

ob.display(); // вызов функции display

summa=ob.fun(ob); // вызов функции fun

cout << "сумма элементов массива" << setw(4) << summa << endl;

ob.display(); // при вызове функции display() возникает ошибка

}

В функцию fun() передается значение объекта типа Massiv. Даже если вызванная функция ничего не будет выполнять, произойдет ошибка, связанная с динамическим выделением и освобождением памяти. Параметр в функции fun() является локальным (автоматическим объектом) в теле этой функции. Любой автоматический объект конструируется тогда, когда встречается его объявление, и разрушается, когда блок, в котором он описан, прекращает существование. После завершения функция прекращает существовать, в результате вызывается деструктор объекта ob.

В функции main() выполняются следующие действия:

− описание Massiv ob(n); задается конструирование нового объекта ob. Конструктор объекта оb выделяет (динамически) память под массив mas с помощью оператора new;

− вызывается функция fun(ob);

− значение объекта оb копируется из функции main в стек функции fun();

− копия объекта оb содержит указатель на ту же динамическую память (указатель на динамическую память в объекте-оригинале и объекте-копии имеет одинаковые значения);

− функция fun() завершается;

− вызывается деструктор для копии объекта оb, который разрушает динамически выделенную память под массив mas;

− указатель в оригинале объекта адресует несуществующую удаленную память.

Если в приведенном примере изменить заголовок функции fun(Massiv ob) на заголовок fun(Massiv& ob), то ошибка будет устранена. При необходимости можно оставить и исходное объявление функции. В этом случае надо устранить ошибку в самом классе Massiv. Когда объект оb копируется из функции main() в функцию fun, то должен вызываться конструктор для копирования. Общий вид конструктора копирования имеет следующий вид:

имя\_класса (const имя\_класса & );

Так как в нашем классе такого конструктора нет, то вызывается конструктор, заданный по умолчанию. Этот конструктор строит точную копию всех данных объекта оb, что и приводит к ошибке. Если в классе Massiv задать явно конструктор для копирования, например,

Massiv(const Massiv& ob)

{

n=ob.n;

mas=new int[n];

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

mas[i]=ob.mas[i];

}

то ошибка будет устранена. Таким образом, если в конструкторе некоторого класса Massiv осуществляется динамическое выделение памяти, такой класс должен иметь соответствующий конструктор для копирования, а также деструктор (для освобождения памяти).

**Деструктор.** Противоположные действия, по отношению к действиям конструктора, выполняют функции-деструкторы (destructor), или разрушители, которые уничтожают объект. Деструктор может вызываться явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Явное уничтожение объекта выполняет оператор delete. Деструктор имеет то же имя, как и класс, но перед именем записывается знак тильда «~». Кроме того, деструктор не может иметь аргументы, возвращать значение и наследоваться.

Пример использования деструктора.

#include<iostream>

using namespace std;

class String

{

int i;

public:

String(int j); // объявление конструктора

~String(); // объявление деструктора

void show\_i(void);

}; // конец объявления класса

String::String(int j) // определение конструктора

{

i=j;

cout << "Работает конструктор" << endl;

}

void String::show\_i(void) // определение функции

{

cout << "i= " << i << endl;

}

String::~String() // определение деструктора

{

cout << "Работает деструктор" << endl;

}

int main()

{

setlocale(LC\_ALL,"Russian");

String my\_ob1(25); // инициализация объекта my\_ob1

String my\_ob2(36); // инициализация объекта my\_ob2

my\_ob1.show\_i(); // вызов функции show\_i() класса String для my\_ob1

my\_ob2.show\_i(); // вызов функции show\_i() класса String для my\_ob2

return 0;

}

Результаты работы программы следующие:

Работает конструктор

Работает конструктор

i=25

i=36

Работает деструктор

Работает деструктор

Деструкторы выполняются в обратной последовательности по отношению к конструкторам. Первым разрушается объект, созданный последним.

Пример использования конструкторов и деструктора. Разработать класс вектор (одномерный динамический массив). Методы класса: конструкторы, деструктор и несколько методов, выполняющих преобразование в массиве (например, нахождения максимального значения и сортировки).

#include <iostream>

#include <new>

#include<iomanip>

using namespace std;

// объявление класса

class vect

{

int \*v; // вектор (одномерный массив)

int n; // размерность вектора

public:

vect(); // конструктор без параметров

vect(int, int\*); // конструктор с двумя параметрами

~ vect(); // деструктор

void set(); // инициализация вектора

void print(); // вывод вектора на экран

int funk1(); // нахождения максимума в массиве

};

// реализация методов класса

vect :: vect() : n(0),v(0){} // конструктор без параметров

vect :: vect(int nn, int \*vv) // конструктор с двумя параметрами

{

n=nn; // инициализация размерности

v=new int[n]; // выделение памяти под вектор

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

\*(v+i)=\*(vv+i);

}

vect :: ~ vect() // деструктор

{ delete [] v; }

void vect :: set() // инициализация вектора

{

if(!v)

{

cout << "не выделена память под вектор" << endl;

cout << "введите размерность вектора" << endl;

cin >> n;

v=new int[n];

}

cout << "Введите элементы в вектор" << endl;

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

cin >> \*(v+i);

}

void vect :: print() // функция вывода вектора на экран

{

if(!v)

{

cout << "вектор пустой" << endl;

return ;

}

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

cout << setw(4) << \*(v+i);

cout << endl;

}

int vect :: funk1() // функция нахождения максимума в массиве

{

int max=\*v; // инициализация максимального элемента

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

if(max < \*(v+i))

max=\*(v+i);

return max;

}

int main()

{

setlocale(LC\_ALL,"Russian");

int rez; // максимальное число объекта

vect v1; // объявление объекта, вызывается конструктор

// без параметров

int ms[]={2,1,6,4,5}; // объявление и инициализация массива

vect v2((sizeof(ms)/sizeof(int)),ms); // объявление объекта, вызывается

// конструктор с параметрами

v1.set(); // функция инициализирует объект v1

cout << "Вектор v1" << endl;

v1.print(); // функция выводит на экран объект v1

cout << "Вектор v2" << endl;

v2.print(); // функция выводит на экран объект v2

rez=v2.funk1(); // функция возвращает максимальное // число объекта

cout << "Максимальное число = " << rez << endl;

return 0;

}

**Указатель this.** Каждый новый объект имеет скрытый от пользователя свой указатель. Иначе это можно объяснить так. Когда объявляется объект, под него выделяется память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах-функциях объекта можно с помощью ключевого слова this (рис. 2). Для любой функции, принадлежащей классу my\_class, указатель this неявно объявлен так:

my\_class \*const this;

Если объявлен класс и объекты, то размещение объектов в памяти будет выглядеть следующим образом:

class string

{ . . .} str1, str2;

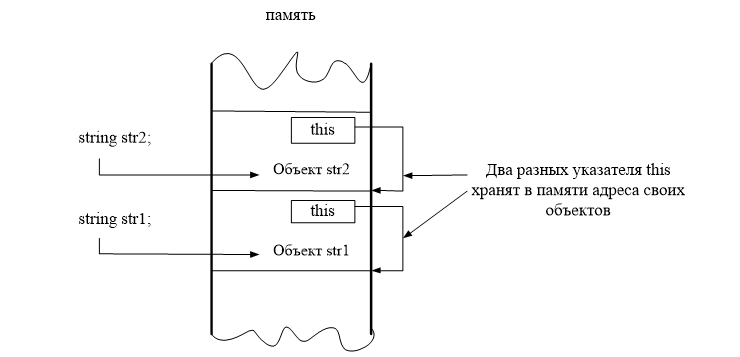


Рис. 2. Указатель this

**Основные свойства и правила использования указателя this:**

− каждый новый объект имеет свой скрытый указатель this;

− указывает на начало своего объекта в памяти компьютера;

− не надо дополнительно объявлять;

− передается как скрытый аргумент во все нестатические (т. е. не имеющие спецификатора static) компоненты-функции своего объекта;

− является локальной переменной, которая недоступна за пределами объекта (она доступна только во всех нестатических компонентах-функциях своего объекта);

− разрешается обращаться к указателю this непосредственно в виде this или \*this.

Пример использования указателя this.

#include<iostream>

using namespace std;

#include<string.h>

class String // объявление класса

{

char str[100];

int k;

int \*r;

public:

String() {k=123; r=&k;} // конструктор

void read(){ gets(str);} // функция чтения строки

void print() { puts(str); } // функция печати строки

void read\_print();

};

void String::read\_print()

{

char c;

cout << this->k << endl;

cout << "Введите строку" << endl;

// ключевое слово this содержит скрытый указатель на класс String,

// поэтому конструкция this->read() выбирает через указатель

// функцию read() этого класса

this->read();

cout << "Введенная строка" << endl;

this->print();

// ниже, в цикле for, одинаково удаленные от середины строки

// символы меняются местами

for(int i=0, j=strlen(str)-1; i<j; i++,j--)

{

c=str[i];

str[i]=str[j];

str[j]=c;

}

cout << "Измененная строка" << endl;

(\*this).print();

}

int main()

{

setlocale(LC\_ALL,"Russian");

String S; // объявление объекта

S.read\_print(); // вызов функции класса

}

**Static-члены (данные) класса.** Компоненты-данные могут быть объявлены с модификатором класса памяти static. Класс, содержащий static компоненты-данные, объявляется как глобальный (локальные классы не могут иметь статических членов). static-компонента совместно используется всеми объектами этого класса и хранится в одном месте. Статическая компонента глобального класса должна быть явно определена в контексте файла. **Основные правила использования статических компонент:**

− статические компоненты будут одними для всех объектов данного класса, т. е. ими используется одна область памяти;

− статические компоненты не являются частью объектов класса;

− объявление статических компонент-данных в классе не является их описанием. Они должны быть явно описаны в контексте файла;

− локальный класс не может иметь статических компонент;

− к статической компоненте st класса cls можно обращаться cls::st независимо от объектов этого класса, а также при помощи операций «.» и «->» при использовании объектов этого класса;

− статическая компонента существует даже при отсутствии объектов этого класса;

− статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

**Компоненты-функции static и const.** В С++ компоненты-функции могут использоваться с модификатором static и const. Обычная компонента-функция имеет явный список параметров и неявный список параметров. Неявные параметры можно представить как список параметров, доступных через указатель this. Статическая (static) компонента-функция не может обращаться к любой из компонент посредством указателя this. Компонента-функция const не может изменять неявные параметры.

Основные свойства и правила использования static- и const-функций:

− статические компоненты-функции не имеют указателя this, поэтому обращаться к нестатическим компонентам класса можно только с использованием «.» или «->»;

− не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов, чтобы при этом одна была статической, а другая нет;

− статические компоненты-функции не могут быть виртуальными.

Пример использования static, const данных и методов.

#include <iostream>

using namespace std;

class cls

{

int kl; // количество изделий

double zp; // зарплата на производство одного изделия

double nl1,nl2; // два налога на зарплату

double sr; // количество сырья на производство одного изделия

static double cs; // цена сырья на одно изделие

public:

cls(){} // конструктор по умолчанию

~cls(){} // деструктор

void inpt(int);

static void vvod\_cn(double);

double seb() const;

};

double cls::cs; // явное определение static-члена в контексте файла

void cls::inpt(int k)

{

kl=k;

cin >> nl1 >> nl2 >> zp;

}

void cls::vvod\_cn(double c)

{

cs=c; // можно обращаться в функции только к static-компонентам

}

double cls::seb() const

{

return kl\*(zp+zp\*nl1+zp\*nl2+sr\*cs); // в функции нельзя изменить ни один

} // неявный параметр (kl zp nl1 nl2 sr)

int main()

{

setlocale(LC\_ALL,"Russian");

cls c1,c2;

cout << "Введите зарплату и два налога для объекта с1" << endl;

c1.inpt(100); // инициализация первого объекта

cout << "Введите зарплату и два налога для объекта с2" << endl;

c2.inpt(200); // инициализация второго объекта

cls::vvod\_cn(500.);

cout << "\nc1 = " << c1.seb() << "\nc2 = " << c2.seb() << endl;

return 0;

}

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |
| **5** |  |
| **6** |  |
| **7** |  |
| **8** |  |
| **9** |  |
| **10** |  |

# **Лабораторная работа №3**

**Тема работы**: Дружественные функции и классы, перегрузка операторов

**Цель работы:** Понять назначение дружественных функций и классов, изучить принципы перегрузки бинарных и унарных операций.

**Теоретические материалы.**

**Дружественные функции.** Иногда возникает необходимость организации доступа к локальным данным нескольких классов из одной функции. Для реализации этого в С++ введен спецификатор friend. Если некоторая функция определена как friend-функция для некоторого класса, то эта функция называется дружественной и она:

− не является компонентом-функцией этого класса;

− имеет доступ ко всем компонентам этого класса (private, public и protected).

Пример использования дружественной функции.

#include <iostream>

using namespace std;

class kls

{

int i,j;

public:

kls(int I,int J) : i(I),j(J) {} // конструктор

int max() {return i>j? i : j;} // функция-компонент класса kls

friend double fun(int, kls&); /friend-объявление внешней функции fun

};

double fun(int i, kls &x) // описание дружественной функции

{

return (double)i/x.i;

}

int main()

{

kls obj(2,3); // объявление объекта

cout << obj.max() << endl;

cout << fun(3,obj) << endl; // вызов дружественной функции

return 0;

}

Функции со спецификатором friend, не являясь компонентами класса, не имеют и, следовательно, не могут использовать this указатель. Следует также отметить ошибочность следующей заголовочной записи функции

double kls :: fun(int i,int j),

т. к. fun не является компонентом-функцией класса kls.

В общем случае friend-функция является глобальной независимо от секции, в которой она объявлена (public, protected, private), при условии, что она не объявлена ни в одном другом классе без спецификатора friend. Функция friend, объявленная в классе, может рассматриваться как часть интерфейса класса с внешней средой.

Вызов компоненты-функции класса осуществляется с использованием операции доступа к компоненте «.» или «->». Вызов же friend-функции производится по ее имени. В friend-функцию не передается this-указатель и доступ к компонентам класса выполняется либо явно «.», либо косвенно «->».

Компонент-функция одного класса может быть объявлена со спецификатором friend для другого класса:

class X{ ……….

void fun (….);

};

class Y{ ……….

friend void X:: fun (….);

};

В приведенном фрагменте функция fun() имеет доступ к локальным компонентам класса Y. Запись вида friend void X:: fun (….) говорит о том, что функция fun принадлежит классу Х, а спецификатор friend разрешает доступ к локальным компонентам класса Y (т. к. она объявлена со спецификатором в классе Y).

Ниже приведен пример программы расчета суммы двух налогов на зарплату. Используется дружественная функция.

#include <iostream>

#include "string.h"

#include <iomanip>

using namespace std;

class nalogi; // неполное объявление класса nalogi

class work

{

char s[20]; // фамилия работника

int zp; // зарплата

public:

float raschet(nalogi); // компонент-функция класса work

void inpt()

{

cout << "вводите фамилию и зарплату" << endl;

cin >> s >> zp;

}

work(){}

~work(){};

};

class nalogi

{

float pd, // подоходный налог

st; // налог на социальное страхование

friend float work::raschet(nalogi); // friend-функция класса nalogi

public:

nalogi(float f1,float f2) : pd(f1),st(f2){};

~nalogi(void){};

};

float work::raschet(nalogi nl)

{

cout << s << setw(6) << zp <<endl; // доступ к данным класса work

cout << nl.pd << setw(8) << nl.st <<endl; // доступ к данным класса nalogi

return zp\*nl.pd/100+zp\*nl.st/100;

}

int main()

{

setlocale(LC\_ALL,"Russian");

nalogi nlg((float)12,(float)2.3); //описание и инициализация объекта

work wr[2]; // описание массива объектов

for(int i=0;i<2;i++)

wr[i].inpt(); // инициализация массива объектов

cout << setiosflags(ios::fixed) << setprecision(3) << endl;

cout << wr[0].raschet(nlg) << endl; // расчет налога для объекта wr[0]

cout << wr[1].raschet(nlg) << endl; // расчет налога для объекта wr[1]

return 0;

}

Следует отметить необходимость выполнения неполного (предварительного) объявления класса nalogi, т. к. в прототипе функции raschet класса work используется объект класса nalogi, объявляемого далее. Все функции одного класса можно объявить со спецификатором friend по отношению к другому классу следующим образом.

class X{ ………..

friend class Y;

………..

};

class Y{ ……….

};

В этом случае все компоненты-функции класса Y имеют спецификатор friend для класса Х (имеют доступ к компонентам класса Х). Класс Y является дружественным классу X.

В приведенном ниже примере оба класса являются дружественными.

#include <iostream>

using namespace std;

class A

{

int i; // компонент-данное класса А

public:

friend class B; // объявление класса В другом класса А

A():i(1){} // конструктор

~A(){} // деструктор

void f1\_A(B &); // метод, оперирующий данными обоих классов

};

class B

{

int j; // компонент-данное класса В

public:

friend A; // объявление класса А другом класса В

B():j(2){} // конструктор

~B(){} // деструктор

void f1\_B(A &a) // метод, оперирующий данными обоих классов

{

cout << a.i + j << endl;

}

};

void A :: f1\_A(B &b)

{

cout << i << ' ' << b.j<< endl;

}

int main()

{

A aa;

B bb;

aa.f1\_A(bb);

bb.f1\_B(aa);

}

Результат выполнения программы:

1 2

3

В объявлении класса А содержится инструкция friend class B, являющаяся и предварительным объявлением класса В и объявлением класса В дружественным классу А. Отметим также необходимость описания функции f1\_A после явного объявления класса В (в противном случае не может быть создан объект b еще не объявленного типа).

Отметим **основные свойства и правила использования спецификатора friend:**

− friend-функции не являются компонентами класса, но имеют доступ ко всем его компонентам независимо от их атрибута доступа;

− friend-функции не имеют указателя this;

− friend-функции не наследуются в производных классах;

− отношение friend не является *ни симметричным* (т. е. если класс А friend классу В, то это не означает, что В также является friend классу А), *ни транзитивным* (т. е. если A friend B и B friend C, то не следует, что A friend C);

− друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

**Перегрузка операторов.** Программы на языке С++ используют некоторые ранее определенные простейшие классы (типы), такие, как int, char, float и т. д. Мы можем описать объекты указанных классов, например:

int a,b;

char c,d,e;

float f;

Здесь переменные a, b, c, d, e, f можно рассматривать как простейшие объекты. В языке определены множество операций над простейшими объектами, выражаемых через операторы, такие, как «+», «-», «\*», «/», «%» и т. д. Каждый оператор можно применить к операндам определенного типа.

float a, b=3.123, c=6.871;

a=c+b; // нет ошибки

a=c%b; // ошибка

Второе является ошибочным, поскольку оператор «%» должен быть приложен лишь к объектам целого типа. Из этого следует, что операторы языка можно применить к тем объектам, для которых они были определены.

К сожалению, лишь определенное число типов непосредственно поддерживается любым языком программирования. Однако многие операторы можно определить через классы в С++. Рассмотрим пример.

Пусть заданы множества А и В:

А={а1,а2,а3}; B={a3,a4,a5};

Необходимо выполнить операции пересечение множеств «&» и объединение множеств «|»:

A&B={a1,a2,a3} & {a3,a4,a5}={a3};

A|B={a1,a2,a3} | {a3,a4,a5}={a1,a2,a3a4,a5};

Языки С/С++ не поддерживают непосредственно эти операции, однако в языке С++ можно объявить класс, назвав его set (множество). Далее можно определить операции над этим классом, выразив их с помощью знаков операторов, которые уже есть в языке С++, например «&» и «|». В результате операции «&» и «|» можно будет использовать как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения множеств). Как определить, какую функцию должен выполнять оператор: старую или новую? Надо посмотреть на тип операндов в соответствующем выражении. Если операнды – это объекты целого типа, то нужно выполнить операцию «побитового И» или «побитового ИЛИ». Если же операнды – это объекты типа set, то надо выполнить объединение или пересечение соответствующих множеств.

**Функция operator.** Функция operator может быть использована для расширения области приложения следующих операторов: «+», «-», «\*», «%», «&», «|» и т.д. Операторы, которые не могут быть перегружены: «.», «.\*», «::», «?:», sizeof и typeid.

Для перегрузки (доопределения) оператора разрабатываются функции, являющиеся либо компонентами, либо friend-функциями того класса, для которого они используются. Для того, чтобы перегрузить оператор, требуется определить действие этой оператора внутри класса. Общая форма записи функции-оператора являющейся компонентой класса, имеет вид

тип\_возв\_значения имя\_класса::operator#(список аргументов)

{

действия, выполняемые применительно к классу

}

Вместо символа «#» ставится значок перегружаемого оператора. Оператор всегда определяется по отношению к компонентам некоторого класса. В результате ее старое предназначение сохраняет силу. Функция operator является компонентом класса. При этом в случае унарного оператора функция operator не будет иметь аргументов, а в случае бинарного оператора будет иметь один аргумент. В качестве отсутствующего аргумента используется указатель this на тот объект, в котором определен оператор. Объявление и вызов функции operator осуществляется так же, как и любой другой функции. Единственное ее отличие заключается в том, что разрешается использовать сокращенную форму ее вызова. Так, выражение operator#(a,b), можно записать в сокращенной форме a#b.

Рассмотрим пример. Программа доопределяет значение оператора «&». В результате ее можно будет использовать для выполнения операции пересечения множеств.

#include<iostream>

using namespace std;

class set // класс «множество»

{

char str[80]; // строка

public:

set(char \*ss) // это конструктор

{

strcpy(str,ss);

}

char \* operator&(set); // объявление функции operator

};

char \* set::operator&(set S) // описание функции operator

{

int t=0, l=0;

while(str[t++]!='\0'); // вычисление длины строки

char \*s1=new char[t]; // выделение памяти под строку

for(int j=0; str[j]!='\0'; j++)

for(int k=0; S.str[k]!='\0'; k++)

if(str[j]==S.str[k])

{

s1[l]=str[j];

l++;

break;

}

s1[l]='\0';

return s1;

}

int main()

{

set S1="1f2bg5e6", S2="abcdef"; // задаются два множества

cout << (S1 & S2) << endl; // результат fbe

cout << (set("123") & set("426")) << endl; // результат 2

}

Ниже приведен пример программы перегрузки унарного оператора.

#include <iostream>

using namespace std;

class dek\_koord

{

int x,y; // декартовы координаты точки

public:

dek\_koord(){}; // конструктор без параметров

dek\_koord(int X,int Y) // конструктор с параметрами

{

x=X;

y=Y;

}

void operator++(); // перегрузка префиксного оператора ++

void operator++(int); // перегрузка постфиксного оператора ++

dek\_koord operator=(dek\_koord); // перегрузка оператора =

void see();

};

void dek\_koord::operator++() // определение перегрузки оператора ++A

{

x++;

}

void dek\_koord::operator++(int) // определение перегрузки оператора A++

{

y++;

}

dek\_koord dek\_koord::operator =(dek\_koord a)

{

x=a.x; // определение перегрузки оператора =

y=a.y;

return \*this;

}

void dek\_koord::see()

{

cout << "координата х = " << x << endl;

cout << "координата y = " << y << endl;

}

int main()

{

setlocale(LC\_ALL,"Russian");

dek\_koord A(1,2), B, C; // объявление объектов

A.see();

A++; // увеличение значения компонента х объекта А

A.see(); // просмотр содержимого объекта А

++A; // увеличение значения компонента у объекта А

A.see(); // просмотр содержимого объекта А

C=B=A; // множественное присваивание

B.see();

C.see();

return 0;

}

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |
| **5** |  |
| **6** |  |
| **7** |  |
| **8** |  |
| **9** |  |
| **10** |  |

# **Лабораторная работа №4**

**Тема работы**: Наследование, простое наследование.

**Цель работы:** Изучить принципы наследования.

**Теоретические материалы.**

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |
| **5** |  |
| **6** |  |
| **7** |  |
| **8** |  |
| **9** |  |
| **10** |  |