**17.04.2024 года Классы Объекты Указатели**

**Объектно-ориентированное программирование (ООП).** **Понятие класса, объекта. Присваивание, сравнение объектов. Конструкторы. Конструктор по умолчанию. Перегрузка конструкторов. Локальные и глобальные объекты Деструкторы. Массивы объектов. Указатели и объекты. Указатель this. Динамические объекты, массивы объектов. Ссылка на объект.**

**Объектно-ориентированное программирование**

В **ООП** программа представляется как ***совокупность объектов***. Каждый объект характеризуется *свойствами* и совершаемыми им *действиями* (поведением). Объекты, обладающие похожими свойствами и поведением, принадлежат к одним и тем же, или похожим ***классам****.*

Все языки **ООП**, включая и язык программирования **С++**, основаны на трёх основополагающих концепциях – *инкапсуляции, полиморфизма, наследования.*

**Понятие класса, объекта.**

***Класс*** представляет собой тип, определяемый пользователем:

**сlass** имя\_класса{

*тело класса*

};

Определяя класс, объявляются **данные**(*переменные, массивы, указатели* ит. д*.*), и **функции** (методы), предназначенные для выполнения действий над этими данными. Данные и функции – это **члены** класса.

Класс задаёт **формат** объекта. О**бъект** – это *экземпляр* класса или ***переменная*** *типа класс*. Для создания объектов используется ***оператор объявления***.

Класс задаёт целую категорию объектов. Все объекты класса имеют ***общие функции***, но каждый объект класса создаёт и поддерживает ***свои собственные данные***.

**Спецификаторы доступа к членам класса.**Каждый член класса обладает ***статусом доступа***, для указания которого используются *спецификаторы доступа* **private**и **public**, всегда заканчивающиеся *символом двоеточия* **(:),** причём, могут появляться в определении класса ***много раз*** и в ***любом порядке***.

**По умолчанию**статус доступа для членов классов – **private**(*закрытый*), Обычно данные класса объявляются как **private**-члены, а функции – как **public**-члены.

Открытые и закрытые члены доступны для функций **этого класса** *по имени,* то есть напрямую. Открытыечленыкласса ***доступны извне*** для любых функций в программе (в том числе и для функции **main()**), но только через объект. Закрытые члены класса ***недоступны извне****.* Доступом к закрытым членам класса извне можно эффективно управлять с помощью *открытых функций* класса, называемых ***функциями доступа***.

**Определение (описание) функций класса.** Функции класса могут быть определены **внутри** класса (**inline**-функции). И второй способ – в определение класса включить лишь ***прототипы*** функций (объявления), а саму функцию определить ***вне класса***.

Функции класса ***можно перегружать****,* но только с помощью *других функций класса* – просто задать в определении класса прототип для каждой версии функции, и каждую версию функции обеспечить отдельным определением.

Определение класса вводит свою ***область видимости***, которой принадлежат имена всех членов класса (*т.е. все члены класса имеют область видимости – класс*), поэтому наличие в двух разных классах данных и функций с одинаковыми именами ***не является ошибкой***.

**// Пример 1.** Определение класса. Создание и использование объектов.

// **Доступ** к полям объекта – используется ***операция точка*** (.).

// !!! **Имя класса ---- c большой буквы**!!

#include <iostream>

using namespace std;

class **First** {

int num; // **private-**переменная **num**

**public**:

void **setnum**(int x){num = x;} // установка **num,** прямой доступ к **num**

void **getnum();**

void **print();**

};

int **main(){**

**First** ob1, ob2; // создание объектов **ob1, ob2**

ob1.setnum(35);

ob1.print(); // num = **35**

ob2.getnum(); // например, введём **25**

ob2.print(); // num = **25**

}

void First::**getnum()**{ // определение **getnum**() вне класса

cout<<"Enter num: ";

cin>>num; // ввод значения **num,** прямой доступ к **num**

}

void First::**print()**{ // определение **print()** вне класса

cout<<"num = "<<num<<endl; // прямой доступ к **num** в функции **print()**

}

В **Примере** **1** класс **First** содержит **private-**переменную **num,** функции доступа к **num** --- **setnum()** и **getnum()**, а также функцию **print()** для вывода **num**.

**// Пример 2.** Переменные **класса, функций**. Область видимости

// переменных – **глобальных, локальных, класса**.

#include <iostream>

using namespace std;

int num = **777**; // глобальная переменная **num**

class **Samp{ //** класс **Samp**

int num; // **private**-переменная **num**  класса

**public:**

int b; // **public**-переменная **b** класса

void set(int x);

int get(){return num;} // функция **get() – inline**-функция

**};**

int **main(){**

Samp ob; // создание объекта **ob** типа **Samp**

ob.set(5); // установка **num** и **b**

cout**<<**"Function **main()** "<<endl;

cout<<" num-class = "<<ob.get()<<**' '**<<"b-class = "<<**ob.b**<<endl; // 5 25

cout<<" num-glob = " << ::num << endl; // **num** глобальная = **777**

cout<<" num-glob = " << num << endl; // можно опустить **::**

}

void Samp::**set**(int x){ // функция set()

cout**<<**"Function **set()** "<<endl;

num = x; // прямой доступ к **private**-**num**

b = x \* x; // прямой доступ к **public**-**b**

int b = 999; // локальная **b** в функции

cout<<" b-local = "<<b<<endl; // **b** локальнаяl = **999**

cout<<" b-class = "<<**Samp::b**<<endl; // **b** класса = **25**

cout<<" num-class = "<<num<<endl; // **num** класса = 5

cout<<" num-glob = " << ::num << endl; // **num** глобальная = **777**

}

В **Примере 2.** следует отметить следующее:

* если **имя глобальной переменной** не совпадает с именем переменной класса, например, имя не **num**, а **glob**, то в функции **set**() можно опустить в **cout** ***операцию разрешения области видимости*** **(::)** для вывода глобальной переменной **glob** --- cout<<glob.
* если **функция класса** определяет переменную с тем же именем, что и у переменной, определённой в классе, последняя становится ***невидимой*** в функции. И тогда такая *переменная* (открытая или закрытая) может быть доступна посредством *операции разрешения области видимости* **(::),** например, как в нашем примере **–** **Samp::b**.

**Присваивание, сравнение объектов**

Для присваивания объектов одного класса используется *операция присваивания* **(=).** Объекты после выполнения операции приобретают одинаковые значения данных, оставаясь при этом независимыми:

Samp ob1, ob2; ob1 = ob2;

Присваивать можно лишь *объекты* ***одного******типа*** (т.е. с одним именем типа).

Сравнение объектов даже одного типа выполнять **нельзя**, --- операции сравнения можно использовать только для полей объектов класса.

**Конструкторы**

**Конструктор без параметров.** В языке программирования С++ для инициализации объектов можно использовать особые функциикласса – **конструкторы. В Примере 3** показано, что конструктор может не иметь параметров, но его можно использовать для инициализации полей объекта.

**// Пример 3**. Использование в программе **конструктора.**

#include <iostream>

using namespace std;

class **Counter**{

unsigned int count; // переменная-счётчик

**public:**

Counter(){count = 0;} // **конструктор**

void incCount(){count++;} // увеличение счётчика

int getCount(){return count;} // получение значения счётчика

};

int **main()**{

**Counte**r ob1, ob2; // создание объектов, вызов конструктора

cout<<"ob1 = "<<ob1.getCount()<<endl; // ob1 = **0**

cout<<"ob2 = "<<ob2.getCount()<<endl; // ob2 = **0**

ob1.incCount();

ob2.incCount(); ob2.incCount();

cout<<"ob1 = "<<ob1.getCount()<<endl; // ob1 = 1

cout<<"ob2 = "<<ob2.getCount()<<endl; // ob2 = 2

}

При создании объектов **ob1**, **ob2** , конструктор **Counter()** вызывается ***автоматически***, и для каждого объекта поля-счётчики обнуляются. Конечно, можно было выполнить инициализацию с помощью, например, функцию **setCount()** с аргументом, равным нулю. Но в этом случае функцию надо было вызывать явно каждый раз при создании объекта**.**

**Конструктор –** это функция класса, которая *вызывается* *автоматически* при создании объекта и предназначена ***для инициализации его полей***.

Конструктор должен иметь то же**имя***,* что и **класс**. Конструктор не может возвращать значение, даже типа **void**. Конструктор должен быть ***открытым*** членом класса.

Конструкторы глобальных объектов вызываются *до вызова* функции **main()**. Для локальных объектов конструктор вызывается всякий раз при создании объекта.

**Конструктор с параметрами. К**ак и любая другая функция, конструктор может иметь параметры, которыми можно воспользуются для инициализации полей объекта определёнными значениями.

// **Пример 4.** Определение класса для работы с целыми числами.

// Нахождение суммы двух чисел. Использование **конструктора с параметрами**.

#include <iostream>

using namespace std;

class **Chisla**{ // определение класса **Chisla**

int num1, num2;

public:

Chisla(int x, int y) { num1 = x; num2 = y; } // конструктор с параметрами

// Chisla(int x, int y) : num1(x), num2(y){} // список **инициализаторов**

int sum() { return num1 + num2; } // **sum()** – сумма чисел

void show(); // **show()** – вывод чисел

};

int **main()** {

int number1, number2, sumChisel;

cout << "Enter 2 number: ";

cin >> number1 >> number2; // введём **5 6**

Chisla ob(number1, number2); // создание объекта **ob**

ob.show(); // num1 = 5 num2 = 6

sumChisel = ob.sum();

cout << "summa = " << sumChisel << endl; // summa = 11

}

void Chisla::**show()** {

cout << "num1 = " << num1 << " num2 = " << num2 << endl;

}

Если в классе определен конструктор, то при создании объекта этого класса, например, класса **Chisla** можно вызвать конструктор **явно**:

Chisla ob1 = Chisla(number1, number2); // **явный** вызов конструктора

Вызов конструктора получает значения для параметров и возвращает **объект** класса. Переменные класса **num1** и **num2** инициализируются теми значениями, которые передаются конструктору в качестве аргументов.

**Конструктор по умолчанию**

// **Пример 5.** Неявный конструктор **по умолчанию**.

#include <iostream>

using namespace std;

class **Samp** { // определение класса **Samp**

int num;

public:

void show() { cout << "num = " << num << endl; } // функция **show()**

};

int **main()** {

Samp ob; ob.show(); // **num** = -858993460

}

В программе **Примера** **5** нет конструктора, но объект ***создаётся*** и программа работает, т. е. ошибок нет. Почему? Это объясняется тем, что *компилятор автоматически* *встраивает* в программу **неявный** конструктор, который создаёт объект класса **ob** и инициализирует **num** значением по умолчанию:

**Samp::Samp(){}** // **неявный** конструктор по умолчанию

Если в классе не определено **явно****ни одного**конструктора, компилятор сгенерирует **неявный** конструктор **без параметров и с пустым телом**, который называется ***конструктом по умолчанию***.

Используя ***конструктор без параметров***, полям объекта можно присваивать начальные значения:

**Samp::Samp(){**a = 0**;}** // **явный** конструктор по умолчанию

**Явный конструктор по умолчанию** – это конструктор, который не имеет параметров или у которого все параметры заданы ***как* *аргументы по умолчанию***(причём параметры можно задавать ***в прототипе***функции-конструктора или в заголовке определения функции).

**!!!** Класс может иметь**только один *конструктор по умолчанию***.

// **Пример 6.** Определение класса для вычисления объёма куба.

// Использование **явного** **конструктора по умолчанию**.

#include <iostream>

using namespace std;

class **Cub** { // определение класса **Cub**

float len, wid, hei;

float v;

public:

Cub(int le = 1, int wi = 2, int he = 3); // конструктор по умолчанию

void show();

void volum();

};

int **main()** {

Cub ob1, ob2(4), ob3(4, 5), ob4(4, 5, 6);

ob1.volum(); ob1.show(); // 1 2 3 **v = 6**

ob2.volum(); ob2.show(); // 4 2 3 **v = 24**

ob3.volum(); ob3.show(); // 4 5 3 **v = 60**

ob4.volum(); ob4.show(); // 4 5 6 **v = 120**

}

Cub::Cub(int le, int wi, int he){

len = le; wid = wi; hei = he;

}

void Cub::show() {

cout << len << ' ' << wid << ' ' << hei<<" v = "<<v<<endl ;

}

void Cub::volum() {

v = len \* wid \* hei;

}

**Перегрузка конструкторов**

Конструкторы в языке С++ можно перегружать, чтобы обеспечить множество начальных значений *для инициализации объектов класса*.

// **Пример 7.** Нахождение суммы цифр натурального числа. Перегрузка

// **конструкторов** для случая инициализации объекта явно указанными

// значениями и значениями, заданными по умолчанию.

#include <ctime>

#include <iostream>

using namespace std;

class **Cifra**{ // определение класса **Cifra**

int num; // privite-**num**

int sum; // privite-**sum** -- сумма цифр

public:

Cifra(int x) { num = x; } // конструктор с параметром

Cifra(); // конструктор по умолчанию

void sumCifr();

void show();

};

int **main()** {

int number;

Cifra ob1;

ob1.sumCifr(); ob1.show(); // summa cifr **725** = **14**

cout << "Enter number: "; cin >> number; // **12345**

Cifra ob2(number);

ob2.sumCifr(); ob2.show(); // summa cifr **12345** = **15**

}

Cifra::Cifra() { // определение конструктор

srand(time(NULL));

num = rand() % 1000;

}

void Cifra::**sumCifr**() { // функция **sumCifr()**

sum = 0;

long rab = num;

while (rab) {

sum = sum + rab % 10; rab = rab / 10;

}

}

void Cifra::show() { // функция **show()**

cout << "summa cifr " << num << " = " << sum << endl;

}

Класс может иметь несколько конструкторов с разными параметрами *для разных видов* *инициализации*, т. е. их можно ***перегружать***.

**Локальные и глобальные объекты**

Объект, объявленный внутри блока или функции, является ***локальным объектом***. Для него ***область видимости*** (часть программы, где объект доступен) и ***время жизни*** (время, в течение которого объект сохранит своё значение) – **функция или блок**, где данный объект создан.

Объект, объявленный вне любого блока или функции, является ***глобальным объектом***. Для такого объекта ***область видимости* и *время жизни*** – **вся программа**, кроме того блока или функции, где данный объект переопределён.

**// Пример 8**. Локальные и глобальныеобъекты.

#include <iostream>

using namespace std;

class **Samp{** // определение класса **Samp**

**public:**

int num;

Samp () {}

Samp (int x){ num = x; }

**};**

Samp ob1; // глобальный объект **ob1**

Samp ob2(777); // глобальный объект **ob2**

Samp ob3(999); // глобальный объект **ob3**

int **main(){**

Samp ob1(555);

cout<< ob1. num << **' '<<::**ob1. num << endl; // **555 0**

Samp ob2(444);

cout<< ob2. num << **' '<<::**ob2. num <<endl; // **444 777**

cout<< ob3. num << endl; // **999**

}

**Деструкторы**

Деструктор – это функция, ***автоматически вызываемая*** при удалении объекта. Деструктор гарантирует, что память, выделенная для объекта, будет возвращена системе при уничтожении объекта.

Привыходе из функции объект (л*окальный объект*) оказывается *вне области видимости*, и для него вызывается деструктор. При завершениипрограммыдеструктор вызывается для глобальных объектов. Если в классе выделяется память с помощью операции **new**, то в деструкторе можно освободить подобную память.

Имя деструктора начинается c *символа* *тильды* **(~),** непосредственно за которым следует **имя класса**. Деструктор **не может** иметь параметров, **не может** возвращать значение. **Нельзя** перегружать деструктор. Класс может иметь ***один*** деструктор ***или ни одного***, в этом случае он создаётся компилятором как пустая функция (без параметров и с пустым телом).

// **Пример 9.** Использование **конструктора и деструктора**.

#include <iostream>

using namespace std;

class **Samp** { // определение класса **Samp**

int num;

public:

Samp() { num = 0; } // конструктор

void show() { cout << "num = " << num << endl; } // функция **show()**

~Samp() { cout << "Destructor" << endl; } // деструктор

};

int **main()** {

Samp ob; ob.show(); // **num = 0** Destructor

}

Деструктор можно вызвать и **явным** способом (в случае необходимости):

ob.~Samp(); ob.Samp::~Samp()**;**

**Массивы объектов**

Объекты – это переменные, и они имеют те же возможности и свойства, что и переменные любых других типов. Поэтому вполне допустимо создавать ***массивы объектов***. Каждый элемент массива – объект.

**// Пример 10.** Создание и использование одномерного массива объектов.

// Явная инициализация одномерного массива объектов.

// Использование конструктора по умолчанию и конструктора с параметром.

#include <iostream>

using namespace std;

class **Samp{** // определение класса **Samp**

int num;

**public:**

Samp(){} // конструктор по умолчанию

Samp(int x){num = x;} // конструктор с параметром

voidset(int x){num = x;} // функция установки **num**

int get(){return num;} // функция возврата **num**

**};**

int **main(){**

**const** int size = 4;

Samp ob1[size]; // массив объектов из **4**-х элементов

// вызов конструктора по умолчанию **4** раза

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

ob1[i].set(i\*i); // установка значений **num** объектов

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

cout<< ob1[i].get()<<' '; // **0 1 4 9**

cout<<endl;

Samp ob2[size] { 1, 2, 3, 4 }; // вызов конструктора c параметром **4** раза

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

cout<< ob2[i].get()<<' '; // **1 2 3 4**

}

**Указатели и объекты**

**Указатель на объект**. Это **переменная,** которая хранит адрес объекта в памяти компьютера:

**Samp** \* p; // объявление указателя на тип **Samp**

Помимо объявления, указатели требуют ***инициализации***, так как в противном случае они содержат *непредсказуемые* значения.

Доступ к отдельному **public**-полю объекта можно получить, используя *операцию* *точка* **(.) – прямой доступ,** или через указатель на объект, используя *операцию**стрелка* **(–>) –** **косвенный доступ**.

// **Пример 11**. Использование **указателя** на объект, массив объектов.

#include <iostream>

using namespace std;

class **Samp**{

int num; // **private**-**num** класса **Samp**

**public**:

Samp(){} // конструктор по умолчанию

Samp (int x){num = x;} // конструктор с параметром

void set(int x){num = x;} // функция установки **num**

int get(){return num;} // функция доступа, возвращает num

void show(); // функция вывода

};

int **main()**{

Samp ob1(1), ob2, \*p; // создание **ob1, ob2** и указателя **p**

ob1.show(); // x = 1

p = &ob2; // **инициализация указателя**

p->set(5); p->show(); // операция **(->) num** = 5

(\*p).set(55); (\*p).show(); // операция **(.) num** = 55

Samp array[] { 1, 2, 3, 4 }; // вызов конструктора **4** раза

p = array; // **p** = адрес массива **array**

**int n = sizeof(array) / sizeof(array[1]);**

for(int i = 0; i < n; i++) // чтение массива объектов

cout<< p[i].get()<<' '; // или (p+i)-> **get()**; (\*(p+i)). **get()**;

cout<<endl;// или array[i].get() **1 2 3 4**

}

void Samp::**show()**{ // определение **show()**

cout<<" num = "<<num<<endl; // прямой доступ к **private**-**num**

}

**Операции с указателями на объекты**.Все допустимые операции с указателями на объекты (++, -- и др.) выполняются относительно объекта. Например, результат прибавления к указателю *единицы* равносилен прибавлению значения*,* равного ***количеству байтов***в объекте, на который ссылается указатель, к исходному значению адреса. Указатели на объекты одного типа можно присваивать один другому, вычитать друг из друга, сравнивать. При этом сравниваются ***адреса***, а не значения величин, на которые данные указатели ссылаются. Например:

Samp ob(5); // создание объекта **ob**

Samp \*p, \*q; // **p, q** – указатели на тип **Samp**

p = &ob; // инициализация **p**

q = p; // **q** присваивается значение **p**

if(p == q) cout<<"\nYes"; else cout<<"\nNo";

**Указатель this**

Функции каждого объекта класса имеют доступ к указателю **this,** который*ссылается на сам объект***.** Указатель**thi*s*** *неявно передаётся каждой функции при её вызове*. Таким образом, любая функция может узнать адрес своего ***родного*** объекта. При этом следует отметить, что **\*this** представляет сам объект.

**// Пример 12.** Использование указателя **this** для доступа к полям объекта.

#include <iostream>

using namespace std;

class **Point**{

int x, y;

**public:**

Point(int x, int y){**this**->x = x; **this**->y = y;}

void show();

};

int **main()**{

Point ob(20, 50);

ob.show(); // **x = 20 y = 50 x = 20 y = 50**

}

void Point::show(){

cout<<"adres object = "<<**this**<<endl; // adres object = **00D8F9A4**

cout<<" x = "<< this->x<<"\t y = "<<this->y <<endl; // **20 50**

cout<<" x = "<<x<<"\t y = "<<y <<endl; // **20 50**

cout << (\*this).x << ' ' << (\*this).y << endl; // **20 50**

}

**Динамические объекты, массивы объектов.**

В языке программирования **C++** можно создавать *динамические объекты*. Объекты существуют до тех пор, пока не будут удалены явным образом. Динамические объекты размещаются в динамической области памяти.

Создание динамического объекта:

Samp \* p = **new** Samp;

При создании динамических объектов возможна их **инициализация**, если в классе есть **конструктор** с параметрами:

Samp \* p = **new** Samp(5, 6); // создание объекта с инициализацией

delete **p**; // освобождение памяти

Доступ к **public**-полю объекта, например, **у**:

p->y = 7; (\*p).y = 7;

Для доступа к **private**-полю объекта, например, **x**, используется функция доступа:

p->setx(1); (\*p). setx(1);

**Динамическими объекты** создаются и уничтожаются в процессе выполнения программы. Для доступа к членам объекта применяется *операция стрелка* (**->)** или *операция разыменования* указателя **(\*) и** *операция**точка* (.).

**// Пример 13.** Динамический массив объектов. Для доступа к элементам

// массива применяется ***операция стрелка*** (->) или ***операция точка*** (.)

#include <iostream>

using namespace std;

class **Samp{** // класс **Samp**

int num;

**public:**

Samp(){num = 0;} // конструктор по умолчанию

void set(int x){num = x;} // функция установки **num**

int get(){return num;} // возврат **num**

**};**

int **main(){**

int size; // **size** – размер динам. массива

cout<<"Enter size: "; cin>>size; // **4**

Samp\* p = new Samp[size]; // динамический массив объектов

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

cout<<(p+i)->get()<<' '; // **0 0 0 0**

// cout<<(\*(p+i)).get()<<' ';

}

cout<<endl;

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

p[i].set(i); // или (\*(p+i)).set(i);

cout<<p[i].get()<<' '; // **0 1 2 3**

}

delete [] **p**;

}

Можно использовать и другие способы организации циклов:

1. for (Samp \*t = p; t != p + size; t++){

(\*t).set(5);

cout <<t->get()<<' '; // **5 5 5 5**

**}**

1. for(int i = 0; i < size; i++){

p->set(i);

p++;

}

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

p--;

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

cout<<p[i].get()<<' '; // **0 1 2 3**

Следует отметить, что динамический массив объектов можно так же явно инициализировать, как и числовой динамический массив:

Samp\* p = new Samp[4] {1, 2, 3, 4};

В этом случае в класс необходимо включить конструктор с параметром;

Samp(int n){x = n;}

Если при явной инициализации инициализаторов меньше, чем размер массива

Samp\* p = new Samp[4] {1, 2};

то в классе должен быть конструктор по умолчанию, например:

Samp(){x = 0;}

Тогда результат инициализации --- 1 2 0 0. Иначе --- **ошибка**.

Если же инициализаторов больше, чем размер массива, то ---- **ошибка**.

Можно использовать и другие способы организации циклов:

// **Пример 14**. Определить класс для нахождения суммы цифр натурального

// числа **n**. Создать **динамический массив** объектов данного класса и для

// каждого элемента массива (объекта) найти сумму цифр числа.

// Числа вводятся в **main(). // Вариант 1.**

#include <iostream>

using namespace std;

class **Cifra**{

int num; // **private**-**num**

int sum; // **private-sum**

**public:**

Cifra(){} // конструктор по умолчанию **!!!**

Cifra(int x){ num = x;} // конструктор с параметром

void summa(); // функция суммы цифр числа

void show(); // функция вывода

};

int **main(){**

int size, number;

cout<<"Enter size array: "; cin>>size;

Cifra \* p = new Cifra[size]; // динамический массив объектов

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

cout<<"Enter number: "; cin>> number;

Cifra ob(number); // создание одного объекта

p[i] = ob;

}

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

p[i].summa();

}

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

p[i].show();

}

void Cifra::summa(){ // определение функции **summa()**

sum = 0;

int rab = num;

while(rab){

sum += rab % 10; rab /= 10;

}

}

void Cifra::show(){

cout<<"num = "<<setw(7)<<num<<setw(10)<<" summa cifr = "<<sum<<endl;

}

**Результат:**

Enter size array**: 2**

Enter number: 12345

Enter number: 5432

num = 12345 summa cifr = 15

num = 5432 summa cifr = 14

// **Пример 15**. Определить класс для нахождения суммы цифр числа **n**. Создать

// **динамический массив** объектов данного класса. Ввод чисел **в конструкторе**.

// **Вариант 2**;

#include <iostream>

using namespace std;

class **Cifra**{

int num;

int sum;

**public:**

Cifra();

void summa();

void show();

};

int **main(){**

int size;

cout<<" Enter size array: "; cin>>size;

Cifra\* p = new Cifra[size];  // динамический массив объектов

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

p[i].summa();

p[i].show();

}

}

Cifra::Cifra(){ // используется конструктор

cout<<"Enter number: "; cin>>num; // для инициализации массива

}

void Cifra::summa(){ // определение функции **summa()**

sum = 0; int rab = num;

while(rab){

sum = sum + rab % 10; rab = rab / 10;

}

}

void Cifra::show(){

cout<<"num = "<<setw(5)<<num<<setw(10)<<" summa cifr = "<<sum<<endl;

}

**Результат:** тот же, что и в **Примере 14**.

**Ссылка на объект**

**Э**то ещё одно имя объекта. Определение **ссылки** на объект **:**

Samp **ob**;

Samp **& ref** = ob; // **ob** и **ref** обозначают один и тот же объект

Определяя ссылку на объект, её необходимо сразу проинициализировать **объектом**, как показано выше. Нельзя определить ссылку как:

Samp **& ref**;

Нельзя переопределить ссылку в программе.

Для доступа к **public**-членам (данным и функциям) объекта через ссылку применяется *операция* *точки* **(.)**.

class **Samp{** // класс **Samp**

**public:**

int x;

};

int **main()**{

Samp **ob**; // объект **ob** класса **Samp**

ob.x = 4; // доступ через объект

Samp **&ref** = ob; // ссылка **ref** на объект **ob**

ref.x = 6; // доступ к **x** через ссылку

cout<<ob.x<<' '<<ref.x<<endl; // **6 6**

Samp \* p = &ob; // **p** – указатель на объект **ob**

p->x = 5; // доступ к **x** через указатель

cout<<ob.x<<' '<<ref.x<<' '<<p->x<<endl; // **5 5 5**

}