**Теория. Классы и объекты**

**Объектно-ориентированное программирование (ООП) – основные концепции.**

в **ООП** программа представляется в виде ***совокупности объектов***.

трёх основополагающих концепциях – *инкапсуляции, полиморфизма, наследования.*

**Определение класса.**

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

Определяя класс, объявляются **данные и код**

**Функции и данные – члены класса**

**Создание и использование объектов.**

класс задаёт ***формат объекта***. А объект представляет ***конкретную*** реализацию класса.

***Объект*** – это *экземпляр* класса или ***переменная*** *типа класс*.

Для создания объектов следует использовать ***оператор объявления***:

**First** ob1, ob2;

определение переменной данного типа соответствует **созданию объекта**

**Управление доступом к членам класса.**

Каждый член класса обладает ***статусом доступа***

*спецификаторы доступа* **private**и **public *много раз*** и в ***любом порядке***.

**Закрытые** члены класса могут быть доступны ***только* для функций этого класса.**

*операцию точки* **(.) для доступа к объекту**

***функциями доступа***. Можно управлять доступом к закрытым членам класса.

**Определение функций класса.**

Функции класса могут быть определены внутри класса, как функция **print(**) класса **First**. Так и снаружи, но внутри будет прототип этой функции тогда используют *разрешения области видимости* void First::**print**

Функции класса, ***в отличие*** от других функций, ***имеют доступ*** к **private**-членам класса (закрытым членам класса).

Функции класса ***можно перегружать****,* но только с помощью *других функций класса*.

Определение класса вводит свою ***область видимости***, это влияет на задание названий функций.

**Присваивание объектов Локальные и глобальные объекты.**

Объект, объявленный внутри блока или функции, является ***локальным объектом***. Для него *область видимости* *время жизни* **функция или блок**, где данный объект определён.

вне любого блока и вне любой функции, является ***глобальным объектом***. *область видимости* и *время жизни* – **вся программа**

**Конструкторы, правила их использования.**

***Конструктор* –** это специальная 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()** и всякий раз при создании объекта.

Конструктор может иметь параметры

Инициализируем используя ***список инициализации*:**

Chisla(int x, int y) : a(x), b(y){}

Такая Инициализация выполнится ***до начала выполнения тела конструктора*** чтобы задать начальное значение ***константе***.

**Конструктор по умолчанию.**

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

**Перегрузка конструкторов.** C объявл и без

**Деструкторы, правила их использования.**

*автоматически вызываемая функция* при уничтожении объекта, называется *деструктором*

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

ob.~Samp();

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

**для объек** *вне области видимости* вызывается деструктор.

* 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).

**Объектно-ориентированное программирование (ООП).** Предлагает новый подход к созданию программ. Вместо того чтобы рассматривать программу как набор последовательно выполняемых инструкций, в **ООП** программа представляется в виде ***совокупности 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 **С++**, основаны на трёх основополагающих концепциях – *инкапсуляции, полиморфизма, наследования.*

**Определение класса. *Класс*** представляет собой тип, *определяемый пользователем*:

**сlass** имя\_класса{

*тело класса*

};

Определяя класс, объявляются **данные**(*переменные, массивы, указатели и т.д.*), которые он содержит, и **код** (*набор функций*), предназначенный для выполнения действий над этими данными. Функции и данные – это ***члены*** (или ***поля***) класса***.*** Например, определим простейший класс **First**:

class **First**{

**public**:

int num;

void print(){cout<<"num = "<<num<<endl;}

};

В классе **First** переменная **num** и функция **print**() – ***члены класса***.

**Создание и использование объектов.**Класс предназначен для описания некоторого *типа объектов*, т.е. по сути, класс задаёт ***формат объекта***. А объект представляет ***конкретную*** реализацию класса.

***Объект*** – это *экземпляр* класса или ***переменная*** *типа класс*.

Для создания объектов следует использовать ***оператор объявления***:

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

Определение класса соответствует введению ***типа*** пользователя, а определение переменной данного типа – ***созданию объекта***. Класс задаёт целую категорию объектов. Все объекты класса имеют ***общие функции***, но каждый объект класса создаёт и поддерживает ***свои собственные данные***.

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

**По умолчанию**режим доступа для членов классов – **private**(*закрытый*), поэтому все члены класса после заголовка класса и до **первого** *спецификатора* *доступа* считаются *закрытыми*. Для объявления *открытых* членов класса используется *спецификатор доступа* **public**(*открытый)*. Определённый режим доступа действует до следующего спецификатора или до конца определения класса. Обычно *данные* класса объявляются как **private**-члены, а *функции* – как **public**-члены.

Закрытые члены класса могут быть доступны ***только***для функций этого класса. Открытые члены класса могут быть *доступны извне*, для любых функций в программе (в том числе и для функции **main()**).

После создания объектов **ob1**и **ob2** класса **First** доступ к открытым членам классаиз функции, например, **main()** можно получить *напрямую* через объект, используя *операцию точки* **(.)**, например, **ob1.num** = 35; **ob2.num** = 20**; ob1.print.**

Изменим определение класса **First**, сделав переменную **num** закрытой:

class **First** {

int num;

**public**:

void **print()**{cout<<"num = "<<num<<endl;}

};

В этом случае операторы

ob1.num = 35; ob2.num = 25;

приводят к ошибке компиляции, так как **num –** *закрытая переменная* и к ней из функции **main()** **нет** **прямого доступа**. Доступом к закрытым членам класса можно эффективно управлять с помощью *открытых функций* класса, называемых ***функциями доступа***.

**Определение (описание) функций класса.** Функции класса могут быть определены внутри класса, как функция **print(**) класса **First**.

Определение функций ***внутри класса*** не является обязательным. В определение класса можно включить лишь ***прототипы*** функций (объявления), а саму функцию определить ***вне класса*** (в другом месте программы). В этом случае перед именем функции следует записать имя класса и *операцию разрешения области видимости* **(::)**, которая указывает, какой *области видимости* (т.е. какому классу) принадлежит функция.

Например, определение функции **print()** вне класса **First** может иметь следующий вид:

void First::**print()**{

cout<<"num = " <<num<<endl;

}

Для вызова функций классадля конкретного объекта также следует использовать *операцию точки* **(.)**, связывающую функцию с именем объекта, например:

ob.**print()**; // вызов функции **print()**

First::ob.**print()**; // вызов функции **print()**

Функции класса, ***в отличие*** от других функций, ***имеют доступ*** к **private**-членам класса (закрытым членам класса).

Функции класса ***можно перегружать****,* но только с помощью *других функций класса*. При этом следует просто задать в определении класса прототип для каждой версии перегруженной функции, и каждую версию функции обеспечить отдельным определением (описанием).

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 членам класса.

#include <iostream>

using namespace std;

class **First** {

int num;

**public**:

void **setnum**(int x){num = x;} // установка значения **num**

void **getnum()**{

cout<<"Enter num: "; // ввод с клавиатуры

cin>>num;

}

void **print();**

};

void First::**print()**{

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

}

int **main(){**

First ob1, ob2;

ob1.setnum(35);

ob1.print(); // num = 35

ob2.getnum(); // 25

ob2.print(); // num = 25

}

**// Пример 2.** Переменные класса и переменные функций.

#include <iostream>

using namespace std;

int **xGlob** = **777**; // глобальная **xGlob**

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

int a; // **private**-переменная **a**

**public:** // **public**-члены

int b;

void setab(int n);

int geta(){return a;} // функция **geta() – inline**-функция

**};**

void Samp::**setab**(int n){ // функция **setab()**

a = n; // прямой доступ к переменной класса **a**

b = n \* n; // прямой доступ к переменной класса **b**

int b = 999; // локальная переменная **b**

cout<<"b-localnaya = "<<b<<endl; // **b**- localnaya = **999**

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

cout<< "xGlob = " << **::xGlob** << endl; // xGlob = **777**

}

int **main(){**

Samp ob; // объект **ob** типа **Samp**

ob.setab(5); // установка **a** и **b**

cout<<ob.geta()<<**' '**<<**ob.b**<<endl; // **a** = 5 **b** = 25

ob.b = 55; // прямой доступ к **b**

cout<<ob.geta()<<**' '**<<**ob.b**<<endl; // **a** = 5 **b** = 55

cout<< "xGlob = " << **::xGlob** << endl; // xGlob = **777**

}

**Локальные и глобальные объекты.** Объект, объявленный внутри блока или функции, является ***локальным объектом***. Для него *область видимости* (та часть в программе, где объект доступен) и *время жизни* (время, в течение которого, объект хранит своё значение) – **функция или блок**, где данный объект определён.

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

**Присваивание объектов.** Для присваивания одного объекта другому объекту используется *операция присваивания* **(=).** Объекты после выполнения операции присваивания 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, а не типов, одинаковых физически).

**Конструкторы.** В примере **2** для инициализации полей объекта используется функция **setab(**). Но, конечно, удобнее инициализировать поля объекта автоматически в момент его создания, а не явно вызывать в программе соответствующую функцию. Такой способ инициализации реализуется в **С++** с помощью *особой функции класса*, называемой **конструктором.**

**// Пример 3**. Использование 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;

cout<<"\nnob1 = "<<ob1.getCount(); // **0**

cout<<"\nnob2 = "<<ob2.getCount(); // **0**

ob1.incCount();

ob2.incCount();

ob2.incCount();

cout<<"\nnob1 = "<<ob1.getCount(); // 1

cout<<"\nnob2 = "<<ob2.getCount(); // 2

cout<<endl;

}

Когда в программе создаются объекты **ob1**, **ob2** класса, конструктор **Counter()** вызывается автоматически, и для каждого объекта счётчики обнуляются. Конечно, можно было выполнить инициализацию с помощью, например, функции **setCount()** с аргументом, равным нулю. Но в этом случае функцию пришлось бы вызывать явно каждый раз при создании объекта типа **Counter.**

В процедурной программе на **С++,** счётчик, скорее всего, был бы представлен *глобальной переменной*, что небезопасно, или *статической переменной*. В нашем примере используется *переменная-счётчик* класса, значение которой может быть изменено только с помощью собственной функции класса.

***Конструктор* –** это специальная 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.*** Конструктор может иметь параметры, как любая другая функция, которыми можно воспользоваться 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 a, b;

public:

Chisla(int x, int y) { a = x; b = y; } // конструктор с параметрами

int sum() { return a + b; } // функция суммы чисел

void show(); // функция вывода чисел

};

void Chisla::show() { cout << "a = " << a << " b = " << b << endl; }

int **main()** { // функция main()

int n, m, s;

cout << "vvedi 2 chisla: "; cin >> n >> m;

Chisla ob(n, m); // объект ob класса Chisla

ob.show(); // вызов функций show() и sum()

s = ob.sum();

cout << "summa = " << s << endl;

}

**Результат:**

vvedi 2 chisla: **5 6**

a = 5 b = 6

summa = 11

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

cout<<"vvedi 2 chisla: "; cin>>n>>m;

Chisla ob1 = Chisla(n, m); // явный вызов конструктора

Вызов конструктора получает значения для параметров и возвращает объект класса. Поля **a** и **b** инициализируются теми значениями, которые передаются конструктору в качестве аргументов.

***Список инициализации конструктора.*** Одной из наиболее возлагаемых на конструктор задач является инициализация полей объекта класса. В **примерах 3** и **4** для каждого объекта класса конструкторы выполняют инициализацию полей объектов в теле конструктора.

Однако рекомендуется выполнять инициализацию, используя ***список инициализации*:**

Chisla(int x, int y) : a(x), b(y){} // тело конструктора – **пустое**!!!

Список инициализации представляет перечисления ***инициализаторов*** для каждого из полей класса **a** и **b** через двоеточие после списка параметров конструктора.

Инициализация полей класса с помощью списка инициализации происходит ***до начала выполнения тела конструктора***, что в некоторых ситуациях бывает важно. Так, например, список инициализации – это единственный способ задать начальное значение ***константе***.

**// Пример 5.** Использование списка инициализации.

#include <iostream>

using namespace std;

class Samp {

public:

const int x; // const-переменная x

Samp(int n) :x(n) {}; // конструктор

};

int **main()** {

Samp ob(15); // создание объекта

cout << "x = " << ob.x << endl; // x = 15

}

**Конструктор по умолчанию.** Определим простейший класс без конструктора и создадим объект класса:

// **Пример 6.** Конструктор по умолчанию.

#include <iostream>

using namespace std;

class Samp { // определение класса Samp

int a;

public:

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

};

int main() {

Samp ob; // создание объекта ob

ob.show(); // a = -858993460

}

В программе нет конструктора, но объект создаётся и программа работает, т.е. ошибок нет. **Почему?** Это объясняется тем, что *компилятор автоматически* *встраивает* в программу конструктор, который создаёт объект класса **ob** и инициализирует его поле **a** 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***.

Но иногда появляется необходимость присваивать начальные значения полям объекта и при использовании *конструктора без параметров*. Тогда следует **явно** определить конструктор по умолчанию:

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

Конструкторс параметрами также может быть***конструктором по умолчанию***, но для этого все параметры конструктора должны быть заданы как ***аргументы по умолчанию*,** причём такие параметры можно задавать ***в прототипе***функции-конструктора внутри определения класса или***в заголовке*** определения *функции*. В этом случае в программе не должно быть другого конструктора по умолчанию:

**Samp::Samp(**int n = 0, int b = 0){a = n**;** b = m;}

**Явный конструктор по умолчанию** – это конструктор, который не имеет параметров или у которого все параметры заданы *как* *аргументы по умолчанию*. Класс может иметь***только один конструктор по умолчанию***.

// **Пример 7.** Определение класса для вычисления объёма куба.

// Использование конструктора по умолчанию.

#include <iostream>

using namespace std;

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

int len, wid, hei;

public:

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

void show() { cout << endl << len << ' ' << wid << ' ' << hei; }

int volum() { return len \* wid \* hei; }

};

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

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

}

int **main()** {

Cub ob1, ob2(3), ob3(5, 4, 3);

ob1.show(); cout << " v = " << ob1.volum(); // 1 1 1 v = 1

ob2.show(); cout << " v = " << ob2.volum(); // 3 1 1 v = 3

ob3.show(); cout << " v = " << ob3.volum() << endl; // 5 4 3 v = 60

}

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

// **Пример 8**. Нахождение суммы цифр натурального числа. Перегрузка

// конструктора для случая инициализации объекта с явно указанными

// значениями и значениями, заданными по умолчанию.

#include <ctime>

#include <iostream>

using namespace std;

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

long chislo; // число

int sum; // сумма цифр

public:

Cifra(long n) { chislo = n; } // конструктор с параметром

Cifra();

void summa();

void show();

};

Cifra::Cifra() { // конструктор по умолчанию

srand(time(NULL));

chislo = rand() % 1000;

}

void Cifra::summa() { // функция summa() – сумма цифр

sum = 0;

long rab = chislo;

while (rab) {

sum = sum + rab % 10;

rab = rab / 10;

}

}

void Cifra::show() { // функция show() – вывод на консоль

cout << "summa cifr chisla " << chislo << " = " << sum << endl;

}

int **main()** {

int n;

Cifra ob1;

ob1.summa();

ob1.show();

cout << "vvedi chislo: "; cin >> n;

Cifra ob2(n);

ob2.summa(); ob2.show();

}

**Результат:**

summa cifr chisla **725** = 14

vvedi chislo: 12345

summa cifr chisla **12345** = 15

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

**Деструкторы*.*** Конструктор вызывается при создании объекта. Существует другая функция, *автоматически вызываемая* при уничтожении объекта, которая называется *деструктором*. Деструктор имеет имя, совпадающее с именем конструктора (а следовательно, и класса) и предваряющееся *символом* **(~)**. Деструктор выполняет освобождение использованных объектом ресурсов, и гарантирует, что память, выделенная под объект, 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.

**Правила использования деструктора:**

* 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 a;

public:

Samp() { a = 0; } // конструктор

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

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

};

int **main()** {

Samp ob; // создание объекта **ob**, вызов конструктора

ob.show(); // a = 0

ob.~Samp();

ob.Samp::~Samp();

}

**Результат:**

a = 0

Destructor

Destructor

Destructor

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

ob.~Samp();

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