КЛАССЫ

- Класс – это дальнейшее развитие понятия структуры. Он позволяет создавать новые типы и определять функции, манипулирующие с этими типами.
- Объект - это представитель определенного класса.
- ООП использует механизмы инкапсуляции, полиморфизма и наследования.

- Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется ***инкапсуляцией.***
- ***Наследование*** позволяет одному объекту наследовать свойства другого объекта, т.е. порожденный класс наследует свойства родительского класса и добавляет собственные свойства.
- ***Полиморфизм*** - возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

- Рассмотрим реализацию понятия даты с использованием структуры для того, чтобы определить представление даты date и множества функций для работы с переменными этого типа:

```
struct date
{
 int month, day, year;
void set(int, int, int);
void get(int*, int*, int*);
void next();
void print();
};
```

- Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними.

Описание класса

```
class <имя>{
  [ private: ]
    <описание скрытых элементов>
  public:
    <описание доступных элементов>
}; // Описание заканчивается точкой с запятой
```

Поля класса

- могут быть простыми переменными любого типа, указателями, массивами и ссылками (т.е. могут иметь практически любой тип, кроме типа этого же класса, но могут быть указателями или ссылками на этот класс);
- могут быть константами (описаны с модификатором const), при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- Инициализация полей при описании не допускается.

```
class monster
{
int health, ammo;
public:
  monster(int he = 100, int am  = 10)
    { health = he; ammo = am;}
  void draw(int x, int y, int scale, int position);
  int get_health(){return health;}
  int get_ammo(){return ammo;}};
```

- В этом классе два скрытых поля - health и ammo, получить значения которых извне можно с помощью методов get_health() и get_ammo().

- Методы класса имеют неограниченный непосредственный доступ к его полям. Внутри метода можно объявлять объекты, указатели и ссылки как своего, так и других классов.
- В приведенном классе содержится три определения методов и одно объявление (метод draw). Если тело метода определено внутри класса, он является встроенным (inline).

- В каждом классе есть метод, имя которого совпадает с именем класса. Он называется конструктором и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

Описание объектов

- Конкретные переменные типа данных "класс" называются экземплярами класса, или объектами.

```
monster Vasia; // Объект класса monster с параметрами по умолчанию
monster Super(200, 300);// Объект с явной инициализацией
monster stado[100]; // Массив объектов с параметрами по умолчанию
/* Динамический объект (второй параметр задается по умолчанию) */
monster *beavis = new monster (10);
monster &butthead = Vasia; // Ссылка на объект
```

- При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор .

- Доступ к открытым (public) элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель.

- Например:

```
int n = Vasia.get_ammo();
stado[5].draw;
cout << beavis->get_health();
```

- Наследование – возможность порождать новые классы на базе уже имеющихся с передачей порожденным классам наследства в виде данных и членов-функций родительского класса (или классов, если прямых родителей несколько).
- Порожденные классы имеют возможность расширять набор данных, полученных по наследству, модифицировать родительские методы и функции, создавать новые данные и новые функции их обработки.

Базовый и производный классы

- Принято называть родительский класс базовым, а вновь созданный класс – производным.

- Механизм наследования (inheritance) предусматривает две возможности.

  - В первом случае, который называют простым наследованием, родительский класс один.
  - Во втором случае родителей два или больше, и соответствующий процесс именуют термином множественное наследование

Простое наследование

```
class D: [virtual][public|private|protected] B
        {
        тело производного класса
    };
```

- Чаще всего производный класс конструируют по следующей схеме, которая носит название открытого наследования:

```
class D: public B {тело производного класса};
```

Уровень доступа в B	Уровень доступа при объявлении D	Уровень доступа в D
public	опущен	private
protected	опущен	private
private	опущен	нет доступа
public	public	public
protected	public	protected
private	public	нет доступа
public	protected	protected
protected	protected	protected
private	protected	нет доступа
public	private	private
protected	private	private
private	private	нет доступа

```
#include <iostream>
#include <conio.h>
class B {
  int x;
public:
  B(){x=0; cout<<"Def_B "<<endl;}
  B(int n){x=n; cout<<"Init_B "<<endl;}
  B(const B &y){x=y.x; cout<<"Copy_B "<<endl;}
  int get_x(){return x;}
  ~B(){cout<<"Destr_B"<<endl;}
};
```

```
class D : public B {
  int y;
public:
  D(){y=0; cout<<"Def_D "<<endl;}
  D(int n){y=n; cout<<"Init_D "<<endl;}
  D(const D &z){y=z.y; cout<<"Copy_D "<<endl;}
  int get_y(){return y;}
  ~D(){cout<<"Destr_D"<<endl;}
};
```

```
void main()
{ B w1;
  cout<<"w1.x="<<w1.get_x()<<endl;
  B w2(2);
  cout<<"w2.x="<<w2.get_x()<<endl;
  B w3(w2);
  cout<<"w3.x="<<w3.get_x()<<endl;
  B w4=w1;
  cout<<"w4.x="<<w4.get_x()<<endl;
  D q1;
  cout<<"q1.x="<<q1.get_x()<<' '<<"q1.y="<<q1.get_y()<<endl;
  D q2(2);
  cout<<"q2.x="<<q2.get_x()<<' '<<"q2.y="<<q2.get_y()<<endl;
  D q3(q2);
  cout<<"q3.x="<<q3.get_x()<<' '<<"q3.y="<<q3.get_y()<<endl;
  D q4=q1;
  cout<<"q4.x="<<q4.get_x()<<' '<<"q4.y="<<q4.get_y()<<endl;
}
```

Динамическое создание и удаление объектов

```
class A {...};     //объявление класса
..............
A *ps=new A;       //объявление указателя и
  создание объекта типа A
A *pa=new A[20];  //объявление указателя и
  создание массива объектов
..............
delete ps;         //удаление объекта по указателю
  ps
delete [] pa;      //удаление массива объектов по
  указателю pa
```

- Создание одиночных объектов может быть совмещено с инициализацией объекта, если в классе предусмотрен соответствующий конструктор:

A *ptr1=new A(5);//создание объекта и вызов конструктора инициализации

- Массив создаваемых объектов проинициализировать таким же образом нельзя.

- Довольно распространенная ситуация, которая может оказаться потенциальным источником ошибок, возникает в процессе создания и удаления динамических объектов. Она заключается в том, что после уничтожения объекта, связанного, например, с указателем ps, этот указатель не чистится. Если после удаления объекта сделать попытку что-то прочитать или записать по этому указателю, то поведение программы предсказать трудно. Поэтому достаточно разумным правилом является засылка нуля в указатель разрушаемого объекта:

```
delete ps;
ps=NULL;     //или ps=0;
```

Виртуальные функции

- Виртуальными называют функции базового класса, которые могут быть переопределены в производном классе.
- В базовом классе их заголовок начинается со служебного слова virtual.

- Рассмотрим пример, в котором базовый класс B содержит защищенное поле n и отображает его содержимое на экране. Производный класс D1 отображает квадрат доставшегося по наследству поля. Еще один класс D2, порожденный тем же родителем B, отображает куб своего наследства.

```
#include <iostream>
#include <conio.h>
class B {
public:
  B(int k):n(k){}   //конструктор инициализации
  virtual void show(){cout<<n<<endl;}     //виртуальная
   функция
protected:
  int n;
};
```

```
class D1: public B {
public:
  D1(int k):B(k){} // конструктор
   инициализации
  virtual void show(){cout<<n*n<<endl;}
};
class D2: public B {
public:
  D2(int k):B(k){} // конструктор
   инициализации
  virtual void show(){cout<<n*n*n<<endl;}
};
```

```
void main()
{
  B bb(2),*ptr;
  D1 dd1(2);
  D2 dd2(2);
  ptr=&bb;
  ptr->show();
  ptr=&dd1;
  ptr->show();
  ptr=&dd2;
  ptr->show();
}
```

Чистые виртуальные функции и абстрактные классы

- Чистая виртуальная функция не совершает никаких действий, и ее описание выглядит следующим образом:

```
virtual тип name_f(тип1 a1,тип2 a2,...)=0;
```

- Класс, содержащий хотя бы одно объявление чистой виртуальной функции, называют абстрактным классом. Для такого класса невозможно создавать объекты, но он может служить базовым для других классов, в которых чистые виртуальные функции должны быть переопределены.

- Объявим абстрактным класс Shape (Геометрическая Фигура), в состав которого включим две чистые виртуальные функции – определение площади фигуры (Get_Area) и определение периметра фигуры (Get_Perim).

```
class Shape {
public:
  Shape(){}   //конструктор
  virtual double Get_Area()=0;
  virtual double Get_Perim()=0;
};
```

```
class Rectangle: public Shape {
  double w,h;    //ширина и высота
public:
  Rectangle(double w1,double h1):w(w1),h(h1) {}
  double Get_Area() {return w*h;}
  double Get_Perim() {return 2*w+2*h;}
};
class Circle: public Shape {
  double r;   //радиус
public:
  Circle(double r1):r(r1) {}
  double Get_Area() {return M_PI*r*r;}
  double Get_Perim() {return 2*M_PI*r;}
};
```

- Если в производном классе хотя бы одна из чисто виртуальных функций не переопределяется, то производный класс продолжает оставаться абстрактным и попытка создать объект (экземпляр класса) будет пресечена компилятором.

Множественное наследование и виртуальные классы

- О множественном наследовании говорят в тех случаях, когда в создании производного класса участвуют два или более родителей:

```
class B1 {//первый базовый класс
  int x;
public:
  B1(int n):x(n) {cout<<"Init_B1"<<endl;}
   //конструктор B1
  int get_x(){return x;}
  ~B1() {cout<<"Destr_B1"<<endl;}
   //деструктор B1
};
```

```
class B2 {//второй базовый класс
  int y;
public:
  B2(int n):y(n) {cout<<"Init_B2"<<endl;}      // конструктор
   B2
  int get_y(){return y;}
  ~B2() {cout<<"Destr_B2"<<endl;}            //деструктор B2
};
class D: public B1, public B2 {
  int z;
public:
  D(int a,int b,int c):B1(a),B2(b),z(c)
   {cout<<"Init_D"<<endl;}                  //конструктор D
  void show() {cout<<"x="<<get_x()<<" y="<<get_y()<<"
   z="<<z<<endl;}
  ~D() {cout<<"Destr_D"<<endl;}              //деструктор D
};
```

```
#include <iostream.h>
void main()
{
  D qq(1,2,3);
  qq.show();
}

                    //=== Результат работы ===
Init_B1
Init_B2
Init_D
x=1 y=2 z=3
Destr_D
Destr_B2
Destr_B1
```

- Последовательность обращений к конструкторам родительских классов определяется очередностью их вызовов в списке инициализации. Если таковой отсутствует, то определяющим является порядок перечисления родителей в объявлении производного класса.