Класс и *Объект*

Класс - принципиально новый тип данных.

Класс представляет собой множество объектов

-имеющих общую структуру

-обладающих одинаковым поведением.

Класс является дальнейшим развитием типа структура (запись)

•**Объект** является представителем (**экземпляром**) какого-либо класса.

•Объект **обладает**

–состоянием

–поведением

–идентичностью.

•**Состояние объекта** характеризуется

– набором его свойств (атрибутов)

– текущими значениями каждого из этих свойств.

•**Поведение объекта** - выполнения определенной последовательности характерных для него действий.

•**Идентичность объекта** – это свойство (или набор свойств) объекта, которое позволяет отличить его от всех прочих объектов того же типа (класса).

Объект объединяет в себе как описывающие его данные (свойства), так и средства обработки этих данных (методы).

Если объект — это “существительное”, свойства объекта — “прилагательные”, методы — “глаголы”.

Управление доступом

Ключи доступа

private - элементы данных могут использоваться только функциями-методами класса, к которому принадлежат эти элементы данных

public - элементы данных могут использоваться любыми функциями программы

protected - элементы данных могут использоваться функциями-методами того же класса, к которому принадлежат эти элементы данных, а также функциями-методами производных классов (классов-потомков)

По умолчанию *ключ доступа* ***private****. Т.е. если ключи доступа не указаны, то все элементы класса являются скрытыми (недоступными).*

Попытка обратиться в программе к скрытым данным или методам вызывает сообщение:

<имя элемента класса> is not accessible

Имя_класса с этого момента становится новым именем типа данных, которое используется для объявления объектов класса.

Члены класса - это переменные состояния и методы этого класса, иными словами членами класса могут быть как переменные, так и функции. Функции и переменные, объявленные внутри объявления класса, становятся членами этого класса. Функции-члены класса будем называть методами этого класса.

По умолчанию, все функции и переменные, объявленные в классе, становятся закрытыми **(private)** . Т.е. они доступны только из других членов этого класса. Для объявления открытых членов класса используется ключевое слово **public**. Все функции-методы и переменные, объявленные после слова **public**, доступны и для других членов класса, и для любой другой части программы, в которой содержится класс. В структурах по умолчанию все члены являются отрытыми.

Данные примеры аналогичны:

```
struct _3d
    {
     double mod ();
     double projection (_3d r);
    private:
     double x, y, z;
    };
```

```
class _3d
    {
     double mod ();
     double projection (_3d r);
    private:
     double x, y, z;
    };
```

Для определения метода - члена класса, нужно связать имя класса, частью которого является метод, с именем класса. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называют ***операцией расширения области видимости***.

```
double _3d::mod ()
{
 return sqrt (x*x + y*y +z*z);
}
double _3d::projection (_3d r)
{
 return (x*r.x + y*r.y + z*r.z) / mod();
}
...
main()
{
 _3d a, b;
 double dPro, dMod;
 dMod = a.mod();
 dPro = b.projection(a);
}
```

Инкапсуляция (encapsulation)

объединение производного ***типа данных с набором функций***, используемых при работе с этим типом данных, в единый класс.

- Функции, включенные в класс, называют **методами класса**
- Данные – элементами или **полями класса**,
- Конкретные представители класса – **объекты, экземпляры**.

Класс (объект) - это то, что поддерживает инкапсуляцию

- Инкапсуляция позволяет сделать класс «самодостаточным» для решения конкретной задачи.
- Класс всегда несет в себе некоторую функциональность.
- Это мощное средство обмена готовыми к работе программными заготовками

Инкапсуляция и ограничение доступа к данным

- Инкапсуляция предполагает возможность ограничения доступа к данным (полям) класса.
- Это позволяет
  - **упростить интерфейс** класса, показав наиболее существенные для внешнего пользователя данные и методы.
  - обеспечить **возможность внесения изменений** в реализацию класса без изменения других классов (важно для дальнейшего сопровождения и модернизации программного кода).
- При сокрытии полей объекта **доступ** к ним осуществляется **только** посредством **методов класса**. Это защищает данные от внешнего вмешательства или неправильного использования (“чёрный ящик”)

Перегружаемые операторы

```
struct _3d
    {
     double x, y, z; //
    координаты
     _3d operator + (_3d);
    };
    _3d _3d::operator + (_3d b)
    {
     _3d c;
     c.x = x + b.x;
     c.y = y + b.y;
     c.z = z + b.z;
     return c;
    }
```

```
_3d A, B, C;
    A.x = 1.0;
    A.y = 1.0;
    A.x = 1.0;
    B = A;
    C = A + B; // это
    эквивалентно вызову
     C = A.operator + (B);

    /operator +(...) - один из
    методов объекта A/
```

Ограничения на замещение операторов по сравнению с замещением функций:

- оператор должен уже существовать в языке (нельзя добавлять совершенно новые, выдуманные вами операторы);
- нельзя переопределять действия встроенных в C++ операторов при работе со встроенными типами данных; так, нельзя изменить способ работы оператора "+" при сложении целых чисел;
- запрещено замещать операторы ".", ".*", "?:", "::" и символы препроцессора "#".

Встраиваемые функции

```
inline _3d _3d::operator + (_3d b)
   {
    _3d c;
    c.x = x + b.x;
    c.y = y + b.y;
    c.z = z + b.z;
    return c;
   }
```

Используется для простых и коротких методов, которые в дальнейшем не предполагается изменять.

```
struct _3d
    {
     double x, y, z;
     double mod () {return sqrt (x*x + y*y +z*z);}
     double projection (_3d r) {return (x*r.x + y*r.y +
    z*r.z) / mod();}
     _3d operator + (_3d b);
    };
```

С точки зрения ООП такой стиль программирования должен быть признан ошибочным:

```
_3d vectorA;
double m;
vectorA.x = 17.56;
vectorA.y = 35.12;
vectorA.z = 1.0;
m = vectorA.mod();
```

- ```
  class _3d
  {
   double x, y, z;
  public:
   double mod () {return sqrt (x*x + y*y +z*z);}
   double projection (_3d r) {return (x*r.x + y*r.y + z*r.z) /
  mod();}
   void set (double newX, double newY, double newZ)
   {
    x = newX; y = newY; z = newZ;
   }
   _3d operator + (_3d b);
  };
  ```
  ```

Конструкторы и деструкторы

C++ дает возможность создать метод, который будет автоматически вызываться для инициализации объекта данного типа при его создании. Такой метод называется ***конструктором***. Конструктор определяет, как будет создаваться новый объект, когда это необходимо, может распределить под него память и инициализировать ее. Он может включать в себя код для распределения памяти, присваивание значений элементам, преобразование из одного типа в другой и многое полезное.

Конструкторы в языке C++ имеют имена, совпадающие с именем класса. Конструктор может быть определен пользователем, или компилятор сам сгенерирует конструктор по умолчанию. Конструктор может вызываться явно, или неявно. Компилятор сам автоматически вызывает соответствующий конструктор там, где Вы определяете новый объект класса. Конструктор не возвращает никакое значение, и при описании конструктора не используется ключевое слово void.

Функцией, обратной конструктору, является ***деструктор***. Эта функция обычно вызывается при удалении объекта. Например, если при создании объекта для него динамически выделялась память, то при удалении объекта ее нужно освободить. Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы.

В языке C++ деструкторы имеют имена: "*~имя_класса*". Как и конструктор, деструктор не возвращает никакое значение, но в отличие от конструктора не может быть вызван явно. Конструктор и деструктор не могут быть описаны в закрытой части класса.

```
class _3d
      {
       double x, y, z;
      public:
       _3d();
       ~_3d()
       {
        cout << 'Работа деструктора _3d \n';
       }
       double mod () {return sqrt (x*x + y*y +z*z);}
       double projection (_3d r) {return (x*r.x +
      y*r.y + z*r.z) / mod();}
       void set (double newX, double newY, double
      newZ)
       {
        x = newX; y = newY; z = newZ;
       }
       };
_3d::_3d() // конструктор класса _3d
      {
       x=y=z=0;
       cout << 'Работа конструктора _3d \n';
      }
```

```
main()
      {
       _3d A; // создается объект A и
      происходит инициализация его
      элементов
        // A.x = A.y = A.z = 0;
       A.set (3,4,0);
        // Теперь A.x = 3.0, A.y = 4.0, A.z = 0.0
            cout << A.mod()<<'\n';
       }
```

Результат работы программы:

Работа конструктора _3d

5.0

Работа деструктора _3d

Конструкторы с параметрами и перегрузка конструкторов

```
class _3d
    {
     double x, y, z;
    public:
     _3d ();
     _3d (double initX, double initY,
    double initZ);
     ...
    };
_3d::_3d(double initX, double initY,
    double initZ)
    //конструктор класса _3d с
    параметрами
    {
     x = initX;
     y = initY;
     z = initZ;
     cout << 'Работа конструктора _3d
    \n';
    }
```

```
main()
    {
     _3d A; //создается объект A и
    происходит инициализация его
    элементов
     // A.x = A.y = A.z = 0;
     A.set (3,4,0); //Теперь A.x = 3.0, A.y
    = 4.0, A.z = 0.0
     _3d B (3,4,0); //создается объект
    B и происходит инициализация
    его элементов
     // B.x = 3.0, B.y = 4.0, B.z = 0.0
    }
```

Такой способ вызова конструктора является сокращенной формой записи выражения

```
_3d B = _3d (3,4,0);
```

Присваивание объектов

```
class ClassName1
    {
     int a, b;
    public:
     void set (int ia, int ib) {a=ia;
    b=ib;}
    };
class ClassName2
    {
     int a, b;
    public:
     void set (int ia, int ib) {a=ia;
    b=ib;}
    };
```

попытка выполнить

- ClassName1 c1;
  ClassName2 c2;
  c2 = c1;

окажется неудачной.

```
class Pair
      {
       int a, *b;
      public:
       void set (int ia, int ib) {a=ia;
      *b=ib;}
       int getb (){return *b;}
       int geta (){return a;}
      };
main()
      {
       Pair c1,c2;
       c1.set(10,11);
       c2 = c1;
       c1.set(100,111);
       cout << 'c2.b = '<< c2.getb();
      }
```

При присваивании происходит побитное копирование элементов данных, в том числе и массивов. Особенно внимательным нужно быть при присваивании объектов, в описании типа которых содержатся указатели.

В результате работы программы получим "c2.b = 111", а не 11, как ожидалось.

Перегруженный оператор присваивания

```
class _3d
{
 double x, y, z;
public:
 _3d ();
 _3d (double initX, double initY, double initZ);
 double mod () {return sqrt (x*x + y*y +z*z);}
 double projection (_3d r) {return (x*r.x + y*r.y + z*r.z) / mod();}
 _3d operator + (_3d b);
 _3d operator = (_3d b);
};
_3d _3d::operator = (_3d b)
{
x = b.x;
y = b.y;
z = b.z;
return *this;
}
```

Передача в функции и возвращение объекта

```
class ClassName
      {
      public:
       ClassName ()
       {
        cout << 'Работа конструктора \n';
       }
       ~ClassName ()
       {
        cout << 'Работа деструктора \n';
       }
      };
void f (ClassName o)
      {
       cout << 'Работа функции f \n';
      }
main()
      {
       ClassName c1;
       f (c1);
      }
```

Эта программа выполнит следующее

- Работа конструктора
  Работа функции f
  Работа деструктора
  Работа деструктора

Конструктор вызывается только один раз. Это происходит при создании c1. Однако деструктор срабатывает дважды: один раз для копии **o**, второй раз для самого объекта **c1**. Тот факт, что деструктор вызывается дважды, может стать потенциальным источником проблем, например, для объектов, деструктор которых высвобождает динамически выделенную область памяти.

```
class ClassName {
      public:
       ClassName ()
       {
        cout << 'Работа конструктора \n';
       }
       ~ClassName ()
       {
        cout << 'Работа деструктора \n';
       }
      };
ClassName f()
      {
       ClassName obj;
       cout << 'Работа функции f \n';
       return obj;
      }
main()
      {
       ClassName c1;
       c1 = f();
      }
```

Эта программа выполнит следующее

- Работа конструктора
  Работа конструктора
  Работа функции f
  Работа деструктора
  Работа деструктора
  Работа деструктора

Конструктор вызывается два раза: для **c1** и **obj**. Однако деструкторов здесь три. Понятно, что один деструктор разрушает c1, еще один - **obj**. "Лишний" вызов деструктора (второй по счету) вызывается для так называемого *временного объекта,* который является копией возвращаемого объекта.

Конструктор копирования

Позволяет точно определить порядок создания копии объекта и имеет следующую форму:

имя_класса (const *имя_класса* & obj)
{
... // тело конструктора
}

Здесь **obj** - это ссылка на объект или адрес объекта.

Замечание: конструктор копирования не влияет на операцию присваивания.

```
class ClassName
    {
    public:
     ClassName ()
     {
      cout << 'Работа конструктора \n';
     }
     ClassName (const ClassName& obj)
     {
      cout << 'Работа конструктора
    копирования\n';
     }
     ~ClassName ()
     {
      cout << 'Работа деструктора \n';
     }
    };
main()
    {
     ClassName c1; // вызов
    конструктора
     ClassName c2 = c1; // вызов
    конструктора копирования
    }
```

Наследование (inheritance)

- это **возможность определять новые классы** посредством добавления полей, свойств и методов к **уже существующим классам**.

Такой механизм получения новых классов называется ***порождением***.

При этом новый, порожденный, класс (**потомок**) **наследует все** поля, методы и свойства своего базового, родительского класса.

Наследование поддерживает концепцию **иерархии классов** (hierarchical classification).

Применение иерархии классов делает управляемыми большие потоки информации.

Наследование **обеспечивает поэтапное создание сложных классов** и разработку собственных библиотек классов

- Класс, который наследуется, называется базовым классом. Наследующий класс называют производным классом.

- Новый класс строится на базе уже существующего с помощью конструкции следующего вида:

  class Parent {....};
  class Child : [модификатор наследования] Parent {....};

Модификатор наследования

Модификатор доступа	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	нет доступа	нет доступа	нет доступа

Конструкторы и деструкторы при наследовании

Если и у базового и у производного классов есть конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы - в обратном порядке.

Т.е. если A базовый класс, B - производный из A, а C - производный из B (A-B-C), то при создании объекта класса C вызов конструкторов будет иметь следующий порядок: **конструктор A - конструктор B - конструктор C**.

Вызов деструкторов при разрушении этого объекта произойдет в обратном порядке:

деструктор C - деструктор B - деструктор A.

```
class BaseClass
    {
    public:
     BaseClass() {cout << ' Работа
    конструктора базового класса
    \n';}
     ~BaseClass() {cout << ' Работа
    деструктора базового класса \n';}
    };
class DerivedClass: public BaseClass
    {
    public:
     DerivedClass() {cout << ' Работа
    конструктора производного
    класса \n';}
     ~DerivedClass() {cout << ' Работа
    деструктора производного
    класса \n';}
    };
main()
    {
     DerivedClass obj;
    }
```

Эта программа выводит следующее:

Работа конструктора базового класса

Работа конструктора производного класса

Работа деструктора производного класса

Работа деструктора базового класса

Расширенная запись конструктора производного класса

```
конструктор_производного_класса
(список формальных параметров)
 : конструктор_базового_класса
(список фактических параметров)
{
 ... // тело конструктора
производного класса
}
```

```
class BaseClass
{
 int i;
public:
 BaseClass (int ii) {i=ii;}
 ~BaseClass() {cout << ' Работа деструктора базового класса \n';}
};
class DerivedClass: public BaseClass
{
 int n;
public:
 DerivedClass (int nn, int m): BaseClass (m) {n=nn;}
 ~DerivedClass() {cout << ' Работа деструктора производного класса
\n';}
};
main()
{
 DerivedClass obj(2,3);
}
```

Допускается также, что конструктор базового класса может иметь больше параметров, чем конструктор производного класса.

```
class BaseClass
{
 int j, i;
public:
 BaseClass (int jj, int ii) {j=jj; i=ii;}
 ~BaseClass() {cout << ' Работа деструктора базового класса \n';}
};
class DerivedClass: public BaseClass
{
 int n;
public:
 DerivedClass (int nn);
 ~DerivedClass() {cout << ' Работа деструктора производного класса \n';}
};
DerivedClass :: DerivedClass (int nn): BaseClass (nn/2, nn%2)
{ n=nn; }
main()
{
 DerivedClass obj(15);
}
```