# Введение в объектно-ориентированное программирование

Дисциплина «Процессы разработки программ»

- Исторически сложилось так, что программирование возникло и развивалось как процедурное программирование, которое предполагает, что основой программы является алгоритм, процедура обработки данных.

- **Объектно-ориентированное программирование** (ООП) — это методика разработки программ, в основе которой лежит понятие объект.
- **Объект** — это некоторая структура, соответствующая объекту реального мира, его поведению.

- Задача, решаемая с использованием методики ООП, описывается в терминах объектов и операций над ними, а программа при таком подходе представляет собой набор объектов и связей между ними.

- Язык C++ позволяет программисту определять свои собственные сложные типы данных — записи (struct).
- Язык C++, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять классы.
- **Класс** — это сложная структура, включающая, помимо описания данных, описание процедур и функций, которые могут быть выполнены над представителем класса — объектом.

# Объявление простого класса

Объявление классов в C++

```
class /*имя класса*/
{
  private:
  /* список свойств и методов для использования внутри класса */
  public:
  /* список методов доступных другим функциям и объектам программы */
  protected:
  /*список средств, доступных при наследовании*/
};
```

- Описание класса помещают в программе в раздел описания типов (Type).
- Объекты как представители класса объявляются в программе в разделе student.TPerson;

  professor.TPerson;

В C++ объект — это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных.

Выделение памяти осуществляется при помощи специального метода класса — конструктора, которому обычно присваивают имя Create (создать).

Для того чтобы подчеркнуть особую роль и поведение конструктора, в описании класса вместо слова procedure используется слово constructor.

```
class CppStudio // имя класса
{
public: // спецификатор доступа
    void message() // функция (метод класса) выводящая сообщение на экран
    {
        cout << "website: cppstudio.com\ntheme: Classes and Objects in C + +\n";
    }
}; // конец объявления класса CppStudio
```

Выделение памяти для данных объекта происходит путем присваивания значения результата применения метода-конструктора к типу (классу) объекта.

Например, после выполнения инструкции

```
professor = TPerson.Create;
```

выделяется необходимая память для данных объекта professor.

Помимо выделения памяти, конструктор, как правило, решает задачу присваивания полям объекта начальных значений, т. е. осуществляет инициализацию объекта.
Ниже приведен пример реализации конструктора для объектов A и B :

```
class AB //класс
{
   private:
   int a;
   int b;
   public:
   AB()    //это конструктор:  1) у конструктора нет типа возвращаемого значения! в том числе
     void!!!
   //   2) имя должно быть таким как и у класса (в нашем случае AB)
   {
      a = 0;//присвоим начальные значения переменным
      b = 0;
      cout << "Работа конструктора при создании нового объекта: " << endl;//и здесь же их
    отобразим на экран
      cout << "a = " << a << endl;
      cout << "b = " << b << endl << endl;
   }
```

Реализация конструктора несколько необычна

- . Во-первых, в теле конструктора нет привычных инструкций New, обеспечивающих выделение динамической памяти (всю необходимую работу по выделению памяти выполняет компилятор).
- Во-вторых, формально конструктор не возвращает значения, хотя в программе обращение к конструктору осуществляется как к методу-функции.

- После объявления и инициализации объект можно использовать, например, установить значение поля объекта.

- Доступ к полю объекта осуществляется указанием имени объекта и имени поля, которые отделяются друг от друга точкой.
- Хотя объект является ссылкой, правило доступа к данным с помощью ссылки, согласно которому после имени переменной, являющейся ссылкой, надо ставить значок ^, на объекты не распространяется.

Например, для доступа к полю fname объекта professor вместо professor^.fname надо писать

professor.fname

Очевидно, что такой способ доступа к полям объекта более естественен.

- Если в программе какой-либо объект больше не используется, то можно освободить память, занимаемую полями данного объекта.
- Для выполнения этого действия используют метод-деструктор Free. Например, для того, чтобы освободить память, занимаемую полями объекта professor, достаточно записать

```
professor.Free;
```

# Метод

- Методы класса (процедуры и функции, объявление которых включено в описание класса) выполняют действия над объектами класса.
- Для того чтобы метод был выполнен, необходимо указать имя объекта и имя метода, отделив одно имя от другого точкой.

Например, инструкция

```
professor.Show;
```

вызывает применение метода show к объекту professor.

Фактически инструкция применения метода к объекту — это специфический способ записи инструкции вызова процедуры.

Методы класса определяются в программе точно так же, как и обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, состоит из двух частей:

- имени класса, к которому принадлежит метод, и
- имени метода.

Имя класса от имени метода отделяется точкой.

Пример определения метода show класса TPerson

// метод Show класса TPerson

```
void TPerson.Show();
 begin
   ShowMessage( 'Имя:' + fname + #13
    + 'Адрес:' + faddress );
```

В инструкциях метода доступ к полям объекта осуществляется без указания имени объекта.

# Инкапсуляция и свойства объекта

- Под инкапсуляцией понимается скрытие полей объекта с целью обеспечения доступа к ним только посредством методов класса.

- В языке C++ ограничение доступа к полям объекта реализуется при помощи свойств объекта.
- Свойство объекта характеризуется полем, сохраняющим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства.
- Метод установки значения свойства называется методом записи свойства (write), а метод получения значения свойства — методом чтения свойства (read).

- В описании класса перед именем свойства записывают слово property (свойство).
- После имени свойства указывается его тип, затем — имена методов, обеспечивающих доступ к значению свойства.
- После слова read указывается имя метода, обеспечивающего чтение свойства, после слова write — имя метода, отвечающего за запись свойства.

```
type
    TName = string[15];
     TAddress = string[35];

TPerson = class                                          // класс
private
    FName: TName;                                       // значение свойства Name
    FAddress: TAddress;                                 // значение свойства Address
Constructor Create(Name:Tname);
Procedure Show;
Function GetName:TName;
Function GetAddress:TAddress;
Procedure SetAddress(NewAddress:TAddress);
public
    property Name:TName read GetName;  // свойство Name
                                       // доступно только для чтения
     property Address:TAddress read GetAddress // свойство Address
        write SetAddress;                //  доступно для чтения и записи
end;
```

В программе для установки значения свойства не нужно записывать инструкцию применения к объекту метода установки значения свойства, а надо записать обычную инструкцию присваивания значения свойству.

Например, чтобы присвоить значение свойству Address объекта student, достаточно записать

```
student.Address = “С.Петербург, ул.Садовая 21, кв.3”;
```

Компилятор перетранслирует приведенную инструкцию присваивания значения свойству в инструкцию вызова метода

```
student.SetAddress(“С.Петербург, ул.Садовая 21, кв.3”);
```

- Внешне применение свойств в программе ничем не отличается от использования полей объекта. Однако между свойством и полем объекта существует принципиальное отличие: при присвоении и чтении значения свойства автоматически вызывается процедура, которая выполняет некоторую работу.

- В программе на методы свойства можно возложить некоторые дополнительные задачи. Например, с помощью метода можно проверить корректность присваиваемых свойству значений, установить значения других полей, логически связанных со свойством, вызвать вспомогательную процедуру.

- Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта: например, можно разрешить только чтение.
- Для того чтобы инструкции программы не могли изменить значение свойства, в описании свойства надо указать лишь имя метода чтения.
- Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку времени компиляции.
- В приведенном выше описании класса TPerson свойство Name доступно только для чтения, а свойство Address — для чтения и записи.

- Установить значение свойства, защищенного от записи, можно во время инициализации объекта.

Ниже приведены методы класса TPerson, обеспечивающие создание объекта класса TPerson и доступ к его свойствам.

```
Constructor TPerson.Create(Name:TName);   // конструктор объекта TPerson
   begin
    FName:=Name;
   end;

Function TPerson.GetName;                 // метод получения значения свойства Name
   begin
    Result:=FName;
   end;

Function TPerson.GetAddress;              // метод получения значения свойства Address
   begin
    Result:=FAddress;
   end;

 Procedure TPerson.SetAddress(NewAddress:TAddress);  // метод изменения
                                                     // значения свойства Address
   begin
    if FAddress =' ‘  then   FAddress := NewAddress;
   end;
```

Приведенный конструктор объекта TPerson создает объект и устанавливает значение поля FName, определяющего значение свойства Name.

- Инструкции программы, обеспечивающие создание объекта класса TPerson и установку его свойства, могут быть, например, такими:

```
student = TPerson.Create(“Иванов”);
student.Address = “ул. Садовая, д.3, кв.25”;
```

# Наследование

- Концепция ООП предполагает 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 свойства и методы своего базового, родительского класса.

- В объявлении класса-потомка указывается класс родителя.

Например, класс TEmployee (сотрудник) может быть порожден от рассмотренного выше класса TPerson путем добавления поля FDepartment (отдел).

Объявление класса TEmplioyee в этом случае может выглядеть так:

```
TEmployee = class(TPerson)
    FDepartment: integer;                    // номер отдела
    constructor Create(Name:TName; Dep:integer);
end;
```

Класс TEmpioyee должен иметь свой собственный конструктор, обеспечивающий инициализацию класса-родителя и своих полей.

Пример реализации конструктора класса TEmployee:

```
constructor TEmpioyee.Create(Name:Tname; Dep:integer);
  begin
   inherited Create(Name);
   FDepartment:=Dep;
  end;
```

В приведенном примере директивой inherited вызывается конструктор родительского класса. После этого присваивается значение полю класса-потомка.

После создания объекта производного класса в программе можно использовать поля и методы родительского класса.

Ниже приведен фрагмент программы, демонстрирующий эту возможность.

```
engineer = TEmployee.Create(“Сидоров”,413);
engineer.address = “ул.Блохина, д.8, кв.10”;
```

Первая инструкция создает объект типа TEmployee,

Вторая — устанавливает значение свойства, которое относится к родительскому классу.

# Директивы области видимости: protected и private

- Помимо объявления элементов класса (полей, методов, свойств) описание класса, как правило, содержит директивы protected (защищенный) и private (закрытый), которые устанавливают степень видимости элементов класса в программе.

- Элементы класса, объявленные в секции protected, доступны только в порожденных от него классах. Область видимости элементов класса этой секции не ограничивается модулем, в котором находится описание класса. Обычно в секцию protected помещают описание методов класса.

- Элементы класса, объявленные в секции private, видимы только внутри модуля. Эти элементы не доступны за пределами модуля, даже в производных классах. Обычно в секцию private помещают описание полей класса, а методы, обеспечивающие доступ к этим полям, помещают в секцию protected.

Описание класса TPerson, в которое включены директивы управления доступом

```
TPerson = class
private
   FName: TName; // значение свойства Name
    FAddress: TAddress; // значение свойства Address
protected
   Constructor Create(Name:TName);
    Function GetName: TName;
    Function GetAddress: TAddress;
    Procedure SetAddress(NewAddress:TAddress);
    Property Name: TName read GetName;
    Property Address: TAddress read GetAddress
    write SetAddress;
end;
```

Полиморфизм — это возможность использовать одинаковые имена для методов, входящих в различные классы.
Концепция полиморфизма обеспечивает при применении метода к объекту использование именно того метода, который соответствует классу объекта.

Например, пусть определены три класса, один из которых является базовым для двух других:

```
type
    // базовый класс TPerson
    TPerson = class
        FTame : string;  { имя }
        constructor Create(name : string) ;
        function info: string; virtual;
    end;

    // производный от базового TPerson
    TStud = class(TPerson)
        FGr : integer; { номер группы }
        constructor Create (name : string; gr: integer);
        function info: string; override
    end;

    // производный от базового TPerson
    TProf = class(TPerson)
        FDep : string; { название кафедры }
        constructor Create(name : string; dep : string);
        function info: string; override;
    end;
```

- В каждом из этих классов определен метод info. В базовом классе при помощи директивы virtual метод info объявлен виртуальным.
- Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным.
- В каждом дочернем классе определен свой метод info, который замещает соответствующий метод родительского класса (метод порожденного класса, замещающий виртуальный метод родительского класса, помечается директивой override).

- Для вызова родительского метода в перекрытом методе можно использовать директиву inherited.

# Абстрактные и перегруженные методы

- Виртуальные методы могут также быть абстрактными, что означает, что в данном классе не существует реализации метода. Данный метод должен быть реализован в порожденных классах. Для описания абстрактного метода используется ключевое слово abstract, следующее после virtual.

- Методы могут быть перегружены – то есть можно определить несколько методов с одним и тем же именем (с добавлением в конце ключевого слова overload):

```
double  function Min(double A, double B; overload;
int function Min(int A, int B; overload;
```

- Данное объявление говорит о том, что существует две функции Min, принимающие различные параметры. Выбор нужной функции осуществляется компилятором в зависимости от типа переданных параметров.

# Классы и объекты Delphi

- Для реализации интерфейса C++ использует библиотеку классов, которая содержит большое количество разнообразных классов, поддерживающих форму и различные компоненты формы (командные кнопки, поля редактирования и т. д.).
- Во время проектирования формы приложения C++ автоматически добавляет в текст программы необходимые объекты. Если сразу после запуска C++ просмотреть содержимое окна редактора кода, то там можно обнаружить следующие строки:

```
type
TForm1 = class(TForm)
private
{ Private declarations }
public
{ Public declarations }
end;
var
Form1: Tform1
```

Это описание класса исходной, пустой формы приложения и объявление объекта — формы приложения.

- Когда программист, добавляя необходимые компоненты, создает форму, C++ формирует описание класса формы.
- Когда программист создает функцию обработки события формы или ее компонента, C++ добавляет объявление метода в описание класса формы приложения.

- Помимо классов визуальных компонентов в библиотеку классов входят и классы так называемых невизуальных (невидимых) компонентов, которые обеспечивают создание соответствующих объектов и доступ к их методам и свойствам.

- Типичным примером невизуального компонента является таймер (тип TTimer) и компоненты доступа и управления базами данных. Существует еще множество других классов, однако их рассмотрение в задачу данной работы не входит.

# Задание

1. Повторить программу-пример «Тест-Приложение» (папка Labs, работа №1, параграф 4)
2. Повторить программу «Полиморфизм» (Файл «Полиморфизм»)
3. Повторить программу «Работа с базовым и дочерними классами» (Файл «Наследование», стр.18)
4. Разработать, создать и отладить программу по варианту «Многооконные приложения» (папка Labs, работа №1)
5. Разработать, создать и отладить программу по варианту «Работа с массивами» (Файл «Меню_массивы»). Создать значок приложения и Справку.