«Высокоуровневые методы информатики и программирования»

В результате изучения дисциплины студент должен

- знать:
  - теоретические основы разработки ПО;
  - основные положения объектно-ориентированного программирования;
  - состав, структуру и основные характеристики сред программирования, основанных на объектно-ориентированном методе.
- уметь:
  - ✔ решать типовые задачи программирования на основе объектно-ориентированного подхода и разрабатывать их ПО в интегральной среде программирования типа Delphi;
  - ✔ создавать сопровождающую документацию к программному продукту.

На изучение данной дисциплины отводится 108 часов, из них:

- лекции 18 часов;
- лабораторные работы 36 часов;
- самостоятельная работа 54 часа.

Форма отчетности – курсовая работа и экзамен.

Содержание дисциплины

- РАЗДЕЛ 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ МЕТОДОВ ИНФОРМАТИКИ И ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ
- Тема 1. Эволюция методов разработки программного обеспечения
- Тема 2. Базовые понятия и основные свойства объектно-ориентированного программирования
- Тема 3. Методы
- Тема 4. Полиморфизм
- Тема 5. Динамические объекты в ООП

- РАЗДЕЛ 2. ОСОБЕННОСТИ РАЗРАБОТКИ ПРОГРАММ В ИНТЕГРИРОВАННЫХ СРЕДАХ ПРОГРАММИРОВАНИЯ
- Тема 6. Состав и характеристики интегрированных систем программирования
- Тема 7. Управление проектом и подготовка среды визуального программирования к работе

Тема **1.** Эволюция методов разработки **программного обеспечения**

- От линейного к модульному программированию
- Структура процедурной программы
- Предпосылки объектно-ориентированного программирования
- Пример использования переменной типа «ЗАПИСЬ»

От линейного к модульному программированию

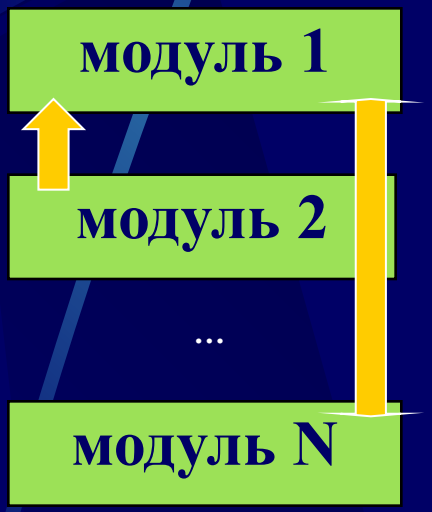
- *Линейное (алгоритмическое) программирование* - текст программы представляется в виде простой линейной последовательности операторов присваивания, цикла и условных операторов, с помощью которого можно решать не очень сложные задачи и составлять модульные программы, содержащие несколько сот строк кода.

- *Сруктурное программирование* - основано на понятии *подпрограммы* как набора операторов, выполняющих нужное действие и не зависящих от других частей исходного кода. Служит для создания средних по размеру приложений (несколько тысяч строк исходного кода).

Структура процедурной программы

Program < имя программы>;

Uses < имена модулей>;



Begin
<операторы>
...
End.

Unit < имя модуля >;

Uses <библиотеки>;
Label …;
Const …;
Type …;
Var …;
Function …;
Procedure …;
Implimentation
Uses < имена связанных модулей>;

Begin
<операторы>
…

End.

Function < имя>: < тип>;
…
Begin
<операторы>
…
end;

Procedure < имя>;
…
Begin
<операторы>
…
end;

Предпосылки объектно-ориентированного программирования

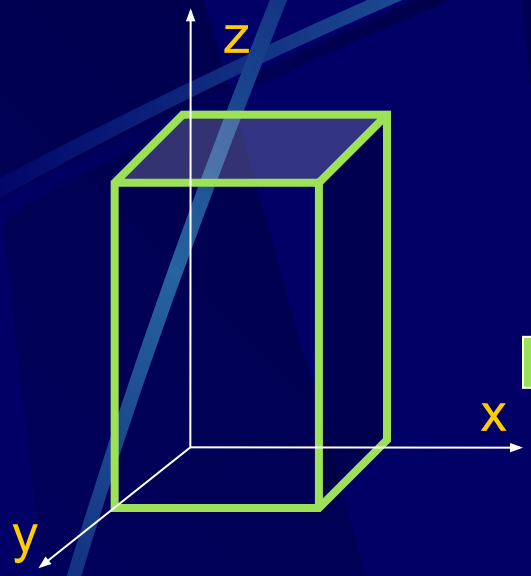
- Замечено, что реальные объекты окружающего мира обладают тремя базовыми характеристиками:
  - ✔ они имеют набор свойств,
  - ✔ способны разными методами изменять эти свойства и
  - ✔ реагировать на события, возникающие как в окружающем мире, так и внутри самого объекта.
- Именно поэтому, современные языки программирования высокого уровня позволяют применять сложные типы данных, составленные, как из базовых, так и определенных ранее, сложных типов.

Бейсик	Паскаль	Си

TYPE
<имя-струкгуры>
<имя поля> AS <имя типа>
…
END TYPE
	Type
	<имя струкгуры>= record
	<имя поля>: <имя типа>;
	…
	end;
		struct <имя струкгуры>
		{
		<имя типа> <имя поля>;
		…
		};

В результате объединения групп разных данных под одним общим названием удается организовывать структуры данных произвольной сложности, тип которых получил название «запись» и обеспечивает, однако, отражение только свойств реальных объектов.

Пример использования переменной типа «ЗАПИСЬ»



Объект – Cub

Свойства:

длина - **x**

ширина - **y**

высота - **z**

- Фрагмент программы

```
Program Example;
Type
        TCub = record
                x: real;
                y: integer;
                z: real;
end;
Var Cub: TCub;

begin
With Cub do begin
                x = 21,3;
                y = 13;
                z = 42,7;
end;
End.
```

Cub
x: real
y: integer
z: real

Cub
21,3
13
42,7

Тема **2.** Базовые понятия и основные свойства объектно-ориентированного программирования

- Идея объектно-ориентированного программирования
- Базовые понятия ООП
- Основные операции над переменной типа «объект»
- Пример использования переменной типа «объект»
- Структура объектно-ориентированной программы
- Основные свойства ООП

Идея объектно-ориентированного программирования

- Очевидно, что, в отличии от процедурного программирования, для реализации способностей объектов:
  - ✔ изменять свои свойства и
  - ✔ реагировать на определенные события

в их состав должны быть включены фрагменты программ, обеспечивающие выполнение соответствующих действий.

- Такие фрагменты, как известно, оформляются в виде процедур или функций.

Бейсик	Паскаль	Си
TYPE <имя струкгуры> <имя поля> AS <имя типа> … <имя функции> AS <имя типа> <имя процедуры>	Type <имя струкгуры>= record <имя поля>: <имя типа>; … <имя функции > :<имя типа> ; <имя процедуры> ;	struct <имя струкгуры> { <имя типа> <имя поля>; … <имя функции > [<имя типа>]; …
END TYPE	end;	};

Базовые понятия ООП

- Процедура или функция, объединенная с данным в рамках описания некоторого реального объекта, получила специальное название метод.
- Совокупность структур данных (свойств), характерных для такого объекта, а также - подпрограмм их изменения и обработки событий (методов), определяются с помощью зарезервированного слова object.
- Если несколько объектов имеют идентичную структуру и отличаются только значениями своих свойств, то в таких случаях в программе создается новый тип, данных, основанный на единой структуре объекта - «класс» (class), а каждый конкретный объект, имеющий структуру этого класса, называют его экземпляром.

Бейсик	Паскаль	Си
TYPE <имя струкгуры> <имя свойства> AS <имя типа> … <имя метода> [AS <имя типа>] …	Type <имя струкгуры>= object <имя свойства>: <имя типа>; … <имя метода> [:<имя типа>] ; …	object <имя струкгуры > { <имя типа> <имя свойства>; … <имя метода> [<имя типа>] …
END TYPE	end;	};

Основные операции над переменной типа «объект»

Описание объекта:

Type
<Тимя>= object

свойство 1

...

свойство N

метод 1

...

метод M

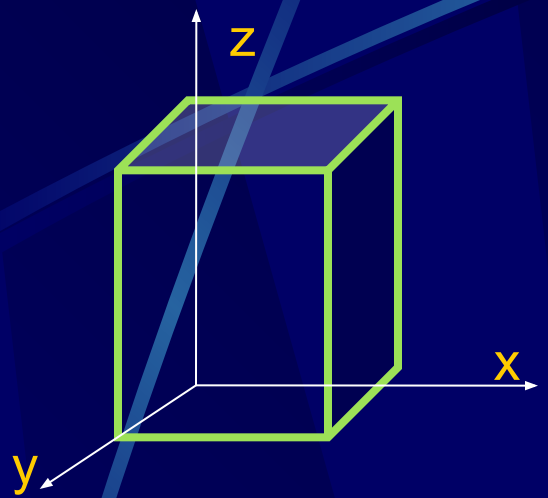
end;

Объявление переменной типа «object»:
var
<имя объекта>: <Тимя типа>
student: TStudent;

Обращение к свойствам и методам:
переменной типа «object»:
<имя объекта>.<имя свойства>;
Student[i].Name;
<имя объекта>.<имя метода>;
Student[i].PutName(s);

Примечание: объекты, как и любая переменная, могут передаваться в качестве параметров процедурам и функциям

Пример использования переменной типа «объект»



Объект – Cub

Свойства:

длинна - **x**

ширина - **y**

высота - **z**

Методы:

Основание - **xy**

объем - **xyz**

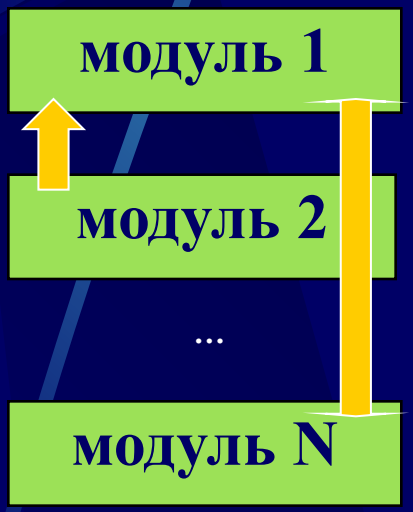
- Фрагмент ООП программы

```
Program Example;
Type
      TCub = class
            x: real;
            y: integer;
            z: real;
      Function xy: real;
      Function xyz: real;
end;
Var Cub: TCub;
Function TCub.xy: real; begin Result:= x*y end;
Function TCub.xyz: real; begin Result:= x*y*z
    end;
begin
With Cub do begin
            x = 21,3;
            y = 13;
            z = 42,7;
end;
Write (Cub.xy, Cub.xyz);
End.
```

Структура объектно-ориентированной программы

Program < имя программы>;

Uses < имена модулей>;



Begin
<операторы>
...
End.

unit < имя модуля >;

Uses <библиотеки>;
Label …;
Const …;
Type …;
Var …;
Function …;
Procedure …;
Implimentation
Uses < имена связанных модулей>;

Begin
<операторы>
…

End.

Function < имя>: < тип>;
…
Begin
<операторы>
…
end;

Procedure < имя>;
…
Begin
<операторы>
…
end;

Основные свойства объектно-ориентированного программирования

Таким образом, за счет реализации в современных языках программирования понятия объекта, как совокупности свойств (структур данных, характерных для этого объекта), методов их обработки (подпрограмм изменения свойств) и событий, на которые данный объект может реагировать и, которые приводят, как правило, к изменению свойств объекта, появляется новый способ программирования, отличающийся от известных следующими свойствами:

- возможностью комбинирования записей с процедурами и функциями, манипулирующими полями этих записей (инкапсуляция) и формированием более сложного типа данных – классы объектов.
- определением объекта и его дальнейшего использования для построения иерархии порожденных объектов с возможностью для каждого порожденного объекта, относящегося к этой иерархии, доступа к коду и данным всех порождающих объектов (*наследование*).
- возможностью присваивания действию одного имени, которое затем совместно используется вниз и вверх по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, именно ему подходящим (*полиморфизм*).

Тема **3.** Методы

Использование процедур и функций, объединенных с данным в рамках описания переменнных объектного типа, имеет целый ряд особенностей, которые обуславливают присвоение им специального общего названия – «методы» или «методы - члены класса» , делают их одними из наиболее примечательных атрибутов ООП и требуют к себе, в связи с этим, более пристального внимания.

- Порядок определения методов
- Область действия метода
- Экспорт объектов
- Секция «private»
- Задача инициализации

Определение методов

- Для определения отдельного объекта или целого класса объектов в модуле не требуется никаких специальных соглашений.
- Определять объекты в модуле принято аналогично описанию типа переменных в его интерфейсной части, а тела методов объекта - в секции реализации.
- Внутри объекта метод определяется заголовком процедуры или функции, действующей как метод. При этом поля данных (свойства) объекта должны быть описаны перед первым описанием метода.
- В связи с тем, что методы и их объекты разделяют общую область действия, формальные параметры метода не могут быть идентичными любому из полей данных объекта.
- В качестве параметров метода, могут выступать, в том числе и объекты.
- Если метод полностью определяется вне объекта, то имени метода должно предшествовать имя типа объекта, которому принадлежит этот метод, с последующей точкой

Порядок определения методов напоминает структуру модуля программы:

Область действия метода

- Хотя в исходном коде поля данных объекта и тела методов разделены, на самом деле они совместно используют одну и ту же область действия (памяти).
- Если объект вызывает метод, то выполняется неявный оператор with myself do method, связывающий объект и его методы в области действия.
- Неявный оператор with выполняется путем передачи невидимого параметра методу всякий раз, когда этот метод вызывается. Этот параметр называется Self и в действительности является 32-разрядным указателем на экземпляр объекта, осуществляющего вызов метода.
- Именно поэтому методы объектов могут обращаться к своим свойствам без каких-либо квалификаторов перед ними.
- И именно поэтому свойства, встречающиеся в описании метода, принадлежат тому объекту, который вызывает данный метод.
- И именно поэтому методы называют также – «членами класса», в котором они объявлены.

Примечание: явное использование параметра Self допускается, но желательно избегать ситуаций, в которых это требуется.

Экспорт объектов

- Объекты, определенные в интерфейсной части модуля, являются экспортируемыми (видны – или могут быть использованы через оператор *uses* в других модулях).
- Как и в случаях с записями, объекты могут передаваться в качестве параметра процедуре, а также (как вы увидите позднее) - размещаться в динамически распределяемой памяти.
- Модули могут иметь и свои собственные приватные (частные) определения типов объектов внутри выполняемой секции, и эти типы подвержены тем же ограничениям, как и всякие другие типы, определенные в секции реализации, т.е. будут неэкспортируемые.
- При этом, типы объектов, определенные в интерфейсной части модуля, могут иметь дочерние типы объектов, определенные в секции реализации модуля.

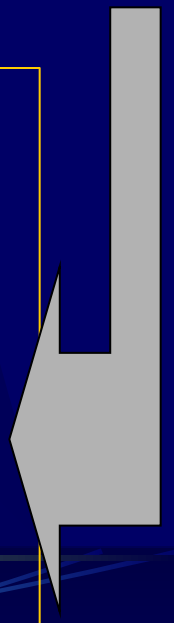
```
unit Example;
interface
Type
      TCub = class
                …
end;
Var Cub_1: TCub;

implimentation
Begin
Type
      TKub = object;
                …
      end;
Var Cub_2: TKub;
      …
End.
```

Секция «private»

- В тех случаях когда нежелательно экспортировать отдельные части описаний объектов, ООП позволяет задавать внутри объектов приватные (закрытые) поля и методы.
- Приватные поля и методы доступны только внутри того модуля, в котором описан объект.
- Приватные поля и методы в Паскале описываются непосредственно после обычных полей и методов, вслед за зарезервированным словом private:

В общем же (например, в C++) метод как член класса может иметь следующие модификаторы доступа:
public - общедоступный метод;
protected - метод, доступный только для членов и объектов данного класса и наследуемых классов (наследуемых с модификаторами доступа public или protected);
private - защищенный метод, доступный только внутри класса

```
NewObject = object(родитель)
поля; {общедоступные}
методы; {общедоступные}
private
поля; {приватные}
методы; {приватные}
end;
```



Задача инициализации

Из определения переменной типа «объект» следует, что поля (свойства) данных объекта - это то, что объект "знает", а методы объекта - это то, что объект "делает".

В этом случае задача инициализации переменной данного типа , как реализация способностей объектов «изменять свои свойства» может быть решена за счет организации доступа к полям его данных с помощью своих же методов (инкапсуляция).

```
Program Example;
Type
        TCub = class
                x: real;
                y: integer;
                z: real;
procedure Init (New_X: Real,
      New_Y: integer, New_Z: Real);
end;
Var Cub_1: TCub;
procedure TCub.Init (New_X: Real,
      New_Y: integer, New_Z: Real);
begin
                x = New_X ;
                z = New_Y;
                y = New_Z;
end;

begin
Cub_1.Init (21,3, 13, 42,7);  // !!!
Write (Cub_1.xy, Cub_1.xyz);
End.
```

Конструктор

- Метод, который решает задачу инициализации (установки начальных значений свойствам объекта) или конструирует значения данного типа, называется в ООП конструктором.
- Конструкторы могут иметь параметры, что позволяет определить начальное состояние объекта при его порождении.
- Конструкторы в C++, обычно, имеют то же имя, что и имя класса, в котором они определены, так что если класс имеет несколько конструкторов, то они должны различаться числом и типом своих параметров.
- Например: class date {date(int, int, int);};
- Для конструкторов объекта в ОП чаще используется идентификатор Init.
- Когда класс имеет конструктор, все объекты этого класса будут инициализироваться. Если для конструктора нужны параметры, они должны даваться:
  - date today = date(23,6,1983);
    date xmas(25,12,0); // сокращенная форма (xmas - рождество)
    date my_burthday; // недопустимо, опущена инициализация
- Часто необходимо обеспечить несколько способов инициализации объекта класса. Это можно сделать, задав несколько конструкторов.
- Конструктор без параметров (по умолчанию). Один из способов сократить число родственных функций - использовать параметры по умолчанию.

Тема **4.** Полиморфизм

Процедурные языки достатачно «жестко» определяют алгоритмы процедур и функций.
Объектно-ориентированный подход с помощью механизма наследования может обеспечить им существенно большую гибкость :
если определен порожденный тип, то методы порождающего типа, как члены класса наследуются, однако при желании они могут переопределяться

- Наследование статических методов
- Виртуальные методы и полиморфизм
- Полиморфические объекты
- Совместимость типов объектов
- Раннее связывание против позднего связывания
- Абстрактный класс

Наследование статических методов

- Для простого переопределения наследуемого метода достаточно описать новый метод с тем же именем, что и наследуемый метод, но с другим телом и (при необходимости) с другим множеством параметров
- При таком варианте переопределения транслятор разрешает ссылки на методы во время компиляции (раннее связывание) и алгоритм его работы будет таков:
  - при вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта;
  - если метод с таким именем найден, то адрес родительского метода замещает имя в исходном коде дочернего метода;
  - если метод с таким именем не найден, то компилятор продолжает его поиск, продвигаясь вверх по родительским объектам;
  - если компилятор наталкивается на самый первый (высший, абстрактный) тип объекта, то он выдает сообщение об ошибке, указывающее, что ни одного такого метода не определено.
- Однако, если такой «статический» наследуемый метод найден и используется, то программист должен помнить, что вызываемый метод является в точности таким, как он определен и компилирован для родительского типа. Если родительский метод вызывает другие методы, то вызываемые методы будут также родительскими методами, даже если дочерний объект содержит методы, которые переопределяют родительские.

Примечание: поля данных переопределяться не могут. После того, как вы определили поле данных в иерархии объекта, никакой дочерний тип не может определить поле данных в точности с таким же именем.

Виртуальные методы и полиморфизм

- Использование статических методов не лучший способ для управления этими членами классов при составлении сложных программ. Дело в том,что в их работе могут возникнуть проблемы, из-за разрешения ссылок на методы во время компиляции, так как определение вызываемого метода происходит именно на этапе «раннего связывания».
- Очевидно, что для исключения ошибок, связанных с «разночтением» транслятором переопределенных дочерними объектами родительских методов, их определение должно строго зависеть от типа объекта, для которых они вызывается. И скорее всего, это можно сделать после инициализации самих объектов уже в ходе выполнения программы, т.е. – динамически.
- Такой механизм в программировании, как известно, имеет название «позднего связывания» и, применительно к процессу переопределения методов, предполагает транслятору разрешать ссылки на них не на этапе компиляции, а именно во время выполнения программы (например, dll).
- В отличие от статических, методы, определяемые на основе этого механизма получили название виртуальных методов.
- Виртуальные методы объявляются в базовом классе с ключевым словом virtual, а в производном классе могут быть переопределены. Прототипы виртуальных методов как в базовом, так и в производном классе должны быть одинаковы.
- На этапе компиляции строится таблица виртуальных методов, а их конкретные адреса проставляется уже на этапе выполнения.
- При вызове метода с использованием указателя на класс действуют следующие правила:
  - для виртуального метода вызывается метод, соответствующий типу объекта, на который указывает указатель.
  - для не виртуального метода вызывается метод, соответствующий типу самого указателя.
- Виртуальные методы предоставляют мощный инструмент, называемый полиморфизмом. Полиморфизм является способом присвоения действию имени, которое используется всеми объектами иерархии, причем каждый объект иерархии использует это действие определенным образом.

Полиморфические объекты

- Как уже отмечалось при определении, объекты могут передаваться в качестве параметра процедуре или функции (т.е. - методу).
- Применительно к возможности наследования в ООП это означает, что и «любой порожденный от типа параметра тип может передаваться в качестве параметра».
- Однако реализация механизма «позднего связывания» при использовании виртуальных методов предполагает отсутствие сведений о точном типе фактического параметра во время компиляции.
- Следовательно, фактические параметры виртуальных методов могут быть объектами любого дочернего от параметра-переменной типа, и именно поэтому, они называется полиморфическими объектами.
- Прежде всего полиморфические объекты позволяют обрабатывать объекты, чей тип неизвестен на момент компиляции.

Совместимость типов объектов

- Таким образом, наследование до некоторой степени изменяет правила совместимости типов в ООП. Помимо всего прочего, порожденный тип наследует совместимость типов всех своих порождающих типов. Эта расширенная совместимость типов принимает три формы:
  - - между реализациями объектов;
  - - между указателями на реализации объектов;
  - - между формальными и фактическими параметрами.
- Однако очень важно помнить, что во всех трех формах совместимость типов расширяется только от потомка к родителю. Другими словами, дочерние типы могут свободно использоваться вместо родительских, но не наоборот.
- Для совместимости объектов при реализации данного механизма ООП существует три правила:
  - Объекту родительского класса можно присвоить любой объект порожденного класса (обратное - недопустимо).
  - Указатели на родительский объект можно присвоить на любой порожденный объект.
  - Если фактический параметр является некоторым объектом или указателем на объект, то формальный параметр может быть либо родительским объектом либо указателем на родительский объект.

Раннее связывание против позднего

- И так, метод становится виртуальным, если за его объявлением в типе объекта стоит зарезервированное слово virtual.
- Если метод объявлен в родительском типе как virtual, то все методы с аналогичными именами в дочерних типах должны быть объявлены виртуальными.
- Каждый тип объекта, содержащий виртуальные методы, имеет таблицу виртуальных методов (ТВМ), хранящуюся в сегменте данных. ТВМ содержит размер типа объекта и для каждого виртуального метода указатель кода, исполняющий данный метод. Имеется только одна ТВМ для каждого типа объекта.
- Каждый вызов виртуального метода должен проходить через ТВМ, тогда как статические методы вызываются непосредственно. Причем, если объект наследует виртуальные методы, то он также наследует ТВМ. Инициализация ТВМ осуществляется конструктором.
- ТВМ создается компилятором автоматически и программа никогда не манипулирует ими непосредственно. Указатели на ТВМ также запоминаются автоматически.
  Первое слово ТВМ содержит размер экземпляров соответствующего объектного типа. Эта информация нужна для конструктора или деструктора, для определения количества байт.
- Имеются два убедительных довода в пользу того, чтобы везде, где это только возможно, использовать статические методы:
  - Во-первых, интеллектуальный компоновщик Turbo Pascal не может отменять неиспользуемые виртуальные методы, а только неиспользуемые статические методы; простой акт ввода объекта данного типа приводит к тому, что в программу должны быть скомпонованы все виртуальные методы этого объекта.
  - И во-вторых, чем больше виртуальных методов имеет объект, тем обширнее его таблица виртуальных методов VMT: объект, имеющий 100 виртуальных методов, использовал бы более 400 байт пространства в сегменте данных.
- Полагая, что из практических соображений невозможно во всех случаях применять только одни виртуальные методы, очевидно целесообразно создавать виртуальные методы только в следующих трех случаях:
  - а) если мы знаем, что он должен быть отменен объектом-потомком,
  - б) если метод предназначен для того, чтобы быть отмененным, и для этой цели и существует,
  - в) если в самой природе метода заложена возможность того, что желательно его отменить в объекте-потомке.

Раннее связывание против позднего

- В общем случае, следует делать методы виртуальными.
- Используйте статические методы только в том случае, если вы хотите получить большую скорость выполнения программы и эффективность использования памяти. Однако в этом случае, теряется часть возможности расширения.
- При определении, каким должен быть метод, виртуальным или статическим можно использовать правило «большого пальца»:
  - сделайте метод виртуальным, если имеется вероятность, что будущие наследники объекта будут переопределять данный метод, а вы хотите, чтобы будущий код был доступен и самому объекту.
- С другой стороны, помните, что если у объекта имеются любые виртуальные методы, то для этого объекта в сегменте данных будет создана таблица виртуальных методов (ТВМ) и каждый экземпляр этого объекта будет иметь связь с ТВМ. Каждый вызов виртуального метода должен проходить через ТВМ, тогда как статические методы вызываются непосредственно. Хотя просмотр ТВМ является весьма эффективным, вызов статического метода все равно остается более быстрым, чем вызов виртуального. И если в вашем объекте нет виртуальных методов, то и ТВМ отсутствует в сегменте данных и (что более важно) в каждом экземпляре объекта отсутствуют связи с ТВМ.
- Дополнительная скорость и эффективное использование памяти для статических методов должно уравновешиваться гибкостью, которую допускают виртуальные методы: вы можете расширить имеющийся код спустя много времени после его компиляции.
- Помните, что пользователь вашего типа объекта может рассматривать пути его использования, которые вам и не снились, что, в конечном счете, имеет основное значение.

Абстрактный класс

- Полиморфизм - это придание одного и того же имени действию, которое выполняется над различными объектами классов, находящихся в иерархическом подчинении. Для каждого из объектов это действие может выполняться по своему.
- Для того чтобы при множественном наследовании один и тот же базовый класс не порождал для объекта производного класса несколько объектов базового класса, при объявлении производного класса такой базовый класс указывается с ключевым словом virtual и называется виртуальным классом.
  - Например:
  - class A : virtual public B { } **Абстрактные классы**
- Абстрактным классом называется класс, который содержит хотя бы одну чисто виртуальную? функцию.
- Абстрактный класс не может быть явно использован для создания объектов.
- Как правило, абстрактный класс применяется для описания интерфейса, который должен быть реализован всеми его производными классами.
- Если класс, производный от абстрактного класса, не содержит реализации всех его чисто виртуальных функций, то он также является абстрактным классом.

Полиморфизм и конструкторы

- Механизм «позднего связывания» определяет и специфику решения задачи инициализации объектов, имеющих виртуальные методы.
- В этом случае кроме установки начальных значений свойствам объекта конструктору необходимо устанавить и связь между вызывающим его экземпляром объекта и ТВМ этого объекта.
- Следовательно, более точно: конструктор является специальным типом процедуры, которая выполняет некоторую установочную работу для механизма виртуальных методов.
- Поэтому каждый тип объекта, имеющий виртуальные методы, обязан иметь конструктор. Более того, конструктор должен вызываться перед вызовом любого виртуального метода. Вызов виртуального метода без предварительного вызова конструктора может привести к блокированию системы, а у компилятора нет способа проверить порядок вызова методов.
- Каждый отдельный экземпляр объекта должен инициализироваться отдельным вызовом конструктора. Недостаточно инициализировать один экземпляр объекта и затем присваивать этот экземпляр другим. Другие экземпляры, даже если они могут содержать правильные данные, не будут инициализированы оператором присваивания и заблокируют систему при любых вызовах их виртуальных методов.

Основные свойства и правила использования конструкторов:

- конструктор не возвращает значения даже типа void;
- конструктор не наследуется в производных классах. Если необходимо, то конструктор производного класса может вызвать конструкторы для его базовых классов;
- конструктор может иметь аргументы, заданные по умолчанию;
- конструктор - это метод, но он не может быть виртуальным, его нельзя объявить виртуальным;
- невозможно получить в программе адрес конструктора (указатель на конструктор);
- Если конструктор не задан в программе, то он будет автоматически сгенерирован компилятором для построения соответствующих объектов.
- Все конструкторы сгенерированные компилятором, имеют атрибут public;
- Конструктор по умолчанию для класса X - это конструктор, который может быть вызван без аргументов;
- Конструктор вызывается автоматически только при описании объекта;
- Объект, содержащий конструктор, нельзя включить в виде компонента в объединение (union);
- Конструктор класса X не может иметь аргумент типа X;
- Конструктор, заданный в виде X::X(const X &), называется конструктором для копирования (copy constructor) класса X;

Тема 5. Динамические объекты в ООП

Как известно, одно из правил программирования утверждает, что «памяти, как и денег, всегда не хватает».
Поэтому при любом способе программирования по ходу работы программ стараются инициализировать только те переменные, которые непосредственно необходимы для ее выполнения.
После того, как необходимость в той или иной переменной отпадает ее также принудительно удаляют из памяти. Такие переменные называют **динамическими.**

- Размещение и инициализация объектов с помощью процедуры New
- Удаление динамических объектов. Деструкторы
- Определение деструктора
- Основные свойства и правила использования деструкторов

Размещение и инициализация объектов с помощью процедуры **New**

- Borland Pascal расширяет синтаксис стандартной процедуры New для выделения пространства в динамически распределяемой области памяти и переменным объектного типа, что является более компактным и более удобным средством инициализации объекта с помощью одной операции.
- Теперь процедура New может вызываться с двумя параметрами: имя указателя используется в качестве первого параметра, а вызов конструктора - в качестве второго параметра:

```
New(P, Init('Sara Adams', 'Account manager', 2400));
```

- Если для процедуры New используется расширенный синтаксис, то конструктор Init действительно выполняет динамическое размещение, используя специальный код входа, сгенерированного как часть компиляции конструктора.
- Имя реализации не может предшествовать Init, т.к. в то время, когда процедура New вызвана, реализация, инициализируемая с помощью Init, еще не существует. Компилятор идентифицирует правильный вызываемый метод Init посредством типа указателя, пересылаемого в качестве первого параметра.
- Процедура New расширена также и для возможности использования ее как функции, которая возвращает значение указателя. Посылаемый New параметр является типом указателя на объект, а не самой переменной-указателем:

```
var
P: PSalaried;
P := New(PSalaried);
```

- Обратите внимание, что в данной версии функциональная форма расширения процедуры New применима ко всем типам данных, а не только к типам объектов.
- Функциональная форма New, как и процедурная форма, также может воспринимать конструктор объектного типа в качестве второго параметра:

```
P := New(PSalaried, Init('Sara Adams', 'Account manager', 2400));
```

Примечание: порождающему объекту можно присвоить экземпляр любого из его порожденных типов. Обратные присваивания недопустимы.

Удаление динамических объектов. Деструктор

- Когда объект уничтожается при завершении программы или при выходе из области действия определения соответствующего класса, необходимы противоположные операции, самая важная из которых - освобождение памяти. Эти операции могут и должны выполняться по-разному в зависимости от особенностей конкретного класса. Поэтому в определении класса явно или по умолчанию включают специальную принадлежащую классу функцию - деструктор.
- Деструктор имеет строго фиксированное имя вида: имя_класса
- У деструктора не может быть параметров (даже типа void), и деструктор не имеет возможности возвращать какой-либо результат, даже типа void.
- Статус доступа деструктора по умолчанию public (т.е. деструктор доступен во всей области действия определения класса). В несложных классах деструктор обычно определяется по умолчанию.
- Деструкторы не наследуются, поэтому даже при отсутствии в производном классе деструктора он не передается из базового, а формируется компилятором как умалчиваемый со статусом доступа public. Этот деструктор вызывает деструкторы базовых классов.
- Деструкторы базовых классов выполняются в порядке, обратном перечислению классов в определении производного класса. Таким образом порядок уничтожения объекта противоположен по отношению к порядку его конструирования.
- Вызовы деструкторов для объектов класса и для базовых классов выполняются неявно. Однако вызов деструктора того класса, объект которого уничтожается в соответствии с логикой выполнения программы, может быть явным. Это может быть, например, случай, когда при создании объекта для него явно выделялась память.

Определение деструктора

- Также, как и обычные записи Паскаля, размещаемые в динамически распределяемой области памяти, объекты могут удаляться процедурой Dispose, если они больше не нужны: Dispose (P);
- Однако, в отличии от простай переменной при избавлении от ненужного объекта может понадобиться нечто большее, чем простое освобождение занимаемой им динамической памяти. Дело в том, что объект может содержать указатели на динамические структуры или объекты, которые нужно освободить или очистить в определенном порядке, особенно если вы оперируете сложной динамической структурой данных.
- Что бы ни нужно было сделать для очистки динамического объекта в каком-либо порядке, это все должно быть объединено в один метод таким образом, чтобы объект мог быть уничтожен с помощью одного вызова метода. Мы советуем использовать для удаления методов, работающих с объектами, которые более не нужны, использовать идентификатор Done

MyComplexObject.Done;

- Метод Done должен инкапсулировать все детали очистки своего объекта, а также всех структур данных и вложенных объектов.
- В зависимости от того, как они размещены или используются, или в зависимости от состояния и режима объекта на момент очистки, сложные объекты могут потребовать очистки несколькими различными путями. Поэтому допустимо и часто бывает полезно определять несколько методов очистки для данного типа объекта.
- Borland Pascal предоставляет именно такой специальный тип метода, называемый "сборщиком мусора" или деструктором, для очистки и удаления динамически размещенного объекта. Деструктор объединяет шаг удаления объекта с какими-либо другими действиями или задачами, необходимыми для данного типа объекта. Для единственного типа объекта можно определить несколько деструкторов.
- Деструктор объявляется совместно со всеми другими методами объекта в определении типа объекта, например:

```
type
TEmployee = object
...
constructor Init(AName, ATitle: String; ARate: Real);
destructor Done; virtual;
...
end;
```

- Деструкторы можно наследовать, и они могут быть либо статическими, либо виртуальными. Поскольку различные программы завершения обычно требуют различные типы объектов, рекомендуется, чтобы деструкторы всегда были виртуальными, тогда для каждого типа объекта будет выполнен правильный деструктор.

Основные свойства и правила использования деструкторов:

- деструктор имеет то же имя, что и класс, в котором он объявляется, с префиксом ~(тильдой);
- деструктор не возвращает значения даже типа void;
- деструктор не наследуется в производных классах;
- производный класс может вызвать деструкторы для его базовых классов;
- деструктор не имеет параметров (аргументов);
- класс может иметь только один деструктор;
- деструктор - это функция, и он может быть виртуальным, его можно объявить виртуальным;
- невозможно получить в программе адрес деструктора (указатель на деструктор);
- если деструктор не задан в программе, то он будет автоматически сгенерирован компилятором для уничтожения соответствующих объектов.
- все деструкторы, сгенерированные компилятором, имеют атрибут public;
- деструктор вызывается автоматически при разрушении объекта;
- когда вызывается библиотечная функция exit, вызываются деструкторы только для глобальных объектов;
- когда вызывается библиотечная функция abort, никакие деструкторы не вызываются.

Тема 6. Интегрированные системы программирования

Возможности расширения и повторного использования существующего кода, которые заложены в идее объектно-ориентированного программирования, не только вносит рациональный порядок в структуру программного обеспечения ЭВМ, но и позволяют строить системы программирования нового типа.

Это, так называемые - интегральные среды визуального программирования (ИСВП), к которым, например, относятся продукты производства корпорации Borland: C++Builder, Borland C++, Delphi, IntraBuilder и Jbuilder.

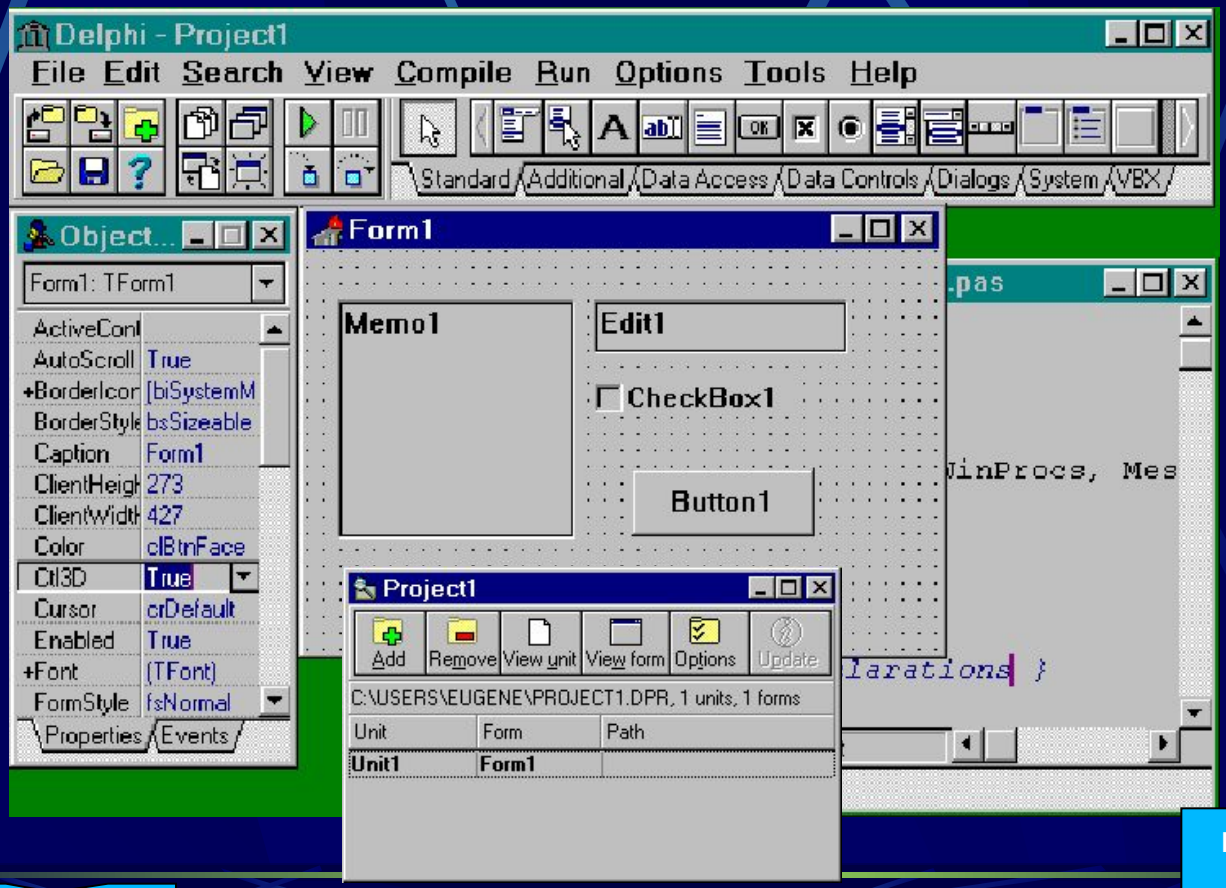
В состав таких систем, как правило, входят:

- среда визуального построителя приложений;
- объектно-ориентированная библиотека компонент;
- высокопроизводительный компилятор в машинный код;
- генератор отчетов;
- масштабируемые средства для построения баз данных.

Среда визуального построителя приложений

главное меню и частично дублирующий его набор быстрых кнопок

линейка визуальных и не визуальных компонент



Визуальный построитель интерфейсов

инспектор объектов

менеджер проектов

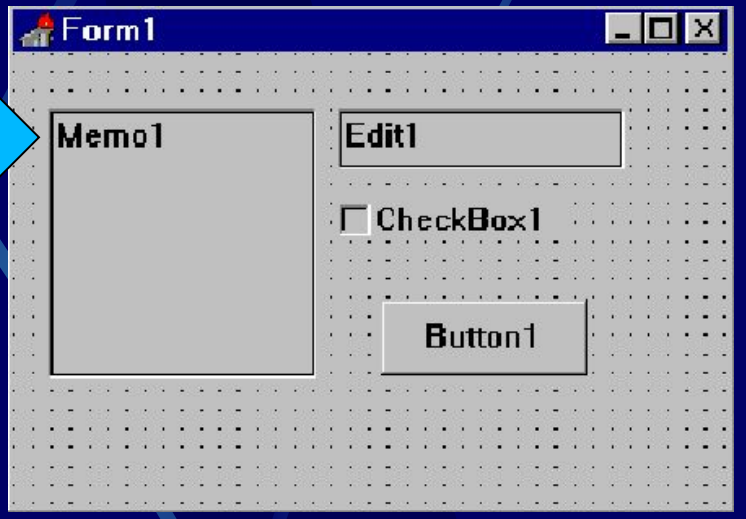
многостраничное окно интеллектуального редактора

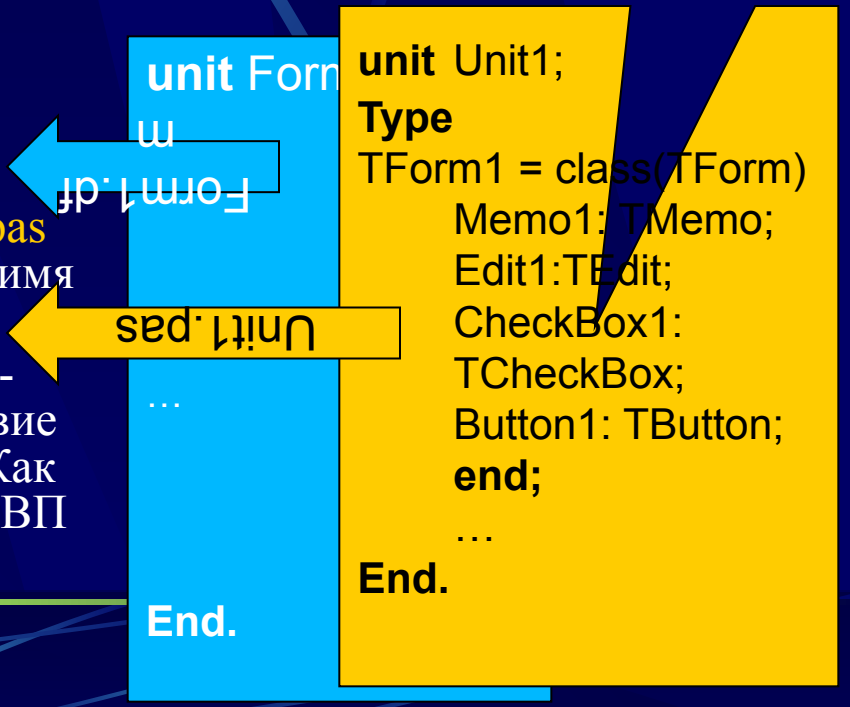
Библиотека объектных Визуальных Компонент

- Компоненты, используемые при разработке приложений в ИСВП (и также собственно самих ИСВП), встроены в среду разработки приложений и представляют собой набор типов объектов, используемых в качестве фундамента при строительстве приложения. Классы объектов построены в виде иерархии, состоящей из абстрактных, промежуточных и готовых компонент.
- Этот костяк называется Visual Component Library (VCL). В VCL есть такие стандартные элементы управления, как строки редактирования, статические элементы управления, строки редактирования со списками, списки объектов и т.п.
- Еще имеются такие компоненты, которые ранее были доступны только в библиотеках третьих фирм: табличные элементы управления, закладки, многостраничные записные книжки и т.д.
- VCL содержит специальный объект, предоставляющий интерфейс графических устройств Windows, и позволяющий разработчикам рисовать, не заботясь об обычных для программирования в среде Windows деталях.
- Ключевой особенностью ИСВП является возможность не только использовать визуальные компоненты для строительства приложений, но и создавать новые компоненты. Такая возможность позволяет разработчикам не переходить в другую среду разработки, а наоборот, встраивать новые инструменты в существующую среду. Кроме того, можно улучшить или полностью заменить существующие по умолчанию в ИСВП компоненты.
- Последовательность введения новой компоненты состоит из трех шагов:
  - наследование из уже существующего типа компоненты;
  - определение новых полей, свойств и методов;
  - регистрация компоненты.

Метод разработки «Two-Way Tools»

- Формы - это объекты, в которые вы помещаете другие объекты для создания пользовательского интерфейса вашего приложения.
- Модули же состоят из кода, который реализует функционирование вашего приложения, а также - обработчиков событий (описаний методов) для форм и их компонент.
- Информация о формах хранится в двух типах файлов - *.dfm и *.pas (или *.cpp), причем первый тип файла - двоичный - хранит образ формы и ее свойства, второй тип описывает функционирование обработчиков событий и поведение компонент.
- Оба файла автоматически синхронизируются ИСВП, так что если добавить новую форму в проект приложения, связанный с ним файл *.pas (или *.cpp) будет создан автоматически, и его имя будет добавлено в проект.
- Такая синхронизация и делает ИСВП two-way-инструментом, обеспечивая полное соответствие между кодом и визуальным представлением. Как только вы добавите новый объект или код, ИСВП устанавливает т.н. «кодовую синхронизацию» между визуальными элементами и соответствующими им кодовыми представлениями.





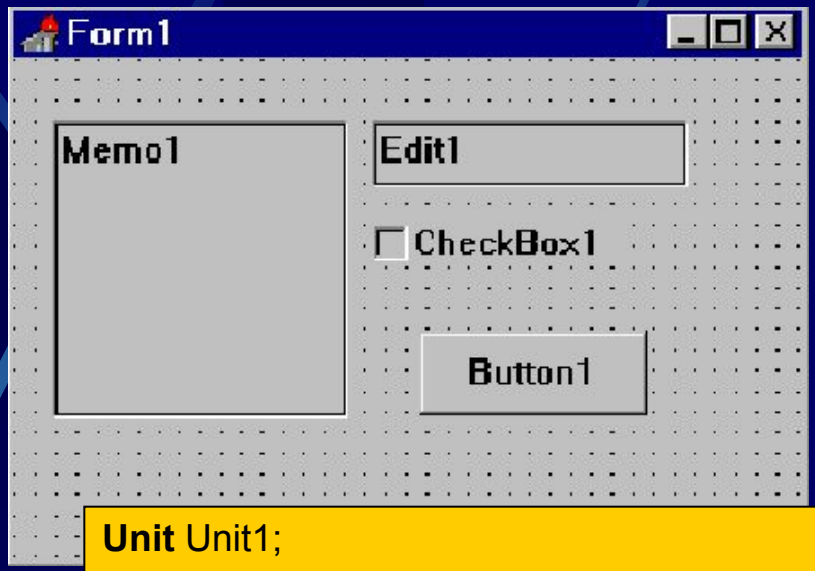
Делегирование и ссылки на классы

- Под «делегированием» понимается совокупность операций (механизм) предоставления неким объектом другому объекту «права» отвечать на некоторые события.
- Так, например, если предполагается показывать окно сообщения по нажатию кнопки, то следует дважды щелкнуть мышкой непосредственно на оъект Button в форме или дважды щелкнуть мышью на строчку OnClick на странице Events в Инспекторе объектов.
- В этом случае ИСВП автоматически:

- сформирует декларацию объекта TForm1, которая содержит процедуру ButtonClick, представляющую собой собственно обработчик события, а также
- создаст процедуру или заголовок метода, куда вы можете добавить необходимый код.

- Используемые при этом «ссылки на классы» придают дополнительный уровень гибкости, так как предоставляется возможность динамически создавать объекты, чьи типы могут быть известны только во время выполнения кода.

- Собственно, эта технология используется и при построении самих ИСВП.




```
Unit Unit1;
...
TForm1 = class(TForm)
...
Button1: TButton;
procedure Button1Click(Sender: TObject);
end;
...
procedure TForm1.Button1Click(Sender:
TObject);
Begin
...
end;
…
End.
```

Обработка исключительных ситуаций

- Серьезные приложения должны надежным образом обрабатывать исключительные ситуации, сохранять, если возможно, выполнение программы или, если это невозможно, аккуратно ее завершать.
- Написание кода, обрабатывающего исключительные ситуации, всегда было непростой задачей, и являлось источником дополнительных ошибок.
- В ИСВП это устроено в стиле C++. Исключения представлены в виде объектов, содержащих специфическую информацию о соответствующей ошибке (тип и место - нахождение ошибки).
- Разработчик может оставить обработку ошибки, существовавшую по умолчанию, или написать свой собственный обработчик.
- Обработка исключений реализована в виде exception-handling blocks (также еще называется protected blocks), которые устанавливаются ключевыми словами try и end.
- Существуют два типа таких блоков: try...except и try...finally.
- Конструкция try....finally предназначена для того, чтобы разработчик мог быть полностью уверен в том, что перед обработкой исключительной ситуации всегда будет выполнен некоторый код (например, освобождение ресурсов).

```
Unit Unit1;
...
try
{ выполняемые операторы }
except
on exception1 do statement1;
{ реакция на ситуации }
on exception2 do statement2;
else
{ операторы по умолчанию }
end;
…
try
{ выполняемые операторы }
finally
{ операторы, выполняемые
безусловно }
end;
...
End.
```

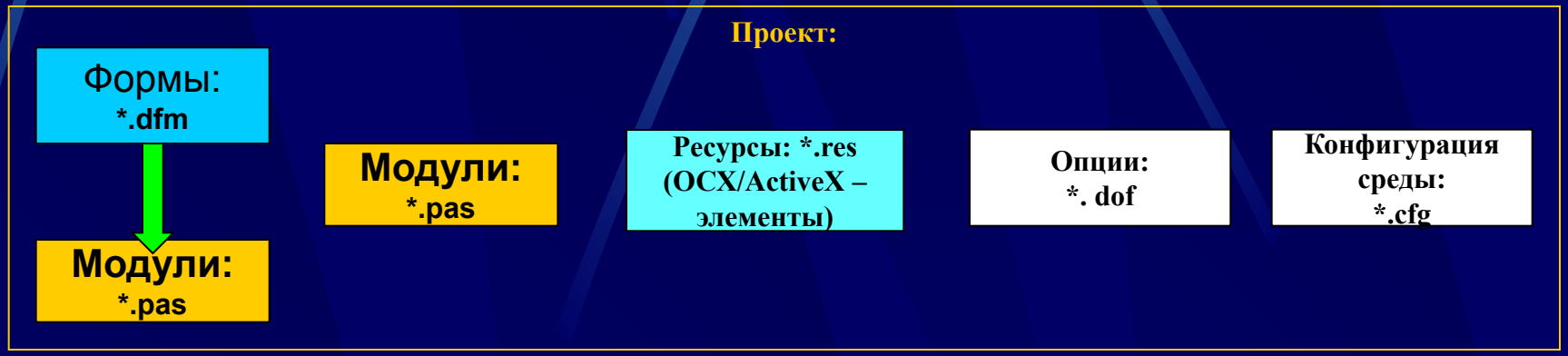
Тема 7. Управление проектом и подготовка среды визуального программирования к работе

Из теории программирования нам уже известно, что любое приложение на этапе разработки представляется, как правило, некоторой совокупностью модулей, оформляемых в виде отдельных файлов, которые связанны с этим приложением и определяют его. Такой набор файлов называется, обычно, *проектом* данного приложения.

- Состав проекта
- Назначение файлов проекта
- Формированиеь приложения
- Установка опций проекта

Состав проекта

- Так как в ИСВП для создания пользовательского интерфейса приложения используются визуальные объекты - формы, в которые помещаются все другие объекты интерфейса, то синхронизированные с ними модули состоят из кода, который реализует функционирование приложения, а также - обработку событий для форм и их компонент.
- Следовательно, каждому файлу формы обязательно соответствует файл с исходным текстом модуля, но файл с исходным текстом модуля не обязательно должен иметь соответствующую ему форму.
- Помимо этих основных типов файлов в проект могут входить файлы, созданные в других программах. Такие файлы включаются в приложение с помощью директив компилятора.



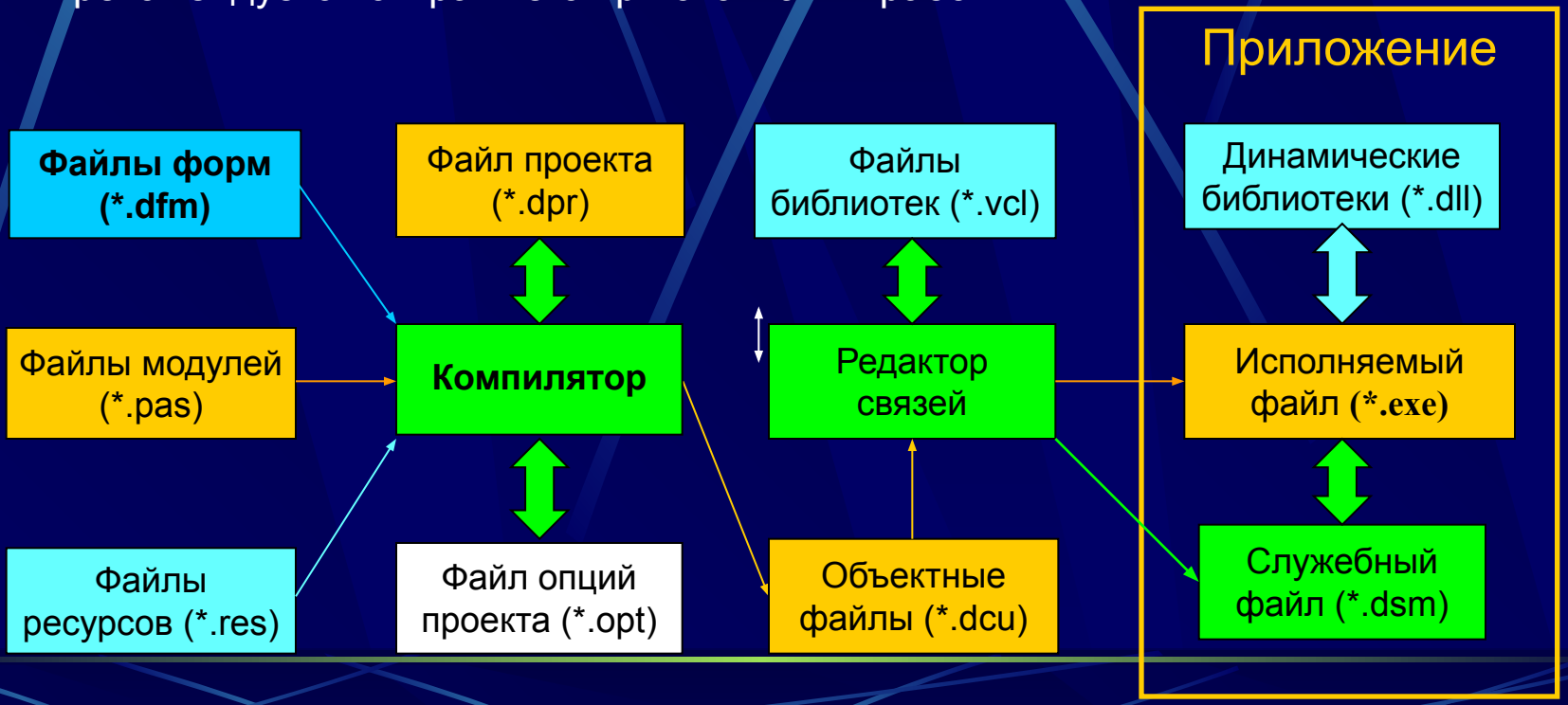
- Кроме этого, в состав проекта приложения, могут быть включены файлы, которые относятся к управлению проектом из среды, и напрямую программистом не меняются (см. табл.):

Назначение файлов проекта

Интегрированная среда разработки		Назначение файла
Delphi	C++Builder	
PROJECT1.DPR	PROJECT1.BPR	Главный файл проекта – текстовый файл, используемый для хранения информации о формах и модулях. В нем содержатся операторы инициализации и запуска программ на выполнение
UNIT1.PAS	UNIT1.CPP	Файл модуля - текстовый файл для хранения кода. Каждой создаваемой форме и каждому фрейму соответствует текстовый файл модуля. Модули создаются и не связанные с формами. Первая форма и ее модуль /unit/ автоматически появляются в начале работы.
UNIT1.DFM	UNIT1.DFM	Файл формы – двоичный файл, создаваемый для хранения информации о формах и фреймах приложения. Каждому файлу формы соответствует файл модуля.
PROJECT1.RES	PROJECT1.RES	Файл, содержащий иконку для проекта и прочие ресурсы.
PROJECT1.OPT	PROJECT1.OPT	Файл опций проекта - текстовым файлом для сохранения установок, связанных с данным проектом.
PROJECT1.CFG	PROJECT1.CFG	Файл конфигурации проекта, где хранятся данные о состоянии рабочего пространства среды (установки проекта), в частности, используемые директивы компилятора.
PROJECT1.BPG	PROJECT1.BPG	Файл группы файлов, формируемый при создании программистом группы файлов

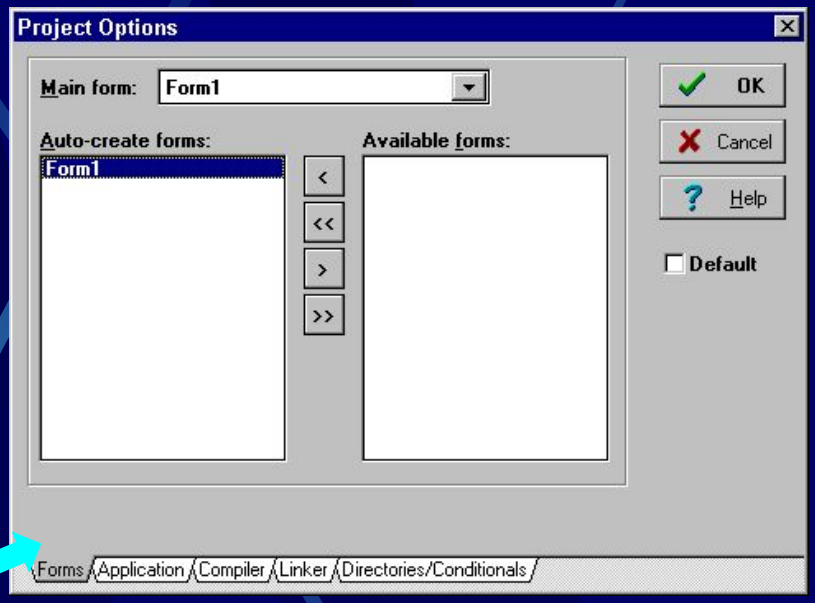
Формирование приложения

- На этапе трансляции создается также следующая группа файлов :
  - объектные файлы модулей (*.dcu или*.obj) – это откомпилированные файлы модулей (*.pas или *.cpp), которые компонуются долее редактором связей (линковщиком) в окончательный исполняемый файл;
  - исполняемый файл (*.exe) – это исполняемый файл приложения. Он является автономным исполняемым файлом, для которого больше ничего не требуется, если только вы не используете библиотеки, содержащиеся в DLL, OCX и т.д.;
  - DSM - служебный файл для запуска программы в среде, очень большой, рекомендуется стирать его при окончании работы.



Установка опций проекта

- Формирование файла проекта, как и любой группы файлов возможно с помощью традиционного пункта меню «File».
- В ИСВП для этой цели имеется также и специальное средство - Менеджер Проекта (Project Manager), который можно вызвать из пункта меню View.
- Данная стратегия типична для визуальных сред: она предоставляет несколько путей для решения одной и той же задачи, Вы сами можете решать, какой из них более эффективен в данной ситуации.
- Пункт меню «Options | Project» *наиболее сложная часть системного меню*. Это центр управления, из которого можно менять установки для проекта и для всей рабочей среды программирования.
- В «Options» наиболее часто встречаются следующие пункты (закладки):
  - «Forms», где перечисляются все формы, включенные в проект и отмечаются те, которые будут создаваться автоматически при старте программы и те, которые будут создаваться и исчезать в ходе выполнения программы с поиощью процедуры Create.



- «Application» определяются такие элементы программы, как заголовок, файл помощи и иконка
- «Compiler», на которой включаются установки для генерации кода, управления обработкой ошибок времени выполнения, синтаксиса, отладки и др.
- «Linker» можно определить условия для процесса линковки приложения
- «Directories/Conditionals» - здесь указываются директории, специфичные для данного проекта.

Постановка задачи

ФИНАНСОВЫЕ СРЕДСТВА

Средства предприятия

Собственные резервные средства – $C_{срс}$

Кредиты - $C_{кр}$

\+

Доходная часть

Выручка от продаж продукции – Function Doxod()

=

Расходная часть

Постоянные затраты **Z(n)**

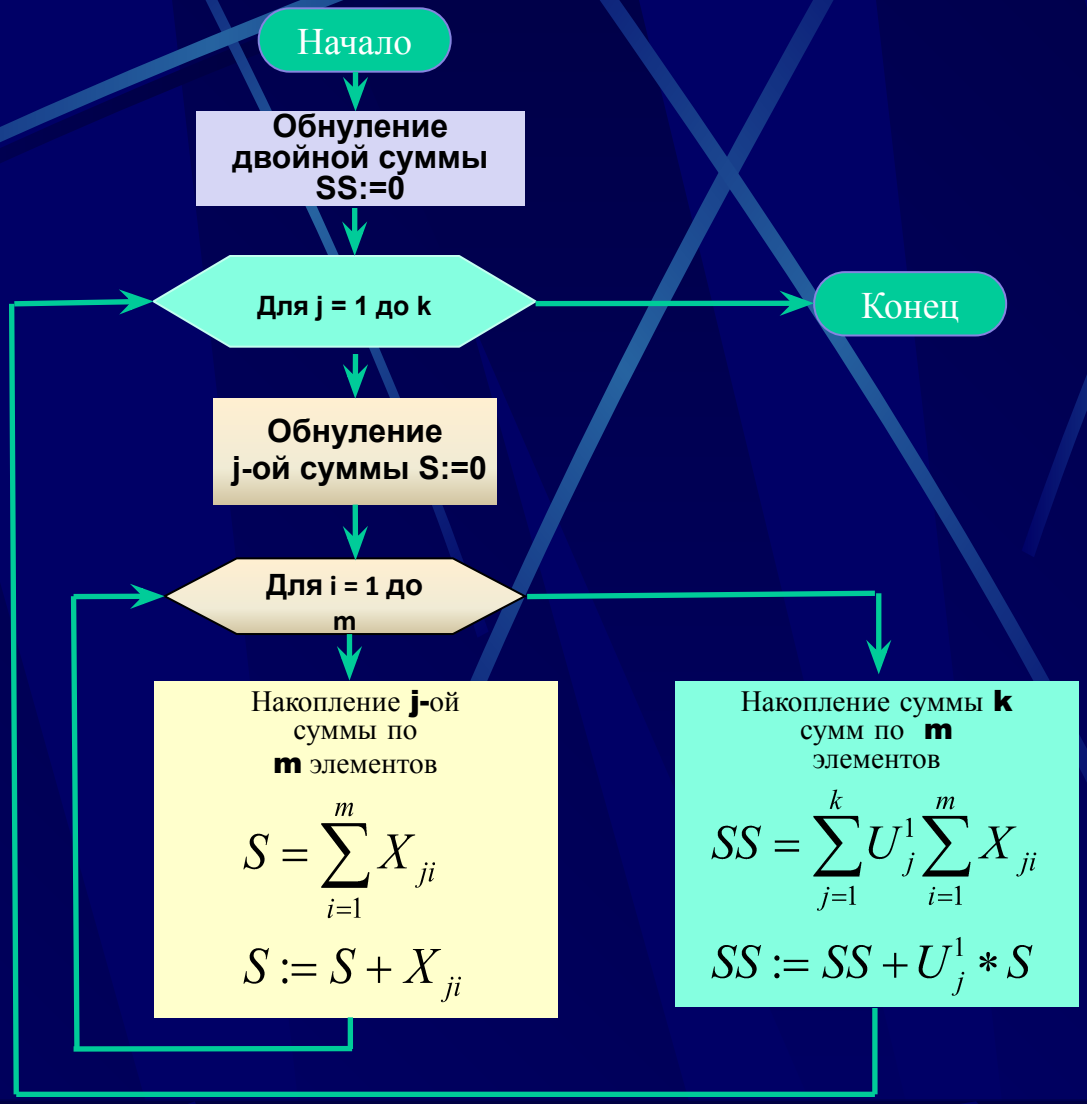
\+

Переменные затраты ***X(n) U^1(n)***

Исходные данные

Периоды / Продукты		1	2	…	i	…	m	$\sum_{i=1}^{m} X_{ji}$	U_j	$U_j \sum_{i=1}^{m} X_{ji}$
Стол	1	3	5	…	7	…	9	24	1000	24000
Стул	2	7	13		17		23	50	300	15000
…	…	…	…	…	…	…	…			
Кресло	j	4	6	…	10	…	12	32	5000	160000
…	…	…	…	…	…	…	…			
Диван	k	5	3	…	7	…	11	26	10000	260000
$\sum_{j=1}^{k} U_j \sum_{i=1}^{m} X_{ji}$										459000

Алгоритм процедуры двойного суммирования



$$X(n)U^{(1)}(n)+Z(n)=C_{cpc}(n)+C_{кр}(n)+(1-\beta)[\sum_{r=1}^{m}X(n-r)A_rЦ^{(1)}+X(n)A_0Ц^{(1)}]$$

Уравнения баланса финансовых средств

- для всех видов продукции

$$C_{cpc}+C_{кр}+(1-\alpha-\beta)\sum_{j=1}^{k}P_j^1\sum_{i=1}^{m}X_{ji}A_{ji}=\sum_{j=1}^{k}U_j^1\sum_{i=1}^{m}X_{ji}+Z$$

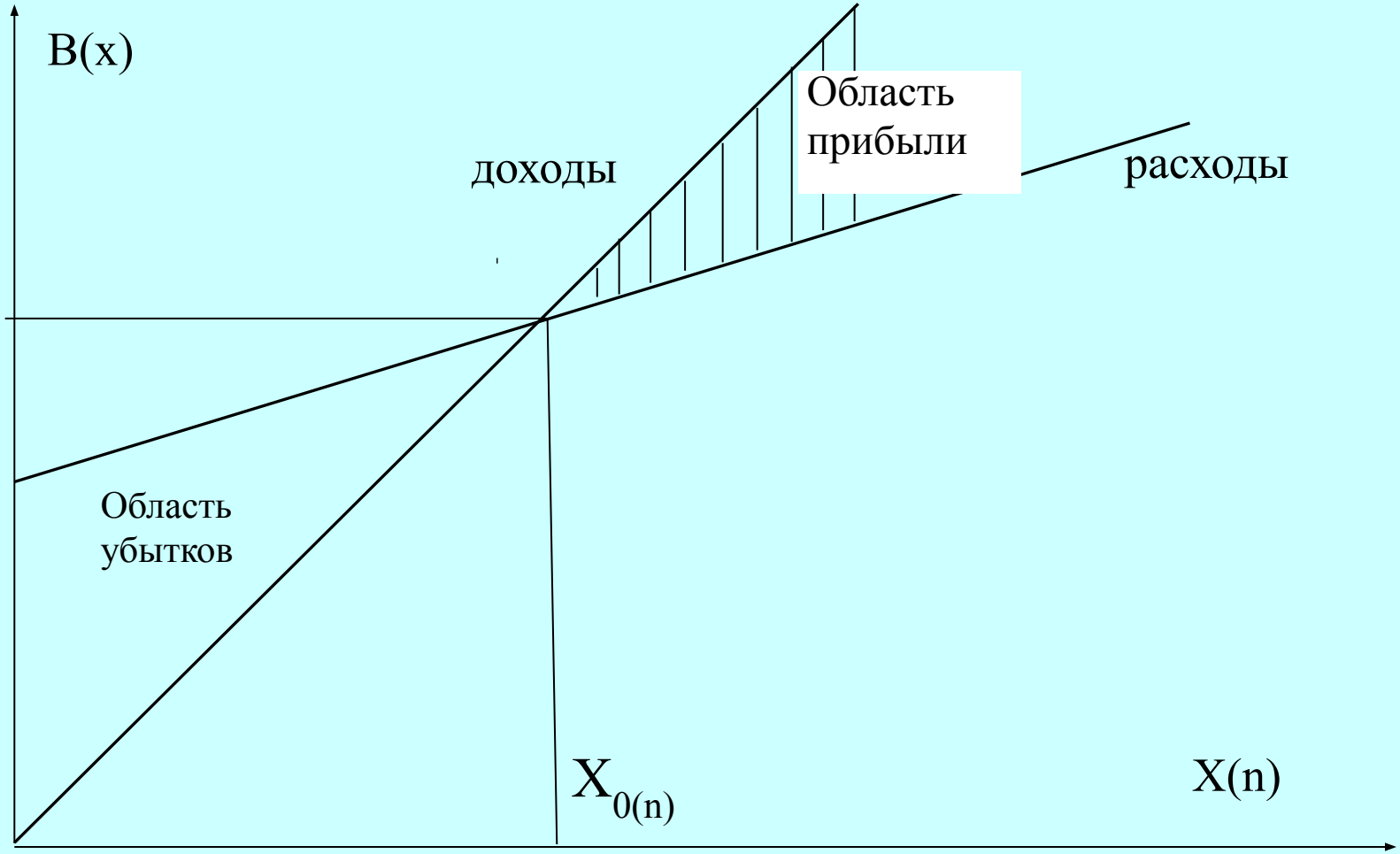
- для одного вида продукции

$$C_{cpc}+C_{кр}+(1-\alpha-\beta)P_j^1\sum_{i=1}^{m}X_{ji}A_{ji}=U_j^1\sum_{i=1}^{m}X_{ji}+Z$$

- Точка безубыточного производства для одного вида продукции

$$X_{0jm}=\frac{(1-\alpha-\beta)P_j^1\sum_{i=1}^{m-1}X_{ji}A_{ji}-Z}{U_j^1-(1-\alpha-\beta)A_{jm}P_j^1}$$

Точка безубыточного производства



Описание объектов

- Предприятие

Type

<TPredpr>= object

Свойства:

SRS: real;
Kredit: real;
PostZatr: real;
Nalog: real;
Product: array[0..k] of TProduct;

Методы:

Function Doxod: real;

end;

- Продукт

Type

<TProduct>= object

Свойства:

Name: string;
ZatrEdProd: real;
PriceEdProd: real;
V_Pprod: array[0..m] of Integer;
A_Pprod: array[0..m] of Integer;

Методы:

Function SetName: string;
Function SetPrice: real;
…

end;

Описание методов

```
Function TPredpr.Dohod: Real;
var  j,i:  integer; temp: Real;
Begin
      temp:=0;
      result:=0;
      for j:=1 to j do begin
        for i:=1 to m do begin
 temp:=temp+Product[j].V_Prod[i]*Product[j].A_Prod[i];
        end;
        result:=result+temp*(1-Nalog)*Product.PriseEdProd;
      end;
end;
```

SAUKPGP.RU

- SAUK