**IV. Программное обеспечение для объектно-ориентированного программирования и разработки приложений (Kdevelop, Lazarus, Gambas) ... 2**
1. Введение в ООП ... 2
2. Разработка ПО с использованием Kdevelop ... 16
2.1. Основы языка программирования C++ ... 17
2.2. Работа со средой разработки ПО KDevelop ... 30
3. Разработка ПО с использованием Lazarus ... 37
3.1. Основы языка программирования Pascal ... 38
3.2. Работа со средой разработки ПО Lazarus ... 48
4. Разработка ПО с использованием Gambas ... 54
4.1. Основы языка программирования BASIC (ОО диалект) ... 55
4.2. Работа со средой разработки ПО Gambas ... 61

# IV. Программное обеспечение для объектно-ориентированного программирования и разработки приложений (Kdevelop, Lazarus, Gambas)

## 1. Введение в ООП

*Объектно-ориентированное программирование (ООП)* — парадигма программирования (иными словами совокупность идей и понятий, определяющая стиль написания программ), в которой основными концепциями являются понятия **объектов** и **классов.** Парадигма программирования определяет то, в каких терминах программист описывает логику программы. Например, в *императивном* программировании (самом прямолинейном подходе, где программа эквивалентна набору инструкций, т.е. это последовательность команд, которые должен выполнить компьютер) программа описывается как последовательность действий, а в *функциональном* программировании представляется в виде выражения и множества определений функций. В самом же популярном и рассматриваемом в данном курсе объектно-ориентированном программировании программу принято рассматривать как набор взаимодействующих объектов. ООП есть, по сути, императивное программирование, дополненное принципом инкапсуляции данных и методов в объект и наследованием (принципом повторного использования разработанного функционала).

Многие современные языки специально созданы для облегчения объектно-ориентированного программирования. Однако следует отметить, что можно применять техники ООП и для не объектно-ориентированного языка и наоборот, применение объектно-ориентированного языка вовсе не означает, что код автоматически становится объектно-ориентированным.

Современный объектно-ориентированный язык предлагает, как правило, следующий обязательный набор синтаксических средств:

- Объявление классов с полями (данными) и методами (функциями).
- Механизм расширения класса (наследования) — порождение нового класса от существующего с автоматическим включением всех особенностей реализации класса-предка в состав класса-потомка.
- Средства защиты внутренней структуры классов от несанкционированного использования извне. Обычно это модификаторы доступа к

полям и методам, типа public, private, обычно также protected, иногда некоторые другие.

- Полиморфные переменные и параметры функций (методов), позволяющие присваивать одной и той же переменной экземпляры различных классов.
- Полиморфное поведение экземпляров классов за счёт использования виртуальных методов. В некоторых ООП-языках все методы классов являются виртуальными.

Видимо, минимальным традиционным объектно-ориентированным языком можно считать язык Оберон, который не содержит никаких других объектных средств, кроме вышеперечисленных. Но большинство языков добавляют к указанному минимальному набору те или иные дополнительные средства синтаксиса. В их числе:

- конструкторы, деструкторы, финализаторы;
- свойства (аксессоры);
- индексаторы;
- интерфейсы — как альтернатива множественному наследованию;
- переопределение операторов для классов;
- множество других усовершенствований базовых конструкций ООП парадигмы.

Часть языков (иногда называемых "чисто объектными") целиком построена вокруг объектных средств — в них любые данные (возможно, за небольшим числом исключений в виде встроенных скалярных типов данных) являются объектами, любой код — методом какого-либо класса и невозможно написать программу, в которой не использовались бы объекты. Примеры подобных языков — Java, C# или Ruby. Программа, на этих языках состоящая из хотя бы одной строки кода, гарантировано будет содержать хотя бы один класс и/или объект. Другие языки (иногда используется термин "гибридные") включают **ООП-подсистему** в исходно процедурный язык. В них существует возможность программировать как с привлечением ООП парадигмы, так и не обращаясь к объектным средствам. Классические примеры таких языков — **C++** и **Delphi** (Object Pascal).

Еще раз выделим разницу между объектно-ориентированным программированием (ООП) и объектно-ориентированным языком программирования (ОО-язык). **ООП** - это набор идей, концепций и методик облегчающий (как принято считать, хотя ООП постоянно

подвергается довольно аргументированной критике) разработку сложных вычислительных систем. А **ОО-язык** берет на вооружение эти идеи и методики и средствами присущего только ему синтаксиса позволяет разработчику воспользоваться ими. В целом, берясь за любой новый ОО-язык, программист может быть практически уверен, что он сможет описать класс (или его аналог в данном языке), создать экземпляр класса, вызвать методы этого класса и т.д. Единственно существенный вопрос, который может (и должен) его волновать - КАК? Как все это реализуется, т.е. буквально - с чего начать описание класса, или какой конструкцией можно вызвать его метод (функцию). Для ответов на эти вопросы и изучают справочники и спецификации по синтаксису того или иного ОО-языка.

Разумеется, каждый конкретный ОО-язык может реализовывать ОО-идеи несколько специфичным способом. Более того он может (и чаще всего так и происходит) расширять эти идеи какими-то собственными, присущими только ему или значительно более узкому кругу ОО-языков. Тем не менее, есть минимально допустимый набор концепций, который тем или иным способом реализован в абсолютном большинстве современных ОО-языков. Можно условно назвать этот набор термином **«наименьший ОО-знаменатель»**. Язык, его не реализующий, вряд ли может быть помещен в таблицу **«Современные ОО-языки»**.

Итак - что же это за минимальный набор или "наименьший ОО-знаменатель"? Без чего решительно невозможно представить ОО-язык? Какие концепции являются буквально несущей конструкцией ООП? Их четыре:

**Абстракция данных**

Классы представляют собою упрощенное, идеализированное описание реальных сущностей предметной области. Если соответствующие модели адекватны решаемой задаче, то работать с ними оказывается намного удобнее, чем с низкоуровневым описанием всех возможных свойств и реакций класса.

Пример: машина (автомобиль) - реальная сущность реального мира, класс Car - ее модель в мире программном. Вообще любой автомобиль характеризуется просто огромной кучей параметров - цвет, марка, модель, габариты, вес, грузоподъемность, число и диаметр колес и т.д. Обязан ли программист в классе Car полностью смоделировать машину реальную и указать все ее характеристики? Нет, ему достаточно создать модель достаточную и адекватную решаемой задаче. Например, мы можем постановить, что для создаваемой нами программы необходимо и достаточно знать всего два параметра: дату изготовления автомобиля и тип его кузова.

Тогда, очевидно, мы разместим в классе Car 2 поля (поскольку атрибуты объектов реальных отражаются как раз на поля классов) с условными именами ДатаВыпуска и КузовТип. Если мы не ошиблись в наших оценках и этих двух полей действительно достаточно - все отлично, мы можем абстрагироваться от наличия прочих характеристик автомобиля реального. Однако если выяснится что программе (в лице других классов-потребителей) требуются вовсе не ДатаВыпуска и КузовТип а как раз ЦветКузова, МаксимальныйВес и Производитель нам придется имеющийся класс Car выкинуть и попросту начать с нуля. Дабы избежать подобных накладок и не тратить попусту труд и время разработчика ООП должно начинаться вовсе не с написания кода, а с вещи, возможно, даже более важной чем правильный и красивый код - с объектно-ориентированного *проектирования* будущей программы. Именно тут архитектор ПО должен принять стратегические решения - какие из характеристик объектов реальных необходимы и достаточны для их программного моделирования.

**Инкапсуляция**

Это принцип, согласно которому любой класс должен рассматриваться как **«чёрный ящик»** — пользователь класса (а им, очевидно, будет какой-то второй класс, класс-потребитель) должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

Так же пример: телефон - реальная сущность реального мира, класс Phone - ее модель в мире программном. В телефоне реальном человек-пользователь (аналог класса-потребителя в мире программном) снимает трубку - набирает номер - кладет трубку. Все, это вся информация которую ему необходимо знать. Аналогично, класс Phone должен "выставлять наружу" примерно следующие 3 метода:

- *ТрубкуПоднять()*
- *НажатьКлавишу(номер_клавиши)*
- *ТрубкуОпустить()*

То, что при каждом нажатии клавиши телефона программного (как и реального) «реле FR45 посылает пиловидный импульс в выходную телефонную линию» *не является* необходимым знанием для ежедневной эксплуатации телефона и этот функционал не должен быть доступен ни реальному человеку, ни классу-потребителю. Поэтому метод *Импульс_В_Линию()* был бы объявлен внутри класса Phone как приватный, т.к. очевидно сам телефон (внутри) должен уметь посылать сигналы в линию, но вот сделать этот метод доступным извне, прочим классам, было бы грубой ошибкой и нарушением обсуждаемого принципа «черного ящика». Вызов публичного метода *НажатьКлавишу(номер_клавиши)* приводил бы к вызову метода приватного *Импульс_В_Линию()* что прекрасно согласуется с моделируемым объектом: нажатие физической клавиши действительно приводит к активности физического реле. Однако ситуация, когда никакая клавиша не нажата, а импульс все же сгенерен смотрелась бы очень странно и никак не укладывается в привычные рамки реального мира. «Черный ящик» нас от таких ситуаций успешно охраняет.

Второй пример, освещающий иной аспект того же самого принципа - аспект контролируемого доступа к состоянию класса. Предположим существование реального объекта - музыкального mp3-плеера. Не программного, а именно в виде коробочки с наушниками. Так же предположим, что весь его "интерфейс" состоит из одной физической кнопки А. Короткое нажатие на кнопку А переключает плеер между состояниями проигрывание/пауза и больше никакого управления не предусмотрено. Очевидно, что при программном моделировании класса Player мы бы ввели в него поле *ПоложениеКнопкиА*, т.к. статус этой кнопки является одной из характеристик, а они, как уже известно, отражаются на поля. Сначала давайте рассмотрим вариант, когда мы даем классу-потребителю прямой доступ к этому полю. Какие значения этот класс-потребитель мог бы в это поле поместить? Очевидно, он мог бы указать значение НАЖАТО (условный цифровой код 1) и ОТПУЩЕНО (условный цифровой код 0). Если бы классу- потребитель гарантировано предавал только эти 2 значения, да еще гарнировано в правильном порядке (что бы за каждым НАЖАТО непременно следовало ОТПУЩЕНО) - проблем бы не было никаких. Плеер реальный, как и Player-класс прекрасно понимают что надо делать при переходе из одного состояния в другое. Однако - как эти плееры должны реагировать при смене состояния НАЖАТО (1) опять в НАЖАТО (1)? Или - что делать при помещении в поле *ПоложениеКнопкиА*

значения СДВИГ_ВПРАВО (условный цифровой код 2)? Наша кнопка А не умеет быть нажатой дважды без отпускания и уж тем более не умеет сдвигаться ни вправо, ни влево! Вот потому-то принцип инкапсуляции и утверждает - никаких непосредственных изменений внутреннего состояния класса извне, а только через предусмотренный специально для этих целей интерфейс. В случае класса Player было бы грамотным шагом введение публичного (доступного любому заинтересованному классу) метода с именем вроде *А_Нажать_И_Отпустить()*. Тогда бы мы полностью и корректно смоделировали поведение нашего реального прототипа. Точно как и на "живом" mp3-плеере потребители класса Player теперь могут лишь нажать_и_отпустить кнопку А. А вот уже внутри метода *А_Нажать_И_Отпустить()* автор этого класса сам может изменять значение поля *ПоложениеКнопкиА*. Но такое изменение статуса является куда как более контролируемым и подвержено значительно меньшей вероятности возникновения ошибок. Авторы классов прекрасно осведомлены о внутреннем состоянии своих "питомцев" и, почти наверняка, автору класса Player и в голову не придет попытаться двигать кнопку в стороны вместо того, что бы нажимать на нее. Проблема заключается в том, что бы помешать сделать это гораздо менее искушенным во внутреннем устройстве класса Player потребителям и принцип "черного ящика" дает ответ этой проблеме. Коротко этот принцип формулируется так: классы-потребители в принципе не имеют доступа к приватным членам потребляемого ими класса, а, следовательно, не могут произвольно менять его состояние на любое им понравившееся. Любое изменение в статусе производится только с одобрения самого этого класса.

**Наследование**

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность.

Традиционный пример, снова обратимся к машинам (автомобилям). Допустим нам необходимо реализовать классы ЛегковойАвтомобиль и ГрузовойАвтомобиль. Очевидно, эти два класса имеют общую функциональность. Так, оба они имеют 4 колеса (для упрощения сделаем в вид, что 6-ти и более колесного транспорта не

существует), двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, независимо от того, грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный класс, например, Автомобиль и наследовать от него два класса - ЛегковойАвтомобиль и ГрузовойАвтомобиль. Тогда поле *МестПосадочных* достаточно ввести только в классе Автомобиль, а два унаследованных класса получат это поле буквально "по наследству". Останется лишь установить нужное значение этого поля через публичный интерфейс. Более точно, если мы хотим, что бы потребители наших классов ЛегковойАвтомобиль и ГрузовойАвтомобиль могли изменять значение этого поля мы должны предусмотреть для них публичный интерфейс. Почему именно интерфейс? Потому что так диктует принцип инкапсуляции, он же "черный ящик". А что если мы попробуем игнорировать этот принцип? В самом деле - ну зачем нам еще дополнительная прокладка в виде метода *УстановитьЧислоМестПосадочных()*? Почему бы не позволить потребителям быстренько менять значение поля *МестПосадочных* без всяких посредников? Ну то что "быстренько" это, конечно, хорошо, но вот что делать двум проектируемым классам когда они однажды вдруг выяснят что могут вместить 7500 человек? Или, что еще "лучше", -5 человек? Вот для отсечения таких ситуаций обсужденный ранее принцип инкапсуляции соблюдать не просто "хорошо бы", а категорически необходимо. *УстановитьЧислоМестПосадочных()* встанет на стражу наших интересов и не позволит моделировать нам (а более точно - потребителям создаваемых нами классов) автомобили с пассажирской емкостью в 20 самолетов. Кстати, и сам этот метод можно объявить в базовом классе, Автомобиль, наряду с полем *МестПосадочных*. Это еще больше подчеркнет преимущества обсуждаемого принципа - создается метод лишь однажды, а доступен потребителям как базового класса (Автомобиль) так и потребителям любого унаследованного от него класса.

**Полиморфизм**

Полиморфизмом называют явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением сигнатуры метода. Иными словами

требуется сохранить имя метода, тип и число его параметров, но можно изменить его тело. Это обеспечивает сохранение неизменным интерфейса класса-предка и позволяет осуществить связывание имени метода в коде с разными классами — из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем. Такой механизм называется динамическим (или поздним) связыванием — в отличие от статического (раннего) связывания, осуществляемого на этапе компиляции.

В качестве примера этой концепции продолжим работу с базовым классом Автомобиль и двумя его унаследованными классами из предыдущего раздела. Как вы помните, мы закончили созданием публичного метода *УстановитьЧислоМестПосадочных()* причем объявили его именно в базовом классе. В настоящий момент метод работает совершенно идентично, вызовем ли мы его из класса Автомобиль, или из класса ЛегковойАвтомобиль, или ГрузовойАвтомобиль. Напомню, что два последних класса получили этот метод "по наследству" из базового. Т.к. у нас есть физически один метод не удивительно, что именно он и будет отрабатывать из какого бы класса мы его не вызывали. Усложним задачу. Введем новое приватное поля в класс Автомобиль - *ЧислоОбращений*. Изначально значением этого поля будет 0, и, точно как и поле *МестПосадочных*, оно будет унаследовано обоими производными (иное название класса, унаследованного от какого-то базового класса) классами. Теперь сформулируем новые требования:

- если метод *УстановитьЧислоМестПосадочных()* вызывается из класса Автомобиль - изменить (после проверки) значение поля *МестПосадочных* на указанное значение и больше ничего не делать
- если метод *УстановитьЧислоМестПосадочных()* вызывается из класса ЛегковойАвтомобиль - изменить (после проверки) значение поля *МестПосадочных* на указанное значение, а поле *ЧислоОбращений* увеличить на единицу
- если метод *УстановитьЧислоМестПосадочных()* вызывается из класса ГрузовойАвтомобиль - изменить (после проверки) значение поля *МестПосадочных* на указанное значение, а поле *ЧислоОбращений* уменьшить на единицу

Итак, теперь единый до этого момента метод *УстановитьЧислоМестПосадочных* распадается на 3 части:

- часть общая для <u>всех</u> классов - проверка переданного ему значения и если все хорошо присвоение этого значения полю *МестПосадочных*
- часть характерная только для класса ЛегковойАвтомобиль, а именно +1 к *ЧислоОбращений*
- часть характерная только для класса ГрузовойАвтомобиль, а именно -1 к *ЧислоОбращений*

Разумеется практической смысловой нагрузки в таких требованиях не много, но как иллюстративный пример они подходят как нельзя лучше. Как же полиморфизм позволит нам изящно подстроиться под изменившиеся условия? А вот как:

- в классе Автомобиль оставим в методе *УстановитьЧислоМестПосадочных()* только общую часть и обозначим его как "специальный" (с термином "специальный метод" мы разберемся чуть позже). Что бы не привязываться к синтаксису конкретного ОО-языка запишем это на псевдоязыке:

```
класс Автомобиль
{
    специальный
УстановитьЧислоМестПосадочных(места)
    {
        Проверить что число места разумно
        Если проверка прошла -
        поле МестПосадочных=места
}
}
```

- в унаследованных классах объявим <u>тот же самый</u> метод который во-первых, вызовет свою реализацию из вышестоящего класса Автомобиль (иногда ее называют базовой реализаций метода), а во-вторых предпримет действия уникальные именно для этого класса. Снова псевдокод:

```
класс ЛегковойАвтомобиль унаследован_от Автомобиль
{
    специальный
УстановитьЧислоМестПосадочных(места)
```

```
        {
    Базовый.УстановитьЧислоМестПосадочных(места)
            ЧислоОбращений=ЧислоОбращений+1
    }
    }

    класс ГрузовойАвтомобиль унаследован_от Автомобиль
    {
        специальный
УстановитьЧислоМестПосадочных(места)
        {
    Базовый.УстановитьЧислоМестПосадочных(места)
            ЧислоОбращений=ЧислоОбращений-1
    }
    }
```

Что получилось в результате? В результате у нас не 1, а 3 варианта одного и того же метода. Какой из них будет вызван зависит от того на каком классе будет вызван этот метод. Т.е. теперь ЛегковойАвтомобиль.*УстановитьЧислоМестПосадочных(5)* совсем не тоже самое, что Автомобиль.*УстановитьЧислоМестПосадочных(5)*, хотя первый метод и вызовет второй. Это, кстати, вовсе не обязанность унаследованного метода. Т.е. метод в ЛегковойАвтомобиль совершенно законно мог бы игнорировать своего коллегу-предшественника из класса Автомобиль, просто в нашей конкретной задаче оказалось удобным все же к нему обратиться.

Почему же в новой реализации потребовалось метод *УстановитьЧислоМестПосадочных()* обозначать как специальный и что вообще означает термин "специальный метод"? Специальным он стал потому, что компилятор должен обработать его особым образом, а вот методика и синтаксис такого "специального" обозначения разнится от ОО-языка к ОО-языку. Однако в большинстве ОО-языков "специальные" (в смысле разбираемой темы) методы называются *виртуальными*, а их синтаксическое обозначение обыгрывает или

крутится вокруг слова *virtual*. Как отмечалось во введении к этому уроку есть ОО-языки где все методы изначально являются *virtual*, в них никаких особых обозначений не требуется. Тем не менее характерный ОО-язык все же проводит явное синтаксическое различие между методами обычными (не виртуальными) и виртуальными.

Ну а в чем же выражается упомянутая "специальная обработка" виртуального метода компилятором? Выражается она в том, что связь между виртуальным методом и вызывающими его процедурами и функциями устанавливается не во время компиляции (это называется *ранним связыванием*), а во время выполнения программы (*позднее связывание*). Иными словами, в то время когда метод *УстановитьЧислоМестПосадочных()* еще не был переделан нами в виртуальный, запись ЛегковойАвтомобиль.*УстановитьЧислоМестПосадочных(5)* однозначно транслировалась компилятором в Автомобиль.*УстановитьЧислоМестПосадочных(5)*. Невиртуальный метод всегда однозначно вызывается из того класса, который его объявил. После переделки того же метода в виртуальный компилятор встретив ту же строчку кода уже подобной трансляции не производит, а оставляет вопрос "какой же на самом деле метод вызвать?" до момента непосредственного вызова *УстановитьЧислоМестПосадочных()* во время выполнения программы. Когда теперь во время выполнения класс-потребитель заявляет что ему требуется именно метод класса ЛегковойАвтомобиль - именно так и случается. А если другой потребитель затребует тот же метод из ГрузовойАвтомобиль - то получит ожидаемое. Отметим, что невиртуальный метод вызывается чуть-чуть быстрее, т.к. исполняющая программа изначально (еще до запуска) точно знает адрес целевого метода (*УстановитьЧислоМестПосадочных*, в нашем примере). Адрес же виртуального метода рассчитывается непосредственно перед его вызовом (в момент когда программа уже работает). Однако расчеты эти столь ничтожны, а мощности современных настольных (не говоря уже о серверных) компьютеров столь значительны, что указанные потери в производительности программы можно полностью игнорировать.

*Итак - резюмируем четыре постулата ООП:*

**Абстракция данных** - классы моделируют объекты реального мира, однако делают это лишь до известной степени, не пытаясь достичь соответствия "один-к-одному"

**Инкапсуляция** - классы выставляют для потребителей абсолютно минимально необходимый публичный интерфейс, все остальное должно быть скрыто

**Наследование** - взаимоотношения между классами напоминают взаимоотношения между объектами реального мира и их группами, где Кошка - специализированный вид (класс) Млекопитающего, а те, в свою очередь, специализированный вид (класс) Животного. Каждый унаследованный класс точно умеет делать все что делает класс базовый (Кошка умеет кормить потомство молоком, как все Млекопитающие), но может (хоть и не обязан) привносить свою собственную функциональность (только Кошка умеет мяукать, ловить мышей, и шипеть на собак)

**Полиморфизм** - одни и те же функции (методы) в классе базовом и производном могут иметь функционал и полностью идентичный, и частично совпадающий (см. наш последний пример с *УстановитьЧислоМестПосадочных*), и абсолютно различный.

А кто является главными "действующими лицами" в ООП-мире? С какими сущностями чаще всего доведется работать ООП-программисту? Их всего две - класс и объект. И о первом (особенно), и о втором мы уже упоминали неоднократно, однако надо дать формальное их определение и, что более важно, очертить их взаимосвязь и их отличие друг от друга.

Итак, класс, как можно уже было предположить из предшествующего рассказа, это программная сущность - модель реального объекта реального мира (хотя, разумеется, никто не может помешать вам смоделировать выдуманный объект мира фэнтезийного). Обычно классы состоят из полей (там хранятся атрибуты объектов типа цвета кузова авто и их состояния, типа запущен ли двигатель, или остановлен) и функций/методов (действия, который класс "умеет" выполнять - *ДвигательСтарт()*, *ДвигательЗаглушить()*, *ЛеваяПередняяДверьЗакрыть()* и т.д.).

Что же до объектов, то в большинстве объектно- ориентированных языков программирования (таких как Java, C++ или C#), объекты являются *экземплярами* некоторого заранее описанного класса. Объекты в таких языках создаются с помощью *конструктора класса*, и уничтожаются (когда они становятся ненужными) либо с помощью деструктора класса (например, в C++), либо автоматически с использованием сборщика мусора (в Java, C#). Иными словами, класс - это план, по которому будет создан объект. Рассмотрим пример на псевдокоде:

класс Дом

```
{
    цветКрыши
    количествоОкон

    цветКрышиЗадать()
    количествоОконЗадать()
    всеОкнаОткрыть()
}
```

Примерно так мы бы могли смоделировать реальный дом программно. Из целой кучи характеристик дома реального мы могли бы остановиться всего на двух (цвете крыши и количестве окон) и ограничиться двумя методами вспомогательными, помогающими эти характеристики устанавливать классам потребителям("черный ящик" - помните?) и одним методом, утверждающим что наш класс будет уметь открывать окна. Самое интересно, что последняя фраза не совсем корректна. Класс - это проект, а не объект программного мира. Класс действительно мог бы открыть окна если бы они у него были. Но их нет! Они только будут у объекта созданного "по кальке" класса.

Тут не совсем удачно пересекаются термины объект (реального мира; из него архитектор ПО придумывает класс) и одноименный ему объект (мира программного, получаемый из чертежа-класса). Для прояснения картины запомним схему - кто откуда "вытекает":

ОБЪЕКТ реальный (автомобиль) → класс Car → ОБЪЕКТ программный (MyGreenCar)

Так же отметим, что ОБЪЕКТ программный иногда называют *экземпляром*, однако термин объект все же употребляется чаще даже невзирая на его неоднозначность.

Имея "чертеж" дома (класс Дом) мы можем наштамповать сколько угодно домов-объектов, например:

```
МойДом=новый Дом()
МойДом.цветКрышиЗадать("зеленый")
МойДом.количествоОконЗадать(3)
ДомРодителей=новый Дом()
ДомРодителей.цветКрышиЗадать("синий")
ДомРодителей.количествоОконЗадать(4)
ДомПриятеля=новый Дом()
```

ДомПриятеля.цветКрышиЗадать("красный")

ДомПриятеля.количествоОконЗадать(5)

...

Элемент кода вида `новый Дом()` присутствует в том или ином синтаксисе в любом ОО-языке и называется *конструктором объекта*. Его назначение именно такое какое и вытекает из названия - сконструировать новый объект по классу-чертежу и позволить нам получить доступ к его функциям / методам.

Как видите проблема только одна - придумать каждому объекту уникальное имя, и в общем случае совпадение с именем класса не допускается. И вот только теперь, когда у нас есть программный объект - копия (пусть и сильно усеченная) объекта реального мы можем начинать открывать окна:

МойДом.всеОкнаОткрыть()

ДомРодителей.всеОкнаОткрыть()

ДомПриятеля.всеОкнаОткрыть()

При этом в общем случае (однако - см. примечание в конце урока) строка

Дом.всеОкнаОткрыть()

вызовет ошибку компиляции. Класс определяет, что будут уметь делать полученные из него (или, если хотите, "от него") объекты, но не он сам.

Итого, ответ на поставленный ранее вопрос: объект (программный) создается по "чертежу"-классу и в этом смысле является полностью от него зависящим. Не будет класса - определенно не будет объекта. В то же время, класс всего лишь планирует характеристики (атрибуты) и функционал будущих своих объектов, но сам ими не обладает и производить какую-либо активность не может. Иными словами, что бы воспользоваться функциональностью заложенной в классе (открыть окно в Доме) потребитель обязан создать его объект; иначе требуемая функциональность будет попросту недоступна.

## 2. Разработка ПО с использованием Kdevelop

*KDevelop* — свободная среда разработки программного обеспечения для UNIX-подобных операционных систем. KDevelop не включает в свой состав компилятор; иными словами сам по себе KDevelop не умеет переводить текст написанный программистом в код годный для выполнения на компьютере. KDevelop только предоставляет удобную среду для набора такого текста, его редактирования, отладки и прочих типичных задач разработчика. Однако для самого важного этапа в разработке ПО - перевод текста программы на языке высокого уровня (допустим Pascal), в эквивалентную программу на машинном языке (или, как еще говорят, перевод в двоичный код) - ему требуется помощь внешней программы-компилятора. Такие программы существуют сами по себе, независимо от KDevelop и в целом текст набранный в любом редакторе (в KDevelop в т.ч.) можно использовать как входной файл для компилятора вне зависимости от того установлена у нас среда разработки или ее нет вообще. Компилятор наличием/отсутствием KDevelop не интересуется; ему нужен синтаксически корректный файл на входе и более ничего, методика создания этого файла компилятору безразлична.

Однако KDevelop умеет очень гладко осуществлять передачу набранного в нем текста разрабатываемой программы компилятору, прием сообщений от компилятора о ходе компиляции и, по окончании этого процесса, выдачу соответствующих сообщений программисту. Таким сообщением может быть известие о том, что ошибок компиляции не обнаружено, а поэтому исполнительный (двоичный) модуль успешно собран и готов к тестовому запуску. Или, напротив, сообщение о, например, 3-х ошибках компиляции с указанием строк исходной программы их вызвавших. Ясно, что такой подход сильно повышает эффективность работы разработчика по сравнению с "прямой" работой с компилятором и чтением выдаваемых им сообщений в консольном окне, да и редактор встроенный в эту среду значительно более удобен и продвинут по сравнению со стандартным.

KDevelop умеет взаимодействовать описанным выше способом не с одним конкретным компилятором, а с их множеством. Т.е. используя эту среду можно создавать программы на многих языках программирования. Далеко не полный их перечень включает в себя такие языки как Ада, Bash, C, C++, Фортран, Java, Pascal, Perl, PHP, Python, Ruby и SQL. При этом и программист на Фортране и Паскаль-программист работают в одной и той же графической среде, с одним и тем же редактором и могут пользоваться всеми инструментами

и вспомогательными окнами встроенными в KDevelop. Различие (да и то не столь явно выраженное визуально) проявится в момент когда каждый из них даст команду на компиляцию своей программы (или, как еще говорят, на сборку проекта): у первого отработает Фортран-компилятор, а у второго Паскаль- компилятор. Однако они оба получат соответствующие уведомления в конце этого процесса.

Разумеется, в рамках нашего курса не возможно рассмотреть создание программ на всех и каждом языке из списка поддерживаемых KDevelop. Поэтому в качестве нашего рабочего инструмента выберем C++ как один из (если не самый) популярных ОО-языков. Т.е. мы готовы попробовать создавать несложные программы на языке C++ в среде KDevelop. Для этого, очевидно, мы должны быть компетентны в двух не связанных областях компьютерной науки:

- мы должны знать и уметь писать синтаксически корректные программы на C++ как таковом, вне зависимости от используемой среды разработки
- мы должны знать KDevelop именно как приложение и уметь пользоваться предлагаемым им функционалом опять же, вне зависимости от языка на котором мы будем работать

Соединение этих двух компетенций приведет к созданию именно C++ - программ именно с помощью среды KDevelop.

Рассмотрим эти вопросы последовательно в следующих двух подразделах.

## 2.1. Основы языка программирования C++

*C++* — компилируемый строго типизированный язык программирования общего назначения. Поддерживает разные парадигмы программирования, но наибольшее внимание уделено поддержке объектно-ориентированного подхода. C++ справедливо считают трудным для изучения языком в виду его сложности и избыточности.

В виду ограниченного объема нашего курса и несколько иной его направленности мы ограничимся лишь самыми базовыми конструкциями языка, да и то не всеми. При желании обучение может быть продолжено самостоятельно по учебникам и статьям, опубликованным в печатной и электронной прессе (Интернет). Объем информации находящейся в сети даже в свободном доступе более чем достаточен для изучения C++ с любой степенью погружения в его нюансы. Отметим, что базовое знание английского языка (технического) хоть и не является обязательным условием для успешного освоения C++, способно сильно облегчить и ускорить учебный процесс.

Итак, простейшая **C++** программа производящая какую-то активность может иметь вид:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Учимся программировать на языке C++!";
  return 0;
}
```

Разберем ее строчка за строчкой:

**#include <iostream>**

При компиляции программы директива препроцессора **#include** заставляет компилятор включить содержимое заданного файла в начало вашей программы. В данном случае компилятор включит содержимое файла *iostream*. Файлы которые вы включаете в начало (или заголовок) вашей программы, называются заголовочными файлами. Обычно они имеют расширение .h, однако данный случай - особый, нам нужен именно файл iostream (без расширения), а не iostream.h. В данный момент не беспокойтесь о внутреннем содержимом заголовочных файлов. Просто поймите, что оператор #include позволяет вам использовать эти файлы. Каждая создаваемая вами программа на C++ начинается с одного или нескольких операторов #include. Эти операторы указывают компилятору включить содержимое заданного файла (заголовочного файла) в вашу программу, как если бы программа содержала операторы, которые находятся во включаемом файле. Замечание: никогда не изменяйте содержимое заголовочных файлов. Это может привести к ошибкам компиляции в каждой создаваемой вами программе. Однако просмотр (без редактирования) этих файлов является хорошей практикой для постижения внутренней структуры C++ - программ.

**using namespace std;**

Говорит о том, что в дальнейшем тексте программы необходимо использовать идентификаторы из указанного именного пространства (*namespace* - пространство имен). Пока можно просто принять как данность, что наша первая C++-программа "желает" работать в пространстве имен с именем **std**, и отложить рассмотрение самой концепции *namespace* до лучших времен.

**int main()**

В каждой С++ программе должна быть ровно одна функция с именем **main().**Исполнение программы начинается с выполнения первой инструкции функции main(), в нашем случае – cout. Затем одна за другой исполняются все дальнейшие инструкции (которых в нашем минимальном примере нет), и, выполнив последнюю инструкцию функции main(), программа заканчивает работу.

Функция (не только main, а любая) состоит их четырех частей: типа возвращаемого значения, имени, списка параметров и тела функции. Первые три части составляют прототип функции.

Список параметров заключается в круглые скобки и следует всегда за именем функции. В нашем случае пустые круглые скобки означают, что main не ждет никаких параметров. Слово **int** на месте типа возвращаемого значения говорит о том метод main собирается вернуть какое-то значение целого типа (int - целое число). Тело функции содержит последовательность исполняемых инструкций и ограничено фигурными скобками. Второе назначение таких скобок - группировка операторов. Это такой набор операторов, которые компьютер должен выполнить определенное число раз (цикл), или другой набор операторов, которые компьютер должен выполнить, если выполняется определенное условие. В обоих случаях мы увидим открывающую и закрывающую фигурную скобки внутри тела той или иной функции. Наш пример слишком прост для циклов и условий и никаких группировок не использует.

**cout << "Учимся программировать на языке С++!";**

Третья инструкция является инструкцией вывода. cout – это выходной поток, направленный на терминал, << – оператор вывода. Полностью эта инструкция выводит в cout – то есть на терминал – символьную константу, заключенную в двойные кавычки. В результате выполнения данной инструкции мы получим на терминале сообщение:

**Учимся программировать на языке С++**

**return 0;**

Наш метод main обязался вернуть некоторое целое значение. Для этого следует воспользоваться оператором return после которого и указать это самое значение. Если оператор return опустить ошибки компиляции не произойдет, однако во время выполнения наш метод будет возвращать некоторое случайное значение ("мусор").

Точка с запятой в конце инструкции собственно, обозначает именно этот факт - конец иструкции. Программист обязан ставить этот символ (;) в конце каждой инструкции.

Типичная программа с функциональностью хотя бы чуть-чуть сложнее только что показанной определенно должна хранить какую-то информацию во время своего выполнения. Например, программе, печатающей файл, нужно знать имя файла и, возможно, число копий, которые вы хотите напечатать. В процессе выполнения программы хранят такую информацию в памяти компьютера. Чтобы использовать определенные ячейки памяти, программы применяют переменные. Проще говоря, переменная представляет собой имя ячейки памяти, которая может хранить конкретное значение. Когда вы присваиваете значение переменной, представьте себе переменную в виде ящика, в который можно поместить значение. Если вам позже потребуется использовать значение переменной, компьютер просто посмотрит значение, содержащееся в ящике.

Ваши программы используют переменные для хранения информации. В зависимости от типа хранимого значения, например, целое число, буква алфавита или число с плавающей точкой, тип вашей переменной будет разным. Тип переменной указывает тип значения, хранимого в переменной, а также набор операций (таких как сложение, умножение и другие), которые программа может выполнять над значением переменной. Большинство программ на С+ + будут использовать типы переменных, перечисленные в табл. 3.1

| Тип | Диапазон хранимых значений |
|---|---|
| char | Значения в диапазоне от -128 до 127. Обычно используется для хранения букв алфавита |
| int | Значения в диапазоне от -32768 до 32767 |
| unsigned | Значения в диапазоне от 0 до 65535 |
| long | Значения в диапазоне от -2147483648 до 2147483647 |
| float | Значения в диапазоне от -3.4 х 10-38 до 3.4 х 1038 |
| double | Значения в диапазоне от 1.7х 10-308 до 1.7х 10308 |

Таблица .1. Наиболее востребованные типы переменных С++.

Прежде чем вы сможете использовать переменную, ваша программа должна ее объявить. Другими словами, вам следует представить переменную компилятору С++. Чтобы объявить переменную в программе, вам следует указать тип переменной и ее имя, по

которому программа будет обращаться к данной переменной. Указывайте тип и имя переменной после открывающей фигурной скобки главной программы, как показано ниже:

**тип_переменной имя_переменной;**

Как правило, тип переменной будет одним из типов, перечисленных в табл. 3.1. Выбираемое вами имя переменной должно нести смысловую нагрузку, которая описывает (для всех, кто читает вашу программу) использование переменной. Например, ваша программа могла бы использовать переменные, такие как employee_name, employee_age и т. д. Обратите внимание на точку с запятой, которая следует за именем переменной. В С++ объявление переменной считается инструкцией. Поэтому вы должны поставить после объявления точку с запятой, точно так же как ставите ее в конце инструкции производящей какую-то активность типа печати строки.

Фрагмент следующей программы объявляет три переменные, используя типы **int, float** и **long**:

```
#include <iostream>
int main()
{
  int test_score;
  float salary;
  long distance_to_mars;
}
```

Важно обратить внимание, что данная программа ничего не выполняет, а только объявляет переменные (тем не менее, формально это полноценная С++ - программа). Как видите, объявление каждой переменной заканчивается точкой с запятой. Если вы объявляете несколько переменных одного и того же типа, можно разделять их имена запятой. Следующий оператор, например, объявляет три переменных с плавающей точкой:

**float salary, income_tax, retirement_fund;**

После объявления переменной вы используете оператор присваивания С++ (знак равно), чтобы присвоить значение переменной. Фрагмент следующей программы сначала объявляет переменные, а затем использует оператор присваивания, чтобы присвоить переменным значения:

```
#include <iostream>
int main()
```

```
{
  int age;
  float salary;
  long distance_to_the_moon;

  age = 32;
  salary = 25000.75;
  distance_to_the_moon = 238857;
}
```

При объявлении переменной часто удобно присваивать ей начальное значение. Чтобы упростить такую процедуру, C++ позволяет присваивать значение во время объявления переменной (т.е. свести объявление и присвоение в одну инструкцию), как показано ниже:

```
int age = 32;
float salary = 25000.75;
long distance_to_the_moon = 238857;
```

После присвоения значения переменной ваши программы могут использовать это значение, просто обращаясь к ее имени. Следующая программа присваивает значения трем переменным и затем выводит значение каждой переменной, используя cout:

```
#include <iostream>
using namespace std;
int main()
{
  int age = 32;
  float salary = 25000.7;
  long dis_to_moon = 238857;
  cout << "Служащему " << age << " года (лет)" << endl;
  cout << "Оклад служащего составляет $" << salary << endl;
  cout << "От земли до луны " << dis_to_moon << " миль" << endl;
}
```

Когда вы откомпилируете и запустите эту программу, на экране появится следующий вывод:

Служащему 32 года (лет)

Оклад служащего составляет $25000.7

От земли до луны 238857 миль

Независимо от целевого назначения большинство ваших программ на С++ будут складывать, вычитать, умножать или делить те или иные значения. Ваши программы могут выполнять арифметические операции с константами (например, 3*5) или с переменными (например, payment — total). Таблица 2 перечисляет основные математические операции С++:

| Операция | Назначение | Пример |
|---|---|---|
| + | Сложение | total = cost + tax; |
| - | Вычитание | change = payment - total; |
| * | Умножение | tax = cost * tax_rate; |
| / | Деление | average = total / count; |

Таблица2. Основные математические операции С++.

Следующая программа использует cout для вывода результата нескольких простых арифметических операций:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "5 + 7 = " << 5 + 7 << endl;
  cout << "12 - 7 = " << 12 - 7 << endl;
  cout << "1.2345 * 2 = " << 1.2345 * 2 << endl;
  cout << "15 / 3 = " << 15 / 3 << endl;
}
```

Посмотрите внимательно на операторы программы. Обратите внимание, что каждое выражение сначала появляется в кавычках, которые обеспечивают вывод символов (например, 5 + 7 =) на экран. Затем программа выводит результат операции и символ новой

строки. Когда вы откомпилируете и запустите эту программу, на вашем экране появится следующий вывод:

```
5 + 7 = 12
12 - 7 = 5
1.2345 * 2 = 2.469
15 / 3 = 5
```

В данном случае программа выполняла арифметические операции, используя только постоянные значения. Следующая программа выполняет арифметические операции, используя переменные:

```
#include <iostream>
using namespace std;
int main()
{
  float cost =15.50; // Стоимость покупки
  float sales_tax = 0.06; // Налог на продажу 6%
  float amount_paid = 20.00; // Деньги покупателя
  float tax, change, total; // Налог на продажу, сдача покупателю и общий счет
  tax = cost * sales_tax;
  total = cost + tax;
  change = amount_paid - total;
  cout << "Стоимость покупки: $" << cost << "\tНалог: $" << tax << "\tОбщий счет: $"
<< total << endl;
  cout << "Сдача покупателю: $" << change << endl;
}
```

В данном случае программа использует только переменные с плавающей точкой. Как видите, программа присваивает значения переменным при объявлении и комментирует назначение каждой (о комментариях будет сказано в конце данного раздела). Далее программа выполняет арифметические операции над переменными для определения налога на продажу, общего счета и сдачи покупателю. Общая стоимость покупки, Налог и Общий счет выводятся в одну строку, но отделяются друг от друга знаками табуляции. Об этом

говорят *специальные символы*, составленные из косой черты и буквы t. Сдача выводится отдельной строкой. Когда вы откомпилируете и запустите эту программу, на вашем экране появится следующий вывод:

Стоимость покупки: $15.5 Налог: $0.93 Общий счет: $16.43

Сдача покупателю: $3.57

Класс представляет собой главное инструментальное средство C++ для объектно-ориентированного программирования. Т.е. конструкция определяемая в C++ программе с привлечением ключевого слова class позволяет реализовывать те идеи и концепции ООП о которых шла речь в первой части курса. Как вы уже знаете класс группирует элементы, соответствующие данным о некотором объекте реального мира, и оперирующие этими данными функции (называемые методами). Группируя данные об объекте и кодируя их в одной переменной-классе, вы упрощаете процесс программирования и увеличиваете возможность повторного использования своего кода.

Класс C++ должен иметь уникальное имя, за которым следует открывающая фигурная скобка, один или несколько элементов (а ими, традиционно, будут поля и методы) и закрывающая фигурная скобка. Т.е. схема объявления класса такая:

```
class class_name
{
  int data_member; // Элемент данных, поле
  void show_member(int); // Функция-элемент, метод
};
```

После определения класса вы можете объявлять *переменные* типа этого класса (называемые объектами, или, для избегания путаницы между объектами реальными и программными - *экземплярами*), как показано ниже:

```
class_name object_one, object_two, object_three;
```

Следующее определение создает класс employee, который содержит определения трех полей и одного метода:

```
class employee
{
  public:
  char name[64];
  long employee_id;
```

```
    float salary;

    void show_employee(void)
    {
      cout << "Имя: " << name << endl;
      cout << "Номер служащего: " << employee_id << endl;
      cout << "Оклад: " << salary << endl;
    };
  };
```

Обратите внимание на использование метки **public** внутри определения класса. Элементы класса могут быть частными (private) или общими (public), от чего зависит, как ваши программы обращаются к элементам класса. В данном случае все элементы являются общими (т.к. в теле нашего класса есть секция public и отсутствует секция private), это означает, что программа (т.е. другие классы отличные от `employee`, классы-потребители) может обращаться к любому элементу класса **employee**, используя оператор точку(.) и мы вскоре увидим как именно этот оператор применяется. Как мы знаем это прямое нарушение принципа инкапсуляции ("черный ящик") и не должно быть допустимо в проектах реальных. Здесь же такой подход применен единственно с целью не переусложнять пример новой конструкции языка - **class**.

Так же следует обратить внимание на определение поля name, которое включает число 64 в квадратных скобках. Если это число со скобками убрать name был бы объявлен как обычный `char`. Т.е. он мог бы содержать ровно один символ(цифру или букву или знак пунктуации). Привлечение квадратных скобок сообщает компилятору о том, что name будет представлен не как единичный символ, а как *массив* таких символов. Массив – это набор данных одного типа, например массив целых чисел или массив символов, как в данном случае. По итогу получается, что каждой переменной типа `char[64]` будет выделено достаточно памяти, что бы разместить 64 символа. В совокупности они как раз достаточны, что бы указать с их помощью имя сотрудника.

Продолжим работу с классом `employee`. После определения класса внутри вашей программы (мы сделали это только что) вы можете объявить объекты (экземпляры) типа этого класса, как показано ниже:

**employee worker, boss, secretary;**

Т.е. совершенно аналогично элементарным типам из таблицы 3.1 - сначала идет тип экземпляра, затем - его уникальное имя или несколько имен через запятую как в данном случае.

Следующая программа создает два объекта (экземпляр) типа employee. Используя оператор точку, программа затем присваивает значения полям этих объектов. Затем программа использует метод *show_employee()* для вывода информации о служащем:

```
#include <iostream.h>
#include <string.h>
using namespace std;

class employee
{
public:
   char name [64];
   long employee_id;
   float salary;

   void show_employee(void)
   {
      cout << "Имя: " << name << endl;
      cout << "Номер служащего: " << employee_id << endl;
      cout << "Оклад: " << salary << endl;
   };
};

int main()
{
   employee worker, boss;
   strcpy(worker.name, "John Doe");
   worker.employee_id = 12345;
   worker.salary = 25000;
```

```
    strcpy(boss.name, "Happy Jamsa");
    boss.employee_id = 101;
    boss.salary = 101101.00;
    worker.show_employee();
    boss.show_employee();
}
```

Как видите, программа объявляет два объекта типа **employee** — **worker** и **boss**, а затем использует оператор точку (.) для присваивания значений элементам и вызова функции **show_employee().** Еще раз вернитесь к заключительной части урока первого "Введение в ООП" и взяв в качестве примера последнюю программу подумайте - хорошо ли вы уяснили разницу между классом (employee) и объектом (worker,boss) от него порождаемым? Ну а итогом работы последней программы будет такой вывод на консоль:

Имя: John Doe

Номер служащего: 12345

Оклад: 25000

Имя: Happy Jamsa

Номер служащего: 101

Оклад: 101101

Наконец, скажем пару слов о *комментариях*. Комментарии помогают человеку читать текст программы; писать их грамотно считается правилом хорошего тона. Комментарии могут характеризовать используемый алгоритм, пояснять назначение тех или иных переменных, разъяснять непонятные места. При компиляции комментарии выкидываются из текста программы поэтому размер получающегося исполняемого модуля не увеличивается.

В С++ есть два типа комментариев. Один использует символы /* для обозначения начала и */ для обозначения конца комментария. Между этими парами символов может находиться любой текст, занимающий одну или несколько строк: вся последовательность между /* и */ считается комментарием. Например:

```
/*
 * Первое знакомство с определением класса в C++.
 * Классы используются как в объектном, так и в
 * объектно-ориентированном программировании.
 */
class Screen {
```

```
    /* Это называется телом класса */
public:
    void home();    /* переместить курсор в позицию 0,0 */
    void refresh (); /* перерисовать экран              */
private:
    /* Классы поддерживают "сокрытие информации"    */
    /* Сокрытие информации ограничивает доступ из   */
    /* программы к внутреннему представлению класса */
    /* (его данным). Для этого используется метка   */
    /* "private:"                                   */
    int height, width;
};
```

Второй тип комментариев – однострочный. Он начинается последовательностью символов // и ограничен концом строки. Часть строки вправо от двух косых черт игнорируется компилятором. Вот пример того же класса Screen с использованием двух строчных комментариев:

```
/*
 * Первое знакомство с определением класса в C++.
 * Классы используются как в объектном, так и в
 * объектно-ориентированном программировании.
 */
class Screen {
    // Это называется телом класса
public:
    void home();     // переместить курсор в позицию 0,0
    void refresh (); // перерисовать экран
private:
    /* Классы поддерживают "сокрытие информации".   */
    /* Сокрытие информации ограничивает доступ из   */
    /* программы к внутреннему представлению класса */
    /* (его данным). Для этого используется метка   */
    /* "private:"                                   */
    int height, width;
```

```
};
```

После того как программа на C++ создана (т.е. мы закончили редактировать файл с ее исходным текстом) следует важный этап - этап компиляции. Как уже было сказано, на этом этапе программа переводится из C++ языка в машинный (двоичный) код. Непосредственно сам процесс сильно варьируется от того используем ли мы среду разработки (и какую) или нет. Но даже если мы ее не используем компиляция будет запускаться и протекать по-разному в зависимости от платформы и используемого компилятора. Дело в том, что нет единого и универсального компилятора C++. Под каждую платформу есть свой компилятор, и не один. Так что в общем случае выбор нужного и подходящего варианта является прерогативой разработчика. Однако общая последовательность шагов этого процесса остается неизменной для всех компиляторов:

- запустить компилятор и указать в качестве входного файла наш файл с исходным текстом
- в случае отсутствия ошибок компиляции "забрать" (т.е. просто найти на локальном диске в выходной директории) выходной файл представляющий исполнимый модуль готовый к запуску на данной платформе

Например, если мы сохранили исходный текст в файле FIRST.CPP а в качестве компилятора выбрали Borland C++ на платформе MS-DOS, то мы бы задали такую команду:

**C:\>BCC FIRST.CPP**

где *BCC* - как раз программа-компилятор. После нажатия на **Enter** мы могли бы проанализировать сообщения компилятора и в случае отсутствия синтаксических ошибок у нас на диске C: был бы размещен файл FIRST.EXE - результат работы компилятора. Запуск этого последнего файла привел к исполнению всей нашей программы.

Однако использование сред разработок освобождает нас от набора консольных команд и предоставляет отчет о ходе и результате процесса компиляции в гораздо более удобном для анализа формате чем строчки в консоли. Это одно из многочисленных преимуществ таких сред вообще и KDevelop как одного из представителей в частности.

## 2.2. Работа со средой разработки ПО KDevelop

Итак, если мы чувствуем себя уверено в создании C++ программ и уже попробовали их писать и компилировать вне среды разработки, самое время обратиться к KDevelop и выяснить - как он может повысить эффективность работы по созданию ПО и облегчить

процесс создания таких программ? Напомню, что приемы работы с рассматриваемым приложением от конкретного языка программирования не зависят, а C++ был выбран нами исключительно в силу его популярности. Поэтому все методики и приемы, обсуждаемые в этом подразделе, будут не менее полезны и применимы при создании Pascal-, Фортран-, Perl- и прочих программ из списка языков приведенного в начале текущего урока.

Есть несколько способов запустить среду разработки KDevelop, однако если мы собираемся разрабатывать программу именно на C++ оптимальным будет путь через главное меню **КДЕ-Прочие-Разработка-KDevelop-Среда** разработки на **C/C++** (Kdevelop: C/C++). При первом запуске нам будет показан **«Совет дня»** и мы можем указать - предъявлять ли его при каждом запуске или нет. Наконец, мы оказываемся в пустом рабочем пространстве (workspace), Рис. 1.

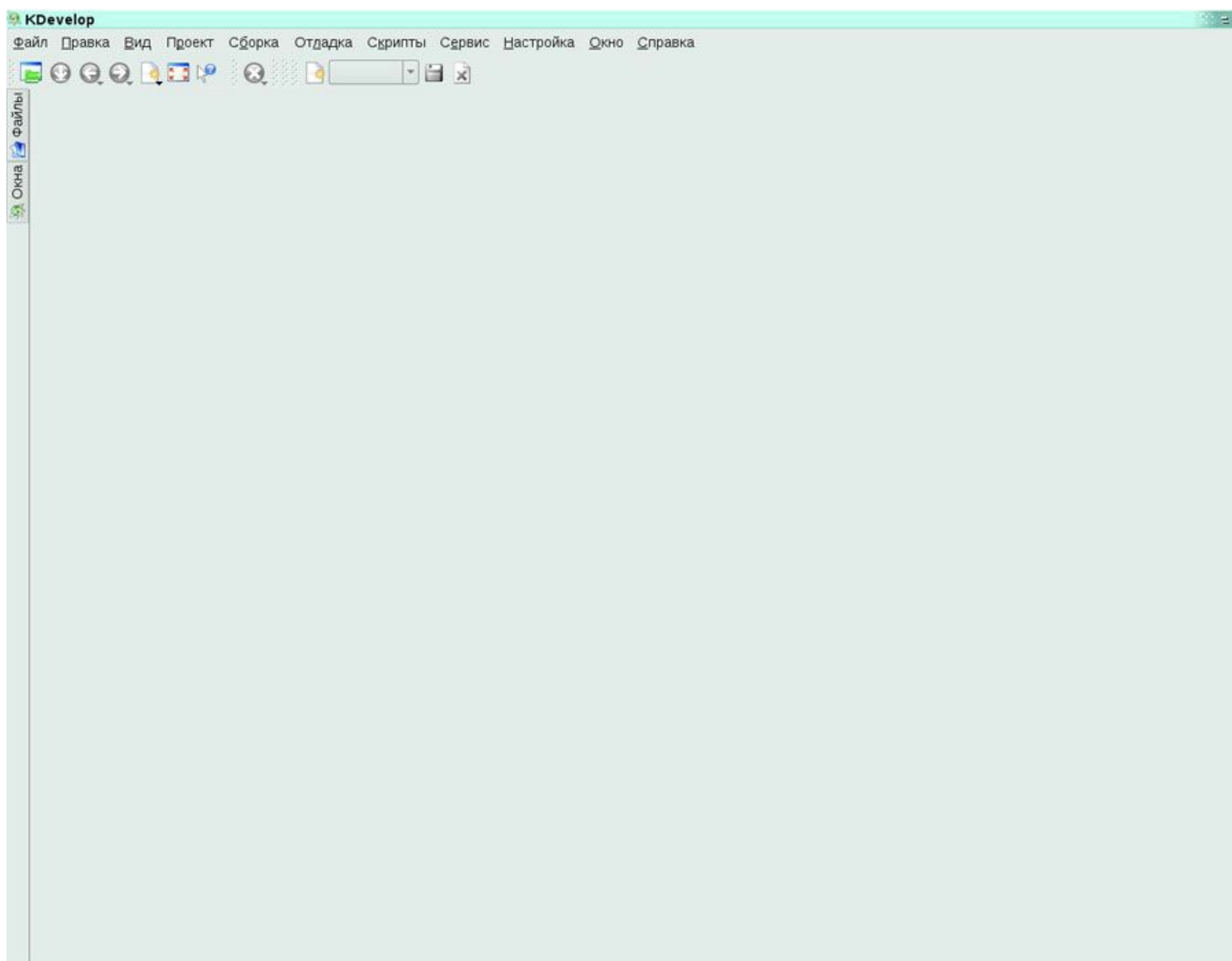


Рис. 1. Пустое рабочее пространство среды KDevelop

Именно в таком состоянии мы будем видеть среду не часто. Обычно после запуска будет автоматически открываться последний *проект* над которым мы работаем.

Именно в пустом рабочем пространстве у нас альтернатив не много - мы можем либо открыть один из существующих проектов, либо начать проект новый. И то, и другое делается через меню **Проект**. Именно тут мы можем вызвать или пункт **«Создать проект…»** или **«Открыть проект…»**. При старте нового проекта мы попадаем в мастер **«Новый проект»** и нам предстоит пройти несколько шагов. Самый важный, пожалуй, шаг первый, где мы выбираем *шаблон* будущего проекта. Давно было замечено, что средний разработчик начиная работать над тем, что по итогу станет новым консольным приложением обычно в качестве отправной точки использует один набор файлов, причем и содержимое этих файлов очень типично. Разумеется, по мере разработки эти файлы очень видоизменяются и становятся совсем не похожи на то, с чего все началось, однако начало-то у всех было одинаковым! То же можно сказать о разработке программ с *графическим интерфейсом пользователя (GUI)*. Все GUI-программисты начинают примерно одинаково, и только по мере разработки у одного получается Калькулятор, а у другого Блокнот. Т.о. можно предположить, что всем разработчикам однотипных (консольных, графических, библиотечных и т.д.) приложений требуется одна и та же (но отличная от приложений другого типа) точка старта. Шаблон и есть эта самая точка, т.е. набор файлов предопределенного содержания. И выбор этот предстоит сделать в первом шаге мастера. Причем выбор является "многоуровневым". Сначала мы выбираем язык программирования (вспомним, что KDevelop является средой мультиязычной), затем, возможно, набор близких по назначению шаблонов и уже в нем - шаблон конкретного типа. При этом в правой части мастера нам покажут внешний вид целевого приложения и краткое описание шаблона, Рис. 2.

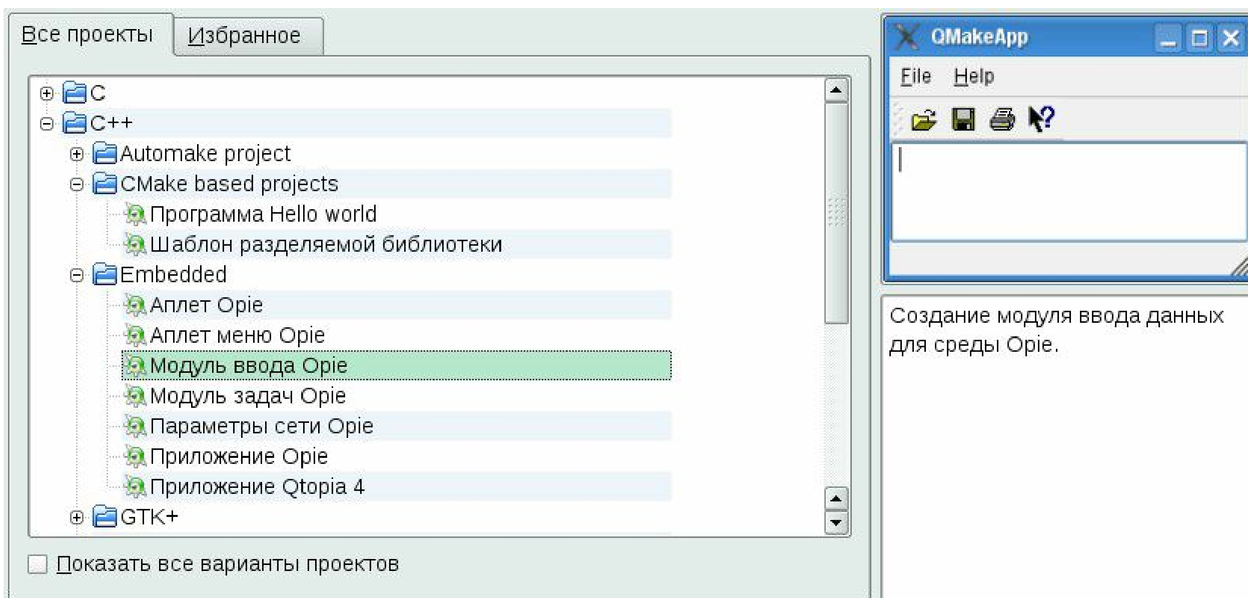


Рис. 2. Шаг первый мастера нового проекта, выбор шаблона.

В нашем курсе мы будем придерживаться шаблона находящегося в папке, разумеется, C++ и имеющего имя **Простая программа Hello world**. Этот проект является прекрасной отправной точкой для создания консольного приложения.

На том же первом шаге мастера мы должны указать вторую важную информацию - где (в каком каталоге) разместятся файлы нашего проекта. Для этого предназначено поле Имя приложения. Здесь мы указываем, фактически, корневую папку проекта и по мере работы над последним она будет наполняться файлами и подкаталогами. Одновременно с этим и все наше приложение будет называться так же. Т.е. если мы введем сюда **MyConsole**, то:

- корневая папка проекта будет названа MyConsole
- исполнимый модуль (то, что получается в результате успешной компиляции проекта) будет иметь то же самое имя, только KDevelop приведет все буквы в его имени к нижнему регистру - myconsole. Именно этот модуль нужно запускать для того что бы увидеть нашу программу в работе. При этом найти на его диске можно будет по пути MyConsole/debug/src.

На вопрос «где на диске искать саму корневую папку MyConsole» отвечает еще одно поле ввода на том же первом шаге - **Расположение**. По умолчанию MyConsole станет подкаталогом домашней папки (/home/admin/, если мы логинились как admin) но мы можем этот путь изменить. В любом случае в поле **«Конечное расположение»** мы увидим полный путь к корневой папке нашего проекта, например /home/admin/MyConsole и можем быть уверены - все, что имеет хоть какое-то отношение к нашему проекту (файлы исходного кода,

заголовочные файлы, финальные исполнимые файлы и т.д.) будет расположено или в этой папке или в одной из ее подпапок.

Наконец мы можем пробежаться по оставшимся шагам мастера и, для целей нашего курса, просто принять значения по умолчанию. Правда с одним исключением - шаг второй не даст нам перейти к шагу, следующему пока в поле **Автор**, не будет введено хоть какое-то значение. А вот все остальные шаги можно действительно "пролистать" нажатием кнопки **Вперед.**

По окончанию мастера будет открыто окно редактора с текстом главного метода (Main). Это и есть отправная точка, предложенная нам выбранным шаблоном.

Со всех сторон (кроме верхней) окна редактора располагаются множество ярлычков (т.н. табов) вспомогательным окон. Их порядка полутора-двух десятков и вопрос назначения каждого оказывается за рамками нашего курса. Скажем лишь об одном из важнейших - Сообщения. Его ярлычок первый в нижнем ряду ярлыков. Именно в этом окне появляются сообщения от компилятора и мы узнаем о допущенных в исходном тексте ошибках. Любое из вспомогательных окон (Сообщения в т.ч.) открывается/закрывается простым щелчком по соответствующему ярлычку.

Обычно даже несложная C++ программа состоит из нескольких файлов исходного кода, причем есть 2 типа таких файлов - т.н. .cpp-файлы и т.н. .h-файлы.. Изначально выбранный нами шаблон **Простая программа Hello world** создает проект содержащий всего один файл по имени **<имя_проекта>.cpp** (например myconsole.cpp). Его наличие и необходимость не обсуждается, т.к. он содержит «точку входа» нашей программы - метод Main. Если же мы хотим добавить новые файлы (а чаще всего мы захотим это сделать) то следует воспользоваться диалогом вызываемом из меню **Проект-Создать класс**. Обычно проекты строятся по схеме 1 класс-1 логический файл, поэтому желание добавить новый файл почти всегда означает желание добавить новый класс и наоборот. А почему файл логический? Потому что в C++ класс принято объявлять и реализовывать в разных файлах (как раз те самые .h/.cpp файлы). Поэтому добавление 1 класса означает добавление двух физических файлов, хотя решают они совместными усилиями одну задачу, объявить и предоставить доступ к новому типу (классу) в нашей программе.

Диалог «Новый класс» имеет 3 вкладки и изрядное количество различных опций и настроек, хотя по минимуму достаточно в поле **Имя** ввести имя нового класса. Тогда после нажатия кнопки OK будут созданы два файла - **<имя_класса>.cpp** и **<имя_класса>.h** - и оба

они будут открыты для редактирования в своих закладках редактора. Переключаться между редактируемыми файлами можно щелчком по заголовкам (ярлыкам) этих закладок, причем «активная» вкладка будет «приподнята» над своими соседями Рис. 3.

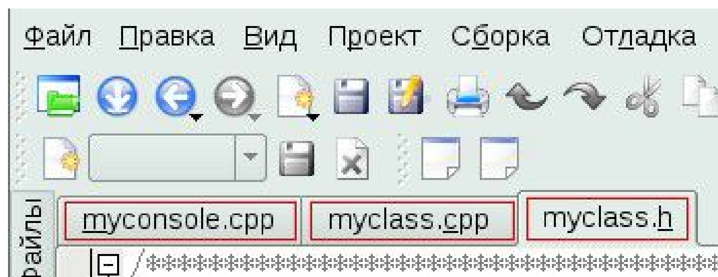


Рис. 3. Можно редактировать несколько файлов исходного кода одновременно. В настоящий момент редактируется файл myclass.h.

Разумеется, редактор встроенный в среду KDevelop отвечает всем функциональным возможностям ожидаемым от серьезного продукта - перемещение, выделение, отступы, настраиваемые поиск/замена и т.д. Однако одна особенность заслуживает особенного упоминания. Допустим мы задекларировали 2 переменных

**int my_var1,my_var2;**

и теперь готовы присвоить значение 5 одной из них. Конечно мы можем полностью набрать

**my_var1=5;**

но можно и воспользоваться обсуждаемой возможностью редактора, а именно: набрать часть (2-3 символа, допустим my_) имени переменной и нажать **Ctrl+J** (или, что эквивалентно, выбрать в меню **Правка-Завершить текст**). Результат последовательности таких действий приведен на Рис. 4.

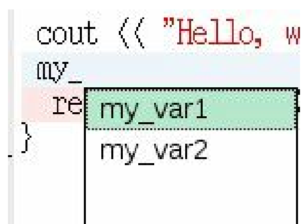


Рис. 4. Одна из многочисленных особенностей редактора исходного кода среды KDevelop - Завершить текст.

Остается лишь выбрать имя нужной переменной, и оно будет вставлено в исходный текст. Причем подсказка дается не только по именам переменных / методов, но и по ключевым словам C++.

Наконец когда мы закончим набор исходных текстов всех файлов составляющих наш проект мы, разумеется, захотим нашу программу собрать (скомпилировать) и опробовать ее в работе. Компиляция проекта запускается через меню **Сборка-Собрать проект** или просто по нажатию **F8**. Для сборки проекта в среде KDevelop последней требуется особый файл - т.н. **Makefile.** Он нужен для автоматизации процесса компиляции нашего проекта и должен быть создан лишь однажды, поэтому при самой первой компиляции KDevelop предложит его создать и лишь после этого перейдет непосредственно к компиляции. Все последующие компиляции будут пользоваться уже готовым Makefile и поэтому будут проходить значительно быстрее чем самая первая.

Как уже упоминалось, все ошибки компиляции будут отображены в окне **Сообщения**. Когда же мы добьемся их отсутствия то у нас будет 2 пути опробовать нашу программу в работе. Во-первых, мы можем открыть Konsole, перейти в каталог, в котором был сгенерирован выходной исполнимый модуль (в нашем разбираемом примере - /home/admin/MyConsole/debug/src) и запустить приложение по его имени:

**./myconsole**

Но есть второй способ значительно более удобный, особенно если мы уверены, что нам нужен «пробный запуск» нашего приложения после которого мы вернемся к его разработке в среде KDevelop. Способ этот сводится к одному из трех вариантов:

- выбору пункта меню **Сборка-Выполнить программу**
- нажатию Shift+F9
- нажатие кнопки на панели, Рис. 5

Рис. 5. Кнопка выполнения программы после ее успешной компиляции.

Результатом любого из этого действия будет открытие консоли «силами» KDevelop и запуск нашей программы в этой консоли. По окончании работы нашей программы эта консоль не «схлопывается» немедленно, а предлагает нам нажать для этого **Enter**. Такая предусмотрительность позволит нам просмотреть вывод нашей программы, даже если мы сами не озаботились созданием подобной «паузы перед окончанием».

Завершим разговор о языке C++ и среде разработки программ на этом языке KDevelop предложением нескольких ссылок на ресурсы посвященным этим двум областям компьютерной науки:

Официальный сайт проекта KDevelop

http://www.kdevelop.org

Учебник по KDevelop

http://www.beginning-kdevelop-programming.co.uk

Учебник по С++ для начинающих

http://programmersclub.ru/main

**3.** Разработка **ПО с использованием Lazarus**

*Lazarus* — свободная среда разработки программного обеспечения. Иными словами тип и предназначение этого программного пакета полностью аналогично разобранному в предыдущем уроке пакету KDevelop. Но есть и существенное отличие. Если KDevelop изначально задумывался как мультиязычный (в смысле поддержки массы языков программирования), то Lazarus поддерживает всего один компилятор и, соответственно, язык именно этого компилятора. В данном случае речь идет о *Free Pascal Compiler (FPC)* - свободно распространяемом компилятора языка Паскаль с открытыми исходными кодами. Этот вариант языка совместим с широко известными вариантами того же языка *Borland Pascal 7* и *Object Pascal* – Delphi, но при этом обладает рядом дополнительных возможностей, например, поддерживает перегрузку операторов. Сам компилятор этого языка (FPC) — кроссплатформенный инструмент, поддерживающий огромное количество платформ. Среди них DOS, Linux, MacOS(X) и Win32. Ну а Lazarus просто удобная графическая оболочка для работы с FPC. Его часто сравнивают с другой популярной интегрированной средой разработки ПО фирмы Borland - *Delphi.* Это сравнение совершенно справедливо, т.к. Lazarus предоставляет разработчику подобное Delphi окружение и, более того, поддерживает преобразование проектов Delphi.

Как и в случае с Kdevelop, для профессиональной работы в среде Lazarus мы должны быть компетентны в двух областях:

- мы должны знать и уметь писать синтаксически корректные программы на языке Паскаль как таковом, причем желательно, что бы это был вариант языка поддерживаемый именно Free Pascal Compiler (FPC)
- мы должны знать Lazarus именно как приложение и уметь пользоваться предлагаемым им функционалом

Рассмотрим эти вопросы последовательно в следующих двух подразделах.

## 3.1. Основы языка программирования Pascal

Снова ограничимся довольно скромной задачей - рассмотреть избранные базовые конструкции языка Pascal в том его варианте, который поддерживается компилятором FPC.

Минимальной программой производящей какую-то активность будет примерно следующая:

```
program Hello;
begin
  writeln ('Hello, world.');
end.
```

Разберем ее построчно.

**program Hello;**

Любая программа на Паскале начинается ключевым словом program вслед за которым следует название данной конкретной программы. Целиком эта строчка называется заголовком.

**begin…end.**

Ограничивают собой раздел операторов и называются операторными скобками. Иными словами играют ту же роль что скобки фигурные в C++ программе. Все что находится между этими двумя зарезервированными словами есть сама программа. В Паскале программа состоит из операторов, причем есть операторы простые и операторы составные. Составной оператор состоит из нескольких простых операторов и ограничен как раз операторными скобками begin и end. Т.о. разбираемая нами минимальная программа состоит из одного оператора. Операторы как и в C++ разделяются точкой с запятой. За телом всей программы должна следовать точка — признак того, что здесь находится конечная точка останова программы. Обратите внимание на "end." с точкой в конце - это именно требование языка.

**writeln ('Hello, world.');**

Как мы уже выяснили - это единственный оператор нашей программы. Он просто выводит указанную строку на консоль. Любая программа на Паскале состоит из множества операторов.

Мы посмотрели описания заголовка и раздела операторов. Без них обычная программа на Паскале в принципе не возможна. В целом ряд программ могут быть созданы только с их помощью. Однако есть и другие составные блоки, используемые при написании программ, и поэтому общую структуру Pascal-программы принято выражать примерно такой схемой:

```
PROGRAM ProgramName;

CONST
  (* Декларация констант *)

TYPE
  (* Декларация типов *)

VAR
  (* Декларация переменных *)

(* Определение процедур и функций *)

BEGIN
  (* Тело программы *)
END.
```

Блоки PROGRAM и BEGIN…END. нам известны. Остальные блоки несут такую нагрузку:

**CONST** - представляет раздел описания констант. Константа в программировании — это способ адресования к данным, изменение которых рассматриваемой программой запрещено. Использование констант, особенно, именованных — мощный инструмент, повышающий надёжность и безошибочность программ. Т.е. если в нашей программе постоянно используется в алгоритмах число 3.14 (приближенное значение широко известной математической константой "пи") то вместо того, что бы из раза в раз набирать эти три цифры с риском ошибиться, поставить десятичную точку не там или просто забыть о ней мы можем писать просто - PI. Или myPI. Или даже primernoPI. Это уж какое символьное имя мы сами присвоим числу 3.14. А выбор этого имени и непосредственное его связывание с конкретным значением как раз происходит в обсуждаемом блоке.

В общем виде раздел описания констант выглядит так:

**Const**

<имя>=<значение>; ... ; <имя>=<значение>;

Конкретный пример этого раздела:

**Const**

Nmin = 0; Nmax = 100; SIMV = 'начало';

Заметим кстати, что если в качестве символьного имени числа «пи» нас устраивает строчка Pi - то ее объявлять не нужно. Компилятор Паскаля обладает «встроенным знанием» о том, что все вхождения строчки Pi должны быть заменены числом 3,14159… . Иными словами это одна из «системных» констант. Но, как было показано, мы можем объявить любое число своих собственных констант.

**TYPE** - в этом разделе описывают нестандартные типы данных, образованные программистом. Что значит "нестандартные"? Таблица 3 приводит описание простейших типов языка Pascal.

| Имя типа | Характеристика типа |
|---|---|
| BYTE | целое число от 0 до 255, занимает одну ячейку памяти (байт). |
| BOOLEAN | логическое значение (байт, заполненный единицами, или нулями), true, или false. |
| WORD | целое число от 0 до 65535, занимает два байта. |
| INTEGER | целое число от –32768 до 32767, занимает два байта. |
| LONGINT | целое число от –2147483648 до 2147483647, занимает четыре байта. |
| REAL | число с дробной частью от 2.9*10-39.до 1.7*1038, может принимать и отрицательные значения, на экран выводится с точностью до 12-го знака после запятой, если результат какой либо операции с REAL меньше, чем 2.9*10-39, он трактуется как ноль. Переменная типа REAL занимает шесть байт. |
| DOUBLE | число с дробной частью от 5.0*10-324.до.1.7*10308, может принимать и отрицательные значения, на экран выводится с точностью до 16-го знака после запятой ,если результат какой либо операции с DOUBLE меньше, чем 5.0*10-324, он трактуется как ноль. Переменная типа DOUBLE занимает восемь байт. |
| CHAR | символ, буква, при отображении на экран выводится тот символ, код которого хранится в выводимой переменной типа CHAR, переменная |

| | занимает один байт. |
|---|---|
| STRING | строка символов, на экран выводится как строка символов, коды которых хранятся в последовательности байт, занимаемой выводимой переменной типа STRING; в памяти занимает от 1 до 256 байт – по количеству символов в строке, плюс один байт, в котором хранится длина самой строки. |

Таблица 3. Простейшие типы, определенные в языке Pascal.

Так вот все типы из последней таблицы описывать в обсуждаемом блоке не нужно. Ими можно просто пользоваться - объявлять переменные такого типа и т.д. Однако в Паскале разрешено введение новых типов определяемых программистом. Их описание происходит как раз в обсуждаемом блоке. В общем виде раздел описания типов выглядит так:

**Type**

<имя>=<тип>;

...

<имя>=<тип>;

Конкретный пример этого раздела:

**Type**

LatBukva = 'a'..'z';

Dni = 1..31;

В этом примере описаны два перечислимых типа: тип **LatBukva**, состоящий из символов латинского алфавита, и тип **Dni**, состоящий из целых зчисленных значений в диапазоне от 1 до 31. В этом же блоке объявляются «строительные кирпичики» ООП - классы.

**VAR** - блок описаний переменных. Каждая встречающаяся в программе переменная должна быть описана. В общем виде раздел описания переменных выглядит так:

**Var**

<имя >, ... ,<имя >:<тип>;

...

<имя >, ... ,<имя >:<тип>;

Иными словами после ключевого слова VAR мы через запятую перечисляем однотипные переменные которые мы планируем использовать в программе, далее через

двоеточие указываем тот или иной тип (причем тип может быть взят как из таблицы 3.3 так и из секции TYPE) и ставим точку с запятой. Повторяем все тоже самое для следующей группы однотипных переменных.

Конкретный пример этого раздела:

**Var**

a, b, c, x, nomer: integer;

y, z, result: real;

В разделе описания переменных обычно описывают также массивы. Пример описания одномерного массива из пятидесяти элементов:

**Var a:array[1..50] of real;**

Или, что тоже самое, но с привлечением константы:

Const  Nmax = 50;

Var  a:array[1..Nmax] of real;

Если предположить, что мы ранее описали блок TYPE таким образом:

**Type**

Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

(т.е. Month есть тип состоящий из 12-ти предопределенных значений), то блок VAR можно объявить так:

**Var**

M: Month;

(т.е. в переменную M можно поместить одно из 12-ти предопределенных значений).

После объявления переменной в данном разделе ей можно присвоить какое-то значение в теле программы (или процедуры, или функции):

**variable_name := expression;**

Т.е. оператор присваивания в Паскале - двоеточие и знак равно. Примеры (предполагаем, что переменная some_real уже объявлена):

some_real := 385.385837;

some_real := 37573.5 * 37593 + 385.8 / 367.1;

Как можно видеть к переменным и числовым константам применимы ожидаемые в любом языке программирования базовые арифметические операторы: +,-,*,/, mod (остаток от деления).

Блок, который в схеме выше условно обозначен комментарием (* Определение процедур и функций *) на самом деле состоит из двух подсекций. Секции маркируемой ключевым словом **PROCEDURE** и секции с ключевым словом **FUNCTION.** Обе эти подсекции описывают подпрограммы, причем именно подпрограммы определяемые программистом. Паскаль предлагает изрядное количество стандартных процедур и функций и их описывать <u>не требуется</u>, ими можно просто пользоваться (вызывать). Примеры стандартных процедур:

- **ClrScr** – очистка экрана, курсор перемещается в верхний левый угол.
- **Delay(t)** – задержка программы на t миллисекунд.
- **GotoXY(x,y)** – перемещение курсора в точку с координатами (x,y).
- **Exit** – выход из текущего блока или из программы.

Нестандартные процедуры и функции должны быть описаны. Их структура (т.е. описание), в принципе, такая же, как и основной программы, однако, прежде всего, следует уяснить разницу между процедурой и функцией, причем эта разница одинаково применима и к их стандартным разновидностям, и к определяемым программистом.

*Процедура* - это обычная подпрограмма в самом широком смысле этого термина. Т.е. это подпрограмма, которая обрабатывает данные, принимает или выводит информацию, меняет или нет каким-либо образом глобальные и локальные переменные, вообще "что-то делает".

*Функция*, это тоже подпрограмма, которая, как и процедура, что-то делает, но, помимо этого, она обязательно возвращает значение, тип которого задается при описании заголовка процедуры. Вызов функции является одним из допустимых операндов выражения, обозначая в нём то значение, которое вычисляет функция, то есть, если подпрограмма с идентификатором **«Function1»** – функция, то можно произвести следующие действия и записать такие операторы:

X:=Function1(a,d,l,n,…);

X:=2*Function1(a,e,…,m,…) - 1/Function1(c,f,…,k,…);

Если бы Function1 была объявлена как процедура, и ее идентификатор был бы изменен на «Proc1» ее можно было бы только вызвать, но НЕ использовать в выражении:

Proc1(a,e,…,m,…);

Каждая объявляемая нами процедура/функция является «программой в программе» (потому их и называют подпрограммами). Вот пример объявления процедуры по имени **Print**. а затем ее вызова в теле главной программы:

```
program Procedures;
procedure Print(s: String; i: Integer);
begin
  Writeln(s);
  Writeln(i);
end;

begin
  Print('Hello',3);
end.
```

А теперь пример объявления и вызова функции **Add**:

```
program Functions;

var
  Answer: Integer;

function Add(i, j:Integer): Integer;
begin
  Add := i + j;
end;

begin
  Answer := Add(1,2); {вызов ф-ии Add}
  Writeln(Answer);
  Writeln(Add(7,11)); {второй вызов ф-ии Add}
end.
```

Итак, мы разобрали основные структурные блоки Паскаль-программы. Отметим, что любой раздел описания (CONST, TYPE, VAR, PROCEDURE, FUNCTION) может

отсутствовать. Т.е. если не планируем объявлять ни одной своей собственной функции, то просто не используем блок FUNCTION. Разделы описаний следуют в произвольном порядке. Каждый раздел описаний может использоваться любое число раз. После всех разделов описания следуют операторные скобки begin…end ограничивающие тело программы.

Нам осталось выяснить вопрос приложимости ООП-принципов к Паскаль-программам. Т.е. как мы можем объявить класс (модель реального объекта) и как из класса получить новый его экземпляр.

Класс в Паскале описывается как новый тип (и поэтому его описание вполне ожидаемо размещается именно в блоке TYPE) по примерно такой схеме:

```
type <имя объекта>=object
  <список имён полей>: <тип полей>;
  ....
  <список имён полей>: <тип полей>;
  <объявление метода>
  ...
  <объявление метода>
end;
```

Вполне ожидаемая структура - тип, инкапсулирующий свои атрибуты (поля) и свой функционал (методы). Почти, как и в С++. Но есть и существенная разница - тела методов в объявлении класса не реализуются. Указываются только их (методов) заголовки, а реализация идет после описания типа:

```
procedure <имя объекта>.<имя метода>(<параметры>)
<тело процедуры>
```

Обратите внимание на оператор точка (.). Аналогично можно описать и метод-функцию.

Приведём реализацию объекта "окно" (объект хорошо всем знакомый по операционной системе Windows). Мы можем считать, что окно на дисплее задаётся координатами левого верхнего угла и размерами по горизонтали и вертикали, а также оно должно иметь флаг видимости (true означает, что окно на экран выведено).

```
Type CWindow=object
  x,y:     integer; {координаты окна}
```

```
  lenx,leny: integer; {размеры окна}
  visible:   boolean; {флаг видимости}
  procedure Init (_x,_y,_lenx,_leny: integer);
  procedure Show;
  function isVisible: boolean;
end;
procedure CWindow.Init (_x,_y,_lenx,_leny: integer);
{Задаёт начальные параметры окна}
begin
  Self.x:=_x;
  Self.y:=_y;
  Self.lenx:=_lenx;
  Self.leny:=_leny;
  visible:=true;
end;

procedure CWindow.show;
{Рисует окно}
  var i,j: integer;
begin
  visible:=true;
  {Остальная часть реализации процедуры опущена}
end;

function CWindow.isVisible: boolean;
begin
  isVisible:=visible;
end;
```

Еще раз обратите внимание на вынос тел процедур и функций (методов) за границы определения класса. Более того, специальные компоновочные *юниты* (вспомогательные файлы кода), где и принято объявлять новые типы, требуют еще разнести объявление и

реализацию методов по разным секциям внутри юнита. Мы поговорим о юнитах и об их двух главных секциях чуть позже.

Создать объект нового типа и воспользоваться заложенным в нем функционалом можно способом практически совпадающим с применением типов встроенных. Сначала объявим переменную (пусть ее имя будет просто Window) целевого типа:

**var Window: CWindow;**

А затем, в теле программы, обращаемся к ее полям и/или методам с привлечением оператора точка:

```
begin
  Window.Init (20,5,40,10);
  Window.Show;
end.
```

При написании проектов уровня хотя бы выше элементарного описание классов принято выносить в отдельный (от файла непосредственно программы) файл. В Паскале такие «вспомогательные» файлы называются *юнитами* (units) или, что тоже самое, модулями. Файл же содержащий текст самой программы так и будет называться - программа, и будет начинаться ключевым словом program, как это было показано в примерах выше. Файлы-юниты будут начинаться другим ключевым словом - unit через пробел от которого будет записано имя этого юнита, а после него обязана находиться точка с запятой. Юниты обязаны состоять из двух частей:

- части декларативной; она начинается с ключевого слова **interface;**
- части реализационной; она начинается с ключевого слова **implementation.**

Часть декларативная содержит объявления всех глобальных объектов модуля (типов, констант, переменных и подпрограмм), которые должны стать доступными основной программе и/или другим модулям. Т.е., например класс со всеми его полями/методами объявляется именно здесь. Однако тела этих методов, их код, будет и должен располагаться не здесь, а во второй секции - реализационной. Собственно это и есть ее предназначение - содержать тела процедур и функций, объявленных в interface части. Все описание юнита завершается словом **end** с непременной точкой после него. Т.о. абсолютно минимальный, но синтаксически корректный юнит по имени, скажем, a имеет вид:

```
unit a;
```

```
interface
implementation
end.
```

Главная программа должна явно выразить свое намерение и желание воспользоваться теми типами, константами, переменными и т.д. что были объявлены и реализованы в том или ином модуле. Делает она это через раздел **uses** в своем заголовке:

```
program testv;
uses a;
Var
Z : Variant;
I : integer;
// остальные разделы описания
begin
// текст программы
end.
```

После такого объявления программа **testv** может использовать все идентификаторы и сущности объявленные /реализованные в модуле a.

Традиционно завершим краткий обзор языка правилами написания *комментария*. FPC поддерживает целых 3 их варианта:

```
(* Комментарий в старом стиле *)
{  Комментарий в стиле Turbo Pascal  }
// Комментарий в стиле Delphi. Все символы
// до конца строки пропускаются компилятором.
```

Первые два ограничиваются парами символов (* или { с одной стороны и *) или } с другой. Оба варианта позволяют писать многострочные комментарии. Комментарий стиля «две черты» полностью аналогичен своему «коллеге» из языка C++. Т.е. он начинается последовательностью символов // и ограничен концом строки.

### 3.2. Работа со средой разработки ПО Lazarus

Среда Lazarus запускается из главного меню: **КДЕ-Прочие-Разработка-Lazarus**. Открывшаяся в результате этого действия среда имеет вид довольно своеобразный и не типичный для современных оконных приложений. У Lazarus нет рабочего пространства (т.е. одного главного окна) в понятиях типичных RAD-сред. Его рабочее пространство состоит из

независимых окон выполняющих какую-либо функцию (редактирование кода, вывод сообщений компилятора, просмотр структуры и состава текущего проекта и т.д.), см. Рис. 6. Независимость этих окон выражается в том, что каждое из них можно свернуть, закрыть, переместить, не оказывая никакого влияния на окна прочие. Стандартная системная комбинация клавиш Alt+Tab (переключение окон) тоже выбирает их и выносит на передний план по одному. «Условно главным» можно считать узкое окно под номером 1 на Рис. 6. Его закрытие приведет к окончанию работы всего приложения Lazarus. Все прочие окна, как уже было сказано, могут открываться/закрываться независимо. «Условно главное» окно содержит лишь меню среды и несколько панелей управления с привычными кнопками. Реальная работа программиста протекает в основном в окнах дополнительных, число которых довольно значительно. Рассмотреть работу и назначение каждого не представляется возможным в силу ограничения времени нашего курса, однако, несомненно, одним из самых востребованных дополнительных окон будет окно редактора исходного кода (номер 2 на Рис. 6). Оно имеет точно такой же «закладочный» дизайн что и редактор предыдущей среды - KDevelop. Ясно, что мы можем редактировать сразу несколько файлов входящих в наш проект. Редактор среды Lazarus умеренно-продвинут и обладает базовым набором функционала ожидаемого в редакторе подобного класса. В частности присутствует аналог функции **Завершить текст** среды KDevelop (см. «Работа со средой разработки ПО KDevelop»), только здесь он называется *identifier completion (завершение идентификатора)* и вызывается сочетанием клавиш **Ctrl+Space** после набора первых 2-3 символов имени переменной.

Так же отметим еще одно вспомогательное окно - **Сообщения** (номер 3 на Рис. 6). Именно здесь мы увидим сообщения компилятора в ходе процесса компиляции нашей программы. В целом трех указанных окон достаточно для написания не сложных консольных приложений..

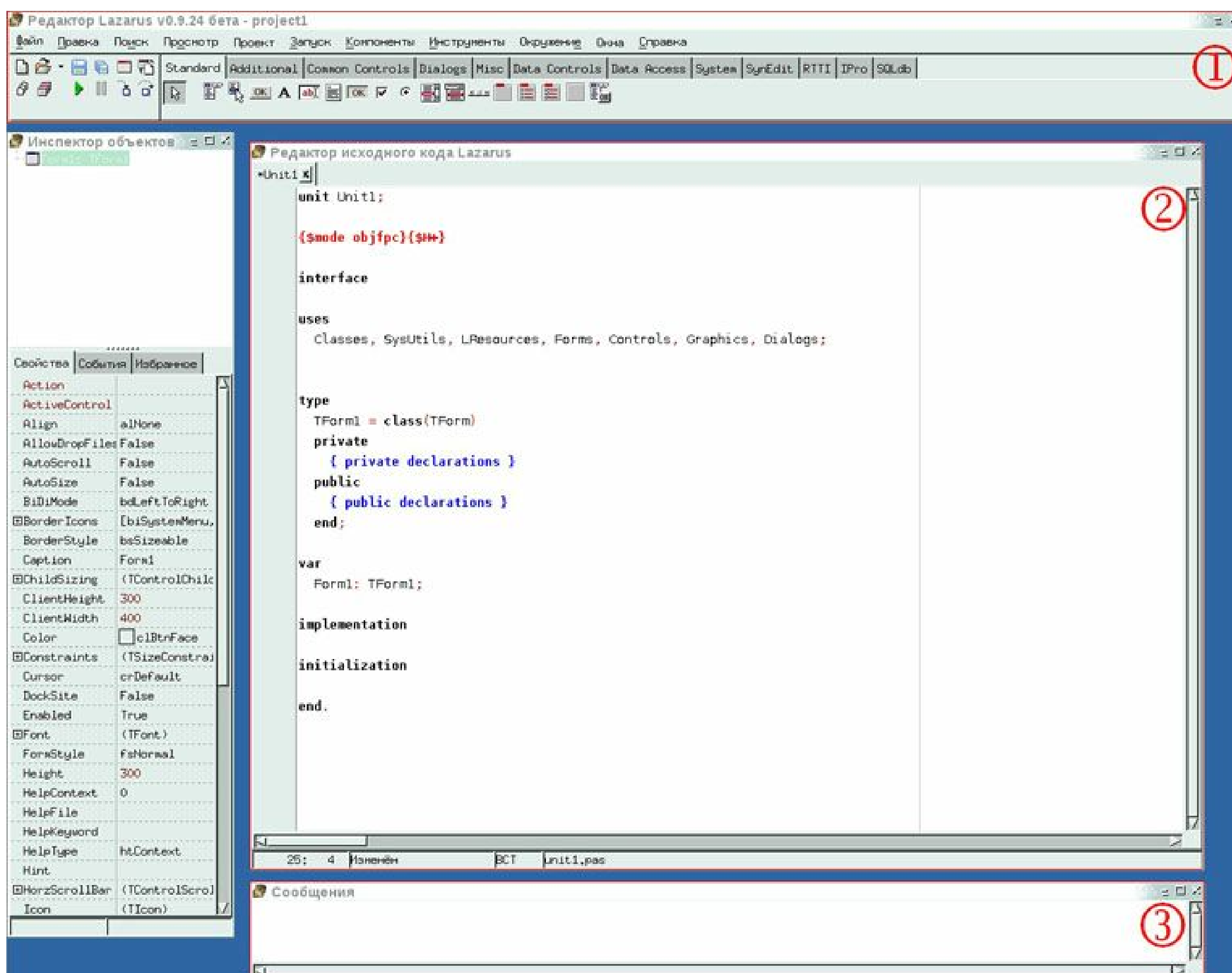


Рис. 6. Рабочее пространство среды Lazarus. Выделены окна: 1 – «главное», 2 - редактор исходного кода, 3 - сообщений компилятора.

Еще одно отличие рабочего пространства Lazarus от такого же в KDevelop - оно «не умеет» быть пустым. Т.е. Lazarus требует что бы постоянно был открыт какой-то проект. Если в последнем сеансе мы работали над проектом **MyProj_01** и при запуске среды он найден на диске - MyProj_01 и открывается как рабочий проект. Если последний проект не найден или это просто первый запуск среды разработки - Lazarus создает проект сам. В качестве *шаблона* (о них см.. «Работа со средой разработки ПО KDevelop») выбирается шаблон приложения с графическим интерфейсом пользователя. Это разумно, т.к. большинство создаваемых приложений будут именно этого типа. Однако, как мы помним, это не тот тип проекта, который нужен нам в изучаемом курсе, и мы решили придерживаться шаблонов приложений консольного типа. Тогда этот созданный средой "на удачу" проект можно закрыть через меню **Проект-Закрыть проект**. Никаких следов на диске он не оставляет если только в открывшемся окне-предупреждении мы укажем пункт «Отбросить

изменения». Если этот стартовый проект вообще не менялся, то и закрывается он безо всяких предупреждений. Появившееся вслед за этим еще одно окно предложит нам выбор из трех опций (Рис. 7). Назначение каждой очевидно.

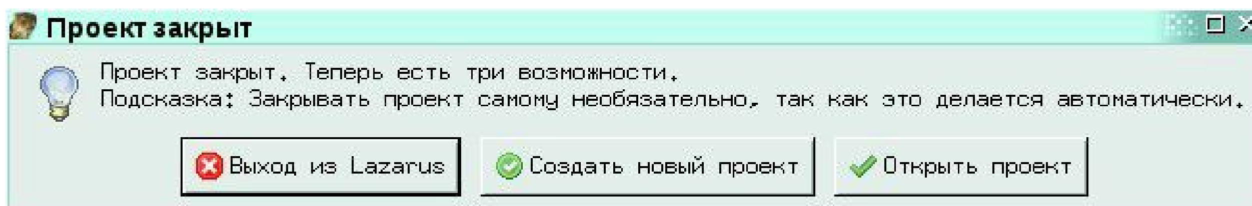


Рис. 7. Окно выбора варианта продолжения работы после закрытия активного проекта.

После выбора в этом последнем окне опции **«Создать новый проект»** среда предложит нам окно диалога **«Создать новый проект»** (Рис. 8). Фактически мы выбираем шаблон для нашего приложения. Список имеющихся далеко не столь внушителен как у среды KDevelop, но достаточен для типовой разработки. Минималистическим шаблоном консольного приложения будет вариант **«Программа пользователя»** (Custom Program). Именно этот шаблон мы используем в практической части занятий.

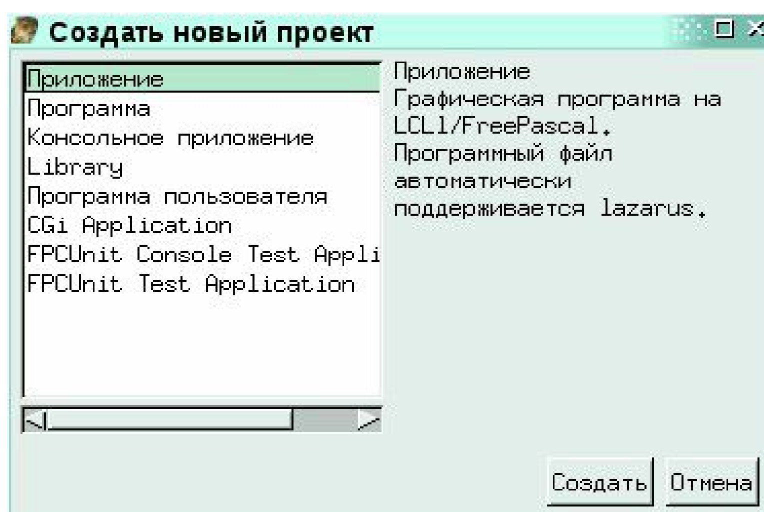


Рис. 8. Окно создания нового проекта. Выбор шаблона будущего приложения.

После выбора этого варианта в редакторе будет открыт исходный текст единственного (пока) файла составляющего наш проект - **project1**. Этот файл содержит код главной программы (которая по умолчанию имеет тоже самое имя - Project1) к редактированию которого и можно приступать. Если программист желает добавить к проекту новые файлы (а в проектах реальных вероятность такого желания близка к 100%) то он может пойти несколькими путями, в зависимости от типа добавляемого файла. Мы воспользуемся путем самым прямолинейным – **Файл-Создать модуль**. Новый модуль (он же *юнит*, unit) добавляется к проекту и открывается в своей закладке редактора. По умолчанию он имеет

имя Unit1, а в текст главной программы добавляется строка **uses Unit1**. Имя нового модуля, конечно, можно изменить прямо в редакторе.

Что касается сохранений, то лучшей стратегией будет сохранение каждого нового модуля (а так же файла главной программы) непосредственно в момент их добавления к проекту и лишь затем переход к их редактированию. Для этого достаточно в любой момент выбрать пункт меню **Файл-Сохранить как…**. Данная команда воздействует на файл в активной закладке редактора. Пока это не сделано файл существует лишь в памяти компьютера и может быть в любой момент утерян. В ответ на выбор указанного пункта меню откроется диалог сохранения активного файла (Рис. 9).

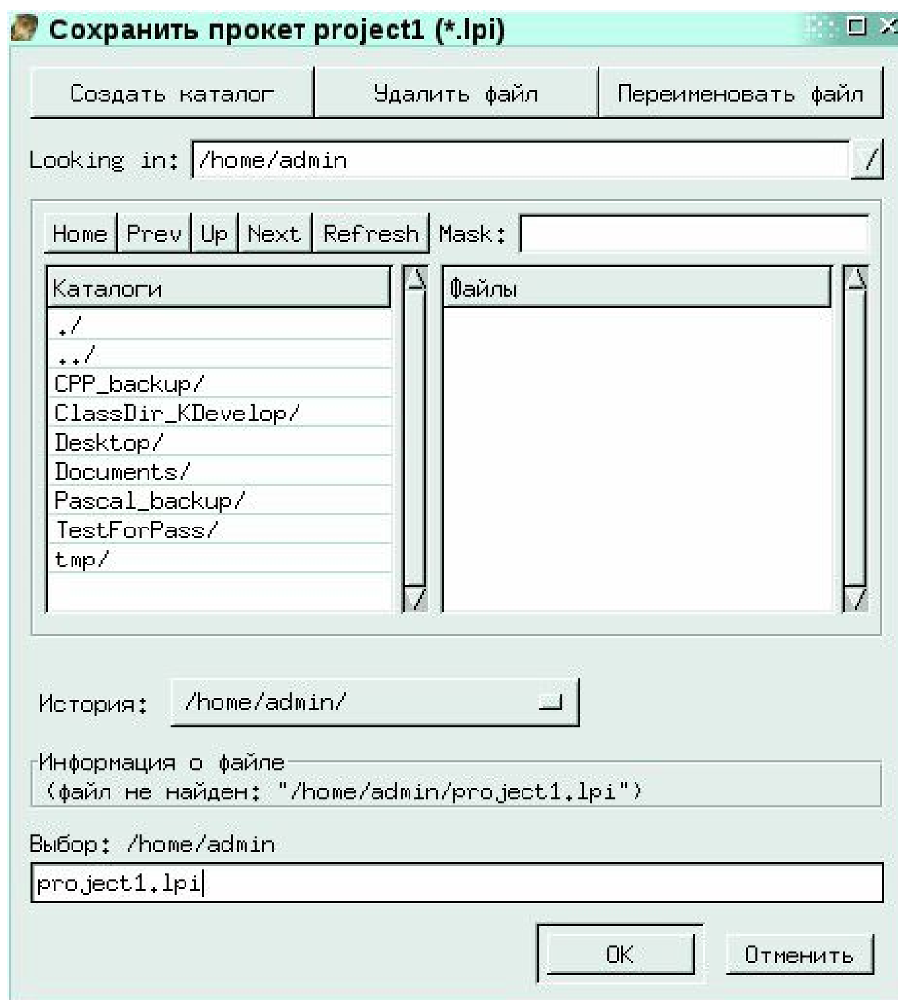


Рис. 9. Окно диалога сохранения файла/проекта.

Здесь надо иметь в виду, что в отличии от среды KDevelop Lazarus самостоятельно не создает т.н. *корневой папки проекта*. А она очень удобна в плане управления файлами, входящими в наш проект. По счастью, из окна сохранения проекта (Рис. 9) мы такую папку

можем создать самостоятельно, для этого предусмотрена кнопка **Создать каталог**. Новый каталог станет подпапкой каталога указанного в поле ***«Looking in»***. После создания желаемой корневой папки проекта остается перейти в нее двойным щелчком по ее имени в списке **Каталоги** и в поле **Выбор: <полный_путь_до_ корневой_папки_проекта>** ввести имя сохраняемого файла. При этом если мы сохраняем файл главной программы, то на диск в указанную папку запишутся 2 файла:

- с расширением **.pas** - непосредственно текст программы;
- с расширением **.lpi** - файл самого проекта

Иными словами имена проекта и главной программы всегда совпадают. При сохранении же нового модуля (unit) на диск пишется одни файл, с расширением .pas и (возможно, но не обязательно) вносятся необходимые изменения в текст файла проекта. Т.о. созданная нами корневая папка проекта будет содержать все файлы необходимые программисту в работе. После успешной компиляции итоговый исполнимый модуль будет помещен сюда же.

Желательно указывать имена всех файлов проекта в нижнем регистре дабы избежать предупреждения о возможных проблемах (Рис. 10). Впрочем, прямо из этого окна можно позволить среде привести имена к требуемому регистру. KDevelop выполняет подобный же «маневр» не интересуясь мнением разработчика.

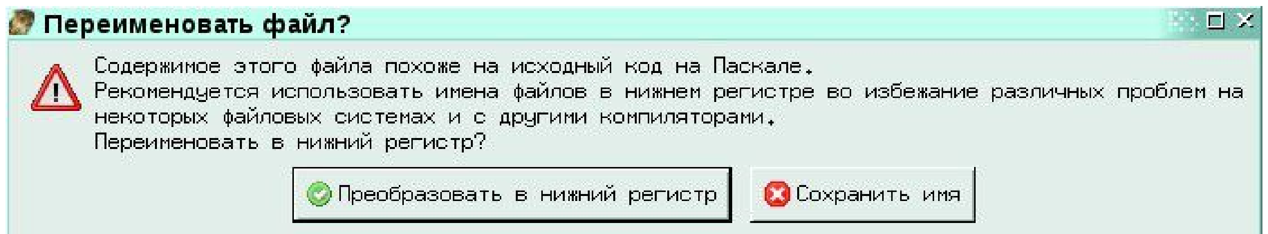


Рис. 10. Окно предупреждения о регистре имени проекта.

Когда все файлы отредактированы должным образом и сохранены на диск можно попробовать собрать программу. Процесс компиляции инициируется выбором пункта меню **Запуск-Собрать** или просто нажатием **Ctrl+F9**. Как уже отмечалось ход и результат этого процесса сообщается в особом окне - Сообщения (номер 3 на Рис. 6). После успешной компиляции возможен тестовый прогон свежей программы прямо из среды. Однако если KDevelop готов к подобному прогону «со старта», консоль, открываемую изнутри Lazarus нужно предварительно настроить. Диалог такой настройки вызывается через меню ***Запуск-Параметры запуска…***. В открывшемся одноименном окне (Рис. 11) требуется:

- разрешить использовать внешнее приложение (консоль) для запуска нашей программы. За это отвечает флажок **Использовать приложение для запуска**;
- указать путь к этому внешнему приложению. По умолчанию таковым выступает *xterm* - стандартный эмулятор терминала для среды X Window System в Unix. остается лишь проверить правильность пути к нему.

После всех этих настроек запуск откомпилированной программы проблем не составляет: ***Запуск-Запуск*** или просто **F9**. Должно открыться окно консоли в котором мы можем осуществлять ввод для нашей программы и/или наблюдать выводимую ей информацию. Точно так же как и в случае с KDevelop эта консоль не "схлопывается" немедленно по окончанию нашей программы, а любезно предлагает нам нажать для этого **Enter**.

Традиционно завершим разговор о языке Pascal и среде разработки программ на этом языке Lazarus предложением нескольких ссылок на ресурсы посвященным этим двум областям компьютерной науки:

Официальный сайт проекта Lazarus

http://lazarus.freepascal.org

Русскоязычный сайт с материалами по компилятору Free Pascal и среде Lazarus

http://freepascal.ru

Lazarus wiki - энциклопедия по Free Pascal Compiler/Lazarus

http://wiki.lazarus.freepascal.org

## 4. Разработка ПО с использованием Gambas

Суть и предназначение третьего из рассматриваемых нами пакетов - *Gambas* - полностью аналогично двум предыдущим. Как и с KDevelop, и с Lazarus мы имеем дело с Rapid Application Development (RAD), т.е. со средой разработки программного обеспечения. По своим возможностям Gambas значительно ближе к Lazarus, т.к. настроен на поддержку всего одного языка. В данном случае речь идет о языке Basic с объектным расширением. Т.е. такой вариант языка начального уровня, который позволяет применять при разработке программ принципы ООП. Этот диалект языка Basic тоже называется Gambas. Т.о. слово Gambas одновременно обозначает И пакет облегчающий написание на " Basic-о подобном" языке, И сам этот язык. Собственно если разобрать слово Gambas то выясняется что это — рекурсивный акроним от англ. Gambas Almost Means BASic, что в дословном русском переводе выглядит как «Gambas Почти Означает Basic», а более привычно русскому уху:

«Gambas — почти Basic». Само слово Gambas с испанского переводится как "креветка", которая изображена на логотипе обсуждаемого пакета.

Gambas задумывался как альтернатива для Microsoft Visual Basic (MVB) разработчиков, решивших перейти на GNU/Linux. Действительно, программист на MVB сможет без труда переключиться на Gambas, но надо иметь в виду, что второй не является клоном первого и определенно не поддерживает запуск программ созданных в MVB.

И снова, как и в случае двух предыдущих сред разработки, для профессиональной работы с Gambas мы должны быть компетентны в двух областях:

- мы должны знать и уметь писать синтаксически корректные программы на языке Basic и лучше, если это будет его вариант с объектным расширением. Разработчики, знающие Basic классический, получают заметное преимущество в этом аспекте обучения;
- мы должны знать Gambas именно как приложение и уметь пользоваться предлагаемым им функционалом.

Рассмотрим эти вопросы последовательно в следующих двух подразделах.

### 4.1. Основы языка программирования BASIC (ОО диалект)

Традиционно начнем обзор базовых конструкций языка с примера программы близкой к минимальной:

```
PUBLIC SUB Main()
     DIM m AS Float
     m=doMultiply()
     PRINT m
END
PRIVATE FUNCTION doMultiply() AS Float
     DIM x AS Float
     DIM y AS Float
     PRINT "Enter x:"
     INPUT x
     PRINT "Enter y:"
     INPUT y
     RETURN x*y
END
```

Что получится в результате запуска этой программы? На консоль будет выведено три строки - первая пригласит пользователя указать число **x**, вторая - **y** и третья просто выведет результат умножения этих двух чисел.

Перейдем теперь к ее построчному разбору.

**PUBLIC SUB Main()**

Декларирует главный метод (точку входа) создаваемой программы. Отсюда начнется ее выполнение. Имя главного метода - **Main**. Вообще Gambas поддерживает декларацию двух разновидностей методов: процедур и функций. Разница между этими сущностями полностью аналогична разнице между ними же в языке Pascal, т.е. функция есть процедура возвращающая значение заданного типа. Поскольку данная декларация не включает ключевого слова **AS** можно сделать вывод, что декларируется именно процедура. Пример декларации функции будет приведен чуть ниже.

**DIM m AS Float**

Любая переменная, планируемая к использованию в пределах класса / процедуры / функции, должна быть предварительно *декларирована*. Переменные могут быть *глобальными* по отношению к классу их декларирующему или *локальными* по отношению к процедуре/функции так же их декларирующих. Ключевое слово DIM необходимо именно для декларации *локальных* переменных. После этого слова идет имя переменной (или несколько, через запятую, если они все будут однотипны), за ним еще одно ключевое слово AS и, наконец, тип переменной. Какие типы можно указывать? Поскольку мы уже определили, что «Gambas=Basic+объектные расширения» то, разумеется, программист может создавать свои собственные типы (классы). Но и, конечно, Gambas предлагает встроенные, элементарные типы, которые описывать не нужно, а можно просто указывать после ключевого слова **AS.**

Таблица 3.4 приводит такие типы, а точнее самые востребованные из них.

| Имя типа | Диапазон значений, комментарий | Занимаемый в памяти размер |
|---|---|---|
| Boolean | true / false | 1 байт |
| Short | -32.768...+32.767 | 2 байта |
| Integer | -2.147.483.648...+2.147.483.647 | 4 байта |
| Single | -1,7014118+38…+1,7014118E+38 | 4 байта |
| Date | Дата/время (например, 07/19/2005 | 8 байт |

| | 02:20:08) сохраненные в виде одного целого числа | |
|---|---|---|
| String | Строчка символов переменной длины | 4 байта (указатель на строку) + память необходимая для хранения всех символов строки |

После декларации локальные переменные могут быть использованы в любом месте только той процедуры/функции, что их декларировала. Переменная m в нашем примере доступна и "видна" только процедуре **Main**.

**m=doMultiply()**

Вызывается функция **doMultiply** и результат ее вызова присваивается только что задекларированной переменной **m**. Вызов процедур/функций в Gambas происходит по самому типичному шаблону всех языков: имя вызываемой процедуры/функции - круглые скобки - в скобках аргументы вызова через запятую. В данном случае функция doMultiply никаких аргументов не принимает и их список в данной строчке вызова пуст. Конечно, саму функцию doMultiply надо тоже декларировать и реализовать и мы вскоре увидим, как это делается.

При декларации аргументов процедур / функций Gambas предлагает вариант их опционального применения при вызове. Осуществляется такой подход с привлечением ключевого слова **OPTIONAL**, например:

```
PRIVATE SUB DoIt(sCommand AS String, OPTIONAL bSaveIt AS Boolean = TRUE)
…
…
END
```

Такая декларация говорит, что процедура **DoIt** может быть вызвана с одним (как минимум, т.к. первый аргумент является обязательным, а не опциональным) или с двумя аргументами. В первом случае (вызов с одним аргументом) DoIt считает, что во втором аргументе было передано значение **TRUE**.

**PRINT m**

Инструкция **PRINT** просто печатает на консоли выражение указанное после нее, в данном случае им станет переменная m содержащая результат умножения.

**END**

Закрывает описание (и завершает выполнение) процедуры/функции. Завершение главной процедуры **Main** так же означает окончание работы программы.

**PRIVATE FUNCTION doMultiply() AS Float**

Декларируем функцию **doMultiply** вызываемую из главной процедуры. То что это декларация именно функции (а не процедуры) говорит наличие ключевого слова AS. Функция обязана вернуть какое-то значение и тип указанный после этого ключевого слова говорит, какого типа это значение будет. В данном случае doMultiply имеет право вернуть любое число из диапазона значений типа **Float** (-8.98E+307…+8.98E+307).

DIM x AS Float

DIM y AS Float

Декларируются две *локальных* (по отношению к функции doMultiply) однотипных переменных **x** и **y**. Могло бы быть для краткости записано и как

```
DIM x,y AS Float
```

PRINT "Enter x:"

Печатает (в консоли) приглашение пользователю.

INPUT x

Принимает ввод пользователя и помещает его в указанную переменную.

PRINT "Enter y:"

INPUT y

Аналогично x, но для переменной **y**

RETURN x*y

Заканчивает выполнение (но не описание!) функции doMultiply путем возврата в вызывающий код значения заданного типа (умножение одного Float на другой даст в результате еще один Float, а именно такой тип наша функция и обязана вернуть).

END

Закрывает описание функции.

Как легко видеть из приведенного примера в Gambas (в отличии от тех же C++ / Pascal) каждая отдельная инструкция не обязана завершаться каким-либо спецсимволом вроде точки с запятой.

К числовым переменным и константам применимы ожидаемые базовые арифметические операторы: +,-,*,/, mod (остаток от деления), ^ (возведение в степень).

Присвоение какого-либо значения переменной осуществляется простым знаком равенства, например:

```
DIM N AS Integer
DIM R AS Integer
N = 3
R = 6
```

С помощью ключевого слова CONST поддерживается декларация констант:

```
PUBLIC CONST MAX_VAL AS Integer = 30
PUBLIC CONST HEADER AS String="# Gambas!!"
```

После подобной декларации отрывок кода

```
DIM b,p,t AS Integer
b=1 + 30
p=7 * 30
t=(b+p) / 30
```

может быть переписан как

```
DIM b,p,t AS Integer
b=1 + MAX_VAL
p=7 * MAX_VAL
t=(b+p) / MAX_VAL
```

что гораздо практичнее, если мы предвидим возможное изменение значения константы **MAX_VAL** и особенно если такие изменения будут частыми.

Нам осталось выяснить вопрос приложимости ООП-принципов к Gambas-программам. Т.е. как мы можем объявить класс (модель реального объекта) и как из класса получить новый его экземпляр. Здесь Gambas предлагает такой подход. Декларация нового класса начинается с создания нового *файла* специального назначения (реализуемого как просто новый файл на HDD содержащий удобную для начала работы с новым классом шаблон-заготовку). Для этого щелкните правой кнопкой мыши в окне проекта (подробнее об этом окне см. следующий раздел, «Работа со средой разработки ПО Gambas») и выберите **New - Class** (Новый - Класс). Вам будет предложено назвать новый файл и класс (т.е. имя файла и становится именем класса). Допустим, мы бы могли назвать этот файл **Person**, имея в виду, что мы приступаем к моделированию объекта реального мира - человека (персоны).

Все место в новом файле отведено для одной задачи - описания полей (атрибутов) и процедур / функций (методов) нового класса. Начать мы бы могли с полей:

```
' Gambas class file
PUBLIC firstname AS String
PUBLIC surname AS String
PUBLIC salary AS Float
```

Как легко догадаться из декларации все три поля будут доступны и из кода нашего класса Person, и из кода любого другого класса / процедуры / функции. Это, конечно, нарушение одного из базовых принципов ООП (инкапсуляции) и мы идем на это только с целью максимально упростить пример для новой темы.

После определения всех полей мы могли бы перейти к декларации и реализации методов. В любом создаваемом классе мы можем указать «особую» процедуру по имени **_new**. Эта процедура нужна для инициализации всех полей нашего класса и вызывается в момент создания его экземпляра. Вспомним, что метод, обладающий подобными характеристиками, называется *конструктором*, и наша _new-процедура представляет именно его:

```
PUBLIC SUB _new(ifirstname AS String, isurname AS String, isalary AS Float)
    firstname = ifirstname
    surname = isurname
    salary = isalary
END
```

И далее мы бы продолжили разработкой любого потребного функционала. В данном случае ограничимся единственной функцией рассчитывающей размер налога исходя из значения поля **salary**:

```
PUBLIC FUNCTION tax() AS Float
    DIM tsalary AS Float
    DIM ttax AS Float
    tsalary = salary
    ttax = 0
    ' сам расчет реализован здесь
    RETURN ttax
END
```

После этого мы могли бы сохранить данный (новый файл) и вернуться в файл с главной процедурой **Main**. В ней мы могли бы написать примерно такой код:

```
' Gambas module file
```

```
PUBLIC SUB Main()
  ' Декларация
  DIM p1 AS Person
  DIM p2 AS Person
  DIM tax1, tax2 AS Float
  ' Инициализация
  p1 = NEW Person("Jane", "Jones", 35000)
  p2 = NEW Person("Fred", "Smith", 23000)
  ' Использование
  tax1 = p1.tax()
  tax2 = p2.tax()
  PRINT tax1
  PRINT tax2
END
```

Здесь мы сначала декларируем 4 переменных: p1 и p2 будут переменными нового типа, а tax1 и tax2 являются переменными элементарного типа Float и нужны нам для сохранения рассчитанного для каждой персоны налога. Далее мы инициализируем две переменные типа Person. В завершении главная процедурой Main использует имеющийся в классе Person функционал расчета налога и запоминает результат в двух своих локальных переменных, содержимое которых и выводится на консоль двумя финальными инструкциями **PRINT**. Для обращения к методам того или иного объекта используется самый популярный для этой цели оператор - точка (.).

И, по традиции, завершим краткий обзор языка правилами написания *комментария*. Тут все крайне не сложно - любая строчка начинающаяся знаком апострофа (') и до конца считается именно им. Уточним, что речь идет о кавычке-апострофе традиционно делящей на стандартной клавиатуре одну клавишу с кавычками двойными. Использование вместо него машинописного обратного апострофа (символ на одной клавише с тильдой, ~) будет расценено компилятором как ошибка. Многострочные комментарии (аналоги /* */ в C++) в Gambas отсутствуют.

### 4.2. Работа со средой разработки ПО Gambas

Как известно все RAD-среды имеют один и тот же базис и основываются на одних и тех же идеях. Везде будут шаблоны, проекты, файлы исходного кода и т.п. Везде будет главное меню с панелями ниже его и масса рабочих окон, открываемых/закрываемых по

необходимости. Всюду можно скомпилировать исходный код, запустить его, отладить и т.п. Рассмотрение двух предыдущих сред разработки - KDevelop и Lazarus - лишь подтверждает этот факт.

Никуда не деться из этой «генеральной колеи» и среде третьей - Gambas. Да, все тот же набор шаблонов для старта нового проекта, тот же «многозакладочный» редактор с характерным функционалом (в частности - на месте функция автозавершения имен идентификаторов, причем вам не нужно для ее вызова нажимать какую-то особую «горячую» клавишу; просто начните набирать имя переменной - спустя 3-4 символа появится окно со списком вариантов, Рис. 11), такая же возможность запустить разрабатываемое приложение (а нас снова будет интересовать их консольная разновидность) прямо из среды и провести анализ выводимой им информации и т.д.

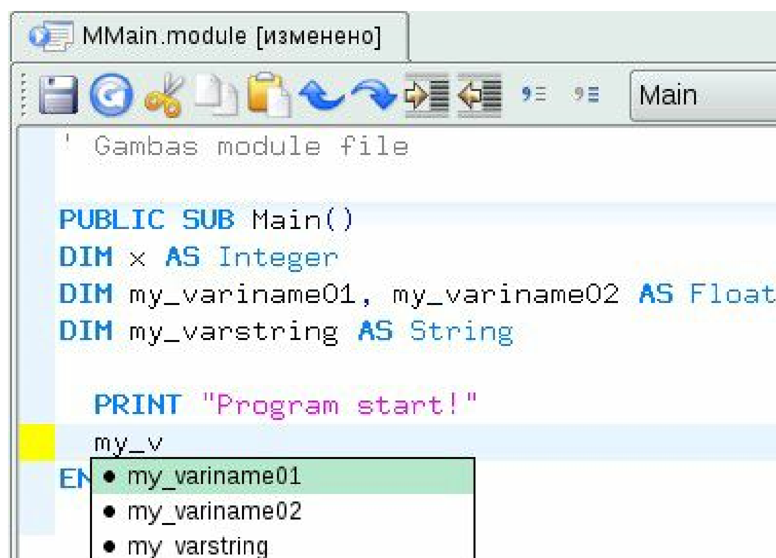


Рис. 11. Функция автозавершения имен в Gambas срабатывает автоматически.

Поскольку все необходимые концепции и типовой функционал сред разработки нам теперь известны и понятны, рассмотрим еще одного их представителя - Gambas - в сжато-конспективном режиме. А именно: применим подход **«Как мне…?»**, который максимально скупо дает ответ на один конкретный вопрос - как произвести то или иной действие - не погружаясь в нюансы. Необходимость того или иного действия и порядок их следования (например тот факт, что сначала надо создать проект и лишь затем переходить к редактированию кода), считаем, нам понятен из опыта работы с предыдущими двумя RAD-средами.

Итак, таблица 3.5 приводит набор типичных **«Как мне…?»** действий требуемых для создания простого консольного приложения на языке BASIC, его компиляции и тест-прогона прямо из среды.

| Как мне…. | Предлагаемое действие |
|---|---|
| **…запустить среду Gambas?** | Выбрать в главном меню: **КДЕ-Прочие-Разработка-Интегрированная среда разработки Gambas (Gambas IDE)** |
| **…начать новый проект?** | Выбрать пункт **«Новый проект»** в окне **«Добро пожаловать в Gambas II»**, появляющееся немедленно после запуска среды |
| **…выбрать консольный тип проекта?** | После выбора пункта **«Новый проект»** (см. предыдущий вопрос) будет предложен список из нескольких типов проектов (шаблонов). Один из них - **Консольное приложение.** |
| **…создать корневую папку проекта?** | На шаге **«2. Директория проекта»** мастера создания нового проекта мы выбираем родительскую папку для будущей корневой папки проекта. Далее, на шаге **«3. Информация о проекте»** мы указываем имя проекта. В результате в родительской папке создается одноименная подпапка. Эта последняя и будет корневой папкой проекта. |
| **…добавить к проекту новые файлы?** | Щелкнуть правой кнопкой мыши в любом месте окна проекта среды (окно 1, Рис. 12). Выбрать из контекстного меню **Новый-Модуль….** Это добавит файл где можно описывать и реализовывать новые функции/подпрограммы, но не новые типы (классы). Или, в том же меню выбрать **Новый-Класс….** Это добавит файл, где можно описывать и реализовывать новые типы (классы). |
| **…перейти к редактированию того или иного файла исходного кода входящего в проект?** | Если файл с кодом уже открыт в окне редактора (окно 2, Рис. 12) то достаточно щелкнуть по закладке с именем файла. Если требуемый файл пока закрыт, необходим будет двойной щелчок по его имени в окне проекта среды (окно 1, Рис. 12). |
| **…сохранить** | Выбрать в главном меню среды **Файл-Сохранить проект** |

| | |
|---|---|
| **внесенные в файлы исходного кода изменения на диск?** | или просто нажать **Ctrl+Alt+S**. Сохраняются сразу все файлы проекта. |
| **…запустить компиляцию проекта?** | Выбрать в главном меню среды ***Проект-Компилировать Все***. «Быстрая клавиша» для этой команды - **Alt+F7**. |
| **…узнать об ошибках в исходном коде и исправить их?** | В отличии от других языков программирования файл исходного кода не воспринимается компилятором как единое целое, а интерпретируется строка-за-строкой, сверху вниз. Первая же строка, выходящая за рамки понимания интерпретатора прерывает весь процесс компиляции и:<br>• закладка с файлом, содержащим ошибочный текст становится активной в окне редактора (окно 2, Рис. 12);<br>• выводится дополнительное окно-сообщение с описанием ошибки; для продолжения работы потребуется нажать OK в этом информационном окне;<br>• после нажатия на OK курсор редактора спозиционируется на строке с ошибкой, можно приступать к ее исправлению и затем к новой попытке компиляции. |
| …собрать исполнимый модуль? | Выбрать в главном меню среды **Проект-Собрать-Executable…**. «Быстрая клавиша» для этой команды - **Ctrl+Alt+M.** |
| …запустить разрабатываемое приложение из среды разработки? | Добиться успешной компиляции проекта а затем выбрать в главном меню среды ***Отладка-Старт***. Тоже самое делает клавиша **F5**. Если проект еще не компилировался с момента последних изменений в файлах исходного кода, то сначала запускается этот процесс, а после его успешного завершения происходит старт приложения. |
| …провести пробный сеанс работы с разрабатываемым | Окно консоли встроено прямо в среду разработки (окно 3, Рис. 12). Весь консольный вывод поступает сюда. Отсюда же может быть осуществлен пользовательский ввод, если |

| консольным приложением не выходя из среды разработки? | разрабатываемое приложение запрограммировано на таковой. |
|---|---|

Таблица 3.5. Типичные действия, совершаемые разработчиком в RAD-среде Gambas.

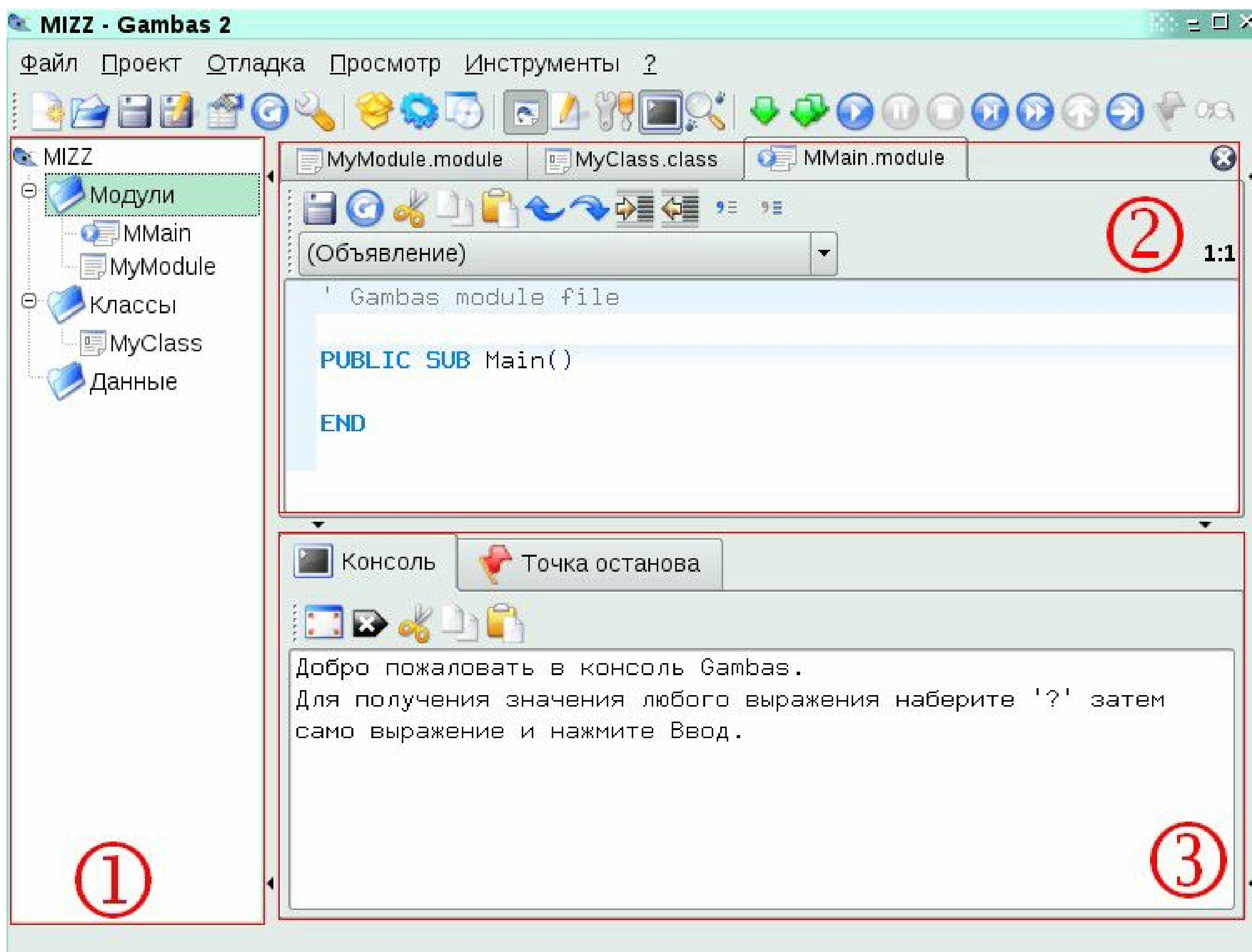


Рис. 12. Окно среды разработки Gambas с консольным проектом, состоящим из трех файлов исходного кода.

Из приведенной таблицы легко видеть, что все шаги действительно крайне характерны для прочих сред разработки и неоднократно выполнялись нами при изучении двух предыдущих пакетов, KDevelop и Lazarus. Недоумение может вызвать лишь наличие действия «компиляция» параллельно с действием «собрать исполнимый модуль». Разве одно не следует из другого? Разве после успешной компиляции исполнимый модуль не создается автоматически? Ответы на эти вопросы утвердительны для подавляющего числа компиляторов, но НЕ для компилятора с языка BASIC (и его ближайшего «родственника»

языка Gambas). Дело в том, что компиляторы BASIC/Gambas являются интерпретаторами. Они попросту выполняют исходный код строка-за-строкой, пока не достигнут конца главной процедуры (Main). Компиляция в данном случае на диск вообще ничего не пишет, а всего лишь удостоверяет тот факт (в случае успешного окончания процесса компиляции, конечно) что интерпретатору ясна каждая строка во всех файлах исходного кода входящих в наш проект. И он готов после запуска все их последовательно исполнить. Действие же «собрать исполнимый модуль» сначала проводит компиляцию (как мы выяснили это эквивалентно простой проверки синтаксиса) а затем этот проверенный исходный текст всех файлов проекта сплавляет в один выходной исполнимый модуль. Среда предлагает образовывать имя этого модуля по шаблону ***<имя_проекта>*.gambas** однако оно может быть любым и у нас будет возможность это имя указать. Однако, и это требуется понимать очень хорошо, между исполнимыми модулями произведенными средами KDevelop/Lazarus и обсуждаемым есть огромная, принципиальная разница:

- исполнимые модули, полученные после компиляции в средах KDevelop/Lazarus являются файлами содержащими «чистый» машинный код. ОС Linux остается просто «скармливать» байт за байтом содержимое таких модулей центральному процессору компьютера. Для исполнения тех же модулей на другом компьютере, не том где они были скомпилированы, требуется просто идентичная (или совместимая) ОС Linux
- исполнимые модули, полученные после процесса, названным нами «собрать исполнимый модуль»,в среде Gambas, машинного кода не содержат вообще. Они по-прежнему содержат исходный код на языке Gambas. Просто этот код агрегирован из всех файлов проекта и слегка упакован для экономии места. Так же это удобно для переноса готового приложения на другой компьютер (перенос одного пусть и большого файла много проще и гораздо лучше управляем, чем перенос пятисот мелких файлов). Тем не менее, принципиально это картину не меняет, код содержащийся в этом итоговом пакете «не понятен» центральному процессору. А понятен он лишь все тому же интерпретатору языка Gambas. Поэтому выход только один - запустить интерпретатор и указать ему данный пакет в виде входного файла. Тогда код на языке Gambas будет «на лету» преобразован интерпретатором в код машинный и, наконец-то, исполнен.

В сухом остатке мы имеем следующий факт: исполнимый модуль произведенный средой Gambas не будет запускаться на другом компьютере даже имеющего идентичную ОС Linux. Для успешного запуска такого модуля (пакета) целевому компьютеру (помимо идентичной/совместимой операционной системы) требуется специальное приложение - интерпретатор языка Gambas. Однако полноценная среда разработки Gambas для запуска готового пакета не требуется, хотя, конечно, и не помешает такому запуску.

Т.о. собрав выходной пакет с именем, скажем *MyProj.gambas* мы можем передать его на другой компьютер, предварительно удостоверившись в наличии интерпретатора языка Gambas на последнем. Приложение-интерпретатор представляет собой утилиту командной строки **gbx2**. Однако интерпретатор, как и положено классическому интерпретатору, умеет воспринимать строки на не упакованном языке Gambas, так как мы их набираем в редакторе. Поэтому gbx2 можно использовать для запуска из консоли файлов исходного кода с расширениями **.module** и **.class**. Это, кстати, еще один вариант распространения нашей программы - просто передать файлы исходного кода. Для их запуска, как легко видеть, среда разработки не нужна, нужна лишь утилита интерпретатора, gbx2. В .gambas-пакете же строки упакованы, поэтому требуется специальная разновидность интерпретатора - так же приложение командной строки **gbr2**. Так вот убедившись в наличии на целевом компьютере этой утилиты и набрав в его консоли команду **gbr2 *MyProj.gambas*** мы, наконец-то, запустим нашу готовую Gambas-программу на исполнение.

Традиционно завершим разговор о языке Gambas и одноименной среде разработки предложением нескольких ссылок на ресурсы посвященным этим двум областям компьютерной науки:

Официальный сайт проекта Gambas

http://gambasrad.org

Краткий учебник по языку Gambas

http://en.wikibooks.org/wiki/Gambas

Официальная документация по языку Gambas

http://gambasdoc.org/help?en