**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Поволжский государственный университет телекоммуникаций и информатики»**

**КОЛЛЕДЖ СВЯЗИ**

**«ОСНОВЫ АЛГОРИТМИЗАЦИИ**

**И ПРОГРАММИРОВАНИЯ»**

**Учебное пособие**

**для специальностей:**

09.02.03 – «Программирование в компьютерных системах»

09.02.04 – «Информационные системы (по отраслям)»

09.02.07 – «Информационные системы и программирование»

Самара,

2019

|  |  |
| --- | --- |
| Рассмотрено на заседании П(Ц)К  «Информационные системы и технологии»  Протокол №\_\_\_\_ от «\_\_\_\_»\_\_\_\_\_\_\_\_\_2019г.  Председатель П(Ц)К\_\_\_\_\_\_\_\_\_\_\_\_\_Шомас Е.А. | УТВЕРЖДАЮ:  Зам. директора по УВР  \_\_\_\_\_\_\_\_\_\_\_\_ /Логвинов А.В./  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2019г. |

Лобачева М.Е.

Основы алгоритмизации и программирвоания. Учебное пособие. – Самара: КС ПГУТИ, 2019. – 160с

**Содержание**

[РАЗДЕЛ 1. ОСНОВНЫЕ ПРИНЦИПЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ 4](#_Toc9338354)

[1.1 Основные понятия алгоритмизации 4](#_Toc9338355)

[1.2 Языки и системы программирования 9](#_Toc9338356)

[1.3 Методы программирования 13](#_Toc9338357)

[РАЗДЕЛ 2. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PASCAL 16](#_Toc9338358)

[2.1 Элементы языка 16](#_Toc9338359)

[2.2 Операторы языка 25](#_Toc9338382)

[2.3 Массивы 38](#_Toc9338398)

[2.4 Строки и множества 47](#_Toc9338399)

[2.5 Подпрограммы 56](#_Toc9338402)

[2.6 Файлы 60](#_Toc9338404)

[2.7 Указатели и динамическая память 68](#_Toc9338405)

[2.8 Модули 79](#_Toc9338411)

[РАЗДЕЛ 3. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++ 85](#_Toc9338428)

[3.1 Лексика языка С++ 85](#_Toc9338429)

[3.2 Операторы языка С++ 91](#_Toc9338449)

[3.3 Массивы 100](#_Toc9338454)

[3.4 Указатели 105](#_Toc9338463)

[3.5 Динамические массивы 107](#_Toc9338467)

[3.6 Строки 111](#_Toc9338469)

[3.7 Функции 113](#_Toc9338470)

[3.8 Структуры и объединения 114](#_Toc9338471)

[РАЗДЕЛ 4. ПРОГРАММИРОВАНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ СРЕДЕ 117](#_Toc9338473)

[4.1 Основные принципы объектно-ориентированного программирования 117](#_Toc9338474)

[4.2 Интегрированная среда программирования Delphi 128](#_Toc9338475)

[4.3 Технология создания приложений Delphi 132](#_Toc9338476)

[4.4 Визуальные компоненты Delphi 138](#_Toc9338478)

[4.5 Работа с базами данных 143](#_Toc9338479)

[4.6 Работа с элементами управления данными 145](#_Toc9338480)

[4.7 Программирование динамически подключаемых библиотек 147](#_Toc9338481)

[4.8 Создание пользовательских компонент. Проектирование компонент 151](#_Toc9338482)

[4.9 Разработка windows приложений 155](#_Toc9338486)

[Литература 160](#_Toc9338487)

**РАЗДЕЛ 1. ОСНОВНЫЕ ПРИНЦИПЫ**

**АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ**

## 1.1 Основные понятия алгоритмизации

Одним из фундаментальных понятий в информатике является понятие алгоритма. Происхождение самого термина «алгоритм» связано с математикой. Это слово происходит от Algorithmi – латинского написания имени Мухаммеда аль-Хорезми (787 – 850) выдающегося математика средневекового Востока. В своей книге "Об индийском счете" он сформулировал правила записи натуральных чисел с помощью арабских цифр и правила действий над ними столбиком. Алгоритм может быть предназначен для выполнения его человеком или автоматическим устройством. В XII в. был выполнен латинский перевод его математического трактата, из которого европейцы узнали о десятичной позиционной системе счисления и правилах арифметики многозначных чисел. Именно эти правила в то время называли алгоритмами.

В настоящее время понятие алгоритма трактуется шире.

***Алгоритмом*** называется точное описание, определяющее последовательность действий исполнителя, направленных на решение поставленной задачи.

Мы постоянно сталкиваемся с этим понятием в различных сферах деятельности (кулинарные книги, инструкции по использованию различных приборов, правила решения математических задач и т.д.). Мир алгоритмов очень разнообразен. Несмотря на это, удается выделить общие свойства, которыми обладает любой алгоритм.

***Свойства алгоритма***

1. Дискретность - разбиение алгоритма на ряд отдельных законченных действий - шагов. Каждое действие должно быть закончено исполнителем алгоритма прежде, чем он приступит к исполнению следующего действия.
2. Понятность - однозначное понимание и исполнение каждого шага алгоритма его исполнителем. Алгоритм должен быть записан на понятном для исполнителя языке.
3. Результативность (или конечность) - обязательное получение результата за конечное число шагов. Каждый шаг (и алгоритм в целом) после своего завершения дает среду, в которой все объекты однозначно определены. Если это по каким-либо причинам невозможно, то алгоритм должен сообщать, что решение задачи не существует. Работа алгоритма должна быть завершена за конечное число шагов.
4. Определенность (детерминированность, точность) - свойство алгоритма, указывающее на то, что каждый шаг алгоритма должен быть строго определен и не допускать различных толкований. Также строго должен быть определен порядок выполнения отдельных шагов.
5. Массовость - применимость алгоритма ко всем задачам рассматриваемого типа, при любых исходных данных

***Формы записи алгоритмов***

На практике наиболее распространены следующие формы представления алгоритмов: словесная, графическая (блок-схемы), псевдокод и программа.

***Словесный способ*** записи алгоритмов представляет собой описание алгоритма на естественном языке.

Пример. Записать алгоритм нахождения наибольшего общего делителя (НОД) двух натуральных чисел.

Алгоритм может быть следующим:

1. задать два числа;

2. если числа равны, то взять любое из них в качестве ответа и остановиться, в противном случае продолжить выполнение алгоритма;

3. определить большее из чисел;

4. заменить большее из чисел разностью большего и меньшего из чисел;

5. повторить алгоритм с шага 2.

Описанный алгоритм применим к любым натуральным числам и должен приводить к решению поставленной задачи.

Словесный способ не имеет широкого распространения по следующим причинам:

- такие описания строго не формализуемы;

- страдают многословностью записей;

- допускают неоднозначность толкования отдельных предписаний.

***Графический способ*** (или блок-схема) - описание структуры алгоритма с помощью последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий. В блок-схеме каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий. В таблице приведены наиболее часто употребляемые символы.

|  |  |  |
| --- | --- | --- |
| **Название символа** | **Обозначение и**  **пример заполнения** | **Пояснение** |
| Процесс | 0007 | Вычислительное действие или последовательность  действий |
| Решение | 0008 | Проверка условий |
| Модификация | 0009 | Начало цикла |
| Предопределенный  процесс | 0010 | Вычисления по подпрограмме, стандартной подпрограмме |
| Ввод-вывод | 0011 | Ввод-вывод в общем виде |
| Пуск-останов | 0012 | Начало, конец алгоритма, вход и выход в подпрограмму |
| Документ | 0013 | Вывод результатов на печать |

Блок **"процесс"** применяется для обозначения действия или последовательности действий, изменяющих значение, форму представления или размещения данных. Для улучшения наглядности схемы несколько отдельных блоков обработки можно объединять в один блок. Представление отдельных операций достаточно свободно.

Блок **"решение"** используется для обозначения переходов управления по условию. В каждом блоке "решение" должны быть указаны вопрос, условие или сравнение, которые он определяет.

Блок **"модификация"** используется для организации циклических конструкций. (Слово модификация означает видоизменение, преобразование). Внутри блока записывается параметр цикла, для которого указываются его начальное значение, граничное условие и шаг изменения значения параметра для каждого повторения.

Блок **"предопределенный процесс"** используется для указания обращений к вспомогательным алгоритмам, существующим автономно в виде некоторых самостоятельных модулей, и для обращений к библиотечным подпрограммам.

***Псевдокод*** - описание структуры алгоритма на естественном, частично формализованном языке, позволяющее выявить основные этапы решения задачи, перед точной его записью на языке программирования. В псевдокоде используются некоторые формальные конструкции и общепринятая математическая символика.

Псевдокод занимает промежуточное место между естественным и формальным языками.

• Он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст.

• В псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя.

Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова, смысл которых определен раз и навсегда. Они выделяются в печатном тексте жирным шрифтом, а в рукописном тексте подчеркиваются.

Единого или формального определения псевдокода не существует, поэтому возможны различные псевдокоды, отличающиеся набором служебных слов и основных (базовых) конструкций.

***Программа*** - описание структуры алгоритма на языке алгоритмического программирования.

**Этапы решения задач на ЭВМ**

Решение задачи разбивается на этапы:

1. Постановка задачи. При постановке задачи выясняется конечная цель и вырабатывается общий подход к решению задачи. Выясняется, сколько решений имеет задача и имеет ли их вообще. Изучаются общие свойства рассматриваемого физического явления или объекта, анализируются возможности данной системы программирования.
2. Формализация (математическая постановка). На этом этапе все объекты задачи описываются на языке математики, выбирается форма хранения данных, составляются все необходимые формулы.
3. Построение алгоритма. На этом этапе метод решения записывается применительно к данной задаче на одном из алгоритмических языков (чаще на графическом).
4. Составление программы на языке программирования. Переводим решение задачи на язык, понятный машине.
5. Отладка и тестирование программы.
6. Проведение расчетов и анализ полученных результатов.

**Основные алгоритмические конструкции**

Элементарные шаги алгоритма можно объединить в следующие алгоритмические конструкции: линейные (последовательные), разветвляющиеся, циклические.

Линейной называют алгоритмическую конструкцию, реализованную в виде последовательности действий (шагов), в которой каждое действие (шаг) алгоритма выполняется ровно один раз.

Разветвляющейся называется алгоритмическая конструкция, обеспечивающая выбор между двумя альтернативами в зависимости от значения входных данных.

**Линейный алгоритм Разветвляющийся алгоритм**

Действие 1

Действие 2

Условие

Да

Нет

Действие 1

Действие 2

Циклической называют алгоритмическую конструкцию, в которой некоторая, идущая подряд группа действий (шагов) алгоритма может выполняться несколько раз, в зависимости от входных данных или условия задачи.

Параметры

цикла

Параметры

цикла

условие

условие

Тело цикла

Тело цикла

нет

да

нет

**Данные и величины**

Совокупность величин, с которыми работает компьютер, принято называть *данными*. По отношению к программе данные делятся на исходные, результаты (окончательные данные) и промежуточные, которые получаются в процессе вычислений.

Любые данные, т.е. константы, переменные, значения функций или выражения характеризуются своими типами. *Тип* определяет множество допустимых значений, которые может принимать тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет формат внутреннего представления данных в памяти ПК. В каждом языке программирования существует своя концепция типов данных. Тем не менее, в любой язык входит минимальный набор основных типов данных, к которому относятся: целый, вещественный, логический и символьный типы.

## 1.2 Языки и системы программирования

Развитие вычислительной техники сопровождается созданием новых и совершенствованием существующих средств общения программистов с ЭВМ - языков программирования (ЯП).

Под ЯП понимают правила представления данных и записи алгоритмов их обработки, которые автоматически выполняются ЭВМ. В более абстрактном виде ЯП является средством создания программных моделей объектов и явлений внешнего мира.

К настоящему времени созданы десятки различных ЯП от самых примитивных до близких к естественному языку человека. Чтобы разобраться во всем многообразии ЯП, нужно знать их классификацию, а также историю создания, эволюцию и тенденции развития.

Чтобы понимать тенденции развития ЯП, нужно знать движущие силы их эволюции. Для выяснения этого вопроса будем рассматривать ЯП с различных точек зрения.

Во-первых, ЯП является инструментом программиста для создания программ. Для создания хороших программ нужны хорошие ЯП. Поэтому одной из движущих сил эволюции ЯП является стремление разработчиков к созданию более совершенных программ.

Во-вторых, процесс разработки программы можно сравнивать с промышленным производством, в котором определяющими факторами являются производительность труда коллектива программистов, себестоимость и качество программной продукции. Создаются различные технологии разработки программ (структурное, модульное, объектно-ориентированное программирование и другие), которые должны поддерживаться ЯП. Поэтому второй движущей силой эволюции ЯП является стремление к повышению эффективности процесса производства программной продукции.

В-третьих, программы можно рассматривать как аналог радиоэлектронных устройств обработки информации, в которых вместо радиодеталей и микросхем используют конструкции ЯП (элементная база программы). Как и электронные устройства, программы могут быть простейшими (уровня детекторного приемника) и очень сложными (уровня автоматической космической станции), при этом уровень инструмента должен соответствовать сложности изделия. Кроме того, человеку удобнее описывать моделируемый объект в терминах предметной области, а не языком цифр. Поэтому третьей движущей силой, ведущей к созданию новых, специализированных, ориентированных на проблемную область и более мощных ЯП, является увеличение разнообразия и повышение сложности задач, решаемых с помощью ЭВМ.

В-четвертых, совершенствование самих ЭВМ приводит к необходимости создания языков, максимально реализующих новые возможности ЭВМ.

В-пятых, программы являются интеллектуальным продуктом, который нужно накапливать и приумножать. Но программы, как и технические изделия, обладают свойством морального старения, одной из причин которого является их зависимость от типа ЭВМ и операционной среды. С моральным старением программ борются путем их модернизации и выпуска новых версий, однако при высокой динамике смены типов ЭВМ и операционных сред разработчики будут только тем и заниматься, что модернизировать старые программы. Поэтому, ЯП должен обеспечивать продолжительный жизненный цикл программы, и стремление к этому является пятой движущей силой развития ЯП.

**История развития языков программирования**

Первые ЭВМ, созданные человеком, имели небольшой набор команд и встроенных типов данных, но позволяли выполнять программы на машинном языке. Машинный язык (МЯ) - единственный язык, понятный ЭВМ. Он реализуется аппаратно: каждую команду выполняет некоторое электронное устройство. Программа на МЯ представляет собой последовательность команд и данных, заданных в цифровом виде. Данные на МЯ представлены числами и символами. Операции являются элементарными, и из них строится вся программа. Ввод программы в цифровом виде производился непосредственно в память с пульта ЭВМ либо с примитивных устройств ввода. Естественно, что процесс программирования был очень трудоемким, разобраться в программе даже автору было довольно сложно, а эффект от применения ЭВМ был довольно низким. Этот этап в развитии ЯП показал, что программирование является сложной проблемой, трудно поддающейся автоматизации, но именно программное обеспечение определяет в конечном счете эффективность применения ЭВМ. Поэтому на всех последующих этапах усилия направлялись на совершенствование интерфейса между программистом и ЭВМ - языка программирования.

Стремление программистов оперировать не цифрами, а символами, привело к созданию мнемонического языка программирования, который называют ассемблером, мнемокодом, автокодом. Этот язык имеет определенный синтаксис записи программ, в котором, в частности, цифровой код операции заменен мнемоническим кодом. Например, команда сложения записывается в виде AR 1,2 и означает сложение (Addition) типа регистр-регистр (Register) для регистров 1 и 2. Теперь программа имеет более удобочитаемую форму, но ее не понимает ЭВМ. Поэтому понадобилось создать специальную программу транслятор, который преобразует программу с языка ассемблера на МЯ. Эта проблема потребовала, в свою очередь, глубоких научных исследований и разработки различных теорий, например теорию формальных языков, которые легли в основу создания трансляторов. Практически любой класс ЭВМ имеет свой язык ассемблера. На сегодняшний день язык ассемблера используется для создания системных программ, использующих специфические аппаратные возможности данного класса ЭВМ.

Следующий этап характеризуется созданием языков высокого уровня (ЯВУ). Эти языки являются универсальными (на них можно создавать любые прикладные программы) и алгоритмически полными, имеют более широкий спектр типов данных и операций, поддерживают технологии программирования. На этих языках создается неисчислимое множество различных прикладных программ. Принципиальными отличиями ЯВУ от языков низкого уровня являются:

· использование переменных;

· возможность записи сложных выражений;

· расширяемость типов данных за счет конструирования новых типов из базовых;

· расширяемость набора операций за счет подключения библиотек подпрограмм;

· слабая зависимость от типа ЭВМ.

С усложнением ЯП усложняются и трансляторы для них. Теперь в набор инструментов программиста, кроме транслятора, входит текстовый редактор для ввода текста программ, отладчик для устранения ошибок, библиотекарь для создания библиотек программных модулей и множество других служебных программ. Все вместе это называется системой программирования. Наиболее яркими представителями ЯВУ являются FORTRAN, PL/1, Pascal, C, Basic, Ada.

Одновременно с развитием универсальных ЯВУ стали развиваться проблемно-ориентированные ЯП, которые решали экономические задачи (COBOL), задачи реального времени (Modula-2, Ada), символьной обработки ( Snobol), моделирования (GPSS, Simula, SmallTalk), численно-аналитические задачи (Analitic) и другие. Эти специализированные языки позволяли более адекватно описывать объекты и явления реального мира, приближая язык программирования к языку специалиста в проблемной области.

Другим направлением развития ЯП является создание языков сверхвысокого уровня (ЯСВУ). На языке высокого уровня программист задает процедуру (алгоритм) получения результата по известным исходным данным, поэтому они называются процедурными ЯП. На ЯСВУ программист задает отношения между объектами в программе, например систему линейных уравнений, и определяет, что нужно найти, но не задает как получить результат. Такие языки еще называют непроцедурными, т.к. сама процедура поиска решения встроена в язык (в его интерпретатор). Такие языки используются, например, для решения задач искусственного интеллекта (Lisp, Prolog) и позволяют моделировать мыслительную деятельность человека в процессе поиска решений. К непроцедурным языкам относят и языки запросов систем управления базами данных (QBE, SQL).

**Классификация языков программирования**

Исходя из вышесказанного, ЯП можно классифицировать по следующим признакам.

1. По степени ориентации на специфические возможности ЭВМ ЯП делятся на:

· машинно-зависимые;

· машинно-независимые.

К машинно-зависимым ЯП относятся машинные языки, ассемблеры и автокоды, которые используются в системном программировании. Программа на машинно-зависимом ЯП может выполняться только на ЭВМ данного типа. Программа на машинно-независимом ЯП после трансляции на машинный язык становится машинно-зависимой. Этот признак ЯП определяет мобильность получаемых программ (возможность переноса на ЭВМ другого типа).

2. По степени детализации алгоритма получения результата ЯП делятся на:

· языки низкого уровня;

· языки высокого уровня;

· языки сверхвысокого уровня.

3. По степени ориентации на решение определенного класса задач:

· проблемно-ориентированные;

· универсальные.

4. По возможности дополнения новыми типами данных и операциями:

· расширяемые;

· нерасширяемые.

5. По возможности управления реальными объектами и процессами:

· языки систем реального времени;

· языки систем условного времени.

6. По способу получения результата:

· процедурные;

· непроцедурные.

7. По типу решаемых задач:

· языки системного программирования;

· языки прикладного программирования.

8. Непроцедурные языки по типу встроенной процедуры поиска решений делятся на:

· реляционные;

· функциональные;

· логические.

Рассмотренная схема классификации позволяет каждому ЯП присвоить один из признаков каждого класса.

**Системы программирования**

Система программирования – это комплекс средств, предназначенный для создания и эксплуатации программ на конкретном языке программирования на ЭВМ определенного типа.

Традиционными средствами разработки программ являются алгоритмические (процедурные) языки программирования. Для создания программы на выбранном языке программирования нужно иметь следующие компоненты:

- Текстовый редактор – это редактор, который позволяет набрать текст программы на языке программирования. Для этой цели можно использовать любые текстовые редакторы, но лучше пользоваться специализированным текстовым редактором.

- Транслятор – это основа систем программирования. Трансляторы языков программирования, т. е. программы, обеспечивающие перевод исходного текста программы на машинный язык (объектный код), бывают двух типов: интерпретаторы и компиляторы.

- Интерпретатор - это транслятор, который обеспечивает последовательный синхронный «перевод» и выполнение каждой строки программы, причем при каждом запуске программы на выполнение вся процедура полностью повторяется. Достоинством интерпретатора является удобство отладки программы в интерактивном режиме, а недостатком - малая скорость работы.

- Компилятор – это транслятор, который исходный текст программы переводит в машинный код. Если в тексте программы нет синтаксических ошибок, то машинный код будет создан. Но это, как правило, не работоспособный код, т.к. в этой программе не хватает подпрограмм стандартных функций, поэтому компилятор выдает промежуточный код, который называется объектным кодом и имеет расширение .obj.

- Редактор связей (сборщик) – это программа, которая объединяет объектные модули отдельных частей программы и добавляет к ним стандартные модули подпрограмм стандартных функций (файлы с расширением .lib), которые содержатся в библиотеках, поставляемых вместе с компилятором, в единую программу, готовую к исполнению, т.е. создает исполнимый .exe файл. Этот файл имеет самостоятельное значение и может работать под управлением той (или такой же) операционной системы, в которой он создан.

В стандартную поставку, как правило, входят текстовый редактор, компилятор, редактор связей (сборщик), библиотеки стандартных функций. Но хорошая интегрированная система обязательно включает в себя специализированный текстовый редактор, в котором выделяются ключевые слова различными цветами и шрифтами. Все этапы создания программы в ней автоматизированы: после того, как исходный текст программы введен, его компиляция и сборка осуществляются одним нажатием клавиши.

В современных интегрированных системах имеется еще один компонент – отладчик. Он позволяет анализировать работу программы по шагам во время ее выполнения, наблюдая, как меняются значения различных переменных.

В последние несколько лет созданы среды быстрого проектирования, в которых программирование, по сути, заменяется проектированием. В проектируемое окно готовые визуальные компоненты перетаскиваются с помощью мыши, затем свойства и поведение компонентов настраивается с помощью редактора. Исходный же текст программы, ответственный за работу этих элементов, генерируется автоматически с помощью среды быстрого проектирования, которая называется RAD-средой. Подобный подход называется визуальным программированием.

## 1.3 Методы программирования

**Структурное программирование**

Основная идея структурного программирования состоит в том, что структура программы должна отражать структуру решаемой задачи, чтобы алгоритм программы был ясно виден из исходного текста. Следовательно, надо разбить программу на последовательность модулей, каждый из которых выполняет одно или несколько действий. Требование к модулю – чтобы его выполнение начиналось с первой команды и заканчивалось последней. Модульность – это основная характеристика структурного программирования. А для этого надо иметь средства для создания программы не только с помощью трех простых операторов, но и с помощью средств более точно отражающих конкретную структуру алгоритма.

С этой целью в программирование введено понятие подпрограммы – набора операторов, выполняющих нужное действие и не зависящих от других частей исходного кода. Программа разбивается на множество подпрограмм, каждая из которых выполняет одно из действий исходного кода. Комбинируя эти блоки, удается сформировать итоговый алгоритм уже не из операторов, а из законченных блоков. Обращаться к блокам надо по названиям, а название несет смысловую нагрузку. Например, Call Summa, означает обращение к подпрограмме с именем Summa, Call - вызов. При структурном подходе к составлению алгоритмов и программ используются три основные типа алгоритмов: условные, циклические алгоритмы и подпрограммы.

**Модульное программирование**

Модульное программирование является развитием и совершенствованием процедурного программирования и библиотек специальных программ. Основная черта модульного программирования - стандартизация интерфейса между отдельными программными единицами. Модуль - это отдельная функционально-законченная программная единица, которая структурно оформляется стандартным образом по отношению к компилятору и по отношению к объединению ее с другими аналогичными единицами и загрузке. Как правило, каждый модуль содержит паспорт, в котором указаны все основные его характеристики: язык программирования, объем, входные и выходные переменные, их формат, ограничения на них, точки входа, параметры настройки и т.д.

Модульное программирование - это искусство разбиения задачи на некоторое число различных модулей, умение широко использовать стандартные модули путем их параметрической настройки, автоматизация сборки готовых модулей из библиотек, банков модулей.

Основные концепции модульного программирования:

-каждый модуль реализует единственную независимую функцию;

-каждый модуль имеет единственную точку входа и выхода;

-размер модуля по возможности должен быть минимизирован;

-каждый модуль может быть разработан и закодирован различными членами бригады программистов и может быть отдельно протестирован;

-вся система построена из модулей;

-модуль не должен давать побочных эффектов;

-каждый модуль не зависит от того, как реализованы другие модули.

**Объектно-ориентированное программирование**

Концепция ООП возникла в середине 80-х годов. Главная ее идея в том, что программное приложение, как и окружающий нас мир, должно состоять из объектов, обладающих собственными свойствами и поведением. ООП объединяет исполняемый код программы и ее данные в объекты, что упрощает создание сложных программных приложений. Например, можно организовать коллективную работу над проектом, где каждый участник создает собственный класс объектов, который становится доступным другим участникам проекта.

При объектно-ориентированном подходе программные задачи распределяются между объектами программы. Объекты обладают определенным набором свойств, методов и способностью реагировать на события (нажатие кнопок мыши, интервалы времени и т.д.). В отличие от процедурного программирования, где порядок выполнения операторов программы определяется порядком их следования и командами управления, в ООП порядок выполнения процедур и функций определяется, прежде всего, событиями.

Чтобы проект можно было считать объектно-ориентированным, объекты должны удовлетворять некоторым требованиям. Этими требованиями являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция — означает, что объекты скрывают детали своей работы. Инкапсуляция позволяет разработчику объекта изменять внутренние принципы его функционирования, не оказывая никакого влияния на пользователя объекта.

Наследование — означает, что новый объект можно определить на основе уже существующих объектов, при этом он будет содержать все свойства и методы родительского. Наследование полезно, когда требуется создать новый объект, обладающий дополнительными свойствами по сравнению со старым.

Полиморфизм — многие объекты могут иметь одноименные методы, которые могут выполнять разные действия для разных объектов. Например, оператор "+" для числовых величин выполняет сложение, а для текстовых — склеивание.

В ООП центральным является понятие класса. Класс – это шаблон, по которому создаются объекты определенного типа. Класс объединяет в себе данные и методы их обработки.

Объекты — это экземпляры определенного класса.

Элементы управления — это объекты, используемые при разработке пользовательского интерфейса.

# РАЗДЕЛ 2. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PASCAL

## 2.1 Элементы языка

# Разработка программ на Паскале включает в себя следующие действия (этапы разработки программы): ввод и редактирование текста программы на языке про­граммирования Паскаль, ее трансляцию, отладку.

# Для выполнения каждого этапа применяются специальные средства: для ввода и редактирования текста используется *редактор текстов*, для трансляции програм­мы - *компилятор*, для построения исполняемого компьютером программного мо­дуля с объединением разрозненных откомпилированных модулей и библиотекой стандартных процедур Паскаля - *компоновщик (linker),* для отладки программ с анализом ее поведения, поиском ошибок, просмотром и изменением содержимого ячеек памяти компьютера- *отладчик (debugger).*

**Основные элементы языка Pascal**

***Алфавит*** языка состоит из множества символов, включающих в себя буквы, цифры и специальные символы.

*Латинские буквы:* от А до Z (прописные) и от а до z (строчные), символ “подчеркивания”, который в языке считается буквой.

*Цифры*: арабские от 0 до 9 и шестнадцатеричные (первые 10 цифр от 0 до 9 - арабские, остальные шесть - латинские буквы: А, B, C, D, E, F).

*Специальные символы:* + - \* / = , . : ; < > [ ] ( ) { } ', $, пары <> <= >= := (\* \*) (. .), пробел (символы (. .) соответствуют символам [ ], несколько пробелов считаются одним).

К спецсимволам относятся также служебные слова - abs, and, array, begin, case, const, dir, do, downto, else, end, for, function, goto, if, int, label, mod, not, of, or, procedure, program, repeat, shr, then, to, type, var, while, with и др. Смысл зарезервированных слов фиксирован строго. При этом набор зарезервированных слов может меняться от версии к версии.

***Идентификатор*** - это последовательность букв, цифр и знаков подчеркивания, начинающихся не с цифры. Под идентификатором мы будем понимать ячейку памяти ЭВМ, которая имеет свое имя и в которой хранится информация. В Паскале строчные и прописные буквы в идентификаторах и служебных словах не различаются. Идентификаторы могут иметь произвольную длину, но значащими являются только первые 63 символа. Хорошим стилем является осмысленный выбор имени идентификатора. Зарезервированные слова не могут использоваться в качестве идентификаторов.

***Комментарии*** заключаются либо в фигурные скобки **{** комментарий 1 **}**, либо в символы **(\*** комментарий 2 **\*)** и могут занимать любое количество строк. Последовательность из трех символов **(\*)** начинает комментарий до конца строки. Текст комментария игнорируется при компиляции, если это не директивы компилятора, которые имеют вид **{$ }**.

# Типы данных

Концепция типов данных является одной из центральных в любом языке программирования. Паскаль характеризуется разветвлённой структурой типов данных.

## Целые

## Порядковые

## Простые типы

# ТИПЫ

## Объекты

Процедурные

## Массивы

Структурированные

## Файлы

## Записи

Множества

## Строки

## Указатели

### Тип-диапазон

### Логический

Символьный

Перечисляемый

## Вещественные

## *ПРОСТЫЕ ТИПЫ*

К простым типам относятся порядковые и вещественные типы.

*Порядковые типы* отличаются тем, что каждый из них имеет конечное число возможных значений. Эти значения можно определённым образом упорядочить (отсюда – название типов) и, следовательно, с каждым из них можно сопоставить некоторое целое число – порядковый номер значения.

*Вещественные типы* тоже имеют конечное число значений, которое определяется форматом внутреннего представления вещественного числа. Однако количество возможных значений вещественных типов настолько велико, что сопоставить с каждым из них целое число (его номер) не представляется возможным.

### *Порядковые типы*

К порядковым типам относятся целые, логический, символьный, перечисляемый и тип-диапазон.

*Целые типы.* Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два или четыре байта

|  |  |  |
| --- | --- | --- |
| ЦЕЛЫЕ ТИПЫ | | |
| Название | Длина, байт | Диапазон значений |
| Byte | 1 | 0…255 |
| ShortInt | 1 | -128…+127 |
| Word | 2 | 0…65535 |
| Integer | 2 | -32768…+32767 |
| LongInt | 4 | -2 147 483 648…+2 147 483 647 |

При использовании процедур и функций с целочисленными параметрами следует руководствоваться «вложенностью» типов, т.е. везде, где может использоваться Word, допускается использование Byte (но не наоборот), в LongInt “входит” Integer, который, в свою очередь, включает в себя ShortInt.

При действии с целыми числами тип результата будет соответствовать типу операндов, если операнды относятся к различным целым типам,- типу того операнда, который имеет максимальную мощность (максимальный диапазон значений).

*Логический тип.* Значениями логического типа может быть одна из предварительно объявленных констант FALSE (ложь) или TRUE (истина).

Поскольку логический тип относится к порядковым типам, его можно использовать в операторе счётного типа.

|  |  |  |
| --- | --- | --- |
| ЛОГИЧЕСКИЕ ТИПЫ | | |
| Название | Длина, байт | Значение |
| Boolean | 1 | False, True |
| Bytebool | 1 | False, True |
| Wordbool | 2 | False, True |
| Longbool | 4 | False, True |

*Символьный тип.* CHAR – занимает 1 байт. Значением символьного типа является множество всех символов ПК. Каждому символу присваивается целое число в диапазоне 0…255. Это число служит кодом внутреннего представления символа.

*Перечисляемый тип.* Перечисляемый тип задаётся перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамлённом круглыми скобками, например,

Type

Colors = (red, white, blue);

Применение перечисляемых типов делает программы нагляднее.

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение списке получает порядковый номер 0, второе – 1 и т.д. максимальная мощность перечисляемого типа составляет 65536 значений, поэтому фактически перечисляемый тип задаёт некоторое подмножество целого типа WORD и может рассматриваться как компактное объявление сразу группы целочисленных констант со значениями 0,1 и т.д.

Использование перечисляемых типов повышает надёжность программы, благодаря возможности контроля тех значений, которые получают соответствующие переменные.

*Тип-диапазон.* Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона.

Тип-диапазон задаётся границами своих значений внутри базового типа:

<мин.знач.>..<макс.знач.>, где

<мин.знач.> - минимальное значение типа-диапазона, <макс.знач.> - максимальное значение.

Например,

Type

Digit = ‘0’ ..’9’;

Dig2 = 48 .. 57;

##### При определении типа-диапазона нужно руководствоваться следующими правилами:

1. два символа «..» рассматриваются как один символ, поэтому между ними недопустимы пробелы.
2. левая граница диапазона не должна превышать его правую границу.

Тип-диапазон наследует все свойства базового типа, но с ограничениями, связанными с его меньшей мощностью.

###### **Вещественные типы**

В отличие от порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляется в ПК абсолютно точно, значения вещественных типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

|  |  |  |  |
| --- | --- | --- | --- |
| Длина, байт | Название | Количество значащих цифр | Диапазон десятичного порядка |
| 6 | Real | 11…12 | -39…+38 |
| 4 | Single | 7…8 | -45…+38 |
| 8 | Double | 15…16 | -324…+308 |
| 10 | Extended | 19…20 | -4951…+4932 |
| 8 | comp | 19…20 | -2\*1063+1…+2\*1063-1 |

# *СТРУКТУРИРОВАННЫЕ ТИПЫ*

Любой из структурированных типов характеризуется множественностью образующих этот тип элементов, т.е. переменная или константа структурированного типа всегда имеет несколько компонентов. Каждый компонент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить о возможной вложености типов. В Паскале допускается произвольная глубина вложености типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 65520 байт.

***Массивы***в Паскале во многом схожи с аналогичными типами данных в других языках программирования. Отличительная особенность массивов заключается в том, что все их компоненты суть данные одного типа (возможно структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием порядкового номера.

***Запись***– это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от массива, компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

***Множества*** *–* это набор однотипных логических связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Паскалем. Количество элементов, входящих в множество, может меняться в пределах от 0до 256 (множество, не содержащее элементов, называется пустым).именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Для задания множества используется так называемый конструктор множества: список спецификаций элементов множества, отделяемых друг от друга запятыми; список обрамляется квадратными скобками. Спецификациями элементов могут быть константы или выражения базового типа, а также – тип-диапазон того же базового типа.

***Строки.*** Тип STRING (строка) в Турбо Паскале широко используется для обработки текстов. Он во многом похож на одномерный массив символов ARRAY [0..N] OF CHAR, однако, в отличие от последнего, количество символов в строке – переменной может меняться от 0 до N, где N – максимальное количество символов в строке. Значение N определяется объявлением типа STRING[N] N и может быть любой константой порядкового типа, но не больше 255. Турбо Паскаль разрешает не указывать N, в том случае длина строки принимается максимально возможной, а именно N=255.

***Файлы.***Под файломпонимается либо именованная область внешней памяти ПК, либо логическое устройство – потенциальный источник или приёмник информации.

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что даёт возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Турбо Паскаля, кроме файлов. В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только ёмкостью устройств внешней памяти.

***Процедурные типы.*** Основное назначение этих типов — дать программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям.

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы.

***Объект*** — это структура, состоящая из фиксированного числа компонент. Каждая компонента — это или поле, которое содержит данные определенного типа, или метод, который производит операции над объектом. Тип объект может наследовать компоненты от другого типа объекта.

***Указатель*** (ссылочный тип) определяет множество значений, которые указывают на динамические переменные определенного типа, называемого базовым типом. Переменная с типом указатель содержит адрес динамической переменной в памяти.

Если базовый тип является еще не описанным идентификатором, то он должен быть описан в той же самой части описания типов, что и тип указатель.

**Переменные и константы**

***Константа*** - это величина, которая в ходе выполнения программы принимает одно значение. Ее значение устанавливается еще до того, как программа начнет выполняться, а в ходе ее запуска сохраняет свое значение неизменной на всем протяжении работы программы.

В качестве констант могут использоваться целые, вещественные, шестнадцатеричные числа, логические константы, символы, строки символов, множества:

а) целые числа записываются без дробной части со знаком или без него;

б) вещественные числа записываются со знаком или без него, с фиксированной или плавающей точкой (например, +3.14 или -19е-5);

в) логическая константа - либо false либо true (ложь или истина);

г) символьная константа - любой символ, заключенный в апострофы (например, '<>', 'Y');

д) строковая константа - любая последовательность символов, заключенная в апострофы (например, 'это моя строка', ' "').

В Паскале существуют типизированные константы, представляющие собой переменные простых или составных типов (кроме файлов) с начальным значением.

***Переменными*** называются параметры программы, которые могут менять свое значение в процессе ее выполнения. Все без исключения переменные должны быть описаны в разделе программы, начинающемся со слова **VAR**. Затем следуют конструкции вида:

список идентификаторов переменных**:** тип1;

список идентификаторов переменных**:** тип2;

В списке имена переменных перечисляются через запятую. Кроме базовых типов Турбо Паскаля здесь можно использовать свои типы (описанные ранее в разделе **Type**).

Арифметические операции, функции, выражения

В Паскале определены следующие операции:

* Унарные: not (отрицание);
* Мультипликативные: \* (умножение), / (деление), div (деление нацело), mod (остаток от целочисленного деления), and (логическое “И”);
* Аддитивные: + (сложение), - (вычитание), or (логическое “ИЛИ”);
* Отношения: = (равно), <> (не равно), < (меньше), > (больше), <= (меньше или равно), >= (больше или равно).

*Примечание*: в Паскале нет операции возведения в степень. Вместо этого для возведения в целую степень можно использовать операцию умножения, а в дробную – выражение, использующее операцию логарифмирования и потенцирования: xy=ey\*lnx, что на языке Паскаль соответствует выражению exp(y\*ln(x)) с использованием двух стандартных функций exp(<выражение>) и ln(<выражение>).

***Стандартные функции Паскаля***

Для вычисления наиболее распространенных математических функций в Турбо-Паскале предусмотрены следующие стандартные функции:

|  |  |
| --- | --- |
| **Обращение** | **Функция** |
| Pi | Число π |
| Sin(x) | Синус х (х в радианах) |
| Cos(x) | Косинус х (х в радианах) |
| Arctan(x) | Арктангенс х (х в радианах) |
| Exp(x) | еx – экспонента |
| Ln(x) | Натуральный логарифм x |
| Sqr(x) | Квадрат числа х |
| Sqrt(x) | Квадратный корень из числа х |
| Abs(x) | Абсолютная величина числа х |
| Trunc(x) | Ближайшее целое, не превышающее х по модулю |
| Frac(x) | Дробная часть числа х |
| Int(x) | Целая часть числа х |
| Round(x) | Округление числа до ближайшего целого |
| Succ(x) | Определение следующего по порядку элемента из списка |
| Pred(x) | Определение предыдущего элемента из списка |
| Random | Псевдослучайное число в интервале [0; 1) |
| Random(x) | Псевдослучайное число в интервале [0; х) |

.

***Выражение*** - это единица языка, которая определяет способ вычисления некоторого значения. Выражения формируются из констант, переменных, функций, знаков операций и круглых скобок по определенным синтаксическим правилам.

В Паскале имеется большое количество встроенных функций для работы с данными каждого типа. Имена (указатели) этих функций с аргументом в круглых скобках могут также встречаться в выражениях.

Круглые скобки используются для изменения порядка вычисления частей выражения. Выражения без скобок вычисляются в порядке, соответствующем приоритету операций. Приоритеты расставлены таким образом:

* вычисления в круглых скобках;
* вычисление значений функций;
* унарные операции ( **not**,+,- );
* операции типа умножения ( \*,/,**div**,**mod**,**and** );
* операции типа сложения ( +,-, **or**, **xor** );
* операции отношения ( =, <>, <, >, <=, >= ).

В логическом выражении 2<=4 **and** 5>3 Паскаль выдаст ошибку, поскольку операция **and** будет выполнена раньше операций сравнения. Верная запись - (2<=4) **and** (5>3).

Структура программы

Программа, написанная на Паскале, состоит из заголовка и тела (блока), в конце которого следует точка – признак конца программы. В свою очередь, блок содержит разделы описаний и раздел операторов:

**Program** <имя программы>; { Заголовок программы }

**Uses** … ; { Подключение модулей }

**Label** ... ; { Раздел объявления меток }

**Const** ... ; { Раздел объявления констант }

**Type** ... ; { Раздел объявления новых типов }

**Var** ... ; { Раздел объявления переменных }

**Procedure** ... ; { Описание своих процедур }

**Function** ... ; { Описание своих функций }

**Begin** { начало основной программы }

<раздел операторов>

**End.**

***Заголовок программы***, начинается с зарезервированного слова program и имени, состоящего из букв латинского алфавита и цифр. Турбо-Паскаль позволяет опускать объявление program, поэтому данная строка является необязательной. Однако использование слова program является признаком хорошего стиля программирования.

***Раздел описаний*** содержит описание каждого элемента информации, над которыми эти действия будут производиться. Эта часть должна предшествовать исполнительной части.

1. В разделе подключения модулей объявляются имена всех внешних модулей, ресурсы которых используются в программе.
2. Определение меток, начинающееся с зарезервированного слова Label и перечисления через запятую используемых в программе меток, представляющих собой любые цифры от 1 до 9999 или символьные имена.
3. Раздел объявления констант начинается с зарезервированного слова Const и содержит перечисления используемых в программе констант с присвоенными им именами, знаками = и их значениями и отделяемыми друг от друга точкой с запятой. Типизированные константы также объявляются после слова const, но имена через двоеточие связываются с типом, и лишь потом ставится знак равенства, и указываются значения.
4. Объявление новых типов начинается с зарезервированного слова Type и состоит из перечисления через точку с запятой имен типов, вводимых пользователем, с пояснением через знак равенства (=), от каких базовых типов и как они происходят.
5. Раздел описания переменных начинается с зарезервированного слова Var и содержит перечисления используемых в программе переменных с указанием их типа. При этом имена переменных одного и того же типа, могут перечисляться через запятую, с указанием в конце записи после двоеточия их типа, а имена переменных разных типов описываются отдельно и отделяются друг от друга точкой с запятой. В языке Турбо-Паскаль все используемые в программе переменные должны быть объявлены.
6. Раздел описания процедур и функций, начинается с зарезервированного слова Procedure и имени процедуры или с зарезервированного слова Function и имени функции. В раздел описания каждой процедуры и функции могут входить все перечисленные выше разделы, а также раздел операторов, составляющий непосредственно сами процедуры и функции.

***Раздел операторов***, начинающийся со слова begin и заканчивающийся словом end, включает в себя непосредственно весь процесс вычислений. Зарезервированные слова begin и end отмечают начало и конец программы.

Порядок разделов описаний может быть любым. Разделы друг от друга отделяются точкой с запятой. Раздел операторов является основным в программе. Все другие разделы, кроме раздела операторов могут отсутствовать. Операторы друг от друга отделяются точкой с запятой. В конце раздела операторов после ключевого слова end обязательно ставится точка, которая является признаком конца записи программы.

При записи программы на каждой строке можно писать либо по одному, либо по несколько операторов. Запись оператора можно начинать в любом месте строки. Ее можно продолжить на следующую строку, не разрывая имен констант, переменных, типов и символов. Кроме того, можно использовать пустые строки, чтобы отделить одну часть программы от другой.

## 2.2 Операторы языка

Оператор – это описание действий, которые будут выполнены при реализации алгоритма. В Паскале существуют 2 вида операторов: простые и структурные.

Простые операторы: оператор присваивания, оператор перехода, процедуры и функции, пустой оператор. Структурные операторы: составной оператор, оператор условия, оператор варианта, оператор цикла.

Пустой оператор

В программе может применяться пустой оператор, не выполняющий никакого действия, представляющий собой точку с запятой. Он может потребоваться для осуществления на него безусловного перехода или для более наглядного просмотра программы.

Goto M1;

. . . .

M1:

. . . .

Оператор присваивания

Оператор присваивания используется для задания значения переменных и имеет следующий синтаксис:

имя\_переменной**:=** выражение;

Вычисляется выражение, стоящее в правой части оператора, после чего его значение записывается в переменную, имя которой стоит слева. Тип выражения и тип переменной должны быть совместимы, т.е. множество допустимых значений для типа выражения содержится во множестве допустимых значений для типа переменной.

**Составной оператор**

Составным оператором считается последовательность произвольных операторов, заключенная в операторные скобки - зарезервированные слова **begin** ... **end**. Допускается произвольная глубина вложенности составных операторов. Составной оператор применяется там, где по синтаксическим правилам языка может стоять только один оператор, а нам надо выполнить несколько действий. В этом случае набор необходимых команд должен быть оформлен как составной оператор. По сути, все тело программы представляет собой один составной оператор.

Простейший ввод и вывод

Рассмотрим простейшие процедуры ввода и вывода. По умолчанию ввод осуществляется с клавиатуры, а вывод на экран. К операторам ввода относятся:

**Read**(<список переменных через запятую>);

**Readln**(<список переменных>);

**Readln**;

Второй отличается от первого тем, что после ввода переводит курсор на новую строку, точнее, в конце своей работы считывает с клавиатуры код клавиши <**Enter**>. Третий оператор используется для организации паузы - выполнение программы продолжится, как правило, только после нажатия на клавиатуре клавиши <**Enter**>.

К операторам вывода относятся:

**Write**(<список вывода>);

**Writeln**(<список вывода>);

**Writeln**;

В списке вывода кроме имен переменных можно писать строковые константы (последовательность символов в апострофах) и даже выражения (выводятся их значения). Второй оператор отличается от первого тем, что после вывода переводит курсор на новую строку. Третий оператор просто переводит курсор на новую строку.

**Форматированный вывод**

В процедурах вывода *Write* и *WriteLn* имеется возможность записи выражения, определяющего ширину поля вывода. В приведенных ниже форматах используются следующие обозначения:

*I, p, q* – целочисленное выражение;

*R* –выражение вещественного типа;

*B* – выражение булевского типа;

*Ch* –выражение символьного типа;

*S* –выражение строкового типа;

*#* -цифра;

*\** - знак “+” или “-“;

*\_*-пробел.

**I**-выводится десятичное представление величины I, начиная с позиции расположения курсора.

*Значение I Выражение Результат*

134 Write (I); 134

287 Write (I,I,I); 287287287

**I:p** -выводится десятичное представление величины I в крайние правые позиции поля шириной p.

*Значение I Выражение Результат*

134 Write (I: 6); \_ \_ \_134

1 Write (I: 1); \_ \_ \_ \_ \_ \_ \_ \_ \_ 1

312 Write (I+I:7) \_ \_ \_ \_624

**R**- в поле шириной 18 символов выводится десятичное представление величины R в формате с плавающей точкой. Если R>=0.0, используется формат \_#.##########E\*##.

Если R<0.0, формат имеет вид: -#.##########E\*##.

*Значение R Выражение Результат*

715.432 Write (R); \_ \_ 7.1543200000E+02

-1.919E+01 Write (R); \_ -1.9190000000E+01

567.986 Write (R/2); \_ \_ 2.8399300000E+02

**R:p** –в крайние правые позиции поля шириной p символов выводится десятичное представление значения R в формате с плавающей точкой. Если R>=0.0, используется формат \_ \_…\_##..#E\*##, причем минимальная длина поля вывода составляет 7 символов. Если R<0.0, формат имеет вид:

\_ \_ …\_--#.##..#E\*##. Минимальная длина поля вывода 8 символов. После десятичной точки выводится, по крайней мере, одна цифра.

*Значение R Выражение Результат* 511.04 Write (R:15); 5.110400000E+02

-511.04 Write (R:15); -5.11040000E+02

46.78 Write (-R:12); -4.67800E+01

**R:p:q** –в крайние правые позиции поля шириной p символов выводится десятичное представление значения R в формате с фиксированной точкой, причем после десятичной точки выводится q цифр (0<=q<=24), представляющих дробную часть числа. Если q=0, ни дробная часть, ни десятичная точка не выводится. Если q>24, то при выводе используется формат с плавающей точкой.

*Значение R Выражение Результат*

511.04 Write (R:8:4); 511.0400

-46.78 Write (R:15); \_-46.78

-46.78 Write (R:9:4); \_-46.7800

**Ch**-начиная с позиции курсора выводится значение Ch.

*Значение Ch Выражение Результат*

’X’ Write (Ch); X

‘Y’ Write (Ch); Y

’!’ Write (Ch, Ch, Ch); !!!

**Ch:p**-в крайнюю правую позицию поля шириной p выводится значение Ch.

*Значение Ch Выражение Результат*

’X’ Write (Ch:3); \_ \_X

‘Y’ Write (Ch:5); \_ \_ \_ \_Y

’!’ Write (Ch:2, Ch:4); \_!\_ \_ \_!

###### S- начиная с позиции курсора, выводится значение S (строка или массив символов, если его длина соответствует длине строки).

*Значение S Выражение Результат*

’Day N’ Write (S); Day N ‘Ведомость 11’ Write (S); Ведомость 11

’RRRDDD’ Write (S, S); RRRDDDRRRDDD

###### S:p- значение S выводится в крайние правые позиции поля шириной p символов.

*Значение S Выражение Результат*

’Day N’ Write (S:10); \_ \_ \_ \_ \_ Day N ‘Ведомость 11’ Write (S:13); \_Ведомость 11

’RRRDDD’ Write (S:7, S:7); \_RRRDDD\_RRRDDD

###### B- выводится результат выражения B True или False, начиная с текущей позиции курсора.

*Значение B Выражение Результат*

True Write (B); True

False Write (B, not B); False True

###### B:p- в крайние правые позиции поля шириной p символов выводится результат булевского выражения B True или False.

*Значение B Выражение Результат*

True Write (B:6); \_ \_ True

###### False Write (B:10); \_ \_ \_ \_ \_False

True Write (B:5,not B:7); \_True\_ \_False

###### Оператор записи WriteLn аналогичен процедуре Write, но после вывода последнего в списке значения для текущей процедуры WriteLn происходит перевод курсора к началу следующей строки.

Процедура WriteLn, записанная без параметров, вызывает перевод строки.

#### Пример программы с использованием процедур ввода-вывода данных с различными форматами выводимых данных

Program Demo;

Var

A,B,S:Integer;

Begin

Writeln(‘Введите сторону А = ’);

Readln(A);

Writeln(‘Введите сторону B = ’);

Readln(B);

S:=A\*B;

Writeln(‘--------------------------------------------‘);

Writeln(‘ | Сторона А | | Сторона B | | Площадь | ‘);

Writeln(‘---------------------------------------------‘);

Writeln(‘|‘,А:7,B:11,S:11, ‘|‘ :5);

Writeln(‘---------------------------------------------‘);

End.

В результате работы данной программы на экране будет изображена следующая таблица:

--------------------------------------------------

| Сторона А | | Сторона B | | Площадь |

--------------------------------------------------

| 8 4 32 |

--------------------------------------------------

каждая строка которой будет печататься с первой позиции новой строки экрана.

РАЗВЕТВЛЯЮЩИЕСЯ АЛГОРИТМЫ

В Паскале имеется возможность нелинейного хода программы, т.е. выполнения операторов не в том порядке, в котором они записаны. Такую возможность нам предоставляют разветвляющиеся алгоритмы. Они могут быть реализованы одним из трех способов: с использованием операторов перехода, условного оператора или оператора выбора.

Оператор перехода

Оператор перехода имеет вид:

**GOTO** <метка>.

Он позволяет передать управление непосредственно на нужный оператор программы. Перед этим оператором должна располагаться метка, отделенная от него двоеточием. В Паскале в качестве меток выступают либо целые числа от 0 до 9999, либо идентификаторы. Все метки должны быть описаны в разделе объявления меток следующим образом:

**label** <список меток через запятую> ;

Каждой меткой в программе может быть помечен только один оператор. Операторов перехода с одной и той же меткой можно писать любое количество. Необходимо, чтобы раздел описания метки, сама метка и оператор перехода с ее использованием располагались в пределах одного блока программы. Кроме того, нельзя передавать управление внутрь структурированных операторов (например, if, for, while, repeat и др.).

Условный оператор

Условный оператор **IF** позволяет изменить порядок выполнения команд в зависимости от некоторого логического условия, т.е. он осуществляет ветвление вычислительного процесса. Условный оператор имеет вид:

**IF** <условие> **THEN** <оператор1> [**ELSE** <оператор2>];

В случае истинности логического выражения, стоящего в условии, выполняется <оператор1>, а <оператор2> пропускается. При ложном значении логического выражения пропускается <оператор1> и выполняется <оператор2>.

Оператор **IF** может быть полным (присутствуют обе ветви) или неполным (**Else**-ветви нет, при ложном условии ничего не делается). По правилам каждая из ветвей может содержать либо один выполняемый оператор, либо несколько, объединенных в составной. Точка с запятой перед **Else** считается ошибкой.

***Пример****. Ввести целое число. Вывести соответствующий ему символ ASCII-таблицы, либо сообщить, что такого символа нет (0-31 - управляющие коды, затем до 256 - печатаемые символы).*

**program** ascii\_symbol;

**var** i:word;

**begin**

**write**('Введите целое число: '); **readln**(i);

**if** (i>31) **and** (i<256) **then**

**writeln**('Соответствующий символ - ', Chr(i))

**else** **writeln**('Такого символа нет');

**readln**

**end**.

Оператор выбора

Оператор выбора является обобщением условного оператора: он дает возможность выполнить один из нескольких операторов в зависимости от значения некоторого выражения, называемого селектором.

**CASE <селектор> OF**

<**список меток 1**> **:** <**оператор 1**>;

<**список меток 2**> **:** <**оператор 2**>;

. . . . . . . . . . . . .;

<**список меток N**> **:** <**оператор N**>;

**ELSE <оператор>**

**END;**

*селектор* - выражение любого перечисляемого типа, кроме вещественного;

*оператор* - любой оператор языка, в том числе и составной;

*список меток* - список разделенных запятыми значений выражения 'селектор' или одно его значение; тип метки и селектора должен быть одинаков;

Оператор варианта выбирает для исполнения тот ОПЕРАТОР, одна из меток которого равна текущему значению выражения СЕЛЕКТОР. Если ни одна из меток не равна текущему значению селектора, то никакие операторы не выполняются, либо выполняются операторы, следующие за зарезервированным словом ELSE (если такое имеется).

Ветвь **Else** не обязательна, и в отличие от оператора **if**, перед ней ставится точка с запятой. Если для нескольких значений <селектора> действия совпадают, то эти константы можно перечислить через запятую перед двоеточием или задать диапазон значений (нижняя граница .. верхняя граница).

***Пример****: Вводится целое число, если это цифра, то определить четная она или нет, а если число, то определить попадает ли оно в диапазон от 10 до 100, если нет, то выдать соответствующее сообщение.*

**program** chislo;

**var** i:integer;

**begin**

**write**('Введите целое число: ');

**readln**(i);

**case** i **of**

0,2,4,6,8 : **writeln**('Четная цифра');

1,3,5,7,9 : **writeln**('Нечетная цифра');

10...100,200 : **writeln**('Число от 10 до 100 или 200');

**else** **writeln**('Число либо отрицательное, либо > 100, но не 200');

**end**;

**readln**

**end**.

ЦИКЛИЧЕСКИЕ АЛГОРИТМЫ

Последовательность команд, выполняющихся несколько раз в зависимости от некоторого условия, называется циклом.

Цикл с параметром

Если заранее известно число повторений цикла, то в программе используются циклы с параметром. Оператор цикла **For** организует выполнение одного оператора заранее определенное число раз. Его еще называют цикл со счетчиком. В общем виде цикл с параметром задается следующим образом:

**FOR** <параметр> := <nz> **TO** <kz> **DO** <оператор>

Здесь параметр цикла (счетчик) представляет собой переменную порядкового типа; <nz> и <kz> - выражения, определяющие начальное и конечное значение счетчика; <оператор> - один (возможно составной) оператор, который называют телом цикла, повторяемый определенное число раз.

На первом шаге цикла параметр принимает значение nz. После каждого выполнения тела цикла, если параметр цикла не равен kz, происходит увеличение параметра на единицу. Если nz > kz, то <тело цикла> не будет выполнено ни разу и выполнение цикла с параметром сразу же закончится.

Возможна другая форма цикла с параметром.

**FOR** <параметр> := < kz > **DOWNTO** < nz > **DO** <оператор>,

которая выполняется аналогичным образом, но значение <параметра> изменяется с шагом, равным -1.

***Рекомендации*:** Использовать цикл **for** при заранее известном количестве повторений. Не изменять параметр в теле цикла. При использовании кратных (вложенных) циклов применять разные переменные в качестве параметров. Определять до цикла значения всех используемых в нем переменных. Не ставить точку с запятой после **do**.

#### Примеры программ с использованием оператора for

***Пример 1.*** Программа *DemoFor1*  выводит на экран таблицу перевода из градусов по шкале Цельсия(С) в градусы по Фаренгейту(Р) для значений от 15°С до 30°С с шагом 1 градус. Перевод осуществляется по формуле: F = С\*1.8+32.

program DemoFor1;

var

I: integer;

F: real;

begin

Writeln (' Температура ') ;

for I:= 15 to 30 do *{Заголовок цикла с параметром}*

begin *{Начало тела цикла}*

F:= I\*1.8+32;

Writeln('no Цельсию= ',I,' по Фаренгейту= ', F:5:2)

end; *{Конец тела цикла}*

end.

В блоке описания переменных описаны параметр цикла *I* типа *integer* и переменная *F* — температура по Фаренгейту типа *real*. Переменная *I,* помимо функций управляющей переменной, является переменной, хранящей целочисленные значения температуры по шкале Цельсия. В начале выполнения программы на экран выводится надпись *' Температура* *'*, а затем оператором повтора выводится таблица соотношения температуры в шкалах Цельсия и Фаренгейта. Печать таблицы выполняется оператором *Writeln('По Цельсию= ',I,' по Фаренгейту= ' , F: 5:2).*

Цикл выполняется следующим образом.

При первом обращении к оператору *for* вычисляются значения начального *(15)* конечного *(30)* параметров цикла, и управляющей переменной *I* присваивается начальное значение *15*.

Затем циклически выполняется следующее:

1. Проверяется условие *I<=30.*
2. Если оно соблюдается, то выполняется составной оператор в теле цикла, т.е. рассчитывается значение выражения *I\* 1.8+32*, затем оно присваивается переменной *F,* и на экран выводится сообщение: 'По Цельсию= ', I, ' по Фаренгейту= ', F:5:2.

Если условие *I<=30* не соблюдается, т. е. как только *I* станет *> 30,* оператор тела цикла не выполняется, а управление в программе передается за пределы оператора *for*, в нашем примере на оператор *end*. Программа завершает работу.

3. Значение параметра цикла *I* увеличивается на единицу, и управление передается в заголовок цикла *for* для проверки условия.

Далее цикл повторяется, начиная с пункта 1.

***Пример 2.*** Программа *DemoFor2*, которая печатает на экране символы американскою стандартного кода обмена информацией *(ASCII)* в порядке *убывания* кода.

program DemoFor2;

var

A: integer;

begin

for A:= 255 downto 0 do {*Цикл с убыванием параметра}*

Writeln('код символа = ',А, ' символ == ',Chr(A));

end.

В данной программе применяется *цикл for с убыванием значения управляющей переменной* А (используется указание *downto* - убывание).

***Пример 3****: Вводятся 10 чисел, посчитать среди них количество положительных.*

**program** cycle\_for1;

**var** i,kn:byte; x:real;

**begin**

kn:=0;

**for** i:=1 **to** 10 **do**

**begin**

**writeln**('Введите ',i,' число: ');

**readln**(x);

**if** x>0 **then** kn:=kn+1 {увеличиваем количество на 1}

**end**;

**writeln**('Вы ввели ',kn,' положительных чисел.');

**readln**

**end**.

***Пример 4****: Напечатать буквы от 'Z' до 'A'.*

**program** cycle\_for2;

**var** c:char;

**begin**

**for** c:='Z' **downto** 'A' **do** **write**(c);

**readln**

**end**

***Пример 5****: Вычислить N-е число Фиббоначчи. Числа Фиббоначчи строятся следующим образом: F(0)=F(1)=1; F(i+1)=F(i)+F(i-1); для i>=1. Это пример вычислений по рекуррентным формулам.*

**program** Fib;

**var** a,b,c:word; i,n:byte;

**begin**

**write**('введите номер числа Фиббоначчи ');

**readln**(N);

a:=1; {a=F(0), a соответствует F(i-2)}

b:=1; {b=F(1), b соответствует F(i-1)}

**for** i:=2 **to** N **do**

**begin**

c:=a+b; {c соответствует F(i)}

a:=b; b:=c; {в качестве a и b берется следующая пара чисел}

**end**;

**writeln**(N,'-е число Фиббоначчи =',b); {для N>=2 b=c}

**readln**

**end**.

Циклы с условием

Если заранее неизвестно число повторений цикла, то используются циклы с условием. В Паскале имеется два типа таких циклов. Это циклы с предусловием и постусловием.

**Цикл с предусловием**

**WHILE** <логическое выражение> **DO** <оператор>;

Выполнение оператора цикла с предусловием начинается с проверки условия, записанного после слова while. Если оно соблюдается, то выполняется <тело цикла>, а затем вновь проверяется условие и т.д. Как только на очередном шаге окажется, что условие не соблюдается, то выполнение <тела цикла> прекратится.

Если <тело цикла> состоит из нескольких операторов, то они объединяются операторными скобками.

В теле цикла обязательно должен быть оператор, влияющий на соблюдение условия, в противном случае произойдет зацикливание.

**Пример программы с использованием цикла while**

Программа *DemoWhile* производит суммирование 10 произвольно введенных целых чисел.

program DemoWhile;

const

Limit =10; *{Ограничение на количество вводимых чисел}*

var Count, Item, Sum: integer;

begin

Count:=0; *{Счетчик чисел}*

Sum:= 0; *{Сумма чисел}*

while (Count < Limit) do *{Условие выполнения цикла}*

begin

Count:= Count+1;

Write('Введите ', Count, ' - e целое число: ');

Readln(Item);*{Ввод очередного числа с клавиатуры}*

Sum:= Sum+Item;

end;

Writeln('Сумма введенных чисел равна ', Sum) ;

end.

В данном примере в разделе описания констант описана константа *Limit=10,* задающая ограничение на количество вводимых чисел. В разделе описания переменных описаны переменные *Count, Item, Sum* целочисленного типа. В начале выполнения программы обнуляются значения счетчика введенных чисел *Count* и их суммы *Sum*. Затем выполняются цикл ввода 10 чисел и их суммирование. Вначале оператор *условия while* проверяет условие *Count < Limit*. Если условие верно, то выполняется составной оператор в теле цикла:

begin

Count:= Count+1;

Write('Введите ', Count, '-e целое число: ');

Readln(Item) ;

Sum:= Sum+Item;

End;

в котором вводится значение очередного числа, и на это значение увеличивается значение суммы. После этого управление в программе вновь передается оператору цикла *while*, опять проверяется условие *Count < Limit*. Если условие верно, то выполняется составной оператор и т. д., пока значение переменной *Count* будет меньше 10. Как только значение *Count* станет равно 10 и условие *Count < Limit* не будет соблюдено, выполнение цикла завершится, а управление в программе будет передано на оператор, находящийся за словом *end*, т. e. первый оператор за границей *while.* Это вызов процедуры *Writeln,* которая выведет сообщение *'Сумма введенных чисел равна'* и напечатает значение переменной *Sum*.

**Цикл с постусловием**

**REPEAT**

<оператор 1>

...

<оператор N>

**UNTIL** <логическое выражение>

Оператор **Repeat** организует повторяющееся выполнение нескольких операторов до тех пор пока не станет истинным условие, стоящее в **Until**-части. Тело цикла обязательно выполняется хотя бы один раз. Таким образом, в этом цикле логическое выражение - это условие выхода из цикла.

При создании циклических алгоритмов Турбо Паскаль позволяет использовать процедуры **Continue** и **Break**. Процедура **Continue** досрочно завершает очередной шаг цикла, передает управление на заголовок. Процедура **Break** реализует немедленный выход из цикла.

***Рекомендации*:** Для того чтобы избежать зацикливания программы необходимо обеспечить изменение на каждом шаге цикла значения хотя бы одной переменной, входящей в условие цикла. После выхода из цикла со сложным условием (с использованием операций **and**, **or**, **xor**) как правило, необходима проверка того, по какому условию цикл завершен.

***Пример 1****: Пары неотрицательных вещественных чисел вводятся с клавиатуры. Посчитать произведение для каждой пары и сумму всех чисел.*

**program** cycle\_while;

**var** x,y,sum:real; otv:char;

**begin**

sum:=0;

otv='Д';

**while** (otv='Д') **or** (otv='д') **do**

**begin**

**write**('Введите числа x,y > 0 ');

**readln**(x,y);

**writeln**('Их произведение = ',x\*y:8:3);

sum:=sum+x+y;

**write**('Завершить программу (Д/Н)? ');

**readln**(otv);

**end**;

**writeln**('Общая сумма = ',sum:8:3);

**readln**

**end.**

***Пример 2****: В той же задаче можно использовать другой цикл с условием:*

**program** cycle\_repeat;

**var** x,y,sum:real; otv:char;

**begin**

sum:=0;

**repeat**

**write**('Введите числа x,y > 0 ');

**readln**(x,y);

**writeln**('Их произведение = ',x\*y:8:3);

sum:=sum+x+y;

**write**('Завершить программу (Д/Н)? ');

**readln**(otv);

**until** (otv='Д') **or** (otv='д');

**writeln**('Общая сумма = ',sum:8:3);

**readln**

**end**.

***Пример 3****: Нахождение наибольшего общего делителя двух целых чисел с помощью Алгоритма Эвклида.*

**program** Evklid;

**var** a,b,c:integer;

**begin**

**write**('введите два целых числа : ');

**readln**(a,b);

**while** b<>0 **do**

**begin**

c:=a **mod** b;

a:=b;

b:=c;

**end**;

**writeln**('наибольший общий делитель = ',a);

**readln**

**end**.

**Вложенные циклы**

Если для решения задачи требуется организовать два цикла, один из которых помещается внутрь другого, то такие конструкции называют вложенными циклами.

Например, даны натуральные числа n и k. Составить программу вычисления выражения 1k + 2k +...+ nk.

Для вычисления указанной суммы целесообразно организовать цикл с параметром i, в котором, во-первых, вычислялось бы очередное значение y = ik и, во-вторых, осуществлялось бы накопление суммы прибавлением полученного слагаемого к сумме всех предшествующих (s = s + y).

Program Example\_13;

Var n, k, y, i, s, m: Integer;

Begin

Writeln ('Введите исходные данные n и k');

Readln(n,k);

s:=0;

For i:=1 To n Do

Begin y:=1;

For m:=1 To k Do y:=y\*i; {нахождение степени k числа i}

s:=s+y;

End;

Writeln('Ответ: ',s);

End.

Внутренний и внешний циклы могут быть любыми из трёх рассмотренных ранее видов: циклами с параметром, циклами с предусловием или циклами с постусловием. Правила организации как внешнего, так и внутреннего циклов такие же, как и для простых циклов каждого из этих видов. Но при использовании вложенных циклов необходимо соблюдать следующее условие: внутренний цикл должен полностью укладываться в циклическую часть внешнего цикла.

* 1. **Массивы**

Рассмотренные ранее *простые типы данных* определяют различные множества атомарных (неразделимых) значений. В отличие от них *структурные типы данных* задают множества сложных значений, каждое из которых образует совокупность нескольких значений другого типа. В структурных типах выделяют регулярный тип (массивы).

С понятием "массив" приходится сталкиваться при решении научно-технических и экономических задач обработки совокупностей большого количества значений. В общем случае ***массив*** - это структурированный тип данных, состоящий из фиксированного числа элементов, имеющих один и тот же тип.

Название *регулярный тип* (или ряды) массивы получили за то, что в них объединены однотипные (логически однородные) элементы, упорядоченные по индексам, определяющим положение каждого элемента в массиве.

В качестве элементов массива можно использовать и любой другой ранее описанный тип, поэтому вполне правомерно существование массивов записей, массивов указателей, массивов строк, массивов и т.д. Элементами массива могут быть данные любого типа, включая структурированные. Тип элементов массива называется *базовым*. Особенностью языка Паскаль является то, что *число элементов массива фиксируется* при описании и в процессе выполнения программы не меняется.

Элементы, образующие массив, упорядочены таким образом, что каждому элементу соответствует совокупность номеров (индексов), определяющих его местоположение в общей последовательности. *Доступ* к каждому отдельному элементу осуществляется путем индексирования элементов массива. *Индексы* представляют собой выражения любого скалярного типа, кроме вещественного. Тип индекса определяет границы изменения значений индекса. Для описания массива предназначено словосочетание: *array of .*

Формат записи массивов:

Type

<имя типа> = array [тип индекса] of <тип компонента>;

Var

<идентификатор> : <имя типа>;

Массив может быть описан и без представления типа в разделе описания типов данных:

Var <идентификатор> : array [тип индекса] of <тип компонента>;

**Примеры описания одномерных массивов**

Если в описании массива задан один индекс, массив называется *одномерным*. Одномерный массив соответствует понятию линейной таблицы (вектора).

*Одномерные массивы описываются следующим образом:*

Type

Klass = (К1, К2, КЗ, К4) ;

Znak = array [1..255] of char;

Var

Ml: Znak; {Тип Znak предварительно описан в разделе типов}

М2: array[1..4] of Klass;

М3: array[1..60] of integer; {Прямое описание массива М3}

Mas: array[1..4] of integer;

В Паскале количество элементов массива всегда должно быть фиксировано, т. е. определяться при трансляции программы.

###### **Действия над массивами**

Для работы с массивом как единым целым используется идентификатор массива без указания индекса в квадратных скобках. Массив может участвовать только в операциях отношения *"равно", "не равно" и в операторе присваивания*. Массивы, участвующие в этих действиях, должны быть идентичны по структуре, т. е. иметь одинаковые типы индексов и одинаковые типы компонентов.

Например, если массивы А и В описаны как

*Var А, В: array[1..20] of real,*

то применение к ним допустимых операций даст следующий результат:

*Выражение Результат*

*А=В* *True*, если значение каждого элемента массива А равно соответствующему значению элемента массива В

*А<>В* *True*, если хотя бы одно значение элемента массива А не равно значению соответствующего элемента массива В

*А:=В* Все значения элементов массива В присваиваются соответствующим элементам массива А. Значения элементов массива В остаются неизменны.

###### **Действия над элементами массива**

После объявления массива каждый его элемент можно обработать, указав *идентификатор (имя) массива* и *индекс элемента* в квадратных скобках. Например, запись *Mas[2],* *Vector[10]* позволяет обратиться ко второму элементу массива *Mas* и десятому элементу массива *Vector*.

Индексированные элементы массива называются *индексированными переменными* и могут быть использованы так же, как и простые переменные. Например, они могут находиться в выражениях в качестве операндов, использоваться в операторах *for, while, repeat*, входить в качестве параметров в операторы *Read, Readin, Write, Writeln*; им можно присваивать любые значения, соответствующие их типу.

Рассмотрим типичные ситуации, возникающие при работе с данными типа *array*. Для этого опишем три массива и четыре вспомогательные переменные:

Var

A,D : array[l.,4] of real;

В : array[1..10,1..15] of integer;

I, J, К : integer;

S : real;

*Инициализация (присваивание начальных значений) массива* заключается в присваивании каждому элементу массива одного и того же значения, соответствующего базовому типу. Наиболее эффективно эта операция выполняется с помощью оператора for, например:

*Инициализация элементов одномерного массива А:*

*for I := 1 to 4 do*

*A[I] := 0;*

###### **Ввод-вывод элементов массива**

Паскаль не имеет средств ввода-вывода элементов массива сразу, поэтому ввод и вывод значений производится поэлементно. Значения элементам массива можно присвоить с помощью оператора присваивания, как показано в примере инициализации, однако чаще всего они вводятся с экрана с помощью оператора *Read* или *Readln* с использованием оператора организации *цикла for*:

*Ввод элементов одномерного массива А:*

*for I:=l to 4 do*

*Readln(A[I]) ;*

В связи с тем, что использовался оператор *Readln*, каждое значение будет вводиться с новой строки. Можно ввести и значения отдельных элементов, а не всего массива. Так, операторами:

*Read(A[3]);*

вводится значение третьего элемента массива А.

*Вывод значений элементов массива* выполняется аналогичным образом, но используются операторы *Write* или *Writeln*:

*Вывод элементов одномерного массива А:*

for I := I to 4 do

Writeln (A[I]);

*Копированием* массивов называется присваивание значений всех элементов одного массива всем соответствующим элементам другого массива. Копирование можно выполнить одним оператором присваивания, например *А:=D;* или с помощью оператора *for*:

for I := 1 to 4 do

A[I] := D[I];

В обоих случаях значение элементов массива D не изменяется, а значения элементов массива А становятся равными значениям соответствующих элементов массива D. Очевидно, что оба массива должны быть идентичны по структуре.

Иногда требуется осуществить *поиск в массиве* каких-либо элементов, удовлетворяющих некоторым известным условиям. Пусть, например, надо выяснить, сколько элементов массива А имеют нулевое значение.

К := 0;

For I := 1 to 4 do

if A[I] = 0 then К := К + 1;

После выполнения цикла переменная *К* будет содержать количество элементов массива *А* с нулевым значением.

*Перестановка значений элементов массива* осуществляется с помощью дополнительной переменной того же типа, что и базовый тип массива.

Например, так запишется фрагмент программы, обменивающий значения первого и пятого элементов массива А:

Vs:= A[5]; {Vs - вспомогательная переменная}

А[5]:= А[1];

А[1]:= Vs;

**Примеры решения типовых задач**

###### **Сформировать и вывести на экран последовательность из п элементов одномерного массива, вводимых с клавиатуры.**

program primer1;

var

mas:array [1..100] of integer; n,i:integer;

BEGIN

writeln('Введите количество элементов массива не больше 100: ');

readln(n);

for i:=1 to n do *{ввод элементов массива}*

begin

writeln('Введите ',i,' элемент массива');

readln(mas[i]);

end; *{конец ввода}*

writeln('Введенный массив: ');

for i:=1 to n do *{вывод элементов массива}*

begin

write(' ',mas[i],' ');

end; *{конец вывода}*

END.

1. ***Сформировать и вывести на экран последовательность из n элементов, заданных датчиком случайных чисел на интервале [-23, 34].***

Program posled;

Var a: array[1..100] of integer;

i, n: integer;

Begin

Write (‘Сколько элементов? ’); Readln (n);

For i=1 to n do

begin

a[i]:= Random(58)-23;

writeln (a[i],’ ‘);

end;

End.

***3. Найти сумму элементов одномерного массива. Размер произвольный. Элементы вводятся с клавиатуры.***

Program summa;

Var a: array[1..100] of real;

i, n: integer;

s: real;

Begin

Write (‘n=’); Readln (n);

s:=0;

For i:=1 to n do

begin

write (‘введите число’); readln (a[i]);

s:=s+a[i];

end;

writeln(‘сумма элементов равна ‘,s);

End.

***4. Найти номер наименьшего элемента в массиве, заданного датчиком случайных чисел на интервале [-20, 25]. Размер произвольный.***

Program numberminim;

Var a: array[1..100] of integer;

i, n, num, min: integer;

Begin

Write (‘n=’); Readln (n);

For i:=1 to n do

begin

a[i]:= Random(46)-20;

writeln (a[i]);

end;

min:=a[1];

num:=1;

For i:=2 to n do

If a[i]< min then

begin

min:=a[i];

num:=i;

end;

Writeln(‘ номер наименьшего элемента: ‘,num);

End.

***5. Найти произведение элементов целочисленного одномерного массива с четными номерами, состоящего из n элементов. Элементы вводятся с клавиатуры.***

Program proizved\_chet;

Var a: array [1..100] of integer;

i, n, p: integer;

Begin

p:=1;

write ('n='); readln (n);

for i:=1 to n do

begin

write ('a[',i,']='); readln (a[i]);

if i mod 2=0 then p:=p\*a[i];

end;

Writeln ('произведение элементов массива с четными номерами равно ',p);

End.

***6. Сортировка целочисленного массива в порядке возрастания***

program z5\_3; uses crt;

const n=10;

var x:array [1..n] of integer; i,j,h:integer;

BEGIN

clrscr; randomize;

writeln('исходный массив:');

for i:=1 to n do begin

x[i]:=random(n+1); write(x[i]:3);

end;

writeln; writeln('упорядоч. массив:');

for i:=2 to n do begin

for j:=n downto i do begin

if x[j-1]>x[j] then begin

h:=x[j-1];

x[j-1]:=x[j];

x[j]:=h;

end; end; end;

for j:=1 to n do write(x[j]:3);

END.

**Двумерные массивы**

Двумерный массив можно представить как таблицу или матрицу. Для получения доступа к его элементам используются два индекса: номер строки и номер столбца. При описании в типе индексов надо указать диапазон для двух индексов массива.

*Примеры описания двумерных массивов:*

Type

Vector = array[1..4] of integer;

Massiv = array[1..4] of Vector;

Var

Matrix : Massiv;

Ту же структуру можно получить, используя другую форму записи:

Var

Matrix : array[1..4,1..4] of integer ;

Для описания массива можно использовать предварительно определенные константы:

Const

G1 = 4; G2 = 6;

Var

MasY: array[1..Gl, l..G2] of real;

*Элементы массива располагаются в памяти последовательно.* *Элементы с меньшими значениями индекса хранятся в более низких адресах памяти.* *Многомерные массивы располагаются таким образом, что самый правый индекс возраста­ет самым первым.*

Например, если имеется массив:

A:array[1..5,1..5] of integer;

то в памяти элементы массива будут размещены по возрастанию адресов:

А[1,1] А[1,2] … А[1,5] А[2,1] А[2,2] … А[5,5]

При работе с *двумерным массивом* указываются два индекса, с n-мерным массивом - n индексов. Например, запись *Matr[4,4]* делает доступным для обработки значение элемента, находящегося в *четвертой строке* *четвертого столбца* массива *Matr*.

**Процедуры обработки матриц**

***Сумма указанной строки:***

S:=0; i:=3;

For J:=1 to m do S:=S+b[i,j];

***Транспонирование квадратной матрицы:***

For i:=1 to n-1 do

For j:=1 to n do

Begin

P:= a[i,j];

a[i,j]:=a[j,i];

a[j,i]:=p;

end;

***Удаление строки из матрицы:***

n:=n-1;

For i:=1 to n do

For j:=1 to m do b[i,j]:=b[i+1,j];

***Включение строки в матицу:***

i:=n;

while i>=k do

begin

for j:=1 to m do b[i+1,j]:=b[i,j];

i:=i-1;

end;

for j:=1 to m do b[k,j]:=c[j];

n:=n+1;

***Перестановка строк матрицы:***

For k:=1 to k do c[k]:=a[i,k];

For k:=1 to k do a[i,k]:=a[j,k];

For k:=1 to k do a[j,k]:=c[k];

***Поиск минимального элемента матрицы:***

Min:= a[1,1]; k;=1; L:=1;

For i:=1 to n do

For j:=1 to m do

IF min>a[i,j] then begin min:=a[i,j]; k:=i; l:=j; end;

***Сложение строк матрицы:***

For j:=1 to m do

a[k,j]:=a[k,j]+a[l,j]\*b;

**Примеры решения типовых задач**

###### **Программа ввода-вывода двумерного массива**

program primer;

var

i,j,n,m:integer;

massiv:array[1..10,1..10] of integer; *{описан двумерный массив с именем massiv}*

BEGiN

write('Введите количество строк массива (не больше 10): ');

read(n); *{n-количество строк}*

write('Введите количество столбцов массива (не больше 10):');

read(m); *{m-количество столбцов}*

for i:=1 to n do *{ввод элементов двумерного массива}*

begin

for j:=1 to m do

read(massiv[i,j]);

end; *{конец ввода}*

writeln('Введенный массив: ');

for i:=1 to n do *{вывод элементов двумерного массива}*

begin

for j:=1 to m do

write(massiv[i,j]:5);

end; *{конец вывода}*

END.

1. ***Сформировать с помощью датчика случайных чисел и вывести на экран матрицу, размером МхN. Элементы задаются на интервале [-20, 25].***

Var a: array[1..50,1..50] of integer;

i, j, n, m: integer;

Begin

Write(‘сколько строк?’); Readln(m);

Write(‘сколько столбцов?’); Readln(n);

For i:=1 to m do

begin

For j:=1 to n do

begin

a[i,j]:=int(rnd\*46)-20;

write(a[i,j],’ ‘);

end;

writeln;

end;

End.

1. ***В двумерном массиве, состоящем из целых чисел, найти наименьший элемент и номер строки, в которой он находится. Элементы вводятся с клавиатуры. Размер MXN.***

Program minim;

Var a: array[1..50,1..50] of integer;

i, j, m, n, min, K: integer;

Begin

Write(‘сколько строк?’); Readln(m);

Write(‘сколько столбцов?’); Readln(n);

For i:=1 to m do

For j:=1 to n do

begin write(‘a[‘,i,’,’,j,’]=’); readln (a[i,j]); end;

min:=a[1,1]; K:=1;

For i:=1 to m do

For j:=1 to n do

If a[i,j]< min then

begin

min:=a[i,j]; K:=i;

end;

Writeln(‘наименьшее число ‘,min,’ находится в ‘, k , ‘ строке’);

End.

* 1. **Строки и множества**

***Строка*** *—* это последовательность символов кодовой таблицы персонального компьютера. При использовании в выражениях строка заключается в апострофы. Количество символов в строке (длина строки) может динамически изменяться от 0 до 255. Для определения данных строкового типа используется идентифи­катор *String*, за которым следует заключенное в квадратные скобки значение мак­симально допустимой длины строки данного типа. Если это значение не указывается, то по умолчанию длина строки равна 255 байт.

Переменную строкового типа можно определить через описание типа в разделе определения типов или непосредственно в разделе описания переменных. Строковые данные могут использоваться в программе также в качестве констант.

Недопустимо применение строковых переменных в качестве *селектора* в операторе *Case.*

Определение строкового типа устанавливает максимальное количество символов, которое может содержать строка.

**Формат описания строкового типа**

*Type*

*<имя типа> =String [максимальная длина строки];*

*Var*

*<идентификатор, . . . > : <имя типа>;*

Переменную типа *String* можно задать и без описания типа:

*Var*

*<идентификатор, . . . > : String [максимальная длина строки];*

**Пример описания строковых данных**

*Const*

*Address = 'ул. Переверткина, 25'; {Строковая константа}*

*Type*

*Flot = string[125];*

*Var*

*Fstr : Flot; {Описание с заданием типа)*

*St1 : String; {По умолчанию длина строки St1= 255}*

*St2, St3 : string[50] ;*

*Nazv : string[280]; (Ошибка, длина строки Nazv превышает 255}*

Для ввода и вывода переменной типа STRING используются операторы READLN и WRITELN.

***Операции над строками***

1. Операции сравнения. Сравнение происходит посимвольно слева направо: сравниваются коды соответствующих символов, пока не нарушится равенство. Две строки считаются равными, если они равны по длине и совпадают посимвольно.
2. Строки можно объединять с помощью операции сцепления. Знак операции - «+».
3. Индексирование. Так как строку можно рассматривать как массив символов, то при помощи индексирования можно организовать доступ к его отдельным символам по их номерам.

### Стандартные строковые процедуры и функции

***Delete (Str,Poz,N)*** — удаление *N* символов строки *Str*, начиная с позиции *Poz*. Если значение Poz > 255, возникает программное прерывание.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение Str* | *Выражение* | *Результат* |
| 'абвгде'  'река Волга' | Delete(Str, 4, 2);  Delete(Str, 1, 5); | 'абве'  'Волга' |

***Insert (Strl, Str2, Роz)*** –вставка строки *Str1* в строку *Str2*, начиная с позиции *Poz*.

*Например:*

Var

Sl, S2 : string[ll] ;

…

S1 := ' ЕС ';

S2 := 'ЭВМ1841';

Insert(S1,S2,4) ;

В результате выполнения последнего выражения значение строки *S2* станет равным *'ЭВМ ЕС 1841'*.

***Str (IBR,St)*** — преобразование числового значения величины *IBR* и помещение результата в строку *St*. После IBR может записываться формат, аналогичный фор­мату вывода. Если в формате указано недостаточное для вывода количество разря­дов, поле вывода расширяется автоматически до нужной длины.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение IBR* | *Выражение* | *Результат* |
| 1500  4.8Е+03  76854 | Str(IBR:6,St)  Str(IBR:10,St)  Str(-IBR:3,St) | '\_1500'  '\_\_\_\_4800'  '—76854' |

***Val (St,IBR,Code)*** — преобразует значение *St* в величину целочисленного или вещественного типа и помещает результат в *IBR*. Значение *St* не должно содер­жать незначащих пробелов в начале и в конце. *Code* — целочисленная переменная. Если во время операции преобразования ошибки не обнаружено, значение *Code* равно нулю, если ошибка обнаружена (например, литерное значение переводится в цифровое), *Code* будет содержать номер позиции первого ошибочного символа, а значение *IBR* не определено.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение St* | *Выражение* | *Результат* |
| '1450'  '14.2Е+02'  '14.2А+02 | Val(St,IBR,Cod) Val(St,lBR,Cod) Val(St,IBR,Cod) | Code=0  Code=0  Code=5 |

***Copy (St,Poz,N)*** — выделяет из строки *St* подстроку длиной *N* символов, начиная с по­зиции *Poz.* Если *Poz > Length(St),* то результатом будет пробел; если *Poz > 255*, воз­никнет ошибка при выполнении. Функция *Length* описана ниже. *Poz, N* — цело­численные выражения.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение St* | *Выражение* | *Результат* |
| 'ABCDEFG' 'ABCDEFG' | Copy(St, 2, 3) Copy(St, 4, 10) | 'BCD'  'DEFG' |

***Concat (Strl,Str2,...,StrN)*** — выполняет сцепление строк *Strl, Str2,..,StrN* в том порядке, в каком они указаны в списке параметров. Сумма символов всех сцеплен­ных строк не должна превышать 255.

*Например:*

|  |  |
| --- | --- |
| *Выражение* | *Результат* |
| Concat('AA','XX','Y')  Соnсаt('Индекс','394063') | 'AAXXY' 'Индекс 394063' |

***Length (St)*** — вычисляет текущую длину в символах строки *St.* Результат имеет целочисленный тип.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение St* | *Выражение* | *Результат* |
| '123456789' 'System 370' | Length(St) Length(St) | 9  10 |

***Pos (Strl,Str2)*** — обнаруживает первое появление в строке *Str2* подстроки *Strl*. Результат имеет целочисленный тип и равен номеру той позиции, где находится первый символ подстроки *Strl.* Если в *Str2* подстроки *Strl* не найдено, результат равен 0.

*Например:*

|  |  |  |
| --- | --- | --- |
| *Значение Str1* | *Выражение* | *Результат* |
| 'abcdef '  'abcdef' | Pos('de',Strl) Pos('r',Strl) | 4  0 |

***UpCase (Ch)*** — преобразует строчную букву в прописную. Параметр и резуль­тат имеют литерный тип. Обрабатывает буквы только латинского алфавита.

|  |  |  |
| --- | --- | --- |
| *Значение Ch* | *Выражение* | *Результат* |
| 'd' | UpCase(Ch) | 'D' |

**Множества**

*Множество* – это структурированный тип данных, представляющий собой набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое. Каждый объект во множестве называется *элементом множества*.

Все элементы множества должны принадлежать одному из скалярных типов, кроме вещественного. Этот тип называется *базовым типом множества*. Базовый тип задается диапазоном или перечислением. *Область значений* типа множество – набо р всевозможных подмножеств, составленных из элементов базового типа.

В выражениях на языке Паскаль значения элементов множества указываются в квадратных скобках:[1,2,3,4], [‘a’,’b’,’c’], [‘a’..’z’]. Если множество не имеет элементов, оно называется *пустым* и обозначается, как [ ]. Количество элементов множества называется его *мощностью*. В отличие от массива порядок перечисления элементов во множестве не имеет значения, и количество элементов заранее не определено. Количество элементов, входящих во множество, может меняться от 0 до 256. Значения множества задаются в квадратных скобках перечислением элементов через запятую.

Например,

[ ]- пустое множество;

[2, 3, 7,11] - множество из целых чисел;

['а', 'с', Т]; - множество из символов;

[1..10] - множество из элемента ограниченного типа;

[k,.2\*k] - элемент множества задастся текущим значением переменной k.

***Формат записи множественных типов:***

*Type*

<имя типа> = **set** of <элемент 1,…., элементN>;

*Var*

<идентификатор, ….> : <имя типа>;

Можно задать множественный тип и без предварительного описания:

*Var*

<идентификатор, …> : set of <элемент1, …>;

Например,

*Type*

Simply = **set** of ‘a’ ..’h’;

Number = **set** of 1..31;

*Var*

Pr : Simply;

N : Number;

Letter : **set** of char; {определение множества без предварительного описания в разделе типов}

В данном примере переменная *Pr* может принимать значения символов латинского алфавита от ‘a’ до ‘h’; *N* – любое значение в диапазоне 1..31; *Letter* – любой символ. Попытка присвоить другие значения вызовет программное прерывание.

Количество элементов множества не должно превышать 256, соответственно номера значений базового типа должны находиться в диапазоне 0..255.

### Операции над множествами

При работе с множествами допускается использование операций отношения “=”, “<>”, “>=”, “<=”, объединения, пересечения, разности множеств и операции in. Результатом выражения с применением этих операций является значение *True* или *False*.

**Операция “равно” (=).** Два множества А и В считаются равными, если состоят из одних и тех же элементов. Порядок следования элементов в сравниваемых множествах значения не имеет.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3,4]  [‘a’, ’b’, ’c’]  [‘a’..’z’] | [1,2,3,4]  [‘c’, ‘a’]  [‘z’..’a’] | A=B  A=B  A=B | True  False  True |

**Операция “не равно” (<>).** Два множества А и В считаются не равными, если они отличаются по мощности или по значению хотя бы одного элемента.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3]  [‘a’..’z’]  [‘c’..’t’] | [3,1,2,4]  [‘b’.. ‘z’]  [‘t’..’c’] | A<>B  A<>B  A<>B | True  True  False |

**Операция “больше или равно” (>=).** Операция “больше или равно” (>=) используется для определения принадлежности множеств. Результат операции А >=В равен True, если все элементы множества В содержаться в множестве А. В противном случае результат равен False.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3,4]  [‘a’..’z’]  [‘z’,’x’,’c’] | [2,3,4]  [‘b’.. ‘t’]  [‘c’,’x’] | A>=B  A>=B  A>=B | True  True  True |

**Операция “меньше или равно” (<=).** Эта операция используется аналогично предыдущей операции, но результат выражения А<=В равен True, если все элементы множества А содержаться во множестве В. В противном случае результат равен False.

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3]  [‘d’..’h’]  [‘a’,’v’] | [1,2,3,4]  [‘z’.. ‘a’]  [‘a’,’n’,’v’] | A<=B  A<=B  A<=B | True  True  True |

**Операция in**. Операция in используется для проверки принадлежности какого-либо значения указанному множеству. Обычно применяется в условных операторах.

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* |  | *Результат* |
| 2  ‘v’  X1 | if A in [1,2,3] then..  if A in [‘a’..’n’] then..  if A in [X0,X1,X2,X3] then.. |  | True  True  True |

При использовании операции in проверяемое на принадлежность значение и множество в квадратных скобках не обязательно предварительно описывать в разделе описаний. Операция in позволяет эффективно и наглядно производить сложные проверки условий, заменяя иногда десятки других операций. Например, выражение if(a=1) or (a=2) or (a=3) or (a=4) or (a=5) or (a=6) then…можно заменить более коротким выражением

if a in [1..6] then… .

Часто операцию in пытаются записать с отрицанием: *X NOT in M.* Такая запись является ошибочной, так как две операции следуют подряд; правильная инструкция имеет вид*: NOT (X in M).*

**Объединение множеств (+).** Объединением двух множеств является третье множество, содержащее элементы обоих множеств.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3]  [‘A’..’D’]  [] | [1,4,5]  [‘E’.. ‘Z’]  [] | A+B  A+B  A+B | [1,2,3,4,5]  [‘A’..’Z’]  [] |

**Пересечение множеств (\*).** Пересечением двух множеств является третье множество, которое содержит элементы, входящие одновременно в оба множества.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3]  [‘A’..’Z’]  [] | [1,4,2,5]  [‘B’.. ‘R’]  [] | A\*B  A\*B  A\*B | [1,2]  [‘B’..’R’]  [] |

**Разность множеств (-).** Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество.

*Например:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Значение А* | *Значение В* | *Выражение* | *Результат* |
| [1,2,3,4]  [‘A’..’Z’]  [X1,X2,X3,X4]] | [3,4,1]  [‘D’.. ‘Z’]  [X4,X1] | A-B  A-B  A-B | [2]  [‘A’..’C’]  [X2,X3] |

Результат операций рад двумя множествами можно наглядно представить с помощью закрашенных частей двух кружочков:

###### E:\Безимени.bmp **Объединение Пересечение Разность**

Использование в программе данных типа **set** дает ряд преимуществ: значительно упрощаются сложные операторы if , увеличивается степень наглядности программы и понимания алгоритма решения задачи, экономятся память, время компиляции и выполнения.

Имеются и отрицательные моменты, основной из них – отсутствие в языке Паскаль средств ввода-вывода элементов множества, поэтому программист сам должен писать соответствующие процедуры.

***Пример 1.***

**Program** Dem\_Mno; {демонстрация операций над множествами}

**Type**

Digits=**set** of 0..9;

**Var**

D1, D2, D3, D:Digits;

**Begin**

D1:=[2,4,6,8]; {заполнение множеств}

D2:=[0..3,5];

D3:=[1,3,5,7,9];

D:=D1+D2; {объединение множеств D1 и D2}

D:=D+D3; {объединение множеств D и D3 }

D:=D-D2; {разность множеств D и D2 }

D:=D\*D1; {пересечение множеств D и D1}

**end**.

Сначала описан тип *Digits=set of 0..9,* затем описаны переменные *D1,D2,D3,D* этого типа. В первой части программы осуществляется заполнение множеств, а затем над множествами выполняются операции объединения, пересечения, разности.

***Пример 2.*** *Описать множество М (1..50) и сделать его пустым. Вводя целые числа с клавиатуры, заполнить множество 10 элементами.*

**Program** Input\_Mno;

Var

M:**set** of 1..50;

X,I:integer;

**Begin**

M:= [ ]; {М – пустое множество}

**for** I:=1 **to** 10 **do**

**begin**

write(‘введите ‘, I, ‘ –й элемент множества: ‘);

readln (X);

**if** (X **in** M) **then**  {если введенное число входит в множество М}

**begin**

writeln(X,’ помещен в множество 1..50’);

M:=M+[X];

**end;**

**end;**

writeln;

**end.**

В разделе описания переменных описано множество целых чисел от 1 до 50, переменная Х целого типа, которая используется для считывания числа-кандидата в множество, и целая переменная I, используемая для подсчета количества введенных чисел. В начале программы применена операция инициализации множества М, так как оно не имеет элементов и является пустым:

M:= [ ];

Заполнение множества элементами производится с использованием оператора повтора for, параметр которого I будет указывать порядковый номер вводимого элемента. Операция заполнения множества записывается оператором присваивания:

M:=M+[X];

Контроль заполнения множества записан операцией проверки принадлежности in. Если условие X in M выполняется, выводится сообщение о том, что число Х помещено в множество.

***Пример 3.*** Описывается множество гласных и согласных букв русского языка и определяется количество гласных и согласных букв в предложении, введенном с клавиатуры пользователем.

**Program** Glasn\_Sogl;

Type

Letters=set of ‘A’..’я’;

Var

Glasn, Sogl:Letters;

Text:String;

I:Byte;

G,S:Byte;

Begin

Glasn:=[‘A’,’a’, ‘Е’,’е’,’И’,’и’,’О’,’о’,’У’,’у’,’Э’,’Ю’,’ю’, ‘Я’,’я’];

Sogl:=[‘Б’..’Д’,’б’..’д’,’Ж’,’ж’,’З’,’з’,’К’..’Н’,’к’..’н’,’П’..’Т’,’п’..’т’,’Ф’..’Щ’,’ ‘ф’..’щ’,’Ъ’,’ъ’,’Ь’,’ь’];

Write(‘Введите предложение’);

Readln(Text);

G:=0;

S:=0;

**For** I:=1 **to** Length(Text) **do**

Begin

**If** Text[I] **in** Glasn then G:=G+1;

**If** Text[I] **in** Sogl then S:=S+1;

**End;**

Writeln(‘В предложении” ’,Text,’ ”‘,G,’гласных и ‘,S,’согласных букв’);

**End.**

Зададим тип Letters –множество букв русского языка, затем опишем переменные этого типа: Glasn-множество гласных букв, Sogl-множество согласных букв. Вводимое с клавиатуры предложение опишем переменной Text типа String. Для указания символа в строке Text применим переменную I типа Byte.Для подсчета количества гласных и согласных букв опишем переменные Gи S. Проверку принадлежности символов, составляющих предложение множествам гласных или согласных букв русского языка запишем с использованием цикла for , параметр I которого, изменяясь от 1 до значения длины предложения, будет указывать порядковый номер символа в предложении. Принадлежность очередного символа предложения множеству гласных или согласных букв запишем операцией in. Если символ является гласной буквой (Text[I] in Glasn), то счетчик гласных букв G увеличивается на 1. Аналогично с согласными буквами.

## 2.5 Подпрограммы

Подпрограмма – именованная, логически законченная группа операторов языка, которую можно вызвать для выполнения любое количество раз из разных мест программы. В языке Паскаль существует два вида подпрограмм: процедуры и функции.

**Процедуры**

Процедура – это подпрограмма, которая решает некоторую частную задачу или объединяет группу часто встречающихся операторов. Каждая процедура должна быть предварительно описана в разделе описаний процедур и функций. Описание процедуры состоит из заголовка и тела процедуры. Описание процедуры содержит служебное слово procedure, имя процедуры и заключенный в скобки список формальных параметров с указанием их типов. Формальные параметры отделяются точкой с запятой (список однотипных параметров может быть перечислен через запятую).

Procedure <имя процедуры> (<список формальных параметров>);

После заголовка идут разделы описаний (констант, типов, переменных, процедур и функций, используемых в процедуре) и операторы языка Паскаль, реализующие алгоритм процедуры.

const ... ;

type . . . ;

var . . . ;

begin

<операторы>

end ;

Формальные параметры фиктивно присутствуют в процедуре и определяют тип и действия, которые над ними производятся. Их нельзя описывать в разделе описаний процедуры.

Во время выполнения формальные параметры заменяются фактическими параметрами. Фактические параметры – это параметры, которые передаются процедуре при обращении к ней.

Формальные параметры делятся на параметры – значения и параметры - переменные. Параметры – значения используются в качестве входных данных подпрограммы, при обращении к которой фактические параметры передают свое значение формальным и больше не меняются. Параметры-значения задаются следующим образом:

<Имя параметра>:< тип параметра>.

Параметры – переменные могут использоваться как в качестве входных, так и в качестве выходных. В заголовке процедуры перед ними необходимо указывать служебное слово var:

Var <имя параметра>:<тип параметра>

При обращении к подпрограмме фактические параметры замещают формальные. В результате выполнения подпрограммы изменяются фактические параметры.

***Вызов процедуры.*** Для обращения к процедуре необходимо использовать оператор вызова процедуры. Он имеет следующий вид:

<имя процедуры> (<список фактических параметров>);

При вызове процедуры основная программа приостанавливает свою работу и передает управление в процедуру. При завершении процедуры передается управление на команду, следующую за вызовом процедуры.

Количество, типы и порядок следования формальных и фактических параметров должны совпадать.

***Пример.*** Подпрограмма должна заполнить массив исходными данными.

Program proba;

Const k=100

Type Tm=array[1…k] of integer;

Var M:TM ; i; fk; byte;

Procedure InputMas (Var M:TM ; i; fk; byte;)

var i: byte;

begin

for i:=1 to10 do

begin

Mas[i]:=round(random(50)-25);

{= = = = = = = = = = = = = = = = = = = = = = =}

begin

write(‘Введите количество элементов’);

repeat read (fk); until fk<=k;

inputMas(M,fk);

for i:=1 to fk do

write(M[i]:5);

end.

**Функции**

Функция, определенная пользователем, состоит из *заголовка* и тела *функции*.

*Заголовок* содержит зарезервированное слово *function,* имя функции, заключенное в круглые скобки, необязательный список формальных параметров и тип возвращаемого функцией значения. Тело функции представляет собой локальный блок, по структуре аналогичный программе. В целом структура функции, определенной пользователем имеет вид:

*function <имя> (формальные параметры) : <тип результата>;*

*const ...*

*type ...*

*var*

*begin*

*<операторы>*

*end;*

В разделе операторов должен находиться, по крайней мере, один оператор, *присваивающий имени функции значение*. В точку вызова возвращается результат последнего присваивания.

Обращение к функции осуществляется по имени с необязательным указанием списка аргументов. Каждый аргумент должен соответствовать формальным параметрам, указанным в заголовке, и иметь тот же тип.

### *Пример.* Составить программу вычисления выражения Z=(+)/2\*, в которой возведение в степень выполняется *функцией Step*.

program DemoFunc;

Var

М : integer;

А,Z,R : real ;

*{Функция вычисления степени. N - степень, X – число, возводимое в данную степень. N, X — формальные параметры; результат, возвращаемый функцией в точку вызова, имеет вещественный тип}*

function Step(N : integer; X : real): real;

Var

I : integer;

Y *:* real;

begin

Y:=1;

for I:=1 to N do*{Цикл вычисления N—й степени числа X)*

Y:=Y\*X;

Step:=Y ; *{Присваивание функции результата вычисления степени}*

end; *{Конец функции}*

Begin *{Начало основной программы}*

Write('Введите значение числа А и показатель степени М');

Readln(A,M) ;

Z:=Step(5,А) ; *{Вызов функции с передачей ей фактических параметров N=5, X=А}*

Z:=Z+ Step(3,l/A); *{Вызов функции с передачей ей фактических параметров N=3, X=1/А}*

if M=0 then R:=l *{если число возводится в нулевую степень, то результат всегда равен 1}*

else if M>0 then R:=Step(M,A)*{Вызов функции Step с передачей ей фактических параметров М, А}*

else R:=Step(-M,A); {*Вызов функции с передачей ей фактических параметров: - М, отрицательная степень}*

Z:=Z/(2\*R) ;

Writeln(' Для А= ',А,'М= ',М,' Значение выражения= ',Z);

end.

В начале программы описываются переменная целого типа *М* и переменные вещественного типа *А, Z, R*, после этого описывается функция вычисления степени числа *Step* с формальными параметрами *N* и *X,* результат, возвращаемый функцией в точку вызова, - вещественного типа.

В описании функции вводятся две *локальных (местных) переменных* *I* и *Y*. Переменная *I* служит для подсчета числа повторений цикла, а в *Y* накапливается значение степени как произведения *N* одинаковых сомножителей. В заключение функции  присваивается значение вычисленного произведения.

В начале выполнения основной программы на экран выводится запрос *"Введите значение числа А и показатель степени М"* и считывается с клавиатуры значение вещественного числа А и целого числа М.

Затем выполняется оператор:

*Z:=Step(5,A);*

Осуществляется вызов функции *Step* с передачей ей фактических параметров *5, А*. Их значения присваиваются формальным параметрам функции *N* и *X*. По окончании вычисления степени числа значение функции *Step*, вычисленное для фактических параметров *5* и *А*, присваивается переменной *Z*. Аналогично в операторе:

*Z := Z + Step(3,l/A);*

сначала осуществляется вызов функции *Step* с передачей ей фактических параметров *3, 1/A,* после чего значение переменной *Z* увеличивается на величину возвращенного в основную программу результата вычисления функции *Step*.

Операторы:

*if M=0 then R:=1*

*else if M>0 then R:=Step(M,A)*

*else R:=Step(- M,A);*

проверяют условия *М=0, М>0* и в зависимости от их соблюдения либо при­сваивает переменной *R* значение 1 *(при М=О)*, либо выполняет вызов функции *Step* для *фактических значений М, А* или *-М, А*, а после вычисления значения функции *Step* присваивает его переменной R.

Оператор:

*Z:=Z/(2\*R);*

выполняет вычисление значения выражения, а затем присваивает вычисленное значение переменной *Z*.

В заключение программы стандартная процедура *Writeln* выводит на экран сообщение о результате вычислений степени *М* числа *А*.

## Файлы

Существенной особенностью всех рассмотренных до сих пор значений производных типов является наличие в них конечного, наперед заданного числа компонент. Так, в значении многомерного массива это число можно определить, зная количество компонент по каждому измерению, а в значении записи это число определяется количеством и типом полей. Таким образом, заранее, еще до выполнения программы, по этому описанию можно выделить необходимый объем памяти машины для хранения значений переменных этих типов. Но существует определенный класс задач и определенные ситуации, когда количество компонент (пусть даже одного и того же из известных уже типов) заранее определить невозможно, оно выясняется только в процессе решения задачи. Поэтому возникает необходимость в специальном типе значений, которые представляют собой произвольные последовательности элементов одного и того же типа, причем длина этих последовательностей заранее не определяется, а конкретизируется в процессе выполнения программы. Этот тип значений получил название файлового типа. Условно файл в Паскале можно изобразить как некоторую ленту, у которой есть начало, а конец не фиксируется. Элементы файла записываются на эту ленту последовательно друг за другом:



где F – имя файла, а F1, F2, F3, F4 – его элементы. Файл во многом напоминает магнитную ленту, начало которой заполнено записями, а конец пока свободен. В программировании существует несколько разновидностей файлов, отличающихся методом доступа к его компонентам: файлы последовательного доступа и файлы произвольного доступа.

Простейший метод доступа состоит в том, что по файлу можно двигаться только последовательно, начиная с первого его элемента, и, кроме этого, всегда существует возможность начать просмотр файла с его начала. Таким образом, чтобы добраться до пятого элемента файла, необходимо, начав с первого элемента, пройти через предыдущие четыре. Такие файлы называют файлами последовательного доступа. У последовательного файла доступен всегда лишь очередной элемент. Если в процессе решения задачи необходим какой-либо из предыдущих элементов, то необходимо вернуться в начало файла и последовательно пройти все его элементы до нужного.

Файлы произвольного доступа Паскаля позволяют вызывать компоненты в любом порядке по их номеру.

Важной особенностью файлов является то, что данные, содержащиеся в файле, переносятся на внешние носители. Файловый тип Паскаля – это единственный тип значений, посредством которого данные, обрабатываемые программой, могут быть получены извне, а результаты могут быть переданы во внешний мир. Это единственный тип значений, который связывает программу с внешними устройствами ЭВМ.

**Работа с файлами в Паскале**

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что дает возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Паскаля, кроме файлов. Иными словами, нельзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа в Паскале можно задать одним из трех способов:

Type <имя\_ф\_типа>=file of<тип\_элементов>;

<имя\_ф\_типа>=text;

<имя\_ф\_типа>=file;

Здесь <имя\_ф\_типа> – имя файлового типа (правильный идентификатор); File, of – зарезервированные слова (файл, из); <тип\_элементов> – любой тип Паскаля, кроме файлов.

*Пример описания файлового типа в Паскале*

Type

Product= record

Name: string;

Code: word;

End;

Text80= file of string[80];

Var

F1: file of char;

F2: text;

F3: file;

F4: Text80;

F5: file of Product;

В зависимости от способа объявления можно выделить три вида файлов Паскаля:

- типизированные файлы Паскаля(задаются предложением file of..);

- текстовые файлы Паскаля(определяются типом text);

- нетипизированные файлы Паскаля(определяются типом file).

Следует помнить, что физические файлы на магнитных дисках и переменные файлового типа в программе на Паскале – объекты различные. Переменные файлового типа в Паскале могут соответствовать не только физическим файлам, но и логическим устройствам, связанным с вводом/выводом информации. Например, клавиатуре и экрану соответствуют файлы со стандартными именами Input, Output.

Каждый тип данных в Паскале, вообще говоря, определяет множество значений и множество операций над значениями этого типа. Однако над значениями файлового типа Паскаля не определены какие-либо операции, в том числе операции отношения и присваивания, так что даже такое простое действие, как присваивание значения одной файловой переменной другой файловой переменной, имеющей тот же самый тип, запрещено. Все операции могут производиться лишь с элементами (компонентами) файлов. Естественно, что множество операций над компонентами файла определяется типом компонент.

Переменные файлового типа используются в программе только в качестве параметров собственных и стандартных процедур и функций.

**Основные процедуры и функции для работы с файлами**

1. До начала работы с файлами в Паскале необходимо установить связь между файловой переменной и именем физического дискового файла:

Assign(<файловая\_переменная>, <имя\_дискового\_файла>)

Следует помнить, что имя дискового файла при необходимости должно содержать путь доступа к этому файлу, включая имя дисковода. При этом имя дискового файла – строковая величина, т.е. должна быть заключена в апострофы. Например:

Assign (chf, 'G:\Home\ Student\ Lang\ Pascal\ primer.dat');

Если путь не указан, то программа будет искать файл в своем рабочем каталоге и по указанным путям в autoexec.bat.

Вместо имени дискового файла можно указать имя логического устройства, каждое из которых имеет стандартное имя:

CON – консоль, т.е. клавиатура-дисплей;

PRN – принтер. Если к компьютеру подключено несколько принтеров, доступ к ним осуществляется по именам LPT1, LPT2, LPT3.

Не разрешается связывать с одним физическим файлом более одной файловой переменной.

1. После окончания работы с файлами на Паскале, они должны быть закрыты.

Close(<список файловых переменных>);

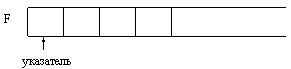
При выполнении этой процедуры закрываются соответствующие физические файлы и фиксируются сделанные изменения. Следует иметь в виду, что при выполнении процедуры close связь файловой переменной с именем дискового файла, установленная ранее процедурой assign, сохраняется, следовательно, файл можно повторно открыть без дополнительного использования процедуры assign.

Работа с файлами заключается, в основном, в записи элементов в файл и считывании их из файла. Для удобства описания этих процедур введем понятие указателя, который определяет позицию доступа, т.е. ту позицию файла, которая доступна для чтения (в режиме чтения), либо для записи (в режиме записи). Позиция файла, следующая за последней компонентой файла (или первая позиция пустого файла) помечается специальным маркером, который отличается от любых компонент файла. Благодаря этому маркеру определяется конец файла.

1. Подготовка к записи в файл Паскаля

Rewrite(<имя\_ф\_переменной>);

Процедура Rewrite(f) (где f – имя файловой переменной) устанавливает файл с именем f в начальное состояние режима записи, в результате чего указатель устанавливается на первую позицию файла. Если ранее в этот файл были записаны какие-либо элементы, то они становятся недоступными. Результат выполнения процедуры rewrite(f); выглядит следующим образом:

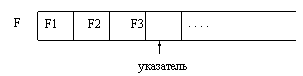


1. Запись в файл Паскаля

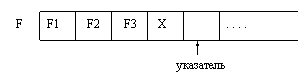
Write(<имя\_ф\_переменной>, <список записи>);

При выполнении процедуры write(f, x) в ту позицию, на которую показывает указатель, записывается очередная компонента, после чего указатель смещается на следующую позицию. Естественно, тип выражения х должен совпадать с типом компонент файла. Результат действия процедуры write(f, x) можно изобразить так:

Состояние файла f до выполнения процедуры



Состояние файла f после выполнения процедуры

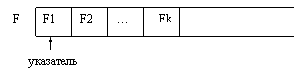


Для типизированных файлов выполняется следующее утверждение: если в списке записи перечислено несколько выражений, то они записываются в файл, начиная с первой доступной позиции, а указатель смещается на число позиций, равное числу записываемых выражений.

1. Подготовка файла к чтению Паскаля

Reset(<имя\_ф\_переменной>);

Эта процедура ищет на диске уже существующий файл и переводит его в режим чтения, устанавливая указатель на первую позицию файла. Результат выполнения этой процедуры можно изобразить следующим образом:



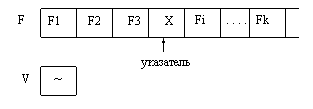
Если происходит попытка открыть для чтения не существующий еще на диске файл, то возникает ошибка ввода/вывода, и выполнение программы будет прервано.

1. Чтение из файла в Паскале

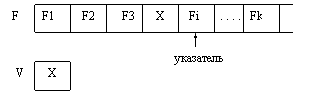
Read(<имя\_ф\_переменной>,<список переменных>);

Рассмотрим результат действия процедуры read(f, v):

Состояние файла f и переменной v до выполнения процедуры:



Состояние файла f и переменной v после выполнения процедуры:



Для типизированных файлов при выполнении процедуры read() последовательно считывается, начиная с текущей позиции указателя, число компонент файла, соответствующее числу переменных в списке, а указатель смещается на это число позиций.

В большинстве задач, в которых используются файлы, необходимо последовательно перебрать компоненты и произвести их обработку. В таком случае необходимо иметь возможность определять, указывает ли указатель на какую-то компоненту файла, или он уже вышел за пределы файла и указывает на маркер конца файла.

1. Функция определения достижения конца файла в Паскале

Eof(<имя\_ф\_переменной>);

Название этой функции является сложносокращенным словом от end of file. Значение этой функции имеет значение true, если конец файла уже достигнут, т.е. указатель стоит на позиции, следующей за последней компонентой файла. В противном случае значение функции – false.

1. Изменение имени файла в Паскале

Rename(<имя\_ф\_переменной>, <новое\_имя\_файла>);

Здесь новое\_ имя\_ файла – строковое выражение, содержащее новое имя файла, возможно с указанием пути доступа к нему.

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

1. Уничтожение файла в Паскале

Erase(<имя\_ф\_переменной>);

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

1. Уничтожение части файла от текущей позиции указателя до конца в Паскале

Truncate(<имя\_ф\_переменной>);

1. Файл Паскаля может быть открыт для добавления записей в конец файла

Append(<имя\_ф\_переменной>);

**Типизированные файлы Паскаля**

Длина любого компонента типизированного файла строго постоянна, т.к. тип компонент определяется при описании, а, следовательно, определяется объем памяти, отводимый под каждую компоненту. Это дает возможность организовать прямой доступ к каждой компоненте (т.е. доступ по порядковому номеру).

Перед первым обращением к процедурам ввода/вывода указатель файла стоит в его начале и указывает на его первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные и выражения в списках ввода и вывода в процедурах read() и write() должны иметь тот же тип, что и компоненты файла Паскаля. Если этих переменных или выражений в списке несколько, то указатель будет смещаться после каждой операции обмена данными на соответствующее число позиций.

Для облегчения перемещения указателя по файлу и доступа к компонентам типизированного файла существуют специальные процедуры и функции:

- функция Паскаля, определяющая число компонентов в файле:

fileSize(<имя\_ф\_переменной>)

– функция Паскаля, значением которой является текущая позиция указателя:

filePos(<имя\_ф\_переменной>)

– процедура Паскаля, смещающая указатель на компоненту файла с номером n:

seek(<имя\_ф\_переменной>,n)

Так, процедура seek(<имя\_ф\_переменной>,0) установит указатель в начало файла, а процедура seek(<имя\_ф\_переменной>, FileSize(<имя\_ф\_переменной>)) установит указатель на признак конца файла.

**Текстовые файлы Паскаля**

Текстовые файлы предназначены для хранения текстовой информации. Именно в таких файлах хранятся, например, исходные тексты программ. Компоненты текстовых файлов могут иметь переменную длину, что существенно влияет на характер работы с ними. Доступ к каждой строке текстового файла Паскаля возможен лишь последовательно, начиная с первой. К текстовым файлам применимы процедуры assign, reset, rewrite, read, write и функция eof. Процедуры и функции seek, filepos, filesize к ним не применяются. При создании текстового файла в конце каждой записи (строки) ставится специальный признак EOLN(end of line – конец строки). Для определения достижения конца строки существует одноименная логическая функция EOLN(<имя\_ф\_переменной>), которая принимает значение true, если конец строки достигнут.

Форма обращения к процедурам write и read для текстовых и типизированных файлов одинакова, но их использование принципиально различается.

В списке записываемых в текстовый файл элементов могут чередоваться в произвольном порядке числовые, символьные, строковые выражения. При этом строковые и символьные элементы записываются непосредственно, а числовые из машинной формы автоматически преобразуются в строку символов.

текстовые файлы удобнее для восприятия человеком, а типизированные соответствуют машинному представлению объектов;

текстовые файлы, как правило, длиннее типизированных;

длина текстовых файлов зависит не только от количества записей, но и от величины переменных.

Так, в типизированном файле числа 6, 65 и 165 как целые будут представлены одним и тем же числом байт. А в текстовых файлах, после преобразования в строку, они будут иметь разную длину. Это вызывает проблемы при расшифровке текстовых файлов. Пусть в текстовый файл пишутся подряд целые числа (типа byte): 2, 12, 2, 128. Тогда в файле образуется запись 2122128. При попытке прочитать из такого файла переменную типа byte программа прочитает всю строку и выдаст сообщение об ошибке, связанной с переполнением диапазона.

Чтобы избежать этой ошибки, достаточно вставить при записи в файл после каждой переменной пробел. Тогда программа при каждом чтении берет символы от пробела до пробела и правильно преобразует текстовое представление в число.

Кроме процедур read и write при работе с текстовыми файлами используются их разновидности readln и writeln. Отличие заключается в том, что процедура writeln после записи заданного списка записывает в файл специальный маркер конца строки. Этот признак воспринимается как переход к новой строке. Процедура readln после считывания заданного списка ищет в файле следующий признак конца строки и подготавливается к чтению с начала следующей строки.

***Пример.*** *Пусть нам необходимо сформировать текстовый файл с помощью Паскаля, а затем переписать из данного файла во второй только те строки, которые начинаются с буквы «А» или «а».*

Пояснения: нам понадобятся две файловые переменные f1 и f2, поскольку оба файла текстовые, то тип переменных будет text. Задача разбивается на два этапа: первый – формирование первого файла; второй – чтение первого файла и формирование второго.

Для завершенности решения задачи есть смысл добавить еще одну часть, которая в задаче явно не указана – вывод на экран содержимого второго файла.

Program primer;

Var f1,f2:text;

I,n: integer;

S: string;

Begin

{формируем первый файл}

Assign(f1, ‘file1.txt’); {устанавливаем связь файловой переменной с физическим файлом на диске}

Rewrite(f1); {открываем файл для записи}

Readln(n) {определим количество вводимых строк}

for i:=1 to n do

begin

readln(s); {вводим с клавиатуры строки}

writeln(f1,s); {записываем последовательно строки в файл}

end;

close(f1); {заканчиваем работу с первым файлом, теперь на диске существует файл с именем file1.txt, содержащий введенные нами строки. На этом программу можно закончить, работу с файлом можно продолжить в другой программе, в другое время, но мы продолжим}

{часть вторая: чтение из первого файла и формирование второго}

Reset(f1); {открываем первый файл для чтения}

Assign(f2, ‘file2.txt’); {устанавливаем связь второй файловой переменной с физическим файлом}

Rewrite(f2); {открываем второй файл для записи}

{Дальше необходимо последовательно считывать строки из первого файла, проверять выполнение условия и записывать нужные строки во второй файл. Для чтения из текстового файла рекомендуется использовать цикл по условию «пока не конец файла»}

While not eof(f1) do

Begin

Readln(f1,s);{считываем очередную строку из первого файла}

If (s[1]=’A’) or (s[1]=’a’) then

Writeln(f2,s); {записываем во второй файл строки, удовлетворяющие условию}

End;

Close(f1,f2); {заканчиваем работу с файлами}

{часть третья: выводим на экран второй файл}

Writeln;

Writeln(‘Второй файл содержит строки:’);

Reset(f2); {открываем второй файл для чтения}

While not eof(f2) do {пока не конец второго файла}

Begin

Readln(f2,s);{считываем очередную строку из второго файла}

Writeln(s); {выводим строку на экран}

End;

End.

## 2.7 Указатели и динамическая память

**Динамические структуры данных**

Объект данных обладает динамической структурой, если его размер изменяется в процессе выполнения программы или он потенциально бесконечен.

**Классификация структур данных**

Используемые в программировании данные можно разделить на две большие группы:

Данные

## Данные

статической структуры

## Данные

динамической структуры

*Данные статической структуры* – это данные, взаиморасположение и взаимосвязи элементов которых всегда остаются постоянными.

*Данные динамической структуры* – это данные, внутреннее строение которых формируется по какому-либо закону, но количество элементов, их взаиморасположение и взаимосвязи могут динамически изменяться во время выполнения программы, согласно закону формирования.

Данные динамической структуры

Связанные динам. структуры

Файлы

Текстовые

Типизированные

Нетипизированные

Несвязанные динам. структуры

Классифицируются аналогично данным статической структуры

Линейной структуры

Кольцевой структуры

Разветвленной структуры

#### Односвязные

Очередь

#### Стек

Дек

Список

#### Многосвязные

Многосвязный линейный список

Односвязный кольцевой список

Многосвязный кольцевой список

Деревья

#### Графы

Двоичные (бинарные)

#### Разветвленные

К данным динамической структуры относят файлы, несвязанные и связанные динамические данные.

Файлы в данной классификации отнесены к динамическим структурам данных. Это сделано исходя из вышеприведенного определения. Хотя удаление и вставка элементов в середину файла не допускается, зато длина файла в процессе работы программы может изменяться – увеличиваться или уменьшаться до нуля. А это уже динамическое свойство файла как структуры данных.

**Статические и динамические переменные в Паскале**

В Паскале одной из задач описания типов является то, чтобы зафиксировать на время выполнения программы размер значений, а, следовательно, и размер выделяемой области памяти для них. Описанные таким образом переменные называются *статическими*.

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти – сегмент данных. Длина сегмента данных определяется архитектурой микропроцессора и составляет обычно 65536 байт.

Однако порой заранее не известны не только размеры значений, но и сам факт существования значения той или иной переменной. Для результата переменной приходится отводить память в расчете на самое большое значение, что приводит к нерациональному использованию памяти. Особенно это затруднительно при обработке больших массивов данных.

Предположим, например, что у вас есть программа, требующая массива в 400 строк по 100 символов каждая. Для этого массива требуется примерно 40К, что меньше максимума в 64К. Если остальные ваши переменные помещаются в оставшиеся 24К, массив такого объема проблемы не представляет.

Но что если вам нужно два таких массива? Это потребовало бы 80К, и 64К сегмента данных не хватит.

Другим общим примером является сортировка. Обычно когда вы сортируете большой объем данных, то делаете копию массива, сортируете копию, а затем записываете отсортированные данные обратно в исходный массив. Это сохраняет целостность ваших данных, но требует также наличия во время сортировки двух копий данных.

С другой стороны объем памяти компьютера достаточно велик для успешного решения задач с большой размерностью данных. Выходом из положения может служить использование так называемой *динамической памяти.*

*Динамическая память (ДП)* – это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кб), стека (16 Кб) и собственно тела программы. Размер динамической памяти можно варьировать. По умолчанию ДП – вся доступная память ПК.

ДП – это фактически единственная возможность обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования ДП. Например, при разработке САПР статическое распределение памяти невозможно, т.к. размерность математических моделей в разных проектах может значительно различаться.

И статические и динамические переменные вызываются по их адресам. Без адреса не получить доступа к нужной ячейке памяти, но при использовании статических переменных, адрес непосредственно не указывается. Обращение осуществляется по имени. Компилятор размещает переменные в памяти и подставляет нужные адреса в коды команд.

Адресация динамических переменных осуществляется через *указатели*. Их значения определяют адрес объекта.

Для работы с динамическими переменными в программе должны быть выполнены следующие действия:

* Выделение памяти под динамическую переменную;
* Инициализация указателя;
* Освобождение памяти после использования динамической переменной.

Программист должен сам резервировать место, определять значение указателей, освобождать ДП.

Вместо любой статической переменной можно использовать динамическую, но без реальной необходимости этого делать не стоит.

**Указатели**

Для работы с динамическими программными объектами в Паскале предусмотрен *ссылочный тип* или *тип указателей*. В переменной ссылочного типа хранится ссылка на программный объект (адрес объекта).

*Указатель* – это переменная, которая в качестве своего значения содержит адрес байта памяти.

***Объявление указателей***

Указатель, связанный с некоторым определенным типом данных, называют *типизированным указателем*. Его описание имеет вид:

Имя\_переменной: ^ базовый-тип;

Например:

Type A= array [1..100] of integer;

TA= ^A ; {тип указатель на массив}

Var

P1: ^integer; {переменная типа указатель на целое число}

P2: ^ real; {переменная типа указатель на вещественное число}

Указатель, не связанный с каким-либо конкретным типом данных, называется *нетипизированным указателем*. Для описания нетипизированного указателя в Паскале существует стандартный тип ***pointer***. Описание такого указателя имеет вид:

Имя-переменной: pointer;

С помощью нетипизированных указателей удобно динамически размещать данные, структура и тип которых меняются в ходе выполнения программы.

Значения указателей – адреса переменных в памяти. Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе – смещение.

Следовало бы ожидать, что значение одного указателя можно передать другому. На самом деле можно передавать значения только между указателями, связанными с одним типом данных. Указатели на различные типы данных имеют различный тип, причем эти типы несовместимы.

Например,

Var p1,p2: ^integer;

p3: ^real;

pp: pointer;

………

p1:= p2; {допустимое действие}

p1:= p3; {недопустимое действие}

Однако это ограничение не распространяется на нетипизированный указатель. В программе допустимы будут следующие действия:

pp:= p3;

p1:= pp;

***Выделение и освобождение динамической памяти***

Вся ДП рассматривается как сплошной массив байтов, который называется *кучей*.

Расположение кучи в памяти ПК.

Системная область

Куча

Программа

Системная область

##### Старшие адреса

##### Heapend

Heapptr

Heaporg

Младшие адреса

Существуют стандартные переменные, в которых хранятся значения адресов начала, конца и текущей границы кучи:

Heaporg – начало кучи;

Heapend – конец кучи;

Heapptr – текущая граница незанятой ДП.

*Выделение памяти под динамическую переменную* осуществляется процедурой:

New (переменная\_типа\_указатель)

В результате обращения к этой процедуре указатель получает значение, соответствующее адресу в динамической памяти, начиная с которого можно разместить данные.

Например,

Var i, j: ^integer;

r: ^real;

begin

new(i); {после этого указатель i приобретает значение адреса  
 Heapptr, а Heapptr смещается на 2 байта}

……………

new(r) ; {r приобретает значение Heapptr, а Heapptr смещается  
 на 6 байт}

Графически действие процедуры new можно изобразить так:

Указатель

Динам. переменная

Здесь хранится адрес переменной

В этой области памяти будет храниться значение переменной

***Освобождение динамической памяти*** осуществляется процедурой:

Dispose (переменная\_типа\_указатель)

Например:

Dispose (i); {возвращает в кучу 2 байта}

Dispose (r); {возвращает в кучу 6 байт}

Следует помнить, что повторное применение процедуры dispose к свободному указателю может привести к ошибке.

Процедура dispose освобождает память, занятую динамической переменной. При этом значение указателя становится неопределенным.

Указатель

Динам. переменная

Любые действия над указателем в программе располагаются между процедурами new и dispose.

При использовании динамически распределяемых переменных часто возникает общая проблема, называемая утечкой динамической памяти. Утечка памяти – это ситуация, когда пространство выделяется в динамически распределяемой памяти и затем теряется – по каким-то причинам ваш указатель не указывает больше на распределенную область, так что вы не можете освободить пространство.

Общей причиной утечек памяти является переприсваивание динамических переменных без освобождения предыдущих. Простейшим случаем является следующий:

var IntPointer: ^Integer;

begin

New(IntPointer);

New(IntPointer);

end.

При первом вызове New в динамически распределяемой памяти выделяется 2 байта, и на них устанавливается указатель IntPointer. Второй вызов New выделяет другие 2 байта, и IntPointer устанавливается на них. Теперь у вас нет указателя, ссылающегося на первые 2 байта, поэтому вы не можете их освободить. В программе эти байты будут потеряны.

***Присваивание значений указателю***

Для инициализации указателей существует несколько возможностей.

1. процедура new отводит блок памяти в области динамических переменных и сохраняет адрес этой области в указателе;
2. специальная операция **@** ориентирует переменную-указатель на область памяти, содержащую уже существующую переменную. Эту операцию можно применять для ориентации на статическую и динамическую переменную.  
   Например,  
   type A= array [0..99] of char;  
   var X: array [0..49] of integer;  
   pA: ^A; {указатель на массив символов}   
   Объявлены переменные разных типов: массив из 50 целых чисел и указатель на массив символов. Чтобы указатель pA указывал на массив X, надо присвоить ему адрес X

pA:= @X;

1. Существует единственная константа ссылочного типа ***nil***, которая обозначает «пустой» адрес. Ее можно присваивать любому указателю.
2. Переменной-указателю можно присвоить значение другого указателя того же типа. Используя указательный тип pointer как промежуточный, можно присвоить значение одного указателя другому при несовпадении типов.

***Операции с указателями***

Для указателей определены только операции присваивания и проверки на равенство и неравенство. В Паскале запрещаются любые арифметические операции с указателями, их ввод-вывод и сравнение на больше-меньше.

Еще раз повторим правила присваивания указателей:

* любому указателю можно присвоить стандартную константу nil, которая означает, что указатель не ссылается на какую-либо конкретную ячейку памяти;
* указатели стандартного типа pointer совместимы с указателями любого типа;
* указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа данных.

Указатели можно сравнивать на равенство и неравенство, например:

If p1=p2 then …..

If p1<>p2 then …..

В Паскале определены стандартные функции для работы с указателями:

* addr(x) – тип результата pointer, возвращает адрес x (аналогично операции @), где x – имя переменной или подпрограммы;
* seg(x) – тип результата word, возвращает адрес сегмента для x;
* ofs(x) – тип результата word, возвращает смещение для x;
* ptr(seg, ofs) – тип результата pointer, по заданному сегменту и смещению формирует адрес типа pointer.

***Присваивание значений динамическим переменным***

После того, как динамическая переменная объявлена, ей можно присваивать значения, изменять их, использовать в выражениях и т.д. Для этого используют следующее обращение: переменная\_указатель^. Такое обращение называется *операция разадресации (разыменования)*. Таким образом происходит обращение к значению, на которое указывает указатель, т.е. к данным. Если же за переменной\_указателем значок ^ не стоит, то имеется в виду адрес, по которому расположены данные.

Динамически размещенные данные можно использовать в любом месте программы, где допустимо использование выражений соответствующего типа.

Например:

R^:= sqr(R^) + I^ -17;

q^:= 2; inc(q^); writeln(q^)

Рассмотрим пример работы с указателями:

P Q

Var P,Q: ^integer;

nil

nil

nil

Значение не определено

nil

5

5

nil

5

nil

8

begin

new (P);

P^:= 5;

Q:= P;

P:= nil;

Q^:= 8;

Write(Q^);

**Динамические структуры**

***Линейные списки (однонаправленные цепочки)***

***Списком*** называется структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Каждый элемент связанного списка, во-первых, хранит какую-либо информацию, во-вторых, указывает на следующий за ним элемент. Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется *звеном*, а структура из таких записей называется *списком или цепочкой*.

Лишь на самый первый элемент списка (голову) имеется отдельный указатель. Последний элемент списка никуда не указывает.

Указатель на голову списка

U

Значение1

Значение2

nil

Значение3

***Описание списка***

Type ukazat= ^S;

S= record

Inf: integer;

Next: ukazat;

End;

В Паскале существует основное правило: перед использованием какого-либо объекта он должен быть описан. Исключение сделано лишь для указателей, которые могут ссылаться на еще не объявленный тип.

***Формирование списка***

Чтобы список существовал, надо определить указатель на его начало.

Type ukazat= ^S;

S= record

Inf: integer;

Next: ukazat;

End;

Var u,x: ukazat;

U

Nil

3

U^.Next

U^.Inf

Создадим первый элемент списка:

New (u); {выделяем место в памяти}

u^. Next:= nil; {указатель пуст}

u^.Inf:=3;

Продолжим формирование списка. Для этого нужно добавить элемент либо в конец списка, либо в голову.

А) *Добавим элемент в голову списка*. Для этого необходимо выполнить последовательность действий:

1. получить память для нового элемента;
2. поместить туда информацию;
3. присоединить элемент к голове списка.

New(x);

Readln(x^.Inf);

x^.Next:=u;

u:=x;

Б) *Добавление элемента в конец списка*. Для этого введем вспомогательную переменную, которая будет хранить адрес последнего элемента. Пусть это будет указатель с именем hv (хвост).

x:= hv;

hv

### X

Указатель на голову списка

U

Значение1

Значение2

nil

Значение3

New(x^.next); {выделяем память для следующего элемента}

hv

### X

U

Значение1

Значение2

Значение3

nil

5

### X

x:= x^.next;

x^.next:= nil;

x^.inf:= 5;

hv:=x;

***Просмотр списка***

While u<> nil do

Begin

Writeln (u^.inf);

u:= u^.next;

end;

***Удаление элемента из списка***

А) *Удаление первого элемента*. Для этого во вспомогательном указателе запомним первый элемент, указатель на голову списка переключим на следующий элемент списка и освободим область динамической памяти, на которую указывает вспомогательный указатель.

x:= u;

U

X

u:= u^.next;

dispose(x);

Б) *Удаление элемента из середины списка*. Для этого нужно знать адреса удаляемого элемента и элемента, стоящего перед ним. Допустим, что digit – это значение удаляемого элемента.

x:= u;

while (x<> nil) and (x^.inf<> digit) do

U

X

DX

begin

dx:= x;

x:= x^.next;

end;

dx:= x^.next:

dispose(x);

В) *Удаление из конца списка*. Для этого нужно найти предпоследний элемент.

x:= u; dx:= u;

while x^.next<> nil do

begin

dx:= x; x:= x^.next;

end;

dx^.next:= nil;

dispose(x);

***Прохождение списка***

Важно уметь перебирать элементы списка, выполняя над ними какую-либо операцию. Пусть необходимо найти сумму элементов списка.

summa:= 0;

x:= u;

while x<> nil do

begin

summa:= summa+ x^.inf;

x:= x^.next;

end;

***Динамические объекты сложной структуры***

Использование однонаправленных списков при решении ряда задач может вызвать определенные трудности. Дело в том, что по однонаправленному списку можно двигаться только в одном направлении, от головы списка к последнему звену. Между тем нередко возникает необходимость произвести какую-либо операцию с элементом, предшествующим элементу с заданным свойством. Однако после нахождения элемента с данным свойством в однонаправленном списке у нас нет возможности получить удобный и быстрый способ доступа к предыдущему элементу.

Для устранения этого неудобства добавим в каждое звено списка еще одно поле, значением которого будет ссылка на предыдущее звено.

Type ukazat= ^S;

S= record

Inf: integer;

Next: ukazat;

Pred: ukazat;

End;

Динамическая структура, состоящая из звеньев такого типа, называется ***двунаправленным списком***, который схематично можно изобразить так:

Значение1

Значение2

nil

Значение3

Указатель на голову списка

U

nil

Наличие в каждом звене двунаправленного списка ссылки как на следующее, так и на предыдущее звено позволяет от каждого звена двигаться по списку в любом направлении. По аналогии с однонаправленным списком здесь есть заглавное звено. В поле Pred этого звена фигурирует пустая ссылка nil, свидетельствующая, что у заглавного звена нет предыдущего (так же, как у последнего нет следующего).

В программировании двунаправленные списки часто обобщают следующим образом: в качестве значения поля Next последнего звена принимают ссылку на заглавное звено, а в качестве значения поля Pred заглавного звена – ссылку на последнее звено:

Значение1

Значение2

Значение3

Указатель на голову списка

U

Как видно, здесь список замыкается в своеобразное «кольцо»: двигаясь по ссылкам, можно от последнего звена переходить к заглавному звену, а при движении в обратном направлении – от заглавного звена переходить к последнему. Списки подобного рода называют ***кольцевыми списками****.*

Существуют различные методы использования динамических списков:

1. ***Стек*** – особый вид списка, обращение к которому идет только через указатель на первый элемент. Если в стек нужно добавить элемент, то он добавляется впереди первого элемента, при этом указатель на начало стека переключается на новый элемент. Алгоритм работы со стеком характеризуется правилом: «последним пришел – первым вышел».
2. ***Очередь*** – это вид списка, имеющего два указателя на первый и последний элемент цепочки. Новые элементы записываются вслед за последним, а выборка элементов идет с первого. Этот алгоритм типа «первым пришел – первым вышел».
3. Возможно организовать списки с произвольным доступом к элементам. В этом случае необходим дополнительный указатель на текущий элемент.

## 2.8 Модули

Модуль — это набор ресурсов (функций, процедур, констант, переменных, типов и т.д.), разрабатываемых и хранимых независимо от использующих их программ. В отличие от внешних подпрограмм модуль может содержать достаточно большой набор процедур и функций, а также других ресурсов для разработки программ. Обычно каждый модуль содержит логически связанные между собой программные ресурсы.

В основе идеи модульности лежат принципы структурного программирования. Существуют стандартные модули Turbo Pascal, которые обычно описываются в литературе по данному языку.

Модуль имеет следующую структуру:

Unit <имя модуля>; {заголовок модуля}

Interface

{интерфейсная часть}

Implementation

{раздел реализации}

Begin

{раздел инициализации модуля}

End.

После служебного слова Unit записывается имя модуля, которое (для удобства дальнейших действий) должно совпадать с именем файла, содержащего данный модуль. Поэтому (как принято в MS DOS) имя не должно содержать более 8 символов.

В разделе Interface объявляются все ресурсы, которые будут в дальнейшем доступны программисту при подключении модуля. Для подпрограмм здесь указывается лишь полный заголовок.

В разделе Implementation реализуются все подпрограммы, которые были ранее объявлены. Кроме того, здесь могут содержаться свои константы, переменные, типы, подпрограммы и т.д., которые носят вспомогательный характер и используются для написания основных подпрограмм. В отличие от ресурсов, объявленных в разделе Interface, все, что дополнительно объявляется в Implementation, уже не будет доступно при подключении модуля. При написании основных подпрограмм достаточно указать их имя (т.е. не нужно полностью переписывать весь заголовок), а затем записать тело подпрограммы.

Раздел инициализации (который часто отсутствует) содержит операторы, которые должны быть выполнены сразу же после запуска программы, использующей модуль.

***Пример 1.*** *Модуль для работы с одномерными массивами до 100 целых чисел.*

{модуль описаний, глобальных для основной программы и всех модулей}

**Unit** Globals;

**Interface**

**const** Len=100;

**type** Vector = **array**[1..Len] **of** integer;

**Implementation**

**End**.

**Unit** Vectors;

**Interface**

**uses** Globals;

{находит максимальный элемент массива}

**function** Max\_V(A:Vector; n:byte):integer;

{поэлементное сложение двух векторов}

**procedure** Add\_V(A,B:Vector; n:byte; **var** C:Vector);

{скалярное произведение векторов}

**function** Scal\_V(A,B:Vector; n:byte):integer;

**Implementation**

**function** Max\_V; {заголовок без параметров}

**var** i,max:integer;

**begin**

max:=A[1];

**for** i:=2 **to** n **do** **if** A[i]>max **then** max:=A[i];

Max\_V:=max;

**end**;

**procedure** Add\_V;

**var** i:integer;

**begin**

**for** i:=1 **to** n **do** C[i]:=A[i]+B[i];

**end**;

**function** Scal\_V(A,B:Vector; n:byte):integer;

{заголовок из interface}

**var** s:integer; i:byte;

**begin**

s:=0;

**for** i:=1 **to** n **do** s:=s+A[i]\*B[i];

Scal\_V:=s;

**end**;

**End**. {раздел инициализации модуля отсутствует}

### *Пример 2*. *Разработать личную библиотеку, включив в нее процедуры:*

* *ввода элементов числовой матрицы размером N\*N;*
* *транспонирования матрицы;*
* *вывода результирующей матрицы.*

*В основной программе ввести размер матрицы N.*

Начнем разработку модуля, который будет носить название Matrix. Программно это будет выглядеть так:

**Unit** Matrix;

{Зарезервированное слово *Unit* служит для указания имени библиотеки. Это имя

должно совпадать с именем PAS-файла библиотеки (т.е библиотека Matrix должна

находиться с файле *Matrix.Pas*), а иначе компилятор даст ошибку при попытке

использования такой библиотеки}

**Interface**

{Секция *Interface* содержит описания общедоступных типов данных, констант,

процедур и функций. Т.е. все, что будет здесь находиться можно будет

использовать при подключении данной библиотеки.}

**Type**

TMatrix = array [1..10,1..10] of Integer; { Квадратная матрица }

procedure MatrInput (Var m : TMatrix; n : Integer); { ввод матрицы }

procedure MatrOutput (Var m : TMatrix; n : Integer); { вывод матрицы }

procedure MatrTransp (Var m : TMatrix; n : Integer); { транспонирование }

**Implementation**

{Секция *Implementation* содержит реализацию тел процедур и функций, описанных

в *Interface*. Также здесь могут содержаться типы данных, константы, процедуры

и функции, необходимые для работы, но которые не будут видны программе при

подключении библиотеки.}

{Процедура обмена местами двух элементов матрицы (x1,y1) и (x2,y2).

Эта процедура используется при транспонировании матрицы, но ее

нельзя вызвать при подключении библиотеки, т.к. она не объявлена

в секции *Interface*.}

procedure Swap (Var m : TMatrix; x1,y1,x2,y2 : Integer);

var

temp : Integer;

begin

temp := m[x1,y1];

m[x1,y1] := m[x2,y2];

m[x2,y2] := temp;

end;

{Ввод матрицы с клавиатуры. Параметры процедуры здесь не указаны, т.к. они есть в секции Interface }

procedure MatrInput;

var

i,j : Integer;

begin

for i:=1 to n do

begin

Write(i:3,'-я строка : ');

for j:=1 to n do Read(m[i,j]);

ReadLn;

end;

end;

{Транспонирование матрицы.}

procedure MatrTransp;

var

i,j : Integer;

begin

for i:=1 to n-1 do

for j:=i+1 to n do

Swap (m,i,j,j,i);

end;

{Вывод матрицы на экран.}

procedure MatrOutput;

var

i,j : Integer;

begin

for i:=1 to n do

begin

Write(i:3,'-я строка : ');

for j:=1 to n do Write (m[i,j]:4);

WriteLn;

end;

end;

{Эта секция может использоваться для инициализации работы библиотеки.}

Begin

End.

Создание модуля закончено.

Теперь необходимо создать файл, который будет содержать текст основной программы, в которой будет подключаться разработанный выше модуль.

{Это отдельный файл, содержащий основную программу}

Uses

Crt, { Библиотека стандартных процедур управления экраном и клавиатурой }

Matrix; {Наш разработанный модуль-библиотека работы с квадратными матрицами (личная)}

Var

: TMatrix; {Объявляем матрицу - максимальный размер 10\*10 }

n : Integer; { Размер матрицы }

Begin

{ Повторяем ввод размера, пока не будет введено корректное значение}

repeat

ClrScr;

Write('Введите размер матрицы (1..10) : ');

ReadLn(n);

until (n >= 1) and (n <= 10);

WriteLn;

WriteLn('Введите матрицу размера',n,'\*',n,'по строкам:');

MatrInput (m,n); {вызов процедуры ввода матрицы, определенной в модуле Matrix}

{Транспонируем ее }

MatrTransp (m,n); {вызов процедуры транспонирования матрицы, определенной в модуле Matrix}

{ Выведем результат на экран }

WriteLn;

WriteLn('Транспонированная матрица :');

MatrOutput (m,n); {вызов процедуры вывода матрицы, определенной в модуле Matrix}

End.

Как видно из примеров, для подключения модуля используется служебное слово USES, после чего указывается имя модуля и происходит это сразу же после заголовка программы. Если необходимо подключить несколько модулей, они перечисляются через запятую.

При использовании ресурсов модуля совсем не нужно знать, как работают его подпрограммы. Достаточно обладать информацией, как выглядят их заголовки, и какое действие эти подпрограммы выполняют. По такому принципу осуществляется работа со всеми стандартными модулями. Поэтому, если программист разрабатывает модули не только для личного пользования, ему необходимо сделать полное описание всех доступных при подключении ресурсов. В таком случае возможна полноценная работа с таким продуктом.

Если в программе, использующей модуль, имеются идентификаторы, совпадающие с точностью до символа с идентификаторами модуля, то они «перекрывают» соответствующие ресурсы модуля. Тем не менее, даже в такой ситуации доступ к этим ресурсам модуля может быть получен таким образом: <имя модуля>.<имя ресурса>.

***Пример*** модуля, выводящего требуемую последовательность символов в заданное место экрана.

Unit Display; {Содержит простую программу вывода на экран}

Interface {Интерфейсная часть}

{Заголовок процедуры}

Procedure WriteXY(X,Y: Integer; Message: String);

Implementation {Раздел реализации}

Uses Crt; {Подключение модуля Crt для выполнения процедуры GotoXY}

{Реализация процедуры}

Procedure WriteXY(X,Y: Integer; Message: String);

Begin

If (X in [1..80]) And (Y in [1..25]) Then Begin

GoToXY(X, Y);

Write(Message);

End;

End;

End. {Конец модуля}

Все модули хранятся в файлах с расширением TPU, имена которых совпадают с именами модулей. Для создания на диске файла с именем Display.tpu нужно выполнить компиляцию этого модуля (клавиши Alt+F9), предварительно установив опцию Compile/Distination/Disk.

# РАЗДЕЛ 3. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

## 3.1 Лексика языка С++

Язык C++ – это язык программирования общего назначения. За исключением несущественных деталей, язык C++ является надмножеством языка С. Помимо возможностей, предоставляемых языком С, язык C++ обеспечивает гибкие и эффективные средства определения новых типов.

Язык программирования служит двум взаимосвязанным целям: он предоставляет программисту инструмент для описания подлежащих выполнению действий и набор концепций, которыми оперирует программист, обдумывая, что можно сделать. Первая цель в идеале требует языка, близкого к компьютеру, чтобы все важные элементы компьютера управлялись просто и эффективно способом, достаточно очевидным для программиста. Вторая цель в идеале требует языка, близкого к решаемой задаче, чтобы концепции решения могли быть выражены понятно и непосредственно. Эта идея привела к пополнению языка С свойствами, превратившими его в язык C++.

Ключевое понятие в языке C++ – **класс**. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью и механизм перегрузки операторов. Язык C++ предоставляет гораздо лучшие, чем язык С, средства для проверки типов и поддержки модульного программирования. Кроме того, язык содержит усовершенствования, непосредственно не связанные с классами, такие как: символические константы, встраивание функций в место вызова, параметры функций по умолчанию, перегруженные имена функций, операторы управления свободной памятью и ссылки. Язык C++ сохраняет способность языка С эффективно работать с аппаратной частью на уровне битов, байтов, слов, адресов и т.д. Это позволяет реализовывать пользовательские типы с достаточной степенью эффективности.

**Алфавит**

Множество символов языка C++ включает:

* прописные буквы латинского алфавита;
* строчные буквы латинского алфавита;
* арабские цифры;
* разделители: , . ; : ? ! ' " | / \ ~ \_ ^ ( ) { } [ ] < > # % & - = + \*

Остальные символы могут быть использованы только в символьных строках, символьных константах и комментариях.

Из символов алфавита формируют ключевые слова и идентификаторы. Ключевые слова – это зарезервированные слова, которые имеют специальное значение для компилятора и используются только в том смысле, в котором они определены (операторы языка, типы данных и т п.). Идентификатор - это имя программного объекта, представляющее собой совокупность букв, цифр и символа подчеркивания. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Идентификатор не может содержать пробел. Каждое имя (идентификатор) должно быть уникальным в пределах функции и не должно совпадать с ключевыми словами. Язык C++ различает прописные и строчные буквы, таким образом, *name* и *Name* – **разные идентификаторы**.

В тексте программы можно использовать комментарии. Комментарий – это последовательность символов, которая игнорируется компилятором языка C++. Комментарий имеет следующий вид: /\*<символы>\*/. Комментарии могут занимать несколько строк, но не могут быть вложенными. Кроме того, часть строки, следующая за символами //, также рассматривается как комментарий.

Удачно подобранный и написанный набор комментариев является существенной частью хорошей программы. Написание «правильных» комментариев может оказаться не менее сложной задачей, чем написание самой программы. При неправильном использовании комментариев читабельность программы может, напротив, серьезно пострадать.

**Литералы**

Литералы в языке C++ могут быть целые, вещественные, символьные и строковые.

* Целые:
  + десятичные: 10, 132, -32179;
  + восьмеричные (предваряются символом «0»): 010, 0204, -076663;
  + шестнадцатеричные (предваряются символами «0х»): 0хА, 0x84, 0x7db3.
* Вещественные: 15.75, 1.575e1, .75, -.125
* Символьные: 'a', 'e', '.', '?', '2'.
* Строковые: "строка".

**Типы данных языка C++**

Для решения задачи в любой программе выполняется обработка каких -либо данных . Данные могут быть различных типов: целые и вещественные числа, символы, строки, массивы. Данные в языке С++ принято описывать в начале функции. Обязательное описание типов данных позволяет компилятору контролировать допустимость программных кодов.

Типы данных определяют способ хранения чисел или символов в памяти компьютера. Они задают размер ячейки, в которую будет записано то или иное значение, определяя тем самым его максимальную величину или точность задания. Типы языка С++ можно разделить на основные и составные. К основным типам данных языка относят:

* char – символьный;
* int –целый;
* float –с плавающей точкой;
* double – двойной точности;
* bool –логический.

Для формирования других типов данных используют основные типы и так называемые спецификаторы. Типы данных, созданные на базе стандартных типов с использованием спецификаторов, называют составными типами данных . В С++ определены четыре спецификатора типов данных :

* short – короткий;
* long – длинный;
* signed – знаковый;
* unsigned – беззнаковый.

# Символьный тип

Данные типа char в памяти компьютера всегда занимают один байт. Это связано с тем, что обычно под величину символьного типа отводят столько памяти, сколько необходимо для хранения любого из 256 символов клавиатуры. Символьный тип может быть со знаком или без знака.

|  |  |  |
| --- | --- | --- |
| **Тип** | **Диапазон** | **Размер** |
| char | -128..127 | 1 байт |
| unsigned char | 0..255 | 1 байт |
| signed char | -128..127 | 1 байт |

При работе с символьными данными нужно помнить, что если в выражении встречается *одиночный символ*, он должен быть заключен в одинарные кавычки. Последовательность символов, то есть *строка*, при использовании в выражениях заключается в двойные кавычки.

**Целочисленный тип**

Переменная типа int в памяти компьютера может занимать либо два, либо четыре байта. Это зависит от разрядности процессора. По умолчанию все целые типы считаются знаковыми, т.е. спецификатор signed можно не указывать. Спецификатор unsigned позволяет представлять только положительные числа.

Целые типы данных

|  |  |  |
| --- | --- | --- |
| **Тип** | **Диапазон** | **Размер** |
| int | -32767 .. 32767 | 4 байта |
| unsigned int | 0 .. 65535 | 4 байта |
| signed int | -32767 .. 32767 | 4 байта |
| short int | -32767 .. 32767 | 2 байта |
| long int | -2147483647 .. 2147483647 | 4 байта |
| unsigned short int | 0 .. 65535 | 2 байта |
| signed short int | -32767 .. 32767 | 2 байта |
| long long int | -(263-1) .. (263-1) | 8 байт |
| signed long int | -2147483647 .. 2147483647 | 4 байта |
| unsigned long int | 0 .. 4294967295 | 4 байта |
| unsigned long long int | 0 .. 264-1 | 8 байт |

# Вещественный тип

Внутреннее представление вещественного числа в памяти компьютера отличается от представления целого числа. Число с плавающей точкой представлено в экспоненциальной

форме **mE±p**, где **m** мантисса (целое или дробное число с десятичной точкой), **p** порядок (целое число).

Для того чтобы перевести число в экспоненциальной форме к обычному представлению с фиксированной точкой, необходимо мантиссу умножить на десять в степени порядок. Например,

-6.42Е+2 = -6.42.102 = -642,

3.2E-6 = 3.2.10-6 =0.0000032

Вещественные типы данных

|  |  |  |
| --- | --- | --- |
| **Тип** | **Диапазон** | **Размер** |
| float | 3.4Е-38 .. 3.4E+38 | 4 байта |
| double | 1.7Е-308 .. 1.7E+308 | 8 байт |
| long double | 3.4Е-4932 .. 3.4E+4932 | 8 байт |

# Логический тип

Переменная типа bool может принимать только два значения true (истина) или false (ложь). Любое значение не равное нулю интерпретируется как true, а при преобразовании к целому типу принимает значение равное 1. Значение false представлено в памяти как 0.

**Тип void**

Это пустой тип, который имеет специальное назначение. Он используется для объявления функций, которые не возвращают никакого значения, а также для объявления [*указателей*](http://natalia.appmat.ru/c&c++/lezione5.php) на значение типа *void*. Такие указатели могут быть преобразованы к указателям на любой другой тип.

В языке C++ **нет специальных типов для массивов и строк**, которые представляются массивом символов.

При выполнении бинарных операций производятся **преобразования по умолчанию** для приведения операндов к одному и тому же типу, который потом используется как тип результата.

В языке С++ нет операций преобразования между символом и кодом символа, т.к. в оперативной памяти символ и так храниться в виде его кода. Поэтому можно к переменной, хранящей символ, прибавить 1 и получить следующий символ.

**Переменные в языке С++**

Переменная – это поименованный участок памяти, в котором хранится значение определенного типа. У переменной есть имя (идентификатор) и значение. Имя служит для обращения к области памяти, в которой хранится значение. Во время выполнения программы значение переменной можно изменить. Перед использованием любая переменная должна быть описана: тип список\_переменных. Например, int a, b, c, f; float g, u, h12.

***Примеры объявления переменных***

int a; // объявление переменной a целого типа.

float b; // объявление переменной b типа данных с плавающей запятой.

double c = 14.2; // инициализация переменной типа double.

char d = 's'; // инициализация переменной типа char.

bool k = true; // инициализация логической переменной k.

**Константы в языке С++**

Константы – это величины, которые не изменяют своего значения в процессе выполнения программы. Оператор описания константы имеет вид:

const тип имя = значение;

Константы в языке С++ могут быть целыми вещественными символьными или строковыми. Например,

const int a=100;

const double pi=3.141592653589793

**Операции и выражения**

Выражение задает порядок выполнения действий над данными и состоит из операндов (констант, переменных, обращений к функциям), круглых скобок и знаков операций (например, а+b\*sin(cos(x))). Операции делятся на унарные (например, -с) и бинарные (например а+b).

**Основные операции языка C++**

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| Унарные операции | |
| ++ | увеличение значения на единицу |
| -- | уменьшение значения на единицу |
| ~ | поразрядное отрицание |
| ! | логическое отрицание |
| - | арифметическое отрицание (унарный минус) |
| + | унарный плюс |
| & | взятие адресаа |
| \* | разадресация |
| (type) | преобразование типа |
| Бинарные операции | |
| + | сложение |
| - | вычитание |
| \* | умножение |
| / | деление |
| % | остаток от деления |
| << | сдвиг влево |
| >> | сдвиг вправо |
| < | меньше |
| > | больше |
| <= | меньше или равно |
| >= | больше или равно |
| == | Равно |
| != | не равно |
| & | поразрядная конъюнкция (И) |
| ^ | поразрядное исключающее ИЛИ |
| | | поразрядная дизъюнкция (ИЛИ) |
| && | логическое И |
| || | логическое ИЛИ |
| = | присваивание |
| \*= | умножение с присваиванием |
| /= | деление с присваиванием |
| += | сложение с присваиванием |
| -= | вычитание с присваиванием |
| %= | остаток от деления с присваиванием |
| <<= | сдвиг влево с присваиванием |
| >>= | сдвиг вправо с присваиванием |
| &= | поразрядная конъюнкция с присваиванием |
| |= | поразрядная дизъюнкция с присваиванием |
|  |  |
| ^= | поразрядное исключающее ИЛИ с присваиванием |
| Другие операции | |
| ?: | условная операция |
| , | последовательное вычисление |
| sizeof | определение размера |
| (тип) | преобразование типа |

**Операции присваивания**

* **Простое присваивание**. Операция простого присваивания обозначается знаком «=». Значение правого операнда присваивается левому операнду. Операция вырабатывает результат, который может быть далее использован в выражении. Результатом операции является присвоенное значение. Например, выражение ***a*** = ***b*** = ***c*** = 0 присваивает всем переменным значение 0, а в результате вычисления выражения ***a***= (***b*** = 3) + (***c*** = 5) переменная ***c*** будет иметь значение 5, переменная ***b*** будет иметь значение 3, и переменная ***a*** будет иметь значение 8.
* **Составное присваивание**. Операция составного присваивания состоит из простой операции присваивания, скомбинированной с какой-либо другой бинарной операцией. При составном присваивании вначале выполняется действие, специфицированное бинарной операцией, а затем результат присваивается левому операнду. Оператор ***n*** += 5 эквивалентен оператору ***n*** = ***n*** + 5, но при этом первый оператор легче для понимания и выполняется быстрее.
* Операции **++** и **--** инкрементируют (увеличивают на 1) и декрементируют (уменьшают на 1) свой операнд. Операнд должен иметь целый, вещественный тип или быть указателем. Операции инкремента и декремента могут записываться как перед своим операндом (префиксная форма записи), так и после него (постфиксная форма записи).
* **sizeof** – вычисление размера в байтах переменной или типа.
* **Операция приведения типа** записывается следующим образом: (<*новый тип*>)<*выражение*>. Например, (long int)n приводит переменную ***n*** к типу *long int*. При преобразовании типов надо помнить, что при преобразовании между знаковыми/беззнаковыми значениями и при преобразовании от типа с большей размерностью к типу с меньшей размерностью могут возникнуть ошибки. Более безопасным способом преобразования типов является использование операций *static\_cast*, *dynamic\_cast*, *reinterpret\_cast* и *const\_cast*.
* **%** – остаток от деления.
* В языке C++ имеется одна тернарная операция – **условная операция**. Она имеет следующий синтаксис: <*операнд 1*> ? <*операнд 2*> : <*операнд 3*>. Если <*операнд 1*> имеет ненулевое значение, то вычисляется <*операнд 2*> и результатом условной операции является его значение. Если же <*операнд 1*> равен нулю, то вычисляется <*операнд 3*> и результатом является его значение. В любом случае вычисляется только один из операндов, <*операнд 2*> или <*операнд 3*>, но не оба.
* **Операция последовательного вычисления «,»** обычно используется для вычисления нескольких выражений в ситуациях, где по синтаксису допускается только одно выражение. Однако, запятая, разделяющая параметры функции, *не является* операцией последовательного вычисления.

# Стандартные функции

В C++ определены *стандартные функции* над арифметическими операндами

|  |  |
| --- | --- |
| **Обозначение** | **Действие** |
| abs(x) | Модуль целого числа |
| fabs(x) | Модуль вещественного числа |
| sin(x) | Функция синус |
| cos(x) | Функция косинус |
| tan(x) | Функция тангенс |
| atan(x) | Арктангенс x в диапазоне -π/2 до π/2 |
| exp(x) | Экспонента, eх |
| log(x) | Натуральный логарифм (x>0) |
| log10(x) | Десятичный логарифм (x>0) |
| sqrt(x) | Корень квадратный (x≥0) |
| pow(x,y) | x в степени y |

* 1. **Операторы языка С++**

**Структура программы**

Программа на языке С++ состоит из функций, описаний и директив процессора.

**Препроцессор** – это программа, которая обрабатывает текст вашей программы до компилятора. Таким образом, на вход компилятора попадает текст, который может отличаться от того, который видите Вы. Работа препроцессора управляется директивами. С помощью препроцессора можно выполнять следующие операции:

* [*включение в программу текстов из указанных файлов*](http://natalia.appmat.ru/c&c++/lezione2.php#l2_2_3_1);
* [*замена идентификаторов последовательностями символов*](http://natalia.appmat.ru/c&c++/lezione2.php#l2_2_3_2);
* [*макроподстановка*](http://natalia.appmat.ru/c&c++/lezione2.php#l2_2_3_2), т.е. замена обозначения параметризованным текстом, формируемым препроцессором с учетом конкретных аргументов;
* исключение из программы отдельных частей текста ([*условная компиляция*](http://natalia.appmat.ru/c&c++/lezione2.php#l2_2_3_3))

Среди функций выделяется одна главная функция, которая должна иметь имя *main*. С нее начинается выполнение программы, обычно она управляет выполнением программы, организуя вызовы других функций. Для того чтобы программа могла быть скомпилирована и выполнена, она должна содержать, по крайней мере, определение функции *main*.

Элементарное описание функции имеет вид:

тип\_результата имя\_функции (параметры)

{

Оператор 1;

Оператор 2;

..

Оператор N;

Здесь, тип\_результата - это тип того значения, которое функция не должна вычислить (если функция не должна возвращать значение, указывается тип void), имя\_функции – это имя, с которым можно обращаться к этой функции, параметры – список аргументов функции (может отсутствовать), оператор 1, оператор 2, …, оператор N – операторы, представляющие тело функции. Они заключаются в фигурные скобки и каждый оператор заканчивается точкой с запятой. Как правило программа на С++ состоит из одной или нескольких, не вложенных друг в друга функций.

Основному тексту программы предшествуют директивы процессора, которые в общем виде выглядят так:

#include <имя\_файла>

Каждая такая строка дает компилятору команду присоединить программный код, который хранится в отдельном файле с расширением .h. Такие файлы называют файлами заголовков. С их помощью можно выполнять ввод-вывод данных, работать с математическими функциями, преобразовывать данные, распределять память и многое другое. Например, описание стандартных математических функций находится в заголовочном файле math.h.

Общую структуру программы на языке С++ можно записать следующим образом: директивы процессора

описание глобальных переменных тип\_результата main(параметры)

{

операторы главной функции

}

тип\_результата имя1(параметры1)

{

операторы1;

}

тип\_результата имя2(параметры2)

{

операторы2;

}

...................................................

тип\_результата имяN(параметрыN)

{

операторыN;

}

# Ввод-вывод данных

Описание объектов для управления *вводом-выводом* содержится в заголовочном файле iostream.h. При подключении этого файла с помощью директивы #include <iostream.h> в программе автоматически создаются объекты-потоки cin для *ввода* с клавиатуры и cout для *вывода* на экран, а так же операции помещения в поток << и чтения из потока >>.

Итак, с помощью объекта cin и операции >> можно присвоить значение любой переменной. Например, если переменная i описана как целочисленная, то команда cin>> i; означает, что в переменную i будет записано некое целое число, введенное с клавиатуры. Если нужно ввести несколько переменных, следует написать cin>>x>>y>>z;. Объект cout и операция << позволяют вывести на экран значение любой переменной или текст. Текст необходимо заключать в двойные кавычки, кроме того, допустимо применение специальных символов \t и \n. Запись cout<<i; означает вывод на экран значения переменной i. А команда cout<<x<<''\t''<<y; выведет на экран значения переменных x и y разделенные символом табуляции.

**Пример первой программы**

#include <iostream>

#include <cstdlib> // для system

using namespace std;

int main()

{

cout << "Hello, world!" << endl;

system("pause");

return 0;

}

**Описание синтаксиса**

Директива #include используется для подключения других файлов в код. Строка #include <iostream>, будет заменена содержимым файла «iostream.h», который находится в стандартной библиотеке языка и отвечает за ввод и вывод данных на экран.

#include <cstdlib> подключает стандартную библиотеку языка С. Это подключение необходимо для работы функции system.

Содержимое третьей строки — using namespace std; указывает на то, что мы используем по умолчанию [пространство имен](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) с названием «std». Все то, что находится внутри фигурных скобок функции int main() {} будет автоматически выполняться после запуска программы.

Строка cout << "Hello, world!" << endl; говорит программе выводить сообщение с текстом **«Hello, world»** на экран.

Оператор cout предназначен для вывода текста на экран командной строки. После него ставятся две угловые кавычки (<<). Далее идет текст, который должен выводиться. Он помещается в двойные кавычки. Оператор endl переводит строку на уровень ниже.

Если в процессе выполнения произойдет какой-либо сбой, то будет сгенерирован код ошибки, отличный от нуля. Если же работа программы завершилась без сбоев, то код ошибки будет равен нулю. Команда return 0 необходима для того, чтобы передать операционной системе сообщение об удачном завершении программы. В конце каждой команды ставится **точка с запятой.**

# Операторы языка C++

Операторы управляют процессом выполнения программы. Набор операторов языка С++ содержит все управляющие конструкции структурного программирования.

Составной оператор ограничивается фигурными скобками. Все другие операторы заканчиваются точкой с запятой.

* **Пустой оператор –** ;

Пустой оператор – это оператор, состоящий только из точки с запятой. Он может появиться в любом месте программы, где по синтаксису требуется оператор. Выполнение пустого оператора не меняет состояния программы.

* **Составной оператор –** {...}

Действие составного оператора состоит в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда какой-либо оператор явно передает управление в другое место программы.

**Условный оператор**

Оператор *if* имеет следующую структуру:

if (*<условие>*) *<оператор 1>;* [else *<оператор 2>*];

где **условие** - логическое выражение, переменная или константа.

Работает условный оператор следующим образом. Если условие имеет значение истина (true), выполняется оператор 1. В противном случае, выполняется оператор 2.

***Пример 1.*** При вводе числа, меньшего десяти, будет выводиться соответствующее сообщение. Если введенное число окажется большим, либо равным десяти — отобразится другое сообщение.

#include <iostream>

using namespace std;

int main()

{

setlocale(0, "");

double num;

cout << "Введите произвольное число: ";

cin >> num;

if (num < 10) { // Если введенное число меньше 10.

cout << "Это число меньше 10." << endl;

} else { // иначе

cout << "Это число больше либо равно 10." << endl;

}

return 0;

}

***Пример 2.*** Программа вычисления корней квадратноготуравнения.

#include <iostream>

#include <math.h> using namespace std; int main()

{float a,b,c,d,x1,x2;

//Ввод значений коэффициентов квадратного уравнения

cout<<"a=";cin>>a; cout<<"b=";cin>>b; cout<<"c=";cin>>c;

//Вычисление дискриминанта

d=b\*b-4\*a\*c;

//Если дискриминант отрицателен,

if (d<0)

//то вывод сообщения, что корней нет, cout<<"Real roots are not present"; else

{//иначе вычисление корней x1, x2 x1=(-b+sqrt(d))/2/a;

x2=(-b-sqrt(d))/(2\*a);

//и вывод их значений на экран cout<<"X1="<<x1<<"\t X2="<<x2<<"\n";

}

return 0;

}

**Оператор выбора**

Оператор выбора **switch** необходим в тех случаях, когда в зависимости от значений какой-либо переменной надо выполнить те или иные операторы:

switch (*<выражение>*)

{ case *<константное выражение 1>*: *<операторы 1>;* **break;**

case *<константное выражение 2>*: *<операторы 2>;* **break;**

...

case *<константное выражение N>*: *<операторы N>;* **break;**

[default: *<операторы>;*]

}

Оператор выбора предназначен для выбора одного из нескольких альтернативных путей выполнения программы. Вычисление оператора-переключателя начинается с вычисления *выражения*, после чего управление передается *оператору*, помеченному *константным выражением*, равным вычисленному значению *выражения*. Выход из оператора-переключателя осуществляется оператором **break**. Если оператор **break** не указан, то будут выполняться следующие операторы из списка. Если значение *выражения* не равно ни одному *константному выражению*, то управление передается *оператору*, помеченному ключевым словом *default*, если он есть.

***Пример.*** По заданному номеру месяца m вывести на печать его название.

#include <iostream>

using namespace std;

int main()

{

unsigned int m; cout<<"m="; cin>>m; switch (m)

{

//В зависимости от значения m выводится название месяца

case 1: cout<<"Январь \n"; break;

case 2: cout<<"Февраль \n"; break;

case 3: cout<<"Март \n"; break;

case 4: cout<<"Апрель \n"; break;

case 5: cout<<"Май \n"; break;

case 6: cout<<"Июнь \n"; break;

case 7: cout<<"Июль \n"; break;

case 8: cout<<"Август \n"; break;

case 9: cout<<"Сентябрь \n"; break;

case 10:cout<<"Октябрь \n"; break;

case 11:cout<<"Ноябрь \n"; break;

case 12:cout<<"Декабрь\n"; break;

//Если значение переменной m выходит за пределы области допустимых значений, то выдается сообщение об ошибке

default: cout<<"Ошибка! \n"; break;

}

return 0;

}

**Операторы цикла**

Иногда необходимо повторять одно и то же действие несколько раз подряд. Для этого используют циклы.

**Цикл с параметром**

Если мы знаем точное количество действий (итераций) цикла, то можем использовать **цикл for**.

for (*начальное выражение*; *условное выражение*; *выражение приращения>*)

{Тело цикла}

Тело оператора **for** выполняется до тех пор, пока *условное выражение* не станет ложным. *Начальное выражение* и *выражение приращения* обычно используются для инициализации и модификации параметров цикла и других значений. *Начальное выражение* вычисляется один раз до первой проверки *условного выражения*, а *выражение приращения* вычисляется после каждого выполнения *оператора*. Любое из трех выражений заголовка цикла, и даже все три могут быть опущены (не забывайте только оставлять точки с запятой). Если опущено *условное выражение*, то оно считается истинным, и цикл становится бесконечным.

***Пример*** программы, которая будет считать сумму всех чисел от 1 до 1000.

#include <iostream>

using namespace std;

int main()

{

int i; // счетчик цикла

int sum = 0; // сумма чисел от 1 до 1000.

setlocale(0, "");

for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.

{

sum = sum + i;

}

cout << "Сумма чисел от 1 до 1000 = " << sum << endl;

return 0;

}

Если мы скомпилируем этот код и запустим программу, то она покажет нам ответ: 500500. Это и есть сумма всех целых чисел от 1 до 1000. В теле цикла, при каждом проходе программа увеличивает значение переменной sum на i. Еще один очень важный момент — в начале программы переменной sum присвоено значение нуль.

***Пример*** вычисления факториала числа

#include "stdafx.h"

#include <iostream.h>

using namespace std;

int main()

{

unsigned int factorial, N, i;

for(cout<<"N=",cin>>N,factorial=1,i=2;i<=N;factorial\*=i,i++); cout<<"factorial="<<factorial<<"\n";

return 0;

}

Если заранее неизвестно число повторений цикла, используются циклы с условием.

**Цикл с предусловием**

Оператор цикла с предусловием имеет вид:

while (*условие*) *{тело цикла}*

Данный цикл будет выполняться, пока условие, указанное в круглых скобках является истинным.

***Пример*** программы, считающей сумму всех целых чисел от 1 до 1000.

#include <iostream>

using namespace std;

int main()

{

setlocale(0, "");

int i = 0; // инициализируем счетчик цикла.

int sum = 0; // инициализируем счетчик суммы.

while (i < 1000)

{

i++;

sum += i;

}

cout << "Сумма чисел от 1 до 1000 = " << sum << endl;

return 0;

}

Рассмотрим по порядку исходный код программы. Сначала мы инициализируем счетчик цикла и переменную, хранящую сумму чисел.

В данном случае обязательно нужно присвоить счетчику цикла какое-либо значение, затем описывается условие цикла — **«пока переменная i меньше 1000 — выполняй цикл»**. При каждой итерации цикла значение переменной-счетчика i увеличивается на единицу внутри цикла.

Когда выполнится 1000 итераций цикла, счетчик станет равным 999 и следующая итерация уже не выполнится, поскольку 1000 не меньше 1000. Выражение sum += i является укороченной записью sum = sum + i.

После окончания выполнения цикла, выводим сообщение с ответом.

***Пример*** программы поиска наибольшего общего делителя двух чисел

#include <iostream>

using namespace std;

int main()

{

unsigned

int a,b;

cout<<"A=";

cin>>a;

cout<<"B=";

cin>>b;

//Если числа не равны, выполнять тело цикла

while (a!=b)

//Если число А больше, чем В, то уменьшить его значение на В,

if (a>b) a=a-b;

//иначе уменьшить значение числа В на А

else b=b-a;

cout<<"NOD="<<a<<"\n"; return 0;

}

**Цикл с постусловием**

В языке C++ этот оператор отличается от классической реализации цикла с постусловием тем, что при истинности *выражения* происходит продолжение работы цикла, а не выход из цикла.

do *{тело цикла}* while (условие);

Цикл **do while** очень похож на цикл **while**. Единственное их различие в том, что при выполнении цикла **do while** один проход цикла будет выполнен независимо от условия. Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла do while.

#include <iostream>

using namespace std;

int main ()

{

setlocale(0, "");

int i = 0; // инициализируем счетчик цикла.

int sum = 0; // инициализируем счетчик суммы.

do {// выполняем цикл.

i++;

sum += i;

} while (i < 1000); // пока выполняется условие.

cout << "Сумма чисел от 1 до 1000 = " << sum << endl;

return 0;

}

Принципиального отличия нет, но если присвоить переменной i значение, большее, чем 1000, то цикл все равно выполнит хотя бы один проход.

**Операторы передачи управления**

* **Break.** Оператор разрыва break прерывает выполнение операторов **while**, **do**, **for** и **switch**. Он может содержаться только в теле этих операторов. Управление передается оператору программы, следующему за прерванным. Если оператор разрыва записан внутри вложенных операторов **while**, **do**, **for**, **switch**, то он завершает только непосредственно охватывающий его оператор.
* **Continue.** Оператор продолжения continue передает управление на следующую итерацию в операторах цикла **while**, **do**, **for**. Он может содержаться только в теле этих операторов. В операторах **do** и **while** следующая итерация начинается с вычисления условного выражения. В операторе **for** следующая итерация начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения.
* **Return [*<выражение>*].** Оператор возврата return заканчивает выполнение функции, в которой он содержится, и возвращает управление в вызывающую функцию. Управление передается в точку вызывающей функции, непосредственно следующую за оператором вызова. Значение *выражения*, если она задано, вычисляется, приводится к типу, объявленному для функции, содержащей оператор возврата, и возвращается в вызывающую функцию. Если *выражение* опущено, то возвращаемое функцией значение не определено.

С формальной точки зрения операторы **break**, **continue** и **return** не являются операторами структурного программирования. Однако их использование в ограниченных количествах оправдано, когда они упрощают понимание программы и позволяют избегать больших вложенных структур. Например, мы проверяем входные данные на аномалии. Если не использовать эти операторы, то всю обработку придется вложить в условный блок, что ухудшает читабельность программы. Вместо этого можно написать небольшой условный блок, который организует выход из функции при неверных исходных данных.

## 3.3 Массивы

**Массив** – это структурированный тип данных, состоящий из фиксированного числа однотипных элементов.

# Объявление массива

Объявление массива имеет следующий синтаксис:

*<спецификация типа> <имя>* [*<константное выражение>*];

*<спецификация типа> <имя>* [ ];

Здесь квадратные скобки являются элементом синтаксиса, а не признаком необязательности конструкции.

Объявление массива может иметь одну из двух синтаксических форм, указанных выше. Квадратные скобки, следующие за *именем*, – признак того, что переменная является массивом. *Константное выражение*, заключенное в квадратные скобки, определяет число элементов в массиве. **Индексация элементов массива в языке C++ начинается с нуля.** Таким образом, последний элемент массива имеет индекс на единицу меньше, чем число элементов массива.

Во второй синтаксической форме константное выражение в квадратных скобках опущено. Эта форма может быть использована, если в объявлении массива присутствует инициализатор, либо массив объявляется как формальный параметр функции, либо данное объявление является ссылкой на объявление массива где-то в другом месте программы.

Массив может состоять из элементов любого типа, кроме типа [void](http://natalia.appmat.ru/c&c++/lezione1.php#l1_void) и функций, т.е элементы массива могут иметь [базовый](http://natalia.appmat.ru/c&c++/lezione1.php#l1_4), [перечислимый](http://natalia.appmat.ru/c&c++/lezione6.php#l6_1), [структурный](http://natalia.appmat.ru/c&c++/lezione6.php#l6_2) тип, быть [объединением](http://natalia.appmat.ru/c&c++/lezione6.php#l6_3), [указателем](http://natalia.appmat.ru/c&c++/lezione5.php#l5_1) или массивом.

**Примеры объявлений массивов**

int x[10]; // Одномерный массив из 10 целых чисел. Индексы меняются от 0 до 9.

double y[2][10]; // Двумерный массив вещественных чисел из 2 строк и 10 столбцов.

# Инициализация массивов

Как и простые переменные, массивы могут быть инициализированы при объявлении. Инициализатор для объектов составных типов (каким является массив) состоит из списка инициализаторов, разделённых запятыми и заключённых в фигурные скобки. Каждый инициализатор в списке представляет собой либо константу соответствующего типа, либо, в свою очередь, список инициализаторов. Эта конструкция используется для инициализации многомерных массивов.

Наличие списка инициализаторов в объявлении массива позволяет не указывать число элементов по его первой размерности. В этом случае количество элементов в списке инициализаторов и определяет число элементов по первой размерности массива. Тем самым определяется размер памяти, необходимой для хранения массива. Число элементов по остальным размерностям массива, кроме первой, указывать обязательно.

Если в списке инициализаторов меньше элементов, чем в массиве, то оставшиеся элементы неявно инициализируются нулевыми значениями. Если же число инициализаторов больше, чем требуется, то выдаётся сообщение об ошибке.

**Примеры инициализации массивов**

int a[3] = {0, 1, 2}; // Число инициализаторов равно числу элементов

double b[5] = {0.1, 0.2, 0.3}; // Число инициализаторов меньше числа элементов

int c[ ] = {1, 2, 4, 8, 16}; // Число элементов массива определяется по числу инициализаторов

int d[2][3]={{0, 1, 2}, {3, 4, 5}}; // Инициализация двумерного массива. Массив состоит из двух строк, в каждой из которых по 3 элемента. Элементы первой строки получают значения 0, 1 и 2, а второй – значения 3, 4 и 5.

# Работа с массивами

## Доступ к элементу массива

Для доступа к конкретному элементу массива используются так называемые индексные выражения:

*<имя массива>*[*<целочисленное выражение>*]

Здесь квадратные скобки являются требованием синтаксисам языка, а не признаком необязательности конструкции.

Индекс массива может быть не только константой, но и выражением, которое имеет целочисленный тип, например, **a**[**i** + 1] (здесь **a** должно быть именем ранее объявленного массива, а **i** – переменной целого типа).

Объявление массива и индексное выражение, используемое для доступа к элементу массива, имеют схожий синтаксис. Различаются они по месту в программе. Это особенно важно, когда мы определяем индекс последнего элемента массива. Как было сказано ранее, индексы элементов массива в языке C++ начинаются с 0, и номер последнего элемента на 1 меньше количества элементов массива. Поэтому если Вы объявили массив **x** из 10 элементов, Вы не можете написать индексное выражение **x**[10], т.к. в этом случае Вы пытаетесь обратиться к элементу с индексом 10, которого нет в Вашем массиве. Компилятор не выдаст сообщения об ошибке, но результаты работы такой программы будут непредсказуемы.

**Имя массива является адресом его начала!** Оно имеет тип *константный указатель на <тип элементов массива>*. Конструкция **a**[**i**] эквивалентна \*(**a** + **i**).

## Обработка массивов

Для обработки элементов массива обычно используется оператор цикла *for*.

for (i=0; i < n; i++)

<тело цикла>

Присваиваем счетчику цикла значение индекса первого элемента. Условие продолжения цикла – пока значение счетчика меньше количества элементов массива. Увеличиваем счетчик цикла на 1 для перехода к следующему элементу массива. В теле цикла происходит обработка одного элемента массива.

Для обработки многомерного массива используется соответствующее количество циклов.

## Ввод/вывод одномерного массива

В языке C++ нет возможности вводить и выводить весь массив одним оператором ввода/вывода. Можно вводить и выводить только один элемент массива. Следовательно, для того чтобы ввести весь массив, надо использовать цикл.

float x[10];

int i,n; cout<<"\n N="; cin>>n;

cout<<"\n Vvedite massiv X \n"; for(i=0;i<n;i++)

cin>>x[i];

Вывод также осуществляется в цикле.

cout<<"\n Massiv X \n"; for(i=0;i<n;i++) cout<<x[i]<<"\t";

### Обработка одномерного массива

Алгоритм нахождения суммы элементов одномерного массива

for(s=i=0;i<N;i++)

s+=X[i];

Это можно записать и так:

for(s=i=0;i<N;s+=X[i],i++);

Алгоритм нахождения произведения элементов одномерного массива

for(P=1,i=0;i<n;i++)

P\*=X[i];

Поиск максимального элемента и его номера

for(max=X[0],nmax=0,i=1;i<n;i++) if (x[i]>max)

{

max=x[i]; nmax=i;

}

Алгоритм поиска минимума будет отличаться знаком в блоке сравнения. Значительно более интересной является задача поиска минимального (максимального) элемента массива, среди элементов массива, удовлетворяющих некоторому условию.

***Пример*** поиска минимального значения, среди положительных элементов массива.

for(i=k=0;i<n;i++)

// Если элемент положительный,

if (x[i]>0)

{

// то увеличиваем количество положительных элементов на 1.

k++;

// Если это первый положительный элемент, то объявляем его минимальным

if (k==1) {min=x[i];nmin=i;}

// иначе сравниваем его с минимальным

else if (x[i]<min)

{

min=x[i]; nmin=i;

}

}

***Пример***. Удалить элементы с 4-го по 8-й в массиве из N элементов

int main()

{

floxt x[20]; int i,j,n; cout<<"n="; cin>>n;

cout<<"Massiv x\n"; for(i=0;i<n;i++)

cin>>x[i]; for(j=1;j<=5;j++)

{

for(i=3;i<=n-2;i++) x[i]=x[i+1];

n--;

}

cout<<"Massiv x\n"; for(i=0;i<n;i++)

cout<<"x("<<i<<")="<<x[i]<<"\t"; cout<<endl;

return 0;

}

***Пример*** упорядочивания элементов массива

int main()

{

float b,max,a[20]; int i,j,n,k,nom; cout<<"n="; cin>>n;

cout<<"Massiv a\n"; for(i=0;i<n;i++)

cin>>a[i]; k=n;

for(j=0;j<=k-2;j++)

{

max=a[0];nom=0; for(i=1;i<k;i++)

if (a[i]>max)

{

max=a[i]; nom=i;

}

b=a[k-1];

a[k-1]=a[nom]; a[nom]=b;

k--;

}

cout<<"Massiv a\n"; for(i=0;i<n;i++)

cout<<"a("<<i<<")="<<a[i]<<"\t"; cout<<endl;

return 0;

}

# Обработка матриц в С++

Двумерный массив (матрицу) можно объявить так:

###### тип имя\_переменной [n][m];

где **n** – количество строк в матрице(строки нумеруются от **0** до **n-1), m** – количество столбцов (столбцы нумеруются от **0** до **m-1**).

Например, Int x[10][15];

Описана матрица x, состоящая из 10 строк и 15 столбцов (строки нумеруются от 0 до 9, столбцы от 0 до 14).

Для обращения к элементу матрицы необходимо указать ее имя, и в квадратных скобках номер строки, а затем в квадратных скобках – номер столбца. Например, x[2][4] – элемент матрицы x, находящийся в третьей строке и пятом столбце.

В С++ можно описать многомерные массивы, которые можно объявить с помощью оператора следующей структуры:

###### тип имя\_переменной [n1][n2]…[nk];

В С++ ***ввод матрицы*** можно реализовать следующим образом:

cout<<"N=";

cin>>N;

cout<<"M=";

cin>>M;

cout<<"\n Vvedite A \n";

for(i=0;i<N;i++)

for(j=0;j<M;j++)

cin>>a[i][j];

***Построчный вывод матрицы*** можно реализовать следующим образом:

cout<<"\n Matrica A\n";

for(i=0;i<N;cout<<endl,i++)

for(j=0;j<M;j++)

cout<<"\t"<<a[i][j]);

***Пример***. Найти сумму элементов матрицы, лежащих выше главной диагонали.

int main()

{int S,i,j,N,M,a[20][20];

cout<<"N=";cin>>N; cout<<"M=";cin>>M; cout<<"Input Matrix A"<<endl;

for(i=0;i<N;i++) for(j=0;j<M;j++)

cin>>a[i][j]; for(S=i=0;i<N;i++)

for(j=0;j<M;j++)

if (j>i) S+=a[i][j]; cout<<"S="<<S<<endl;}

## 3.4 Указатели

При выполнении любой программы, все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде, например 0x100 или 0x200.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или её зависание.

Если мы объявляем переменные статично, они остаются в памяти до того момента, как программа завершит свою работу, после чего уничтожаются.

Такой подход может быть использован в простых примерах и несложных программах, которые не требуют большого количества ресурсов. Если же наш проект является огромным программным комплексом с высоким функционалом, объявлять таким образом переменные неразумно.

Естественно, все это занимает какое-то место в оперативной памяти компьютера. Если не уничтожать неиспользуемые объекты, очень скоро они заполнят весь объем ресурсов ПК.

По этим причинам, в большинстве языков, в том числе и C++, существует понятие указателя. Указатель — это переменная, хранящая в себе адрес ячейки оперативной памяти, например **0x100**.

Мы можем обращаться, например, к [массиву](http://code-live.ru/post/cpp-arrays/) данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступно для других переменных.

Ниже приведены примеры обращения к переменным напрямую и через указатель.

**Пример использования статических переменных**

#include <iostream>

using namespace std;

int main()

{

int a; // Объявление статической переменной

int b = 5; // Инициализация статической переменной b

a = 10;

b = a + b;

cout << "b is " << b << endl;

return 0;

}

В это примере мы объявляем/инициализируем статичные переменные **a** и **b**, после чего выполняем различные операции напрямую с ними.

**Пример использования динамических переменных**

#include <iostream>

using namespace std;

int main()

{

int \*a = new int; // Объявление указателя для переменной типа int

int \*b = new int(5); // Инициализация указателя

\*a = 10;

\*b = \*a + \*b;

cout << "b is " << \*b << endl;

delete b;

delete a;

return 0;

}

Синтаксис первого примера вам уже должен быть знаком. Во втором примере мы оперируем динамическими переменными посредством указателей. Рассмотрим общий синтаксис указателей в C++.

**Выделение памяти** осуществляется с помощью оператора [**new**](http://ru.wikipedia.org/wiki/New_(C++)) и имеет вид: **тип\_данных \*имя\_указателя = new тип\_данных;**, например **int \*a = new int;**. После удачного выполнения такой операции, в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа **int**.

Логично предположить, что для разных типов данных выделяется разное количество памяти. Следует быть особенно осторожным при работе с памятью, потому что именно ошибки программы, вызванные утечкой памяти, являются одними из самых трудно находимых. На отладку программы в поисках одной ничтожной ошибки, может уйти час, день, неделя, в зависимости от упорности разработчика и объема кода.

**Инициализация значения**, находящегося по адресу указателя выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением: **тип данных \*имя\_указателя = new тип\_данных (значение)**. В нашем примере это **int \*b = new int(5)**.

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом **&**. перед ним.

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель **b** во втором примере, мы пишем cout << "Address of b is " << &b << endl;. В примере получено значение **0x1aba030**. У вас оно может быть другим, потому что адреса в оперативной памяти распределяются таким образом, чтобы максимально уменьшить фрагментацию. Поскольку, в любой системе список запущенных процессов, а также объем и разрядность памяти могут отличаться, система сама распределяет данные для обеспечения минимальной фрагментации.

Для того чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс \***. Данная операция называется **разыменованием указателя**.

Во втором примере мы выводим на экран значение, *которое находится в ячейке памяти* (у меня это **0x1aba030**): cout << "b is " << \*b << endl; . В этом случае необходимо использовать знак **\***.

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере — \*b = \*a + \*b;.

* Когда мы оперируем **данными**, то используем знак **\***
* Когда мы оперируем **адресами**, то используем знак **&**

Для того, чтобы освободить память, выделенную оператором **new**, используется оператор [delete](http://ru.wikipedia.org/wiki/Delete_(C++)).

**Пример освобождения памяти**

#include <iostream>

using namespace std;

int main()

{

// Выделение памяти

int \*a = new int;

int \*b = new int;

float \*c = new float;

// ... Любые действия программы

// Освобождение выделенной памяти

delete c;

delete b;

delete a;

return 0;

}

При использовании оператора delete для указателя, знак \* не используется.

## 3.5 Динамические массивы

В рассмотренных ранее примерах массивы являются статическими, их размер задается при описании. Это не всегда удобно, кроме того, размер массива не всегда известен заранее. В С++ существуют динамические массивы – это массивы переменной длины, они определяются с помощью указателей.

Например, необходимо создать динамический массив из **N** элементов, где значение **N** задается пользователем.

## Создание динамического массива

#include <iostream>

using namespace std;

int main()

{

int num; // *размер массива*

cout << "Enter integer value: ";

cin >> num; // *получение от пользователя размера массива*

int \*p\_darr = new int[num]; // *Выделение памяти для массива*

for (int i = 0; i < num; i++) {

// *Заполнение массива и вывод значений его элементов*

p\_darr[i] = i;

cout << "Value of " << i << " element is " << p\_darr[i] << endl;

}

delete [] p\_darr; // *очистка памяти*

return 0;

}

Синтаксис выделения памяти для массива имеет вид указатель = new тип[размер]. В качестве размера массива может выступать любое целое положительное значение.

***Пример****.* Найти сумму элементов динамического массива вещественных чисел.

#include <iostream>

#include <malloc.h>

using namespace std;

int main()

{

int i,n; float \*a,s;

cout<<"n="; cin>>n;

// *Выделение памяти для динамического массива а*.

a=(float \*)malloc(n\*sizeof(float));

cout << "Vvedite massiv A";

for(i=0;i<n;i++)

// *cin>>a[i];*

cin>>\*(a+i);

for(s=0,i=0;i<n;i++)

//*s+=a[i];*

s+=\*(a+i);

cout << "S="<<s;

//*Освобождение памяти*.

free(a);

return 0;

}

***Пример*** поиска суммы элементов динамического массива вещественных чисел с помощью операции new.

#include <iostream>

using namespace std;

int n;

float \*a;

cin>>n;

a=new float[n];

int main()

{

int i,n;

float \*a,s;

cout<<"n=";

cin>>n;

// *Выделение памяти для динамического массива а*.

a=new float[n];

cout << "Vvedite massiv A";

for(i=0;i<n;i++)

// *cin>>a[i]*;

cin>>\*(a+i);

for(s=0,i=0;i<n;i++)

//*s+=a[i]*; s+=\*(a+i);

cout << "S="<<s;

//*Освобождение памяти*.

delete [] a;

return 0;

}

Указатели можно использовать в качестве аргументов функций. Так, если в роли параметра функции выступает массив, то в функцию передается указатель на его первый элемент. С++ автоматически передает массив в функцию, используя его адрес. В результате вызываемые функции могут изменять значения элементов в исходных массивах и возвращать их в главную функцию. Информация о количестве элементов массива должна передаваться через отдельный параметр.

**Динамические матрицы**

**1-й способ** работы с динамическими матрицами основан на работе с одинарным указателем. При работе с динамическими матрицами следует помнить, что выделенный участок памяти под матрицу A(N,M) представляет собой участок памяти размером NxM элементов. Поэтому выделение памяти будет выглядеть следующим образом:

###### A=(тип \*) calloc(n\*m, sizeof(тип))

или

###### A=(тип \*) malloc(n\*m\*sizeof(тип))

Для обращения к элементу Ai,j необходимо, но номеру строки i и номеру столбца j вычислить номер этого элемента k в одномерном динамическом массиве. Учитывая, что в массиве элементы нумеруются с нуля k=i.M+j.

Статический элемент матрицы **a[i][j]**записывается как **\*(a+i\*m+j)**

**1-й способ** работы с динамическими матрицами **о**снован на использовании двойного указателя – указателя на указатель.

float \*\*a;

Это указатель на float \*, или указатель на массив.

Выделение и очистка памяти в этом случае осуществляется следующим образом:

void main()

{

int n,m; float \*\*a;

//Создали массив указателей в количестве **n** штук на **float**, каждый элемент массива, является адресом, в котором хранится указатель на float.

a=new float \*[n];

//Осталось определить значение этого указателя. Для этого организуем цикл от 0 до n-1, в котором каждый указатель будет адресовать участок памяти, в котором хранится m элементов.

for(i=0;i<n;i++)

a[i]=new float(m);

//Обращение к элементу матрицы в этом случае будет идти стандартным образом a[i]][j]

//По окончании работы необходимо освободить память

for(i=0;i<n;i++)

delete a[i];

delete [];

}

***Пример***. Задана матрица *A*(*N,M*). Поменять местами ее максимальный и минимальный элементы.

#include <iostream>

using namespace std;

//Решение задачи с использованием одномерного динамического массива

int main()

{

int i, j, imax, jmax, imin, jmin, N, M;

double min, max, b, \*a;

cout<<"N=";

cin>>N;

cout<<"M=";

cin>>M;

A=(double \*) calloc(n\*m, sizeof(double));

cout<<"Input Matrix A"<<endl;

for(i=0;i<N;i++)

for(j=0;j<M;j++)

cin>>\*(a+i\*M+j);

for(max=min=\*a,imax=jmax=imin=jmin=i=0;i<N;i++)

for(j=0;j<M;j++)

{

if ((\*(a+i\*M+j))>max)

{max=\*(a+i\*M+j); imax=i;jmax=j;}

if ((\*(a+i\*M+j))<min)

{min=\*(a+i\*M+j); imin=i;jmin=j;}}

b=\*(a+imax\*M+jmax);

\*(a+imax\*M+jmax)=\*(a+imin\*M+jmin);

\*(a+imin\*M+jmin)=b;

cout<<"Output Matrix A"<<endl;

for(i=0;i<N;cout<<endl,i++)

for(j=0;j<M;j++)

cout<<\*(a+i\*M+j)<<"\t";}

## 3.6 Строки

**Строка** – это последовательность (массив) символов (типа [*char*](http://natalia.appmat.ru/c&c++/lezione1.php#l1_4)), которая заканчивается специальным символом – признаком конца строки. Это символ записывается как **'\0'** (не путайте с символом переноса строки **'\n'**) и равен 0. При вводе строки символ конца строки добавляется автоматически. Все функции работы со строками должны ориентироваться на этот символ. Если требуется сформировать новую строку, то обязательно надо добавлять признак конца строки. Если этого не сделать, то при дальнейшей работе возникнут ошибки.

'a' // Символьная константа - один символ

"a" // Строковый литерал - массив из **двух** символов 'a' и '\0', заменяется на **адрес**.

Например, описание строки из 25 символов должно выглядеть так:

char s[25]; Здесь элемент s[25] предназначен для хранения символа конца строки.

char s[7]="Привет";

Можно описать и массив строк

char m[3][25]={"Пример ", "использования", " строк"}. Определен массив из 3 строк по 25 байт в каждой.

Для работы с указателями можно использовать и указатели (char \*). Адреспервого символа и будет начальным значением указателя.

*Строковым литералом* называется последовательность символов, заключённых в двойные кавычки. В строковом литерале на один символ больше, чем используется при его записи – добавляется символ '\0'.

Тип строкового литерала есть «массив с надлежащим количеством константных символов». Строковый литерал можно присвоить переменной типа *char\**. Однако изменение строкового литерала через такой указатель является ошибкой.

То, что строковые литералы являются константами, позволяет при реализации произвести значительную оптимизацию методов хранения и доступа к строковым литералам. Если же нужна строка, которую можно модифицировать, следует объявить и инициализировать массив символов.

Память под строковые литералы выделяется статически, поэтому их можно возвращать в качестве значения функции.

const char \*error\_massage()

{ return "Недостаточно параметров"; } // После выхода из функции память, содержащая строку, не будет освобождена.

Поскольку мы не знаем, сколько в строке содержится символов, но знаем, что в конце стоит символ конца строки, цикл для обработки строки пишется следующим образом:

for (int i = 0; str[i] != '\0'; i++) { **...** }.

Можно опустить сравнение с нулем, для C++ это будет эквивалентно:

for (int i = 0; str[i]; i++) { **...** }.

Можно использовать указатели для обработки строк:

char str[50], \*p;

...

for (p = str; \*p; p++) { **...** }.

Заголовки стандартных функций работы со строками хранятся в файле <string.h>. Основными из этих функций являются:

Определение длины строки – int **strlen**(const char \***str**);

Сравнение строк – int **strcmp**(const char \***str1**, const char \***str2**);

Копирование – char \***strcpy**(char \***str1**, const char \***str2**);

Конкатенация строк – char \***strcat**(char \***str1**, const char \***str2**);

Поиск символа в строке – char \***strchr**(const char \***str**, char **c**);

Поиск подстроки – char \***strstr**(const char \***str1**, const char \***str2**);

**Ввод/вывод строки**:

Ввод строки до пробела или другого разделителя – функция *scanf* с форматом *%s*;

Ввод строки, содержащей пробелы – char \***gets** (char \***buffer**);

Ввод строки из файла, **n** задаёт максимальное количество символов для ввода – char \***fgets**(char \***string**, int **n**, FILE \***stream**);

Вывод строки с форматированием – функция *printf* с форматом *%s*;

Вывод строки – int **puts**(const char \***string**);

Вывод строки в файл – int **fputs**(const char \***string**, FILE \***stream**);

***Пример 1*.** Функция, которая меняет все вхождения буквы «я» на «а», «а» – на «б», «б» – на «в» и т.д. Остальные символы остаются без изменения

char\* Change(char \*str)

{ char \*p;

for (p = str; \*p; p++)

if (\*p == 'я')

\*p = 'а';

else if ('а'<= \*p && \*p <= 'ю')

(\*p)++;

return str;

}

***Пример 2.*** Функция, формирующая строку, состоящую из символов исходной строки, не входящих в заданное множество

char\* NotEntered(char \*dest, const char \*source, const char \*symbols)

{ int i, j;

for (i = 0, j = 0; source[i]; i++)

if (!strchr(symbols, source[i]))

dest[j++] = source[i];

**dest[j] = '\0';**

return dest;

} // Обязательно добавляем признак конца строки в формируемую строку

## 3.7 Функции

*Функция* — именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Одна из функций программы должна иметь имя main. Функция начинает выполняться в момент вызова. Любая функция должна быть объявлена и определена.

Конструкция функции имеет следующий вид:

<тип возвращаемого значения> name (<аргументы функции>

{

/ / ….

return <выражение>;

}

Аргументы функции – значения, которые можно передать ей при вызове. В возвращаемом значении указывается результат, который функция возвращает по окончании работы. Например, в вызове функции возведения в квадрат square(10) 10 – это аргумент, а возвращаемое значение равно 100. Если какой-либо элемент отсутствует, вместо него используется ключевое слово void.

***Простые функции***

Простая функция sumSequence() возвращает вычисляемое ею целое значение. Функции могут возвращать значение любого стандартного типа.

Функции различаются по типу возвращаемого значения. Так, целочисленной функцией называют ту, которая возвращает целое значение. Функция, которая ничего не возвращает, известна как void-функция.

Пример функции, выполняющей некоторые действия, но не возвращающей никаких значений.

void echoSquare()

{

cout << ” Vvedite znachenie: “;

cin >> value;

cout << endl<< value\*value << endl;

return;

}

***Функции с аргументами***

Аргументами функции называют значения, которые передаются функции во время вызова. Функции могут иметь несколько аргументов. В этом случае аргументы разделяются запятыми, например:

int product(int arg1, int arg2)

{

return arg1\*arg2;

}

***Функция main( )***

Ключевое слова main( ) в нашей стандартной программе означает ни что иное, как функцию, не требующую прототипа.

## 3.8 Структуры и объединения

**Структуры**

**Структура** является собранием одного или более объектов (переменных, массивов, указателей, других объектов), которые для удобства работы с ними объединены под одним именем.

Определение структуры состоит из двух шагов:

1. Объявление структуры (задание нового типа данных определенного пользователем), структура состоит из полей;

struct student

{

char fio[30];// определено поле fio

char group[8];// определено поле group int year;

int informatika, math, fizika, history;

}

1. Определение переменных типа структура;

student Vasya, ES[50];

Для обращения к полям структуры надо указать имя переменной и через точку имя поля

**Structura.Pole**

Например,

Vasya.Year

ES[4].math

**Структура** позволяет объединить в одном объекте совокупность значений, которые могут иметь различные типы. Однако в языке С реализован очень ограниченный набор операций над структурами как единым целым: передача функции в качестве аргумента, возврат в качестве значения функции, получение адреса. Можно присваивать одну структуру другой, если они имеют одинаковый тип.

Объявление структуры задает имя структурного типа и/или последовательность объявлений переменных, называемых элементами структуры. Эти элементы могут иметь различные типы.

struct [*<тег>*] {*<список объявлений элементов>*} *<описатель>* [, *<описатель>* ...];

struct *<тег>* *<описатель>* [, *<описатель>* ...];

*Тег* предназначен для различения нескольких структур, объявленных в одной программе.

Список объявлений элементов представляет собой последовательность из одного или более объявлений переменных. Каждая переменная, объявленная в этом списке, называется *элементом структуры*. Особенность синтаксиса объявлений элементов структуры состоит в том, что они не могут содержать [*спецификаций класса памяти*](http://natalia.appmat.ru/c&c++/lezione9.php#l9_classi_di_memoria) и [*инициализаторов*](http://natalia.appmat.ru/c&c++/lezione2.php#l2_ini). Элементы структуры могут иметь [*базовый тип*](http://natalia.appmat.ru/c&c++/lezione1.php#l1_4), либо быть [*массивом*](http://natalia.appmat.ru/c&c++/lezione3.php), [*указателем*](http://natalia.appmat.ru/c&c++/lezione5.php#l5_1), [*объединением*](http://natalia.appmat.ru/c&c++/lezione6.php#l6_3) или структурой.

struct

{ char str[50];

int a, b; // *Объявляем структуру, не задавая тег*

} s; // *и сразу же объявляем переменную*

struct S

{ char str[50];

int a, b;

};

// *Объявляем структуру с тегом S*

struct S s; // *Объявляем переменную*

Элемент структуры не может быть структурой того же типа, в которой он содержится. Однако он может быть указателем на тип структуры, в которую он входит. Размер указателя стандартный, поэтому компилятор знает, сколько памяти потребуется под указатель. Для работы с указателем надо знать размер типа, на который он указывает, но к моменту работы с указателем структура будет полностью объявлена, и, следовательно, размер её будет известен.

Идентификаторы элементов структуры должны различаться между собой. Идентификаторы элементов разных структур могут совпадать.

Для **инициализации** структуры, как и других составных типов, надо записать список инициализаторов через запятую в фигурных скобках.

struct S s = {"Str", 0, 1}; // *Используем тег S, объявленный в предыдущем примере*

Выбор элемента структуры осуществляется с помощью одной из следующих конструкций:

<переменная> . <идентификатор элемента структуры>

<указатель> -> <идентификатор элемента структуры>

Выражение выбора элемента позволяет получить доступ к элементу структуры. Выражение имеет значение и тип выбранного элемента.

struct S s, \*p = &s;

s.a = 10;

p->b = 20; // *Объявляем переменную* ***s*** *и указатель* ***p****, в который заносим адрес переменной* ***s***

Две структуры являются разными типами, даже если их объявления полностью совпадают.

# Объединения

**Объединение** позволяет в разные моменты времени хранить в одном объекте значения разных типов. В процессе объявления объединения с ним ассоциируется набор типов значений, которые могут храниться в данном объединении. В каждый момент времени объединение может хранить значение только одного типа из набора. Контроль над тем, какого типа значение хранится в данный момент в объединении, возлагается на программиста.

union [*<тег>*] {*<список объявлений элементов>*} *<описатель>* [, *<описатель>* ...];

union *<тег>* *<описатель>* [, *<описатель>* ...];

*Тег* предназначен для различения нескольких объединений, объявленных в одной программе.

Память, которая выделяется переменной типа *объединение*, определяется размером наиболее длинного элемента объединения. Все элементы объединения размещаются в одной и той же области памяти с одного и того же адреса. Значение текущего элемента объединения теряется, когда другому элементу объединения присваивается значение.

# РАЗДЕЛ 4. ПРОГРАММИРОВАНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ СРЕДЕ

* 1. **Основные принципы объектно-**

**ориентированного программирования**

**История создания объектно-ориентированного программирования**

Практически сразу после появления языков третьего поколения (1967) ведущие специалисты в области программирования выдвинули идею преобразования постулата фон Неймана: "данные и программы неразличимы в памяти машины". Их цель заключалась в максимальном сближении данных и кода программы. Решая поставленную задачу, они столкнулись с задачей, решить которую без декомпозиции оказалось невозможно, а традиционные структурные декомпозиции не сильно упрощали задачу. Усилия многих программистов и системных аналитиков, направленные на формализацию подхода, увенчались успехом.

Были разработаны три основополагающих принципа того, что потом стало называться объектно-ориентированным программированием (ООП): наследование; инкапсуляция; полиморфизм.

Результатом их первого применения стал язык Симула-1 (Simula-1), в котором был введен новый тип — объект. В описании этого типа одновременно указывались данные (поля) и процедуры, их обрабатывающие — методы. Родственные объекты объединялись в классы, описания которых оформлялись в виде блоков программы. При этом класс можно использовать в качестве префикса к другим классам, которые становятся в этом случае подклассами первого. Впоследствии Симула-1 был обобщен, и появился первый универсальный объектно-ориентированный язык программирования — Симула-67 (67 — по году создания).

Объектно-ориентированный подход помогает справиться с такими сложными проблемами, как уменьшение сложности программного обеспечения; повышение надежности программного обеспечения; обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов; обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

**Введение в объектно-ориентированный подход к разработке программ**

В основу структурного мышления положены структуризация и декомпозиция окружающего мира. Задача любой сложности разбивается на подзадачи, а те в свою очередь разбиваются далее и т. д., пока каждая подзадача не станет простой, соответствующей модулю.

Модуль в понятии структурного программирования — это подпрограмма (функция или процедура), оформленная определенным образом и выполняющая строго одно действие. Методы структурного проектирования используют модули в качестве строительных блоков программы, а структура программы представляется иерархией подчиненности модулей.

Модуль ООП — файл описаний объектов и действий над ними.

Методы объектно-ориентированного проектирования используют в качестве строительных блоков объекты. Каждая структурная составляющая является самостоятельным объектом, содержащим свои собственные коды и данные. Благодаря этому уменьшена или отсутствует область глобальных данных.

Объектно-ориентированное мышление адекватно способу естественного человеческого мышления, ибо человек мыслит "образами" и "абстракциями". Чтобы проиллюстрировать некоторые из принципов объектно-ориентированного мышления, обратимся к следующему примеру, основанному на аналогии мира объектов реальному миру.

Рассмотрим ситуацию из обыденной жизни. Допустим, вы решили поехать в другой город на поезде. Для этого вы приходите на ближайшую железнодорожную станцию и сообщаете кассиру номер нужного поезда и дату, когда планируете уехать. Теперь можете быть уверены, что ваш запрос будет удовлетворен (при условии, что вы покупаете билет заранее).

Таким образом, для решения своей проблемы вы нашли *объект*"кассир железнодорожной кассы" и передали ему *сообщение,*содержащее запрос. *Обязанностью*объекта "кассир железнодорожной кассы" является удовлетворение запроса.

У кассира имеется некоторый определенный *метод,*или эвроритм, или последовательность операций (процедура), которые используют работники кассы для выполнения вашего запроса. Имеются у кассира и другие методы, например по сдаче денег, — инкассации

Вам совершенно не обязательно знать не только детально метод, который используется кассиром, но даже весь набор методов работы кассира. Однако если бы вас заинтересовал вопрос как работает кассир, то обнаружили бы, что кассир пошлет свое сообщение автоматизированной системе железнодорожного вокзала. Та, в свою очередь, примет необходимые меры и т. д. Тем самым ваш запрос, в конечном счете, будет удовлетворен через последовательность запросов, пересылаемых от одного объекта к другому.

Таким образом, действие в объектно-ориентированном программировании инициируется посредством передачи сообщений объекту, ответственному за действие. Сообщение содержит запрос на осуществление действия конкретным объектом и сопровождается дополнительными аргументами, необходимыми для его выполнения. Пример аргументов вашего сообщения: дата отъезда, номер поезда, тип вагона. Сообщения кассира: дайте паспорт, заплатите такую-то сумму, получите билет и сдачу.

Кассир, находящийся на рабочем месте, не обязан отвлекаться от работы для пустой болтовни с покупателем билета, например, сообщать ему свой домашний телефон или сумму денег, находящуюся в сейфе кассы. Таким образом, кассир взаимодействует с другими объектами ("покупатель билета", "автоматизированная система", "инкассатор", "бригадир" и т. д.) только по строго регламентированному *интерфейсу.*Интерфейс — это набор форматов допустимых сообщений. Для исключения возможных, но недопустимых сообщений используется механизм *сокрытия информации*(инструкция, запрещающая кассиру болтать впустую на рабочем месте).

Помимо методов, кассир для успешной работы должен располагать наборами чистых бланков билетов, купюрами и монетами наличных денег (хотя бы для сдачи покупателю). Такие наборы хранятся в особых отсеках кассы, особых коробках. Места хранения этих наборов называют *полями объектов.*В программах полям объектов соответствуют переменные, которые могут хранить какие-то значения.

Покупатель билета не может положить деньги непосредственно в отсек кассового аппарата или сейф кассира, а также самостоятельно отсчитать себе сдачу. Таким образом, кассир как бы заключен в оболочку, или капсулу, которая отделяет его и покупателя от лишних взаимодействий. Помещение кассы (капсула) имеет особое устройство, исключающее доступ покупателей билетов к деньгам. Это и есть *инкапсуляция*объектов, позволяющая использовать только допустимый интерфейс — обмен информацией и предметами только посредством допустимых сообщений, а может быть, еще и подаваемых в нужной последовательности. Именно только через вызов сообщениями особых методов осуществляется обмен данных, отделяя покупателей от полей. Благодаря инкапсуляции покупатель может лишь отдавать в качестве оплаты деньги за билет в форме сообщения с аргументом "сумма". Аналогично, но в обратном направлении кассир возвращает сдачу.

Вы можете передать свое сообщение, например, объекту "свой приятель", и он его, скорее всего, поймет, и как результат — действие будет выполнено (а именно билеты будут куплены). Но если вы попросите о том же объект "продавец магазина", у него может не оказаться подходящего метода для решения поставленной задачи. Если предположить, что объект "продавец магазина" вообще воспримет этот запрос, то он "выдаст" надлежащее сообщение об ошибке. В отличие от программ, люди работают не по алгоритмам, а по эвроритмам. Человек может самостоятельно менять правила методов своей работы. Так, продавец магазина при виде аргумента "очень большая сумма", может закрыть магазин и побежать покупать железнодорожный билет. Напомним, что такие ситуации для программ пока еще невозможны.

Различие между вызовом процедуры и пересылкой сообщения состоит в том, что в последнем случае существует определенный получатель и интерпретация (т. е. выбор подходящего метода, запускаемого в ответ на сообщение), которая может быть различной для разных получателей.

Обычно конкретный объект-получатель неизвестен вплоть до выполнения программы, так что определить, какой метод, какого объекта будет вызван, заранее невозможно (конкретный кассир заранее не знает, кто и когда из конкретных покупателей обратится к нему). В таком случае говорят, что имеет место позднее связывание между сообщением (именем процедуры или функции) и фрагментом кода (методом), исполняемым в ответ на сообщение. Эта ситуация противопоставляется раннему связыванию (на этапе компилирования или компоновки программы) имени с фрагментом кода, что происходит при традиционных вызовах процедур.

Фундаментальной концепцией в объектно-ориентированном программировании является понятие *классов.*Все объекты являются представителями, или экземплярами, классов. Например: у вас наверняка есть примерное представление о реакции кассира на запрос о заказе билетов, поскольку вы имеете общую информацию о людях данной профессии (например, кассире кинотеатра) и ожидаете, что он, будучи представителем данной категории, в общих чертах будет соответствовать шаблону. То же самое можно сказать и о представителях других профессий, что позволяет разделить человеческое общество на определенные категории по профессиональному признаку (на классы). Каждая категория в свою очередь делится на представителей этой категории. Таким образом, человеческое общество представляется в виде иерархической структуры с наследованием свойств классов объектов всех категорий. В корне такой классификации может находиться класс "HomoSapience" или даже класс "млекопитающие".

Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Классы могут быть организованы в иерархическую структуру с *наследованием*свойств. *Класс-потомок*наследует атрибуты *родительского класса,*расположенного ниже в иерархическом дереве (если дерево иерархии наследования растет вверх). *Абстрактный родительский класс*— это класс, не имеющий экземпляров объектов. Он используется только для порождения потомков. Класс "HomoSapience", скорее всего, будет абстрактным, поскольку для практического применения, например работодателю, экземпляры его объектов не интересны.

Итак, пусть абстрактным родительским классом у работодателя будет класс "трудоспособный человек", который включает методы доступа к внутренним данным, а также поля самих внутренних данных: фамилия; имя; отчество; дата рождения; домашний адрес; домашний телефон; сведения об образовании; сведения о трудовом стаже и т. д. От данного класса могут быть унаследованы классы: "кассир", "водитель автомобиля", "музыкант". Класс "кассир" располагает методами работы: общение с клиентом по правилам, получение денег, выдача денег, общение с инкассатором и т. д. От класса "кассир" могут быть унаследованы классы: "кассир, выдающий зарплату", "кассир железнодорожной кассы". Кассир железнодорожной кассы отличается от кассира, выдающего зарплату, дополнительными знаниями и навыками работы.

От класса "кассир железнодорожной кассы" могут быть получены экземпляры объектов: "кассир кассы № 1", "кассир кассы № 2", "кассир кассы № 3" и т. д.

В помещении большого вокзала можно обнаружить множество одинаково оборудованных объектов — касс. Однако среди касс можно выделить различающиеся кассы: суточные, предварительные, воинские, работающие по бронированию билетов и т. д. Для того чтобы начальнику вокзала поменять один вид кассы на другой, нет необходимости перестраивать помещение кассы и менять оборудование. Ему достаточно заменить в кассе кассира с одними навыками на кассира с другими навыками. Кассир вставляет табличку с новой надписью вида кассы — и все. Заметим, что смена функции касс произошла без остановки работы вокзала. Такая замена становится простой именно потому, что все помещения касс имеют одинаковый интерфейс с кассирами и клиентами. Теперь разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на запросы разные операции.

Ассоциация запроса с объектом и одной из его операций во время выполнения называется *динамическим связыванием.*Динамическое связывание позволяет во время выполнения подставить вместо одного объекта другой, если он имеет точно такой же интерфейс. Такая взаимозаменяемость называется *полиморфизмом*и является еще одной фундаментальной особенностью объектно-ориентированных систем.

Пусть, согласно произведенной классификации, объекты "скрипач с фамилией Петров" и "водитель автомобиля Сидоров" будут экземплярами разных классов. Для того чтобы получить объект "Иванов, являющийся одновременно скрипачом и водителем", необходим особый класс, который может быть получен из классов "скрипач" и "водитель автомобиля" *множественным наследованием*. Теперь работодатель, послав особое *сообщение делегирования,*может поручить (делегировать) объекту "Иванов" выполнять функцию либо водителя, либо скрипача. Объект "Иванов", находящийся за рулем автомобиля, не должен начать играть на скрипке. Для этого должен быть реализован механизм самоделегирования полномочий — объект "Иванов", находясь за рулем, запрещает сам себе игру на скрипке. Таким образом, понятие обязанности или ответственности за выполнение действия является фундаментальным в объектно-ориентированном программировании.

*В системах программирования с отсутствующим множественным наследованием задачи, требующие множественного наследования, всегда могут быть решены композицией (агрегированием) с последующим делегированием полномочий.*

*Композиция объектов —*это реализация составного объекта, состоящего из нескольких совместно работающих объектов и образующих единое целое с новой, более сложной функциональностью.

Наследование класса определяется статически на этапе компиляции; его проще использовать, поскольку оно напрямую поддержано языком программирования.

Но у наследования класса есть и минусы. Во-первых, нельзя изменить унаследованную от родителя реализацию во время выполнения программы, поскольку само наследование фиксировано на этапе компиляции. Во-вторых, родительский класс нередко, хотя бы частично, определяет физическое представление своих подклассов. Поскольку подклассу доступны детали реализации родительского класса, то часто говорят, что *наследование нарушает инкапсуляцию.*Реализации подкласса и родительского класса настолько тесно связаны, что любые изменения последней требуют изменять и реализацию подкласса.

Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик, поскольку доступ к объектам осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку при реализации объекта кодируются прежде всего его интерфейсы, то зависимость от реализации резко снижается.

Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), и поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе.

Это подводит еще к одному правилу объектно-ориентированного проектирования: *предпочитайте композицию наследованию класса.*

В идеале, чтобы добиться повторного использования кода, вообще не следовало бы создавать новые компоненты. Хорошо бы, чтобы можно было получить всю нужную функциональность, просто собирая вместе уже существующие компоненты. На практике, однако, так получается редко, поскольку набор имеющихся компонентов все же недостаточно широк. Повторное использование за счет наследования упрощает создание новых компонентов, которые можно было бы применять со старыми. Поэтому наследование и композиция часто используются вместе.

Тем не менее опыт показывает, что проектировщики злоупотребляют наследованием. Нередко программы могли бы стать проще, если бы их авторы больше полагались на композицию объектов.

С помощью *делегирования*композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование. При делегировании в процесс обработки запроса вовлечено *два*объекта: получатель поручает выполнение операций другому объекту — *уполномоченному.*Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную-член (в C++) или переменную self (в Smalltalk). Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту, чтобы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса.

Например, вместо того чтобы делать класс Window (окно) подклассом класса Rectangle (прямоугольник) — ведь окно является прямоугольником, — мы можем воспользоваться внутри Window поведением класса Rectangle, поместив в класс Window переменную экземпляра типа Rectangle и делегируя ей операции, специфичные для прямоугольников. Другими словами, окно не *является*прямоугольником, а *содержит*его. Теперь класс Window может явно перенаправлять запросы своему члену Rectangle, а не наследовать его операции.

Главное достоинство делегирования в том, что оно упрощает композицию поведения во время выполнения. При этом способ комбинирования поведения можно изменять. Внутреннюю область окна разрешается сделать круговой во время выполнения простой подставкой вместо экземпляра класса Rectangle экземпляра класса Circle. Предполагается, конечно, что оба эти класса имеют одинаковый тип.

У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Заключается он в том, что динамическую, в высокой степени параметризованную программу труднее понять, чем статическую. Есть, конечно, и некоторая потеря машинной производительности, но неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения, а не усложнения. Нелегко сформулировать правила, ясно говорящие, когда следует пользоваться делегированием, поскольку эффективность его зависит от контекста и личного опыта программиста.

Таким образом, можно выделить следующие фундаментальные характеристики объектно-ориентированного мышления:

* Любой предмет или явление могут рассматриваться как объект.
* Объект может размещать в своей памяти (в полях) личную информацию, независимую от других объектов. Рекомендуется использовать инкапсулированный (через особые методы) доступ к информации полей.
* Объекты могут иметь открытые по интерфейсу методы обработки сообщений. Сами сообщения вызовов методов посылаются другими объектами, но для осуществления разумного интерфейса между объектами некоторые методы могут быть скрыты.
* Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие (метод). Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. Объект — получатель сообщения — обрабатывает сообщения своими внутренними методами.
* Каждый объект является представителем класса, который выражает общие свойства объектов данного класса в виде одинаковых списков набора данных (полей) в своей памяти и внутренних методов, обрабатывающих сообщения. В классе методы задают поведение объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
* Классы организованы в единую квазидревовидную структуру с общим корнем, которая называется иерархией наследования. Обычно корень иерархии направлен вверх. При множественном наследовании ветви могут срастаться, образуя сеть наследования. Память и поведение, связанные с экземплярами определенного класса, автоматически являются доступными любому классу, расположенному ниже в иерархическом дереве.
* Благодаря полиморфизму — способности подставлять во время выполнения вместо одного объекта другой, с совместимым интерфейсом, в периоде выполнения одни и те же объекты могут разными методами исполнять одни и те же запросы сообщений.
* Композиция является предпочтительной альтернативой множественному наследованию и позволяет изменять состав объектов агрегата в процессе выполнения программы.
* Структура объектно-ориентированной программы на этапе выполнения часто имеет мало общего со структурой ее исходного кода. Последняя фиксируется на этапе компиляции. Ее код состоит из классов, отношения наследования между которыми неизменны. На этапе же выполнения структура программы — быстро изменяющаяся сеть из взаимодействующих объектов. Две эти структуры почти независимы.

**Основные понятия объектно-ориентированного программирования**

Объектно-ориентированное проектирование (ООП) — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированное программирование — это процесс реализации программ, основанный на представлении программы в виде совокупности объектов. ООП предполагает, что любая функция (процедура) в программе представляет собой метод объекта некоторого класса, причем класс должен формироваться в программе естественным образом, как только в программе возникает необходимость описания новых физических предметов или их абстрактных понятий (объектов программирования). Каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих классов, т. е. технология ООП иначе может быть названа как программирование "от класса к классу".

Класс в одном из значений этого термина обозначает тип структурированных данных.

Объект — это структурированная переменная типа класс. Каждый объект является представителем (экземпляром) определенного класса. В программе может быть несколько объектов, являющихся экземплярами одного и того же класса. Все объекты — экземпляры данного класса — аналогичны друг другу, поскольку имеют одинаковый интерфейс, один и тот же набор операций (методов) и полей, определяемых в их классе. Интерфейс класса иногда называют особенностями класса.

Класс является описанием того, как будет выглядеть и вести себя его представитель. Обычно проектируют класс как образование (матрицу), отвечающее за создание своих новых представителей (экземпляров или объектов). Экземпляр объекта создается при помощи особого метода класса, называемого конструктором, так как необходимо создать экземпляр, прежде чем он станет активным и начнет взаимодействовать с окружающим миром. Уничтожение экземпляров поддерживает сам активный экземпляр, имеющий соответствующий метод — деструктор.

*Объект*— это структурированная переменная типа класс, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии. Другими словами, объект — это расположенные в отдельном участке памяти:

— порция данных объекта или атрибуты исходных данных, называемые еще полями, членами данных *(data members),*значения которых определяют текущее состояние объекта;

— методы объекта *(methods,*в разных языках программирования еще называют подпрограммами, действиями, *member functions*или функциями-членами), реализующие действия (выполнение алгоритмов) в ответ на их вызов в виде переданного сообщения;

— часть методов, называемых свойствами (property), которые, в свою очередь, определяют поведение объекта, т. е. его реакцию на внешние воздействия (в ряде языков программирования свойства оформляются особыми операторами).

Объявления классов определяют уже описанные три характеристики объектов: поля объекта, методы объекта, свойства объектов. Также в объявлениях может указываться предок данного класса.

В соответствии с описанием класса внутри объекта данные и методы могут быть как открытыми по интерфейсу public, так и сокрытыми private.

Во время выполнения программы объекты взаимодействуют друг с другом посредством вызова методов вызываемого объекта — в этом и заключается передача сообщений. Для того чтобы объект послал сообщение другому объекту, в большинстве языков программирования требуется после указания имени вызываемого объекта записать вызов подпрограммы (метода) с соответствующим именем и указанием необходимых фактических параметров (аргументов). Получив сообщение, объект-получатель начинает выполнять код вызванной подпрограммы (метода) с полученными значениями аргументов. Таким образом, функционирование программы (выполнение всего алгоритма программы) осуществляется последовательным вызовом методов от одного объекта к другому.

Хотя можно получить прямой доступ к полям объекта, использование такого подхода не поощряется.

Одно из больших преимуществ ООП— это *инкапсуляция*, предназначенная для разрешения работы с данными в полях объектов только через сообщения. Для реализации методов обработки таких сообщений используются свойства. Свойства — это особым образом оформленные методы, предназначенные как для чтения и контролируемого изменения внутренних данных объекта (полей), так и выполнения действий, связанных с поведением объекта.

Так, например, если в заданном месте экрана уже отображена какая-то строка, и мы хотим изменить положение строки на экране, то мы посылаем объекту новое значение свойства в виде набора нужных координат. Далее свойство автоматически трансформируется в вызов метода, который изменит значение поля координат отображения строки и выполнит действия по уничтожению изображения строки на прежнем месте экрана, а также по отображению строки в новом месте экрана.

Можно выделить несколько преимуществ инкапсуляции:

* Надежность данных. Можно предотвратить изменение элемента данных, выполнив в свойстве (методе) дополнительную проверку значения на допустимость. Тем самым можно гарантировать надежное состояние объекта.
* Целостность ссылок. Перед доступом к объекту, связанному с данным объектом, можно удостовериться, что косвенное поле содержит корректное значение (ссылку на экземпляр).
* Предусмотренные побочные эффекты. Можно гарантировать, что каждый раз, когда выполняется обращение к полю объекта, синхронно с ним выполняется какое-либо специальное действие.
* Сокрытие информации. Когда доступ к данным осуществляется только через методы, можно скрыть детали реализации объекта. Позднее, если реализация изменится, придется изменить лишь реализацию методов доступа к полям. Те же части программы, которые использовали этот класс, не будут затронуты.

Весьма удобно рассматривать объекты как попытку создания активных данных. Смысл, вкладываемый в слова "объект представляет собой активные данные", основан на объектно-ориентированной парадигме выполнения операций, состоящей в посылке сообщений. В посылаемых объекту сообщениях указывается, что мы хотим, что бы он выполнил. Так, например, если мы хотим вывести на экране строку, то мы посылаем объекту строки сообщение, чтобы он изобразил себя. В этом случае строка — это уже не пассивный кусок текста, а активная единица, знающая, как правильно производить над собой различные действия.

Одна из фундаментальных концепций ООП — это понятие наследования классов, устанавливающее между двумя классами отношения "родитель-потомок".

*Наследование* — отношение самого высокого уровня и играет важную роль на стадии проектирования. Наследование — это определение класса и затем использование его для построения иерархии производных классов, причем каждый класс-потомок наследует от класса-предка интерфейс всех классов-предков в виде доступа к коду их методов и данным. При этом, возможно, переопределение или добавление как новых данных, так и методов.

Класс-предок — это класс, предоставляющий свои возможности и характеристики другим классам через механизм наследования. Класс, который использует характеристики другого класса посредством наследования, называется его классом-потомком.

Итак, наследование проявляется в том, что любой класс-потомок имеет доступ или, другими словами, наследует практически все ресурсы (методы, поля и свойства) родительского класса и всех предков до самого верхнего уровня иерархии.

Повторное использование — это использование в программе класса для создания экземпляров или в качестве базового для создания нового класса, наследующего часть или все характеристики родителя. Порождая классы от базовых, вы эффективно повторно используете код базового класса для собственных нужд. Повторное использование сокращает объем кода, который необходимо написать и оттестировать при реализации программы, что сокращает объемы труда.

Таким образом, наследование выполняет в ООП несколько важных функций:

• моделирует концептуальную структуру предметной области;

• экономит описания, позволяя использовать их многократно для задания разных классов;

• обеспечивает пошаговое программирование больших систем путем многократной конкретизации классов.

Обычно если объекты соответствуют конкретным сущностям реального мира, то классы являются абстракциями, выступающими в роли понятий. Между классами, как между понятиями, существует иерархическое отношение конкретизации, связывающее класс с классом-потомком. Это отношение реализуется в системах ООП механизмом наследования. Наследование — это способность одного класса использовать характеристики другого.

Наследование позволяет практически без ограничений последовательно строить и расширять классы, созданные вами или кем-то еще. Начиная с самых простых классов можно создавать производные классы по возрастающей сложности, которые не только легки при отладке, но и просты по внутренней структуре.

Одним из базовых понятий технологии ООП является полиморфизм. Термин "полиморфизм" имеет греческое происхождение и означает приблизительно "много форм" (poly — много, morphos — форма).

Полиморфизм — это средство для придания различных значений одному и тому же событию в зависимости от типа обрабатываемых данных, т. е. полиморфизм определяет различные формы реализации одноименного действия.

Целью полиморфизма применительно к объектно-ориентированному программированию является использование одного имени для задания общих для класса действий, причем каждый объект имеет возможность по-своему реализовать это действие своим собственным, подходящим для него кодом.

Полиморфизм является предпосылкой для расширяемости объектно-ориентированных программ, поскольку он предоставляет способ старым программам воспринимать новые типы данных, которые не были определены во время написания программы.

В более общей трактовке полиморфизм — это способность объектов, принадлежащих к разным типам, демонстрировать одинаковое поведение; способность объектов, принадлежащих к одному типу, демонстрировать разное поведение.

Практический смысл полиморфизма заключается в том, что он позволяет посылать общее сообщение о сборе данных любому классу, причем и родительский класс, и классы-потомки ответят на сообщение соответствующим образом, поскольку производные классы содержат дополнительную информацию. Программист может сделать регулярным процесс обработки несовместимых объектов различных типов при наличии у них такого полиморфного метода.

**4.2 Интегрированная среда программирования Delphi**

**Элементы интерфейса программы**

Delphi – интегрированная среда разработки (Integrated Development Environment – IDE). Delphi дает возможность создавать программы в стиле визуального конструирования формы, разместив на ней какие-либо визуальные элементы. Delphi имеет сложный интерфейс.

В Delphi имеются 10 окон, но после запуска программы на экране появляются пять:

главное окно — Delphi 7;

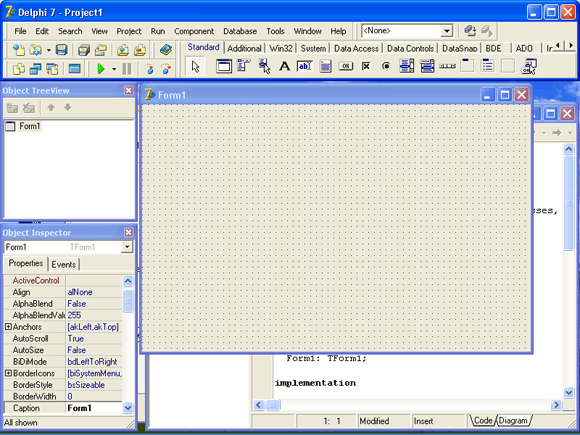
окно стартовой формы — Form 1;

окно редактора свойств объектов — Object Inspector;

окно просмотра списка объектов — Object TreeView;

окно редактора кода — Unitl.pas.

Окно редактора кода почти полностью закрыто окном стартовой формы.



Вид экрана после запуска Delphi

В главном окне реализуется основные функции управления проектом создаваемой программы.Главное управляющее окно системы Delphi обычно располагается в верхней части экрана. Оно содержит средства управления созданием программы и выглядит наиболее загруженным из всех окон

Это окно содержит:

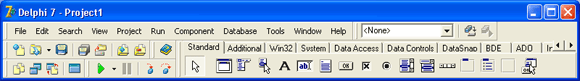
* строку заголовка;
* строку меню;
* панель инструментов;
* палитру компонентов.

*Строка заголовка* главного окна отображает имя открытого в данный момент проекта Project1. Под строкой заголовка окна системы рас­полагается строка меню, содержащая команды системы Delphi.

*Строка* *меню* содержит команды, необходимые для разработки и тестирования приложений, и используется так же, как любое стандартное меню Windows. Строка меню — это очень важный элемент управ­ления любой программы или системы. Кроме строки меню программы обычно содержат и другие элементы управления: командные кнопки, кнопки инструментов и прочие. Кнопками удобно пользоваться для исполне­ния операций, но не для всякой операции есть кнопка, а только для тех, которые встречаются особенно часто. Зато с помощью строки меню можно найти любую ко­манду, на которую система способна.

Под строкой меню располага­ются *панели* *инструментов* с кнопками. Во многих про­граммах имеется только одна такая панель, но в систе­ме Delphi их несколько. Панели инструментов предназначены для выполнения некоторых команд, реализуемых главным меню. Кнопки панелей инструментов обеспечивают доступ к наиболее часто встречающимся командам. Чтобы узнать, как называется та или иная кнопка, надо навести на нее указатель мыши и подо­ждать, пока рядом с ним появится всплывающая под­сказка. На этой панели есть, в частности, кнопка сохранения проекта на диске, кнопка открытия проекта, кнопка запуска программы на выполнение.

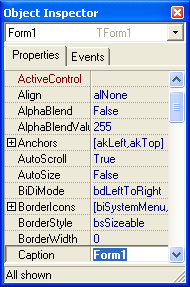
Одна из панелей инструментов, имеющихся в главном окне системы Delphi, заметно от­личается от остальных. Это *палитра* *компонентов*. Палитра компонентов устроена в виде набора пиктограмм. Палитра компонентов содержит множество вкладок. Каждая из них содержит свой набор компонентов. Общее число компонентов до­стигает нескольких сотен. Совокупность наборов составляет библиотеку визуальных компонентов (Visual Components Library - VCL). Имеется несколько категорий компонентов, каждая из которых расположена на своей вкладке. С помощью палитры компонентов создаются экземпляры компонентов (объекты) на форме.



Главное окно

Окно *инспектора**объектов* **(**Object Inspector) отображает свойства какого-либо компонента, активизированного щелчком мышью, или самой формы. Имя активизированного компонента находится под заголовком окна. Именно с помощью инспектораобъектовнастраивают объекты, используемые в программах. Это же окно используется для выбора и настройки со­бытий, на которые будут реагировать объекты нашей программы. С его помощью создаются или вы­бираются нужные процедуры обработки.

В терминологии визуального проектирования *объекты —*это диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.). *Свойства объекта —*это характеристики, определяющие вид, положение и поведение объекта. Например, свойства width и Height задают размер (ширину и высоту) формы, свойства тор и Left — положение формы на экране, свойство caption — текст заголовка.



Окно инспектора объектов

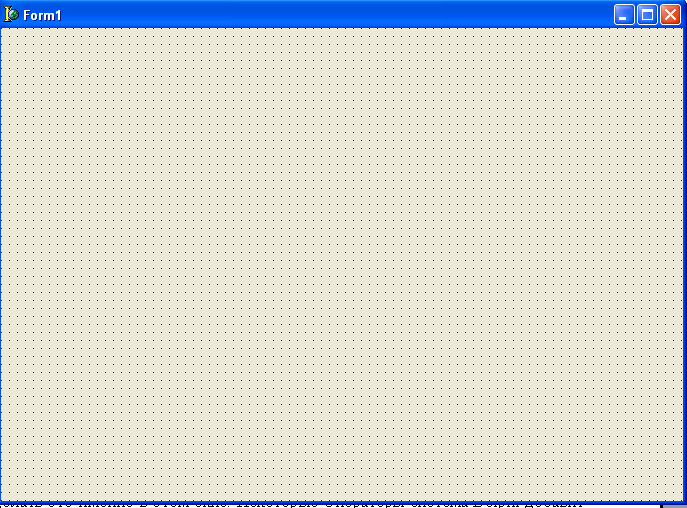
Каждый элемент управления, как и сама форма, обязательно имеет название. Название по умолчанию формирует сама система. Кроме того, практически все элементы управления также характеризуются текстовой информационной строкой - надписью, которая первоначально совпадает с названием элемента. Например, кнопка на форме получает название и надпись Button1, а поле-надпись - название и надпись Label1. Эти надписи можно изменять.

Инспектор - одно из важнейших окон при работе в дизайнере. С его помощью настраиваются всевозможные свойства объектов, используемых в программе, а также возможные реакции программы на различные действия пользователя с этими элементами управления. Многие составляющие части программы (прежде всего элементы управления) представляют собой объекты. Они характеризуются наборами свойств и методами, вызываемыми в различных ситуациях. У каждого элемента управления есть обязательно свойство Name, с помощью которого можно определить название этого объекта (на самом деле это название будет названием переменной в тексте программы, по которому к нему можно обращаться!). А текстовая надпись элемента задается с помощью свойства Caption. Некоторые свойства существуют практически у всех элементов управления, а некоторые - специфичны, так как предназначаются для определения конкретных особенностей работы определенного элемента управления.

В Инспекторе показывается список свойств, связанный с текущим элементом, выделенным на форме. Как только изменяется значение какого-нибудь свойства в окне, внесенные изменения сразу же будут отражены и в главном окне дизайнера.

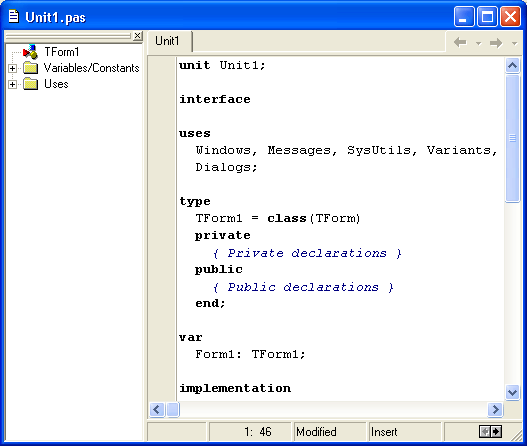
Окно *проектировщика формы* – главное место, где происходит сборка программы из компонентов, содержащихся в палитре компонентов. Сама форма – это уже готовая к выполнению программа. Вначале окно формы пустое, но, создавая программу, в указанное место формы добавляются объекты – экземпляры компонента выбранного типа. Сетка из точек в окне формы поможет разместить объекты ровно и аккуратно. При работе программы ее не ви­дно.

Форма — это заготовка окна будущей про­граммы. Каждая программа содержит хотя бы одно окно и, следовательно, одну форму. Поэтому при работе над программой окно формы мы видим на экране всегда. Заготовка первого окна называется Form1. Если в программе будет два окна, то заготовка второго будет называться Form2 и так далее. Возможна замена стандартного названия более подходящим для программы.



Окно формы

Последнее из открытых окон содержит *код программы*. Часть программы система Delphi формирует автоматически. Даже «пустая» программа для *Windows* собирается из нескольких тысяч операторов языка Pascal. Добавлять нужные операторы система Delphi начинает еще до того, как программист приступил к созданию своей про­граммы. Добавляются операто­ры именно в этом окне. Некоторые операторы система Delphi доба­вит сама, другие мы введем вруч­ную. Именно поэтому окно кода никогда не бывает пустым.



Окно редактора кода

## 4.3 Технология создания приложений Delphi

# Этапы разработки приложения на языке Delphi

При создании приложения необходимо:

* уяснить задачу, которую надо решить;
* разработать эскизы того, что должно появляться на экране компьютера;
* написать сценарий работы будущей программы;
* разработать (при необходимости) алгоритмы процедур, реализующих предусмотренные в сценарии действия;
* реализовать проект;
* выполнить тестирование и отладку;
* подготовить проект к распространению.

Процесс создания простейшего оконного приложения можно разбить на два этапа: этап конструирования формы и этап программирования. Рассмотрим эти этапы более детально.

На этапе конструирования создается интерфейс будущего приложения, который определяет способ взаимодействия пользователя и приложения: какие применяются окна, каким образом пользователь управляет приложением. Интерфейс создаётся путём размещения на форме компонентов.

Интерфейс приложения определяется компонентами, которые разработчик выбирает из палитры компонентов и помещает на форму. При проектировании интерфейса приложения действует принцип WYSIWYG (What You See Is What You Get) – что видите, то и получите. Так как компонентов очень много, то они в палитре распределены по страницам. Для того чтобы поместить компонент на форму, необходимо перейти на нужную страницу, щёлкнуть мышью по выбранному компоненту в палитре, а затем – в нужном месте формы.

Перечислим операции, которые выполняются на этапе конструирования формы.

1. Прежде всего, надо выбрать необходимые компоненты из Палитры Компонентов и разместить их на форме. Для того чтобы разместить компонент на форме, необходимо выполнить щелчок левой кнопкой мыши на компоненте, находящемся в Палитре Компонентов, а затем щелкнуть левой кнопкой мыши в то место формы, куда должен быть помещен компонент. При размещении компонента на форме Delphi автоматически вносит изменения в файл модуля и в файл описания формы. В файле модуля в описании класса формы добавляется строчка с описанием типа компонента.

2. Помещенный на форму компонент можно перемещать по форме при помощи мыши и изменять его размеры, используя маркеры, обрамляющие выделенный компонент.

3. Для того чтобы придать компоненту нужные свойства, следует использовать страницу Properties в окне Инспектора Объектов. Для помещённых на форму компонентов непосредственно на форме или в окне Инспектора объектов задают значения свойств. Сначала выбирают нужный компонент из списка в верхней части Инспектора объектов или, щёлкнув по нему на форме, а затем задают значения свойств. Следует помнить, что некоторые свойства нельзя изменить на этапе дизайна, они доступны только во время выполнения программы. Такие свойства можно изменять программным путём. Свойства же, перечисленные в инспекторе объектов, разрешено изменять как на этапе проектирования, так и программно во время работы приложения.

Если на форме выделено несколько компонентов, то в Инспекторе объектов доступны свойства, общие для всех этих компонентов. При этом сделанные изменения относятся ко всем выделенным компонентам.

В результате выполнения указанных операций будет сформирован внешний вид окна будущей программы.

Для того чтобы программа выполняла какую-то полезную работу, необходимо перейти ко второму этапу – этапу программирования. Рассмотрим основные особенности этого этапа.

Функциональность приложения определяется реакциями на возможные события. Для этого разрабатывают процедуры, которые вызываются при наступлении соответствующих событий.

Для создания процедуры обработки события (обработчика), необходимо выделить компонент и перейти в Инспекторе объектов на страницу событий (Events), на которой перечислены все возможные события компонента. Если выполнить двойной щелчок в области значения нужного события, то Delphi автоматически создаст в модуле формы заготовку процедуры-обработчика. На передний план переместится окно Редактора кода и курсор будет помещён в то место процедуры, где программист должен написать код. При этом в описание класса формы будет добавлен заголовок процедуры-обработчика. Название обработчика образуется добавлением к имени компонента названия события без приставки on. Например, TForm1.Button1Click. Если в Инспекторе объектов изменить имя компонента, то произойдёт автоматическое переименование процедуры во всех файлах проекта. Для удаления обработчика достаточно удалить код, написанный программистом. Тогда при сохранении или компиляции модуля обработчик будет удалён из всех файлов проекта. Не рекомендуется удалять обработчики вручную. Следует помнить, что при удалении компонента все его обработчики остаются в модуле формы.

Разрешено вместо создания нового обработчика использовать уже существующий. В этом случае надо щёлкнуть в области значения события и выбрать из списка подходящую процедуру. Таким образом, одну процедуру можно связать с несколькими событиями, в том числе для разных компонент. Такая процедура называется общим обработчиком и вызывается при возникновении любого связанного с ней события (например, одни и те же действия выполняются через меню и кнопкой).

Среди операторов, вставляемых программистом, могут быть и такие, которые изменяют значения свойств компонентов, расположенных на форме. Это означает, что свойства компонентов можно изменять не только при помощи Инспектора Объектов на этапе конструирования, но и в процессе выполнения программы. Изменение свойств компонентов в процессе выполнения программы называется динамическим изменением свойств.

Каждый компонент Delphi имеет одно так называемое событие по умолчанию. Как правило, это событие, для которого чаще всего составляются обработчики. Например, для многих компонентов Delphi таким событием является OnClick. Для того чтобы создать заготовку обработчика события по умолчанию, достаточно выполнить двойной щелчок мышью по компоненту.

Между содержимым окна формы и окна редактора кода существует неразрывная связь, которая строго контролируется Delphi. Например, размещение на форме компонента приводит к автоматическому изменению кода программы. Как уже было сказано выше, автоматически создаются также заготовки для обработчиков событий. Программист при этом может наполнять заготовки конкретным содержанием – вставлять операторы, добавлять описания собственных переменных, типов, констант и т.д. При этом программист должен помнить, что ему нельзя удалять из текста программы те строки, которые вставила туда среда Delphi.

**Структура проекта delphi**

Для системы Delphi каждая незавершенная программа – это проект. Проект включает в себя множество файлов.

Проект, состоит из следующих файлов:

* файла проекта Project1.dpr,
* файла параметров проекта Project1.dof,
* файла ресурсов проекта Project.res,
* файла настроек проекта Project1.cfg,
* файла описания формы Unit1.dfm,
* файла модуля формы Unit1.pas.

Файл модуля формы доступен для редактирования, именно он отображается в Редакторе Кода. Остальные файлы создаются Delphi автоматически. В процессе компиляции программы файлы преобразуются в исполняемый exe-файл, который, по умолчанию, создается в той же папке, в которой расположен файл проекта.

В проекте могут быть задействованы несколько форм, а также дополнительные модули и файлы ресурсов, при этом схема компиляции остается похожей.

Файл проекта не предназначен для редактирования пользователем и создается автоматически самой системой программирования Delphi. Для того чтобы увидеть содержимое файла проекта, необходимо выполнить команду Project Þ View Source. Содержимое файла проекта может быть, например, следующим:

program Project1;

uses

Forms,

Unit1 program Project1;

uses

Forms,

Unit1 in'Unit1.pas' {Form1};

{$R \*.res}

Begin

Application.Initialize;

Application.CreateForm(TForm1, Form1);

Application.Run;

End.

Файл проекта (главный модуль) начинается словом program, за которым следует имя программы, совпадающее с именем проекта. Имя проекта задается программистом в момент сохранения файла проекта, и оно определяет имя создаваемого средой Delphi исполняемого файла (файла с расширением .ехе). Далее за словом uses следуют имена используемых модулей: стандартного модуля Forms и модуля формы Unit1. Похожая на комментарий директива {$R \*.res} указывает компилятору, что нужно использовать файл ресурсов, который содержит описание ресурсов приложения, например, пиктограммы. Звездочка указывает, что имя файла ресурсов такое же, как и у файла проекта, но с расширением .res.

Исполняемая часть главного модуля находится между операторными скобками begin...end. Операторы исполняемой части обеспечивают инициализацию приложения и вывод на экран Стартового окна.

Помимо главного модуля каждая программа включает как минимум один модуль формы, который содержит описание стартовой формы приложения и поддерживающих ее работу процедур. В Delphi каждой форме соответствует свой модуль. Для переключения между формой и окном редактора кода, содержащего соответствующий модуль, следует выполнить команду главного меню https://konspekta.net/lektsiiorgimg/baza14/70297635406.files/image006.jpg View ÞToggle Form/Unit, либо нажать функциональную клавишу F12, либо на панели инструментов View щелкнуть мышью по кнопке

Модули – это программные единицы, служащие для размещения фрагментов программ. При помощи содержащихся в них текстов программ (программных кодов) и реализуется решаемая пользователем задача.

Модули имеют стандартную конструкцию (последовательность и перечень разделов), предусмотренную языком программирования Object Pascal. Приведем структуру модуля общем виде:

unit < имя модуля >;

interface

 implementation

 initialization

 finalization

 end.

В качестве примера приведем содержимое модуля в том виде, в котором он находит–сразу после загрузки среды Delphi:

 unit Unit1;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs;

type

TForm1 = class(TForm)

procedure Label1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

implementation

{$R \*.dfm}

end.

Начинается модуль словом unit, за которым следует имя модуля. Именно это имя 1 упоминается в списке используемых модулей в операторе uses главного модуля приложения.

Модуль может состоять из четырех разделов: интерфейса, реализации, инициализации завершающей части.

Раздел интерфейса (начинается словом interface) сообщает компилятору, какие данные, располагающиеся в модуле, являются доступными для других модулей программы. В этом разделе перечислены (после, слова uses) стандартные модули, используемые данным модулем, а также находится сформированное Delphi описание типа формы, которое следует за словом type.

Раздел реализации начинается словом implementation и содержит объявления локальных переменных, процедур и функций, поддерживающих работу формы. В начале раздела реализации располагается директива {SR \*.dfm}, указывающая компилятору, что в раздел реализации надо вставить команды установки значений свойств формы, которые находятся в файле с расширением .dfm , имя которого совпадает с именем модуля. Файл в формате dfm генерируется Delphi на основе внешнего вида формы.

За директивой {$R .dfm} располагаются описания процедур обработки событий формы. Сюда же программист может поместить описание своих процедур и функций, которые могут вызываться из процедур обработки событий.

Инициирующая и завершающая части являются необязательными.

Инициирующая часть начинается словом initialization либо заключается в операторные скобки begin... end. Операторы из этой части выполняются до передачи управления основной программе и обычно используются для подготовки ее работы.

Завершающая часть начинается словом finalization и содержит операторы, выполняющиеся в момент окончания программы.

В приведенном выше примере модуля инициирующая и завершающая части отсутствуют.

В отличие от файла проекта, создаваемого автоматически Delphi, модуль может изменяться (редактироваться) программистом. При создании пользователем новой формы, автоматически будет создаваться и новый модуль. Программа может содержать до нескольких десятков форм. Текст модуля при этом будет доступен и пользователю, и самой среде Delphi, которая автоматически будет вставлять в текст модуля описание любого добавленного к форме компонента, а также создавать заготовки (строки кода) для обработчиков событий. Программист при этом может добавлять свои методы в ранее объявленные классы, наполнять обработчики событий конкретным содержанием, вставлять собственные переменные, типы, константы и т.д. Но, как уже было сказано ранее, программисту нельзя удалять строки, вставленные в текст модуля интегрированной средой Delphi.

При компиляции программы Delphi создает файлы с расширениями .dcu для каждого модуля.

Таким образом, pas-файл содержит программный код модуля, который был сформирован в окне редактора кода совместными усилиями программиста и среды Delphi, в файле с расширением .dfm хранится описание содержимого окна формы, а в dcu-файле находится результат преобразования текста из обоих файлов в машинные инструкции. Компоновщик, входящий в интегрированную среду Delphi, преобразует dcu-файлы в единый загрузочный (выполнимый) ехе-файл. Выполнимый файл позволяет запускать программу как автономное приложение.

Чтобы не смешивать файлы разных проектов, желательно каждый новый проект сохранять в отдельную папку.

**4.4 Визуальные компоненты Delphi**

Вместе со средой Delphi поставляется более сотни различных компонентов, а в Интернете можно найти тысячи. Они отличаются по своему внешнему виду, функциям, способам реализации, требованиям к аппаратной части и т.п., но, тем не менее, все они построены на едином фундаменте, единой объектной основе.

Несмотря на различие форм и функций компонентов, их можно разбить на группы и классифицировать. Все компоненты можно разделить на визуальные и невизуальные. Визуальные, в свою очередь, делятся на оконные элементы управления и графические элементы управления.

Визуальные компоненты являются элементами управления, которые добавляются на форму: это кнопки, поля ввода, картинки, панели и т.д. Те из них, что предназначены для ввода данных, могут принимать фокус ввода, обеспечивая управление с клавиатуры.

Невизуальные компоненты служат в основном для обеспечения более удобного доступа к определенным функциям системы. При добавлении их на форму на ней появляется иконка с изображением компонента, невидимая во время исполнения программы. В этом и заключается их «невизуальность». Несмотря на это, некоторые из них служат для обеспечения визуальных по природе эффектов. Например, к таким относятся компоненты меню и диалоги.

Визуальные компоненты составляют интерфейс программы. Визуальные компоненты имеют много общих свойств и событий, связанных, прежде всего, с их визуальным отображением на форме.

Часть свойств отвечает за положение на форме: отступ слева, отступ сверху, высота, ширина, выравнивание.

Некоторые определяют внешний вид: цвет, шрифт, заголовок. Некоторые задают поведение компонента во время исполнения программы: доступен, виден, курсор, подсказка.

Механизм событий позволяет связать действия пользователя с необходимыми действиями программы. События определяют всю структуру программы, поскольку фактически весь код, то есть, собственно работа программиста, пишется в обработчиках событий.

В Windows действия пользователя обычно сводятся к нажатиям мыши в определенных местах экрана, нажатию клавиш клавиатуры, переключению между окнами и компонентами. В соответствии с этим существуют и события, реагирующие на нажатие мыши и клавиатуры, переключение фокуса, активацию окон.

Кроме того, для каждого компонента могут существовать характерные для него события, на которые может потребоваться реакция и обработка.

**1. Стандартные компоненты**

На первой странице Палитры Компонент размещены 14 объектов определенно важных для использования. Мало кто обойдется длительное время без кнопок, списков, окон ввода и т.д. Все эти объекты такая же часть Windows, как мышь или окно.

Набор и порядок компонент на каждой странице являются конфигурируемыми. Так, Вы можете добавить к имеющимся компонентам новые, изменить их количество и порядок. Это можно сделать, вызвав всплывающее меню (нажать правую кнопку мыши, когда указатель над Палитрой).

hello_html_m774879f5.png

Стандартные компоненты Delphi перечислены ниже с некоторыми комментариями по их применению. При изучении данных компонент было бы полезно иметь под рукой компьютер с тем, чтобы посмотреть, как они работают и как ими манипулировать.

hello_html_6f7e79f9.pngКурсор - не компонент, просто пиктограмма для быстрой отмены выбора какого-либо объекта.

hello_html_m2035f59f.pngTMainMenu позволяет Вам поместить главное меню в программу. При помещении TMainMenu на форму это выглядит, как просто иконка. Иконки данного типа называют "невидимыми компонентом", поскольку они невидимы во время выполнения программы. Создание меню включает три шага: (1) помещение TMainMenu на форму, (2) вызов Дизайнера Меню через свойство Items в Инспекторе Объектов, (3) определение пунктов меню в Дизайнере Меню.

hello_html_29ca7e56.pngTPopupMenu позволяет создавать всплывающие меню. Этот тип меню появляется по щелчку правой кнопки мыши на объекте, к которому привязано данное меню. У всех видимых объектов имеется свойство PopupMenu, где и указывается нужное меню. Создается PopupMenu аналогично главному меню.

hello_html_m30fafb6f.pngTLabel служит для отображения текста на экране. Вы можете изменить шрифт и цвет метки, если дважды щелкнете на свойство Font в Инспекторе Объектов. Вы увидите, что это легко сделать и во время выполнения программы, написав всего одну строчку кода.

hello_html_m7c9edad8.pngTEdit - стандартный управляющий элемент Windows для ввода. Он может быть использован для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы.

hello_html_m46df0645.pngTMemo - иная форма TEdit. Подразумевает работу с большими текстами. TMemo может переносить слова, сохранять в ClipBoard фрагменты текста и восстанавливать их, и другие основные функции редактора. TMemo имеет ограничения на объем текста в 32Кб, это составляет 10-20 страниц. (Есть VBX и “родные” компоненты Delphi, где этот предел снят).

hello_html_3900d48a.pngTButton позволяет выполнить какие-либо действия при нажатии кнопки во время выполнения программы. В Delphi все делается очень просто. Поместив TButton на форму, Вы по двойному щелчку можете создать заготовку обработчика события нажатия кнопки. Далеенужно заполнить заготовку кодом:

procedure TForm1.Button1Click(Sender: TObject);

begin

MessageDlg('Are you there?',mtConfirmation,mbYesNoCancel,0);

end;

hello_html_m214685f7.pngTCheckBox отображает строку текста с маленьким окошком рядом. В окошке можно поставить отметку, которая означает, что что-то выбрано. Например, если посмотреть окно диалога настроек компилятора (пункт меню Options | Project, страница Compiler), то можно увидеть, что оно состоит преимущественно из CheckBox’ов.

hello_html_59739e64.pngTRadioButton позволяет выбрать только одну опцию из нескольких. Если Вы опять откроете диалог Options | Project и выберете страницу Linker Options, то Вы можете видеть, что секции Map file и Link buffer file состоят из наборов RadioButton.

hello_html_m47042e53.pngTListBox нужен для показа прокручиваемого списка. Классический пример ListBox’а в среде Windows - выбор файла из списка в пункте меню File | Open многих приложений. Названия файлов или директорий и находятся в ListBox’е.

hello_html_m47042e53.pngTComboBox во многом напоминает ListBox, за исключением того, что позволяет водить информацию в маленьком поле ввода сверху ListBox. Есть несколько типов ComboBox, но наиболее популярен спадающий вниз (drop-down combo box), который можно видеть внизу окна диалога выбора файла.

hello_html_20920a4b.pngTScrollbar - полоса прокрутки, появляется автоматически в объектах редактирования, ListBox’ах при необходимости прокрутки текста для просмотра.

hello_html_493b7868.pngTGroupBox используется для визуальных целей и для указания Windows, каков порядок перемещения по компонентам на форме (при нажатии клавиши TAB).

hello_html_39d8b438.pngTRadioGroup используется аналогично TGroupBox, для группировки объектов TRadioButton.

hello_html_6132ba9d.pngTPanel - управляющий элемент, похожий на TGroupBox, используется в декоративных целях. Чтобы использовать TPanel, просто поместите его на форму и затем положите другие компоненты на него. Теперь при перемещении TPanel будут передвигаться и эти компоненты. TPanel используется также для создания линейки инструментов и окна статуса.

Это полный список объектов на первой странице Палитры Компонент. Если Вам нужна дополнительная информация, то выберите на Палитре объект и нажмите клавишу F1 - появится Справочник с полным описанием данного объекта.

**2. Страница Additional**

На странице Standard представлены управляющие элементы, на странице Additional размещены объекты, позволяющие создать более красивый пользовательский интерфейс программы.

hello_html_m3bb41247.png

Список компонент:

hello_html_3dadb84c.pngTBitBtn - кнопка вроде TButton, однако на ней можно разместить картинку (glyph). TBitBtn имеет несколько предопределенных типов (bkClose, bkOK и др), при выборе которых кнопка принимает соответствующий вид. Кроме того, нажатие кнопки на модальном окне (Form2.ShowModal) приводит к закрытию окна с соответствующим модальным результатом (Form2.ModalResult).

hello_html_m73de1c33.pngTSpeedButton - кнопка для создания панели быстрого доступа к командам (SpeedBar). Пример - SpeedBar слева от Палитры Компонент в среде Delphi. Обычно на данную кнопку помещается только картинка (glyph).

TTabSet - горизонтальные закладки. Обычно используется вместе с TNoteBook для создания многостраничных окон. Название страниц можно задать в свойстве Tabs. Но проще это сделать в программе при создании формы (OnCreate) :

TabSet1.Tabs := Notebook1.Pages;

А для того, чтобы при выборе закладки страницы перелистывались нужно в обработчике события OnClick для TTabSet написать:

Notebook1.PageIndex := TabSet1.TabIndex;

TNoteBook - используется для создания многостраничного диалога, на каждой странице располагается свой набор объектов. Используется совместно с TTabSet.

hello_html_m45036e32.pngTTabbedNotebook - многостраничный диалог со встроенными закладками, в данном случае - закладки сверху.

hello_html_m1df9347e.pngTMaskEdit - аналог TEdit, но с возможностью форматированного ввода. Формат определяется в свойстве EditMask. В редакторе свойств для EditMask есть заготовки некоторых форматов: даты, валюты и т.п. Спец. символы для маски можно посмотреть в Справочнике.

hello_html_1f18469f.pngTOutline - используется для представления иерархических отношений связанных данных. Например - дерево директорий.

hello_html_m2b055324.pngTStringGrid - служит для представления текстовых данных в виде таблицы. Доступ к каждому элементу таблицы происходит через свойство Cell.

hello_html_235b0126.pngTDrawGrid - служит для представления данных любого типа в виде таблицы. Доступ к каждому элементу таблицы происходит через свойство CellRect.

hello_html_2d0c23bc.pngTImage - отображает графическое изображение на форме. Воспринимает форматы BMP, ICO, WMF. Если картинку подключить во время дизайна программы, то она прикомпилируется к EXE файлу.

hello_html_ae98d60.pngTShape - служит для отображения простейших графических объектов на форме: окружность, квадрат и т.п.

hello_html_m1b843809.pngTBevel - элемент для рельефного оформления интерфейса.

hello_html_4414ec65.pngTHeader - элемент оформления для создания заголовков с изменяемыми размерами для таблиц.

hello_html_m56ddb2a6.pngTScrollBox - позволяет создать на форме прокручиваемую область с размерами большими, нежели экран. На этой области можно разместить свои объекты.

**3. Страница System**

hello_html_m4716fbe6.png

Страница представляет набор компонент для доступа к некоторым системным сервисам типа таймер, DDE, OLE и т.п.

hello_html_2ece5df.pngTTimer - таймер, событие OnTimer периодически вызывается через промежуток времени, указанный в свойстве Interval. Период времени может составлять от 1 до 65535 мс.

hello_html_m10141e4f.pngTPaintBox - место для рисования. В обработчики событий, связанных с мышкой передаются относительные координаты мышки в TPaintBox, а не абсолютные в форме.

hello_html_11c199f2.pngTFileListBox - специализированный ListBox, в котором отображаются файлы из указанной директории (св-во Directory). На названия файлов можно наложить маску, для этого служит св-во Mask. Кроме того, в св-ве FileEdit можно указать объект TEdit для редактирования маски.

hello_html_c27b4f4.pngTDirectoryListBox - специализированный ListBox, в котором отображается структура директорий текущего диска. В св-ве FileList можно указать TFileListBox, который будет автоматически отслеживать переход в другую директорию.

hello_html_m7cef45d1.pngTDriveComboBox - специализированный ComboBox для выбора текущего диска. Имеет свойство DirList, в котором можно указать TDirectoryListBox, который будет отслеживать переход на другой диск.

hello_html_m2a093d8c.pngTFilterComboBox - специализированный ComboBox для выбора маски имени файлов. Список масок определяется в свойстве Filter. В свойстве FileList указывается TFileListBox, на который устанавливается маска.

!!!! С помощью последних четырех компонент (TFileListBox, TDirectoryListBox, TDriveComboBox, TFilterComboBox) можно построить свой собственный диалог выбора файла, причем для этого не потребуется написать ни одной строчки кода.

hello_html_m3d8c7080.pngTMediaPlayer - служит для управления мултимедйными устройствами (типа CD-ROM, MIDI и т.п.). Выполнен в виде панели управления с кнопками Play, Stop, Record и др. Для воспроизведения может понадобиться как соответствующее оборудование, так и программное обеспечение. Подключение устройств и установка ПО производится в среде Windows. hello_html_m5dab41f6.pnghello_html_m21e08485.pngTOLEContainer - контейнер, содержащий OLE объекты.

## 4.5 Работа с базами данных

Хотя Delphi не имеет своего формата таблиц БД, она, тем не менее, обеспечивает мощную поддержку большого количества различных СУБД Средства Delphi, предназначенные для работы с БД, можно разделить на два вида:

• инструментальные средства;

• компоненты.

К инструментальным средствам относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений. Компоненты предназначены для создания приложений, осуществляющих операции с БД.

**Инструментальные средства**

Для операций с БД система Delphi предлагает набор инструментальных средств, перечисленных ниже.

• Borland Database Engine (BDE) — процессор баз данных, который представляет собой набор динамических библиотек и драйверов, предназначенных для организации доступа к БД из Delphi-приложений. BDE является центральным звеном при организации доступа к данным.

• BDE Administrator — утилита для настройки различных параметров BDE.

• Database Desktop — программа создания и редактирования таблиц, SQLзапросов и запросов QBE.

• SQL Explorer — Проводник БД, позволяющий просматривать и редактировать БД и словари данных.

• SQL Builder — программа визуального конструирования SQL-запросов.

• Data Pump — программа для переноса данных между БД.

**Компоненты**

Кроме компонентов, Delphi также предоставляет разработчику специальные объекты, например, объекты типа TFieid. Как и другие управляющие элементы Delphi, связанные с БД, компоненты делятся на визуальные и невизуальные.

Невизуальные компоненты предназначены для организации доступа к данным, содержащимся в таблицах. Они представляют собой промежуточное звено между данными таблиц БД и визуальными компонентами. Визуальные компоненты используются для создания интерфейсной части приложения. С их помощью пользователь может выполнять такие операции с таблицами БД, как просмотр или редактирование данных. Визуальные компоненты также называют элементами, чувствительными к данным. Основная часть компонентов, используемых для работы с БД, находится на страницах **Data Access, Data Controls, BDE** и **QReport** Палитры компонентов.

На странице **Data Access** находятся невизуальные компоненты, предназначенные для организации доступа к данным:

• DataSource - источник данных;

• ClietDataset - клиентский набор данных;

• DataSetProvider - провайдер набора данных.

На странице **Data Controls** находятся визуальные компоненты, предназначенные для управления данными:

• DBGrid — сетка (таблица);

• DBNavigator — навигационный интерфейс;

•DBText — надпись;

• DBEdit — однострочный редактор;

• овмето — многострочный редактор;

• DBimage — графический образ;

• DBListBox — простой список;

• DBComboBox — комбинированный список;

• овсьесквох — независимый переключатель;

• DBRadioGroup — группа зависимых переключателей;

• DBLookupListBox — простой список, формируемый по полю другого набора данных;

• DBLookupComboBox — комбинированный список, формируемый по полю другого набора данных;

• DBRichEdit — полнофункциональный тестовый редактор;

• DBCtrlGrid — модифицированная сетка;

• DBChart — диаграмма.

На странице **BDE** (рис. 12) находятся компоненты, предназначенные для управления данными с использованием BDE:

• Table — набор данных, основанный на таблице БД;

• Query — набор данных, основанный на SQL-запросе;

• storedProc — вызов хранимой процедуры сервера;

• DataBase — соединение с БД;

• Session — текущий сеанс работы с БД;

• BatchMove — выполнение операций над группой записей;

• updateSQL — модификация набора данных, основанного на SQL-запросе;

• NestedTabie — **вложенная таблица;**

**•** BDECiientDataSet — **клиентский набор данных.**

На странице QReport находятся компоненты (в основном визуальные), предназначенные для построения отчетов:

• QuickRep — отчет;

• QRSubDetaii — полоса отчета для таблиц, связанных отношением "главный-подчиненный";

**•**  QRStringsBand — строковая полоса отчета;

• QRBand — полоса отчета;

**•**  QRChiidBand — дочерняя полоса отчета;

• QRGroup — группа;

• QRLabel — надпись;

• QRDBText — текстовое поле набора данных;

• QRExpr — выражение;

• QRSysData — системная информация;

• QRMemo — многострочный текст;

• QRExprMemo — многострочное выражение;

• QRRichText — форматированный текст;

• QRDBRichText — форматированный текст поля набора данных;

• QRShape — геометрическая фигура;

• QRimage — графический образ;

• QRDBimage — графический образ поля набора данных;

• QRCompositeReport — составной отчет;

• QRPreview — окно просмотра отчета;

• QRTextFiiter — текстовый фильтр;

• QRCSvFiiter — CSV-фильтр;

О QRHTMLFiiter — HTNL-фильтр;

• QRChart — диаграмма.

Названия многих компонентов, предназначенных для работы с данными, содержат префиксы, например, DB ИЛИ QR. Префикс DB означает, что визуальный компонент связан с данными и используется для построения интерфейсной части приложения. Такие компоненты размешаются на форме и предназначены для управления данными со стороны пользователя. Префикс QR означает, что компонент используется для построения отчетов. Эти компоненты размещаются на компоненте QuickRep отчета и его элементах, например, на полосе QRBand, и служат для оформления внешнего вида отчета.

## 4.6 Работа с элементами управления данными

Компоненты доступа к данным являются невизуальными. В этом разделе рассматриваются основные компоненты доступа к данным.

**Наборы данных**

Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. В терминах системы Delphi *набор данных* представляет собой совокупность записей, взятых из одной или нескольких таблиц БД. Записи, входящие в набор данных, отбираются по определенным правилам, при этом в частных случаях набор данных может включать в себя все записи из связанной с ним таблицы или не содержать ни одной записи. Набор данных является *логической таблицей,* с которой можно работать при выполнении приложения. Взаимодействие таблицы и набора данных напоминает взаимодействие физического файла и файловой переменной.

В Delphi для работы с наборами данных служат такие компоненты, как Table И Query. Они имеют схожие с базовыми классами характеристики и поведение, но каждый из них имеет и свои особенности.

Расположение БД, с таблицами которой выполняются операции, указывает свойство DatabaseName типа string. Значением свойства является имя каталога, в котором находится БД (файлы ее таблиц), или псевдоним, ссылающийся на этот каталог. Если для БД определен псевдоним (alias), то его можно выбрать через Инспектор объектов в раскрывающемся списке.

Для компонента Table использование свойства DatabaseName является единственной возможностью задать местонахождение таблиц БД. Для компонента Query дополнительно можно указать в запросе SQL путь доступа к каждой таблице.

При задании расположения БД программным способом набор данных предварительно необходимо закрыть, установив его свойству Active значение False. В противном случае генерируется исключительная ситуация.

**Состояния наборов данных**

Наборы данных могут находиться в *открытом* или *закрытом* состояниях, на что указывает свойство Active типа Boolean. Набор данных может быть открыт на этапе разработки приложения. Если при этом к набору данных через источник данных DataSource подключены визуальные компоненты, например, DBGrid или DBEdit, то они отображают соответствующие данные таблицы БД.

На этапе проектирования приложения визуальные компоненты отображают данные записей набора данных, но перемещение по набору данных и редактирование записей невозможны. Исключение составляет возможность перемещения текущего указателя с помощью полос прокрутки компонента DBGrid.

**Режимы наборов данных**

Наборы данных могут находиться в различных режимах. Текущий режим набора данных определяется свойством state типа TDataSetstate, которое доступно для чтения во время выполнения приложения. Для перевода набора данных в требуемый режим используются специальные методы. Они могут вызываться явно (указанием имени метода) или косвенно (путем управления соответствующими визуальными компонентами, например, навигатором DBNavigator ИЛИ сеткой DBGrid).

Выделим следующие основные режимы набора данных.

• dsinactive — неактивность; набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после его закрытия, когда свойству Active установлено значение False.

• dsBrowse — осуществляется навигация по записям набора данных и просмотр данных.

• dsEdit — редактирование текущей записи. В этот режим набор данных переходит из режима dsBrowse при вызове метода Edit.

• dsinsert — вставка новой записи. В данный режим набор данных переходит ИЗ режима dsBrowse при вызове методов Insert, InsertRecord, Append ИЛИ AppendRecord.

• dsCalcFields — расчет вычисляемых полей. Используется обработчик события OnCalcFields.

• dsSetKey — поиск записи, удовлетворяющей заданному критерию. В этот режим набор данных переходит из режима dsBrowse при вызове методов SetKey, SetRangeXXX, FindKey, GotoKey, FindNearest ИЛИ GotoNearest. Возможен только для компонента ттаЫе, так как для ком понента Query отбор записей осуществляется средствами языка SQL.

• dsFiiter — фильтрация записей. В этот режим набор данных автоматически переходит из режима dsBrowse каждый раз, когда выполняется обработчик события onFiiterRecord. В режиме блокируются все попытки изменения записей. После завершения работы обработчика события OnFiiterRecord набор данных автоматически переводится в режим dsBrowse.

**Доступ к полям**

Каждое поле набора данных представляет собой отдельный столбец, для работы с которым в Delphi служат объект Field типа TFieid и объекты производных ОТ него ТИПОВ, например, TIntegerField, TFloatField или TStringFieid. Для доступа к этим объектам и, соответственно, к полям записей набор данных имеет соответствующие методы и свойства, доступные при выполнении приложения.

Для доступа к полям удобно использовать метод FieidByName. Функция

FindField(const FieldName: String): TFieid

Возвращает для набора данных поле, имя которого указывает параметр FieldName. Если заданное параметром FieldName поле не найдено, то генерируется исключительная ситуация.

Замечание. Имя поля, определяемое параметром FieldName, является именем физического поля таблицы БД, заданным при создании таблицы, а не именем (свойством Name) объекта Field , которое создано для этого поля. Для набора данных Query имя FieldName физического поля можно переопределить в тексте SQL-запроса.

Метод FieidByName часто используется для доступа к значению поля текущей записи совместно с такими свойствами объекта Field, как AsString, Asinteger, AsFioat или AsBooiean, которые соответственно позволяют обращаться к значению поля как к строковому, целочисленному, вещественному или логическому значению.

## 4.7 Программирование динамически подключаемых библиотек

Динамически подключаемая библиотека (DLL) - это один или несколько логически законченных фрагментов кода, сохраненных в файле с расширением.dll. Этот код может быть запущен на выполнение в процессе функционирования какой-либо другой программы (такие приложения называются вызывающими по отношению к библиотеке), но сама DLL не является запускаемым файлом.

Существует два типа динамических библиотек - исполняемые и библиотеки ресурсов.

Процедуры и функции, содержащиеся в динамической библиотеке, можно разделить на два типа: те, которые могут быть вызваны из других приложений.

**Преимущества использования DLL**

Во-первых, это повторное использование кода.

Во-вторых, использование DLL предоставляет возможность использования один загруженного в оперативную память кода несколькими приложениями. К примеру, если вы разрабатываете программное обеспечение для большого предприятия, то вполне возможно, что в различных созданных вами приложениях будут использоваться одни и те же функции, процедуры, формы и другие ресурсы. Естественно, что при выделении общих для нескольких приложений данных в DLL может привести к экономии как дискового пространства, так и оперативной памяти, иногда очень даже существенному.

В-третьих, возможность распространения ваших приложений в других странах. Соответственно, встает вопрос быстрого перевода интерфейса вашей программы на другие языки. Одним из путей может являться создание ресурсов интерфейсов внутри DLL. К примеру, можно создать одно приложение, которое в зависимости от версии динамической библиотеки будет выводить сообщения на различных языках.

**Основы разработки DLL**

Разработка динамических библиотек не представляет собой некий сверхсложный процесс, доступный лишь избранным. Как и любой другой модуль, модуль динамической библиотеки имеет фиксированный формат.

library MyFirstDLL;

uses

SysUtils,

Classes,

Forms,

Windows;

procedure HelloWorld(AForm : TForm);

begin

MessageBox(AForm.Handle, Hello world!',

DLL Message Box', MB\_OK or MB\_ICONEXCLAMATION);

end;

exports

HelloWorld;

begin

end.

Первое, на что следует обратить внимание, это ключевое слово library, находящееся вверху страницы. Library определяет этот модуль как модуль библиотеки DLL. Далее идет название библиотеки. В нашем примере мы имеем дело с динамической библиотекой, содержащей единственную процедуру: HelloWorld. Данная процедура по структуре ничем не отличается от тех, которые вы помещаете в модули своих приложений. Ключевое слово exports сигнализирует компилятору о том, что перечисленные ниже функции и/или процедуры должны быть доступны из вызывающих приложений (т.е. они как бы "экспортируются" из библиотеки). Подробнее о механизме экспорта мы поговорим чуть позже.

И, наконец, в конце модуля можно увидеть ключевые слова begin и end. Внутри данного блока вы можете поместить код, который должен выполняться в процессе загрузки библиотеки. Достаточно часто этот блок остается пустым.

Все процедуры и функции, помещаемые в DLL, могут быть разделены на две группы: экспортируемые (вызываемые из других приложений) и локальные. Естественно, внутри библиотеки также могут быть описаны классы, которые в свою очередь содержат методы, но в рамках данной статьи я не буду на этом останавливаться.

Описание и реализация процедур и функций, вызываемых в пределах текущей DLL, ничем не отличаются от их аналогов в обычных проектах-приложениях. Их специфика заключается лишь в том, что вызывающая программа не будет иметь к ним доступа. Она просто не будет ничего знать об их существования, так же, как одни классы ничего не знают о тех методах, которые описаны в секции private других классов.

**Экспорт функций из DLL**

Для экспорта процедур и функций из DLL, необходимо использовать ключевое слово export. Поскольку процедура HelloWorld определена как экспортируемая, то она может быть вызвана на выполнение из других библиотек или приложений. Существуют следующие способы экспорта процедур и функций: экспорт по имени и экспорт по порядковому номеру.

**Использование DLLProc**

DLLProc - специальная процедура, вызываемая в определенные моменты функционирования DLL. Динамическая библиотека получает сообщения от Windows в моменты своей загрузки и выгрузки из оперативной памяти, а также в тех случаях, когда какой-нибудь очередной процесс, использующий функции и/или ресурсы, хранящиеся в библиотеке, загружается в память. Такая ситуация возможно в том случае, когда библиотека необходима для функционирования нескольких приложений. А для того, чтобы вы имели возможность указывать, что именно должно происходить в такие моменты, необходимо описать специальную процедуру, которая и будет ответственна за такие действия. К примеру, она может выглядеть следующим образом:

procedure MyFirstDLLProc(Reason: Integer);

begin

if Reason = DLL\_PROCESS\_DETACH then

{DLL is unloading. Cleanup code here.}

end;

Однако системе совершенно не очевидно, что именно процедура MyFirstDllProc ответственна за обработку рассмотренных выше ситуаций. Поэтому вы должны поставить в соответствие адрес нашей процедуры глобальной переменной DLLProc. Это необходимо сделать в блоке begin...end примерно так:

begin

DLLProc := @MyDLLProc;

{ Что-нибудь еще, что должно выполняться в

процессе инициализации библиотеки }

end.

Ниже представлен код, демонстрирующий один из возможных вариантов применения DLLProc.

library MyFirstDLL;

uses

SysUtils,

Classes,

Forms,

Windows;

var

SomeBuffer : Pointer;

procedure MyFirstDLLProc(Reason: Integer);

begin

if Reason = DLL\_PROCESS\_DETACH then

{DLL is выгружается из памяти.

Освобождаем память, выделенную под буфер.}

FreeMem(SomeBuffer);

end;

procedure HelloWorld(AForm : TForm);

begin

MessageBox(AForm.Handle, Hello world!',

DLL Message Box', MB\_OK or MB\_ICONEXCLAMATION);

end;

{Какой-нибудь код, в котором используется SomeBuffer.}

exports

HelloWorld;

begin

{Ставим в соответствие переменной

DLLProc адрес нашей процедуры.}

DLLProc := @MyFirstDLLProc;

SomeBuffer := AllocMem(1024);

end.

Как можно увидеть, в качестве признака того или иного события, в результате которого вызывается процедура MyFirstDll, является значение переменной Reason. Ниже приведены возможные значения этой переменной.

DLL\_PROCESS\_DETACH - библиотека выгружается из памяти; используется один раз;

DLL\_THREAD\_ATTACH - в оперативную память загружается новый процесс, использующий ресурсы и/или код из данной библиотеки;

DLL\_THREAD\_DETACH - один из процессов, использующих библиотеку, "выгружается" из памяти.

**Загрузка DLL**

Прежде чем начать использование какой-либо процедуры или функции, находящейся в динамической библиотеке, вам необходимо загрузить DLL в оперативную память. Загрузка библиотеки может быть осуществлена одним из двух способов: статическая загрузка и динамическая загрузка. Оба метода имеют как преимущества, так и недостатки.

Статическая загрузка означает, что динамическая библиотека загружается автоматически при запуске на выполнение использующего ее приложения. Для того чтобы использовать такой способ загрузки, вам необходимо воспользоваться ключевым словом external при описании экспортируемой из динамической библиотеки функции или процедуры. DLL автоматически загружается при старте программы, и Вы сможете использовать любые экспортируемые из нее подпрограммы точно так же, как если бы они были описаны внутри модулей приложения. Это наиболее легкий способ использования кода, помещенного в DLL. Недостаток метода заключается в том, что если файл библиотеки, на который имеется ссылка в приложении, отсутствует, программа откажется загружаться.

Смысл динамического метода заключается в том, что вы загружаете библиотеку не при старте приложения, а в тот момент, когда вам это действительно необходимо. Выгрузка библиотеки из памяти в данном случае также осуществляется под вашим контролем.

**Вызов процедур и функций, загруженных из DLL.**

Способ вызова процедур и функций зависит от того, каким образом вы загрузили динамическую библиотеку, в которой эти подпрограммы находятся.

Вызов функций и процедур из статически загруженных DLL достаточно прост. Для импорта функции или процедуры, содержащейся в DLL, необходимо использовать модификатор external в их объявлении. К примеру, для рассмотренной нами выше процедуры HelloWorld в вызывающем приложении должна быть помещена следующая строка:

procedure SayHello(AForm : TForm); external myfirstdll.dll';

Ключевое слово external сообщает компилятору, что данная процедура может быть найдена в динамической библиотеке (в нашем случае - myfirstdll.dll). Далее вызов этой процедуры выглядит следующим образом:

...

HelloWorld(self);

...

Импорт из DLL может проводиться по имени процедуры (функции), порядковому номеру или с присвоением другого имени.

## 4.8 Создание пользовательских компонент.

## Проектирование компонент

При разработке приложений с помощью Borland Delphi создавать компоненты удобно по следующим причинам:

1. Простота использования. Компонент помещается на форму, и для него необходимо устанавливать значения свойств и писать код обработчиков событий. Поэтому если в проекте какое-либо сочетание элементов управления и  обработчиков связанных с ними  событий встречается в двух местах, то имеет смысл подумать о создании соответствующего компонента. Если же сочетание элементов управления и  обработчиков связанных с ними  событий встречается более двух раз, то создание компонента гарантированно сэкономит усилия при разработке приложения.
2. Простая организация групповой разработки проекта. При групповой разработке отдельные части проекта можно определить как компоненты и поручить эту работу разным программистам. Компоненты можно отладить отдельно от приложения,  что сделать достаточно легко.
3. Простой и эффективный способ обмена кодом с другими программистами. Имеется немало сайтов, где можно найти свободно распространяемые компоненты или приобрести их за символическую плату.

### Пакеты компонентов

В Delphi компоненты хранятся в пакетах (packages). Список используемых пакетов компонентов можно вызвать с помощью пункта меню Component/Install Packages. При помощи этого диалога можно добавить новый пакет (Add), удалить имеющийся (Remove).

Пакет содержит две секции. В секции Contains  приведен список модулей, формирующих компоненты данного пакета (\*.pas- и \*.dcu-файлы) и их пиктограммы (\*.dcr-файлы). Секция Required содержит ссылки на другие пакеты, необходимые для работы этих компонентов. Добавление нового компонента к пакету выполняется кнопкой Add, удаление имеющегося – кнопкой Remove. До тех пор пока пакет не будет скомпилирован нажатием кнопки Compile, все изменения, вносимые в пакет, не будут появляться в среде разработки. И наконец, команда Install доступна в том случае, когда содержимое пакета удалено из среды разработки посредством снятия отметки напротив имени пакета в предыдущем диалоге.

Команда Option позволяет выбрать для компиляции пакета опции, аналогичные опциям проекта. В них можно определить тип данного пакета: работающий во время выполнения, работающий во время разработки, или тот и другой одновременно (тип пакета по умолчанию). В опциях определяются каталоги, в которых следует искать необходимые модули и сохранять результаты компиляции. В них также определяются действия, необходимые для отладки: проверять или нет диапазон допустимых значений, как осуществлять оптимизацию, как обрабатывать ошибки ввода-вывода. И наконец, в опции может быть включена информация о версии пакета.

### Шаблоны компонентов

Delphi позволяет создавать простейшие составные компоненты из нескольких обычных компонентов, выбранных на форме во время разработки. Соответствующий эксперт  вызывается с помощью пункта меню Components/Create Component Template. Этот пункт меню доступен, если на форме выделен хотя бы один компонент.

При создании шаблона запоминаются как свойства, измененные программистом в инспекторе объектов, так и обработчики событий, связанные с выделенными элементами управления.

Использовать шаблоны компонентов удобно, когда необходимо перенести ряд компонентов вместе с обработчиками событий с одной формы на другую. Для этого все они выделяются, создается шаблон компонентов, который и помещается на новую форму. При этом будут перенесены не только сами компоненты, но и обработчики событий, чего нельзя достичь при вызове команд Copy/Paste – в последнем случае обработчики событий будут утеряны.

### Создание простейшего компонента

При написании нового компонента необходимо ясно представлять, что компонент создается для программистов, а не для конечных пользователей. При этом желательно, чтобы программист не вникал в детали реализации компонента, а просто пользовался экспонируемыми им свойствами и событиями. Это достигается очень тщательным тестированием. Новый компонент необходимо тестировать даже в ситуациях, для работы в которых он явно не предназначен.

Поставим задачу следующим образом. Необходимо создать кнопку, которая будет издавать писк при нажатии, и реализовать ее в виде компонента так, чтобы программист мог поместить ее на форму и воспользоваться ею.

Создание компонента начинается с выбора пункта меню Component/New components. После этого сразу же появляется диалог New Component.

В этом диалоге необходимо определить класс-предок, имя вновь создаваемого класса, страницу на палитре, куда будет помещен новый компонент, имя модуля, содержащего реализацию нового компонента, и путь к нему. Если новый компонент использует другие модули, путь к которым не описан, то их необходимо определить в поле Search Path.

Необходимо в качестве класса-предка выбрать класс, который максимально приближен по свойствам к создаваемому классу. Для нашей задачи можно, например, выбрать в качестве предка TWinControl, но в этом случае нам потребуется реализовывать все визуальные эффекты нажатия кнопки и т.д. Поэтому мы выбираем в качестве предка TButton.

Имя вновь создаваемого класса должно отражать содержание компонента и ни в коем случае не совпадать с именем уже зарегистрированного компонента! При выборе страницы необходимо знать, что если задать имя несуществующей страницы, то будет создана новая.

И наконец, при нажатии как кнопки Install, так и кнопки OK, будет создана заготовка для реализации нового компонента. После того как компонент будет инсталлирован, его можно поместить на форму. Теперь все изменения, вносимые в код реализации компонента, будут компилироваться вместе с проектом, и программист сразу же будет получать сообщения об ошибках.

Итак, после нажатия кнопки Install появляется еще один диалог, который позволяет определить пакет, куда будет помещен данный компонент.

В этом диалоге имеются две страницы, на первой из них можно выбрать один из существующих пакетов, а на второй – создать новый. После выбора пакета и нажатия клавиши OK  вызывается редактор пакета, куда автоматически помещается вновь созданный модуль реализации нового компонента. Полезно не закрывать его, а сдвинуть в один из углов экрана, чтобы он мог быть активирован нажатием клавиши мыши.

Одновременно в редакторе кода будет создана «заготовка» для описания нового компонента:

unit ButtonBeep;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

StdCtrls;

type

TButtonBeep = class(TButton)

private

{ Private declarations }

protected

{ Protected declarations }

public

{ Public declarations }

published

{ Published declarations }

end;

procedure Register;

implementation

procedure Register;

begin

RegisterComponents('Samples', [TButtonBeep]);

end;

end.

Один из способов создания компонентов — переписывание уже имеющихся методов. При рассмотрении файла StdCtrls.pas, где реализованы исходные коды для компонента TButton, можно отметить в нем наличие динамического метода Click, который можно переписать. Поэтому вновь возвращаемся к исходному коду, созданному экспертом Delphi при создании компонента (убираем конструктор и метод BtClick). Затем в секции public определяем заголовок метода:

procedure Click; override;

и приводим реализацию метода:

procedure TButtonBeep.Click;

begin

inherited Click;

beep;

end;

Можно убедиться, что кнопка при нажатии издает писк. Кроме того, при назначении обработчика событий в инспекторе объектов этот обработчик выполняется и писк не исчезает! Компонент реализован корректно.

Теперь поменяем пиктограмму компонента TButtonBeep на палитре. По умолчанию для нового компонента используется  пиктограмма компонента-предка. Для этого вызовем редактор Image Editor командой Tools/Image Editor. В редакторе вызовем команду File/New/Component Resource File (\*.dcr). После команды Resource/New/Bitmap появится диалог, в котором предлагается размер пиктограммы 32х32. Эти размеры по умолчанию следует изменить на 24х24 – такой размер обязаны иметь пиктограммы компонентов! После нажатия кнопки OK следует нарисовать какое-либо изображение при помощи стандартных инструментов, похожих на инструменты редактора Paint. Помните, что цвет левого нижнего пиксела является цветом маски – данный цвет будет «прозрачным».

После этого необходимо переопределить имя ресурса с пиктограммой, по умолчанию его имя ‑ Bitmap1. Новое имя ресурса обязано совпадать с именем класса – в нашем случае TButtonBeep.

Теперь необходимо сохранить файл с пиктограммой в том же самом каталоге, где находится модуль, содержащий процедуру Register для данного компонента, и с тем же самым именем, что и имя модуля. Только вот расширение у файла будет не \*.pas, а \*.dcr. Файл с пиктограммой компонента готов. Для того чтобы поменять пиктограмму, необходима повторная регистрация компонента. Для этого необходимо:

* активизировать окно с пакетом, содержащим данный компонент (в нашем случае – TestComp), если он не был закрыт при выполнении упражнений. Если был закрыт ‑ то вызвать его с помощью пункта меню Component/Install Packages, затем выбрать пакет (в нашем случае – Test Component Creation) и нажать кнопку Edit;
* нажать кнопку Remove. В возникающем диалоге отметить модуль, который необходимо удалить, в нашем случае – ButtonBeep.pas  После этого следует нажать кнопку OK;
* перекомпилировать пакет компонентов. Если все было выполнено правильно, то возникнет сообщение об обновлении палитры компонентов.
* нажмите кнопку Add и в диалоге инсталляции нового компонента определите путь к модулю ButtonBeep. После нажатия кнопки OK в окне‑редакторе пакета появятся как модуль \*.pas, так и ссылка на файл \*.dcr.
* нажмите кнопку Compile. После компиляции пакета появится сообщение, что компонент TButtonBeep был инсталлирован в палитре компонентов. Теперь пиктограмма компонента соответствует нарисованной.

## 4.9 Разработка windows приложений

Delphi относится к системам визуального программирования. Разработка приложения в Delphi включает два взаимосвязанных этапа:

• создание интерфейса приложения;

• определение функциональности приложения.

Интерфейс приложения определяет способ взаимодействия пользователя и приложения, то есть внешний вид формы (форм) при выполнении приложения, и то, каким образом пользователь управляет приложением. Интерфейс создается путем размещения в форме компонентов, которые называются интерфейсными или управляющими компонентами (элементами).

Создание интерфейса приложения выполняется с помощью Конструктора формы.

Функциональность приложения определяется процедурами, которые выполняются при возникновении определенных событий, например, происходящих при действиях пользователя с управляющими элементами формы. Таким образом, в процессе создания приложения на форму помещаются компоненты и для них устанавливаются необходимые свойства и создаются обработчики событий.

**Простейшее приложение**

Для примера создадим простейшее приложение. Непосредственно после начала создания нового приложения Delphi предлагает разработчику "пустую" форму. Данная форма не является пустой в буквальном смысле слова — она содержит основные элементы окна Windows: заголовок **Forml,** кнопки минимизации, максимизации и закрытия окна, изменения размеров окна и кнопку вызова системного меню окна.\_/S Delphi. Быстрый старт. Именно эта форма отображается при первом запуске Delphi в окне Конструктора формы.

Любое приложение Windows выполняется в соответствующем окне и даже если оно ничего не делает в смысле функциональности, то есть является пустым, все равно должно иметь свое окно. Delphi — это среда разработки приложений под Windows, поэтому изначально для любого разрабатываемого приложения автоматически предлагает окно (форму), для которой уже созданы два файла с описанием и модулем.

Таким образом, простейшее приложение создается автоматически каждый раз в начале работы над новым проектом и является отправной точкой для дальнейшей работы. Это приложение имеет минимум того, что нужно любому приложению, выполняемому в среде Windows, но ни одним элементом больше.

Простейшее приложение представляет из себя заготовку или каркас, обеспечивающий разработчика всем необходимым каждому приложению вообще. Каркас приложения представляет собой полностью завершенное и функционирующее приложение, которое просто "ничего не делает". Вместе с тем, это пустое окно выполняет достаточно большую работу. Например, оно ожидает действий пользователя, связанных с мышью и клавиатурой, и реагирует на изменение своего размера, перемещение, закрытие и некоторые другие команды.

При конструировании приложения разработчик добавляет к простейшему приложению новые формы, управляющие элементы, а также новые обработчики событий.

**Создание интерфейса приложения**

Интерфейс приложения составляют компоненты, которые разработчик выбирает из Палитры компонентов и размещает на форме, сами компоненты являются своего рода строительными блоками. Компоненты являются структурными единицами и делятся на визуальные (видимые) и невизуальные (системные). Понятие видимый и невидимый относятся только к этапу выполнения, на этапе проектирования видны все компоненты приложения.

К визуальным компонентам относятся, например, кнопки, списки или переключатели, а также форма. Так как визуальные компоненты используются пользователем для управления приложением, то эти компоненты также называют управляющими компонентами или элементами управления. Именно визуальные компоненты образуют интерфейс приложения.

К невизуальным компонентам относятся компоненты, выполняющие вспомогательные, но не менее важные действия, например, таймер Timer или набор данных Table. Компонент Timer позволяет отсчитывать интервалы времени, а компонент Table представляет записи таблицы базы данных. При создании интерфейса приложения для каждого компонента выполняются следующие операции:

• выбор компонента в Палитре компонентов и размещение его на форме;

• изменение свойств компонента.

После размещения компонента на форме Delphi автоматически вносит изменения в файл модуля и файл описания. В описание класса формы (файл модуля) для каждого нового компонента добавляется строчка формата

<Название компонента>: <Тип компонента>;

Название компонента является значением его свойства Name, а тип определяется выбранным компонентом. Например, для кнопки Button эта строчка первоначально будет иметь вид:

Buttonl: TButton;

Для размещения на форме нескольких одинаковых компонентов удобно перед выбором компонента в Палитре компонентов нажать и удерживать клавишу <Shift>.

После размещения компонента в форму с помощью мыши можно изменять его положение и размеры. Внешний вид компонента определяют его свойства, которые доступны в окне Инспектора объектов, когда компонент выделен на форме и вокруг него отображаются маркеры выделения. Для выделения (выбора) формы достаточно щелкнуть в любом ее месте, свободном от других компонентов. В ниспадающем списке, расположенном в верхней части окна Инспектора объектов, отображается название компонента и его тип. Выбрать тот или иной компонент и соответственно получить доступ к его свойствам также можно, выбрав этот компонент в списке Инспектора объектов. В левой части окна Инспектора объектов приводятся названия всех свойств компонента, которые доступны на этапе разработки приложения. Для каждого свойства справа содержится значение этого свойства. Кроме этих свойств, компонент может иметь свойства, которые доступны только во время выполнения приложения.

Свойства представляют собой атрибуты, определяющие способ отображения и функционирования компонентов при выполнении приложения. Каждый компонент имеет значения свойств по умолчанию. После помещения в форму компонента его свойства можно изменять в процессе проектирования или в ходе выполнения приложения.

Управление свойствами в процессе проектирования заключается в изменении значений свойств компонентов. Разработчик может изменить значение свойства компонента, введя или выбрав нужное значение. Для утверждения нового значения свойства достаточно нажать клавишу <Enter> или перейти к другому свойству или компоненту. Для отмены изменений необходимо нажать клавишу <Esc>. Если свойству введено неправильное значение, то выдается предупреждающее сообщение, а изменения значения отвергаются. Изменения свойств автоматически учитываются в файле описания формы, используемом компилятором при создании формы, а при изменении свойства Name вносятся изменения и в описание класса формы.

Каждый компонент для большинства своих свойств имеет значения по умолчанию.

На этапе разработки приложения программист может изменить имя компонента по умолчанию на более осмысленное и соответствующее назначению компонента.

При выполнении приложения значения свойств компонентов (доступных в окне Инспектора объектов) можно изменять с помощью операторов присваивания, к примеру, в обработчике события создания формы. Например, изменение заголовка кнопки Buttoni можно выполнить следующим образом:

Buttoni.Caption := 'Закрыть';

Однако это требует большего объема работ, чем в случае использования Инспектора объектов, кроме того, такие установки вступают в силу только во время выполнения приложения и на этапе разработки не видны, что в ряде случаев затрудняет управление визуальными компонентами. Тем не менее, для наглядности во многих примерах значения отдельных свойств нами устанавливаются с помощью операторов присваивания, а не через Инспектор объектов.

**Определение функциональности приложения**

На любой стадии разработки интерфейсной части приложение можно запустить на выполнение. После компиляции на экране появляется форма приложения, которая выглядит примерно так же, как и при ее разработке. Форму можно перемещать по экрану, изменять ее размеры, минимизировать и максимизировать, а также закрыть нажатием соответствующей кнопки в заголовке или другим способом. То есть форма ведет себя как обычное окно Windows.

Реакция на приведенные действия присуща каждой форме и не зависит от назначения приложения и его особенностей. На форме, как правило, размещены компоненты, образующие интерфейс приложения, и разработчик должен для этих компонентов определить нужную реакцию на те или иные действия пользователя, например, на нажатие кнопки или включение переключателя. Эта реакция и определяет функциональность приложения.

Допустим, что при создании интерфейса приложения на форме была размещена кнопка Button, предназначенная для закрытия окна.

Чтобы кнопка могла реагировать на какое-либо событие, для него необходимо создать или указать процедуру обработки события, которая будет вызываться при возникновении этого события. Для создания процедуры обработки события, или обработчика, прежде всего, нужно выделить на форме кнопку и перейти на страницу событий Инспектора объектов, где указываются все возможные для кнопки события.

Так как при нажатии на кнопку возникает событие Onclick, следует создать обработчик именно этого события. Для этого выполняется двойной щелчок в области значения события Onclick, в результате Delphi автоматически создает в модуле формы заготовку процедуры-обработчика. При этом окно Редактора кода переводится на передний план, а курсор устанавливается в то место процедуры, где программист должен написать код, выполняемый при нажатии на кнопку Buttoni. Так как кнопка должна закрывать окно, то в этом месте можно указать Formi.close или просто close.

При создании обработчика события Delphi автоматически вносит изменения в файл модуля и файл описания. В описание класса формы добавляется строка

procedure ButtonlClick(Sender: TObject);

В тело модуля автоматически добавляется процедура обработки события, не содержащая функциональности.

procedure TForml.ButtonlClick(Sender: TObject);

begin

end;

Среда Delphi обеспечивает автоматизацию набора кода при вызове свойств и методов объектов и записи стандартных конструкций языка Object Pascal.

Так, после указания имени объекта и разделяющей точки автоматически появляется список, содержащий доступные свойства и методы этого объекта. Название выбранного свойства или метода автоматически добавляется справа от точки. Если метод содержит параметры, то отображается подсказка, содержащая состав и типы параметров.

Название обработчика TForml.ButtonlClick образуется прибавлением к имени компонента названия события без префикса on. Это название является значением события, для которого создан обработчик, в нашем случае — для события нажатия кнопки onclick. При изменении через Инспектор объектов имени кнопки происходит автоматически переименование этой процедуры во всех файлах проекта.

Аналогично создаются обработчики для других событий и других компонентов.

Для удаления процедуры-обработчика достаточно удалить код, который программист вносил в нее самостоятельно. После этого при сохранении или компиляции модуля обработчик будет удален автоматически из всех файлов проекта.

Для каждого события можно назначить обработчик, принадлежащий к типу этого события. После выбора в списке нужной процедуры она назначается обработчиком события.

Одну и ту же процедуру можно связать с несколькими событиями, в том числе для различных компонентов. Такая процедура называется общим (разделяемым) обработчиком и вызывается при возникновении любого из связанных с ней событий. В теле общего обработчика можно предусмотреть анализ, для какого именно компонента или события вызвана процедура и в зависимости от этого выполнить нужные действия.

# Литература

1. Голицына О.Л., Попов И.И. Основы алгоритмизации и программирования. – Форум, 2015
2. Зоткин С.П. Программирование на языке высокого уровня С/С++. Конспект лекций. – МГСУ, 2017
3. [Нагаева И.А.](https://my-shop.ru/shop/search/a/sort/z/page/1.html?f14_39=0&f14_16=0&f14_6=%cd%e0%e3%e0%e5%e2%e0%20%c8%2e%c0%2e&t=12&next=1), [Кузнецов И.А.](https://my-shop.ru/shop/search/a/sort/z/page/1.html?f14_39=0&f14_16=0&f14_6=%ca%f3%e7%ed%e5%f6%ee%e2%20%c8%2e%c0%2e&t=12&next=1) Программирование: delphi. Учебное пособие для СПО. – [ЮРАЙТ](https://my-shop.ru/shop/producer/199/sort/a/page/1.html), 2018
4. Огнева М.В., Кудрина Е.В. Программирование на языке С++: практический курс. Учебное пособие для СПО. – ЮРАЙТ, 2017
5. Озерова М.И., Жигалов И.Е. Программирование в среде Delphi. Учебное пособие. – НОБЕЛЬ ПРЕСС, 2014
6. [Павловская Т.А.](https://my-shop.ru/shop/search/a/sort/z/page/1.html?f14_39=0&f14_16=0&f14_6=%cf%e0%e2%eb%ee%e2%f1%ea%e0%ff%20%d2%2e%c0%2e&t=12&next=1)C/C++. Программирование на языке высокого уровня. – [Питер](https://my-shop.ru/shop/producer/13/sort/a/page/1.html), 2019
7. Семакин И.Г., Шестаков А.П. Основы алгоритмизации и программирования. Практикум – М.: Издательский центр «Академия», 2016