***Программирование- экзамен***

## Алгоритм. Свойства алгоритмов

**Алгоритм** – строго определенная последовательность действий, выполнение которых приводит к решению поставленной задачи за конечное число шагов.

**Алгоритмизация** – процесс разработки алгоритма (плана действий) для решения задачи.

Можно выделить три основных вида алгоритмов: линейные, разветвляющиеся и циклические алгоритмы.

В алгоритмах *линейной* структуры действия выполняются последовательно одно за другим:

В алгоритмах *разветвленной* структуры в зависимости от выполнения или невыполнения какого-либо условия производятся различные последовательности действий. Каждая такая последовательность действий называется *ветвью алгоритма*.

В алгоритмах *циклической* структуры в зависимости от выполнения или невыполнения какого-либо условия выполняется повторяющаяся последовательность действий, называющаяся телом цикла.

**Вспомогательный алгоритм –** алгоритм, который можно использовать в других алгоритмах, указав только его имя*.* Например: вы в детстве учились суммировать единицы, затем десятки, чтобы суммировать двузначные числа содержащие единицы вы не учились новому методу суммирования, а воспользовались старыми методами.

### Свойства алгоритмов

**Понятность** – каждая команда должна входить в систему команд исполнителя.

**Дискретность** – **это разбиение алгоритма на ряд отдельных законченных** команд **(шагов**), каждая из которых должна быть выполнена прежде, чем исполнитель перейдет к выполнению следующей.

**Детерминированность (точность, определенность)** – команда алгоритма исполнителем должна пониматься однозначно. Не должно быть двоякого толкования команды.

**Результативность** и **конечность** – за конечное число шагов алгоритм либо должен приводить к решению задачи, либо после конечного числа шагов останавливаться из-за невозможности получить решение с выдачей соответствующего сообщения, либо неограниченно продолжаться в течение времени, отведенного для исполнения алгоритма, с выдачей промежуточных результатов.

**Массовость** – алгоритм решения задачи разрабатывается в общем виде – он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. (Исходные данные могут выбираться из некоторой области, которая называется *областью применимости алгоритма*.)

*Пример. алгоритм выполнения открывания двери.*

*1. Достать ключ из кармана. 2. Вставить ключ в замочную скважину. 3. Повернуть ключ два раза против часовой стрелки. 4. Вынуть ключ.*

*Результативность*. В алгоритме должны быть предусмотрены все возможные пути решения, в том числе *альтернативные*. Пример: алгоритм нахождения *большего из двух заданных чисел А и В:*

*1. Из числа А вычесть число В. 2. Если получилось отрицательное значение, то сообщить, что число В больше. 3. Если получилось положительное значение, то сообщить, что число А больше.* Ведь если оба числа равны, то не получится *никакого сообщения*. Значит, надо обязательно предусмотреть это вариант, например: 4. *Если получился ноль, то сообщить, что числа равны.*

Массовость – один и тот же алгоритм можно использовать с разными исходными данными.

Например: *алгоритм приготовления любого бутерброда.*

*1. Отрезать ломтик хлеба. 2. Намазать его маслом. 3. Отрезать кусок любого другого пищевого продукта (колбасы, сыра, мяса). 4. Наложить отрезанный кусок на ломоть хлеба.*

*детерминированность (процесс применения правил определен однозначно);*

Блок-схемы позволяют описать алгоритм с использованием минимального *количества слов и независимо от выбранного процедурного языка*. Порядок действий указывается с помощью стрелок.

Программа представляет собой запись алгоритма на каком-либо компьютерном языке программирования.

Данные – понятие, определить которое в общем случае невозможно. Применительно к процессу разработки алгоритмов данные представляют собой объект обработки для алгоритма (программы)

1. **Суть методов нисходящего и восходящего проектирования**

При нисходящей разработке алгоритм большинства подпрограмм заменяется **заглушками** (простейший код для имитации ее работы), которые постепенно (по мере их разработке и тестирования уже написанной программы) заменяются реальным алгоритмом. Программа тестируется так, как будто уже целиком закодирована, при этом текст заглушек должен меняться так, чтобы присваиваемые в нем результаты соответствовали проводимому тесту.

После замены очередной заглушки реальным алгоритмом выполнения подзадачи, вся программа вновь тестируется с помощью всего набора тестов для всей программы, и также для этой конкретной подпрограммы могут быть созданы отдельные тесты для всесторонней проверки ее работы на разных наборах входных и выходных данных, которые в масштабах всей программы могут быть промежуточными.

При восходящей разработке сначала создается код подпрограмм, для тестирования которых пишутся отдельные программы – **драйвера**, в каждой из которых выполняется ввод исходных данных, необходимых для проверки алгоритма конкретной подпрограммы, а после ее выполнения выводятся полученные результаты. Затем отлаженные части собираются вместе для создания алгоритма выполнения всей задачи, и тестируется общая логика работы программы.

В методе нисходящего проектирования Вы вначале пишете основную программу, используя средства вызова подпрограмм, причем в качестве подпрограмм вначале Вы вводите "заглушки".

Затем, будучи уверенным в правильности логического построения основной программы, Вы детально "расписываете" каждую подпрограмму, вызывая по мере необходимости подпрограммы более низкого уровня. Этот последовательный процесс продолжается, пока программа не будет завершена и проверена.

При другом методе - восходящем проектировании (программировании "снизу вверх") - Вы вначале пишете подпрограммы нижнего уровня и тщательно их тестируете и отлаживаете. Далее Вы добавляете подпрограммы более высокого уровня, которые вызывают подпрограммы нижнего уровня, и так до тех пор, пока Вы не достигнете программы самого верхнего уровня. Метод проектирования "снизу вверх" пригоден при наличии больших библиотек стандартных подпрограмм.

Учтите, что иногда лучшим является гибрид двух методов. Однако в обоих случаях каждая подпрограмма должна быть небольшой, так чтобы можно было охватить одним взглядом всю ее логику.

### Этапы разработки программ. Цикл разработки программного обеспечения

● Жизненный цикл программного обеспечения – это весь процесс его создания и применения от начала до конца. Этапы жизненного цикла и удельный вес общих затрат на каждый из этапов приведены на рис. 1.1.

Кратко: 1) определение требований заказчика, определение входных/выходных данных, их типов и т.п., составление спецификации (начало), определение "подводных камней" типа аномалий и пр., составление функциональных тестов и метода решения: 2)Составление алгоритма решения в общем виде. Пока без кода просто придумываете "как будет работать программа"[псевдокод - это такой "недокод", он используется для описания кода программы словами]; 3) Написание кода программы по составленному алгоритму; 4)Поиск ошибок программы, исправление ошибок, прогон тестов, исправление новых ошибок[функциональные тесты - составляются ещё до алгоритма, проверяют работу программы в целом, структурные - проверяют отдельные фрагменты программы (циклы, подпрограммы, прочие алгоритмы)]; 5) Документирование - этап, не прекращающийся на протяжении всей разработки, суть : что-то добавили/изменили в программе - запиши в отсчёт.}

Требования пользователя

Сопровождение

Определение требований-спецификаций

Проектирование

Кодирование

Отладка и тестирование

Документирование

Решение пользователю

Рис. 1.1. Жизненный цикл программного обеспечения

Как видно из рисунка, примерно половина затрат приходится на этап, называемый сопровождением. Суть сопровождения – внесение изменений в уже находящуюся в эксплуатации программу и поддержание её в рабочем состоянии.

● Этапы разработки программы отображены в нижней половине рисунка в той последовательности, как они выполняются. Этапы обладают свойством преемственности, но принципиально важно, чтобы эта преемственность реализовалась сверху вниз, т.е. чтобы предшествующие этапы захватывали последующие, а не наоборот.

Кратко охарактеризуем этапы.

1. Составление внешней спецификации. Цель – уточнение постановки задачи, а именно:

* точная формулировка задачи и всех вводимых ограничений;
* определение состава и структуры входных и выходных данных и формы их ввода и вывода;
* описание аномалий – ситуаций, при которых решение не может быть получено (выход данных за границы диапазона, деление на ноль и т.п.);
* описание тестов – примеров, на которых будет проверяться работа программы;
* описание метода решения – связей входных и выходных величин, общей идеи и схемы решения задачи.

2. Проектирование алгоритма. Цель – создание алгоритма, понятного человеку. Центральный и наиболее трудоемкий этап. Требует творчества и не может быть полностью формализован. Созданный на этом этапе текст алгоритма – носитель всей информации о решении, поэтому для записи алгоритма целесообразно использовать наиболее универсальные и одновременно точные средства. Такими средствами являются полуформальные языки, используемые для проектирования – псевдокоды.

3. Кодирование программы состоит в переводе алгоритма с одного языка, достаточно жесткого (псевдокода), на другой, полностью формализованный (язык программирования).

4. Отладка и тестирование. Цель – получение работоспособной программы.

Отладка – процесс изменения программы с целью исправления ошибок.

Тест – совокупность специально подобранных входных и соответствующих им выходных данных, используемая для контроля правильности программы.

Тестирование – исполнение программы на наборе тестов на компьютере.

Отладка включает тестирование.

Принципиальная трудность проектирования тестов состоит в практической невозможности составления всех тестовых наборов данных для всех возможных режимов работы алгоритма. Поэтому задача проектирования тестов сводится к проектированию ограниченного их набора, гарантирующего с достаточной достоверностью правильную работу программы во всех практически значимых режимах.

Содержание тестов определяется спецификацией задачи и логикой ее решения.

Функциональные тесты составляются на уровне спецификации, до решения.

Структурные тесты составляются для проверки логики решения, или логики работы уже готового алгоритма. Логика определяется последовательностью операций, их условным выполнением или повторением. Совокупность структурных тестов должна обеспечить проверку каждой из таких конструкций.

Чаще всего совокупность тщательно составленных функциональных тестов покрывает множество структурных тестов.

5. Документирование.

Элементы документации – все элементы проекта, полученные при его разработке, т.е. описание результатов отдельных этапов. Если процесс разработки программы организован правильно, то документация появляется постепенно и процесс ее создания сопровождает все этапы. Поэтому стрелка-окружность на рис. 1.1 проходит через все этапы жизни программы

1. **Критерии качества программного продукта**

**Качество** определяется в стандарте ISO 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

1. Работоспособность (при конкретных программных и аппаратных условиях) возможность выполнения программы на имеющейся машине.
2. Правильность (программа должна решать именно поставленную, а не более широкую, более узкую или изменённую задачу)
3. Надёжность (программа должна работать при любых исходных данных - анализировать их правильность и выдавать результаты или диагностику ошибок)
4. Читабельность (текст программы - итоговый носитель всей информации о решении, поэтому он должен иметь четкую и ясную организацию, отображающую решение)
5. Лёгкость отладки и тестирования
6. Модифицируемость (возможно внесение изменений в программу)
7. Документированность (наличие документации по всему процессу разработки, начиная от постановки задачи); документированность обеспечивает возможность передачи программы другим лицам, или отчуждение программы (это заключение договора, по которому правообладатель передает другой стороне исключительные права на программу)
8. Простота пользования, наличие сервиса (дружественный интерфейс (Интерфейс, обеспечивающий человеку удобные формы взаимодействия с информационной системой)+ справка) Дружественный интерфейс обеспечивает диалог с системой, используя для этого пиктограммы, кнопки, меню, подсказки, выдаваемые на экран и динамики.
9. Эффективность применительно к компьютеру (использование минимума машинных ресурсов- памяти и времени выполнения)

Критерии 4,6,7 взаимосвязаны, критерий 5 ими определяется; критерии 3-8 противоречат 9-му, и сам он внутренне противоречив. Цель- повышение производительности труда программистов, причём «производительность» подразумевает оптимальное использование и человеческих, и компьютерных ресурсов.

1. **Базовые структуры алгоритмов (базис Вирта, базис Дейкстры). В виде блок-схем и на Паскале.**

Структурированные алгоритмы - это алгоритмы, которые строятся с использованием только трех правил композиций вычислительных действий (3 базовые управляющие структуры):

- последовательные действия (следование); альтернативные действия (выбор); повторяемые действия (цикл).

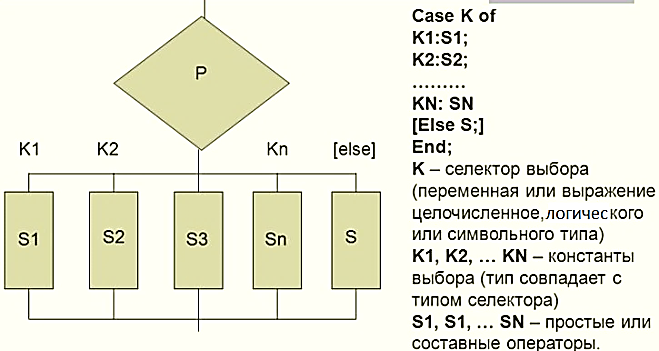
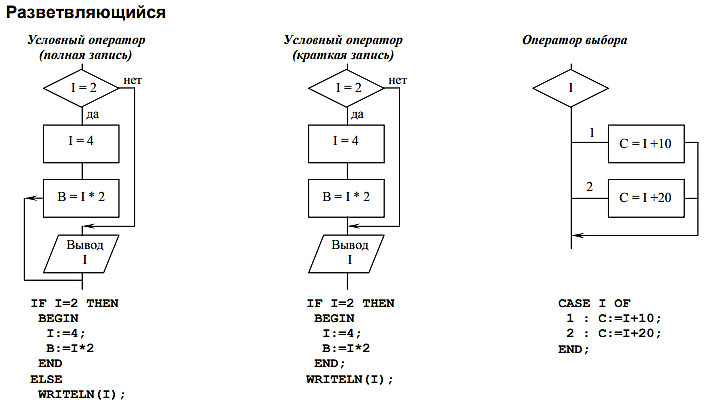
Набор из следования, ветвления и цикла «пока» называется базисом Дейкстры, а набор из следования, ветвления и цикла «до» называется базисом Вирта.  Св-ва базиса:

- независимость (каждая из базисных структур не может быть представлена комбинацией 2-х других);

- полнота (любой алгоритм может быть осуществлен с помощью них);

- эквивалентность базисов (следование и выбор сохраняются, а циклы «до» и «пока» взаимозаменяемы).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Название | | | | Блок-схема | | Операторы псевдокода | |
| 1. Следование  последовательность любых операторов и/или структур. Очередное действие начинается после завершения предыдущего действия или группы действий | | | | S1  S2  Sn  … | | S1; S2; … Sn; | |
| 2  Развилка, ветвление, альтернатива | | 2.1  если-  то-  иначе | | ложь  истина  В  S1  S2 | | если В то  S1  иначе  S2  кесли; | |
| 2.2 если-то  (обход) | | ложь  истина  В  S1 | | если В то  S1  кесли; | |
| 2.3  выбор | | Не равен ни одному из знач.  Знач1  Значn  Знач2  S1  S2  Sn  Sn1  ключ  =  … | | Выбор<ключ> из  <знач1>: S1;  <знач2>: S2;  ………….  <значn>: Sn; [иначе Sn1;] квыб; | |
| • Оператор Sn1 может быть опущен.  • В программировании в том же смысле, что и термин «ключ», используется термин «селектор». | | | |
|  | | | Кодирование в паскале | | | |
| 1 | | | Последовательность операторов, простых или составных.  Составной оператор:  begin  S1; S2; … Sn;  end;  Рассматривается как один оператор. Символ «;» - разделитель операторов. | | | |
| 2 | 2.1 | | if B then  S1  else  S2; | | if (k>0) then  k:=k+1  else  k:=k-1; | |
| 2.2 | | if B then  S1; | | | |
| 2.3 | | сase <селектор > of  <конст1>: S1;  <конст2>: S2;  ………….  <констn>: Sn; [else Sn1;] end {case};  Выполняется только оператор, для которого значение селектора равно константе. | | | |
| S1, S2, …, Sn, Sn1 - один оператор, простой или составной | | | |

****

1. **Виды циклов в языке Паскаль. Параметрические с шагом +1 и -1**

**For <пар\_цик> : = <нач\_знач> to <кон\_знач> do (downto) <оператор>**  
**For, to, do, downto** –зарезервированные слова;<пар\_цик> – параметр цикла; <нач\_знач> – начальное значение; <кон\_знач> – конечное значение; <оператор> – произвольный оператор Паскаля.  
При выполнении оператора***for*** вначале вычисляется выражение <нач\_знач> и осуществляется присваивание <пар\_цик> : = <нач\_знач>. После этого циклически повторяется следующая последовательность действий : 1) Повторяется условие <пар\_цик> <= <кон\_цик>. Если условие не выполняется, оператор for завершает свою работу. 2) Выполняется оператор <оператор>. 3)Происходит наращивание переменной <пар\_цик> на единицу. **to – шаг +1, downto – шаг -1.**

Часто цикл for называют циклом со счетчиком. Этот цикл используется, когда число повторений не связано с тем, что происходит в теле цикла. Т.е. количество повторений может быть вычислено заранее (хотя оно не вычисляется).

В заголовке цикла указываются два значения. Первое значение присваивается так называемой переменной-счетчику, от этого значения начинается отсчет количества итераций (повторений). Отсчет идет всегда с шагом равным единице. Второе значение указывает, при каком значении счетчика цикл должен остановиться. Другими словами, количество итераций цикла определяется разностью между вторым и первым значением плюс единица.  Счетчик – это переменная любого из перечисляемых типов (целого, булевого, символьного, диапазонного, перечисления).

Если телом цикла является другой цикл, то циклы называются вложенными или сложными. Цикл, содержащий в себе другой цикл, называют внешним. Цикл, содержащийся внутри другого цикла, называется внутренним. Внутренний и внешний циклы могут быть любыми из трех видов: FOR, WHILE или REPEAT. При построении вложенных циклов необходимо, чтобы все операторы внутреннего цикла полностью находились в теле внешнего цикла. Возможная глубина вложенности циклов ограничивается объемом памяти компьютера.

При выполнении оператора FOR выполняются следующие действия:

- вычисляется < выражение 1> , которое присваивается параметру цикла;

- проверяется условие окончания цикла : <параметр цикла> больше <выражения 2> при использовании конструкции TO и <параметр цикла> меньше <выражения 2> при использовании конструкции DOWNTO;

- выполняется тело цикла;

- наращивается (TO) или уменьшается (DOWNTO) на единицу параметр цикла;

- все этапы, кроме первого, циклически повторяются.

При использовании оператора необходимо помнить :

- Внутри цикла FOR нельзя изменять начальное , текущее или конечное значения параметра цикла.

- Если в цикле с шагом +1 начальное значение больше конечного, то цикл не выполнится ни разу. Аналогично для шага -1, если начальное значение меньше конечного.

- После завершения цикла значение параметр цикла считается неопределенным, за исключением тех случаев, когда выход из цикла осуществляется оператором GOTO или с помощью процедуры BREAK.

1. **Виды циклов в языке Паскаль. Итерационные циклы( repeat until и while do)**

Итерация в программировании — в широком смысле — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя.

Операторы цикла итерационного типа используются обычно в том случае, если число повторений цикла заранее неизвестно или шаг изменения параметра цикла отличен от +1 или –1.

**While <условие> do <оператор>  
While,do**– зарезервированные слова. <условие> – выражение логического типа; <оператор> – произвольный оператор Паскаля.

(Порядок действий:   
-проверка условия окончания цикла (логическое выражение)

-выполнение тела цикла

-изменение параметра цикла с помощью оператора

-пункты циклически повторяются)

**Repeat <тело цикла> until <условие>**  
**Repeat,until** – зарезервированные слова.<тело\_цикла> – произвольная последовательность операторов. <условие> – выражение логического типа. Операторы <тело\_цикла> выполняется хотя бы один раз, после чего вычисляется выражение <условие> : если его значение есть False, операторы <тело\_цикла> повторяются, в противном случае оператор repeat..until завершает свою работу{ здесь операторные скобки выполняет repeat,until}.

Цикл **while** является циклом с предусловием. В заголовке цикла находится логическое выражение. Если оно возвращает **true**, то тело цикла выполняется, если **false** – то нет.Когда тело цикла было выполнено, то ход программы снова возвращается в заголовок цикла. Условие выполнения тела снова проверяется (находится значение логического выражения). Тело цикла выполнится столько раз, сколько раз логическое выражение вернет **true**. Цикл **while** может не выполниться ни разу, если логическое выражение в заголовке сразу вернуло **false**. Однако такая ситуация не всегда может быть приемлемой. Бывает, что тело цикла должно выполниться хотя бы один раз, не зависимо оттого, что вернет логическое выражение. В таком случае используется цикл repeat – цикл с постусловием.

В цикле **repeat** логическое выражение стоит после тела цикла. Причем, в отличие от цикла **while**, здесь всё наоборот: в случае **true** происходит выход из цикла, в случае **false** – его повторение. В отличие от других операторов цикла оператор данного вида не требует операторных скобок BEGIN - END, так как их роль выполняют REPEAT - UNTIL.Любой алгоритм, реализуемый через оператор FOR, может быть записан с помощью конструкции WHILE. Например, вычисление значения факториала F=N!:

…

F:=1;

I:=1;

WHILE I<=N DO

BEGIN

F:=F\*I;

I:=I+1;

END;

Вычисление F=N! с использованием конструкции REPEAT – UNTIL будет выглядеть следующим образом:

. . .

F:=1;

I:=1;

Repeat

F:=F\*I;

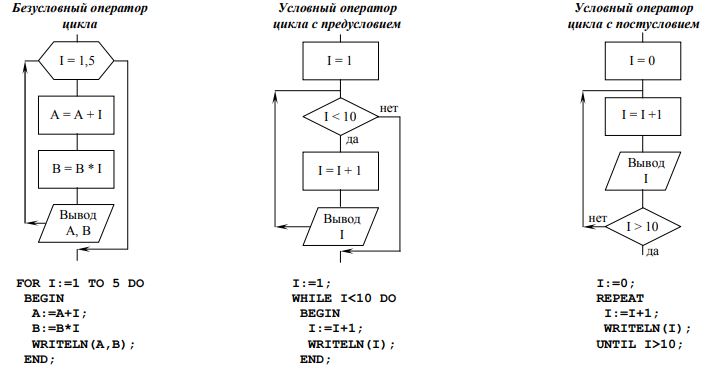
I:=I+1;

Until I>N;

. . .

Здесь используется цикл WHILE для индексации строк, т.к. нас интересуют только четные строки, следовательно, шаг для строк должен быть равен 2. В цикле FOR этого сделать нельзя.

В языке Турбо Паскаль 7.0 имеются процедуры BREAK и CONTINUE . Эти процедуры могут использоваться только внутри циклов FOR, WHILE или REPEAT. Процедура BREAK прерывает выполнение цикла и вызывает переход к оператору, следующему за циклом (может использоваться вместо оператора GOTO).Процедура CONTINUE осуществляет переход к следующему повторению цикла с пропуском последующих операторов тела цикла.

****

1. **Операторы ветвления c единственным и множественным выбором: if и case**

Условный оператор, который называется **IF** (англ. "если") используется для выбора одного из двух направлений хода программы. Выбор последовательности инструкций осуществляется во время выполнения программы в зависимости от выполнения условия.

Общий синтаксис условного оператора **IF**:

**if** {условие} **then**  
  {последовательность команд}

**THEN** - англ. "в таком случае, тогда". Таким образом, строка вполне переводима и на русский язык: "Если условие выполняется, тогда сделать что-то..."

Здесь **условие** - выражение логического типа (логические выражения были рассмотрены нами ранее в уроке №10). Если выражение является истинным (True), т.е. условие выполняется, то выполняется последовательность команд, указанная после THEN. Если же условие не выполняется, данный блок пропускается. В случае, если команд, ассоциированных с условием, несколько, то они заключаются в BEGIN ... END:

**if**{условие} **then**  
**begin**  
  Команда 1;  
  Команда 2;  
  Команда 3  
**end;**

Данную форму записи условного оператора можно назвать сокращённой. Есть и полная форма. Дело в том, что сокращённая форма позволяет выполнить блок команд при выполнении заданного условия, но при этом мы теряем из вида тот случай, когда условие не выполняется. Для нашего примера с делением чисел сокращённой формы недостаточно.

Полная форма даёт возможность указать две последовательности команд: одну - для случая, если условие выполняется, а другую - если не выполняется. Третьего не дано... Синтаксис:

**if** {условие} **then**  
  {команды, если условие выполняется}  
**else**  
  {команды, если условие не выполняется}

**ELSE** - англ. "иначе, в противном случае": "Если условие выполняется, то сделать что-то, а в противном случае сделать что-то другое".  
Аналогично, если требуется выполнить несколько команд, они заключаются в BEGIN ... END:

**if** {условие} **then**  
**begin**  
  {...}  
  {команды, если условие выполняется}  
  {...}  
**end**  
**else**  
**begin**  
  {...}  
  {команды, если условие не выполняется}  
  {...}  
**end**;

Оператор выбора, который называется **CASE**, используется для выбора одного из нескольких направлений дальнейшего хода программы. Выбор последовательности инструкций осуществляется во время выполнения программы в зависимости от равенства значения **переменной-селектора** константе, указанной перед группой инструкций. В качестве переменной-селектора можно использовать переменную только перечислимого типа. Целое число (Integer) - перечислимый тип данных. Другим простым перечислимым типом данных является символьный тип (Char). Синтаксис оператора CASE:

**case** {переменная-селектор} **of**  
  {константа\_1}: {инструкции 1}  
  {константа\_2}: {инструкции 2}  
  {...}  
  **else** {инструкции, если ни одна константа не подошла}  
**end**;

Работает оператор следующим образом: берётся значение переменной-селектора и последовательно сравнивается с каждой из констант. Если значения совпадают, то выполняются инструкции, указанные после данной константы. Если ни одна константа не совпала со значением переменной-селектора, но при этом указан блок else, то выполняются инструкции этого блока. Если блок else не указан, никаких команд выполнено не будет.  
Обратите внимание, что после каждой константы ставится двоеточие ("**:**").  
Помимо одиночных констант можно указывать несколько констант через запятую, а также можно указывать диапазон, задав начальный и конечный элементы и разделив их двумя точками. Пример:

**case** a **of**  
  1,2,3,4,5: {...}  
  **else** {...}  
**end**;  
  
**case** a **of**  
  1..5: {...}  
  **else** {...}  
**end**;

Оба представленных фрагмента кода сработают абсолютно одинаково, но во втором варианте запись констант и короче и удобнее.

**Пример программы с использованием оператора выбора.**Создадим программу, которая будет запрашивать ввод числа от 1 до 12 и выводить название времени года, которому принадлежит месяц с указанным номером.

**var**Month: Integer;  
**begin**  
  Month:=StrToInt(Edit1.Text);  
  **case** Month **of**  
    1,2,12: ShowMessage('Зима');  
    3..5: ShowMessage('Весна');  
    6..8: ShowMessage('Лето');  
    9..11: ShowMessage('Осень')  
    **else** ShowMessage('Число должно быть от 1 до 12.')  
  **end**  
**end**;

* Чтобы задать **диапазон значений**, используется один из двух вариантов:
  + Перечисление через запятую:

|  |
| --- |
| ...  **case** x **of**:  1,2,3,4,5: writeln('от одного до пяти');  ... |

* + Использование ..:

|  |
| --- |
| ...  **case** x **of**:  1..5: writeln('от одного до пяти');  ... |

**Результат выполнения с использованием оператора выбора case:**

|  |  |
| --- | --- |
|  | **var** otmetka:**integer**;  **begin**  writeln('отметка?');  readln(otmetka);  **case** otmetka **of**  1,2 : writeln ('кошмар!');  3: writeln('плохо');  4: writeln('неплохо');  5: writeln('отлично!');  **else** writeln('таких отметок не бывает');  **end**  **end**. |

**Заключение**

Условный оператор IF и оператор выбора CASE позволяют создавать ветвления в программах, т.е. выполнять различные действия в зависимости от заданного условия. Отличие их друг от друга состоит в том, что IF используется для единственного выбора, а CASE - для множественного. Очевидно, что данные операторы могут быть вложены друг в друга (например, в одном из блоков оператора CASE может быть использован IF). При написании какой-либо сравнительно сложной программы без ветвлений не обойтись.

1. **Способы структуризации алгоритмов. Сведение алгоритма с досрочным выходом из цикла for к циклам while и repeat (метод флажка).**

Структурированная программа (или подпрограмма) - это программа, составленная из фиксированного множества базовых конструкций. Рассмотрим основные определения и способы образования этих конструкций в схемах алгоритмов.  
Из операций, развилок и слияний строятся базовые конструкции: следование, ветвление, цикл. Применяя только эти три конструкции, можно реализовать алгоритм решения любой задачи.  
Конструкция, представляющая собой последовательное выполнение двух или более операций, называется следованием.  
Конструкция, состоящая из развилки, двух операций и слияния, называется ветвлением. Одна из операций может отсутствовать.  
Конструкция, имеющая линии управления, ведущие к предыдущим операциям или развилкам, называется циклом.  
Конструкции следование, ветвление и цикл можно представить как операции, так как они имеют единственный вход и единственный выход. Произвольную последовательность операций можно представить как одну операцию.  
В языке ПАСКАЛЬ количество базовых конструкций увеличено до шести, это:  
-следование; -ветвление; -цикл с предусловием; -цикл с постусловием; -цикл с параметром; -вариант.

Методы структурирования – средства, которыми надо воспользоваться для построения алгоритма, а не для преобразования хаотично построенного алгоритма.

Метод флажков: вводится дополнительная переменная с двумя значениями, управляющая циклом. При одном значении цикл продолжается, при другом-завершается. Двузначной переменной служит переменная логического типа, называемая флагом. Пусть сначала флаг=истина. Организуем цикл- пока флаг=истина. Тогда значение флага должно сохранятся истинным, если цикл надо продолжить; изменится на ложь, если цикл надо закончить.Преимущество метода флажка: универсальность для любого числа условий выхода из цикла.

flag := true;  
i := 1;  
while flag do  
begin  
if i>0{любое условие} then  
flag:= false;  
end;

Сведение алгоритма с досрочным выходом:   
У нас есть цикл (работающий) for и в нём необходимо оформить досрочный выход, что надо делать:

-Сначала необходимо грамотно переписать алгоритм, заменив цикл FOR на WHILE или REPEAT

-Убедиться, что новый цикл работает и отсутствуют ошибки

-Добавить в описание переменных новую - флажок. Можно логической переменной, можно перечисляемой, не суть важно. Например, добавляем логическую.

-Добавить в тело цикла алгоритм изменения "флажка". Это может быть простое линейной изменение, условие (for), выбор (case), внутренний цикл или их комбинации. Например, пусть в теле цикла при досрочном выходе флажок принимает значение TRUE

-Добавить в условие "повторения" цикла новое условие. Например:  
While i<=n do ----> While (i<=n) and (notFL) do  
repeat ... until i>n -----> repeat ... until (i>n) or (FL)  
При подобных изменениях, в случае, когда флагу FL присваивается истина TRUE, будет совершён досрочный выход.  
Также следует упомянуть, что досрочный выход из цикла может быть совершён оператором BREAK, но в билете сказано "флажок".

1. **Классификация типов данных в Delphi. Типы с плавающей точкой (Double, Single, Real, Extended)**

ТИПЫ

Простые

Структурированные

Указатели

Процедурный тип

Порядковые

Вещественный

Целый с фикс.точкой

Логический с плав.точкой

Символьный

Перечисляемый

Диапазон (интервальный тип)

Строки

Массивы

статич. и димамические

Записи

и записи с вариантами

Множества

Файлы

текстовые, типизир. и бестиповые

Классы, интерфейсы

Вариант

|  |  |  |  |
| --- | --- | --- | --- |
| Вещественные с плавающей точкой | | | Присвоение значений  r:= 2.6; si:= 2; d:= -1.24e-5; |
| Тип | Размер, байт | Порядок, точность |
| *Single* | 4 | *e*-45..*e*+38  7-8 значащих цифр | Особые операции:  Round(r) 🡪 2 округление (r = 2.5)  Round(r) 🡪 3 округление (r = 2.6)  Trunc(3.7) 🡪 3 целая часть  Int(3.7) 🡪 3.0 обнуление дробной части числа  Frac(3.7) 🡪 0.7 дробная часть  R:= d / 2; //деление с вещественным результатом |
| *Double, Real* | 8 | *e*-324..*e*+308  15-16 значащих цифр |
| *Extended* | 10 | *e*-4951..*e*+4932  19-20 значащих цифр | Вывод значений  Writeln('r =', r:3:1); {X.X 2.6}  Writeln('d =', d:9:7); {+X.XXXXXXX -0.0000124} |
| Примеры:  1.234567  -9876.54321  1*e*-3 (1∙10-3 = 0.001) | | |

Пара замечаний вещественным с "плавающей" запятой: с данными типами можно выполнять стандартные действия типа присвоение, сравнение, ввод/вывод, математические операции (включая деление!), но есть и ряд специальных команд, их лучше выучить!

**Типы в Delphi:**

1. **Простые**
   1. Порядковые
      * Целые
      * Символьные, могут использоваться различные кодировки
      * Логические,имеет два значения: истина(true) и ложь(false).
      * Перечисляемый, может хранить только те значения, которые прямо указаны в его описании.
      * Интервальный (диапазон)
   2. Вещественные

* С плавающей точкой, т. е. число приводится к виду m\*2e, где m — мантисса, e — экспонента причем 1/2<=m<=1, а e — целое число и хранятся знак, и числа m и e.
* С фиксирующей точкой (деньги), то есть хранятся знак и цифры целой и дробной частей

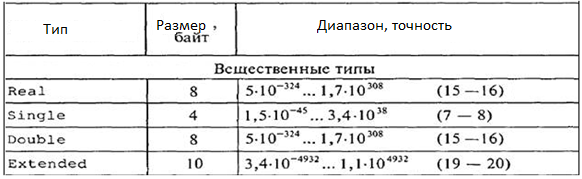
1. **Структурированные (сложные)**
   1. Массивы (совокупность однородных элементов, хранящихся в соседних ячейках памяти, имеющих общее имя и отличающихся по индексам)

* Статические
* Динамические
* Одномерные
* Многомерные
  1. Файловые переменные
* Текстовый
* Типизированный
* Безтиповой (нетипизированный)
  1. Множества (совокупность не более чем 256 неповт. однородных простых элем-тов порядковых типов, хранящихся в соседних битах памяти, имеющих общее имя)
  2. Записи (совокупность неоднородных элементов, хранящихся в соседних ячейках памяти, имеющих общее имя и отличающихся по именам полей)
  3. Объекты
  4. Классы, нтерфейсы

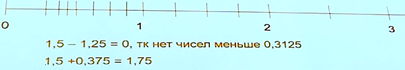
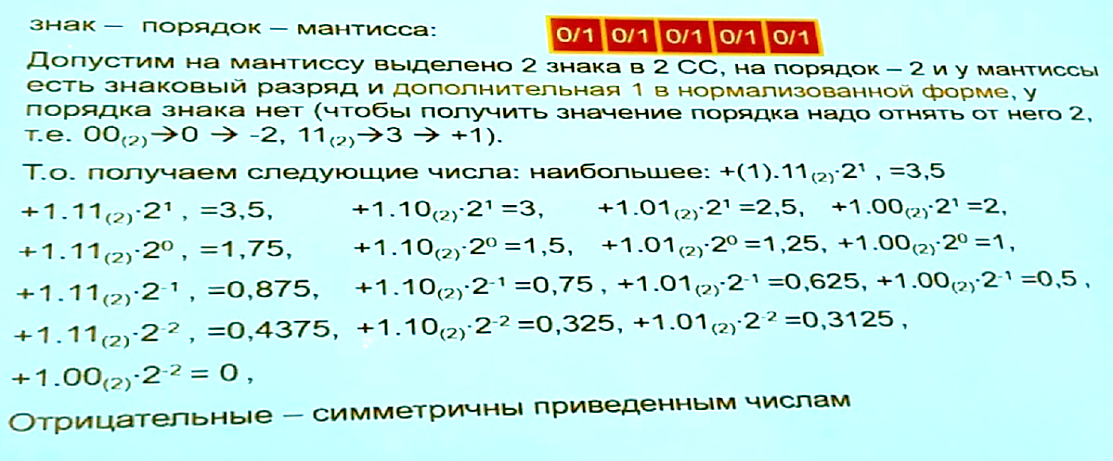
1. **Указатель** (хранит адрес в памяти компьютера, указывающий на какую-либо информацию, как правило — указатель на переменную)
2. **Процедурный**
3. **Строки**

* Статические
* Динамические

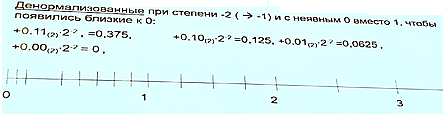
1. **Вариант**

*Вещественные типы с плавающей точкой*

Точность определяет количество значащих цифр, которые представляют число без потери данных.



**Денормализованные числа** - это способ увеличить количество представимых числом с плавающей запятой значений около нуля, дабы повысить точность вычислений.



Мантисса (1).11?. Степень двойки это порядок. *X=s\*m\*2e s- знак, m-мантисса ,e-порядок*

+1.11(2)\*21=1\*21+1\*20+1\*2-1=3,5

+1.01(2)\*2-2=1\*2-2+0\*2-3+1\*2-4=0,3125

1. **Порядковые типы. Целые типы в Delphi, тип диапазон**

К порядковым типам относятся целые -2<-1<0<1<2, логические false<true, символьные ‘0’<’9’<’A’<’Z’<’Я’, перечислимые типы и тип-диапазон (интервальный).

Порядковые типы имеют конечное число упорядоченных значений. К ним относятся функции:

• Ord(x) — возвращает порядковый номер значения данного выражения. Для целых типов возвращает само значение х, для логического 0 или 1, для символьного -значение в диапазоне от 0 до 255, для перечислимого - значение в диапазоне от О до 65535. Для типа-диапазона результат зависит от свойств базового порядкового типа.

• Pred(x) - возвращает величину, предшествующую значению данного выражения.

• Succ(x) - возвращает величину, следующую за значением данного выражения.

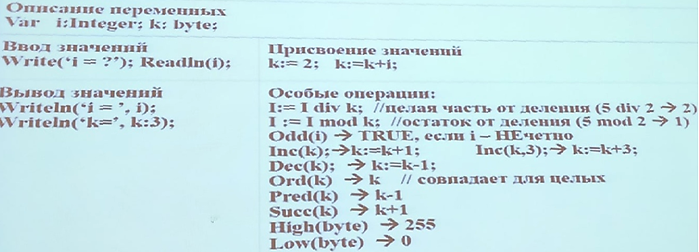
Заметим, что функция Pred не определена для самого меньшего значения порядкового типа, a Succ - для самого большего.

Для константы или переменной порядкового типа определены также следующие функции:

• High(x) - возвращает максимальное возможное значение для аргумента х.

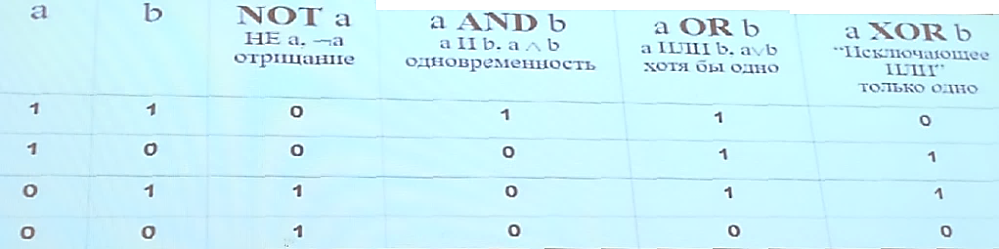
• Low(x) — возвращает минимальное возможное значение для аргумента х.

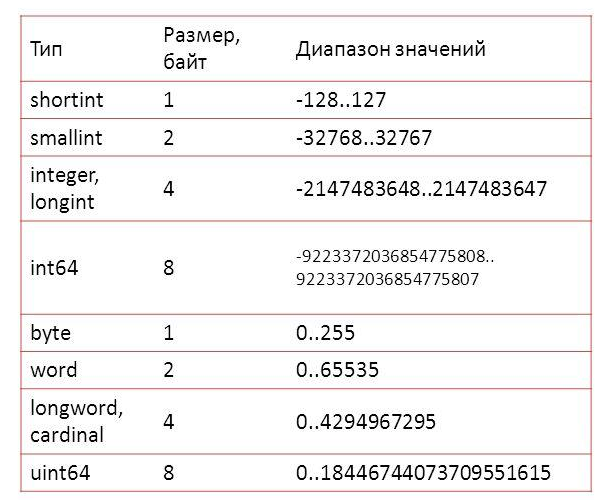
Целые типы базовый



Особые побитовые операции: not, and,xor,or

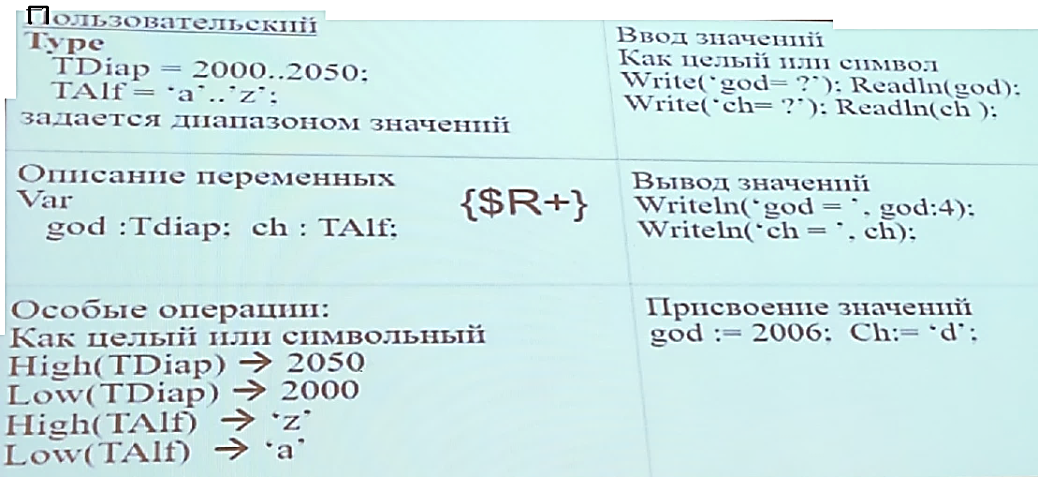
Сдвиги: 7→0,00111 сдвиг вправо 0,00011→3 (7 div 2=7 shr 1=3), сдвиг влево 0,01110→3 (7 \*2=7 shl 1=14)

SHR- shift right, SHL-shift left



Суть в том, что типы "зациклены" сами на себе по диапазону. Например: тип BYTE имеет всего 256 различных значений (от 0 до 255), и если мы к числу 255 типа byte прибавим 2, то результатом будет 1. То есть изменение числа идёт до "конца" диапазона типа, прекращается, а потом продолжается "с другого конца" диапазона. Др пример: тип ShortInt : 120 + 28 = -108.

Интервальный тип (диапазон)



1. **Порядковые типы. Символьный тип. Таблицы символов.**

Значениями ***символьного типа*** является множество символов компьютера. Каждому символу присваивается целое число в диапазоне от 0 до 255. Это число служит кодом внутреннего представления символа, его возвращает функция **ORD**. В Delphi 7 есть три символьных типа :

**Тип *ANSIChar*** представляет собой так называемые Ansi-символы. Это символы, которые используются в операционных системах семейства Windows(размером 1 байт). **Тип *WideChar*** предназначен для хранения так называемых Unicode-символов, которые в отличие от Ansi-симвояов занимают два байта. Это позволяет кодировать символы числами от 0 до 65535 и используется для представления различных азиатских алфавитов. Первые 256 символов в стандарте Unicode совпадают с символами Аnsi.**Тип Char** в Delphi 7 эквивалентен типу AnsiChar и обеспечивает наибольшую производительность. Для отображения множества символов в подмножество натуральных чисел и обратно имеются следующие две стандартные функции:

**ord(c)** - дает порядковый номер символа с;  
***chr(i)*** - дает символ с порядковым номером i.

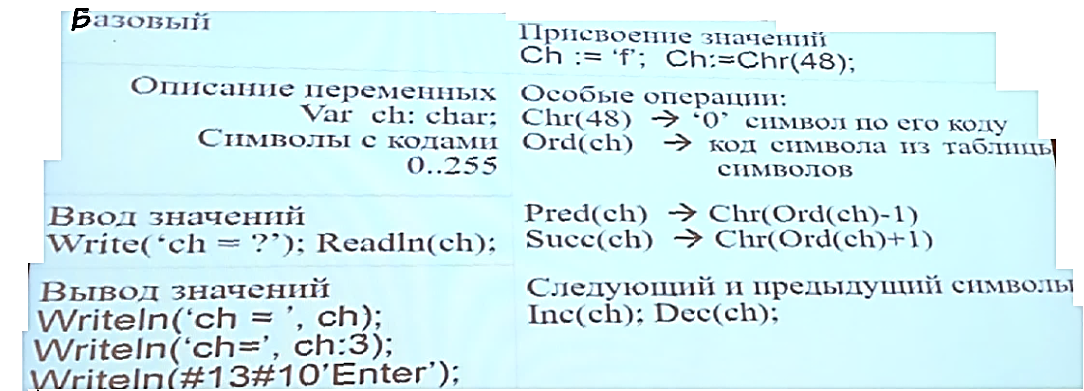
Вместо функции chr можно воспользоваться оператором #, который также возвратит символ, код которого указан после оператора. Например, если переменная s имеет тип char, то мы можем записать следующие операторы, которые будут эквивалентны:

s := chr(72);

s := #72;

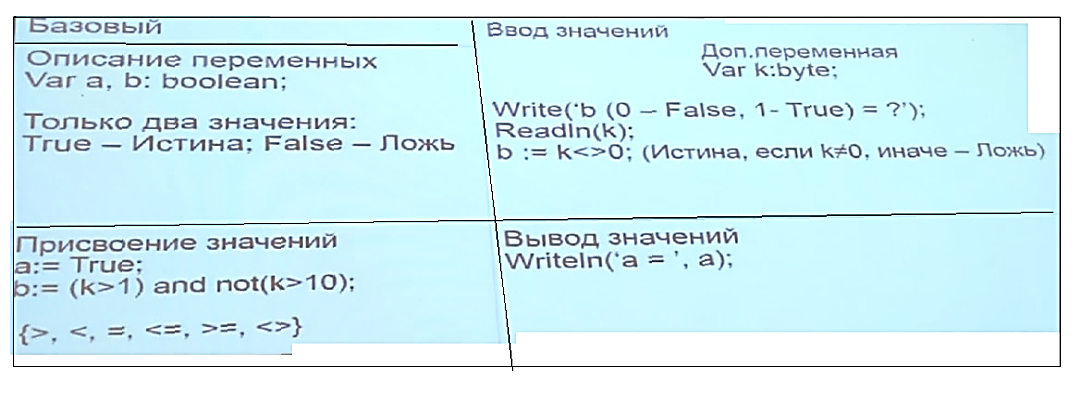
***UpCase(CH)*** – возвращает прописную букву, если CH– строчная латинская буква, в противном случае возвращает сам символ.  
***Length***(<строка>) – функция, результатом которой является длина указанной строки.

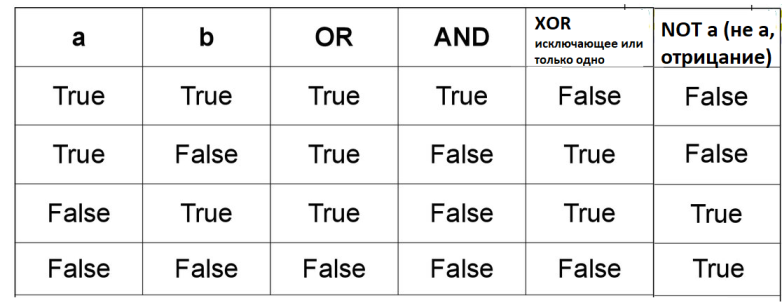
Поскольку символьные типы относятся к порядковым, для них предопределены такие функции, как Pred, Succ. Например, Pred('B') вернет символ 'A', a Succ('B') вернет 'С'.

К значениям символьных типов можно применять операции отношения: <, >, <=, >=, <=, <>, =. При сравнении символов сравниваются соответствующие им коды, и большим будет символ, имеющий больший код.

1. **Логический тип. Логические операторы и операции сравнения**

Значениями ***логического типа*** может быть одна из предварительно объявленных констант ***False*** или ***True.***  
**Ord (False) =0; Ord (True) = 1; False < True ; Succ (False) = True; Pred (True) = False. High(Boolean)=true, Low(Boolean)=false.** Поскольку логический тип относится к порядковым типам, его можно использовать в операторе счетного типа, например :  
**var** i:Boolean;  
**begin**  
**for** i:= False **to** True **do …**

****



Допустимые операции:   
- присваивание;  
- сравнение: <, >, >=, <=, <>, =;  
- логические операции: NOT, OR, AND, XOR

NOT - отрицание(замена значения на противоположное)

OR - "или"(возвращает истину, если хотя бы одно из условий истина)

AND - "И"(возвращает истину, если каждое из условий истина)

XOR - "не-и"(возвращает истину, если одно условие истина, а другое ложь. Т.е. одно условие обязательно истина, другое - обязательно ложь, тогда операция возвращает истину.)

Логический тип

Переменные логического типа могут принимать значения: False(ложь) или True (истина).

Примеры описания:

VAR

b1, b2, b3, b4 : boolean;

**Примеры присвоения значения:**

b1 := True; b2 := False; b3 := not b1;

1. **Порядковые типы. Перечисляемый тип**

Перечисление, или перечисляемый тип, задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном круглыми скобками, например :

Type

TSound = (‘ click, clack, clock’);

Описание переменных : var snd:TSound;

Особые операции:

ord(snd) – возвращает номер значения по порядку начиная с нуля(нумерацию можно начинать с единицы, если в типе указать : Type TSound = (‘click’=1,’ clack, clock’).

В общем виде объявление перечисляемого типа выглядит так:

Тип =( Значение1, Значение2, ... Значение i)

где: тип — имя перечисляемого типа данных; значение i — символьная константа, определяющая одно из значений, которое может принимать переменная типа Тип.

Примеры:

TDayOfWeek = (MON,TUE,WED,THU,FRI,SAT,SUN);

TColor = (Red,Yellow,Green);

Примечание

Согласно принятому в Delphi соглашению, имена типов должны начинаться с буквы Т (от слова Туре — тип).

После объявления типа можно объявить переменную, относящуюся к этому типу, например:

type

TDayOfWeek = (MON,TUE,WED,THU, FRI,SAT,SUN) ;

var

ThisDay, LastDay: TDayOfWeek;

Помимо указания значений, которые может принимать переменная, описание типа задает, как значения соотносятся друг с другом. Считается, что самый левый элемент списка значений является минимальным, а самый правый — максимальным. Для элементов типа DayOfWeek справедливо:

MON < TUE < WED < THU < FRI < SAT < SUN

Свойство упорядоченности элементов перечисляемого типа позволяет использовать переменные перечисляемого типа в управляющих инструкциях, например, так:

if (Day = SAT) OR (Day = SUN) then

begin

{ действия, если день — суббота или воскресенье }

end;

Приведенную инструкцию можно записать и так:

if Day > FRI then begin

{ действия, если день — суббота или воскресенье }

end;

1. **Тип массив(статический) : описание, ввод, вывод. Форматный вывод.**

Это пользовательский тип.

Отличительная особенность массивов заключается в том, что все их компоненты – суть данные одного типа. Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием его порядкового номера.

Описание типа массива : <имя типа> = array [ <сп. инд. типов> ] of <тип> ;

<имя типа> – правильный идентификатор; array,of – зарезервированные слова(массив, из); <сп. инд. типов> – список из одного или нескольких индексных типов, разделенных запятыми; <тип> – любой тип Паскаля.

В качестве индексных типов в Паскале можно использовать любые порядковые типы, кроме LongInt и типов-диапазонов с базовым типом LongInt.

Обычно в качестве индексного типа используется тип-диапазон, в котором задаются границы индексов. Так как тип <тип> за словом of,– это любой тип Паскаля, он может быть, в частности, и другим массивом, например :

type mat = array [0..5,-1..2,Char] of Byte.

Ввод и вывод begin a.b : =100; writeln(a.b); End.

|  |  |
| --- | --- |
| **Статический массив** | |
| Пользовательский тип  Type  TMas1 = array [1..20] of real;  TMas2 = array [-10..0] of byte;  TMas3 = array ['a'..'z'] of integer;  TMas4 = array [char] of byte;  TMas5 = array [boolean] of integer;  TSummer = (June, July, August);  TMas6 = array [TSummer] of real;  TMas7 = array [0..10] of char;  Три аналогичных двухмерных массивов (матрицы):  Type  TMatr = array[1..5] of array[1..20] of real;  Или  TMatrix = array[1..5, 1..20] of real;  Или  TMatrica = array [1..5] of TMas1;  Индексы задаются любым порядковым типом, чей диапазон не превышает 2ГБайта.  Размер структуры также не должен превышать 2ГБайта. | Особые операции:  С элементами – операции в соответствии с их типом  Доступ к элементам по индексам:  A[1], B['a'], C[June]  D[1,1], D[1][1], G[5,20], G[5][20]  Диапазоны индексов:  Low(A) 🡪 1; High(A) 🡪 20  Low(B) 🡪 'a'; High(B) 🡪 'z'  Low(C) 🡪 June; High(C) 🡪 August  Low(D) 🡪 1; High(D) 🡪 5  Low(D[5]) 🡪 1; High(D[5]) 🡪 20  Длина массива:  Length(A) 🡪 20  Length(B) 🡪 26  Length(C) 🡪 3  Length(D) 🡪 5  Length(D[Low(D)]) 🡪 20 |
| Описание переменных  Var  A, E: Tmas1; B: Tmas3; C: Tmas6;  D: TMatrix; G: TMatrica;  S: TMas7; | Вывод значений – поэлементный  Аналогично вводу, например:  Writeln('Массив A: ');  For i:=1 to 5 do Write(A[i]:5:2, ' ');  WriteLn; |
| Ввод значений – поэлементный  (исключение – массив из символов, с индексацией с нуля, например, *S* типа *TMas7*, т.к. он совместим с типом короткой строки *ShortString*)  Доп. переменные  Var i,j: byte; k:char; s: TSummer;  Writeln('Введите массив A');  For i:=1 to 20 do Read(a[i]);  Readln;  Writeln('Введите массив B');  For k:='a' to 'z' do Read(B[k]);  Readln;  Writeln('Введите массив C');  For s:= June to August do  Readln(c[s]);  Writeln('Введите матрицу D');  For i:=1 to 5 do  For j:=1 to 20 do  Read(d[i,j]); {или Read(d[i][j]);}  Readln; | Форматный вывод - это вывод с этими двоеточиями после переменной, нужен для вывод на экран понятной десятичной дроби, например, или для вывода с отступами. |
| Присвоение значений  A[1]:= 2.3; B['b']:=5; C[July]:=pi;  D[5, 20]:= -4.5e-4; D[5][20] := A[5];  Если тип одинаковый *по названию*, а не по описанию или структуре, то  E := A; // тип *TMas1*  E:=G[1]; G[1]:=G[5]; G[5]:=E; //тип *TMas1*  D[1]:=D[5];// безымянный тип |

Массивы содержат фиксированное число элементов одного типа, так называемого типа элемента. На приводимой ниже синтаксической диаграмме тип элемента следует за словом of.

В индексных типах, по одному для каждой размерности массива, указывается число элементов. Допустимыми индексными типами являются все порядковые типы, за исключением длинного целого и поддиапазонов длинного целого. Массив может быть проиндексирован по каждой размерности всеми значениями соответствующего индексного типа; число элементов поэтому равно числу значений в каждом индексном типе. Число размерностей не ограничено.

Приведем пример типа массив:

array[1..100] of Real

Если тип элемента в типе массив также является массивом, то результат можно рассматривать как массив массивов или как один многомерный массив. Например,

array[boolean] of array[1..100] of array[Size] of Real

интерпретируется компилятором точно так же, как массив:

array[boolean,1..10,Size] of Real

Ввод:

For i:=1 to 10 do readln(massiv[i]);

Вывод: тоже, но witeln

1. **Тип запись: описание, ввод, вывод. Оператор With. Запись с вариантами**

|  |  |  |
| --- | --- | --- |
| ТИП | МАССИВ | ЗАПИСЬ |
| Тип элементов | Однотипные элементы | Могут быть разных типов |
| Имя | Общее для всех элементов | Общее для всех элементов |
| Доступ/Обращение к элементам | По индексу элемента: Имя\_массива[индекс] | По имени поля: Имя\_записи.Имя\_поля |
| Память | Смежные ячейки | Смежные ячейки, но (по умолчанию) с выравниванием по 4 или 8 байт (неплотно - пример на 6- 7 страницах ) |

Запись – совокупность неоднородных элементов, хранящихся в соседних ячейках памяти, имеющих общее имя и различающихся по именам полей.

1. **(Запись –) Пользовательский тип**, а значит, как и для массива, (обычно) перед использованием надо определить тип в разделе **type**

Для использования переменной типа запись в качестве **параметра процедуры (функции)** – определение типа обязательно (как и для статического массива),

а также если надо описать переменные одного типа в разных местах программы, но чтобы работало

присваивание.

1. **Определение типа и описание переменных Type** //в разделе type

Trec = **record** // ключевое слово **record**

a, b: integer; // поля с одинаковым типом через «,» можно c: real; // могут быть разные типы a,b,c – имена **полей end**; // описание записи заканчивается словом **end**

**Var** // теперь опишем переменную Rec1: Trec;

Rec2: Trec;

Rec2\_2, Rec2\_3: **record** // похожа на Trec, но не она по имени

a, b: integer; c: real;

**end**;

Rec3: **record** // в некоторых случаях можно и без определения типа N: integer;

A: array [1..20] of real; // элементом записи может быть и массив

// и другая запись , и строка

**End**;

Rec4: **record**

Data: **record** // полем записи может быть запись d,m,g: integer;

**End**; N: string;

**End**;

## Ввод значений

Значение каждого поля вводится отдельно (как и в массиве)

Writeln('Введите Rec3:'); Write('n=?’); readln(**Rec3**.n); Write('Введите массив A:'); For i:=1 to n do

Readln(**Rec3**.A[i]);

Writeln('Введите Rec1:'); Write('a=?'); readln(**Rec1**.a);

Write('b=?'); readln(**Rec1**.b);

Write('c=?'); readln(**Rec1**.c);

Writeln('Введите Rec4:'); Write('День=?'); readln(**Rec4.Data**.d); Write('Месяц=?'); readln(**Rec4.Data**.m); Write('Год=?'); readln(**Rec4.Data**.g); Write('Имя=?'); readln(**Rec4**.N);

## чтобы не упоминать каждый раз имя записи

можно использовать оператор **WITH,**

**Например**, изменим код, указанный слева для

*Rec*1 и *Rec*4

Writeln('Введите Rec1:');

## With Rec1 do

begin

Write('a=?'); readln(**a**);

Write('b=?'); readln(**b**);

Write('c=?'); readln(**c**);

End;

Writeln('Введите Rec4:');

With **Rec4.Data** do begin

Write('День=?'); readln(**d**);

Write('Месяц=?'); readln(**m**);

Write('Год=?'); readln(**g**); End;

Write('Имя=?'); readln(**Rec4**.N);

## Присваивание значений

Каждому полю отдельно

Rec4.Data.d:=31;

Или запись целиком (только при **точном совпадении типа по его имени**, а не по размеру или структуре!)

Rec2:=Rec1; {тип TRec} Rec2\_2:=Rec2\_3; {Неименованный тип}

Rec2:= Rec2\_2; // НЕЛЬЗЯ! Разные типы, хоть и близнецы-братья

## Особые операции

Использование конструкции **With** Имя\_записи **do**

С полями операции производятся соответственно типу каждого поля

## Вывод значений

как и ввод – *каждое поле отдельно* ((типизир. файлы – будут исключением – см 2 семестр)) Writeln(‘Rec4:’);

With **Rec4.Data** do begin

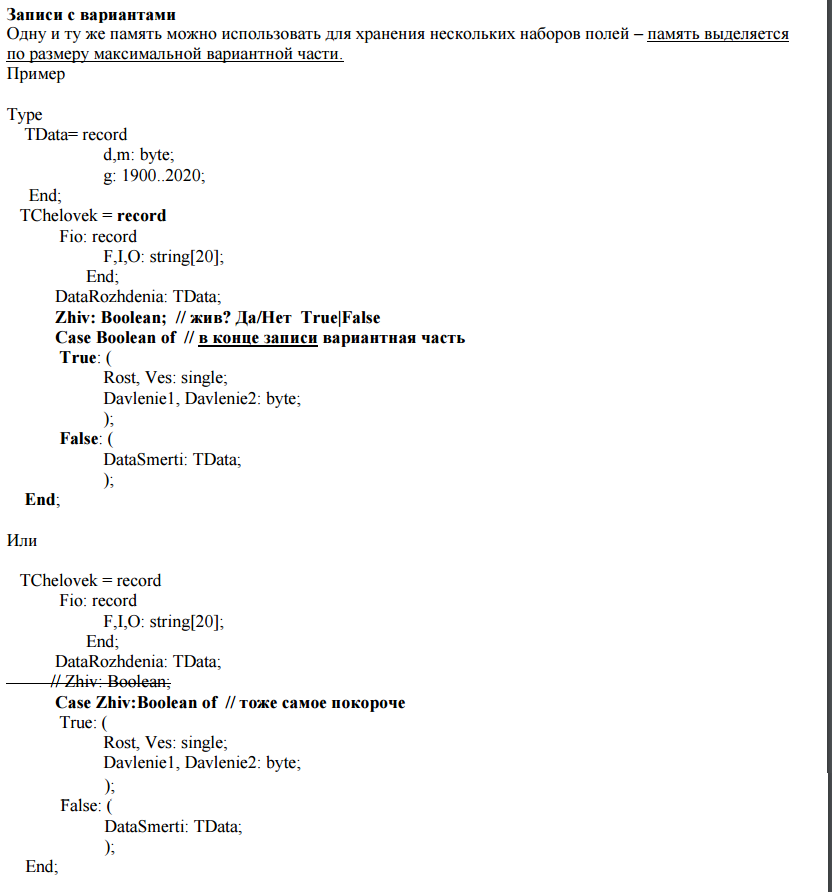
Write('День= ', **d**);

Write('Месяц= ', **m**);

Write('Год= ', **g**);

End;

Write(‘Имя=?’); readln(**Rec4.N**);

****

1. **Тип множество: описание, ввод, вывод, операции над множествами**

|  |  |  |
| --- | --- | --- |
| ТИП | МАССИВ | МНОЖЕСТВО |
| Тип элементов | Однотипные элементы | Однотипные неповторяющиеся элементы простых порядковых типов с диапазонами значений (Ord) не более чем от 0 до 255 (byte, char, логический, перечисляемый, диапазон) |
| Имя | Общее для всех элементов | Общее для всех элементов |
| Доступ/Обращение к элементам | По индексу элемента: Имя\_массива[индекс] | Нет (есть только проверка его наличия) |
| Память | Смежные ячейки | Смежные ячейки памяти – один бит на 1 элемент + округление в большую сторону до целых байтов (по 8 бит) Но не более 256 элементов! (32 байта) Наибольшая эффективность при 4 байтах (32 значения-бита) |

Таким образом, Множество – совокупность не более чем 256 неповторяющихся однородных простых элементов порядковых типов, хранящихся в смежных БИТАХ памяти, имеющих общее имя.

Пример объявлений множеств:

type

colors = (red,green,blue);

smallnumbers = 0..10;

*{определяем множества из наших типов}*

colorset = set of colors;

numberset = set of smallnumbers;

*{можно и не задавать тип отдельно}*

anothernumberset = set of 0..20;

*{объявляем переменные типа множеств}*

var

nset1,nset2,nset3:numberset;

cset:colorset;

begin

nset1 := [0,2,4,6,8,10]; *{задаем в виде конструктора множества}*

cset := [red,blue]; *{простым перечислением элементов}*

nset2 := [1,3,9,7,5]; *{порядок перечисления неважен}*

nset3 := []; *{пустое множество}*

nset1 := [0..5]; {возможно задавать элементы диапазоном}

nset1 := nset2; {одного типа numberset}

Ввод значений Значение каждого элемента добавляется в множество отдельно

set1:=[]; // пустое множество

Writeln(‘Введите кол-во элементов:’);

Write(‘n=?’); readln(n);

For i:=1 to n do

begin Write(‘элемент=?’);

readln(k); Include(set1, k); // или c помощью объединения множеств set1:=set1+[k];

end;

при несовпадении диапазона значения возможны ошибки, как и при введении отрицательных значений для переменной типа byte

Особые операции

1)Включения элемента во множество, объединение множеств

Set1:=set1+[5];

Include(set1, 5);

Set1:=set1+[2..7, 200];

Результатом сложения [1..4]+[4..7] будет [1..7]

2)Исключения элемента(ов) из множества, вычитание множеств

Set1:=set1-[5];

Exclude(set1, 5);

Set1:=set1 – [2..7, 200];

Результатом вычитания [1..4]-[4..7] будет [1..3]

3)Пересечение множеств

Set1:=set1\*[5,6];

Set2:=[1,3,6..12]\*[4..8];

Результатом пересечения [1..4]\*[4..7] будет [4]

4)Проверка наличия элемента в множестве IN

If ‘z’ in set4 then

If not(‘z’ in set4) then

If [‘x’..‘z’] \* set4 = [‘x’..‘z’] then

5)Операции сравнения

<=, =>, > вложено/содержит подмножество

=, <> // одинаковые, разные

Вывод значений через проверку наличия

Writeln(‘set1:’);

For i:=0 to 255 do

If i in set1 then

Write(i,’ ’);

Writeln(‘set4:’);

For ch:=’a’ to ‘z’ do

If ch in set4 then

Write(ch,’ ’);

Над множествами определены следующие операции :  
**а**) пересечение множеств (\*) – результат содержит элементы, общие для обоих множеств.  
**б**) объединение множеств (+) – результат содержит элементы первого множества, дополненные элементами из второго множества.  
**в**) разность множеств (-) – результат содержит элементы первого множества, которые не принадлежат второму.  
**г**) проверка эквивалентности (=) – возвращает True, если оба множества эквивалентны.  
**д**) проверка неэквивалентности (<>) – возвращает True, если оба множества неэквивалентны.  
**е**) проверка вхождения первого множества во второе (<=) – возвращает *True*, если первое множество включено во второе.  
**ж**) проверка вхождения второго множества в первое (>=) – возвращает *True*, если второе множество включено в первое.  
**з**)проверка принадлежности (**in**) – в этой бинарной операции первый элемент – выражение, а второй – множество одного и того же типа; возвращает True если выражение имеет значение, принадлежащее множеству, (см. предыдущий пример) 1 **in** s1 возвращает *True*, а 2\*2 **in** s2 возвращает *False*.  
**и**) ***Include*** – включает новый элемент в множество ( Include (S, I), здесь S- множество, I – элемент)  
***Exclude*** – исключает элемент из множества ( Exclude (S, I)).

1. **Текстовый файл: описание, основные операции. Использование параметров программы для передачи программе имен файлов**

Описание:  
var T:file;

T2:textfile;

Лучше использовать второй вариант. Основные операции:  
assign(файловая переменная, путь к файлу); - привязывает файловую переменную в программе к конкретному файлу на диске. Присвоение файловой переменной.

reset(файловая переменная); - открытие файла для чтения данных с него.

append(файловая переменная); - открытие файла для записи данных в него. При использовании этой операции "курсор-каретка" не смещается в начало. Т.е. если вы считали с файла 3 переменные, а затем применили append, то запись в файл начнётся с 4-ого элемента.

rewrite(файловая переменная); - создаёт файл с указанным именем по указанному пути (если такой файл существует, то стирает все данные в нём и ставит "курсор-каретку" в начало файла) и открывает его для записи.

close(файловая переменная); - завершение работы с файлом, его "закрытие".

Параметр-имя файла можно передать двумя способами: либо через параметры компилятора (в самом Дельфи: меню - Run - Parameters, там в строке Parameters и указываете имена.). Либо через параметры ярлыка, путём приписывания к его адресу в свойствах адресов параметров-имён. Создании Батника - вообще говоря, это множественное использования метода ярлыка.

При этом удобно исходные данные для всех тестов хранить в файлах на диске: *один тест – один текстовый файл*. Чтобы не менять программный код, меняя названия файлов для каждого теста можно использовать возможность передачи параметров программе при ее запуске. Результаты запуска для каждого теста также удобно хранить в отдельных файлах.

Пусть программа тестируется на двух тестах.

Пусть имя входного файла – 1-й параметр, имя выходного файла – 2-й параметр. Пусть prog.dpr – файл с программным кодом головного модуля проекта,

prog.exe – соответствующий .exe-файл,

prog\_dat1.txt – файл входных данных для первого теста, prog\_dat2.txt – файл входных данных для второго теста, prog\_res1.txt – файл выходных данных для первого теста, prog\_res2.txt – файл выходных данных для второго теста.

Создать текстовые файлы с исходными данными можно в Блокноте или в среде *Delphi* (меню *File**New**Other**Other*\_*Files**Text*). Файлы с результатами создадутся автоматически при открытии с помощью *ReWrite*, либо надо создать их самим при открытии для дозаписи с помощью *Append*. Просмотреть результаты можно опять же с помощью простейшего текстового редактора

«Блокнот» или прямо из среды разработки программ.

Изменения, которые надо произвести в тексте программы – минимальны: конкретные имена файлов заменяются на обращения к функции *ParamStr*:

assignFile(dat, ParamStr(1)); assignFile(res, ParamStr(2));

*ParamStr* – стандартная функция для работы с параметрами в *Delphi*, она возвращает параметр с заданным номером. Ее синтаксис:

function **ParamStr**(<№ параметра>: word): string;

Все параметры трактуются как отдельные строки (*string*). Параметры пользователя нумеруются, начиная с *единицы*. В *нулевом* параметре *ParamStr*(0) ОС передает программе полное имя запускаемого приложения (например, *D*:\*Гречкина*\*Project1*.*exe*). Этот (нулевой) параметр не входит в общее число параметров, которое можно узнать с помощью функции *ParamCount*:

function **ParamCount**: word;

Для вывода значений всех параметров можно использовать код:

Program Params;

{$AppType CONSOLE} Var i:byte;

Begin

For i:=0 to **ParamCount** do writeln('Param',i:2,' ', **ParamStr**(i)); Write('Press ENTER'); ReadLN

End.

**Для запуска в среде *Delphi*** перед запуском программы надо указать через пробел параметры в секции *Parameters* (меню *Run**Parameters*) Затем запустить программу.

Текстовый файл – совокупность строк (последовательностей символов) *переменной* длины.  
Описание файловой переменной : <ф.п.>: TextFile; или просто Text.  
Первоначально любой файл данных создается как текстовый. Набранные на клавиатуре данные представляют собой ***стандартный****входной файл*. Содержимое дисплея при просмотре любого файла – ***стандартный****выходной файл.* Эти файлы используются при задании и просмотре данных. Для хранения данных последние записываются в файл на внешнем запоминающем устройстве (диске).

* Основные операторы для работы с текстовыми файлами:  
  ***assignFile( <ф.п.>,’<имя файла>’)***– связывает файловую переменную с файлом;  
  ***rewrite( <ф.п.> )***– создание и открытие нового файла для *записи*;  
  ***reset (<ф.п.> )***– открытие существующего текстового файла (файла, связанного с файловой переменной <ф.п.>) для *чтения*;  
  ***append( <ф.п.> )***– открытие существующего текстового файла (файла, связанного с файловой переменной <ф.п.>) для ***до****записи* в конец;  
  ***closeFile( <ф.п.>)***– закрытие открытого файла.

Операторы ввода-вывода:  
***read( <ф.п.>,<список ввода***>***)*** – чтение данных; элемент списка ввода для текстового файла – число или символ или строка string;  
***write( <ф.п.>,<список вывода***>) - запись данных согласно списку вывода; элемент списка вывода для текстового файла – число или символ или строка string.  
***readln( <ф.п.>,<список ввода***>) - чтение данных согласно списку ввода и переход на следующую строку; если в строке данных остались данные, не вошедшие в список ввода, они игнорируются  
***writeln( <ф.п.>,<список вывода***>) - запись данных в файл согласно списку вывода с добавлением в конце выведенной строки маркера конца строки (переход на следующую строку).  
Параметры :  
assignFile(dat, ParamStr(1));  
assignFile(res, ParamStr(2));  
*ParamStr* – стандартная функция для работы с параметрами в *Delphi*, она возвращает параметр с заданным номером. Ее синтаксис:  
*function* ***ParamStr***(<№ параметра>: *word*): *string*;

Все параметры трактуются как отдельные строки (*string*). Параметры пользователя нумеруются, начиная с *единицы*. В *нулевом* параметре *ParamStr*(0) ОС передает программе полное имя запускаемого приложения (например, *D*:\*Гречкина*\*Project1*.*exe*). Этот (нулевой) параметр не входит в общее число параметров, которое можно узнать с помощью функции *ParamCount*: *function* ***ParamCount***: *word*.

Файл – это совокупность *логически* объединённых данных.

Преимущества текстовых файлов: простота создания; наглядность – возможен просмотр средствами текстовых редакторов; универсальность – можно обойтись только этим видом файлов. С другой стороны, при использовании текстовых файлов доступ к данным осуществляет медленнее, чем при использовании других типов файлов. Текстовые файлы являются файлами *последовательного* доступа, т.е. читать из него данные и записывать в него данные можно только последовательно.

Входные файлы создаются любым подходящим способом вне программы и до её запуска. Выходные файлы формируются программой.

**Использование параметров программы**

Если указать в тексте программы имена файлов для ввода и вывода, то каждый раз придётся использовать одни и те же файлы и менять/копировать их содержимое или менять имена файлов в тексте программы. Оба варианта не удобны, и для того, чтобы избежать этого, можно использовать так называемые *параметры программы*. Эти параметры передаются в программу при запуске.

Необходимо выбрать пункт меню *Программа – Параметры командной строки...* и в появившемся диалоговом окне в поле *Параметры* ввести необходимые параметры (без имени исполняемого файла).

Параметры программы разделяются пробелами или табуляциями. Для того чтобы несколько слов рассматривались как один параметр, параметр должен быть заключён в двойные кавычки.

Для работы с параметрами программы в языке Паскаль существуют две стандартные функции – *ParamCount* и *ParamStr*.

Функция *ParamCount* возвращает количество параметров, переданных в программу. Функция *ParamStr* возвращает параметр с указанным индексом. Пользовательские параметры нумеруются с 1. Если индекс больше числа параметров, возвращается пустая строка.

1. **Назначение и отличия процедур общего вида и функций**

В языке Паскаль имеется два вида подпрограмм - процедуры и функции. Процедуры и функции помещаются в раздел описаний программы. Для обмена информацией между процедурами и функциями и другими блоками программы существует механизм входных и выходных параметров. Входными параметрами называют величины, передающиеся из вызывающего блока в подпрограмму (исходные данные для подпрограммы), а выходными - передающиеся из подрограммы в вызывающий блок (результаты работы подпрограммы).

Одна и та же подпрограмма может вызываться неоднократно, выполняя одни и те же действия с разными наборами входных данных. Параметры, использующиеся при записи текста подпрограммы в разделе описаний, называют формальными, а те, что используются при ее вызове - фактическими.

Функции - это процедуры особого вида, результатом работы которых является некоторое значение, подобное переменной, как и процедура, может иметь список параметров, следующих за именем функции в круглых скобках. Но если имя процедуры используется только для ее вызова, то с именем функции связывается ее значение.

Точно так же, как константа или переменная, вызов функции может использоваться в списке параметров оператора write, что для процедур невозможно.

Ещё одно отличие процедур от функций заключается в необходимости присваивания результата вычисления в теле функции переменной, имя которой совпадает с именем функции. Если такое присваивание в теле функции не выполнено, то функция не возвратит никакого результата (точнее, возвратит произвольный результат).

Оформлять подпрограмму как функцию целесообразно только в том случае, если она имеет один выходной параметр. Если же последовательность команд ориентирована на выполнение некоторого действия (выдача информации на экран и т.п.), целесообразно оформлять ее как процедуру.

Для упрощения разработки программ в Pascal-е можно использовать подпрограммы - процедуры и функции. Они представляют собой инструмент, с помощью которого любая программа

может быть разбита на ряд в известной степени независимых друг от друга

частей.

**Функции** - это процедуры особого характера, результатом работы которых является некоторое значение, подобное переменной.  
Функция, как и процедура, может иметь список параметров, следующих за именем функции в круглых скобках. Но если имя процедуры используется только для ее вызова, то с именем функции связывается ее значение. На примере сложения двух целых чисел проиллюстрируем возможности Турбо Паскаля 7.0 по оформлению программ при помощи процедур и функций, а также рассмотрим различия между этими двумя подходами.

* ***Назначение***. Подпрограммы (процедуры и функции) представляет собой инструмент, с помощью которого любая программа может быть разбита на ряд в известной степени независимых друг от друга частей. Такое разбиение необходимо по двум причинам :  
  1)Средство экономии памяти.  
  2)Применение методики нисходящего проектирования, благодаря которой достигаются достаточно простые алгоритмы, которые можно легко запрограммировать.  
  ***Отличие***: Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем. *Отличие* процедуры от функции заключается в том, что результатом исполнения операторов, образующие тело функции, всегда является некоторое единственное значение или указатель, поэтому вызов функции, поэтому вызов функции можно использовать в соответствующих выражениях наряду с переменными и константами.

1. **Описание и вызов процедур. Отличия от функций**

Формат описания процедуры имеет вид:

procedure *имя процедуры* (*формальные параметры*);

*раздел описаний процедуры*

begin

*исполняемая часть процедуры*

end;

Вызов:

…

*имя процедуры* (*формальные параметры*);

…

* входная, если ее значение должно быть известно до начала поиска среднего арифметического;
* промежуточная, если ей присваивается значение в этом коде;
* выходная, если ее значение было изменено/найдено здесь и должно быть передано вызывающей программе.

Теперь

1. переменные, используемые только для промежуточных вычислений, описываем как ***локальные*** перед блоком оператором процедуры в разделе описания переменных *var*;
2. переменные, значения которых являются только входными описываем как

***параметры-константы*** (***const***) или ***параметры-значения***;

1. переменные, значения которых являются входными и промежуточными описываем как ***параметры-значения***;
2. переменные, значения которых являются входными и промежуточными и выходными описываем как ***параметры-переменные* (*var*)**;
3. переменные, значения которых не являются входными, но являются промежуточными и выходными описываем как ***параметры-переменные* (*out*** или ***var*)**.

|  |  |  |  |
| --- | --- | --- | --- |
| **вх** | **пром** | **вых** | **Описывается как** |
| **+** | **–** | **–** | параметр-константа (с ключевым словом *const*) или как параметр-значение или  как параметр-переменная (с ключевым словом *var*) |
| **+** | **+** | **–** | параметр-значение (по умолчанию, без ключ.слова) |
| **+** | **+** | **+** | параметр-переменная (с ключевым словом *var*) |
| **–** | **+** | **+** | параметр-переменная (с ключевым словом *out*) или  параметр-переменная (с ключевым словом *var*) |
| **–** | **+** | **–** | локальная переменная, не является параметром |

Таким образом, **параметрами** становятся только входные и/или выходные переменные, остальные надо описывать как **локальные переменные**. **Все переменные**, упоминающиеся в коде процедуры, должны быть описаны либо как локальные переменные, либо как ее параметры. **Следуя этим правилам, вы создадите процедуры достаточно автономные и легко переносимые в ближайшем будущем в отдельные файлы-модули.**

**Отличие от функций: сама процедура может быть только как оператор, а не переменная)**

1. Функция возвращает значение, процедура — нет.
2. Заголовок функции обязательно содержит тип результата.
3. В теле функции обязательно наличие оператора, присваивающего значение имени функции.
4. Вызов процедуры требует отдельного оператора, вызов функции возможен в составе выражений.
5. Имя процедуры необходимо для вызова, имя функции — для присвоения значения.
6. На схемах алгоритмов вызов процедуры изображается в отдельном блоке, вызов функции — в блоке процесса или вывода.

Procedure FindSA( \_параметры ); Var локальные переменные

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | вх | пром | вых |  |
| S | - | + | - | лок |
| K | - | + | - | лок |
| i | - | + | - | лок |
| N | + | - | - | const |
| 1;j | - | + | - | лок |
| A | + | - | - | const |
| SA | - | + | + | out |

Begin

S:=0; K:=0;

for i:=1 to N do for j:=1 to N do

if A[i,j]>0 then begin

S:=S+A[i,j]; K:=K+

end; SA:=S/K;

End;

Выпишем в столбик **все переменные**, каждой отмечаем, что она:

• входная, если ее значение должно быть известно до начала поиска среднего арифметического; • промежуточная, если ей присваивается значение в этом коде; • выходная, если ее значение было изменено/найдено здесь и должно быть передано вызывающей программе. Теперь 1. переменные, используемые только для промежуточных вычислений, описываем как локальные перед блоком оператором процедуры в разделе описания переменных var; 2. переменные, значения которых являются только входными описываем как параметры-константы (const) или параметры-значения; 3. переменные, значения которых являются входными и промежуточными описываем как параметры-значения; 4. переменные, значения которых являются входными и промежуточными и выходными описываем как параметры-переменные (var); 5. переменные, значения которых не являются входными, но являются промежуточными и выходными описываем как параметры-переменные (out или var).

Для данной процедуры потребовалось создать новый тип для пользовательского типа массив: Const Nmax = 10; Type Matr = array [1..Nmax, 1..Nmax] of real;

Заголовок: Procedure FindSA( N: byte; var A: Matr; var SA: real);

1. **Описание и вызов функций**

Формат описания функции:

function *имя функции* (*формальные параметры*):*тип результата*;

*раздел описаний функции*

begin

*исполняемая часть функции*

end;

Вызов:

…

*Имя переменной := имя функции* (*формальные параметры*);

…

ИЛИ

…

*имя процедуры* (*имя функции* (*формальные параметры*));

**(+ отличие от процедур: сама процедура может быть только переменная, а не оператор)**

Из процедуры:

Procedure **F**(<список формальных параметров>); Begin

<тело процедуры>

End;

можно сделать функцию:

Function **F**(<список формальных параметров>): Boolean; Begin

**F** := **True**;

Try

<тело процедуры> Except

**F** := **False**

End;

End;

**Function FindKol**( const N: byte; var A: Matr): **byte;**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | вх | пром | вых |  |
| Kol | - | + | + | лок/ Result |
| Flag | - | + | - | лок |
| i | - | + | - | лок |
| N | + | - | - | const |
| A | + | - | - | const/ var |

### Var i, Kol : byte; Flag: boolean; Begin

Kol:=0;

For i:=1 to N do

begin

**Flag := Check\_i( N,i,A);**

if Flag then inc(Kol);

end;

**FindKol := Kol;**

End;

Вызывающая программа с функциями: Var

N: Byte; A: Matr; Kol: byte;

SA, Max: real;

Begin

VvodVyvodNA( N, A); **Kol := FindKol( N, A);** if Kol > N/2 then

begin

**SA := FindSA( N, A);**

write('Kol> N/2 and '); writeln('Sa=', SA:7:3);

end;

writeln('Press ENTER'); readln;

End.

1. **Классы формальных параметров: параметры-константы, параметры-значения и параметры переменные. Ключевые слова const, var, out при описании параметров**

Параметры-значения

Формальный параметр-значение обрабатывается, как локальная по отношению к процедуре или функции переменная, за исключением того, что он получает свое начальное значение из соответствующего фактического параметра при активизации процедуры или функции. Изменения, которые претерпевает формальный параметр-значение, не влияют на значение фактического параметра.

Соответствующее фактическое значение параметра-значения должно быть выражением и его значение не должно иметь файловый тип или какой-либо структурный тип, содержащий в себе файловый тип.

Фактический параметр должен иметь тип, совместимый по присваиванию с типом формального параметра-значения. Если параметр имеет строковый тип, то формальный параметр будет иметь атрибут размера, равный 255.

Параметры-константы

Формальные параметры-константы работают аналогично локальной переменной, доступной только по чтению, которая получает свое значение при активизации процедуры или функции от соответствующего фактического параметра. Присваивания формальному параметру-константе не допускаются. Формальный параметр-константа также не может передаваться в качестве фактического параметра другой процедуре или функции.

Параметр-константа, соответствующий фактическому параметру в операторе процедуры или функции, должен подчиняться тем же правилам, что и фактическое значение параметра.

В тех случаях, когда формальный параметр не изменяет при выполнении процедуры или функции своего значения, вместо параметра-значения следует использовать параметр-константу. Параметры-константы позволяют при реализации процедуры или функции защититься от случайных присваиваний формальному параметру. Кроме того, для параметров структурного и строкового типа компилятор при использовании вместо параметров-значений параметров-констант может генерировать более эффективный код.

Параметры-переменные

Параметр-переменная используется, когда значение должно передаваться из процедуры или функции вызывающей программе. Соответствующий фактический параметр в операторе вызова процедуры или функции должен быть ссылкой на переменную. При активизации процедуры или функции формальный параметр-переменная замещается фактической переменной, любые изменения в значении формального параметра-переменной отражаются на фактическом параметре.

Внутри процедуры или функции любая ссылка на формальный параметр-переменную приводит к доступу к самому фактическому параметру. Тип фактического параметра должен совпадать с типом формального параметра-переменной (вы можете обойти это ограничение с помощью нетипизированного параметра-переменной).

Примечание: Файловый тип может передаваться только, как параметр-переменная.

При описании параметров-значений в заголовках процедур/функций перед идентификаторами таких параметров дополнительные ключевые слова не ставятся.

При описании параметров-переменных, вы можете предопределить тип параметра с помощью Var, если сама переменная должна быть модифицирована подпрограммой. Это позволяет вызывающей программе передавать данные подпрограмме, которая будет изменена подпрограммой.

При описании параметров-констант, вы можете предписать определение параметра как Константу (Const), если значение никогда не модифицируется. Это незначительно улучшает работу, разъясняет операцию подпрограммы, и предотвращает случайные обновления значения.

Директива Out идентифицирует параметр функции или процедуры, являющийся переменной ссылкой только для выведения. Это эквивалентно Var за исключением того, что значение не может быть изменено подпрограммой.)

1. **Массивы и записи как формальные параметры процедур и функций**

Открытые параметры-массивы дают возможность передавать в подпрограмму массивы, размер которых не фиксируется в объявлении подпрограммы. Этот механизм позволяет создавать максимально гибкие и универсальные подпрограммы для обработки массивов произвольного размера.

Синтаксис описания открытых параметров-массивов:

ИмяПодпрограммы ( **VAR** P : **array of** ТипЭлементов );

Это единственное исключение из правила, согласно которому типы формальных параметров в заголовках подпрограмм могут задаваться только посредством идентификаторов. Подпрограмма, содержащая открытый массив в списке своих формальных параметров, может быть вызвана с массивом ЛЮБОГО размера (типы элементов массивов из формального и фактического параметров должны, естественно, совпадать). Заметим, что в качестве открытых массивов допускаются ТОЛЬКО одномерные массивы [1].

Реальные границы переданного в подпрограмму массива можно определить используя две функции **Low** и **High**. Эти функции предназначены для определения граничных значений типов и переменных. В данном случае функция Low будет всегда возвращать 0, а функция High - верхнее значение индекса равное (n-1), где n-число элементов в переданном массиве.

(Например:

PROGRAM TestOpenArray;  
VAR   
  A : array[-5..5] of Byte;

PROCEDURE P( var V : array of Byte );  
BEGIN  
  WriteLn('Нижний индекс V: ', Low(V));  
  WriteLn('Верхний индекс V: ', High(V));   
END;  
  
BEGIN  
  WriteLn('Нижний индекс A: ', Low(A));  
  WriteLn('Верхний индекс A: ', High(A));  
  P(A);  
  ReadLn;  
END.

Выполнение этой программы приведет к выводу на экран следующей информации:

Нижний индекс А: -5  
Верхний индекс А: 5  
Нижний индекс V: 0  
Верхний индекс V: 10

Пояснение: индексы массива в основной программе однозначно задаются, но подпрограмма может обходиться без этих конкретных индексов. На примере: индексы массива -5..5, а подпрограмма способна их воспринимать, как 0..10. Т.е. таким приёмом (используя Low и High) мы в подпрограмму количество самих элементов массива, как бы "перенумеровываем" их в том же порядке.)

1. **Имена процедур и функций как формальные параметры. Процедурный тип**

Процедурные типы

Процедурные типы - это нововведение фирмы Borland (в стандартном Паскале

таких типов нет). Основное назначение этих типов - дать программисту гибкие

средства передачи функций и процедур в качестве фактических параметров

обращения к другим процедурам и функциям.

Для объявления процедурного типа используется заголовок процедур, в

котором опускается ее имя, например:

type

Proc = Procedure (a, b, с : real; Var d : real);

Proc2 = Procedure (var a, b);

РгосЗ = Procedure;

В программе могут быть объявлены переменные процедурных типов, например,

так:

var

р1 : Proc;

ар : array [1..N] of Proc2;

Переменным процедурных типов допускается присваивать в качестве значений

имена соответствующих подпрограмм. После такого присваивания имя переменной

становится синонимом имени подпрограммы.

В отличие от стандартного Паскаля, в Турбо Паскале разрешается

использовать в передаваемой процедуре как параметры-значения, так и

параметры-переменные

(Program ProcType;  
Type  
  Func=Function(x, y : integer) : integer; {Описание процедурного типа}  
Procedure Tab(w,h : integer; Operation : Func);  
Var  
  x, y : integer;  
Begin  
  for y := 1 to w do  
    begin  
      for x := 1 to h do  
        write(Operation(x,y):5);  
      writeln;  
    end;  
End;  
  
Begin  
  Tab(10,10,Add);  
  writeln;  
  Tab(10,10,Mult);  
  writeln;  
  Tab(10,10,Funny);  
End.

Это пример использования процедурного типа, тут есть ещё 3 процедуры, которые не описали, это Add,Mult,Funny. На данном примере видно, что можно создать процедуру, которая будет выполнять определённое действие. А его можно "выбрать", путём добавления одной из трёх процедур - Add,Mult,Funny)

Записи – это один из структурированных типов в Delphi.

Раньше мы рассматривали другой структурированный тип – массивы

**Массив** – совокупность **однородных** элементов, хранящихся в **смежных** ячейках памяти, имеющих общее **имя** и различающихся по **индексам**.

|  |  |  |
| --- | --- | --- |
| Тип | Массив | Запись |
| Тип элементов | Однотипные элементы | Могут быть разных типов |
| Имя | Общее для всех элементов | Общее для всех элементов |
| Доступ/Обращение к элементам | По индексу элемента:  Имя\_массива[индекс] | По имени поля:  Имя\_записи.Имя\_поля |
| Память | Смежные ячейки | Смежные ячейки, но (по умолчанию) с выравниванием по 4 или 8 байт (неплотно - пример на 6-  7 страницах ) |

Таким образом,

**Запись** – совокупность **не**однородных элементов, хранящихся в **соседних** ячейках памяти, имеющих общее **имя** и различающихся по **именам полей**.

1. **Модули в Паскале: назначение, описание, использование. Обязательные и дополнительные разделы**

(блок/модуль/юнит можно представить как маленького "робота-помощника". Раздел INTERFACE - это инструкция к нему, т.е. "меню" из всех возможностей этого робота: с чем он может работать, какие операции может выполнить. А вот IMPLEMENTATION - это "внутренности" робота. Т.е. изменение данного раздела, это как будто вы вскрываете крышку и начинаете копаться в механизмах робота, или влезаете в его систему и начинаете переписывать код. Удобство модуля заключается в том, что, как и робота помощника, вы можете вызывать его только тогда, когда он нужен, и "не таскать за собой" множество подпрограмм и констант из модуля.)

**Модульное программирование** – это организация программы как совокупности небольших независимых блоков, называемых **модулями**, структура и поведение которых подчиняются определенным правилам.

Используя модули, вы можете разбивать программу на отдельные части и компилировать их отдельно. Программный модуль (unit) представляет собой набор констант, типов данных, переменных, процедур и функций, которые могут совместно использоваться несколькими программами. Каждый модуль аналогичен отдельной программе на Паскале: он может иметь основное тело, которое вызывается перед запуском вашей программы и осуществляет необходимую инициализацию.

Структура модуля аналогична структуре программы, однако есть

несколько существенных различий. Например, рассмотрим модуль:

unit <идентификатор>;

interface

uses <список модулей>; { Необязательный }

{ глобальные описания }

implementation

uses <список\_модулей>; { Необязательный }

{ локальные описания }

{ реализация процедур и функций }

begin

{ код инициализации }

end.

Заголовок модуля начинается зарезервированным словом unit, за которым следует имя модуля (идентификатор) точно так же, как и в случае имени программы. Следующим элементом в модуле является ключевое слово interface. Оно обозначает начало интерфейсной секции модуля - части, доступной всем другим модулям или программам, в которых он используется. Программный модуль может использовать другие модули, для этого они определяются в операторе uses. Оператор uses (если он имеет место) может содержаться в двух местах. Во-первых он может следовать сразу после ключевого слова interface. В этом случае любые константы и типы данных, описанные в интерфейсной секции этих модулей, могут использоваться в любом описании в интерфейсной части данного модуля. Во-вторых, он может следовать немедленно за ключевым словом implementation. В этом случае все описания из этих модулей могут использоваться только в секции реализации.

Для использования специального модуля или набора модулей необходимо в начале программы поместить оператор uses, после которого указать список имен тех модулей, которые использоваться. Имена их должны разделяться запятыми:

program MyProg;

uses thisUnit, thatUnit, theOtherUnit;

Когда компилятор встречает такой оператор uses, он прибавляет информацию из секции интерфейса каждого модуля к таблице идентификаторов и присоединяет машинный код, представленный в секции, к самой программе.

Программа на языке Pascal состоит из отдельных частей кода, называемых **модулями.** Модули объединяются друг с другом специальными файлами исходного кода с заголовками program, library или package. Каждый модуль сохраняется в своём собственном файле и компилируется отдельно, скомпилированные модули объединяются в приложение.

Использование модулей позволяет:

* Разделять большую программу на несколько частей, которые могут редактироваться отдельно.
* Создавать библиотеки, разделяемые несколькими приложениями.
* Передавать библиотеки другим разработчикам, оставляя исходный код недоступным.

Файл исходного кода приложения состоит из заголовка program, оператора uses и блока объявлений и исполняемых операторов. Код может быть организован в одном или несколько модулей, а блок основной программы – содержать только вызов одной или нескольких подпрограмм. Можно даже организовать код в модулях таким образом, что блок основной программы останется пустым.

**Структура модуля**

Модуль состоит из объявлений констант, типов, переменных и подпрограмм. Каждый модуль хранится в отдельном файле исходного кода с расширением pas.

unit Unit1;

interface

uses…

implementation

uses…

initialization

finalization

end.

Файл модуля состоит из:

* Заголовка.
* Раздела описания интерфейса.
* Раздела реализации.
* Необязательного раздела инициализации.
* Необязательного раздела завершения.

Заканчивается модуль ключевым словом end с точкой.

**Заголовок модуля**

* Заголовок модуля определяет имя модуля.
* Он состоит из ключевого слова unit, за которым следует идентификатор с точкой и запятой.
* Имя модуля должно совпадать с именем файла.
* Даже если файлы с исходным кодом находятся в разных папках, два модуля с одинаковым именем не могут существовать в проекте.

**Раздел описания интерфейса**

* Раздел описания интерфейса начинается с ключевого слова interface и заканчивается с началом раздела реализации.
* Раздел описания интерфейса содержит объявления констант, типов, переменных, процедур и функций, доступных клиенту, т.е. приложению или другому модулю, где будут использоваться элементы разрабатываемого модуля.
* В раздел описания интерфейса помещаются только заголовки процедур и функций.

**Раздел реализации**

* Раздел реализации начинается с ключевого слова implementation и заканчивается с началом раздела инициализации или, в случае отсутствия этого раздела, с концом модуля.
* Раздел реализации определяет процедуры и функции, заголовки которых помещены в раздел описания интерфейса.
* Процедуры и функции могут определятся в любом порядке.
* Списки параметров процедур и функции, объявленных в разделе описания интерфейса и определяемых в разделе реализации, могут быть опущены (хотя никакого смысла в этом нет).
* Если список параметров присутствует, он должен точно соответствовать объявлению, сделанному в разделе интерфейса.
* Кроме этого, в разделе реализации могут определяться константы, типы, переменные, процедуры и функции, которые будут доступны только в разрабатываемом модуле.
* Раздел реализации также может включать оператор uses, который должен следовать сразу за ключевым словом implementation. Константы, типы, переменные и подпрограммы, импортируемые из такого модуля доступны только в разделе реализации. Ссылаться на них в разделе описания интерфейса нельзя.

**Раздел инициализации**

* Начинается с ключевого слова initialization и заканчивается с началом раздела завершения или, в случае отсутствия этого раздела, с концом модуля.
* Раздел инициализации содержит операторы, которые выполняются при запуске программы.
* Если приложение включает несколько модулей, разделы инициализации этих модулей выполняются в порядке объявления модулей в операторе uses.

**Раздел завершения**

* Раздел завершения является необязательным и может использоваться, только если в модуле есть раздел инициализации.
* Раздел завершения начинается с ключевого слова finalization и продолжается до конца модуля.
* Раздел завершения содержит операторы, которые выполняются при завершении программы, но только если выполнение программы не было прервано процедурой Halt.
* Если приложение включает несколько модулей, разделы завершения этих модулей выполняются в порядке, обратном порядку выполнения разделов инициализации.

**Использование модуля**

После компиляции появляется файл с расширением dcu или pcu, который может быть передан другому разработчику. Для его использования необходимо указать имя модуля в операторе uses.

Оператор uses может использоваться:

* В файле проекта приложения или dll-библиотеки;
* В разделе описания интерфейса модуля;
* В разделе реализации модуля.
* В операторе uses, используемом в файле проекта приложения или dll-библиотеки, после имени модуля может следовать ключевое слово in и имя файла исходного кода с указанием полного или относительного пути в апострофах. Пример: uses UnitA in ‘UnitA.pas’;
* Этот механизм используется при необходимости указать файл исходного кода модуля.
* Поскольку имя модуля совпадает с именем файла, обычно нет необходимости в указании имени файла.
* Такая необходимость может появиться только когда размещение файла с исходным кодом неочевидно: Файл находится не в директории проекта, и эта директория не включена в список поиска компилятора. ИЛИ Разные директории в списке поиска компилятора содержат файлы с одинаковым именем.

Порядок, в котором модули указываются в операторе uses, определяют порядок их инициализации и влияют на способ, которым компилятор определяет местонахождение идентификаторов.

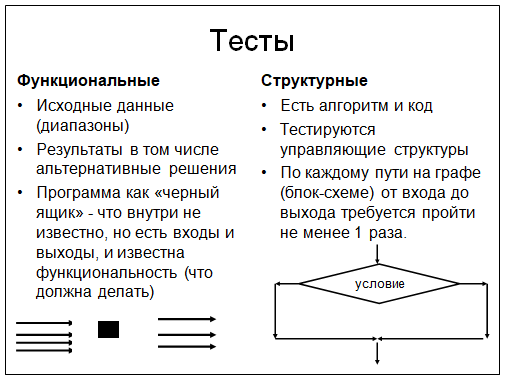
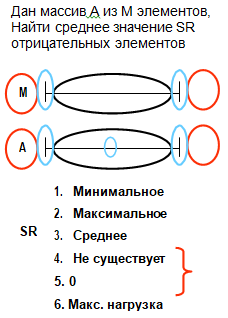
Недостаток реализации модулей в языке Паскаль состоит в том, что описание интерфейса объединено с реализацией, если разработчик хочет скрыть детали реализации, он должен предоставлять скомпилированный файл, но из такого файла нельзя понять, какие определения констант, типов, переменных, процедур и функций доступны при исп. модуля. Значит, кроме скомпилированного модуля разработчик должен предоставить отдельное описание интерфейса модуля.

1. **Составление функциональных и структурных тестов** на примере разработанной процедуры

**Функциональные тесты** составляются на уровне внешней спецификации, до решения задачи. Будущий алгоритм рассматривается как **«черный ящик»** – функция с неизвестной структурой, преобразующая входы в выходы. Суть функциональных тестов: каким бы способом ни решалась задача, при заданных входных значениях должны получиться соответствующие выходные значения.

**Структурные тесты** составляются для проверки логики работы уже готового конкретного алгоритма. Логика определяется последовательностью операций, их условным выполнением (ветвление) или повторением (циклы). Совокупность структурных тестов должна обеспечить проверку каждой из таких конструкций. Также к этим требованиям обычно добавляется необходимость выполнения каждого из операторов программы хотя бы один раз для хотя бы одного тестового набора.

*Функциональные тесты* проверяют правильность работы программы по принципу: «не знаю как сделано, но знаю, что должно быть в результате, и именно это и проверяю».



К функциональным тестам относятся:

·        *«тепличные»,* проверяющие программу при корректных, нормальных значениях исходных данных;

·        *«экстремальные»* («стресс–тесты»), находящиеся на границе области определения (например, наибольшая или наименьшая нагрузка системы по количеству или по времени), проверяющие поведение системы в экстремальных ситуациях, которые могут произойти и на которые программа должна корректно реагировать.

·        *«запредельные»* («тесты обезьяны»), выходящие за границы области определения (а возможно, и здравого смысла), проверяющие ситуации, бессмысленные с точки зрения постановки задачи, но которые могут произойти из-за ошибок пользователя (корректная реакция системы на запрещенный или неподдерживаемый ввод и т.п., так называемая «защита от дурака»)

*Структурные тесты* контролируют (тестируют) работу всех структурных частей программы (функций, процедур, модулей, основной программы) по всем возможным маршрутам (ветвям программы).

При структурном тестировании необходимо осуществлять контроль:

·        обращений к данным (т.е. проверять правильность инициализации переменных; а также размеры массивов и строк; отслеживать, не перепутаны ли строки со столбцами; соответствуют ли входных и выходных значений выбранным типам  данных; проверять правильность обращения к файлам и т.п.);

·        вычислений (порядок следования операторов и запись выражений; переполнение разрядной сетки или получение машинного нуля; соответствие результата заданной точности и т.п.);

·        передачи управления (завершение циклов, функций, программы);

·        межмодульных интерфейсов (списки формальных и фактических параметров; отсутствие побочных эффектов, когда подпрограмма изменяет аргументы, которые не должны меняться и т.п.).

Искусство тестирования сводится к разработке простого, полного и не избыточного набора тестов, а технология тестирования – к испытанию программы на всем наборе тестов, после внесения в нее каждого изменения.

1. **Нисходящее и восходящее тестирование программ**

**Восходящее тестирование.**

При восходящем подходе программа собирается и тестируется снизу вверх. Только модули самого нижнего уровня (модули, не вызывающие других модулей) тестируются независимо, автономно. После того как тестирование этих модулей завершено, вызов их должен быть так же надежен, как вызов встроенной функции языка или оператор присваивания. Затем тестируются модули, непосредственно вызывающие уже проверенные. Эти модули более высокого уровня тестируются не автономно, а вместе с уже проверенными модулями более низкого уровня. Процесс повторяется до тех пор, пока не будет достигнута вершина. Здесь завершаются и тестирование модулей, и тестирование сопряжений программы. Для каждого модуля необходимо написать небольшую ведущую программу.

**Нисходящее тестирование.**

Нисходящее тестирование не является полной противоположностью восходящему, но в первом приближении может рассматриваться как таковое. При нисходящем подходе программа собирается и тестируется сверху вниз. Изолированно тестируется только головной модуль. После того как тестирование этого модуля завершено, с ним соединяются один за другим, модули, непосредственно вызываемые им, и тестируется полученная комбинация. Процесс повторяется до тех пор, пока не будут собраны и проверены все модули. Для имитации функций недостающих модулей программируются модули-”заглушки”, которые моделируют функции отсутствующих модулей.

**Метод сэндвича.**

Тестирование методом сэндвича представляет собой компромисс между восходящим и нисходящим подходами. При использовании этого метода одновременно начинают восходящее и нисходящее тестирование, собирая программу как снизу так и сверху и встречаясь в конце концов где-то в середине. Точка встречи зависит от конкретной тестируемой программы и должна быть заранее определена при изучении ее структуры. Если разработчик может представить свою систему в виде уровня прикладных модулей, затем уровня модулей обработки запросов, затем уровня примитивных функций, то он может решить применять нисходящий метод на уровне прикладных модулей, а на остальных уровнях применить восходящий метод.

В нашем случае применим метод сэндвича, поскольку перед проектированием основной программы необходимо создать комплекс интерфейсных и вспомогательных функций, которые, естественно должны быть проверены до создания основного тела программы.

преимущества и недостатки

Восходящее - наиболее долгий метод, требующий создание "мини-программ" для проверки модулей или процедур, что означает дополнительную работу. Наиболее точный и надёжный метод тестирования;  
Нисходящий - тоже долгий, но занимает МЕНЬШЕ времени, чем восходящий, требующий создания "заглушек", для проверки основной программы и дальше "вниз" по лесенке структур. Тоже предусматривает дополнительную работу. Тоже очень точный и надёжный метод, но чуть уступает восходящему

"Сендвич" - самый быстрый метод, тк происходит параллельное тестирование "высших и низших слоёв" программы. Самый сложный метод, тк по результатам тестирования сложнее определить точное место, где возникает ошибка.

|  |  |
| --- | --- |
| **Нисходящее тестирование (и разработка)** | **Восходящее тестирование (и разработка)** |
| Достоинства:  + Сначала разрабатывается, кодируется и тестируется логика основной программы (с заглушками), и программа сразу готова к демонстрации. | Достоинства:  + Подпрограммы тестируются отдельно вне основного кода программы (с помощью драйверов) и без посредников. |
| Недостатки:  – Тестирование подпрограмм выполняется не  «напрямую» хотя и сразу «на своем месте», а значит, для ее тестирования, надо создать такой набор исходных данных для всей программы, чтобы до тестируемой подпрограммы они дошли в нужном виде. Программа постепенно растет и усложняется, усложняя и процесс создания  тестов. | Недостатки:   * Для тестирования приходится писать «лишний» код (драйвера), который не войдет в окончательный состав кода программы. * Программа не готова для демонстрации, тестирования и отладки как единое целое пока полностью не собрана, что выполняется в самом конце. |

Как и разработку, тестирование можно выполнять смешанным способом (метод «сэндвича»), что позволяет сочетать достоинства, убрав часть недостатков.

1. **Описание констант, переменных и пользовательских типов. Области видимости констант и переменных**

Пример объявления констант:

const

A=10;

B='Linux must die';

E:Real=2.71;

C=A+5;

D:array[0..4] of Integer=(5, A, 1, C, 123);

Константа – именованная область памяти, используемая для хранения фиксированного значения, которое невозможно изменить при выполнении программы.

Имена переменных и констант должны удовлетворять следующим правилам:

- первым символом всегда должна быть буква;

- в составе имени нельзя использовать символы: !, @, &, $, # , пробел;

- в качестве имени нельзя использовать ключевые   
(зарезервированные) слова, входящие в конструкции   
языка VBA;

- длина имени не может быть более 255 символов;

- имя нельзя повторять в пределах области его видимости   
(действия).

Переменные и константы, в зависимости от области действия, подразделяются на глобальные и локальные.

Если переменная или константа описана внутри процедуры, то она является локальной, то есть она определена и может использоваться только в пределах данной процедуры.

Если переменная или константа описана вне процедуры, то она будет глобальной. Такая переменная или константа может быть использована в нескольких процедурах.)

**Объявления констант, типов и переменных.**

Каждое имя (идентификатор) должно быть объявлено. Когда объявляются несколько констант, типов или переменных, соотв. слово может быть записано один раз.

**Объявления констант**

**Константа –** имя, значение которого не может быть изменено в программе. Значение придается один раз в момент объявления.

**const <идентификатор> = <константное выражение>;**

**Константное выражение –** выражение, значение которого компилятор может вычислить без выполнения программы.

### Типизированные константы занимают промежуточное положение между переменными и константами. Они получают значение при описании (как константы), но могут его менять в теле программы (как переменные).

Описываются типизированные константы в разделе констант:

**const <идентификатор> : <тип> = <значение> ;**

**Объявления типов**

Исп. для задания новых имен существующим типам и для создания пользовательских типов, таких как перечислимые типы, типы – диапазоны, массивы и записи.

**type <тип> = < описание типов>;**

Пример:

type mas = array [1..10] of integer;

height = real; (height = type real;)

**Объявления переменных**

**Переменная** – это имя, значение которой меняется в ходе выполнения программы (это имя участка оперативной памяти). Используя это имя можно читать данные из этого участка памяти и записывать туда данные. Поскольку переменная имеет тип, компилятор знает, как трактовать данные, хранящиеся в переменной, и какие действия над ними можно производить.

**var <список имен>: <тип>;**

Переменные могут быть инициализированы во время объявления с использованием следующего синтаксиса:

**var <имя> : <тип> = <константное выражение>;**

Имена переменных могут быть почти любым сочетанием английских букв и цифр (без пробелов). Нельзя чтобы имена переменных совпадали со словами, которые являются какими-либо командами самого языка программирования. Нельзя начинать имена переменных с цифры или специального символа. В Pascal прописные и строчные буквы в именах переменных не различаются. При описании переменных указывается не только их имя, но и тип. Тип переменных сообщает о том, сколько отвести под них памяти и что за данные там планируется сохранять.

**Инициализация переменной** – это присваивание переменной значения, которое в дальнейшем она, переменная, будет хранить, и которое можно будет использовать, путем обращения к имени переменной.

Инициализация может производиться **одновременно** с объявлением, а может – отдельно от него. Во втором случае (отдельно) инициализация переменной **всегда** должна происходить **после** объявления переменной.

Инициализация может быть выполнена *присвоением* переменной значения – для этого используется знак равенства (=) между именем переменной и присваиваемым ей значением.

Инициализация может быть выполнена за счет использования оператора *ввода данных с клавиатуры*.

1. **Описание констант структурированных типов: массивов, записей и множеств**

Типизированные константы типа "массив"

При описании константы типа "массив" компоненты каждой размерности массива заключаются в отдельные скобки и разделяются запятыми. Компоненты, расположенные в самых внутренних скобках соответствуют последней (самой правой) размерности массива.

Примеры констант типа "массив":

одномерный числовой массив

const

DigVector : array [1..7] of Real = (0.1, 3.25, 21.32, 55, 11.99, 78.1, 4.5);

двумерный числовой массив

const DigMatrix : array [1. .3,1..2] of Integer = ( (1,2), (3,4), (5,6) );

В результате элементы матрица DigMatrix имеет вид:

1 2

3 4

5 6

трехмерный числовой массив

const Dig3D : array [1..4,1..3,1..2] of Byte =

( ((1,2), (1,2), (1,2)),

((1,2), (1,2), (1,2)),

((1,2), (1,2), (1,2)),

((1,2), (1,2), (1,2)) );

одномерный массив символов

const CharVect1 : array [1..6] of Char = ('P' ,'A' , 'S' , 'C, 'A', 'L') ;

или более кратко CharVect2 : array [1..6] of Char = 'PASCAL';

Типизированные константы типа "множество"

Синтаксис описания констант типа "множество" определяется следующими правилами.

Каждый элемент константы типа "множество" может представлять собой либо отдельную константу соответствующего типа, либо интервал значений, состоящий из двух констант, разделенных символом . . (две точки).

Примеры констант типа "множество":

type

Digits = set of 0..9;

CharDig = set of '0'..'9';

const

DigSetl : Digits = [0, 2, 4, 6, 8];

DigSet2 : Digits = [1..3, 5..7];

CharDigSet1 : CharDig = [ '0' , '2' , '4' , '6' , '8' ] ;

CharDigSet2 : CharDig = ['0'..'3', '5'..'7'];

CharSet : set of Char = ['a'..'z','A'..'Z'];

Типизированные константы типа "запись". В описании константы типа "запись" указываются как значения всех полей записи, так и их идентификаторы. В типизированных константах типа "запись" не допускаются поля файлового типа. В вариантных константах-записях допускается указание только того варианта полей, который соответствует установленной предварительно константе поля-признака. Поля указываются в том же порядке, в котором они следуют в описании типа. Примеры типизированных констант типа "запись".

type

Reс = record

R : Real;

В : Boolean;

C : Char;

end;

ArrayOfRec = array [1..3] of Rec;

RecOfArray = record

ArrInt : array [1..3] of Integer;

ArrChar : array [1..2] of Char;

end;

RecOfRec = record

I : Integer;

S : String;

Z : Rec;

end;

const

RecElem : Rec = ( R: 3.1415; B: True; С : '\*' ) ;

ArrRec : ArrayOfRec =

(( R: 3.1415; B: True; C : '\*'), ( R: 0.0; B: False; C : '$' ), ( R: 6.2832; B: True; C : '&' ));

RecArr : RecOfArray =

( Arrlnt: (1,2 ,3); ArrChar: (' 1' , ' 2') ) ;

RecRec : RecOfRec =

( I: 32767; S: 'PASCAL'; Z: ( R: 3.1415; В: True; С: '\*' ) ) ;

1. **Спецификация задачи: состав и назначение**

**1) Постановка задачи**

Описание задачи, поставленной перед программой. Собственно, причина создания программы вообще.

**2) УПЗ (Уточнённая постановка задачи)**

"Перевод" постановки от заказчика с человеческого на программные термины. На данном этапе формулируется устное описание работы программы в целом.

**3) Пример работы**

Пример, необходимый для читателя для лучшего понимания программы и задачи.

**4) Таблица данных (основной программы)**

Перечень всех переменных, с которыми работает программа. Необходима для конкретного определения кол-ва переменных и их точного описания для читателя

В таблице данных описываются все переменные, которые будут использоваться в программе.

Переменная – это величина, которая может меняться в процессе вычислений.

*Классы данных*: входные, выходные, промежуточные (не относятся к первым двум классам, но нужны для решения задачи).

*Имя* – это идентификатор, т.е. обозначение для переменных, констант... Имена переменных в программе должны быть различны, а также не должны совпадать с ключевыми словами, зарезервированными компилятором.

*Смысл* – это самая важная для программиста графа. Смысл переменной определяет, как она будет использоваться в программе.

*Тип* определяет, какие значения могут в ней храниться, размер переменной, допустимые операции.

*Структура переменной*: простая переменная – переменная, которая хранит только одно значение;

массив - множество значений одного типа, объединённых в одну структуру, массив имеет одно имя, а для доступа к конкретному элементу используют так называемый индекс элемента; запись – используется для хранения данных разного типа в одной переменной.

*Диапазон* задаёт минимальное и максимальное возможные значения переменной.

*Формат* описывает представление данных, например, количество знаков после запятой для вещественного числа и т.п. Формат обычно задаётся для выходных данных.

**5) Входная форма**

Описывает, какие данные и где должны быть расположены при вводе данных. При вводе данных с клавиатуры необходимо также выводить для пользователя подсказки, определяющие, что именно надо вводить в данный момент. Во входной форме обычно записываются имена переменных в угловых скобках. Имена переменных подсказывают порядок расположения переменных, а угловые скобки указывают, что при вводе надо набирать не само имя, а значение для данной переменной. При вводе с файла расположение переменных не учитывается.

**6) Выходная форма**

Описывает вид выводимых данных, а также пояснения, которые должны или могут быть выведены вместе с выходными данными.

1. **Аномалии**

это исходные данные, при которых невозможна правильная работа программы. Программа, соответственно, должна проверять введённые данные и правильно реагировать на аномальные ситуации – это определяет надёжность программы.

Описание всех аномальных ситуаций работы программы необходимо, чтобы :1) учесть их при написании программы разработчиком, 2)понимания некоторых условностей написания данного кода читателем

8) Функциональные/структурные тесты

Перечень тестов, проверяющих работу программы. Необходимы, чтобы читатель сам мог проверить правильность работы программы.

9) Метод

Подробное описание работы какой-то части программы, необходимо для лучшего понимания работы кода как читателем, так и разработчик.

10) Блок-схема

Схематическое изображение кода программы.

11) Код программы

Собственно, код