Вопросы к государственному экзамену

для направления 010400.62 Прикладная математика и информатика в 2014-2015 учебном году.

Дисциплины программистского цикла

1. Операторы цикла: с параметром, с предусловием, с постусловием.
2. Подпрограммы. Два типа подпрограмм. Обмен информацией между вызывающей программой и подпрограммой. Параметры – значения. Параметры – переменные. Принцип локализации.
3. Страничная организация памяти.
4. Понятие процесса. Состояния процесса. Операции над процессами.
5. Ассемблер. Команды сложения и вычитания ADD и SUB.
6. Ассемблер. Команда цикла LOOP.
7. Комбинированный тип. Иерархические записи. Оператор присоединения.
8. Динамическая память. Адреса и указатели. Операции над указателями. Динамические структуры данных.
9. Модель «Сущность – связь». Сущности. Связи. Атрибуты. Ключи. Их виды. Миграция ключей.
10. Нормализация. Н1НФ, 1НФ, 2НФ, 3НФ, НФБК, 4НФ. Правила приведения к нормальным формам.
11. Объектная и объектно – реляционная модели данных. Типы. Классы. Объекты. Отображение реляционной модели на объектную.
12. Ресурс панели диалога. Модальные и немодальные панели диалога.
13. Интерфейс графических устройств GDI. Контекст устройства. Графические примитивы.
14. Алгоритм разбиения средней точкой для отсечения невидимых линий.
15. Алгоритм плавающего горизонта.
16. Основы создания оконных приложений на Java.
17. Обработка исключений в Java.
18. Свойства, методы и события класса: TForm.
19. Свойства, методы и события класса: TIBTable.
20. Система передачи данных компьютерной сети. Основные понятия и технологии.
21. Модель сетевых взаимодействий OSI.
22. Клиент-серверная модель распределенных сетевых приложений.
23. Задача аутентификации и персонализации пользователей информационной сети.
24. Базовые операторы языков C/C++. Условный (if) и множественного выбора (switch). Порядок вычисления математических выражений. Пре- и пост- инкремент и декремент.
25. Статическая и динамическая память, оператор new/delete. (new[],delete[]).

№1. ОПЕРАТОРЫ ЦИКЛА: С ПАРАМЕТРОМ, С ПРЕДУСЛОВИЕМ, С ПОСТУСЛОВИЕМ. наверх

В Паскале существуют 3 оператора цикла: *с предусловием, с постусловием и параметром.*

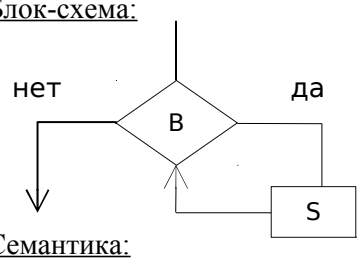
Оператор цикла с предусловием.

Синтаксис:

while B do S;

где B – булевское выражение, S – оператор языка Паскаль.

Блок-схема:



Семантика:

Вычисляется булевское значение B. Если оно равно true, то выполняется операция S, а затем снова вычисляется выражение B. Если значение выражения равно false, то выполняется следующий за циклом оператор.

Пример:

while true do S; *//бесконечный цикл – пример зацикливания*

while false do S; *//цикл не выполнится ни разу*

Оператор S называется *телом цикла*.

Для того, чтобы избежать ситуацию зацикливания необходимо, но недостаточно организовать тело цикла таким образом, чтобы операторы в нем влияли на условие B:

while a<10 do a:=a+1;

Пример: Вычислить n!

```
…
n!: = 1;
i := 1;
while i <= n do begin
        n! := n! * i;
        i  = i + 1;
end;
write('n!=',n!);
```

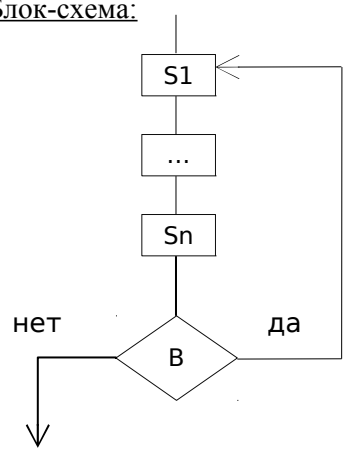
Оператор цикла с постусловием.

Синтаксис:

repeat S1; …Sn until B;

где B – булевское выражение, S1…Sn – оператор языка Паскаль.

Блок-схема:



Семантика:

Выполняются операторы S1,...,Sn затем вычисляется значение выражения B. Если оно равно false, то снова выполняются операторы S1,...,Sn, если оно равно true, то выполняется следующий за циклом оператор.

Отличия оператора while от оператора repeat:

1. Выход из цикла в операторе while выполняется по значению false, а в операторе repeat – true.
2. Тело цикла оператора repeat выполняется хотя бы один раз.
3. Тело цикла оператора repeat может состоять из нескольких операторов без операторных скобок.

Пример: Вычислить n!

```
…
n!: = 1;
i := 1;
repeat
        n! := n! * i;
        i  = i + 1;
until n = i  - 1;
write('n!=',n!);
```

Оператор цикла с параметром.

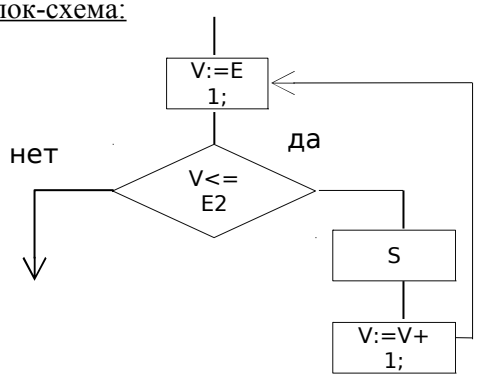
Их можно использовать в тех случаях, когда известно количество итераций тела цикла.

Синтаксис:

for V := E1 to E2 do S; {E1<=E2}

где V – переменная некого скалярного дискретного упорядоченного типа,E1 и E2 – в общем случае, выражения того же типа, что и переменная V, S – оператор языка Паскаль.

Блок-схема:



Семантика:

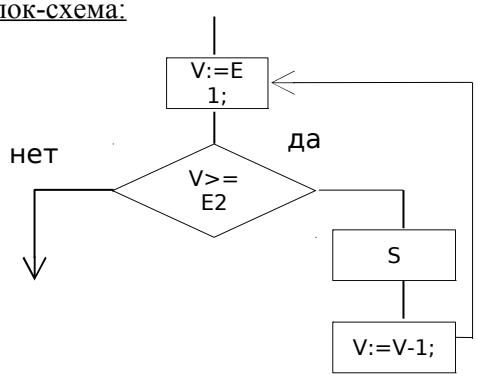
Присваиваем значение переменной V, затем вычисляем значение булевского выражения V<=E2. Если оно равно true, то выполняем действие S, увеличиваем значение переменной V на 1, а затем снова вычисляем значение выражения V<=E2. Если значение выражения равно false, то выполняется следующий за циклом оператор.

Существует следующая модификация:

Синтаксис:

for V := E1 downto E2 do S; {E1>=E2}

Блок-схема:



Семантика:

Присваиваем значение переменной V, затем вычисляем значение булевского выражения V>=E2. Если оно равно true, то выполняем действие S, уменьшаем значение переменной V на 1, а затем снова вычисляем значение выражения V>=E2. Если значение выражения равно false, то выполняется следующий за циклом оператор.

Пример: Вычислить n!

```
…
n!: = 1;
i := 1;
for i := 1 to n do n! := n! * i;
write('n!=',n!);
```

№2. ПОДПРОГРАММЫ. ДВА ТИПА ПОДПРОГРАММ. ОБМЕН ИНФОРМАЦИЕЙ МЕЖДУ ВЫЗЫВАЮЩЕЙ ПРОГРАММОЙ И ПОДПРОГРАММОЙ. ПАРАМЕТРЫ – ЗНАЧЕНИЯ. ПАРАМЕТРЫ – ПЕРЕМЕННЫЕ. ПРИНЦИП ЛОКАЛИЗАЦИИ. наверх

Подпрограммы могут быть описаны в любом месте программы, но до тела главной программы.

Формат (структура) **процедур**:

```
procedure  <имя процедуры> (<параметры>);
label <метки>;
const <объявление констант>;
type  <определения типов данных>;
var <объявления переменных>;
< Объявления подпрограмм >
begin
  <тело процедуры - операторы>
end;
```

Функции имеют такой же формат, как и процедуры, только они начинаются с заголовка function и заканчиваются типом данных возвращаемого значения:

```
function имя функции (параметры): тип данных;
```

Мы видим, что структура подпрограмм практически повторяет структуру главной программы. Имеются только два различия между главной программой и подпрограммами:
- процедуры и функции имеют заголовок procedure или function, соответственно, а не program;
- процедуры и функции заканчиваются точкой с запятой (;), а не точкой (.).

Процедуры и функции могут иметь описания своих констант, типов данных, переменных и свои процедуры и функции. Но все эти элементы могут быть использованы только в тех процедурах и функциях, где они объявлены (**локализация**). *Локальные идентификаторы* описаны в подпрограмме, и их можно использовать только в теле этой подпрограммы. *Глобальные идентификаторы* - это идентификаторы, описанные в главной программе. Они действительны и в теле главной программы и в телах подпрограмм (описание подпрограммы расположено ПОСЛЕ описания этого глобального идентификатора).

ПАРАМЕТРЫ подпрограмм.
При описании подпрограммы **параметры** записываются в круглых скобках после идентификатора подпрограммы. Напишем функцию для вычисления гипотенузы треугольника. Длина гипотенузы зависит от длины катетов. Поэтому заголовок функции можно записать так:

```
function Hyp(a,b: real):real;
```

Эта функция возвращает длину гипотенузы, т. е. вещественное число, например, типа real.
Чтобы вычислить гипотенузу конкретного треугольника в главной программе функцию Hyp нужно ВЫЗВАТЬ фнкцию, записав ее имя в каком-либо выражении.
Например:

```
writeln('Гипотенуза треуг. с катетами 3 и 4 равна ', Hyp(3,4):0:2);
```

Видим, что место параметров a, b занимают 3 и 4 - называемые ФАКТИЧЕСКИМИ параметрами. При вызове подпрограмм места ФОРМАЛЬНЫХ параметров занимают фактические параметры (соответствие определяется по порядку). А формальные параметры используются при записи тела подпрограммы.
Закончим описание функции Hyp:

```
function Hyp(a,b: real):real;
begin Hyp := sqrt(sqr(a)+sqr(b));
end;
```

Видим, что формальные параметры используются в теле функции. Заодно отметим особенность функции: необходимо, чтобы в ее теле присутствовало присваивание имени функции какого-то значения.
Параметры в этом примере - **параметры-значения**. Это означает, что:

1. Фактическим параметром является выражение (в частности - константы, переменные)
2. Фактический параметр защищен от изменений: даже если изменяется величина соответствующего формального параметра, а фактическим параметром служит переменная.

В отличие от параметров-значений **параметры-переменные** описываются начиная с var. Например, заголовок процедуры, вычисляющей гипотенузу, может выглядеть так:

```
procedure hypot(a,b: real; var c: real);
```

Здесь 3-й параметр - **"c"** является параметром-переменной. Только такие параметры можно применять для передачи результатов из подпрограммы в главную программу (второй вариант - через имя функции). При этом тип фактического параметра должен совпадать с типом формального параметра.
Завершим описание этой процедуры:

```
procedure hypot(a,b: real; var c: real);
begin
  c := sqrt(sqr(a)+sqr(b));
end;
```

Как видим, различие с функцией в том, что результат - вычисленная длина гипотенузы присваивается параметру-переменной

№3. СТРАНИЧНАЯ ОРГАНИЗАЦИЯ ПАМЯТИ. наверх

Страничная память — способ организации виртуальной памяти, при котором единицей отображения виртуальных адресов на физические является регион постоянного размера (*страница*). Типичный размер 4Кбайта.
Поддержка такого режима присутствует в большинстве 32битных и 64битных процессоров. Такой режим является классическим для почти всех современных ОС, в том числе Windows и семейства UNIX.

Организация памяти в виде страниц борется с двумя проблемами:

1. Внешней фрагментацией – используются блоки фиксированного размера в виртуальной и физической памяти, т.е. все запросы на выделение памяти будут кратны, не будет оставаться некратных зон.
2. Внутренней фрагментацией – блоки достаточно малого размера, поэтому (К) будет мал.

Трансляция адресов.
Виртуальный адрес состоит из двух частей: номер виртуальной страницы (***VPN***) и смещение внутри страницы.
Номер виртуальной страницы (***VPN***- virtual page number) это индекс в таблице страниц.
Запись в таблице страниц (***PTE*** – page table entry) содержит номер фрейма (***PFN*** –page frame number).
Номер фрейма – это номер физической страницы.
Фрейм – это страница физической памяти.

Смысл таблицы страниц – какая виртуальная страница соответствует какому фрейму физической памяти.
Есть виртуальный адрес, состоящий из двух частей:
- №виртуальной страницы, по нему идет поиск в таблице страниц и находится № фрейма, он составляет одну часть физического адреса;
- Смещение – берется напрямую – вторая часть физического адреса.

По новому адресу осуществляется поиск уже в физической памяти и доступ к данным.

Преимущества страничной памяти:

- Легко выделять физическую память
- Списки свободных фреймов, выделить фрейм
- Просто удалить из списка свободных;
- Внешняя фрагментация не проблема, т.к. фреймы одного размера;
- Естественный подход
- Всей программе не нужно быть резидентной – это «побочный» продукт;
- Все страницы одного размера;
- Основа – устранение внешней фрагментации.

Недостатки страничной памяти:

- Внутренняя фрагментация
- Процессам может быть нужны размеры, некратные размеру страницы;
- По сравнению с размером адресного пространства, размер страницы очень мал.
- Накладные расходы при обращении к памяти– вначале к таблице страниц, а затем уже к памяти.

Решение: аппаратный КЭШ для обращений к таблице страниц (TLB translation lookaside buffer – буфер внутри процессора).

- Большой объем памяти, требуемый для хранения таблиц страниц.
- Один PTE на одну страницу в виртуальном адресном пространстве

№4. ПОНЯТИЕ ПРОЦЕССА. СОСТОЯНИЯ ПРОЦЕССА. ОПЕРАЦИИ НАД ПРОЦЕССАМИ. наверх

Понятие *процесса* характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

Процесс находится под управлением *операционной системы*, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение. В состоянии исполнение происходит непосредственное выполнение программного кода процесса.
Выйти из этого состояния процесс может по трем причинам:

- операционная система прекращает его деятельность;
- он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание ;
- в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность.

Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними:

- создание процесса – завершение процесса ;
- приостановка процесса (перевод из состояния исполнение в состояние готовность) – запуск процесса (перевод из состояния готовность в состояние исполнение);
- блокирование процесса (перевод из состояния исполнение в состояние ожидание) – разблокирование процесса(перевод из состояния ожидание в состояние готовность).

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми.

№5. АССЕМБЛЕР. КОМАНДЫ СЛОЖЕНИЯ И ВЫЧИТАНИЯ ADD И SUB. наверх

Сложение

Для сложения двух чисел предназначена команда **ADD**. Она работает как с числами со знаком, так и с числами без знака (это особенность дополнительного кода).

```
ADD op1,op2
;op1+=op2
SUB op1,op2   ;op1-=op2
```

Операнды должны иметь одинаковый размер (нельзя складывать 16- и 8-битное значение). Результат помещается на место первого операнда.

После выполнения команды изменяются флаги, по которым можно определить характеристики результата:

1. Флаг CF устанавливается, если при сложении произошёл перенос из старшего разряда. Для беззнаковых чисел это будет означать, что произошло переполнение и результат получился некорректным.
2. Флаг OF обозначает переполнение для чисел со знаком.
3. Флаг SF равен знаковому биту результата – 1 если результат отрицательный, 0 иначе (естественно, для чисел со знаком, а для беззнаковых он равен старшему биту и особо смысла не имеет).
4. Флаг ZF устанавливается в единицу, если результат равен 0.
5. Флаг PF — признак чётности, равен 1, если результат содержит нечётное число единиц.

Примеры:

```
add ax,5      ;AX = AX + 5
add dx,cx     ;DX = DX + CX
add dx,cl     ;Ошибка: разный размер операндов.(dx – 2 байта, cl – 1 байт)
```

Вычитание

Вычитание выполняется с помощью команды **SUB**. Результат также помещается на место первого операнда и опять же выставляются флаги. Единственная разница в том, что происходит вычитание, а не сложение.

На самом деле вычитание в процессоре реализовано с помощью сложения. Процессор меняет знак второго операнда на противоположный, а затем складывает два числа.

№6. АССЕМБЛЕР. КОМАНДА ЦИКЛА LOOP. наверх

Для организации цикла с заранее известным количеством итераций предназначена команда **LOOP**. У этой команды один операнд — имя метки, на которую осуществляется переход. В качестве счётчика цикла используется регистр CX. Команда LOOP выполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде.

Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Максимально может быть 65535 повторений(2^{16}). Ещё одно ограничение связано с дальность перехода.

```
Пример:
MOV CX,[к-во
итераций]
metka:
        [тело цикла]
        loop metka
```

Метка должна находиться в диапазоне -127…+128 байт от команды LOOP (если это не так, FASM сообщит об ошибке). Для организации длинных циклов можно использовать команду безусловного перехода jmp:

```
MOV CX,[количество итераций]
metka:
                [тело цикла]
        jmp metka2
       metka1: jmp metka
        metka2:
                [тело цикла]
        Loop metka1
```

Кроме команды LOOP и команд условных переходов существуют ещё две команды, позволяющие организовывать циклы. Это команды LOOPZ (или её синоним LOOPE) и LOOPNZ (синоним — LOOPNE). Действие этих команд очень напоминает LOOP, за исключением того, что дополнительно анализируется флаг нуля ZF.

Переход к метке цикла осуществляется в том случае, если после декремента содержимое CX не равно 0 и выполняется условие: ZF=1 (для команды LOOPZ/LOOPE) или ZF=0 (LOOPNZ/LOOPNE). Эти команды удобно использовать для досрочного выхода из цикла.

Иногда требуется организовать вложенный цикл, то есть цикл внутри другого цикла. В этом случае необходимо сохранить значение CX перед началом вложенного цикла и восстановить после его завершения (перед командой LOOP внешнего цикла). Сохранить значение можно в другой регистр, во временную переменную или в стек.

```
MOV CX,[количество итераций для внешнего цикла]
M1:
        mov BX,CX   ; сохраняем итерации
        movCX,[к-во итераций для внутреннего цикла]
M2:
        [тело внутреннего цикла]
        Loop M2
mov CX,BX   ; возвращаем количество сохраненных
итераций в CX
[тело внешнего цикла]
Loop M1
```

№7. КОМБИНИРОВАННЫЙ ТИП. ИЕРАРХИЧЕСКИЕ ЗАПИСИ. ОПЕРАТОР ПРИСОЕДИНЕНИЯ. наверх

Запись - это структурированный тип, содержащий набор объектов разных типов. Составляющие запись объекты называются ее *полями*. В записи каждое поле имеет свое собственное *имя*. Чтобы описать запись, необходимо указать ее имя, имена объектов, составляющих запись и их типы.

Общий вид такой:

```
Type
"имя записи" = Record
"поле 1" : "тип 1";
"поле 2" : "тип 2";
...
"поле n" : "тип n"
End;
```

Пример:

```
Type
pupil = Record
fam: String[15];        {поле фамилии ученика}
b1, b2, b3, b4, b5 : 2...5;        {поля баллов по дисциплинам}
sb : Real       {поле среднего балла}
End;
```

Доступ к полям записи можно осуществить двумя способами:

1. Указанием имени переменной и имени поля.

Пример:

```
klass[2].fam, klass[3].sb, klass[1].b4;
```

2. Использованием оператора присоединения, который позволяет осуществлять доступ к полям записи, таким образом, как если бы они были простыми переменными.
Его общий вид: **With <имя записи> Do <оператор>.** Внутри оператора к компонентам записи можно обращаться только с помощью имени соответствующего поля.

Пример:

```
With klass [i] Do
Begin
Readln (fam);
ReadLn (b1,b2,b3,b4,b5);
End;
```

В поля переменной комбинированного типа (в поля записи) можно поместить данные любого типа, в том числе комбинированного. Следовательно, компонентом одной записи можно использовать другую запись:

```
Type
Date = Record
  Day: byte;
  Month: integer;
  Year: integer
end;
Name = Record
  first, given, last: string
end;
Address = Record
  City, Street: string;
  Home: byte
end;
Student = Record
   name: Name;
   address: Address;
   birthDate: Date
end;
```

№8. ДИНАМИЧЕСКАЯ ПАМЯТЬ. АДРЕСА И УКАЗАТЕЛИ. ОПЕРАЦИИ НАД УКАЗАТЕЛЯМИ. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. наверх

Динамическая память.

Динамическая память -- это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кбайт), стека (обычно 16 Кбайт) и собственно тела программы. Размер динамической памяти можно варьировать в широких пределах. По умолчанию этот размер определяется всей доступной памятью ПК и, как правило, составляет не менее 200...300 Кбайт.
Динамическая память -- это фактически единственная возможность обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования динамической памяти. Динамическая память широко используется для временного запоминания данных при работе с графическими и звуковыми средствами ПК. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных, к ним нельзя обращаться по именам, как к статическим переменным.

Адреса и указатели.

Оперативная память ПК представляет собой совокупность элементарных ячеек для хранения информации -- байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами*, они позволяют обращаться к любому байту памяти.
Турбо Паскаль предоставляет в распоряжение программиста гибкое средство управления динамической памятью -- так называемые указатели. *Указатель* -- это переменная, которая в качестве своего значения содержит адрес байта памяти.
В ПК адреса задаются совокупностью двух шестнадцатиразрядных слов, которые называются сегментом и смещением. *Сегмент* -- это участок памяти, имеющий длину 65536 байт (64 Кбайт) и начинающийся с физического адреса, кратного 16 (т.е. 0, 16, 32, 48 и т.д.). *Смещение* указывает, сколько байт от начала сегмента необходимо пропустить, чтобы обратиться к нужному адресу. Как правило, в Паскале указатель связывается с некоторым типом данных. Такие указатели будем называть типизированными. Для объявления типизированного указателя используется значок ^ , который помещается перед соответствующим типом, например:

```
var
  p1: ^integer;
  p2: ^real;
type
  PerconPointer = ^PerconRecord;
  PerconRecord = record
    Name: string;
    Job: string;
    Next: PerconPointer
  end;
```

Для указателей допустимы операции сравнения и присваивания.

Присваивание. Указателю можно присвоить содержимое другого указателя того же самого типа или константу NIL – пустой, или адрес объекта с помощью функции ADDR или оператора @.

Пример:

```
P1:=PP;

P2:=nil;

P3:=addr(x);

P4:=@x;
```

Сравнение указателей — сравнение значений указателей. Равенство указателей означает, что две переменных указывают на один и тот же объект данных (участок памяти).

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

Линейные списки.

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

Каждый элемент связанного списка:

- хранит какую-либо информацию,
- указывает на следующий за ним элемент.

Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется звеном, а структура из таких записей называется списком или цепочкой.

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

Описание списка:

```
Type ukazat= ^ S;
  S= record
     Inf: integer;
     Next: ukazat;
  End;
```

Формирование списка:

```
Type ukazat= ^S;
  S= record
     Inf: integer;
     Next: ukazat;
  End;
```

Виды списков:

- Стек – особый вид списка, обращение к которому идет только через указатель на первый элемент. Если в стек нужно добавить элемент, то он добавляется впереди первого элемента, при этом указатель на начало стека переключается на новый элемент. Алгоритм работы со стеком характеризуется правилом: «последним пришел – первым вышел».

- Очередь – это вид списка, имеющего два указателя на первый и последний элемент цепочки. Новые элементы записываются вслед за последним, а выборка элементов идет с первого. Этот алгоритм типа «первым пришел – первым вышел».

- Можно организовать списки с произвольным доступом к элементам. В этом случае необходим дополнительный указатель на текущий элемент.

№9. МОДЕЛЬ «СУЩНОСТЬ – СВЯЗЬ». СУЩНОСТИ. СВЯЗИ. АТРИБУТЫ. КЛЮЧИ. ИХ ВИДЫ. МИГРАЦИЯ КЛЮЧЕЙ. наверх

Семантическая модель “Сущность-Связь” (Entity-Relationship)

Наиболее известна семантическая модель “сущность – связь”

Три основных понятия ER-модели: сущность, связь, атрибут. У атрибута имеется значение. Определены четыре уровня представления информации и данных, в которых рассматриваются все модели данных.

Сущность и набор сущностей

Сущность это воображаемый объект, информация о котором должна сохраняться в своем наборе сущностей.

Сущность определяет тип, а не экземпляр. На ER- диаграммах сущность представляется прямоугольником, в котором обязательно указывается имя сущности. Дополнительно можно указывать примеры экземпляров сущности.

Связи

Связь – это типовое понятие, устанавливающее правила связывания сущностей. Каждый экземпляр типа связи, устанавливается между экземплярами типа сущности.

Концы бинарной связи в ER-модели характеризуется: именем роли (имя конца связи), определяющей функцию связи по отношению к связываемой сущности; степенью конца связи (сколько экземпляров данного типа сущности должно присутствовать в каждом экземпляре данного типа связи); обязательностью связи (т. е. любой ли экземпляр связываемой сущности должен участвовать в некотором экземпляре данного типа связи).

Атрибуты, значения, наборы и типы значений

Атрибут - это свойство сущности или связи, получаемое путем наблюдения или измерения. Информацию об экземпляре сущности выражают набором пар “атрибут – значение”

Атрибут принимает одно или несколько значений из некоторого набора.

Связи также имеют атрибуты. Выделим две разновидности атрибутов связей:

1. Атрибуты, через которые осуществляется привязка к связываемым сущностям.
2. Атрибуты, определяющие свойства сущностей, проявляющиеся только при наличии связи. Такие атрибуты называют эмерджентными.

Ключи.

Свойства первичного ключа

- Уникальным образом идентифицирует экземпляр.
- Не использует NULL значений.
- Не изменяется со временем.

Экземпляр идентифицируется при помощи ключа.

При изменении ключа, меняется экземпляр.

Свойства уникального ключа

Уникальные ключи (Unique Key) отличаются от первичных тем, что в них могут использоваться неопределенные значения null

Внешние ключи

Если сущности связаны, то связь может передать ключ (набор ключевых атрибутов) дочерней сущности. Эти переданные атрибуты называются внешними ключами. Передаваемые атрибуты называют *мигрирующими*.

Сильные и слабые сущности

Может оказаться, что в первичный ключ сущности обязательно необходимо включить внешний ключ. Такая сущность называется слабой или зависимой.

Альтернативные ключи

Потенциальные ключи, не использующиеся как первичные, могут быть определены как *альтернативные ключи* и записаны в секции данных модели с символом (AKn.m), где n.m – номер альтернативного ключа в формате

“номер_сущности”. “номер_ключа”.

№10. НОРМАЛИЗАЦИЯ. Н1НФ, 1НФ, 2НФ, 3НФ, НФБК, 4НФ. ПРАВИЛА ПРИВЕДЕНИЯ К НОРМАЛЬНЫМ ФОРМАМ. наверх

Отношению в реляционной модели данных соответствует сущность в ER-диаграмме. Любая реляционная связь представима связью в ER-модели. Из наличия такого отображения следует возможность переноса понятия и алгоритма нормализации из реляционной модели в модель «сущность-связь». Обратный переход от ER- диаграмм связан с потерей некоторой части информации и, может быть замены одной связи двумя и более. Последнее определяется тем, что связи в реляционной модели более примитивны.

Пример 1NF:

Ненормализованная таблица:

Подписчик	Издание
Иванов	Правда, Известия, Коммерсант
Петров	ДАН

претендует на роль ключа

1НФ:

Подписчик	Издание
Иванов	Правда
Иванов	Известия
Иванов	Коммерсант
Петров	ДАН

ключ

Определение 1НФа (через атрибуты): Отношение находится в 1НФ, если значения всех его атрибутов атомарны.
Определение 1НФк (через ключи): Отношение находится в 1НФ, если оно имеет ключ.
Утверждение: 1НФа ⇒ 1НФк.
Правила приведения к 1НФ: Разделить составные атрибуты на простые (атомарные); Выделить "повторяющиеся" атрибуты; Для каждой такой группы атрибутов создать новую справочную сущность с одним атрибутом для повторяющейся группы; Перенести в нее все значения повторяющихся атрибутов; Установить идентиф. связь 1: N от исходной сущности к каждой созданной справочной сущности.

Определение 2NF: Если набор атрибутов $\mathbf{B = \{B_j\}}$ зависит от всего набора атрибутов $\mathbf{A = \{A_i\}}$, но не зависит от части этого набора, то говорят, что функциональная зависимость **f: A → B** полная.
Определение 2НФа (через атрибуты): Отношение в 1НФ находится в 2НФ, если ни один атрибут вне первичного ключа не находится в функциональной зависимости от атрибутов, образующих часть ключа.
Определение 2НФк (через ключи): Отношение в 1НФ находится в 2НФ, если каждый неключевой атрибут, находится в полной функциональной зависимости от ключа.
Приведения сущности ко 2НФ: Выделить атрибуты, которые зависят только от части первичного ключа, создать новую сущность; Поместить атрибуты, зависящие от части ключа, в их собственную (новую) сущность; Установить идентифицирующую связь от прежней сущности к новой.
Определение 3NF (транзитивной и прямой ФЗ): функциональная зависимость A→C называется транзитивной, если найдется атрибут B такой, что A→B, B→C функциональные зависимости. Если не существует транзитивной зависимости, то функциональная зависимость называется прямой.
Определение 3НФк (через ключи): отношение в 1НФ находится в 3НФ, если все его атрибуты прямо зависят от ключа.
Определение 3НФа (через атрибуты): отношение в 1НФ находится в 3НФ, если оно не содержит зависимостей неключевых атрибутов от других атрибутов, не образующих первичный ключ.
Правило приведения к 3НФ: Найти функциональную зависимость неключевых атрибутов от других неключевых атрибутов; Создать новую сущность. В соответствии с теоремой Хиса все ее атрибуты входят в найденную функциональную зависимость; Вычеркнуть атрибуты-значения найденной функции в исходной сущности; Установить неидентиф. связь от созданной сущности к исходной сущности.

Н1НФ(непервая НФ) – 1НФ удовлетворяющая условию 1НФк, но не удовлетворяющая условию 1НФа называется **непервой нормальной формой (Н1НФ, NFNF, NF^2).**
(НФБК) Необходимость проверки возникает при наличии пересекающихся первичных ключей. Характеризуется отсутствием функ-ьных зависимостей, действующих из атрибутов, принадлежащих только одному из пересекающихся ключей, в атрибуты, принадлежащие только второму из пересекающихся ключей.
Правило приведения.
1)Найти функ-ную зависимость неключ. атрибутов от других неключ. атрибутов.
2)Создать новую сущность. В соответствии с теоремой Хиса все ее атрибуты входят в найденную функц-ную зависимость.
3)Вычеркнуть атрибуты-значения найденной фун-ии в исходной сущности.
4)Установить неидентифицирующую связь от созданной сущности к исходной сущности.

4НФ – Отношение находится в 4НФ, если в нем отсутствуют многозначные зависимости, которые не являются функциональными.
преподаватель(№, должность) преподаватель_дети (№, дети), преподаватель_курс (№, курс)

5НФ. Отношение находится в 5НФ, если не существует ни одной его полной декомпозиции, в которую не входили бы проекции, не имеющие общего ключа.

Пусть R – отношение, а A и B – произвольные подмножества множества атрибутов отношения R. Тогда B функционально зависит от A A->B, тогда и только тогда, когда каждое значение множества A отношения R связано в точности с одним значением множества A отношения R.
Теорема Хиса.
Пусть R(A, B, C) является отношением, Где A, B и C – атрибуты этого отношения. Если B функционально зависит от A, то R равно соединению его проекций {A,B} и {A,C}

№11. ОБЪЕКТНАЯ И ОБЪЕКТНО – РЕЛЯЦИОННАЯ МОДЕЛИ ДАННЫХ. ТИПЫ. КЛАССЫ. ОБЪЕКТЫ. ОТОБРАЖЕНИЕ РЕЛЯЦИОННОЙ МОДЕЛИ НА ОБЪЕКТНУЮ.

наверх

Два пути *расширения модели данных до объектной*:

1. **Объектная модель (переход от ООП)**
- вместо таблиц используют классы эквивалентные таблицам (без учета методов);
- подход можно считать расширением языков ООП на персистентные классы.

2. **Объектно-реляционная модель (переход от РМД)**
- кроме плоских реляционных таблиц вводятся таблицы со сложными многоуровневыми шапками (классы), в которые встроены методы;
- создается переходник (объектные view), объединяющий реляционную и объектную модели;
- подход можно считать прямым расширением реляционной модели.

Особенности объектной модели:

- Основная линия прогресса в программировании определяется способом сегментации программы и характером взаимодействия сегментов. Желательно, чтобы сегменты по возможности не зависели друг от друга. Это необходимо для того, чтобы изменения в одном модуле вызывали как можно меньше изменений в других.
- В любой моделируемой системе обязательны два аспекта:
  - данные (какие сведения хранить)
  - функции (как преобразовывать данные).

В реляционной и иерархической моделях данных отображались только данные. Функциональная сторона плохо представлена. Поэтому, для реализации функциональной стороны в этих моделях пишется отдельный программный текст, не учитываемый моделью данных.

В настоящее время имеется две основных модели. Это расширение объектно ориентированных языков в сторону управления данными (стандарт ODMG - Object Data Management Group), либо добавление объектных свойств в реляционные СУБД.

Объектная модель ODMG обеспечивает набор встроенных объектных типов. Модель включает ряд встроенных структурных типов, позволяющих применять традиционные методы моделирования БД. Модель одновременно включает понятия объектов и литералов, т.е. возраст человека может задаваться целочисленным литералом, а не объектом, имеющим свойство21 возраст.

Объектные расширения реляционных СУБД.

Объектная модель СУБД Cache позволяет работать с данными одновременно в объектной, реляционной и иерархической моделях. Т.е.создавая объект, мы одновременно создаем дерево и объект. Методы написаны на основном для Cache не объектно-ориентированном языке ObjectScript.

Класс задаёт шаблон, по которому создаются объекты определённого им типа. Объект — это экземпляр класса. Метод представляет собой функции или процедуры класса или объекта, определяющие его поведение. Свойства класса считаются определяющими состояния класса, но они не используются для идентификации объектов, как в реляционной модели. Можно считать понятия класса и типа синонимами.

Классы объектов делятся на зарегистрированные и незарегистрированные. Каждый класс объектов имеет уникальное имя в пределах пространства имён, свойства, методы, параметры, запросы и индексы.
Незарегистрированные классы (Non-Registered Class) предназначены для создания пользовательской объектной системы и не имеют предопределенного поведения.
Зарегистрированные классы это временные классы, обладающие предопределенным поведением.
У каждого зарегистрированного объекта имеется уникальная объектная ссылка OREF, обеспечивающая доступ к объекту в памяти.
Зарегистрированные классы могут быть ещё хранимыми и встраиваемыми. Первые хранятся независимо и потому имеют уникальную не изменяемую объектную ссылку OID, по которой объект может быть найден на диске, и ссылку OREF.

В **объектно-реляционной модели данных СУБД Oracle** ситуация с реализацией методов та же. Только методы создаются на встроенном процедурном языке, который называется PL/SQL.

Объекты могут быть постоянными и временными.

Хранимый (persistent) объект может быть как значением столбца обычной таблицы, так и строкой таблицы (в этом случае таблицу называют объектной).
Временный (transient) объект создается в памяти и уничтожается после окончания работы программы.

Можно выделить следующие разновидности объектных типов:
- простой объектный тип, который строится на скалярных предопределённых типах данных;
- составной объектный тип, использующий другие объектные типы;
- ссылочный объектный тип;
- типы коллекций двух разновидностей VARRAY и NESTED TABLES.

Ссылочный объектный тип REF это логический указатель, определяющий отношения между экземплярами классов. Он основывается на одноэлементных или коллекционных типах данных. Указатели REF задают ассоциации UML и заменяют внешние ключи, предоставляя прямую навигацию между объектами разных типов.
VARRAY - это упорядоченная коллекция фиксированной длины. Хранится в сегменте таблицы, использующей такой тип.
Вложенная таблица это неограниченная и неупорядоченная коллекция. Хранится в своём сегменте, не совпадающем с сегментом основной таблицы.

.

№12. РЕСУРС ПАНЕЛИ ДИАЛОГА. МОДАЛЬНЫЕ И НЕМОДАЛЬНЫЕ ПАНЕЛИ ДИАЛОГА. наверх

Панелью диалога называется специализированное программное окно, поддерживающее фирмы IBM для ввода/вывода информации. Состав и расположение элементов управления указывается в ресурсе, описывающем панель диалога. Ядро ОС Windows предполагает 6 типов элементов управления: кнопки, комбинированные списки, строки редактирования, полосы прокрутки и статический текст. Помимо использования предопределенных элементов управления так же существует возможность использования нестандартных элементов управления, создаваемых программистом. Панели диалога бывают 2х типов: модальные и немодальные.

Модальная панель диалога не позволяет пользователю взаимодействовать с др.окнами программы до тех пор, пока пользователь не закроет эту панель.

Немодальная панель по своему поведению подобна окну и пользователь может параллельно взаимодействовать с др. окнами или др. немодальными панелями диалога программы.

Для создания и использования панели диалога необходимо выполнить след. шаги:

1) Создать ресурс типа панели диалога;
2) Создать диалоговую функция, обрабатывающую сообщения, посылаемые панели диалога;
3) Создать диалог, используя функцию DialogBox для модальных панелей диалога и CreateDialog для немодальных панелей диалога;
4) Для завершения работы с панелью диалога необходимо использовать функцию EndDialog для модальных панелей диалога или DestroyWindow для немодельных.

Диалоговая функция *имеет вид*:

```
function ExDial(Dlg: hWind; iMsg, wParam: Word; lParam: longint): WordBool; export;
begin
ExDial : = false;
case iMsg of
        WM_InitDialog:
                ExDial : = true;
        WM_command:
                case wParam of
                        ….      (*)
                end; end; end;
```

Сообщение *WM_InitDialog* посылается перед отображением панели диалога. В обработке этого события можно предусмотреть установку фокуса на любой элемент управления, входящий в состав панели диалога при помощи панели диалога при помощи функции SetFocus. Параметром этой ф-ии является ссылка на эл-т управления, который должен получить фокус. Возвращаемое значение – это ссылка на эл-т, который потерял фокус в результате действия этой функции. Если не требуется никаких спец.действий, то внутри сообщения WM_InitDialog обработчик должен вернуть true. В этом случае ядро Windows устанавливает фокус на 1й элемент управления, имеющий стиль WS_TABSTOP. SetFocus(GetDlgItem(Dlg,ID_EDIT)), где Dlg – ссылка на окошко.

Сообщение *WM_command* посылается диалоговой ф-ей элементами управления, которые входят в панель диалога. Эти сообщения посылаются в случае, когда пользователь производит любые действия с элементами управления. Такие сообщения называются *нотификационными*.

Для модальной панели диалога можно использовать те же стили, что и для окон. При отображении модальной панели диалога начинает работать собственный цикл обработки сообщений. Родительское окно становится неактивным и сообщения ему не посылаются. Для того, чтобы модальная панель диалога завершила свою работу обязательно необходимо использовать ф-ию EndDialog.

```
(*)…
case wParam of
        {ok} 1001: begin
                        ExDial : = true;
                        EndDialog(Dlg,0);
                end;
end;
….
function WndProc(…);
var ProcInst: Pointer;
begin
        case iMsg of
                WM_command:
```

```
                    case wParam of
{вызов диалога – пункта меню}      101: begin
                                   ProcInst : = MakeProcInstance(@ExDial, hInstance);
                                   DialogBox(hInstance, 'Dialog_1', Wnd, ProcInst);
                                   FreeProcInstance(ProcInst);
                                 end; end; end; end;
```

Ф-ия MakeProcInstance обеспечивает доступность данных программы в косвенно вызываемых ф-иях.
Ф-ия FreeProcInstance освобождает выделенную память под панель диалога.
Немод. панель диалога получает сообщение от основного цикла обработки сообщений. Для проверки, предназначено ли сообщение окну или панели диалога, используется ф-ия IsDialogMessage и основной цикл обработки сообщений будет выглядеть след. образом:

```
while GetMessage(Msg,0,0,0) do
            begin
            if (not IsDialog(MyDlg))and(ord(IsDialogMessage(MyDlg,msg))) = 0
            then begin
                TranslateMessage(msg);
                DispatchMessage(msg);
                end;
end;
```

В связи с тем, что немодальная панель диалога может присутствовать не все время работы программы, необходимо проверять ссылку на панель диалога MyDlg для того, чтобы убедиться в присутствии такой панели. Ф-ия IsDialogMessage отправляет сообщения, посылаемые панели диалога, на обработку, поэтому для таких сообщений не должны вызываться ф-ии TranslateMessage и DispatchMessage.

Функции используемые при работе с диалоговыми окнами

1) GetDlgItemInt(Dlg: hWnd; IDDlgItem: integer; Translate: pBool; Signed: Bool): word;

Эта ф-ия преобразует текст заданного органа управления в целочисленное значение.
Параметр Dlg – дескриптор диалогового окна;
IDDlgItem – индикатор органа управления;
Translate – указатель на переменную, которая получает значение успешного или неуспешного выполнения ф-ии. Этот параметр может принимать значение nil, тогда ф-ия не возвращает информацию об успешном или неуспешном выполнении;
Signed – обозначает знаковое или беззнаковое значение будет считываться из органа управления;
h : = GetDlgItemInt(Dlg,ID_EDIT,nil,true) – считывается знаковая ф-ия (целая).

2) GetDlgItemText(Dlg: hWnd; IDDlgItem: integer; S: pChar; MaxCount: integer): integer;

S – указатель на буфер, который получает текст;
MaxCount – максимальное кол-во символов, получаемых в буфер строки. Результатом ф-ии будет число символов, скопированных в буфер и ноль в случае неуспешного выполнения.
m : = GetDlgItemText(Dlg,ID_EDIT,S,10);

3) SetDlgItemInt(Dlg: hWnd; IDDlgItem: integer; V: word; Signed: Bool);

SetDlgItemInt(Dlg,ID_EDIT,100,false);

4) SetDlgItemText(Dlg: hWnd; IDDlgItem: integer; S: pChar);

S – указатель на нек. символьную строку, значение которой будет скопировано в орган упр-я.

5) CheckRadioButton(Dlg: hWnd; IDFirst, IDLast, IDCheck: integer);

6)CheckDlgButton(Dlg: hWnd; ID:integer; Check:word); - встав. или удал. флажок с CheckBox'а.

№13. ИНТЕРФЕЙС ГРАФИЧЕСКИХ УСТРОЙСТВ GDI. КОНТЕКСТ УСТРОЙСТВА. ГРАФИЧЕСКИЕ ПРИМИТИВЫ. наверх

GDI – модуль программы, который обрабатывает операторы работы с графикой. В GDI создаются функции, предназначенные для работы с графикой, такие как рисование точки, прямой, различных фигур. Эти функции вывода базовых геометрических фигур называются *графическими примитивами*.

GDI так же отвечает за преобразование этих графических команд для всех драйверов устройства вывода, включая мониторы и принтеры. Таким образом, GDI является интерфейсом, независимым от устройства вывода.

Все графические функции используют в качестве параметра контекст устройства. Это специальная структура содержит в себе основные характеристики устройства вывода, а так же различные средства отображения - кисти, карандаш, цвета. Если получение и удаление контекстного устройства происходит при обработке сообщения WM_Paint, то используется следующая пара функций:

```
MyDC := BeginPaint(Wnd, MyPaint);
…
EndPaint()Wnd, MyPaint);

var MyDC      : DC;
    MyPaint   : TPainStruct;
```

TPainStruct – запись, содержащая необходимую информацию для отображения графики. Например, она содержит область отображения. Если же нам необходим контекст устройства в любом другом сообщении, то используется пара функций:

```
MyDC := GetDC(Wnd);
….
RealiceDC(Wnd, MyDC);
```

В ОС Windows определены следующие средства для отображения:
-карандаш,
-кисть,
-текст.

Карандаш используется для рисования линий, окружностей, прямоугольников и тд. Можно задавать цвет линии, ее ширину и стиль. Для использования карандаша сначала необходимо получить ссылку на него:

```
Pen :   hPen;
Pen :=  GetStockObject(white_pen);
```

Создаем карандаш:

```
Pen := CreatePen(стиль, толщина, цвет);
```

Стили:

```
PS_Solid      _________
PS_Dash       - - - - - - - - -
PS_Dob        ……….
PS_DashDob    -.-.-.-.-.-.-.-.-
```

Цвет:

```
 RGB(255,255,255);  // 0..255
```

После создания карандаша, его необходимо выбрать:

```
SelectObject(MyDC, Pen);
```

Кисть:

```
Brush : hBrush;
Brush := GetStockObject(White_Brush);
Brush := CreateSolidBrush(цвет);
Brush := CreateHatchBrush(стиль, цвет);
```

Стили:

```
HS_Vertical                 // вертикальная штриховка
HS_Horisontal               // горизонтальная
HS_Cross                    //решетка
HS_BDiagonal                //слева-направо
HS_FDiagonal                //наоборот
HS_DiagCross                //косая решетка
```

После создания кисти, необходимо ее выбрать:

```
SelectObject(MyDC, Brush);
```

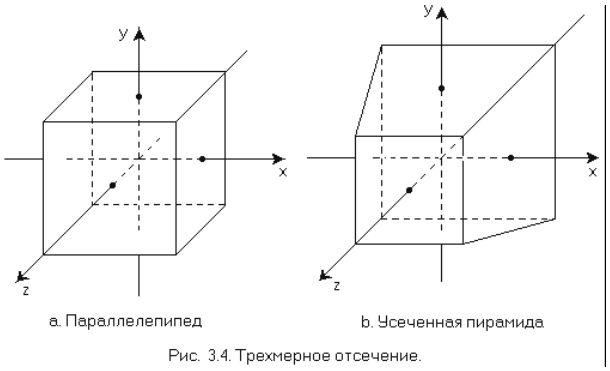
После окончания работы с кистью или карандашом, их необходимо удалить:

```
DeleteObject(MyDC, Pen/Brush);
```

Графические примитивы:

```
MoveTo(MyDC, x,y);
LineTo(MyDC, x,y);
Rectangle(MyDC, x1,y1,x2,y2);
 RoundRect(MyDC, x1,y1,x2,y2,x3,y3);
Ellipse(MyDC, x1,y1,x2,y2);
Arc(MyDC, x1,y1,x2,y2,x3,y3,x4,y4);
Chord( то же самое );
Pie( то же самое );
PolyLine(MyDC,a,b); // ломаная, где a:Tpoint – массив, n  - количество.
                        в массиве записи вида a[i].x , a[i].y
Pilygon(MyDC, a,n); - замкнутый многоугольник.
```

№14. АЛГОРИТМ РАЗБИЕНИЯ СРЕДНЕЙ ТОЧКОЙ ДЛЯ ОТСЕЧЕНИЯ НЕВИДИМЫХ ЛИНИЙ. наверх



a. Параллелепипед b. Усеченная пирамида

Рис. 3.4. Трехмерное отсечение.

Как и при двумерном отсечении, отрезки, которые полностью видимы или тривиально невидимы, можно идентифицировать с использованием обобщения кодов концевых точек Коэна-Сазерленда. В трехмерном случае используется 6-битовый код. Самый правый бит кода считается первым. В биты кода заносятся единицы с помощью обобщения двумерной процедуры. Конкретно единица заносится:

· в первый бит - если конец ребра левее объема,
· во второй бит - если конец ребра правее объема,
· в третий бит - если конец ребра ниже объема,
· в четвертый бит - если конец ребра выше объема,
· в пятый бит - если конец ребра ближе объема,
· в шестой бит - если конец ребра дальше объема.

В противном случае в соответствующие биты заносятся нули. И опять, если коды обоих концов отрезка равны нулю, то оба конца видимы и отрезок тоже будет полностью видимым. Точно так же, если побитовое логическое произведение кодов концов отрезка не равно нулю, то он полностью невидим. Если же это логическое произведение равно нулю, то отрезок может оказаться как частично видимым, так и полностью невидимым. В этом случае необходимо определять пересечения отрезка с гранями отсекающего объема.

Трехмерный алгоритм разбиения средней точкой

//Окно – массив 1x7, в котором содержатся координаты левой, правой,
// нижней, верхней, ближней, дальней сторон и центра ($x_л, x_п, y_н, y_в, z_б, z_д, z_{цп}$)
// прямоугольного отсекающего окна
//P_1, P_2 – концевые точки отрезка

i=1 *//инициализация i*

*/*Вычисление кодов концевых точек;занесение кодов каждого конца в массивы 1x6, именуемые T1код и T2код*/*
//первая концевая точка: P_1
1 **call** Конец(P_1, Окно; T1код, Сумма1)
  //вторая концевая точка: P_2
call Конец(P_2, Окно; T2код, Сумма2)
//проверка полной видимости отрезка
if Сумма1=0 и Сумма2=0 **then** 5
//отрезок не является полностью видимым
//проверка тривиальной невидимости отрезка
call Логическое(T1код, T2код; Произвед)
if Произвед<>0 **then** 6
//отрезок может оказаться частично видимым
//поиск наиболее удаленной от P_1видимой точки отрезка
//запоминание исходной точки P_1
Раб=P_1
//проверка окончания процесса решения
if i>2 **then** 4
//P_2 – наиболее удаленная от P_1 видимая точка отрезка?
if Сумма2=0 **then** 3

//пересечение уже найдено?

2 **if** $|P_1+P_2|$<Погрешность **then** 3

//вычисление средней точки

$P_m=(P¿¿1+P_2)/2¿$

//запоминание текущей точки P_1

Память=P_1

//замена P_1 *на* P_m

$P_1=P_m$

//вычисление нового кода точки P_1

call Конец(P_1, Окно; Т1код, Сумма1)

//проверка тривиальной видимости отрезка $P_m P_2$

call Логическое(Т1код, Т2код; Произвед)

if Произвед=0 **then** 2

//$P_m P_2$ *невидим, продолжить процедуру с* $P_1 P_m$

P_1=Память

$P_2=P_m$

go to 2

//обнаружена наиболее удаленная от P_1 *видимая точка отрезка*

//поиск наиболее удаленной от P_2 *видимой точки отрезка*

//перемена мест P_1 *и* P_2

3 $P_1=P_2$

P_2=Раб

//увеличить счетчик

i=i+1

//начать процесс с «новым» укороченным отрезком

go to 1

//теперь обнаружены оба пересечения

//проверка вырожденной невидимой точки

4 **call** Логическое(Т1код, Т2код; Произвед)

if Произвед<>0 **then** 6

5 Начертить отрезок

6 **finish**

*/*подпрограмма вычисления кодов концевой точки отрезка*/*

subroutine Конец(Р, Окно; Ткод, Сумма)

//P_x, P_y, P_z *– координаты x, y, z точки P*

//Окно – массив 1x7, в котором содержатся координаты левой, правой,

// нижней, верхней, ближней, дальней сторон и центра ($x_л, x_п, y_н, y_в, z_б, z_д, z_{цп}$*)*

// прямоугольного отсекающего окна

//Ткод – массив 1x6, содержащий коды концевой точки

//Сумма – поэлементная сумма Ткод

//вычисление $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$

$\alpha_1=x_п/(z_д-z_{цп})$

$\alpha_2=-\alpha_1 z_{цп}$

$\beta_1=x_л/(z_д-z_{цп})$

$\beta_2=-\beta_1 z_{цп}$

$\gamma_1=y_в/(z_д-z_{цп})$

$\gamma_2=-\gamma_1 z_{цп}$

$\delta_1=y_н/(z_д-z_{цп})$

$\delta_2=-\delta_1 z_{цп}$

//определение кодов концевой точки

if $P_x-P_z\beta_1-\beta_2<0$ **then** Ткод(6)=1 **else** Ткод(6)=0

if $P_x-P_z\alpha_1-\alpha_2<0$ **then** Ткод(5)=1 **else** Ткод(5)=0

if $P_y-P_z\delta_1-\delta_2<0$ **then** Ткод(4)=1 **else** Ткод(4)=0

if $P_y-P_z\gamma_1-\gamma_2<0$ **then** Ткод(3)=1 **else** Ткод(3)=0

if $P_z - z_б$>0 **then** Ткод(2)=1 **else** Ткод(2)=0
if $P_z - z_д$<0 **then** Ткод(1)=1 **else** Ткод(1)=0
//вычисление суммы кодов
Сумма=0
for i=1 **to** 6
    Сумма=Сумма+Ткод(i)
next i
return

*/*подпрограмма вычисления логического произведения*/*
subroutine Логическое(Т1код, Т2код; Произвед)
// Т1код, Т2код – массивы 1х6, содержащие коды концевых точек, Произвед – сумма битов логическом произведении кодов концов
    Произвед=0
    for i=1 **to** 6
        Произвед=Произвед+Целая часть((Т1код(i)+Т2код(i))/2)
    next i
    return

№15. АЛГОРИТМ ПЛАВАЮЩЕГО ГОРИЗОНТА. наверх

Алгоритм плавающего горизонта чаше всего используется для удаления невидимых линий трехмерного представления функций, описывающих поверхность в виде $F(x, y, z) = 0$.

Главная идея данного метода заключается в сведении трехмерной задачи к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координат x, y или z.

На рис. 5.2. приведен пример, где указанные параллельные плоскости определяются постоянными значениями z. Функция $F(x,y,z) = 0$ сводится к последовательности кривых, лежащих в каждой из этих параллельных плоскостей, например к последовательности $y=f(x,z)$ или $x=g(y,z)$, где z постоянно на каждой из заданных параллельных плоскостей.

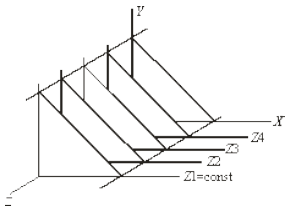


Рис.5.2. Секущие плоскости с постоянной координатой

Итак, поверхность теперь складывается из последовательности кривых, лежащих в каждой из этих плоскостей, как показано на рис. 5.3.**(y – вверх, x – вправо, z – на на нас)**

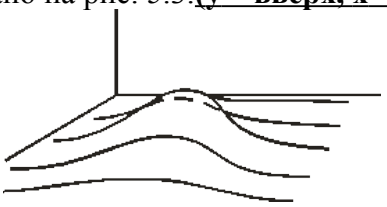


Рис. 5.3. Секущие плоскости с постоянной координатой

Алгоритм сначала упорядочивает плоскости z = const по возрастанию расстояния до них от точки наблюдения. Затем для каждой плоскости, начиная с ближайшей к точке наблюдения, строится кривая, лежащая на ней. Алгоритм удаления невидимой линии заключается в следующем.

Если на текущей плоскости при некотором заданном значении x соответствующее значение y на кривой больше значения y для всех предыдущих кривых при этом значении x, то текущая кривая видима в этой точке; в противном случае она невидима.

Невидимые участки показаны пунктиром на рис. 5.3.

Реализация данного алгоритма достаточно проста. Для хранения максимальных значений y при каждом значении x используется массив, длина которого равна числу различимых точек по оси x в пространстве изображения. Значения, хранящиеся в этом массиве, представляют собой текущие значения "горизонта". Поэтому по мере рисования каждой очередной кривой этот горизонт "всплывает". Фактически этот алгоритм удаления невидимых линий работает каждый раз с одной линией.

Алгоритм работает очень хорошо до тех пор, пока какая-нибудь очередная кривая не окажется ниже самой первой из кривых, как показано на рис. 5.5., а.

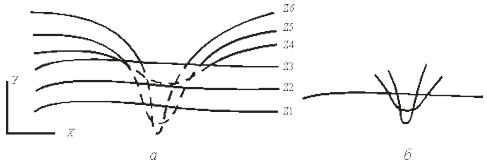


Рис. 5.5. Обработка нижней стороны поверхности

Подобные кривые, естественно, видимы и представляют собой нижнюю сторону исходной поверхности, однако алгоритм будет считать их невидимыми. Нижняя сторона поверхности

делается видимой, если модифицировать этот алгоритм, включив в него нижний горизонт, который опускается вниз по ходу работы алгоритма. Алгоритм теперь становится таким: если на текущей плоскости при некотором заданном значении x соответствующее значение y на кривой больше максимума или меньше минимума по y для всех предыдущих кривых при этом значении x, то текущая кривая видима. В противном случае она невидима.

Полученный результат показан на рис. 5.5., б.

На рис. 5.7 показан типичный результат работы алгоритма плавающего горизонта для функции $y = (1/5)\sin x \cos z - (3/2) \cos (7\alpha/4)\, e^{(-\alpha)}$, где $\alpha = (x - p)^2+(z - p)^2$, в интервале (0, 2p).

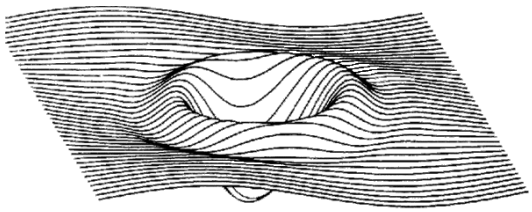


Рис. 5.7. Результат работы алгоритма плавающего горизонта

№16. ОСНОВЫ СОЗДАНИЯ ОКОННЫХ ПРИЛОЖЕНИЙ НА JAVA. наверх

Для создания оконных приложений в Java используется библиотека **SWING и AWT**. Причем в SWING хранятся элементы интерфейса для создания платформы независимых приложений, а в AWT содержится платформа ориентированные средства для работы с окнами.

Окно Верхнего уровня называется *фреймом*. Для такого окна есть класс:

SWING – класс JFrame
AWT – класс Frame

По умолчанию фрейм имеет размер 0x0. Если хотим видимый, то надо задавать. Также необходимо определить, что произойдет при закрытии окна. Создание фрейма не приводит к его автоматическому появлению на экране. По умолчанию фрейм считается невидимым.

Рассмотрим несколько методов JFrame, которые позволяют менять внешний вид форм:

Dispace () – закрывает окно и освобождает все системные ресурсы.
setIconImage (Image image) – получает объект класса Image для применения в качестве пиктограммы окна.
setTitle(String s) – позволяет установить строку s в качестве заголовка окна.
setResizable(Boolean b) – позволяет установить можно ли пользователю изменить размеры окна.

Большинство методов, которые позволяют изменить размеры и форму окна находятся в классах

Component
Window

Рассмотрим **методы класса Component для изменения окон**:

setVisible(Boolean b)-либо скрывает либо отображает компоненты.
is Showing()-позволяет проверить можно ли отображать компонент на экране или нет.
isEnabled()-позволяет проверить доступен компонент или нет
setEnabled(Boolean b)-запрещает или разрешает доступ к компоненту.
Dimension getSize()-возвращает текстовый размер компонента.
setBounds(int x, int y, int width, int neight)-устанавливает размеры и перемещает компоненты.
SetSize(int width, int neight)- устанавливает размеры.

Для создания фрейма необходимо реализовать **касс, который расширяет класс JFrame**

```
Import javax.swing *;
class SimpleFrame extends JFrame {
        public SimpleFrame{
                setSize(DEFAULT_WIDTH,DEFAULT_HEIGHT)
        }
        public static final int DEFAULT_WEIGHT=300;
        public static final int DEFAULT_ HEIGHT =200;
}
public class SimpleFrameTest{
        public static void main(String [] args){
                SimpleFrame frame=new SimpleFrame();
                Frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
                Frame.setVisible(true);
        }
```

}

Фреймы в Java предназначены для того, чтобы быть контейнерами для других компонентов.

Есть еще один компонент – это *панель:*

Swing – панель JPanel
AWT – панель Panel

Панель добавляется к фрейму и может либо так же случить контейнером, либо на ее поверхности можно осуществлять рисование. Для того, что бы рисовать на панели необходимо реализовать класс расширяющий JPanel и переопределить в этом классе метод paintComponent(). Этот метод получает в качестве параметра объект класса Graphics, в котором содержится набор установок для изображения рисунков и текстов.

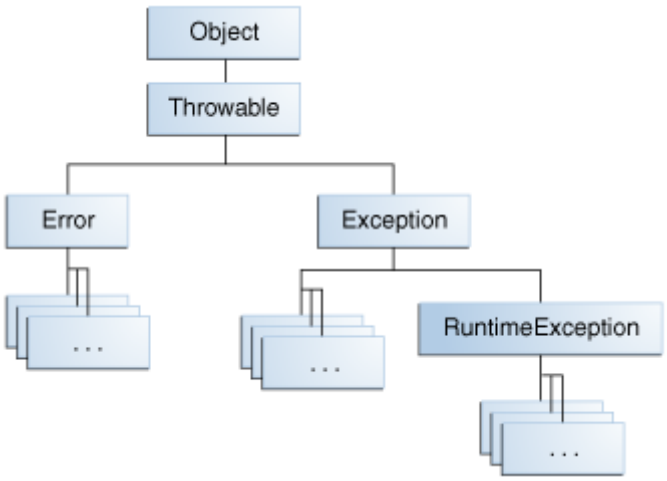
```
class MyPanel extends LPanel{
    public void paintComponent(Graphics g){
    …
    }
}
```

№17. ОБРАБОТКА ИСКЛЮЧЕНИЙ В JAVA. наверх

Исключениями или *исключительными ситуациями* (состояниями) называются ошибки, возникшие в программе во время её работы.

Все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком.

Иерархия классов исключений:



Исключения делятся на несколько классов, но все они имеют общего предка — класс Throwable. Его потомками являются подклассы Exception и Error.

Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы и предсказуемы. Например, произошло деление на ноль в целых числах.

Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM.

В Java все исключения делятся на три типа: **контролируемые исключения** (checked) и **неконтролируемые исключения** (unchecked), к которым относятся ошибки (Errors) и **исключения времени** выполнения (RuntimeExceptions, потомок класса Exception).

Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса Exception (но не RuntimeException).

Обработка исключения может быть произведена с помощью операторов try…catch, либо передана внешней части программы. Например, метод может передавать возникшие в нём исключения выше по иерархии вызовов, сам его не обрабатывая. Вместо стандартного сообщения об ошибке будет выполняться блок catch, параметром которого является объект соответствующего исключению. В блок try при этом помещается тот фрагмент программы, где потенциально может возникнуть исключение. Одному try может соответствовать сразу несколько блоков catch с разными классами исключений.

Пример:

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
     int[] m = {-1,0,1};
        Scanner sc = new Scanner(System.in);
        try {
            int a = sc.nextInt();
            m[a] = 4/a;
            System.out.println(m[a]);
        } catch (ArithmeticException e) {
            System.out.println("Произошла недопустимая арифметическая
операция");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Обращение по недопустимому индексу
массива");
        }
    }
}
```

Если запустив представленную программу, пользователь введётся с клавиатуры 1 или 2, то программа отработает без создания каких-либо исключений. Если пользователь введёт 0, то возникнет исключение класса ArithmeticException, и оно будет обработано первым блоком catch. Если пользователь введёт 3, то возникнет исключение класса ArrayIndexOutOfBoundsException (выход за приделы массива), и оно будет обработано вторым блоком catch. Если пользователь введёт нецелое число, например, 3.14, то возникнет исключение класса InputMismatchException (несоответствие типа вводимого значение), и оно будет выброшено в формате стандартной ошибки, поскольку его мы никак не обрабатывали. Поскольку исключения построены на иерархии классов и подклассов, то сначала надо пытаться обработать более частные исключения и лишь затем более общие.

throw

Оператор throw используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса Throwable, который можно либо получить как параметр оператора catch, либо создать с помощью оператора new. Ниже приведена общая форма оператора throw.

throw ОбъектТипаThrowable;

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор throw возбуждает исключительную ситуацию, после чего-то же исключение возбуждается повторно — на этот раз уже кодом перехватившего его в первый раз раздела catch.

```
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
}
catch (NullPointerException e) {
System.out.println("caught inside demoproc");
throw e;
} }
```

```
public static void main(String args[]) {
try {
demoproc();
}
catch(NulPointerException e) {
System.out.println("recaught: " + e);
}
} }
```

Для задания списка исключений, которые могут возбуждаться методом, используется ключевое слово **throws**. Если метод в явном виде (т.е. с помощью оператора throw) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе throws в объявлении этого метода. Синтаксис определения метода:

```
тип имя_метода(список аргументов) throws список_исключений {}
```

Для того, чтобы мы смогли оттранслировать этот пример, нам придется сообщить транслятору, что procedure может возбуждать исключения типа IllegalAccessException и в методе main добавить код для обработки этого типа исключений :

```
class ThrowsDemo {
     static void procedure() throws IllegalAccessException {
          System.out.println(" inside procedure");
          throw new IllegalAccessException("demo");
     }
     public static void main(String args[]) {
          try {
          procedure();
          }
          catch (IllegalAccessException e) {
               System.out.println("caught " + e);
          }
     }
}
```

Необязательным добавлением к блокам try…catch может быть блок **finally**. Помещенные в него команды будут выполняться в любом случае, вне зависимости от того, произошло ли исключение или нет. При том, что при возникновении необработанного исключения оставшаяся после генерации этого исключения часть программы — не выполняется. Например, если исключение возникло в процессе каких-то длительных вычислений, в блоке finally можно показать или сохранить промежуточные результаты.

№18. СВОЙСТВА, МЕТОДЫ И СОБЫТИЯ КЛАССА: TFORM. наверх

Форма - это главный элемент разрабатываемого приложения, на котором располагаются другие элементы.
Основные свойства формы:
Caption — название окна, отображающееся в строке заголовка.
Name — имя формы (по умолчанию Form1, Form2,...).
Width — ширина формы в пикселях.
Height — высота формы в пикселях.
Left — координата формы относительно левой стороны экрана.
Top — координата формы относительно верхней стороны экрана.
Position — стартовая позиция окна.
Icon — иконка в строке заголовка.
Color — цвет фона формы.
AlphaBlend — использовать ли прозрачность формы.
AlphaBlendValue — степень прозрачности формы (0-прозрачна польностью, 255-непрозрачна).
BorderStyle — тип границы, обрамляющей форму.

Значения BorderStyle:
bsDialog — форму можно только переместить и закрыть, кнопки свернуть/развернуть отсутствуют.
bsNone — нет рамки, кнопок max/min, закрыть и оконного меню.
bsSingle — форму можно свернуть и развернуть, но нельзя изменить размер формы перетаскиванием за края.
bsSizeable — размер формы можно изменять, используется по умолчанию.
bsSizeToolWin — форма отображается аналогично Sizable, отсутствуют кнопки max/min, а текст в заголовке формы выводиться уменьшенным шрифтом.
bsToolWindow — Форма отображается аналогично предыдущему свойству, с тем отличием, что размер формы нельзя будет изменить перетаскиванием за края.

Основные события формы:
onCreate — происходит при создании формы.
onShow — происходит при показе формы на экран (например сворачивание и разворачивание).
onActivate — происходит при активации формы.
onPaint — происходит при перерисовке формы.
onResize — происходит при изменении размера формы.
onCloseQuery — происходит при закрытии формы нажатием на кнопку закрыть.
onClose — происходит при закрытии формы, не обязательно кнопкой закрыть.
onClick — происходит при клике по форме.
onDblClick — происходит при двойном клике по форме.
onMouseMove — происходит при движении мышкой по форме.
onMouseDown — происходит при нажатии кнопки мыши без отпускания.
onMouseUp — происходит при отпускании кнопки мыши.
Основные методы формы:
Create – создание экземпляров форму.
Show – отображает форму в немодальном режиме, при этом свойство Visible устанавливается в значение True, а сама форма переводится на передний план.
Hide - скрывает форму, устанавливая ее свойство Visible в значение False.
Close – используется для закрытия формы. Процедура Close не уничтожает созданный экземпляр формы, и форма может быть снова вызвана на экран, в частности, с помощью методов *Show* или *ShowModal***.**
ShowModal - Если установить диалоговый стиль формы, то она не становится модальной и позволяет пользователю переходить в другие окна приложения. Для запуска формы, в том числе любой диалоговой, в модальном режиме следует использовать метод *ShowModal*. Таким образом, стиль определяет внешний вид формы, но не ее поведение.
Show – позволяет вызвать форму не в модальном варианте. После создания такой формы без ее закрытия можно перейти к главной форме
Release, **Free** или **Destroy –** уничтожение формы, после чего работа с этой формой становится невозможна, и любая попытка обратиться к ней или ее компонентам вызовет исключение (ошибку). Необходимость уничтожения формы может возникнуть при оформлении заставок или при разработке больших приложений, требующих экономии оперативной памяти. Предпочтительным методом удаления формы считается метод Free, поскольку он предварительно проверяет возможность удаления.
ScrollInView - программное управление полосами прокрутки;
Cascade - предназначен для организации многодокументных приложений.

№19. СВОЙСТВА, МЕТОДЫ И СОБЫТИЯ КЛАССА: TIBTABLE. наверх

Компонент **TIBTable** реализует все возможности стандартного компонента .
Использование компонента TTable предоставляет доступ к одной таблице базы данных. Для этой цели наиболее часто используются следующие ***свойства***:

- ***Active*** - указывает, открыта (true) или нет (false) данная таблица.
- ***DatabaseName*** - имя каталога, содержащего искомую таблицу, либо псевдоним (alias) удаленной БД (псевдонимы устанавливаются с помощью утилиты конфигурации BDE, описание которой присутствует во многих источниках, посвященных продуктам Borland, либо с помощью SQL Explorer, вызываемого с помощью пункта меню Database/Explore). Это свойство может быть изменено только в случае, если таблица закрыта (ее свойство Active равно false), например:

```
Table1->Active = false;
Table1->DatabaseName = "DB_NAME"
Table1->Active = true;
```

- ***TableName*** - имя таблицы.
- ***Exclusive*** - если это свойство принимает значение true, то никакой другой пользователь не может открыть таблицу, если она открыта данным приложением. Если это свойство равно false (значение по умолчанию), то другие пользователи могут открывать эту таблицу.
- ***IndexName*** - идентифицирует вторичный индекс для таблицы. Это свойство нельзя изменить, пока таблица открыта.
- ***MasterFields*** - определяет имя поля для создания связи с другой таблицей.
- ***MasterSource*** - имя компонента TDataSource, с помощью которого TTable будет получать данные из связанной таблицы.
- ***ReadOnly*** - если это свойство равно true, таблица открыта в режиме "только для чтения". Нельзя изменить свойство ReadOnly, пока таблица открыта.
- ***Eof***, ***Bof*** - эти свойства принимают значение true, когда указатель текущей записи расположен на последней или соответственно первой записи таблицы.
- ***Fields*** - массив объектов TField. Используя это свойство, можно обращаться к полям по номеру, что удобно, когда заранее неизвестна структура таблицы:

```
Edit1->Text=Table1->Fields[2]->AsString;
```

Наиболее часто при работе с компонентом TTable используются следующие ***методы***:

- ***Open*** и ***Close*** устанавливают значения свойства Active равными True и False соответственно.
- ***Refresh*** позволяет заново считать набор данных из БД.
- ***First***, ***Last***, ***Next***, ***Prior*** перемещают указатель текущей записи на первую, последнюю, следующую и предыдущую записи соответственно, например:

```
Table1->First();
while (!Table1->Eof)
{
//что-то делаем...
Table1->Next();
};
```

- ***MoveBy*** перемещает указатель на указанное число строк (оно может быть и отрицательным) в пределах таблицы
- ***Insert***, ***Edit***, ***Delete***, ***Append*** - переводят таблицу в режимы вставки записи, редактирования, удаления, добавления записи соответственно.
- ***Post*** - осуществляет физическое сохранение измененных данных. Например:

```
Table2->Insert();
Table2->Fields[0]->AsInteger = 100;
Table2->Fields[1]->AsString =Edit1->Text;
Table2->Post();
```

- ***Cancel*** - отменяет внесенные изменения, не сохраненные физически.
- ***FieldByName*** - предоставляет возможность обращения к данным в полях по имени поля:

```
S=Table1->FieldByName("area")->AsString;
```

- ***SetKey*** переключает таблицу в режим поиска.
- ***GotoKey*** начинает поиск строки, значение Fields[n] которой равно выбранному, где n - номер колонки таблицы, начиная с 0:

```
Table1->SetKey();
Table1->Fields[0]->AsString=Edit1->Text;
Table1->GotoKey();
```

- ***SetRangeStart***, ***SetRangeEnd***, ***ApplyRange*** позволяют выбрать нужные строки на основе диапазона значений какого-либо поля.

```
Table1->SetRangeStart();
Table1->Fields[0]->AsString = Edit1->Text;
Table1->SetRangeEnd();
Table1->Fields[0]->AsString = Edit2->Text;
Table1->ApplyRange();
```

- ***FreeBookmark***, ***GetBookmark***, ***GotoBookmark***- позволяют создать помеченную строку в таблице и затем вернуться к ней позже. Методы Bookmark используют класс TBookmark. Метод GetBookmark устанавливает закладку на текущей строке таблицы. GotoBookmark осуществляет перемещение в таблице к строке, ранее отмеченной закладкой. Метод FreeBookmark используется для уничтожения объекта типа TBookmark:

```
TBookmark Marker =Table1->GetBookmark();
Table1->GotoBookmark(Marker);
Table1->FreeBookmark(Marker);
```

События компонента TTable позволяют строить и контролировать поведение приложений БД. Например, событие BeforePost наступает перед вставкой или изменением записи, событие AfterPost - после сохранения вставленной или измененной записи, событие AfterDelete - после удаления записи и т.д.

Дополнительно к свойствам и методам стандартного TTable, стоит обратить внимание на следующие:

При выборе таблицы (свойство TableName) ***свойства:***

```
TIBTableType = (ttSystem, ttview);
TIBTableTypes = set of TIBTableType;
        property TableTypes: TIBTableTypes;
```

type — определяет, какие таблицы доступны для выбора:

ttsystem — доступны системные таблицы и просмотры;

ttview — доступны определенные пользователем просмотры.

При открытии набора данных упорядочивание записей осуществляется в соответствии со значением свойства: property Defaultlndex: Boolean; При значении True записи располагаются в порядке, определяемом первичным индексом таблицы БД.

Во время выполнения, свойство property Exists: Boolean позволяет определить, существует ли в базе данных таблица, имя которой определено свойством TableName.

Метод

```
procedure GotoCurrent(Table: TIBTable)
```

синхронизирует курсоры текущего набора данных и набора данных компонента, заданного параметром Table.

№20. СИСТЕМА ПЕРЕДАЧИ ДАННЫХ КОМПЬЮТЕРНОЙ СЕТИ. ОСНОВНЫЕ ПОНЯТИЯ И ТЕХНОЛОГИИ. наверх

Компьютерная сеть — совокупность трёх и более компьютеров, объединённых каналами передачи данных и коммутирующими устройствами (узлами сети), обеспечивающими обмен сообщениями между всеми устройствами.

Технологии передачи данных в своей работе используют (в зависимости от конкретной их реализации) различные физические интерфейсы.

Физический интерфейс - это конечный порт подключения (разъем с группой электрических контактов). Например - интерфейс сетевой карты компьютера. А пара портов, соединенная с помощью разъемов и кабеля называется *линией (каналом) передачи данных.*

Логический интерфейс - это набор правил (протокол), который определяет саму логику обмена данными между связанными линией (сетью) устройствами.

Обязательным условием для успешной реализации любой из технологий передачи данных является присутствие в потоке данных дополнительного компонента - *протокола передачи*.

Протокол передачи на логическом уровне представляет собой набор правил, которые определяют обмен данными между различными приложениями или устройствами. Эти правила задают единый способ передачи сообщений и обработки ошибок передачи. На физическом уровне протокол это - набор служебных данных, прикрепляющихся к основным пакетам (кадрам) информации, без которых просто невозможно эффективное взаимодействие в сети.

Современные технологи и методы передачи данных, в большинстве случаев, основаны на клиент-серверном взаимодействии.

Клиент - это модуль (программа, служба, отдельный компьютер), служащий для формирования и передачи сообщений (запросов) к ресурсам удаленного устройства (серверу), с последующим приемом результатов от него и передачей их соответствующим приложениям на клиенте.

Сервер это - модуль (программа, служба), который постоянно ожидает прихода из сети запросов от клиентов и обслуживающий (с участием локальной ОС) эти запросы.

Клиент-серверная составляющая, которая предоставляет доступ к какому-то ресурсу компьютера через сеть, называется *сетевой службой*.

Современные технологии передачи данных связаны с их преобразованием (кодированием). Благодаря этому предотвращаются ошибки передачи данных (за счет уверенного распознавания сигнала принимающей стороной) и увеличивается скорость передачи данных (за счет более высокой плотности полезной информации в потоке).

№21. МОДЕЛЬ СЕТЕВЫХ ВЗАИМОДЕЙСТВИЙ OSI. наверх

ISO – International Standart Organisation создала модель сетевого взаимодейтсвия OSI (Open System Interconnection). Модель получилась семиуровневой:

7) Application(прикладной) – уровень приложений – описывает сетевой сервис: удаленный запуск задания, передача файлов, e-mail (HTTP, FTP, SMTP).

6) Presentation(представительский) – перевод данных в единый сетевой формат(ASCII, EBCDIC, JPEG).

5) Session(сеансовый) - сеансовый уровень – логические связи между узлами. Механизм определения прав доступа к ресурсам сети (иногда здесь реализуется кодировка) (RPC, PAP).

4) transport(транспортный) - определяет соединение между узлами, критерий оптимальной передачи.(TCP, UDP).

3) network(сетевой) - определении пути, маршрутизация (IPv4, IPv6).

2) data link(канальный) – определения метода доступа к сети (PPP, IEEE 802.2, L2TP, ARP).

1) physical(физический) – физический уровень - вопросы трансляции цифрового сигнала в сигнал, который передается по сети. Выбор способа модуляции (DSL).

Уровни 7,6,5 –уровни верхней группы, их назначение – определение семантики. Эти уровни реализуются в конкретных сетевых OS.

Уровни 4,3,2,1- транспортная сеть (оперирует доставкой информации). Организуют стек транспортных протоколов.

Любой протокол модели OSI должен взаимодействовать либо с протоколами своего уровня, либо с протоколами на единицу выше и/или ниже своего уровня. Взаимодействия с протоколами своего уровня называются горизонтальными, а с уровнями на единицу выше или ниже — вертикальными. Любой протокол модели OSI может выполнять только функции своего уровня и не может выполнять функций другого уровня, что не выполняется в протоколах альтернативных моделей.

Каждому уровню с некоторой долей условности соответствует свой операнд — логически неделимый элемент данных, которым на отдельном уровне можно оперировать в рамках модели и используемых протоколов: на физическом уровне мельчайшая единица — бит, на канальном уровне информация объединена в кадры, на сетевом — в пакеты (датаграммы), на транспортном — в сегменты. Любой фрагмент данных, логически объединённых для передачи — кадр, пакет, датаграмма — считается сообщением. Именно сообщения в общем виде являются операндами сеансового, представительского и прикладного уровней.

К базовым сетевым технологиям относятся физический и канальный уровни.

№22. КЛИЕНТ-СЕРВЕРНАЯ МОДЕЛЬ РАСПРЕДЕЛЕННЫХ СЕТЕВЫХ ПРИЛОЖЕНИЙ. наверх

Использование технологии " *клиент – сервер* " предполагает наличие некоторого количества компьютеров, объединенных в *сеть*, один из которых выполняет особые *управляющие* функции (является сервером сети).

Так, архитектура " клиент – сервер " разделяет функции приложения пользователя (называемого клиентом) и сервера. Приложение-клиент формирует запрос к серверу, на котором расположена БД, на структурном языке запросов SQL. Удаленный сервер принимает запрос и переадресует его SQL-серверу БД. *SQL-сервер* – специальная программа, управляющая удаленной базой данных. SQL-сервер обеспечивает интерпретацию запроса, его выполнение в базе данных, формирование результата выполнения запроса и выдачу его приложению-клиенту. При этом ресурсы клиентского компьютера не участвуют в физическом выполнении запроса; клиентский компьютер лишь отсылает запрос к серверной БД и получает результат, после чего интерпретирует его необходимым образом и представляет пользователю. Так как клиентскому приложению посылается результат выполнения запроса, по сети "путешествуют" только те данные, которые необходимы клиенту. В итоге снижается нагрузка на сеть. Поскольку выполнение запроса происходит там же, где хранятся данные (на сервере), нет необходимости в пересылке больших пакетов данных. Кроме того, SQL-сервер, если это возможно, оптимизирует полученный запрос таким образом, чтобы он был выполнен в минимальное время с наименьшими накладными расходами.

Все это повышает быстродействие системы и снижает время ожидания результата запроса. При выполнении запросов сервером существенно повышается степень безопасности данных, поскольку правила целостности данных определяются в базе данных на сервере и являются едиными для всех приложений, использующих эту *БД*. Таким образом, исключается возможность определения противоречивых правил поддержания целостности. Мощный аппарат транзакций, поддерживаемый *SQL*-серверами, позволяет исключить одновременное изменение одних и тех же данных различными пользователями и предоставляет возможность откатов к первоначальным значениям при внесении в *БД* изменений, закончившихся аварийно.

Рассмотрим, как выглядит разграничение функций между сервером и клиентом.

- Функции приложения-клиента:
  - Посылка запросов серверу.
  - Интерпретация результатов запросов, полученных от сервера.
  - Представление результатов пользователю в некоторой форме (интерфейс пользователя).
- Функции серверной части:
  - Прием запросов от приложений-клиентов.
  - Интерпретация запросов.
  - Оптимизация и выполнение запросов к БД.
  - Отправка результатов приложению-клиенту.
  - Обеспечение системы безопасности и разграничение доступа.
  - Управление целостностью БД.
  - Реализация стабильности многопользовательского режима работы.

Рассмотрим основные достоинства данной архитектуры:

- Существенно уменьшается сетевой трафик.
- Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть), а, следовательно, снижаются требования к аппаратным мощностям клиентских компьютеров.
- Наличие специального программного средства – SQL-сервера – приводит к тому, что существенная часть проектных и программистских задач становится уже решенной.
- Существенно повышается целостность и безопасность БД.

К числу недостатков можно отнести более высокие финансовые *затраты на аппаратное и программное обеспечение*, а также то, что большое количество клиентских компьютеров, расположенных в разных местах, вызывает определенные *трудности со своевременным обновлением клиентских приложений* на всех компьютерах-клиентах.

№23. ЗАДАЧА АУТЕНТИФИКАЦИИ И ПЕРСОНАЛИЗАЦИИ ПОЛЬЗОВАТЕЛЕЙ ИНФОРМАЦИОННОЙ СЕТИ. наверх

Средства аутентификации и персонализации относятся к категории классических средств по управлению информационной безопасностью информационных сетей. Эти средства включают в себя определение, создание, изменение, удаление и аудит пользовательских учетных записей.

Обычный метод аутентификации в Web — это требование к посетителям предоставить уникальное имя пользователя и пароль. Аутентификация используется для разрешения или запрещения доступа к определенным страницам или ресурсам. Аутентификация может быть необязательной либо использоваться для других целей, например, для персонализации.

Задача аутентификации.
Один из способов решения задачи аутентификации – это создать специальную форму на HTML – странице и в виде диалога предложить пользователю ввести уникальное имя и пароль.

Пример такой формы (фрагмент кода):

```
<form action=hello.php>
        Имя: <input type=text name=login> <br>
        Пароль: <input type=password name=pass> <br>
        <input type=submit va1ue = Отправить><br>
</form>
```

Далее, браузер обратится к сценарию hello.php и передаст их в строке параметров. Затем, с помощью PHP-сценария производится трансляция и анализ введенных значений.

Независимо от того, каким методом — **GET** или **POST** — воспользовался браузер, **PHP** сам определяет, какой метод был задействован.

Все данные из полей формы **PHP** помещает в глобальный массив **$_REQUEST**.

Значение поля login будет храниться в $_REQUEST['login'], а значение поля pass — в $_REQUEST['pass']. Чтобы можно было как-то разделить **GET**-параметры от **POST**-данных, **PHP** также создает массивы **$_GET** и **$_POST**, заполняя их соответствующими значениями. Массив **$_REQUEST** представляет собой объединение этих двух массивов.

Пример такого сценария (фрагмент кода):

```
<?php
        if ($_REQUEST['login']=="root" && $_REQUEST['pass']=="Z10N0101")
        {
                echo "Доступ открыт для пользователя $_REQUEST[login]";
        }
                else
        {
                    echo "Доступ закрыт! Для исправления ошибки нужно вернуться <a
href=index.html>назад</a>";
        }
?>
```

Если при вводе данных будет совершена ошибка, например, неправильно введен пароль, то доступ будет закрыт. Как и в любом другом языке, конструкция **else** может опускаться. В этом случае при получении ложного значения просто ничего не делается.

Вообще говоря, для хранения паролей лучше применить базу данных. Если планируется хранить и производить поиск в более чем 100 элементах, следует отдать предпочтение базе данных.
Использование базы данных для хранения имен и паролей посетителей позволяет быстро проводить аутентификацию множества пользователей. Это также упрощает создание сценария для добавления и удаления пользователей, а также дает возможность пользователям изменять свои пароли.

Задача персонализации.
Персонализация веб-страниц позволяет кардинально изменить отношение сайта к своим посетителям. В результате не только пользователь будет «общаться» с веб-страницей, но и сам сайт будет обращаться к любому, попавшему на страницу, как к конкретному человеку – персонально.

После первого и всех последующих посещений сайт запоминает информацию, связанную как с самим посетителем, так и с действиями, которые им проводились. При каждом новом заходе на страницу человека, который был здесь раньше, сайт моментально подстраивается под его интересы и запросы.

Два основных метода решения такой задачи:

1. Основанный на **правилах**.

Это практика использования исторических данных, поведенческих данных, и данных об окружающей среде для генерации на сайте уникального предложения, основанного на предопределенных заранее правилах.

Пример: Если пользователь пришел по запросу «ноутбуки бу», на главную страницу выводим ноутбуки, стоимостью до 200 долл.

2. Основанный на **алгоритмах**

Это использование математических систем для мониторинга поведения посетителей, с последующей разработкой модели предсказания, чтобы выработать наиболее подходящий контент для каждого посетителя.

В отличие от стратегии основанной на правилах, такую систему до начала работы необходимо обучить, т.е. накопить достаточный для анализа объем данных о посетителях сайта.

Обучившись, система предсказывает и предлагает посетителям сайта наиболее эффективные продукты или услуги (учитывая историю, действия, которые он совершил на сайте, страницы, которые он просмотрел и другие факторы).

А если нет возможности решить такую задачу самостоятельно, то можно воспользоваться уже готовыми сервисами для персонализации сайтов, такими как **Personyze** ($59 - $299), **Monoloop** (free - $2 500), **Vero**($99) и другими.

№ 24. БАЗОВЫЕ ОПЕРАТОРЫ ЯЗЫКОВ C/C++. УСЛОВНЫЙ (IF) И МНОЖЕСТВЕННОГО ВЫБОРА (SWITCH). ПОРЯДОК ВЫЧИСЛЕНИЯ МАТЕМАТИЧЕСКИХ ВЫРАЖЕНИЙ. ПРЕ- И ПОСТ- ИНКРЕМЕНТ И ДЕКРЕМЕНТ.

наверх

Оператором называется элементарная структурная единица программы.
Оператор предназначен как для записи алгоритмических действий по преобразованию данных, так и для задания порядка выполнения других действий.
Операторы выполняются в порядке их следования в программе.
Операторы отделятся друг от друга точкой с запятой.
Операторы делятся на:

- простые (не содержат в себе других операторов);
- составные (включают в себя один или несколько дополнительных операторов).

Оператор if выбирает один из двух вариантов последовательности вычислений.

Синтаксис условного оператора:

```
if (выражение)
  оператор_1;
else
  оператор_2;
```

Выражение должно быть скалярным и может иметь арифметический тип или тип указателя. Если оно не равно нулю (или не есть пустой указатель), то условие считается истинным и выполняется оператор_1. В противном случае выполняется оператор_2. В качестве операторов нельзя использовать описания и определения.

Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки (т.е. использовать составные операторы и блоки):

```
if (x > 0)
{
  x = -x;
  f(x*2);
}
else
{
  int i = 2;
  x *= i;
  f(x);
}
```

Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной. **Оператор выбора switch** используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия.

Оператор switch предназначен для организации выбора из множества различных вариантов. Синтаксис оператора следующий:

```
switch (переключающее_выражение)
{
  case константное_выражение_1: операторы_1;
  case константное_выражение_2: операторы_2;
  case константное_выражение_n: операторы_n;
  default:
     операторы;
}
```

Пример:

```
switch (ic)
  {
     case 0: cout << "ноль"; break;
     case 1: cout << "один,"; break;
     case 2: cout << "два "; break;
     case 3: cout << "три,"; break;
     case 4: cout << "четыре"; break;
```

```
    case 5: cout << "пять,"; break;
    case 6: case 7: cout << "шесть или семь,"; break;
    case 8: case 9: cout << "восемь или девять."; break;
    default: cout << "ОШИБКА!":
  }
```

Так как увеличение значения переменной представляет собой обычную операцию в программах, в C++ есть операция увеличения — двойной знак плюс (++), или **инкремент**. Операция увеличения обеспечивает быстрый способ прибавления единицы к значению переменной:

```
count = count + 1;
count++;
```

Оператор может стоять перед переменной, он называется **префиксным** оператором увеличения.

```
++variable;
```

Аналогично этому, оператор может стоять после переменной и называться **постфиксным** оператором увеличения.

```
variable++;
```

C++ трактует эти два оператора по-разному.

Например, рассмотрим *постфиксный* оператор увеличения:

```
current_count = count++;
```

Этот оператор присваивания указывает C++ присвоить текущее значение count переменной current_count, а потом увеличить текущее значение count:

```
current_count = count;
count = count + 1;
```

Теперь рассмотрим *префиксный* оператор увеличения:

```
current_count = ++count;
```

В этом случае оператор присваивания указывает C++ сначала увеличить значение count, а затем присвоить результат переменной current_coun:

```
count = count + 1;
current_count = count;
```

Двойной знак минус (--), или **декремент**, соответствует оператору уменьшения C++, который уменьшает значение переменной на 1. Как и в случае с операцией увеличения, C++ поддерживает префиксный и постфиксный операторы уменьшения.

№25. СТАТИЧЕСКАЯ И ДИНАМИЧЕСКАЯ ПАМЯТЬ, ОПЕРАТОР NEW/DELETE. (NEW[],DELETE[]). наверх

При выполнении любой программы, все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде, например 0x100 или 0x200.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или её зависание.

Если мы объявляем переменные статично, они остаются в памяти до того момента, как программа завершит свою работу, а после чего уничтожаются.

Такой подход может быть приемлем в простых примерах и несложных программах, которые не требуют большого количества ресурсов.

Поэтому, в большинстве языков, в том числе и C/C++, имеется понятие указателя. *Указатель* — это переменная, хранящая в себе адрес ячейки оперативной памяти, например **0x100**.

Мы можем обращаться, например, к массиву данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступна для других переменных.

Пример использования статических переменных:

```
#include <iostream>
using namespace std;

int main()
{
   int a; // Объявление статической переменной
   int b = 5; // Инициализация статической переменной b

   a = 10;
   b = a + b;
   cout << "b is " << b << endl;
   return 0;
}
```

Пример использования динамических переменных:

```
#include <iostream>
using namespace std;

int main()
{
   int *a = new int; // Объявление указателя для переменной типа int
   int *b = new int(5); // Инициализация указателя

   *a = 10;
   *b = *a + *b;

   cout << "b is " << *b << endl;

   delete b;
   delete a;

   return 0;
}
```

Рассмотрим **общий синтаксис указателей в C++.**
Выделение памяти осуществляется с помощью оператора *new* и имеет вид:
*тип_данных *имя_указателя = new тип_данных;*
Например: *int *a = new int;*

После удачного выполнения такой операции, в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа int.
Для разных типов данных выделяется разное количество памяти.

Инициализация значения, находящегося по адресу указателя:
*тип данных *имя_указателя = new тип_данных(значение)*.
В примере выше это *int *b = new int(5)*.

Для того, чтобы получить адрес в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом &. перед ним.
Например: вывести на экран адрес ячейки памяти, на который ссылается указатель b во втором примере *cout << "Address of b is " << &b << endl;*.

Для того, чтобы получить значение, которое находится по адресу, на который ссылается указатель, используется префикс *. Данная операция называется разыменованием указателя.
Например: во втором примере мы выводим на экран значение, которое находится в ячейке памяти *cout << "b is " << *b << endl*;

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку.
Например: **b = *a + *b;*.

Для того, чтобы освободить память используется оператор *delete*.

Пример освобождения памяти:

```
#include <iostream>
using namespace std;

int main()
{
   // Выделение памяти
   int *a = new int;
   int *b = new int;
   float *c = new float;

   // ... Любые действия программы

   // Освобождение выделенной памяти
   delete c;
   delete b;
   delete a;

   return 0;
}
```

При использовании оператора *delete* для указателя, знак * не используется.