перевод статьи взят с сайта

http://www.transl-gunsmoker.ru/2009/09/blog-post.html

Работа с указателями

Это перевод Addressing pointers. Автор: Rudy Velthuis.
Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover. — Charles Hoare

Указатели, вероятно, являются самыми плохо понимаемыми и страшными типами данных. Поэтому много программистов пытаются их избегать.

Но указатели очень важны. Даже в языках, которые явно не поддерживают указатели, или в которых использование указателей затруднено, указатели являются важными факторами "под капотом" языка. Я считаю, что для программиста понимание указателей является весьма важной вещью. Существует несколько подходов к пониманию указателей.

Эта статья была написана для тех, у кого есть проблемы с пониманием или использованием указателей. Она демонстрирует мою точку зрения на указатели в Delphi для Win32. Может быть, это не будет абсолютно точно во всех аспектах (например, память программы не является одним большим монолитным блоком), но этого более чем достаточно для практических целей. По-моему, таким образом указатели будет проще понять.

Содержание

Память (Memory)

Скорей всего, вы и так уже знаете, что я собираюсь написать в этом параграфе, но, наверное, прочитать его будет не лишним, т.к. он демонстрирует моё видение вещей, которое может отличаться от вашего.

Указатели - это переменные, которые указывают на другие переменные. Чтобы объяснить это, необходимо понять концепцию адреса памяти и концепцию переменной. Для этого я сначала грубо опишу компьютерную память (*).

Кратко говоря, компьютерная память может рассматриваться как один очень-очень длинный ряд байтов. Байт - это единица измерения количества информации, в стандартном виде байт считается равным восьми битам и может хранить одно из `256` различных значений (от `0` до `255`). В текущей `32`-х битной версии Delphi на память можно смотреть (за редкими исключениями) как на массив байт максимальным размером в `2` Гб (`2^31` байт). Что именно содержат эти байты - зависит от того, как интерпретировать их содержимое, т.е. от того, как их используют. Значение `97` может означать число `97`, или же букву `'a'`. Если вы рассматриваете вместе несколько байт, то вы можете хранить и большие значения. Например, в

2-х байтах вы можете хранить одно из 256*256 = 65536 различных значений и т.д.

Чтобы обратиться к конкретному байту в памяти (адресовать его), можно присвоить каждому байту номер, пронумеровав их числами от 0 и до 2147483647 (в предположении, что у вас есть 2 Гб — а даже если у вас их нет, то Windows попытается сделать так, чтобы вам казалось, что они у вас есть). Индекс байта в этом огромном массиве и называется его адресом.

Кто-то может сказать: байт - это наименьший кусок памяти, который можно адресовать.

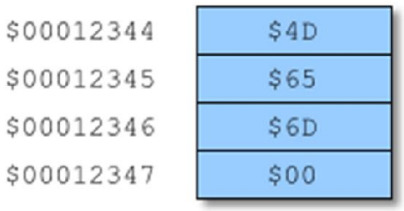
В действительности, память устроена сложнее. Например, существуют компьютеры, байт в которых не равен 8-ми битам, что означает, что они могут содержать больше или меньше 256 значений. Впрочем, для тех машин, на которых работает Delphi для Win32, байт всегда равен 8-ми битам. Память управляется и железом и программами, так что не вся видимая вам память может существовать (менеджеры памяти скрывают это от вас, выгружая память частями на жёсткий диск), но для целей этой статьи мы можем смотреть на память как на один большой блок памяти, разделённый для использования несколькими программами (прим.пер.: для более подробного ознакомления с архитектурой памяти в Windows - рекомендую эту короткую серию статей).

Переменные (Variables)

Переменная - это место из одного или нескольких байт в этом гигантском "массиве", из которого (места) вы можете читать или писать. Это место идентифицируется по *имени*, но также характеризуется своим *типом*, *значением* и *адресом*.

Когда вы объявляете переменную, компилятор резервирует кусочек памяти подходящего размера. Где именно будет лежать эта переменная - определяется компилятором и средой времени выполнения. Вы никогда не должны делать предположения о возможном точном месте переменной в запущенной программе.

Тип переменной определяет, как будет использоваться место в памяти. Тип определяет **размер**, т.е. сколько байт занимает место в памяти, а также **структуру** памяти. Например, на следующей диаграмме показан кусок памяти в 4 байта, начинающихся по адресу $00012344. Байты содержат значения $4D, $65, $6D и $00, соответственно.



Заметьте, что хотя я использовал адрес типа $00012344, в большинстве диаграмм это просто числа, взятые "от балды", которые просто помогают отличить одно место в памяти от другого. Они не отражают настоящие адреса памяти, т.к. эти адреса зависят от множества факторов и их нельзя предсказать заранее.

Тип определяет, как используются эти байты. Например, это может быть число типа *Integer* со значением 7169357 (что есть $006D654D), или же массив символов array[0..3] of AnsiChar, формирующий C-строку (т.е. *PChar*) 'Mem', или что-то совершенно иное, как множество, массив из отдельных байт, небольшая запись, *Single*, часть *Double* и т.д... Другими словами, смысл куска памяти переменной не известен, если только вы не знаете, какого типа (или типов) эта переменная (или переменные).

Адрес переменной - это адрес первого байта места хранения. Например, в диаграмме выше, в предположении, что у нас переменная типа *Integer*, её адрес будет $00012344.

Неинициализированные переменные

Память для переменных может использоваться многократно. Память для переменных обычно резервируется только на время, пока программа может обращаться к ним. Т.е. локальные переменные в процедуре или функции (я буду называть их одним словом: "подпрограммы") доступны только в то время, пока выполняется код подпрограммы. Поля объекта (которые тоже переменные) также доступны только пока объект "существует".

Если вы объявляете переменную, компилятор резервирует требуемое количество байт для этой переменной. Но содержание памяти для этих переменных может быть байтами, которые лежали там при вызове другой функции или процедуры. Другими словами, значение **неинициализированной** переменной (т.е. переменной, которой вы ещё не присвоили значения) *неопределено* (но не обязательно *неопределяемо* в принципе). Простой пример в виде консольной программы демонстрирует это:

```
1
2 program uninitializedVar;
3
4 {$APPTYPE CONSOLE}
5
6 procedure Test;
7 var
8   A: Integer;
9 begin
10  Writeln(A); // ещё не инициализирована
11  A := 12345;
12  Writeln(A); // инициализирована: 12345
13end;
14
15begin
16  Test;
17  Readln;
18end.
19
```

Первое отображаемое значение (значение неинициализированной переменной *A*) зависит от уже существующего содержания памяти, зарезервированной под *A*. В моём случае каждый раз значение было `2147319808` (`$7FFD8000`), но это число может быть совершенно другим на вашей машине. Значение не определено, потому что переменная не была инициализирована. В более сложных программах, особенно (но не только) с участием указателей, это частая причина для вылета программы или вывода неверных результатов. Присваивание инициализирует переменную A значением `12345` (`$00003039`), что и будет вторым выведенным значением.

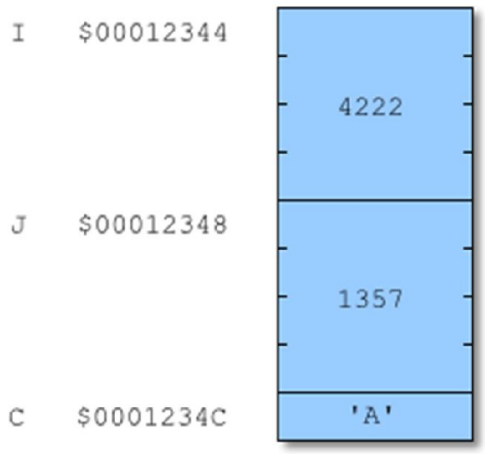
Указатели (Pointers)

Указатели также являются переменными. Но они содержат не символы или числа, а адреса памяти. Если вы рассматриваете память как огромный массив чисел, то *указатель будет одним элементом массива, в котором хранится индекс другого элемента массива*.

Например, пусть у нас есть следующие объявления и инициализация:

```
1
2var
3  I: Integer;
4  J: Integer;
5  C: Char;
6begin
7  I := 4222;
8  J := 1357;
9  C := 'A';
10
```

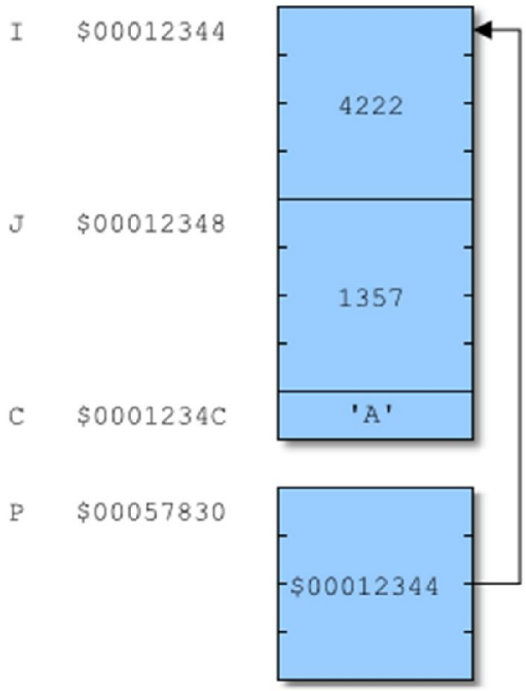
Предположим, что это дало нам такую компоновку памяти:



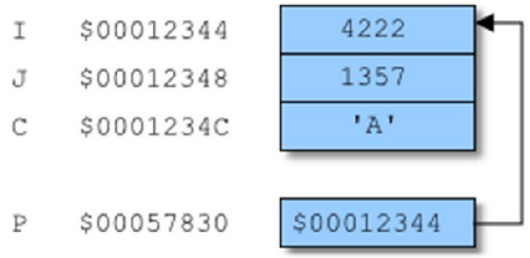
Теперь, после выполнения этого кода (предполагая что P - это указатель):

```
1P := @I;
```

Мы получаем:



В предыдущих диаграммах я всегда показывал все байты. Это обычно не является необходимым,



так что мы можем показать эту же ситуацию как:

Эта диаграмма более не показывает настоящих размеров (C выглядит так же, как и I или J), но этого достаточно, чтобы продемонстрировать, что происходит с указателями.

nil

Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end. — Henry Spencer

nil - это специальное значение указателя. Оно может быть присвоено любому указателю. *nil* означает пустой указатель (*nil* - это сокращение от Лат. *nihil*, что означает *ничего* или *ноль*; кое-кто расшифровывает *NIL* как аббревиатуру от *Not In List* - *не в списке*). Это означает, что указатель был определён (инициализирован, присвоен), но вы не должны пытаться получить значение, на которое он указывает (в языке C, *nil* называется *NULL* — см. цитату в начале секции).

Nil никогда не указывает на допустимую память, но т.к. это вполне конкретное значение, то подпрограммы могут сравнивать указатели с этим значением (например, используя функцию *Assigned()*). Нельзя проверить, является ли любое другое (не-*nil*) значение указателя допустимым. **Мусорный или неинициализированный указатель ничем не отличается от допустимого указателя** (см. ниже). Не существует способа их отличать друг от друга. Программная логика всегда должна гарантировать, что указатель либо допустим, либо равен *nil* (**).

В Delphi, *nil* имеет значение `0` (прим. пер.: т.е. `nil = Pointer(0)` или `0 = Integer(nil))` - т.е. он указывает на самый первый байт в памяти. Очевидно, что этот байт никогда не будет доступен для Delphi кода. Но обычно вы не должны рассчитывать на то, что указатель *nil* будет равен `0`, если только вы точно не знаете, что вы делаете. Числовое значение *nil* может быть изменено в следующих версиях Delphi по какой-то причине (***).

Типизированные указатели

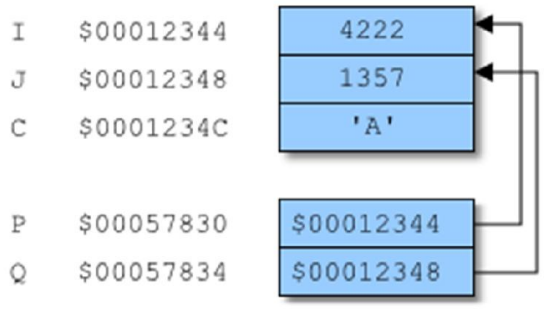
В простом примере выше *P* имеет тип *Pointer*. Это означает, что *P* содержит адрес, но вы не знаете, переменная какого типа лежит по этому адресу. Вот почему обычно используются типизированные указатели, т.е. указатель интерпретируется как указывающий на переменную (область памяти) определённого типа.

Предположим, что у нас есть ещё один указатель, *Q*:

```
1var
2  Q: ^Integer;
```

Q типа `^Integer`, что читается как "указатель на Integer" (мне сказали, что `^Integer` расшифровывается как `↑Integer`). Это означает, что *Q* - это не *Integer*, но вместо этого указывает на память, которая может быть использована как *Integer*. Если вы присвоите адрес *J* в *Q*, используя оператор взятия адреса `@` или эквивалентную функциональность псевдофункции *Addr*,

```
1Q := @J; // Q := Addr(J);
```



то тогда *Q* будет указывать на место по адресу `$00012348` (*Q* **ссылается** (**references**) на место памяти, занимаемое *J*). Но поскольку *Q* является **типизированным указателем**, то компилятор будет трактовать память, на которую указывает *Q*, как число типа *Integer*. *Integer* является **базовым типом** *Q*.

Хотя вы навряд ли увидите псевдофункцию *Addr* в реальном коде, она полностью эквивалентна `@`. Однако у `@` есть недостаток: если его применять к сложному выражению, то не всегда ясно, указатель чего берётся. *Addr* же, используя синтаксис функции, получается намного более читабельным, поскольку целевое выражение заключается в скобки:

```
P := @PMyRec^.Integers^[6];
Q := Addr(PMyRec^.Integers^[6]);
```

Прим. пер.: поэтому неплохо использовать скобки вместе с @, хотя это и не обязательно:

```
P := @(PMyRec^.Integers^[6]);
```

Присваивание с использованием указателей происходит немного иначе, чем при прямом присвоении переменной. Обычно для работы у вас будет только указатель. Если вы присваиваете значение обычной переменной, вы пишите что-то вроде:

```
J := 98765;
```

Это записывает число `98765` (`$000181CD`) в место памяти, занимаемое переменной *J*. Но чтобы получить доступ к этой памяти через указатель *Q*, вы должны работать *косвенно*, используя оператор `^`:

```
Q^ := 98765;
```

Это называется **разыменованием указателя**. Вы должны следовать по воображаемой "стрелочке" до места, на которое указывает *Q* (другими словами, до *Integer* по адресу `$00012348`) и сохранить там значение.

Для записей, синтаксис языка позволяет вам опускать оператор `^`, если код не теряет при этом своего смысла. Но лично я всегда явно указываю оператор для улучшения читабельности.

Обычно полезно определять типы для используемых в программе указателей. Например, вы не можете использовать `^Integer` при указании типа параметра подпрограммы, так что вам придётся объявить новый тип:

```
type
  PInteger = ^Integer;

procedure Abracadabra(I: PInteger);
```

Фактически, тип *PInteger* и некоторые другие часто используемые типы уже определены в библиотеке Delphi (модули *System*, *SysUtils* и *Types*). Начинать имя типов указателей с заглавной буквы P и следующим за ней именем типа, на переменную которого указывает указатель, является традицией, рекомендованной к выполнению. Если базовый тип указателя начинается с заглавной T, то T обычно опускается. Например:

```
type
  PByte = ^Byte;
  PDouble = ^Double;
  PRect = ^TRect;
  PPoint = ^TPoint;
```

Анонимные переменные

(прим. пер.: не путать с захваченными переменными в анонимных методах)

В предыдущих примерах переменные объявлялись только там, где они были необходимы. Иногда вы не знаете, понадобится ли вам переменная или как много переменных. Используя указатели, вы можете

создавать так называемые **анонимные переменные**. Вы можете попросить библиотеку языка выделить вам кусок памяти и вернуть указатель на него, используя псевдофункцию *New()*:

```
1var
2  PI: PInteger;
3begin
4  New(PI);
```

New() - это псевдофункция компилятора. Она резервирует память для базового типа *PI* и записывает адрес на эту память в указатель *PI*. У самой переменной здесь нет имени (т.е. она анонимная) - имя есть только у указателя на переменную. Получить доступ к такой переменной можно только используя указатель. Теперь вы можете присваивать ей значения, передавать её в подпрограммы, избавиться от неё, когда она станет вам не нужна, используя вызов *Dispose*(*PI*):

```
1PI^ := 12345;
2  ListBox1.Add(IntToStr(PI^));
3  // куча кода
4  Dispose(PI);
5end;
```


Вместо использования *New* и *Dispose* вы можете спуститься на уровень пониже и использовать *GetMem* и *FreeMem*. Но подпрограммы *New* и *Dispose* имеют несколько преимуществ: они осведомлены о типе указателя (прим. пер.: поэтому автоматически определяют размер памяти), а также инициализируют и освобождают содержимое участка памяти, если это необходимо. Так что рекомендуется всегда использовать *New* и *Dispose*, вместо *GetMem* и *FreeMem*, там, где это возможно.

Всегда гарантируйте, что каждый вызов *New()* будет иметь пару в виде вызова *Dispose()* с **тем же самым значением и типом указателя**, в противном случае память может быть освобождена неверно или не до конца.

Сейчас может быть не очевидно, чем же это лучше, чем объявлять переменную явно, но бывают ситуации, когда это полезно, обычно, если вы не знаете, как много переменных вам понадобиться. Подумайте об узлах в связанном списке (см. ниже) или о *TList*-е. *TList* хранит указатели, и если вы хотите иметь список значений *Double*, то вы просто вызываете *New()* для каждого значения и храните его в *TList*:

```
1
2var
3  P: PDouble;
4begin
5  while HasValues(SomeThing) do
6  begin
7    New(P);
8    P^ := ReadValue(SomeThing);
9    MyList.Add(P);
10   // и т.д...
```

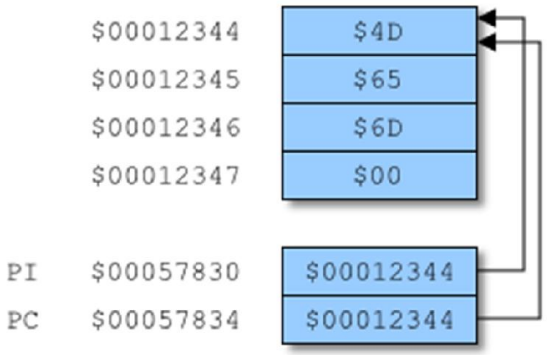
Конечно же, вам нужно будет потом вызывать *Dispose()* для каждого значения, когда список не будет больше нужен.

Используя анонимные переменные, легко показать, что типизированные указатели могут определять, как используется память. Два указателя разных типов, указывающие на одно и то же место в памяти, будут показывать разные значения:

```
1 program InterpretMem;
2
3 {$APPTYPE CONSOLE}
4
5 var
```

```
5    PI: PInteger;
6    PC: PAnsiChar;
7  begin
8    New(PI);
9    PI^ := $006D654D;      // Байты $4D $65 $6D $00
10   PC := PAnsiChar(PI);   // Теперь оба указателя указывают на одно место в памяти
11   Writeln(PI^);          // Печатаем число.
12   Writeln(PC^);          // Печатаем один символ ($4D).
13   Writeln(PC);           // Печатаем строку в стиле C (байты $4D $65 $6D $00
интерпретируются как PChar)
14   Dispose(PI);
15   Readln;
16 end.
17
```

PI заполняет память значением `$006D654D` (`7169357`). На диаграмме (напомню, что все адреса были придуманы мной):



PC указывает на ту же самую память (поскольку базовые типы указателей несовместимы, вы не можете просто так присвоить один указатель другому — вам нужно использовать преобразование типов). Но *PC* является указателем на *AnsiChar*, так что если вы берёте `PC^`, то вы получаете *AnsiChar* - один символ с ASCII значением `$4D` или `'M'`.

Вообще-то, *PC* - это особый случай, поскольку тип *PAnsiChar*, хотя он формально указывает на **символ** *AnsiChar*, трактуется специальным образом, немного иначе, чем остальные типы указателей. Я объясняю это в другой статье. *PC*, если он не разыменовывается, обычно трактуется как указатель на текст, заканчивающийся нулевым символом `#0`, поэтому `Writeln(PC)` покажет текст, сформированный байтами `$4D $65 $6D $00`, т.е. `'Mem'`.


Когда мне нужно подумать об указателях, и особенно о сложных ситуациях с ними, я обычно беру бумажку и ручку и рисую диаграммки типа тех, что вы видели выше. Я также даю переменным выдуманные адреса, если это нужно (не обязательно использовать все 32 бита, адреса типа `30000`, `40000`, `40004`, `40008` и `50000` тоже вполне подойдут).

Плохие указатели

Если использовать их правильно, указатели являются очень мощным и гибким средством. Но если вы сделаете ошибку - они могут стать большой проблемой. Это ещё одна причина, почему некоторые люди стараются избегать указателей. Некоторые из частых ошибок перечислены ниже.

Неинициализированные указатели

Указатели являются переменными, и как любые другие переменные они должны быть инициализированы перед использованием, либо присваиванием им другого указателя, либо используя подпрограммы типа

New или *GetMem*, например:

```
1
2 var
3   P1: PInteger;
4   P2: PInteger;
5   P3: PInteger;
6   I: Integer;
7 begin
8   I := 0;
9   P1 := @I;  // OK: используя оператор @
10  P2 := P1;  // OK: присвоением другого указателя
11  New(P3);   // OK: New
12  Dispose(P3);
13end;
14
```

Если вы просто объявите, скажем, указатель типа *PInteger*, но не проинициализируете его, то указатель, вероятно, будет содержать какие-то случайные байты, т.е. он фактически будет указывать на случайное место в памяти.

Если вы попробуете получить доступ к памяти по такому указателю, будут происходить плохие вещи. Если эта память не будет зарезервирована вашим приложением, то вы получите исключение Access Violation - вылет программы (program crash). Но если эта память будет частью вашей программы, то вы можете переписать данные, которые не должны меняться. Если эти данные используются в другой части вашей программы, то чуть позже, получаемые результаты выполнения вашей программы будут ошибочны. Такие ошибки чрезвычайно тяжело искать.

Поэтому, если вы вдруг увидели сообщение об ошибке типа Access Violation или другое сообщение об очевидном вылете (типа Invalid Pointer), вы, на самом деле, должны быть благодарны (ну если только эта ошибка не испортила ваш жёсткий диск ;) . Ваша программа вылетела, да, это плохо, но такие ошибки хотя бы легко отлаживать и исправлять. Но если ваша программа просто втихую выводит неправильные данные, проблема может быть намного хуже, и вы даже можете не заметить её, пока не станет слишком поздно. Вот почему вы должны использовать указатели с особым вниманием. Всегда дотошно проверяйте код на неинициализированные указатели.

Мусорные указатели

Мусорные указатели (stale pointers) - это указатели, которые когда-то были допустимыми, но теперь уже нет. Это может произойти, если память, на которую указывает указатель, была освобождена и/или повторно использована.

Один из частых случаев мусорного указателя - это когда память освобождается, но сам указатель ещё используется после этого. Для предотвращения этого некоторые программисты всегда устанавливают указатели в *nil*, после освобождения памяти. Они также проверяют на *nil*, прежде чем пытаться получить доступ к памяти. Другими словами, *nil* используется как своего рода флаг, отметка о недопустимости указателя. Это один из подходов, но не всегда самый лучший.

Другая частая ошибка: иметь более чем один указатель на один и тот же кусок памяти. Вы можете освободить память по одному указателю и даже об-*nil*-ить его, но другой указатель всё также будет содержать старое значение, указывающее на уже освобождённую память. Если вам повезёт, вы получите ошибку типа "Access Violation" или "Invalid pointer", но реальное поведение часто неопределено.

Третья, похожая проблема, - это указание на непостоянные (volatile) данные, т.е. данные, которые могут исчезнуть в любой момент. Например, частой грубой ошибкой является функция, возвращающая указатель на свои локальные переменные. Ведь как только мы выходим из подпрограммы, её локальные данные больше не существуют, а возвращённый вызывающей стороне указатель оказывается указывающим в никуда. Классический пример:

```
1
2function VersionData: PChar;
3var
4  V: array[0..11] of Char;
5begin
6  CalculateVersion(V);
7  Result := V;
8end;
9
```

V будет помещена в **процессорном стеке**. Это специальная часть памяти, которая используется для хранения локальных переменных и параметров вызываемых процедур, а также содержит важные данные типа адресов возврата для каждой вызванной подпрограммы. Результат функции указывает на *V* (*PChar* может указывать прямо на массив - см. статью, что я упоминал). Как только *VersionData* завершает выполнение, стек изменяется следующей вызванной подпрограммой, так что данные, вычисленные в *CalculateVersion* оказываются перезаписанными, а указатель указывает на это новое содержимое по всё тому же старому адресу.

Похожая проблема: указывание *PChar* на строку *String*, но это также обсуждалось в статье про *PChar*. Или использование указателя на элемент динамического массива (динамические массивы могут перемещаться по памяти, если их размер меняется вызовами *SetLength*) - вместо этого надо использовать просто индекс.

Использование неверного базового типа

Тот факт, что указатели могут указывать на любое место в памяти и что два указателя разных типов могут указывать на одно и то же место, фактически означает, что вы можете обращаться к одной памяти разными способами. Используя указатель на *Byte* (`^Byte`), вы можете изменять индивидуальные байты *Integer*-а или любого другого типа.

Но вы также можете что-то ошибочно записать или прочитать. Например, если вы получаете доступ к месту, которое хранит только *Byte*, с помощью указателя на *Integer*, вы можете записать `4` байта, среди которых только `1` байт является допустимым, а остальные три - просто смежные с ним, поскольку компилятор будет трактовать эти `4` байта подряд, как одно число типа `Integer`. Также, если вы читаете что-то с места расположения байта, вы можете прочитать слишком много:

```
1
2var
3  PI: PInteger;
4  I, J: Integer;
5  B: Byte;
6begin
7  PI := PInteger(@B);
8  I := PI^;
9  J := B;
10end;
11
```

J будет иметь правильное значение, потому что компилятор добавит код расширения одного байта до (`4`-х байтового) *Integer* с помощью обнуления старшей части *Integer*-а. Но *I* не будет иметь верного значения. Она будет содержать наш байт и ещё `3` каких-то байта, которые следуют за *B*, формируя этим некоторое неопределённое (мусорное) значение.

Указатели также позволяют вам установить значение переменной без присваивания его самой переменной. Это может сильно огорчать вас во время отладки. Вы знаете, что переменная содержит неверное значение, но не можете увидеть в коде, где же это значение было присвоено, потому что значение было установлено через указатель (прим. пер.: для этого могут использоваться точки останова на память).

Владельцы и забытая память (Owners and orphans)

Указатели не только могут иметь различные базовые типы, но и различную семантику владения. Если вы выделяете память, используя *New* или *GetMem* или любую другую более специализированную подпрограмму, вы являетесь **владельцем** этой памяти. Лучше всего, если вы будете держаться за эту память, заныкав указатель на неё в надёжное место. Указатель - это ваш единственный способ получить доступ к этой памяти, и если он будет утерян - вы не сможете ни прочитать данные, ни освободить их. Одно из общих правил: тот, кто выделяет память, обязан её и освободить, так что это ваша обязанность. Хорошо спроектированные программы всегда следуют этому правилу.

Понять правила владения очень важно. Кто владеет памятью - тот её и освобождает. Вы можете делегировать (передать) эту задачу кому-то ещё, но вы должны убедиться, что задача будет выполнена верно.

Частая ошибка: использовать указатель для выделения памяти, а затем снова использовать этот же указатель для выделения другой памяти или присвоить ему другой указатель. Указатель, который содержал адрес старого выделенного блока памяти, начинает указывать на новый блок памяти, а старый адрес оказывается потерян навсегда. Не существует никакого разумного способа получить местоположение первого выделенного блока памяти. Память оказывается **забыта**. Никто не может получить к ней доступ и никто не может её очистить. Это приводит к образованию так называемых **утечек памяти**.

Вот простой пример, взятый (с разрешения автора) из групп обсуждений Borland:

```
1
2 var
3   bitdata: array of Byte;
4   pbBitmap: Pointer;
5 begin
6   SetLength(bitdata, nBufSize);
7   GetMem(pbBitmap, nBufSize);
8   pbBitmap := Addr(bitdata);
9   VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, @pbBitmap, nBufSize);
10
```

Ну, вообще-то этот код делает несколько странных вещей. *SetLength* выделяет байты для *bitdata*. По какой-то причине программист потом использует *GetMem* для выделения такого же количества байт для *pbBitmap*. Но затем он немедленно *присваивает pbBitmap другой адрес*, что приводит к тому, что только что выделенная *GetMem*-ом память становится недоступной любому коду (единственным способом добраться до неё была *pbBitmap*, но он больше на неё не указывает). Другими словами, у нас есть утечка памяти.

Фактически тут есть и другие ошибки. *bitdata* - это динамический массив, и взятие адреса *bitdata* берёт адрес указателя на данные массива, вместо адреса самих данных (см. ниже, *динамические массивы*). Также, поскольку *pbBitmap* уже является указателем, неправильно использовать на него оператор `@`.

Более правильный код выглядел бы так:

```
1
2  var
3    bitdata: array of Byte;
4    pbBitmap: Pointer;
5  begin
6    if nBufSize > 0 then
7    begin
8      SetLength(bitdata, nBufSize);
9      pbBitmap := Addr(bitdata[0]);
10     VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, pbBitmap, nBufSize);
11   end;
12
```

Или даже так:

```
var
  bitdata: array of Byte;
begin
  if nBufSize > 0 then
  begin
    SetLength(bitdata, nBufSize);
    VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, @bitdata[0], nBufSize);
  end;
```

Это может показаться тривиальной проблемой, но в более сложном коде легко допустить подобную ошибку.

Заметим, что указатель не обязан владеть памятью. Указатели часто используются для движения по массиву (см. ниже) или для получения доступа к части структуры. Если вы не выделяете для указателя память, то нет никакой причины не изменять его. При этом указатель используется как временная переменная, которую можно в любой момент выбросить, не заботясь об освобождении памяти.

Арифметика указателей и массивы

You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time. — Bertrand Meyer

Delphi позволяет производить над указателями несколько простых действий. Во-первых, конечно же, вы можете присваивать им значения и сравнивать их на равенство (`if P1 = P2 then`) или неравенство. Но вы также можете увеличивать или уменьшать их, используя псевдофункции *Inc* и *Dec*. Приятным моментом при этом является то, что эти функции **учитывают размер базового типа указателя**. Пример (заметьте, что я руками присвоил указателю фальшивый адрес. Пока я не попытаюсь получить по нему доступ, всё будет в порядке):

```


program PointerArithmetic;

{$APPTYPE CONSOLE}

uses
  SysUtils;

procedure WritePointer(P: PDouble);
begin
  Writeln(Format('%8p', [Integer(P)]));
end;

var
  P: PDouble;
begin
  P := Pointer($50000);
  WritePointer(P);
  Inc(P);
  WritePointer(P);
  Inc(P, 6);
  WritePointer(P);
  Dec(P, 4);
  WritePointer(P);
  Readln;
end.


```

Вывод программы:

```
50000
50008
50038
50018
```

Применение этого: это способ предоставить последовательный доступ к элементам массивов. Поскольку (одномерные) массивы содержат последовательные элементы одного типа (т.е. если элемент расположен по адресу *N*, тогда следующий элемент расположен по адресу *N + SizeOf(element)*), то можно использовать этот сценарий для прохода по массиву в цикле (****). Вы начинаете с адреса первого элемента массива, что-то с ним делаете. На следующей итерации цикла вы увеличиваете указатель, так что он начинает указывать на второй элемент и т.д.:

```
1
2
3  program IterateArray;
4
5  {$APPTYPE CONSOLE}
6
7  var
8    Fractions: packed array[1..8] of Double;
9    I: Integer;
10   PD: ^Double;
11 begin
12   // Заполняем массив случайными значениями
13   Randomize;
14   for I := Low(Fractions) to High(Fractions) do
15     Fractions[I] := 100.0 * Random;
16   // Получаем доступ через указатель
17   PD := @Fractions[Low(Fractions)];
18   for I := Low(Fractions) to High(Fractions) do
19   begin
20     Write(PD^:9:5);
21     Inc(PD);        // Указываем на следующий элемент
22   end;
23   Writeln;
24   // Обычный доступ по индексу
25   for I := Low(Fractions) to High(Fractions) do
26     Write(Fractions[I]:9:5);
27   Writeln;
28   Readln;
29 end.
30
31
```

Увеличение указателя немного быстрее, чем умножение индекса на размер элемента и суммирование с базовым адресом на каждой итерации.

В реальности же, разница между обоими способами незначительна, если вообще заметна. Во-первых, у современных процессоров есть специальные способы для адресования при типичных случаях использования индекса. Во-вторых, компилятор всё равно обычно оптимизирует использование индекса в использование указателя, если это возможно. А в примере выше, незначительная разница и вовсе съедается выполнением такой сложной функции как Write().

Как вы можете видеть в примере выше, вы можете легко забыть увеличить указатель в цикле. И вы всё равно должны использовать либо цикл *for-to-do*, либо какой-то другой способ для организации цикла и условия выхода (с ручным сдвигом и сравнением). Код с использованием указателей обычно труднее поддерживать. И поскольку подход с указателями ничуть не быстрее (ну за исключением очень маленьких циклов), я бы избегал подобного кода в Delphi. Делайте так только если вы прогнали программу под профайлером и считаете, что доступ с указателем действительно будет выигрышным.

Указатели на массивы

Но иногда у вас нет массива, а есть только указатель для доступа к памяти. Функции Windows API часто возвращают данные в буферах, которые содержат массивы записей определённого размера. Но даже в этом случае, вероятно, будет проще преобразованием типа сделать буфер указателем на массив, чем использовать Inc или Dec. Пример:

```
1
2
3 type
4   PIntegerArray = ^TIntegerArray;
5   TIntegerArray = array[0..65535] of Integer;
6 var
7   Buffer: Pointer;     // трактуется как packed array of Integer;
8   PInt: PInteger;
9   PArr: PIntegerArray;
10 ...
11  // С использованием арифметики указателей:
12  PInt := Buffer;
13  for I := 0 to Count - 1 do
14  begin
15    Writeln(PInt^);
16    Inc(PInt);
17  end;
18  // Используя преобразование типа, указатель на массив и индексирование:
19  PArr := PIntegerArray(Buffer);
20  for I := 0 to Count - 1 do
21    Writeln(PArr^[I]);
22 ...
23end;
24
25
```

Delphi 2009

В Delphi 2009 и выше арифметика указателей, помимо типа *PChar* (и *PAnsiChar* и *PWideChar*), также применима и для других типов указателей. Где и когда это становится возможным - контролируется директивой компилятора *$POINTERMATH*.

По умолчанию, арифметика указателей выключена, но она может быть включена для участка кода, используя директиву `{$POINTERMATH ON}`, и выключена после него с помощью `{$POINTERMATH OFF}`. Директива также может применяться при объявлении типа указателя. При этом арифметика с указателями будет доступна для этого типа в любом коде без предварительной обёртки в директивы *$POINTERMATH*. Помимо операций инкремента/декремента, новая арифметика указателей в Delphi 2009 также допускает индексацию.

Сейчас, кроме типов *PChar*, *PAnsiChar* и *PWideChar*, единственным типом с поддержкой арифметики указателей по умолчанию является *PByte*. Но вы можете включить её для любого типа, как например, *PInteger*. Это значительно упростит код из примера выше:

```
1 {$POINTERMATH ON}
2 var
3   Buffer: Pointer;     // трактуется как packed array of Integer;
4   PInt: PInteger;
5 ...
6   // С использованием новой арифметики указателей:
7   PInt := Buffer;
8   for I := 0 to Count - 1 do
9     Writeln(PInt[I]);
10 ...
11 end;
12{$POINTERMATH OFF}
```

11
12

Так что теперь у нас нет необходимости объявлять специальный тип *PIntegerArray*. Также вместо `PInt[I]` можно использовать синтаксис `(PInt + I)^`, который приводит к тому же результату.

 Кажется, в *Delphi 2009* новая арифметика указателей не работает, как ожидается, для указателей на *обобщённые типы* (generics). С каким бы параметрическим типом вы не инстанциировали тип, индексы не масштабируются на `SizeOf(T)`, как это ожидается.

Ссылки (References)

Многие типы в Delphi фактически являются указателями, но притворяются простыми типами. Я называю такие типы **ссылочными**. Примерами таких типов являются динамические массивы, строки, объекты и интерфейсы. Все они являются указателями "под капотом" языка, но с некоторой дополнительной семантикой и часто со скрытым содержанием.

Что отличает ссылки (references) от указателей (pointers):

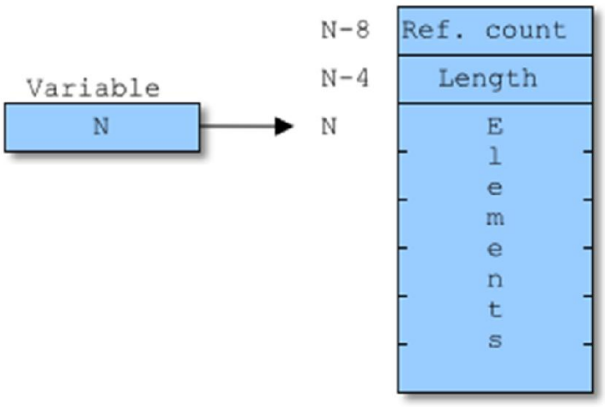
- **Ссылки неизменяемы**. Вы не можете увеличить или уменьшить ссылку. Ссылки указывают на определённые структуры, но никогда не указывают в середину них, как, например, указатели на данные массива в примерах выше.
- **Ссылки не используют синтаксис указателей**. Это скрывает тот факт, что они *являются* указателями, и делает их сложнее для понимания для тех, кто не знаком с этой темой (и поэтому люди делают с ними вещи, которых делать нельзя).

 Не путайте такие ссылки со *ссылочными типами* (reference types) в C++. Они во многом отличаются.

Динамические массивы (dynamic arrays)

До Delphi 4 в языке не было динамических массивов, но они существовали как концепция. Динамический массив - это блок выделенной памяти, которая управляется через указатель. Динамические массивы могут расти или уменьшаться. Фактически это означает, что выделяется новый блок памяти для массива новой длины, в то время как старая память ещё не отпускается. После чего содержимое старой памяти копируется в новую, и старый блок памяти в конце удаляется, а указатель (ссылка) массива начинает указывать на новый блок памяти.

Динамические массивы (например, *array of Integer*) в Delphi работают точно так же. Но библиотека runtime добавляет специальный код, который управляет доступом и присваиваниями. В участке памяти **ниже** адреса, на который указывает ссылка массива, располагаются служебные данные массива: два поля - число выделенных элементов и *счётчик ссылок* (reference count).



Если, как на диаграмме выше, *N* - это адрес в переменной динамического массива, то *счётчик ссылок* массива лежит по адресу `N - 8`, а число выделенных элементов (*указатель длины*) лежит по адресу `N - 4`. Первый элемент массива (сами данные) лежит по адресу `N`.

Для каждой добавляемой ссылки (т.е. при присваивании, передаче как параметр в подпрограмму и т.п.) увеличивается счётчик ссылок, а для каждой удаляемой ссылки (т.е. когда переменная выходит из области видимости или при переприсваивании или присваивании nil) счётчик уменьшается.

Доступ к данным динамических массивов с помощью низкоуровневых процедур типа *Move* или *FillChar*, или любых других подпрограмм, получающих доступ сразу ко всему массиву, наподобие *TStream.Write*, часто выполняется неправильно. Для обычного массива (его часто называют также *статическим* массивом - в противоположность *динамическому* массиву) переменная массива тождественна его данным. Для динамического массива это не так (см. диаграмму выше). Так что если вы хотите получить доступ к данным массива - вы не должны использовать саму переменную массива, а использовать вместо неё первый элемент массива.

```
1
2var
3  Items: array of Integer;
4...
5  // Неправильно: передаётся адрес переменной Items
6  MyStream.Write(Items, Length(Items) * SizeOf(Integer));
7...
8  // Правильно: передаётся адрес первого элемента
9  MyStream.Write(Items[0], Length(Items) * SizeOf(Integer));
10
```

Заметьте, что в примере выше *Stream.Write* использует нетипизированный *var* параметр, который также является ссылочным типом. Мы ещё обсудим их ниже.

Многомерные динамические массивы

Выше мы обсуждали одномерные динамические массивы. Но динамические массивы также могут быть и многомерными. Ну, по-крайней мере, с точки зрения синтаксиса, т.к. в действительности они ими не являются. Многомерный динамический массив является, фактически, одномерным динамическим массивом, в котором каждый элемент является ссылкой на другой одномерный динамический массив.

Предположим, что у нас есть такие объявления:

```
1type
2  TMultiIntegerArray = array of array of Integer;
3var
4  MyIntegers: TMultiIntegerArray;
```

Ну, это выглядит как многомерный динамический массив и, действительно, мы можем обращаться к нему как `MyIntegers[0, 3]`. Но на самом деле это объявление следует скорее читать как (тут я позволяю себе

некоторые вольности с синтаксисом):

```
type
  TMultiIntegerArray = array of (array of Integer);
```

Или, чтобы сделать совсем явным, это фактически означает следующее:

```
type
  TSingleIntegerArray = array of Integer;
  TMultiIntegerArray = array of TSingleIntegerArray;
```

Как вы можете видеть, *TMultiIntegerArray* фактически является одномерным массивом из указателей *TSingleIntegerArray*. Это означает, что данные *TMultiIntegerArray* не хранятся в одном большом непрерывном участке памяти по строкам и столбцам, но является, скорее, *разорванным массивом*, т.е. каждый элемент - просто указатель на другой массив, и каждый из этих подмассивов имеет свой размер. Так что вместо

```
SetLength(MyIntegers, 10, 20);
```

(что создаст 10 *TSingleIntegerArrays* по 20 *Integer* в каждом - т.е. прямоугольный массив), вы можете получить доступ к любому из подмассивов и установить его длину индивидуально:

```
SetLength(MyIntegers, 10);
SetLength(MyIntegers[0], 40);
SetLength(MyIntegers[1], 31);
// и т.д...
```

Строки (Strings)

Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration. — Stan Kelly-Bootle

Строки во многом похожи на динамические массивы. Они также имеют счётчик ссылок, у них похожая внутренняя структура (со счётчиком ссылок и указателем длины ниже самих данных, по тем же смещениям).

Но есть и различия между ними в синтаксисе и семантике. Вы не можете присвоить строке *nil* - вместо этого вы присваиваете ей `''` (пустую строку). Строки могут быть константами (со счётчиком ссылок равным `-1`, что является специальным указанием для библиотеки runtime: она не будет пытаться увеличивать или уменьшать его или удалять строку). Первый элемент строки имеет индекс `1`, в то время как в массиве это `0`.

Объекты (Objects)

Объекты - или, более точно, экземпляры классов (class instances) — не имеют автоматического управления временем жизни. Их внутренняя структура проста. Каждый экземпляр класса содержит по смещению `0` (т.е. ровно по адресу, на который указывает ссылка переменной объекта) указатель на так называемую таблицу VMT. Она представляет собой таблицу с указателями на каждый виртуальный метод класса. По отрицательным смещениям от начала таблицы лежит много другой полезной информации о классе. Я не буду вдаваться в подробности в этой статье. Для каждого класса есть только одна VMT таблица (а не для каждого объекта!).

Классы, которые реализуют интерфейсы (см. ниже), также имеют похожие указатели на таблицы, которые содержат ссылки на методы, реализующие интерфейс, по одному на каждый реализуемый интерфейс. Эти таблицы также содержат некоторую информацию по отрицательным смещениям. По каким смещениям в объекте располагаются эти указатели - определяется структурой (полями)

родительского класса. За этим следит компилятор.

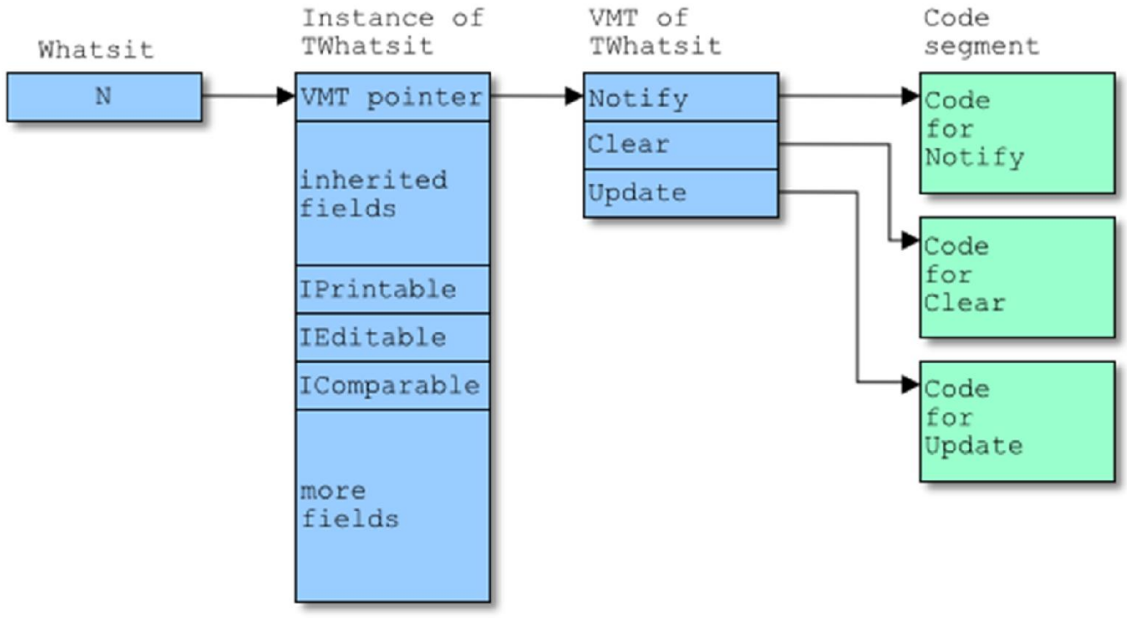
После указателей на VMT и все таблицы интерфейсов, идут обычные поля объекта, которые хранятся ровно как в записях (record).

Данные RTTI и другая информация о классах получаются следованием по ссылке переменной на данные объекта (которая также указывает на указатель на таблицу VMT), а затем следованием по ссылке на VMT. Далее компилятор уже знает, где найти интересующие его данные, обычно через сложные структуры, содержащие указатели на другие структуры, иногда даже с рекурсивными ссылками.

Ниже следует пример. Предположим, что у нас есть такое объявление:

```
1
2 type
3   TWhatsit = class(TAncestor, IPrintable, IEditable, IComparable)
4     // другие поля и объявления методов
5     procedure Notify(Aspect: TAspect); override;
6     procedure Clear; override;
7     procedure Edit;
8     procedure ClearLine(Line: Integer);
9     function Update(Region: Integer): Boolean; virtual;
10    // и т.д...
11  end;
12
13var
14  Whatsit: TWhatsit;
15begin
16  Whatsit := TWhatsit.Create;
17
```

Тогда раскладка объекта в памяти будет выглядеть примерно вот так:



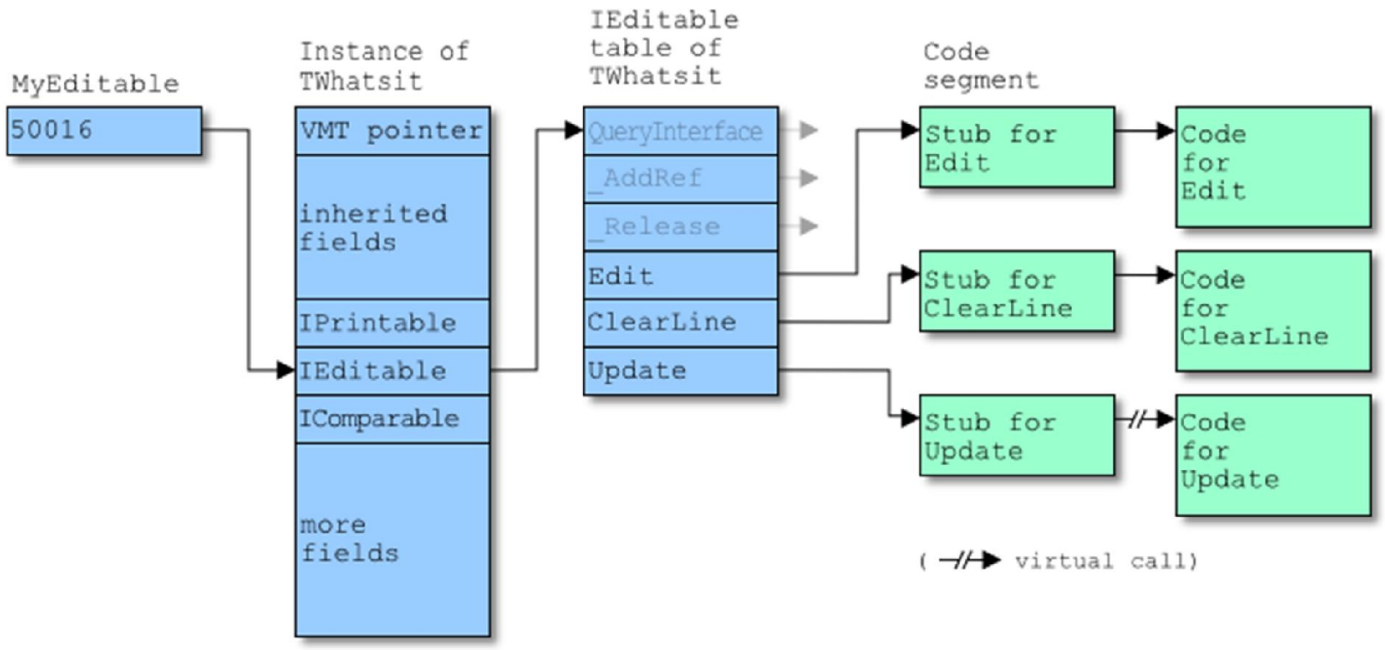
Интерфейсы (Interfaces)

Интерфейсы, фактически, представляют собой просто коллекцию (набор) методов. Внутренне они представлены указателями на указатели на массив указателей на код. Представим, что у нас есть следующие объявления:

```
1 type
2   IEditable = interface
3     procedure Edit;
```

```
3       procedure ClearLine(Line: Integer);
4       function Update(Region: Integer): Boolean;
5     end;
6
7     TWhatsit = class(TAncestor, IPrintable, IEditable, IComparable)
8     public
9       procedure Notify(Aspect: TAspect); override;
10      procedure Clear; override;
11      procedure Edit;
12      procedure ClearLine(Line: Integer);
13      function Update(Region: Integer): Boolean; virtual;
14      // etc...
15    end;
16
17  var
18    MyEditable: IEditable;
19  begin
20    MyEditable := TWhatsit.Create;
21
```

Тогда отношения между интерфейсом, реализующим его объектом и классом, а также методами будет выглядеть вот так:



Переменная *MyEditable* указывает на указатель *IEditable* в объекте, созданном *TMyClass.Create*. Заметим, что *MyEditable* не указывает на начало объекта, а куда-то в его середину. Далее, указатель *MyEditable* в объекте указывает на таблицу указателей - по одному указателю на каждый метод в интерфейсе. Каждая из таких записей указывает на кусочек-кода: заглушку (stub). Этот служебный код корректирует указатель *Self* (который к моменту вызова фактически равен *MyEditable*), чтобы он указывал на начало объекта, с помощью отнимания смещения указателя *IEditable* в объекте от переданного в код указателя, а затем вызывает настоящий метод. Эта заглушка вводится для каждой реализации метода каждого интерфейса, реализуемого классом.

Пример: предположим у нас есть экземпляр по адресу `50000`, а указатель на реализацию *IEditable* классом *TWhatsit* лежит по смещению `16` в каждом экземпляре. Тогда переменная *MyEditable* будет содержать `50016`. Указатель *IEditable* по адресу `50016` будет указывать на таблицу интерфейса для нашего класса (которая лежит, скажем, по адресу `30000`), элементы которой указывают на заглушку (скажем, по адресу `60000`). Заглушка увидит значение `50016` (которое передаётся ей как параметр *Self*), вычтет из него

смещение `16` и получит `50000`. Это и будет настоящий адрес реализующего интерфейс объекта. Затем заглушка вызывает настоящий метод, передавая ему `50000` в качестве параметра *Self*.

В диаграмме я, для ясности, опустил заглушки для методов QueryInterface, _AddRef и _Release

Ну, теперь вы видите, почему иногда я люблю использовать бумагу и ручку? *;-)*

Ссылочные параметры

Ссылочные параметры часто называются *var*-параметрами, а также *out*-параметрами или параметрами, передаваемыми по ссылке.

Ссылочные параметры - это такие параметры подпрограммы, для которых не само значение параметра передаётся и/или возвращается из подпрограммы, а только указатель на него. Пример:

```
1procedure SetBit(var Int: Integer; Bit: Integer);
2begin
3  Int := Int or (1 shl Bit);
4end;
```

Это более или менее эквивалентно следующему:

```
1procedure SetBit(Int: PInteger; Bit: Integer);
2begin
3  Int^ := Int^ or (1 shl Bit);
4end;
```

Хотя, тут есть несколько различий:

- Вы не используете синтаксис указателей. Когда вы пишете имя параметра, вы автоматически разыменовываете указатель, т.е. использование имени параметра означает работу со значением, а не с указателем.
- Ссылочные параметры не могут быть изменены (имеется ввиду сам параметр, а не его значение). Использование имени параметра даёт вам значение параметра - вы не можете изменить указатель или инкрементировать/декрементировать его.
- Вы должны передать что-то, что имеет адрес - т.е. реальную память, если только вы не используете трюк с приведением типов. Так что, имея ссылочный параметр типа Integer, вы не можете передать, например, `17`, `98765` или `Abs(MyInteger)`. Передаваемый параметр должен быть переменной (это включает в себя также элементы массива, поля записей и объектов и т.д.).
- Фактические параметры обязаны быть того же типа, как и параметры в объявлении подпрограммы, т.е. вы не можете передать *TEdit*, если вы объявили параметр как *TObject*. Чтобы избежать этого, вы можете использовать только *нетипизированные ссылочные параметры* (см. ниже). Прим. пер.: как это сделано в, например, FreeAndNil.

Синтаксически, наверное, кажется, что проще использовать ссылочные параметры, чем явные указатели. Но вы должны быть в курсе некоторых особенностей. Чтобы передавать указатели, вы должны увеличить уровень косвенности на единичку. Другими словами, если у вас есть указатель *P* на *Integer*, то чтобы передать его, вы должны синтаксически передавать *P^* (хотя на деле будет передаваться сам P), например:

```
1 procedure SetBit(var Int: Integer; Bit: Integer);
2 begin
3   ...
4 end;
5
6 ...
7
8 var
```

```
  Int: Integer;
  Ptr: PInteger;
  Arr: array of Integer;
begin
  // Инициализация Int, Ptr и Arr не показана...
  SetBit(Ptr^, 3);    // Передаётся сам Ptr
  SetBit(Arr[2], 11); // Передаётся @Arr[2]
  SetBit(Int, 7);     // Передаётся @Int
```

Прим. пер.: на самом деле по ссылке также передаются почти любые типы, размер которых больше SizeOf(Pointer), даже если вы не указали var или out. Например, запись из 8 байт будет передаваться по ссылке. Хотя с точки зрения синтаксиса никаких указателей вы опять не увидите (хотя без наличия модификатора const будет сделана локальная копия). Аналогично: если функция возвращает тип, размер которого больше SizeOf(Pointer), то на самом деле в функцию будет передан указатель на память, куда надо записать результат. Т.е. функция

```
type
  TRec = record
    A: Integer;
    B: Integer;
  end;

function GetRec: TRec;
begin
  Result.A := 1;
  Result.B := 2;
end;
```

На самом деле трактуется как:

```
procedure GetRec(var Result: TRec);
begin
  Result.A := 1;
  Result.B := 2;
end;
```

Или:

```
type
  PRec = ^TRec;

procedure GetRec(Result: PRec);
begin
  Result^.A := 1;
  Result^.B := 2;
end;
```

См. также этот отчёт.

Нетипизированные параметры

Нетипизированные параметры также являются ссылочными параметрами, но они могут быть либо *var*, либо *const* либо *out*. Вы можете передавать в этот параметр любой тип данных, что делает эти виды параметров пригодными для написания подпрограмм, которые принимают почти что угодно, любого размера и типа, но это также означает, что вы должны как-то указать подпрограмме на тип передаваемого аргумента - либо отдельным параметром, либо просто по соглашению, либо же тип параметра значения

не имеет.

Внутренне нетипизированные параметры тоже передаются как указатели на реальное значение параметра. Ниже идёт два примера. Первый из них - это общая подпрограмма для заполнения произвольного буфера набором байт, т.е. тип передаваемой переменной не имеет значения:

```
1
2
3  // Пример подпрограммы, для которой не важен тип параметра
4  procedure FillBytes(var Buffer; Count: Integer; Values: array of Byte);
5  var
6    P: PByte;
7    I: Integer;
8    LenValues: Integer;
9  begin
10   LenValues := Length(Values);
11   if LenValues > 0 then
12   begin
13     P := @Buffer; // Считаем буфер массивом байт.
14     I := 0;
15     while Count > 0 do
16     begin
17       P^ := Values[I];
18       I := (I + 1) mod LenValues;
19       Inc(P);
20       Dec(Count);
21     end;
   end;
end;
```

Второй пример - это метод *TIntegerList* - дочернего класса для TTypedList:

```
1
2  function TIntegerList.Add(const Value): Integer;
3  begin
4    Grow(1);
5    Result := Count - 1;
6    // FInternalArray: array of Integer;
7    FInternalArray[Result] := Integer(Value);
   end;
```

Как вы можете видеть, чтобы использовать указатель, вы должны явно взять адрес параметра, несмотря на тот факт, что фактически параметр внутренне уже является указателем. И опять-таки: уровень косвенности сдвинут на единицу.

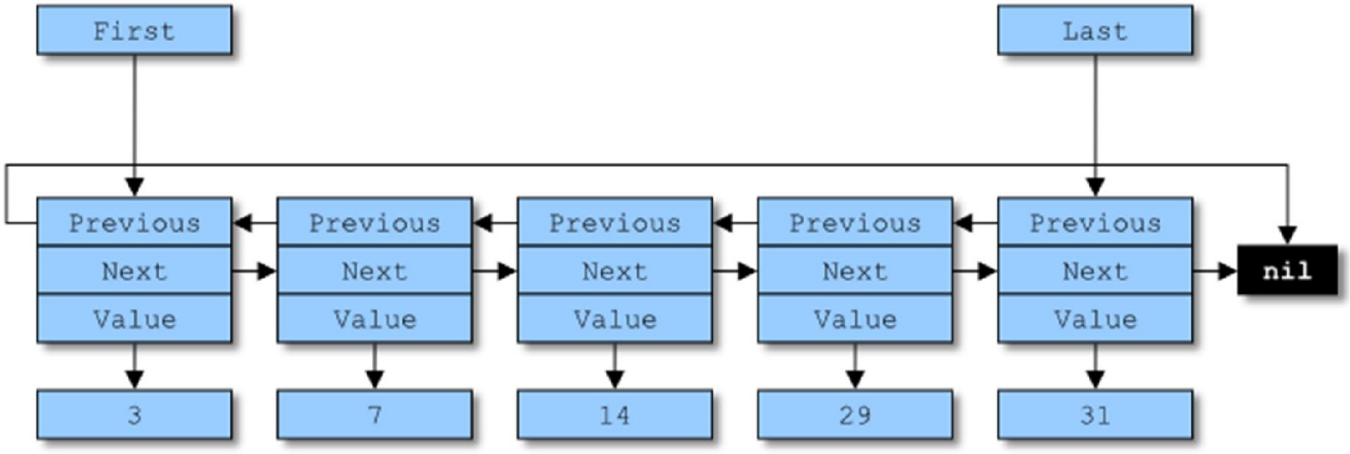
Чтобы получить доступ к значению нетипизированного параметра, вы можете использовать его как обычный ссылочный параметр, но только вы должны сделать приведение к типу, чтобы компилятор знал, как обращаться с параметром.

Я уже упоминал уровень косвенности (level of indirection). Вы уже видели это в действии. Например, если вы хотите инициализировать динамический массив с помощью FillBytes, то вы не передаёте в FillBytes саму переменную массива, а только первый элемент массива. Фактически, вы также можете передать первый элемент статического массива для достижения такого же эффекта. Таким образом, если вы передаёте любой массив в подпрограмму с нетипизированным ссылочным параметром, то, по моему мнению, ваш лучший выбор - всегда передавать первый элемент массива, чтобы не зависеть от типа массива (динамический или статический) (тем более, что вы потом можете изменить объявление - и попробуйте тогда найти ошибку в коде).

Структуры данных

Указатели активно используются в структурах данных типа связанных списков, любых видов деревьев или иных иерархий. Я не буду подробно разбирать их здесь. Достаточно сказать, что такие продвинутые структуры данных не могут существовать без указателей или подобных видов ссылок (например, индекс массива в элементе массива), даже в языках, где формально нет указателей, типа Java (ну, по крайней мере я так думаю). Если вы хотите узнать больше об этих структурах данных - просто возьмите любую книжку по этой теме.

Я только дам простой пример диаграммы структуры данных, которая основательно зависит от указателей - связанный список:



Если у вас есть такие структуры данных, то обычно имеет смысл инкапсулировать всю внутреннюю работу в класс, так что ваша работа с указателями может быть ограничена реализацией класса, и они не появятся в интерфейсе класса. Указатели - это мощное, но сложное и опасное средство. Так что если вы можете избежать его - то избегайте.

Заключение

Я попытался дать вам своё видение указателей. Есть и другие подходы, но, по моему мнению, использование диаграмм (не важно, насколько примитивных) со стрелочками - это неплохой способ разобраться в сложных проблемах с указателями, или для понимания как же связаны вместе интерфейсы, объекты, классы и код. Это не означает, что я начинаю рисовать диаграммы для каждой проблемы. Только для сложных.

Эта статья пыталась показать, что указатели повсюду, даже если вы их не видите. Это не причина для паранойи, но понимание указателей и их механизмов, по моему мнению, необходимо для избегания многих ошибок.

Я надеюсь, что мне удалось подсказать вам несколько полезных советов. Я уверен, что эта статья далеко не полна, и я бы хотел услышать пожелания по улучшению. Просто пишите мне на e-mail (прим. пер.: на английском, разумеется).

Примечания переводчика:

(*) Подробное описание компьютерной памяти вы можете прочитать у Рихтера и цикле статей Марка Руссиновича "Pushing the Limits of Windows" (часть 1, часть 2, часть 3).

(**) Ещё одна причина для повсеместного использования FreeAndNil.

(***) Хотя это и чертовски маловероятно. Писать код без такого предположения довольно тяжело, т.к.

получается, что (к примеру) после ZeroMemory мы не получаем *nil*.

(****) На самом деле, формально такой массив должен быть объявлен с ключевым словом packed.