目　　录

1 DELPHI 的指针-引用

大家都认为，C 语言之所以强大，以及其自由性，很大部分体现在其灵活的指针运用上。因此，说指针是 C 语言的灵魂，一点都不为过。同时，这种说法也让很多人产生误解，似乎只有 C 语言的指针才能算指针。Basic 不支持指针，在此不论。其实，Pascal 语言本身也是支持指针的。从最初的 Pascal 发展至今的 Object Pascal，可以说在指针运用上，丝毫不会逊色于 C 语言的指针。

以下内容分为八个部分，分别是

一、类型指针的定义

二、无类型指针的定义

三、指针的解除引用

四、取地址（指针赋值）

五、指针运算

六、动态内存分配

七、字符数组的运算

八、函数指针

一、类型指针的定义。对于指向特定类型的指针，在 C 中是这样定义的：

```
int *ptr;
char *ptr;
```

与之等价的 Object Pascal 是如何定义的呢？

```
var
ptr : ^Integer;
ptr : ^char;
```

其实也就是符号的差别而已。

二、无类型指针的定义。C **中有** void *类型，**也就是可以指向任何类型数据的指针。**Object Pascal **为其定义了一个专门的类型：**Pointer**。于是，**

ptr : Pointer;

就与 C **中的**

void *ptr;

等价了。

三、指针的解除引用。要解除指针引用（即取出指针所指区域的值），C **的语法是** (*ptr)，Object Pascal **则是** ptr^。

四、取地址（指针赋值）。取某对象的地址并将其赋值给指针变量，C **的语法是**

ptr = &Object;

Object Pascal **则是**

ptr := @Object;

也只是符号的差别而已。

五、指针运算。在 C **中，可以对指针进行移动的运算，如：**

char a[20];

char *ptr=a;

ptr++;

ptr+=2;

当执行 ptr++;**时，编译器会产生让** ptr **前进** sizeof(char)**步长的代码，之后，**ptr **将指向** a[1]。ptr+=2;**这句使得** ptr **前进两个** sizeof(char)**大小的步长。同样，我们来看一下** Object Pascal **中如何实现：**

var

a : array [1..20] of Char;

ptr : PChar; //PChar **可以看作** ^Char

begin

ptr := @a;

Inc(ptr); // **这句等价于** C **的** ptr++;

Inc(ptr, 2); //**这句等价于** C **的** ptr+=2;

end;

六、动态内存分配。C **中，使用** malloc()**库函数分配内存，**free()**函数释放内存。如这样的代码：**

int *ptr, *ptr2;

```
int i;
ptr = (int*) malloc(sizeof(int) * 20);
ptr2 = ptr;
for (i=0; i<20; i++){
*ptr = i;
ptr++;
}
free(ptr2);
```

Object Pascal 中，动态分配内存的函数是 GetMem()，与之对应的释放函数为 FreeMem()（传统 Pascal 中获取内存的函数是 New()和 Dispose()，但 New()只能获得对象的单个实体的内存大小，无法取得连续的存放多个对象的内存块）。因此，与上面那段 C 的代码等价的 Object Pascal 的代码为：

```
var ptr, ptr2 : ^integer;
i : integer;
begin
GetMem(ptr, sizeof(integer) * 20);
//这句等价于 C 的 ptr = (int*) malloc(sizeof(int) * 20);
ptr2 := ptr; //保留原始指针位置
for i := 0 to 19 do
begin
ptr^ := i;
Inc(ptr);
end;
FreeMem(ptr2);
end;
```

对于以上这个例子（无论是 C 版本的，还是 Object Pascal 版本的），都要注意一个问题，就是分配内存的单位是字节（BYTE），因此在使用 GetMem 时，其第二个参数如果想当然的写成 20，那么就会出问题了（内存访问越界）。因为 GetMem(ptr, 20);实际只分配了 20 个字节的内存空间，而一个整形的大小是四个字节，那么访问第五个之后的所有元素都是非法的了（对于 malloc()的参数同样）。

七、字符数组的运算。C 语言中，是没有字符串类型的，因此，字符串都是用字符数组来实现，于是也有一套 str 打头的库函数以进行字符数组的运算，如以下代码：

```
char str[15];
char *pstr;
strcpy(str, "teststr");
```

```
strcat(str, "_testok");
pstr = (char*) malloc(sizeof(char) * 15);
strcpy(pstr, str);
printf(pstr);
free(pstr);
```

而在 Object Pascal **中，有了** String **类型，因此可以很方便的对字符串进行各种运算。但是，有时我们的** Pascal **代码需要与** C **的代码交互（比如：用** Object Pascal **的代码调用** C **写的** DLL **或者用** Object Pascal **写的** DLL **准备允许用** C **写客户端的代码）的话，就不能使用** String **类型了，而必须使用两种语言通用的字符数组。其实，** Object Pascal **提供了完全相似** C **的一整套字符数组的运算函数，以上那段代码的** Object Pascal **版本是这样的：**

```
var str : array [1..15] of char;
pstr : PChar; //Pchar 也就是 ^Char
begin
StrCopy(@str, 'teststr'); //在 C 中，数组的名称可以直接作为数组首地址指针来用
//但 Pascal 不是这样的，因此 str 前要加上取地址的运算符
StrCat(@str, '_testok');
GetMem(pstr, sizeof(char) * 15);
StrCopy(pstr, @str);
Write(pstr);
FreeMem(pstr);
end;
```

八、函数指针。在动态调用 DLL **中的函数时，就会用到函数指针。假设用** C **写的一段代码如下：**

```
typedef int (*PVFN)(int); //定义函数指针类型
int main()
{
HMODULE hModule = LoadLibrary("test.dll");
PVFN pvfn = NULL;
pvfn = (PVFN) GetProcAddress(hModule, "Function1");
pvfn(2);
FreeLibrary(hModule);
}
```

就我个人感觉来说，C **语言中定义函数指针类型的** typedef **代码的语法有些晦涩，而同样的代码在** Object Pascal **中却非常易懂：**

```
type PVFN = Function (para : Integer) : Integer;
```

```
var
fn : PVFN;
//也可以直接在此处定义，如：fn : function (para:Integer):Integer;
hm : HMODULE;
begin
hm := LoadLibrary('test.dll');
fn := GetProcAddress(hm, 'Function1');
fn(2);
FreeLibrary(hm);
end;
```

附：

Delphi 中指针功能非常强大，所有 c 中能实现的指针 Delphi 中都能实现. 上面认为 Delphi 指针不是强项的只是一种误解(或者对指针的机制一知半解).

由于 Pascal 语言的限制，用 Delphi 的指针时很多情况下需要强制类型转换. Delphi 中提供了很多指针类型，而且非常方便的是你可以自定义自己的指针类型.

一个经验: 要掌握一种数据类型并且能够灵活应用,一个比较好的办法是别考虑什么类型是什么名字，而只需要考虑这种类型的变量将占用多少字节. 凡是字节数相同的类型都可以认为是同一类型 :-), 提供不同类型只是为了编译器能够更方便的查找错误而已. 比如: Integer, Pointer, PChar, TSmallPoint 甚至 array [0..4] of Char

你都可以把他们当成是同一类型加以使用(有了这种思路，可以实现很大的程序灵活性和代码高效性). 所以我很不理解的是 JAVA 中不支持指针(因此我也认为用 JAVA 绝对不可能写出很高效的程序，而且会有很多 C/C++/DELPHI 中用一句话可以完成的工作在 JAVA 中需要用一个复杂过程，消耗很多额外内存才能达到相同目的). 就事论事，根据你的问题在 Delphi 中和 C 中的解决方案没什么两样.

http://hi.baidu.com/sswanglei/blog/item/3b55d88bcab41d14c8fc7a94.html

2 delphi 的“引用/值”模型

D 中简单的数据类型（如 integer,char，record 等）无论作为参数还是变量都是按值传递和使用的，通常称为值类型。值类型也是直接类型，即每个变量都有自己的存储数据的实际的

内存空间，更改了变量也就直接更改了它的数值。

D 中复杂的数据类型（如 class)则是按引用传递和使用的。引用类型是间接类型，它存储的是间接数据，即堆该数据的引用（可以理解为指针），也就是存储的是实际数据数据存储内存的地址。

当这两种变量和参数传递时候，值类型传递的是值的副本，引用类型传递的是引用的副本。因为，值类型数据存放在栈中，而引用类型数据存放在堆中。处理器直接使用栈指针分配和访问内存。我们把对象存储在堆中，而把对象的引用和值类型存放在栈中。因为值类型和对象的引用可以确定其大小和生命期，但是对象无法确定其大小和生命期。而编译器无法知道对象要从堆中分配多少内存空间，占用多长内存时间。所以，在栈中的变量不需要程序员手工释放，而堆中的变量需要程序员手工分配空间和销毁。程序中通过对象引用来访问对象，改变对象引用无法改变对象本身。

D 中对象引用和类引用

一个对象引用就是一个句柄或指针，当你分配一个对象引用给一个变量时候，D 仅复制引用，而不是整个对象。当程序不再使用该对象时候，应该 free 显式将其销毁。一个对象引用通常以一个变量形式存在，但也有函数或责属性返回值的形式。

同一个类创建的对象，通过类引用都指向同一个类表。

someobj:=Tobj.create;

someobj 就是对象引用，而 Tobj 就是类引用。

类引用类型是“类的类”（class of class）类型，也称元类（metaclass）。其构造形式为：class of type。type 为任何类型。类引用类型的声明不能直接用于变量或者参数声明中。可以把 nil 赋值给任何类引用变量。如果把类看做对象的模板，那么元类就是类的模板。

类引用是引用以个具体类的表达式。类引用在 D 中用做生成新对象、调用类方法，已经测试或者转换对象类型。类引用实现为指向一个关于该类的信息表的指针，特别是类的虚方法表（VMT)。通常，类引用的函数是一个类的名称，但也可以是类型为元类的变量，或者返回为类引用的函数或属性。

D 中的对象传递

在 D 中，参数传递机制为值传递和引用传递。值传递传递的是数值参数（默认），引用传递传递的是变量参数（var）。

function getsome(x:integer):integer; 值传递

function getany(var x:integer):integer; 引用传递

值传递 getsome(a);中产生的 x 是 a 的一个副本，数值参数此时就好像是一个局部变量，其变化不影响 a 的值。但是引用传递 getany(a)中产生的 x 是指向 a 的指针，变量参数在函数中变化，直接影响到 a 的值。引用传递时候，要注意变量参数及时传递给多个参数，

也不会创建他的副本。引用传递能改变传给它的变量的值。

总之，按值传递时候，行参和实参是两个变量；引用传递的时候是同一个变量。

注意的是，对象在做为参数传递的时候，由于我们在传递对象的时候，实际上传递的都是对象的引用。所以，当对象作为参数传递时候，无论是采用值传递还是引用传递，传递的都是对象的引用，不同的是值传递会产生一个引用的副本（相当于该对象的一个别名，绑定的是同一个对象），但是如果修改形参的话，还是会修改实参的值。

D 中对象的克隆

TPersistent 类下的所有对象都是可以克隆的。

:=赋值操作符，是将一个对象引用赋值给一个对象变量（相当于多了一个别名，实体对象只有一个）；

assign 或 assignto 方法可以将对象属性进行赋值，得到两个状态一样的对象。（实实在在的复制，内存×2，实体对象有两个）。

Delphi 指针大全

Delphi 指针理解

看一个指针用法的例子：

```
1   var
2       X,  Y:  Integer;        //  X  and  Y  整数类型
3       P:  ^Integer;           //  P  指向整数类型的指针
4   begin
5       X  :=17; //  给  X  赋值
6       P  :=  @X;              //  把  x 的地址赋给 p
7       Y  :=  P^;              //  取出 p 所指向的数值赋给 y
8   end;
```

第二行定义了两个变量 X,y. 第三行声明了 p 是指向整数类型的指针;意味着 p 能够指向 x 或者 y 的地址.第五行赋给 x 值,第六行把 x 的地址赋给 p.最

后通过 p 指向的变量赋值给 y.此时,x 和 y 有相同的值.

操作符@用来取出变量的地址,也可以取出过程和函数的地址.

而符号^有两个目标,

当它出现在类型定义的前面时如 ^typename 表示指向这种类型的指针;

当它出现在指针变量后边时 如 point^ 返回指针指向的变量的值;

理解指针比较容易理解面向对象的 pascal 语言,因为指针经常在幕后操作.任何要求动态分配大的内存空间的类型可以用指针类型.例如

,long-string 变量,实际在使用指针进行操作.另外一些高级的编程技术需要使用指针类型.

有时指针是适应 object pascal 严格的类型限制的唯一方法.同过一个通用的指针类型,通过类型转换成不同的指针类型,如下面的例子:

```
type

    PInteger    =    ^Integer;
var
    R:    Single;
    I:    Integer;
    P:    Pointer; //通用的指针
    PI:    PInteger;
begin
    P    :=    @R; //取出 R 的内存地址
    PI    :=    PInteger(P); //把通用类型转换成指向整数类型的指针
    I    :=    PI^;
end;
```

当然了,实数和整数的存储格式不同.这种赋值是把原始的二进制数据从 R 拷贝到 I,而不进行转换.

保留字 nil 是一个特殊的常量可以赋给任何指针类型,当 nil 赋給一个指针时,指针什么也不指向,是一个空指针.

@操作符返回变量的内存中的存储地址,或者是过程\函数\方法;

1.如果变量,@X 返回的是 x 的地址。如果编译选项{$T-}没有打开,着返回的事一个通用的指针,如果编译选项打开了,着返回的是 x 的类型对应的指针.

2.如果是例程(过程\函数),@F 返回的是 F 的入口点，@F 的类型是一个指针。

3.当@用在类的方法中时,则方法的名称必须有类名,例如@TMyclass.Dosomething 指针指向 TMyclass 的 dosomething 方法。

当一个过程变量在赋值语句的左边时，编译器期望一个过程值在赋值语句的右边。这种赋值使得左边的变量可以指向右边定义的过程或者函数

入口点。换句话说，可以通过该变量来引用声明的过程或者函数，可以直接使用参数的引用。

```
var

    F:    function(X:    Integer):    Integer;
    I:    Integer;
function    SomeFunction(X:    Integer):    Integer;
  ...
F    :=    SomeFunction;    //    给 f 赋值
I    :=    F(4);                    //    调用所指向的函数
```

在赋值语句中，左边变量的类型决定了右边的过程或者方法指针解释。

```
var

    F,    G:    function:    Integer;
    I:    Integer;
function    SomeFunction:    Integer;
  ...
F    :=    SomeFunction;    //    给 f 赋值
G    :=    F;                    //    把 F 的值拷贝给 G
I    :=    G;                    //    调用函数
```

第一句获得函数的入口，第二句将指针复制，第三句获得函数的返回值。

有时候还可以这样使用

if F = MyFunction then ...;

在这里，F 的出现导致一个函数调用；编译器调用 F 指向的函数，然后调用 Myfunction，比较结果。这个规则是无论何时一个过程变量（

procedural variable）出现在一个表达式中，它表示调用所指向的函数或者过程。有时 F 指向一个过程（没有返回值），或者 f 指向一个需要参

数的函数，则前面的语句会产生一个编译错误。要比较 F 和 Myfunction 需要用

if @F = @MyFunction then ...;

@F 把 F 转换成一个包含地址的无类型的指针变量,@myfunction 返回 myfunction 的地址。

获得一个过程变量的内存地址使用@@。例如，@@F 返回 F 的地址。

@操作符通常把一个无类型的指针值赋给一个过程变量，例如：

var StrComp: function(Str1, Str2: PChar): Integer;

...

@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');

调用 GetProcAddres 函数,用 strcomp 指向这个值

任何过程变量可以赋成 nil，表示指证什么也不指向。但是试图调用一个 nil 值的过程变量导致一个错误，为了测试一个过程变量是否可以赋值

，用标准的赋值函数 Assigned

if Assigned(OnClick) then OnClick(X);

==========

大家都认为，C 语言之所以强大，以及其自由性，很大部分体现在其灵活的指针运用上。因此，说指针是 C 语言的灵魂，一点都不为过。同时，这种说法也让很多人产生误解，似乎只有 C 语言的指针才能算指针。Basic 不支持指针，在此不论。其实，Pascal 语言本身也是支持指针的。从最初的 Pascal 发展至今的 Object Pascal，可以说在指针运用上，丝毫不会逊色于 C 语言的指针。

以下内容分为八个部分，分别是

一、类型指针的定义

二、无类型指针的定义

三、指针的解除引用

四、取地址（指针赋值）

五、指针运算

六、动态内存分配

七、字符数组的运算

八、函数指针

一、类型指针的定义。对于指向特定类型的指针，在 C **中是这样定义的：**

int *ptr;

char *ptr;

与之等价的 Object Pascal **是如何定义的呢？**

var

ptr : ^Integer;

ptr : ^char;

其实也就是符号的差别而已。

二、无类型指针的定义。C **中有** void *__类型，也就是可以指向任何类型数据的指针。__Object Pascal **为其定义了一个专门的类型：**Pointer**。于是，**

ptr : Pointer;

就与 C **中的**

void *ptr;

等价了。

三、指针的解除引用。要解除指针引用（即取出指针所指区域的值），C **的语法是** (*ptr)**，**Object Pascal **则是** ptr^**。**

四、取地址（指针赋值）。取某对象的地址并将其赋值给指针变量，C **的语法是**

ptr = &Object;

Object Pascal **则是**

ptr := @Object;

也只是符号的差别而已。

五、指针运算。在 C **中，可以对指针进行移动的运算，如：**

char a[20];

char *ptr=a;

ptr++;

ptr+=2;

当执行 ptr++;时，编译器会产生让 ptr 前进 sizeof(char)步长的代码，之后，ptr 将指向 a[1]。ptr+=2;这句使得 ptr 前进两 个 sizeof(char)大小的步长。同样，我们来看一下 Object Pascal 中如何实现：

```
var
    a : array [1..20] of Char;
    ptr : PChar; //PChar 可以看作 ^Char
begin
    ptr := @a;
    Inc(ptr); // 这句等价于 C 的 ptr++;
    Inc(ptr, 2); //这句等价于 C 的 ptr+=2;
end;
```

六、动态内存分配。C 中，使用 malloc()库函数分配内存，free()函数释放内存。如这样的代码：

```
int *ptr, *ptr2;
int i;
ptr = (int*) malloc(sizeof(int) * 20);
ptr2 = ptr;
for (i=0; i<20; i++){
    *ptr = i;
    ptr++;
}
free(ptr2);
```

Object Pascal 中，动态分配内存的函数是 GetMem()，与之对应的释放函数为 FreeMem()（传统 Pascal 中获取内存的函数是 New()和 Dispose()，但 New()只能获得对象的单个实体的内存大小，无法取得连续的存放多个对象的内存块）。因此，与上面那段 C 的代码等价的 Object Pascal 的代码为：

```
var ptr, ptr2 : ^integer;
    i : integer;
begin
    GetMem(ptr, sizeof(integer) * 20);
      //这句等价于 C 的 ptr = (int*) malloc(sizeof(int) * 20);
    ptr2 := ptr; //保留原始指针位置
    for i := 0 to 19 do
    begin
      ptr^ := i;
```

```
    Inc(ptr);
  end;
  FreeMem(ptr2);
end;
```

对于以上这个例子（无论是 C 版本的，还是 Object Pascal 版本的），都要注意一个问题，就是分配内存的单位是字节（BYTE），因此在使用 GetMem 时，其第二个参数如果想当然的写成 20，那么就会出问题了（内存访问越界）。因为 GetMem(ptr, 20);实际只分配了 20 个字节的内存空间，而一个整形的大小是四个字节，那么访问第五个之后的所有元素都是非法的了（对于 malloc()的参数同样）。

Delphi 指针大全

3 谈谈 Delphi 的类型与指针

内存中的数据除了 0 便是 1，你把它当作图片、字符、数字等等，那是你的事，内存只认识 0 和 1.

Win32 系统除了使用硬内存以外，还可以从硬盘上开辟虚拟内存;

因为 Win32 的内存地址范围在 4 个 G 以内(0..232-1)，所以它最多能够给一个应用程序分配 4G 的运行空间；并且其中的 2G 有系统管理，实际上程序只有 2G 的自主空间. 还记得有说 String 最大长度是 2G 吗? 就是这个道理.

有 4G 的内存，就有 4G 个地址，也就是最多可以有 (1024*1024*1024*4 - 1 = 4294967295) 个内存地址，这刚好是 Delphi 中 Cardinal 的最大值，所以 32 位的指针类型追到底都是 Cardinal 类型的一个数字.

一个内存地址是 0..4294967295 之间的一个数字，你可以通过内存地址读取或写入数据；一个指针要用来索引或标识内存，它也是 0..4294967295 之间的一个数字；它们虽不相同，但通过指针可以找到实际存储数据的内存地址，并按指定的类型去读写它.

譬如:

```
var
  str: string;
```

```
  n: Cardinal;
  pstr: PString;
begin
  str := 'ABCDE';
  n := Cardinal(str); {获取内存地址}
  pstr := @str;          {现在 pstr 是 str 的指针}

  {n 与 pstr 的数字结果是(结果是随机的，知道不一样就行了):}
  ShowMessage(IntToStr(n));                    {4571092}
  ShowMessage(IntToStr(Cardinal(pstr))); {1244652}

  {但通过 pstr 可以找到 str}
  ShowMessage(pstr^); {ABCDE}
end;
```

程序运行后，字符串所在的内存基本上是下面这个样子(以字节为单位)，上例中的 n 标识着 ↓ 的位置:

↓

A B C D E

换二进制图示一下:

↓

00001010 00001011 00001100 00001101 00001110

如果只看二进制，这个数据到底是什么很难知道；再说它为什么非得是字符串 "ABCDE" **呢？这可不一定**.

下面的例子中，我们先是权且把它当作字符串，但随着指针的移动，字符串也在变化.

然后，有分别把它分别用 Byte **指针**(PByte) **和** Integer **指针**(PInteger) **去读取它，也会得到相应的值**.

完整示例如下:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  str: string;
  ps: PChar;
  n: Cardinal;
begin
  str := 'ABCDE';
  ps := PChar(str);
  n := Cardinal(ps);
  //n := Cardinal(str); {这行可以代替上面两行}
  ShowMessage(IntToStr(n)); {结果是 Windows 随机管理的}
```

```
  ShowMessage(PChar(n));     {ABCDE}
  ShowMessage(PChar(n+1)); {BCDE}
  ShowMessage(PChar(n+2)); {CDE}
  ShowMessage(PChar(n+3)); {DE}
  ShowMessage(PChar(n+4)); {E}
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  str: string;
  n: Cardinal;
  pb: PByte;
begin
  str := 'ABCDE';
  n := Cardinal(str);
  ShowMessage(IntToStr(n)); {4571140; 这是我这里的结果, 这是随机的}

  pb := PByte(n);
  ShowMessage(IntToStr(pb^)); {65}
  pb := PByte(n+1);
  ShowMessage(IntToStr(pb^)); {66}
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  str: string;
  n: Cardinal;
  pint: PInteger;
begin
  str := 'ABCDE';
  n := Cardinal(str);
  ShowMessage(IntToStr(n)); {4571140; 这是我这里的结果, 这是随机的}

  pint := PInteger(n);
  ShowMessage(IntToStr(pint^)); {1145258561}
```

```
  pint := PInteger(n+1);
  ShowMessage(IntToStr(pint^)); {1162101570}
end;

end.
```

上面的第三个程序段的结果或许让你迷惑:
第一个结果，**应该和** "ABCD" **有点关系才对啊**，**怎么是**: 1145258561 ?
第二个结果，**应该和** "BCDE" **有点关系才对啊**，**怎么是**: 1162101570 ?

为什么呢? **这当然没错**，**听我解释**:
1145258561 **转换成十六进制是**: 44434241，**写得清楚一点是**: $44 $43 $42 $41; **还记得** Intel **等当下流行的** CPU **安排数据是倒着的吗**?

自己算算下一个，**用附件中的计算器即可**.

4 Delphi 7 指针数据类型

指针类型

访问一个内存变量通常有两种方法：一种是通过名字访问，另一种是通过地址访问。指针是通过地址访问变量的一种数据类型。

由于动态数据结构的变量必须在程序执行过程中动态生成，不能预先声明，所以无法预先给这些变量起好名字，也无法通过名字进行访问，因此只能用指针先得到它的地址，然后间接访问它们。

指针类型的定义如下：

type 指针类型标识符＝^类型标识符;

可以对指针变量赋值。对指针变量赋值实际上是将新的指针变量所指向的动态存储单元的首地址赋给该指针，使该指针指向新的动态存储单元。给指针赋值必须类型相同或赋 nil（空值），给指针赋值后，指针原来所指的动态存储单元中还存有数据，只是它已无法被访问。

对指针可以应用比较运算符中的“＝”和“<>”进行比较。两个指针指向同一个动态变量，则两个指针变量相等，否则即为不等。Delphi 的指针分为 "类型指针" 和 "无类型指针" 两类.
Delphi 中的类型，常用的也得有几百个，我们可以给每种类型定义相应的类型指针.
其实 Delphi 已经为很多类型预定义了指针，譬如数据类型:
Integer 有对应的 PInteger;
Char 有对应的 PChar;
string 有对应的 PString;
再譬如:
TPoint 有对应的 PPoint;
TColor 有对应的 PColor 等等.

另外，指针也可以有指针，譬如: PChar 是字符指针，PPChar 又是 PChar 的指针(这都是 Delphi 预定义的).

总结一下类型与指针的命名规则:
类型约定用 T 打头(Delphi 常规的数据类型除外，譬如: String);
指针约定用 P 打头;
指针的指针约定用 PP 打头.
类型和指针是不可分的两个概念，指针本身也是一种类型 - "指针类型".

先认识一下指针相关的操作符(@、^、Addr):
@ @变量 获取变量指针
Addr Addr(变量)
^ 指针^ 获取指针指向的实际数据
var Pxxx: ^类型 定义 Pxxx 某种类型的指针的变量
type Pxxx = ^类型 定义 Pxxx 为某种类型的指针

如:定义整型指针

```
var
a :^Integer;
b :^Integer;
```

也可以用:

```
var
a Integer;
b Integer;
```

Object Pascal 认为每一个指针类型是相异的，为了把 a 的值赋给 b，你必须建立一个新的类型，示例如下：

```
type
Pin = ^integer;
var
a,b :pin;
```

也可以直接写

```
var
a,b :^integer;
```

5 delphi 中的 Pchar 指针

1． 传统的 Pascal 字符串

在 Pascal 中，典型的字符串是一定长度的字符序列。每一字符串有一设定的长度（缺省值为 255），下面是一个例子：

```
Var
Address:String;
Code:String[50];
```

Address 是一长度为 255 的字符串，Code 的最大长度为 50。
传统的 Pascal 字符串长度不能超过 255。
可以用字符串连接操作"+"把字符串连接在一起：
Result:=String1+String2;

2． Delphi 中的长字符串

Delphi 除了支持传统的 Pascal 短字符串还支持长字符串。长字符称为 ANSIString。长字符串动态分配内存,即用字符串时才分配字符串所需内存,所以其长度不受限制。在 Delphi 中你如果用 String1:String 作类型说明，则 String1 既可能是短字符串也可能是长字符串，这取决于编译器中$H 开关的设置。默认值为$H+，代表 ANSI 长字符串，VCL 中的组件使用 ANSI 长字符串。长字符串以 null 结束，这就说明长字符串与 C 语言中的以 null 结束的字符串完全兼容。

可以通过 SetLength 函数设置字符串的最大长度：
SetLength(String1,100);用 TrimLeft,TrimRight 和 Trim 函数分别来消除字符串开头,结尾和首尾的空白区。

如果 Delphi 中的 Exended Syntax 已经设置（缺省值），以 0 为起点的字符数组就和指向字符的指针 Pchar 完全兼容，因为以 0 为起点的字符数组名即指向该字符数组首字符的指针。可以将字符串直接付值给 Pchar 指针。例如：

```
var
P: PChar;
begin
P := 'Hello world';
end;
```

这样 P 就指向存储字符串'Hello world'并以 null 结束的一块内存。

许多 Windows 的应用程序接口 API 函数要求用 Pchar 类型作参数。Pchar 指针在使用是首先用 GetMem(var P: Pointer; Size: Integer)函数申请分配内存，程序结束时用 FreeMem(var P: Pointer[; Size: Integer])函数释放内存。例如：

```
Var WinDir,SysDirchar;
Begin
GetMem(WinDir,256);{为指针分配内存}
GetWindowsDirectory(WinDir,128);{将 Windows 安装目录放至 WinDir}
```

```
ShowMessage('Windows directory is'+WinDir);{显示结果}
End;
```

二：字符串转换

以上介绍了 Delphi **中的四类字符串的定义和使用。由于各类函数对字符串参数类型要求不一，这就需要进行字符串类型转换。**

1． **可以用** StrPas **将以** null **结束的字符串转换为** Pascal **短字符串。**StrpCopy **则完成相反的转换。**

2． **因为长字符串以** null **结束，所以可以用强制类型转换将长字符串转换成** Pchar **类型。用法是：**Pchar(s),s **是一个长字符串。强制类型转换返回一个指向长字符串首字符的指针，并且所指字符串以** null **结束。例如：**

```
Var
Caption,Message:string;
Caption:='Hello World!';
Mssage:='This is a test of long string';
MessageBox(0,Pchar(Message),Pchar(Caption),MB_OK);
```

小结：在使用 Delphi **中的字符串时，要时刻清楚该字符串的类型，以免引起混淆。在理解字符串时要把字符串与指针，内存分配联系起来，加强理解。**