Ponteiros e Alocação Dinâmica de Memória

O Turbo Pascal utiliza diferentes partes, ou segmentos, da memória do seu computador para diferentes propósitos. Alguns segmentos guardam as instruções que o seu computador executa, enquanto outros armazenam dados. Cada um desses segmentos desempenha um papel específico, e você deve entender essas regras e como os segmentos funcionam antes de dominar os conceitos avançados de programação.

Alocação de Memória no Turbo Pascal

O Turbo Pascal divide a memória do seu computador em quatro partes - o *segmento de código*, o *segmento de dados*, o *segmento da área de armazenamento* e a *pilha*. Os programas que usam unidades possuem um segmento de código para cada unidade bem como para o programa principal. Todos os programas, entretanto, têm apenas um segmento de dados, que contém as constantes com tipo definido e as variáveis globais.

Embora o segmento de dados seja claramente destinado ao armazenamento de dados, estes também podem ser armazenados em outras áreas. A área de armazenamento e a pilha guardam dados dinâmicos alocando memória quando necessário. Como a área de armazenamento tem importância fundamental, seu funcionamento é controlando automaticamente pelo Turbo Pascal - você não pode atuar muito sobre a área de armazenamento por conta própria. A pilha, por outro lado, tem importância especial para as técnicas de programação avançada. Este ponto descreve o papel da pilha e mostra como você pode usar a alocação dinâmica em seus programas.

Convenções de Mapeamento de Memória do DOS

O primeiro passo para entender como o Turbo Pascal gerencia a memória é aprender um pouco sobre funcionamento interno do seu microcomputador. Um computador dispõe de uma determinada área de memória RAM (memória de acesso aleatório). Digamos que o seu tenha 640 kbytes. Um kbyte equivale a 1024 bytes, de forma que o seu computador de 640K na realidade possui um total de 655.360 bytes de memória RAM.

Ao ser executado pela primeira vez, o seu programa configura um segmento que guarda suas instruções (um ou mais segmentos de código), um segmento para guardar os dados do programa (o segmento de dados) e um segmento para guardar os dados temporários (o segmento da área de armazenamento). Quando as instruções contidas no segmento de código são executadas, elas tratam tanto os dados contidos no segmento de dados como aqueles contidos no segmento da área de armazenamento.

Como o programa sabe em que byte esses três segmentos começam? E como ele localiza um determinado byte na memória? Usando endereços. Cada byte possui um *endereço,* um valor de 20 bits que identifica essa posição com exclusividade.

Quando um programa precisa acessar um determinado byte, ele usa o endereço para encontrar a posição do byte na memória.

Se um endereço de computador fosse composto de uma única palavra (dois bytes), ele não poderia acessar mais de 64K (65.536 bytes) de memória RAM. Esse era o caso dos primeiros microprocessadores de 8 bits.

O advento dos processadores de 16 bits, em particular a família Intel 8086/88, apresentou um novo esquema de endereçamento de memória, conhecido como *endereçamento segmentado.* O endereçamento segmentado combina *dois* valores de palavra – um segmento e um deslocamento - para formar um endereço de 20 bits. Imagine os segmentos com quarteirões de uma rua e os deslocamentos como as casas de cada quarteirão.

Cada segmento guarda 64K de memória RAM. Os processadores 8086/88 possuem 16 segmentos, resultando em 1.048.560 bytes *(1 megabyte)* de memória endereçável Entretanto, o DOS limita a área de memória que o seu computador pode usar em até 640K.

Segmentos a Deslocamentos

O Turbo Pascal dispõe de duas funções padrão - ***Seg*** e ***Ofs*** - que facilitam a exploração do endereçamento de memória no seu PC. A função Seg fornece o segmento em que uma variável reside, e 0fs fornece seu deslocamento. O programa a seguir utiliza essas funções para mostrar os endereços de quatro variáveis.

```
program memoria;
  uses crt ;

  Type
   St4 = string[4];

  function IntToHex(ai : word) : st4 ;
    const
       Caracterhex: array [0..15] of Char = '0123456789ABCDEF';
    begin
      InttoHex := caracterhex[Hi(ai) shr 4] +
                  caracterhex[Hi(ai) and 15] +
                  caracterhex[Lo(ai) shr 4] +
                  caracterhex[Lo(ai) and 15];
    end ;


var
  i : word ;
  s : string[5];
  r : real;
  c : char;

Begin
 clrscr ;
 writeln('word   ',IntToHex(seg(i)),':',IntToHex(ofs(i)));
 writeln('string ',IntToHex(seg(s)),':',IntToHex(ofs(s)));
 writeln('real   ',IntToHex(seg(r)),':',IntToHex(ofs(r)));
 writeln('char   ',IntToHex(seg(c)),':',IntToHex(ofs(c)));
 write('Pressione ENTER.....');
 readln;
End.
```

Esse programa define quatro variáveis de tipos diferentes e, depois, apresenta o endereço de cada uma delas. Por exemplo, a instrução Seg(i) localiza o segmento da variável i, enquanto Ofs(i) retoma o deslocamento.

A função IntToHex aceita um parâmetro de palavra a retoma o valor hexadecimal como uma cadeia de quatro caracteres. Os segmentos a os deslocamentos são normalmente apresentados no formato hexadecimal.

Ao executar o programa anterior, sua tela mostrará as mensagens apresentadas a seguir. (a saída do programa dependente do computador e da quantidade de memória instalada nele, provavelmente você verá na tela outros números.)

```
Word: 177F:0062
String:177F:0064
Real:  177F:006A
Char:  177F:0070
```

Como você pode observar, todas as variáveis que são globais têm o mesmo segmento. (Todas as variáveis globais residem no segmento de dados.) Além disso, a distância entre os deslocamentos coincide exatamente com o número de bytes necessário para armazenar cada tipo de variável. Por exemplo, uma palavra (tipo Word) começa no deslocamento 0062 e exige dois bytes de armazenamento. A próxima variável da linha (do tipo String) começa no deslocamento 0064.

As constantes de tipo definido do Turbo Pascal ficam armazenadas no segmento de dados, enquanto as constantes sem tipo definido encontram-se no segmento de código.

Na realidade, as constantes sem tipo definido simplesmente se tornam parte do código do seu computador. Por essa razão, as constantes sem tipo definido não possuem endereços.

As variáveis que forem declaradas em rotinas e funções são mantidas na área de armazenamento, que é uma área dinâmica de armazenamento de dados. Quando um programa chama uma rotina, o Turbo Pascal aloca espaço na área de armazenamento para as variáveis locais da rotina. À medida que o Turbo Pascal acrescenta variáveis à área de armazenamento, esta cresce para os endereços mais baixos da memória. Quando a rotina é encerrada, o Turbo Pascal descarta essas variáveis e libera a memória para ser utilizada novamente.

O quarto segmento da memória do Turbo Pascal, a pilha, é uma área de dados dinâmicos controlada por você. A pilha permite o uso eficiente da memória porque elimina a necessidade de preservar todas as estruturas de dados do programa. Em vez disso, você pode criar uma variável na pilha em um determinado ponto do programa, removê-la da pilha em outro ponto, e, depois, reutilizar o espaço para uma outra variável em um terceiro ponto.

O programa a seguir demonstra como os dados podem existir em qualquer um dos quatro segmentos do Turbo Pascal.

```
program segmentos;
  uses crt ;
  Type
   St4 = string[4];

var
  P : ^word ;
  x : word;

  function IntToHex(asegdes : word) : st4 ;
    const
       Caracterhex: array [0..15] of Char = '0123456789ABCDEF';
    begin
      InttoHex := caracterhex[Hi(asegdes) shr 4] +
                  caracterhex[Hi(asegdes) and 15] +
                  caracterhex[Lo(asegdes) shr 4] +
                  caracterhex[Lo(asegdes) and 15];
    end ;


Procedure MostraSegmentoDeCodigo ;
  begin
    Writeln('O segmento de código , ', IntToHex(Cseg));
    writeln;
  end;


Procedure MostraVariavelGlobal ;
  Begin
    Writeln('A posição da variável Global "X" ,
',IntToHex(seg(x)),':',IntToHex(ofs(x))) ;
    Writeln('Ela se encontra no segmento de dados');
    writeln;
  end;

Procedure MostraVariavelLocal ;
  var
    local : word ;
  Begin
    writeln('A posição da variável "Local" ,
',IntToHex(seg(local)),':',IntToHex(ofs(local)));
    writeln('Ela se encontra no segmento da área de armazenamento.');
    writeln;
  End;

Procedure MostraVariavelPonteiro;
  Begin
    Writeln('A posição da variável de ponteiro "P" ,
',IntToHex(seg(p^)),':',IntToHex(ofs(p^)));
    Writeln('Ela se encontra na Pilha');
    writeln;
  End;
```

```
Begin
  clrscr;
  writeln('Os endereços são mostrados no formato Segmento:Deslocamento.');
  writeln;
  new(P);
  MostraSegmentoDeCodigo ;
  MostraVariavelGlobal;
  MostraVariavelPonteiro;
  MostraVariavelLocal;
  writeln;
  Write('Pressione ENTER....');
  readln;
  dispose(P);
end.
```

Ao executar esse programa, seu terminal irá apresentar uma mensagem semelhante à seguinte:

Os endereços são mostrados no formato Segmento:Deslocamento.

O segmento de código é 1653

A posição da variável global “X” é 17C9:0066
Ela se encontra no segmento de dados.

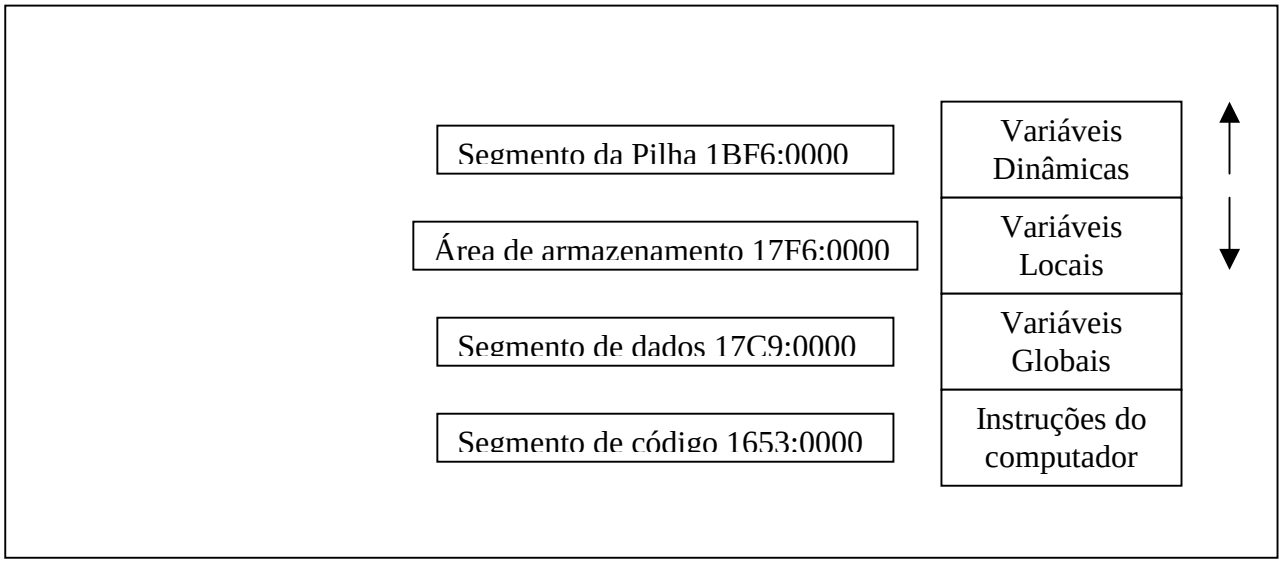
A posição da variável de ponteiro “P” é 1BF6:0000
Ela se encontra na pilha.

A posição da variável “Local” é 17F6:3FF6
Ela se encontra no segmento da Área de armazenamento.

Cada uma das quatro variáveis desse exemplo reside em um segmento diferente. A primeira posição, o segmento de código, é 1653. A variável **x** está localizada no segmento de dados (17C9). A variável de ponteiro **p**, posicionada na pilha com a instrução **New(p)**, está localizada no segmento 1BF6.

A última variável listada é uma variável local declarada em uma rotina. Todas as variáveis locais ficam guardadas na área de armazenamento e, nesse exemplo, o segmento da área de armazenamento começa em 17F6.

A Figura abaixo apresenta o diagrama da memória do Turbo Pascal. As linhas que separam os segmentos são combinadas com os valores hexadecimais do exemplo do programa anterior. O segmento de código ocupa os endereços mais baixos da memória, seguido dos segmentos de dados a da área de armazenamento. A pilha ocupa toda os endereços mais altos da memória restante, até o máximo definido por você com a diretiva **M** do compilador. O diagrama também demonstra que a área de armazenamento cresce para os endereços mais baixos e a pilha cresce para os endereços mais altos.



A Pilha a os Ponteiros

A maioria das variáveis que você declara no Turbo Pascal é estática, ou seja, a memória é alocada para elas do começo ao fim do programa. A pilha, por outro lado, utiliza tipos de dados dinâmicos conhecidos como ponteiros. As variáveis de ponteiro são dinâmicas porque você pode criá-las e dispor delas, durante a execução de um programa. Em resumo, diferentes variáveis de ponteiros podem usar e reutilizar a memória na pilha.

O uso de variáveis de ponteiro na pilha traz duas vantagens importantes. Primeiro, expande o espaço de dados total disponível para um programa. O segmento de dados está limitado a 64 kbytes, mas a pilha está limitada somente à área de memória RAM do seu computador.

A segunda vantagem de usar as variáveis de ponteiro na pilha é que elas permitem que o seu programa seja executado com menos memória. Por exemplo, um programa pode ter duas estruturas de dados bem grandes, mas somente uma delas é usada, de cada vez. Se essas estruturas de dados forem declaradas globalmente, elas residirão no segmento de dados a ocuparão a memória o tempo todo. Entretanto, se essas estruturas de dados forem definidas como ponteiros, poderão ser posicionadas na pilha e retiradas quando necessário, reduzindo, assim, os requisitos de memória do seu programa.

A Variável de Ponteiro

A variável de ponteiro não guarda dados da mesma maneira que as outras variáveis. Em vez disso, ela guarda o endereço que aponta pare uma variável localizada na pilha. Vamos supor que você tenha uma variável de ponteiro denominada **px** que guarda o endereço de uma variável **Integer**. Agora, você pode usar **px** para apontar para a variável **Integer**, mas a própria **px** não é uma variável Integer. Se você estiver confuso, o exemplo a seguir, que demonstra o uso simples de uma variável de ponteiro, poderá ajudá-lo. A seguinte listagem de programa demonstra o use simples de uma variável de ponteiro:

```
Program demonstraPonteiro ;
  uses crt ;
  Type
    st4 = string[4];

   function IntToHex(asegdes : word) : st4 ;
    const
       Caracterhex: array [0..15] of Char = '0123456789ABCDEF';
    begin
      InttoHex := caracterhex[Hi(asegdes) shr 4] +
                  caracterhex[Hi(asegdes) and 15] +
                  caracterhex[Lo(asegdes) shr 4] +
                  caracterhex[Lo(asegdes) and 15];
    end ;

  Var
    i : ^Integer ;
    j : Integer absolute i;

 Begin
   clrscr;
   New(i);
   i^ := 100;
   writeln('O valor de "I" ,: ',IntToHex(j));
   writeln('O valor para o qual "I" aponta ,: ',i^);
   dispose(i);
   writeln;
   write('Pressione ENTER...');
   readln;
 End.
```

O ^(circunflexo) colocado antes do tipo de dados na definição solicita ao Turbo Pascal que defina i com uma variável de ponteiro :

i: ^Integer;

Quando o programa é executado, a pilha encontra-se vazia. Antes de usar o ponteiro **i**, você deve utilizar a instrução **New(i)** para solicitar ao Turbo Pascal que atribua um endereço da pilha ao ponteiro **i**. **Dispose(i)**, que aparece próximo ao final do programa, é o oposto de **New(i)**. **Dispose** retira efetivamente uma variável da pilha, liberando memória para ser usada por outras variáveis.

Uma vez incluída na pilha, a variável pode ser usada em instruções de atribuição e aritméticas, acrescentando o símbolo ^ ao identificador, como é mostrado a seguir.

i^:= 100;

O ^ (circunflexo) informa ao Turbo Pascal que você está fazendo referência à variável contida na pilha e não ao próprio ponteiro. O que aconteceria se a instrução fosse **i := 100**? Essa instrução altera o valor do ponteiro, e não o valor da variável contida na pilha. Agora, i aponta para a posição 100 da memória, em vez de apontar para a sua própria posição.

Quando você executar o programa anterior, seu terminal apresentará a seguinte mensagem:

O valor de i é: 0000

O valor para o qual i aponta é: 100

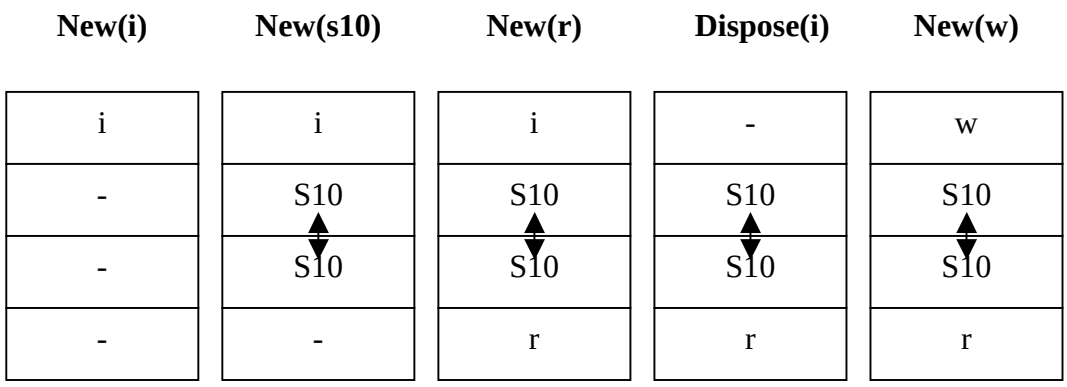
A primeira linha mostra o endereço que o ponteiro está guardando. Nesse caso, o endereço é 0000, indicando que essa variável é a primeira a ser colocada na pilha. A segunda linha é o valor de i^, a variável contida no endereço 0000 da pilha.

New e Dispose

Ao alocar a remover variáveis dinâmicas, o que realmente acontece na pilha? A Figura abaixo descreve o processo de alocação a desalocação. Na figura, o Turbo Pascal declara quatro variáveis de ponteiro -um integer. um Real, e duas String; uma com 5 e a outra com 10 bytes. As colunas representam a memória da pilha. O Turbo Pascal sempre aloca memória na pilha em quantidades exatas. Assim, quando o programa executa a instrução **New(i)**, a pilha fornece 2 bytes, que é o suficiente para armazenar a variável **Integer**.

Na segunda coluna, o Turbo Pascal aloca uma cadeia de 10 caracteres na pilha. Como essa cadeia exige 11 bytes de armazenamento (dez caracteres mais um byte), o Turbo Pascal aloca 11 bytes na pilha. Em resumo, a pilha sempre fornece a quantidade de bytes de memória necessários para conter a estrutura de dados colocada na pilha.

```
Type
  St10 = String[10];
  Var
    i :^Integer;
    r :^Real;
    s10 : ^St10;
    w :^Word;
```



A terceira coluna da Figura acima apresenta a alocação adicional de uma variável **Real**, exigindo 5 bytes de memória. A coluna a seguir demonstra o impacto da instrução **Dispose**. Quando o Turbo Pascal remove **i**, os dois bytes que **i** estava utilizando são liberados para uso por outras variáveis dinâmicas. Isso cria um "buraco" na pilha. Para usar novamente essa área de memória, a estrutura de dados deverá caber dentro desse buraco. Caso contrário, o Turbo Pascal deverá alocar memória em algum outro lugar na pilha.

Na quinta coluna, o Turbo Pascal aloca uma variável de palavra (tipo **Word**) na pilha. Como essa variável se encaixa no buraco deixado por i, o Turbo Pascal reutiliza essa memória.

Usar **New** e **Dispose** exige um planejamento cuidadoso e um teste rigoroso. Um erro comum é alocar a mesma variável na pilha. Por exemplo, as duas instruções seguintes,

New(i);
New(i);

Alocam uma variável **Integer** na pilha, mas apenas uma das variáveis Integer pode ser acessada como tal. Você não poderá somente acessar a primeira variável Integer como não conseguirá sequer se livrar dela. Como o ponteiro **i** aponta para a segunda variável **Integer**, ele não poderá ser usado para remover (com **Dispose**) a primeira variável. Certifique-se sempre de que cada **New** está combinado com um **Dispose**.

Mark a Release

O Turbo Pascal oferece uma alternativa quanto ao use de New e Dispose para alocar memória dinamicamente -**Mark** e **Release**. Em vez de deixar buracos na pilha como fazem **New** e **Dispose**, **Mark** e **Release** cortam uma extremidade inteira da pilha a partir de um determinado ponto em diante. Esse processo é demonstrado no programa a seguir.

```
Program LiberaPilha;
  uses crt;
  Type
   VetorCaracter = array[1..100] of char ;
  var
    topoDaPilha : ^word;
    v1,v2,v3    : ^VetorCaracter;
  Begin
    clrscr;
    writeln('Marcando o início da Pilha....');
    mark(topoDaPilha);
    writeln;
    writeln('Memória Inicial: ',MemAvail);
    writeln;
    writeln('------------------------------');
    writeln;
    new(v1);
    writeln('Memória livre depois de alocar "v1": ',MemAvail);
    new(v2);
    writeln('Memória livre depois de alocar "v2": ',MemAvail);
    new(v3);
    writeln('Memória livre depois de alocar "v3": ',MemAvail);
    writeln;
    writeln('------------------------------');
    writeln;
    writeln('Liberando toda memória alocada na Pilha....');
```

```
  release(topoDaPilha);
  writeln;
  writeln('Memória livre depois da liberação: ',MemAvail);
  writeln;
  write('Pressione ENTER...');
  readln;
End.
```

Esse programa aloca três variáveis de ponteiro - **v1 v2** e **v3** - e utiliza a função padrão **MemAvail** para mostrar a área de memória livre disponível. **MemAvail** mostra a área total, em bytes, de memória disponível nos retornos da pilha. Por exemplo, se **MemAvail** apresentar um valor igual a 20, significa que existem 20 bytes de memória disponível na pilha para uso em alocação dinâmica.

O programa anterior utiliza uma variável de ponteiro denominada **TopoDaPilha** para rastrear o ponto a partir do qual você libera memória. A instrução **Mark(TopoDaPilha)** armazena o endereço atual do topo da pilha para o ponteiro TopoDaPilha. O programa chama **Mark (TopoDaPilha)** antes de colocar quaisquer variáveis na pilha. Como resultado, ao chamar **Release(TopoDaPilha)** ele desloca todas as variáveis da pilha, liberando a memória para outro uso. A execução do programa resulta na seguinte mensagem:

Memória inicial: 536640[1]
Memória livre depois de alocar “v1”: 536536
Memória livre depois de alocar “v2”: 536432
Memória livre depois de alocar “v3”: 536328
Memória livre depois da liberação: 536640

Como você pode observar, toda vez que uma variável é colocada na pilha, a área de memória disponível diminui. Quando **Release(TopoDaPilha)** é chamada no final do programa, a área de memória livre volta à área inicial. Se você tivesse marcado o ponteiro **TopoDaPilha** depois de v1 ser alocada, somente a memória para v2 e v3 seria liberada.

Observe que **Dispose** e **Release** são métodos incompatíveis de recuperação de memória. Você pode optar por um ou outro, mas nunca usar ambos em um mesmo programa.

GetMem e FreeMem

Um terceiro método de alocação de memória dinâmica é **GetMem** e **FreeMem**. Esse é muito parecido com **New** e **Dispose** no sentido de que aloca e desaloca memória com uma variável por vez. A grande vantagem de **GetMem** e **FreeMem** é que você pode especificar a área de memória que desejar alocar independente do tipo de variável utilizada. Por exemplo, você pode alocar 100 bytes para uma variável Integer com a seguinte instrução:

```
GetMem(i,100);
```

As variáveis alocadas com **GetMem** são desalocadas com **FreeMem**, como é mostrado a seguir.

```
GetMem(i,20);
```

[1] A quantidade de memória apresentada depende da quantidade real que o computador tem.

```
 i^:= x + y;
WriteLn(i^);
FreeMem(i,20);
```

O número de bytes especificado na instrução **FreeMem** deve coincidir com o da instrução **GetMem**. Não use **Dispose** no lugar de **FreeMem**. Se você fizer isso, a pilha se tornará irremediavelmente dessincronizada.