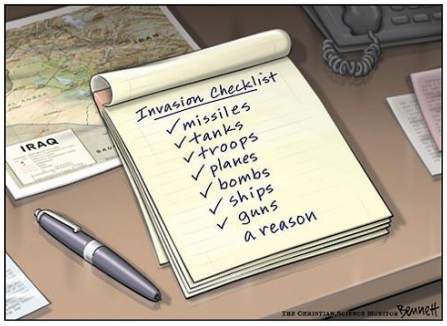
Listas Dinâmicas em C

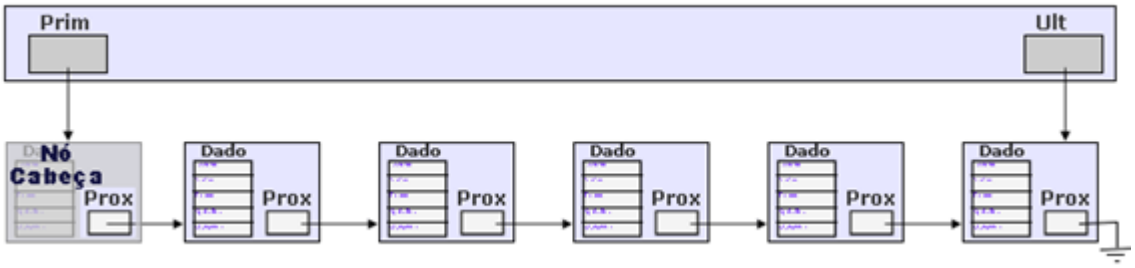


Tendo em vista que a Linguagem C proporciona mecanismos de alocação dinâmica de memória, porque limitar a lista a um vector de tamanho definido?
Tornasse interessante o facto de utilizar a memória disponível do computador para montarmos uma lista.

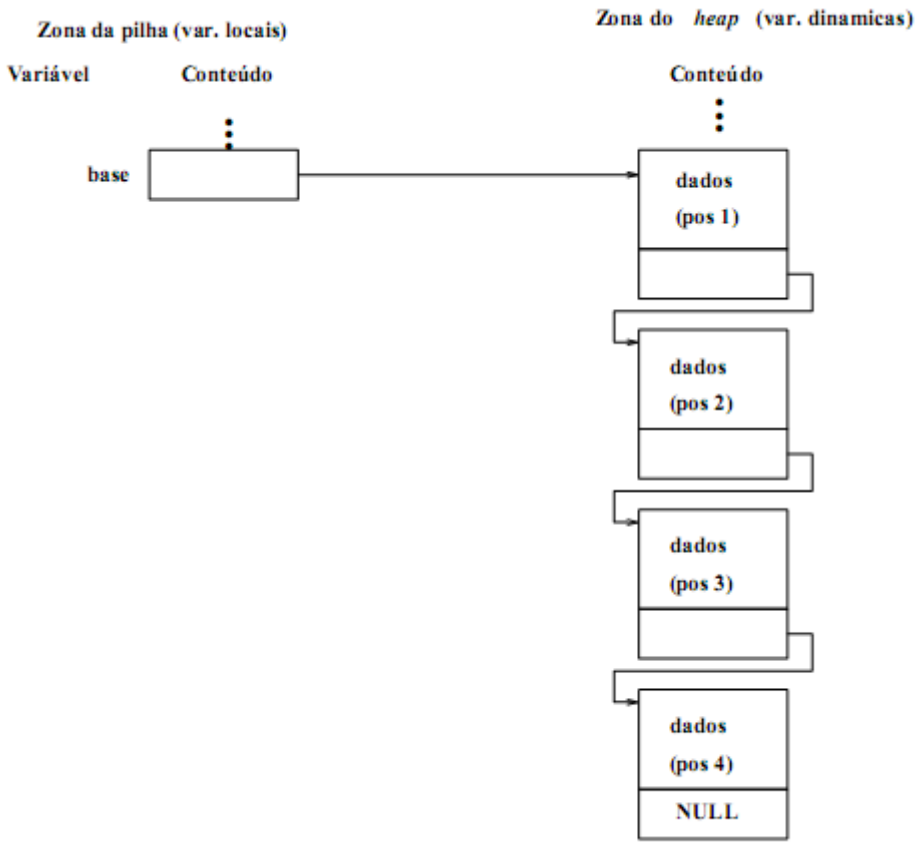
Para que isto seja possível temos que ter em mente algumas coisinhas:

1. Não vamos precisar de um vector para a estrutura de dados da lista.
2. Cada item da lista terá um ponteiro para o próximo item.
3. Acessaremos a lista a partir do início dela (cabeça). Este item estará num ponteiro.
4. Para modificarmos a lista, devemos passar o primeiro item por referência, para que toda alteração nele reflicta no contexto geral. Logo teremos um ponteiro de ponteiro.
5. Se a lista está vazia, o ponteiro da cabeça será nulo.
6. O último elemento terá o ponteiro nulo, pois não há próximo item. Para isso damos o nome de Terra.

Graficamente, podemos representar uma lista assim:



Como havia dito, teremos apenas um ponteiro para a cabeça, sendo assim, observe a figura a seguir e acompanhe o raciocínio já proposto:



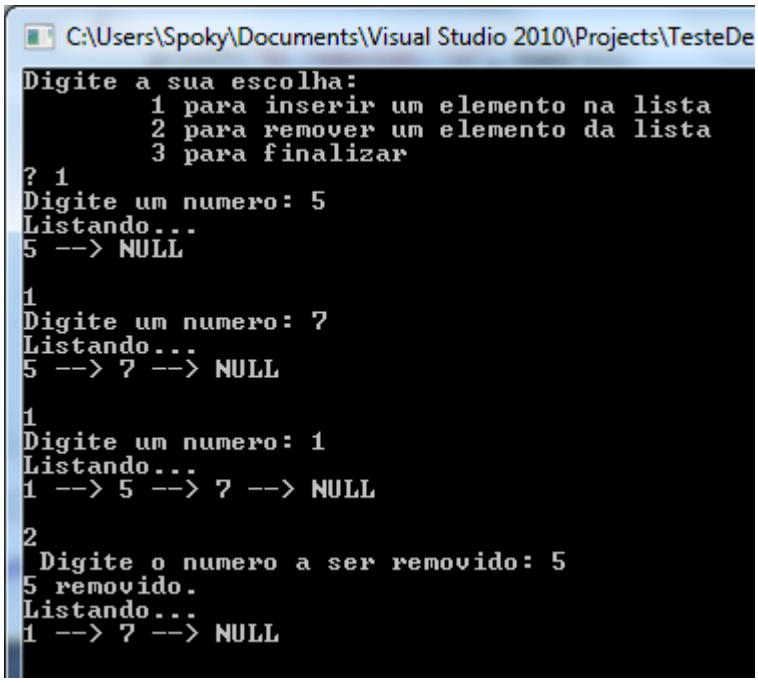
Então é fundamental estar bem afiado em ponteiros para implementar uma estrutura de dados desta. Vamos à uma breve implementação de uma lista dinâmica:

```
#include <stdio.h>
#include <stdlib.h>
struct Item
{
   int numero;
   struct Item *proximo;
};
bool Inserir(Item **cabeca, int valor)
{
   Item *novo, *anterior, *actual;
   // -> Aloca memória para o novo item
   novo = (Item *)malloc(sizeof(Item));
   // -> Checa se foi alocado memória correctamente.
   if (novo != NULL)
   {
      // -> Define e inicializa os valores.
      novo->numero = valor;
      novo->proximo = NULL;
      anterior = NULL;
      actual = *cabeca; // -> Actual é o ponteiro para o item da cabeça
      // -> Ordenação
      //   Verifica se a posição actual não é nula (início ou fim)
      //   Verifica se o valor é maior do que o número do item.
```

```
      while (actual != NULL && valor > actual->numero)
      {
         anterior = actual;
         actual = actual->proximo;
      }
      // -> Se não há posição anterior, então o novo item será a cabeça.
      if (anterior == NULL)
      {
         novo->proximo = *cabeca; // -> recebe o que já tem na cabeça.
         *cabeca = novo; // ->  cabeça nova.
      }
      else // -> Arranjando posição no meio da lista... Remanejando.
      {
         // -> Elemento anterior aponta para o novo.
         anterior->proximo = novo;
         // -> Novo elemento aponta para o resto da lista.
         novo->proximo = actual;
      }
      return true;
   }
   else
   {
      return false;
   }
}
bool Remover(Item **cabeca, int valor)
{
   Item *anterior, *actual, *itemRemover;
   anterior = NULL;
   actual = *cabeca;
   // -> Procura o valor.
   while(actual != NULL && actual->numero != valor)
   {
      anterior = actual;
      actual = actual->proximo;
   }
   // -> Quando encontra o valor.
   if(actual != NULL)
   {
      itemRemover = actual;
      if(anterior != NULL)
      {
         // -> Encurta o caminho.
         anterior->proximo = actual->proximo;
      }
      else // -> Quando o item é a cabeça.
      {
         // -> Ela recebe o próximo do primeiro item.
         *cabeca = (*cabeca)->proximo;
      }
      // -> Libera da memória o item excluído.
      free(itemRemover);
      return true;
   }
   return false;
}
```

```
bool EstaVazia(Item *cabeca)
{
   return cabeca == NULL;
}
void Listar(Item *cabeca)
{
   if (EstaVazia(cabeca))
   {
      printf( " A lista esta vazia.\n\n" );
   }
   else
   {
      printf( "Listando... \n");
      while(cabeca != NULL)
      {
         printf("%i --> ", cabeca->numero);
         cabeca = cabeca->proximo;
      }
      printf("NULL\n\n");
   }
}
void Menu()
{
   printf( "Digite a sua escolha: \n"
      "    1 para inserir um elemento na lista \n"
      "    2 para remover um elemento da lista \n"
      "    3 para finalizar \n"
      "? ");
}
void main()
{
   Item *cabeca = NULL;
   int opcao;
   int numero;
   Menu();
   scanf("%i", &opcao);
   while (opcao != 3)
   {
      switch (opcao)
      {
         case 1:
            printf( "Digite um numero: ");
            scanf("\n%i", &numero);
            Inserir(&cabeca, numero);
            Listar(cabeca);
            break;
         case 2:
            if (!EstaVazia(cabeca))
            {
               printf( " Digite o numero a ser removido: ");
               scanf( "\n%i", &numero);
               if (Remover(&cabeca, numero))
               {
                  printf("%i removido. \n", numero);
                  Listar(cabeca);
               }
```

```
            else
            {
               printf(" %c não encontrado. \n\n", numero);
            }
         }
         else
         {
            printf("A Lista está vazia. \n\n");
         }
         break;
      default:
         printf( "Escolha invalida.\n\n");
         Menu();
         break;
    }
    scanf("%i", &opcao);
  }
  printf("Fim do programa.\n");
}
```



OK, concordo que não foi tão breve assim…
Mas, este é o exemplo de uma lista de inteiros ordenada!

Antes de tudo temos um novo operador, o: –>
Este operador é um indicador, ou seja, indica directo para o dado dentro do ponteiro, sem o uso do “*”.
Ele é o equivalente a fazermos: (*novo).numero = valor;

Explicando então este código:

- Como precisamos de uma estrutura que contenha o dado e um ponteiro para o próximo dado, criei então uma *struct* Item.
- O booleano *Inserir()* retorna o sucesso da inserção do elemento na lista.
- Ele também insere de forma a deixar a lista sempre ordenada.
- O booleano Remover() também retorna o sucesso da operação.
- Ambos recebem o primeiro item (cabeça) por referência, por isto temos um ponteiro de ponteiro (já que o primeiro item já é um ponteiro).

- Para saber se a lista está vazia, basta chegar se a cabeça é nula.
- Para listar, basta ir de próximo em próximo item, até encontrar o terra (nulo).