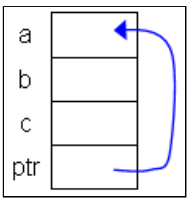
Programação C/C++

Apontadores/ Ponteiros/ Pointers

Página diagramada pelo Estagiário de Docência Rafael Rieder

Um ponteiro é uma variável capaz de armazenar um endereço de memória ou o endereço de outra variável.

```
#include <stdio.h>
void main()
{
  int a;
  int b;
  int c;
  int *ptr;  // declara um ponteiro para um inteiro
             // um ponteiro para uma variável do tipo inteiro
  a = 90;
  b = 2;
  c = 3;
  ptr = &a;
  printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
  printf("B: %d, C: %d"), b, c);
}
```



Impressão de Ponteiros

Em C, pode-se imprimir o valor armazenado no ponteiro (um endereço), usando-se a função **`printf`** com o formatador **`%p`** na string de formato. Por exemplo:

```
#include <stdio.h>
void main()
{
  int x;
  int *ptr;
  ptr = &x;
  printf("O endereço de X é: %p\n", ptr);
}
```

Em C++, usa-se o objeto **cout** normalmente. Por exemplo:

```
#include <iostream>
using namespace std;
void main()
{
  int x;
  int *ptr;
  ptr = &x;
  cout << "O endereço de X é: " << ptr << endl;
}
```

Acessando o Conteúdo de uma Posição de Memória através de um Ponteiro

Para acessar o conteúdo de uma posição de memória, cujo endereço está armazenado em um ponteiro, usa-se o operador de **derreferência** (*). Por exemplo:

```
#include <stdio.h>
void main()
{
  int x;
  int *ptr;
  x = 5;
  ptr = &x;
  printf("O valor da variável X é: %d\n", *ptr);  // derreferenciando um ponteiro
  *ptr = 10;  // usando derreferencia no "lado esquerdo" de uma atribuição
  printf("Agora, X vale: %d\n", *ptr);
}
```

Um ponteiro derrefenciado também pode ser usado à esquerda de uma atribuição, para receber um valor. O mesmo exemplo, em C++ é apresentado abaixo:

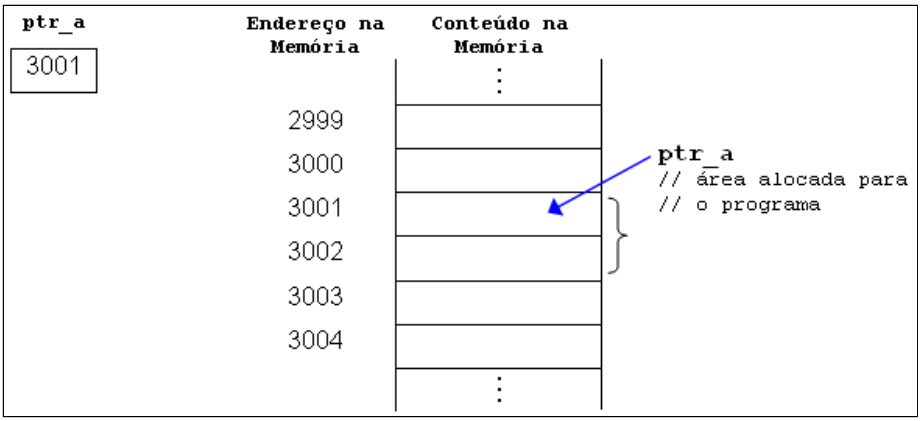
```
#include <iostream>
using namespace std;
void main()
{
  int x;
  int *ptr;
  x = 5;
  ptr = &x;
  cout << "O valor da variável X é: " << *ptr << endl;
  *ptr = 10;
  cout << "Agora, X vale: " << *ptr << endl;
}
```

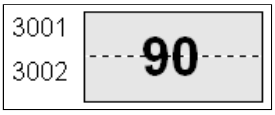
Alocação Dinâmica de Memória

Durante a execução de um programa, pode-se alocar dinamicamente memória para usar como variáveis do programa. Em C, a alocação é feita com a função **malloc** (e, para liberar memória, usa-se a função **free**). Em C++, faz-se o mesmo tipo de alocação usando o operador **new** (para liberar memória, operador **delete**). No exemplo a seguir, pode-se observar o uso das funções **malloc** e **free**:

```
#include <stdlib.h>
#include <stdio.h>

void main()
{
 int *ptr_a;

 ptr_a = malloc(sizeof(int));
 // cria a área necessária para 01 inteiro e
 // coloca em 'ptr_a' o endereço desta área.

  if (ptr_a == NULL)
  {
    printf("Memória insuficiente!\n");
```

```
    exit(1);
  }

  printf("Endereço de ptr_a: %p\n",  ptr_a);
  *ptr_a = 90;
  printf("Conteúdo de ptr_a: %d\n", *ptr_a);  // imprime 90
  free(ptr_a);  // Libera a área alocada
}
```





Conforme citado acima, em C++ pode-se usar o operador **`new`**. O exemplo abaixo faz o mesmo que o exemplo utilizando **`malloc`**:

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

void main()
{
 int *ptr_a;

 ptr_a = new int;
 // cria a área necessária para 01 inteiro e
 // coloca em 'ptr_a' o endereço desta área.

  if (ptr_a == NULL)
  {
    cout << "Memória insuficiente!" << endl;
    exit(1);
  }
  cout << "Endereço de ptr_a: " <<  ptr_a << endl;
  *ptr_a = 90;
  cout << "Conteúdo de ptr_a: " << *ptr_a << endl;
  delete ptr_a;
}
```

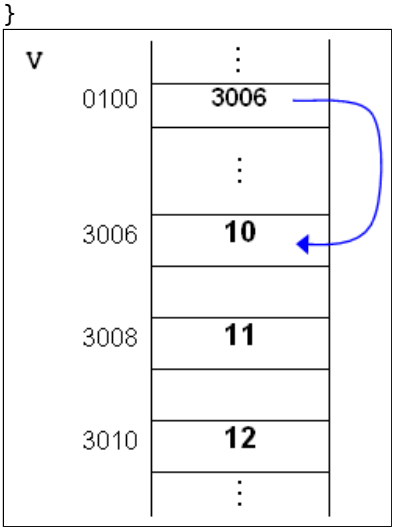
Alocação de Vetores

Nos exemplos anteriores, foi alocado um espaço para armazenar apenas um dado do tipo **`int`**. Entretanto, é mais comum utilizar ponteiros para alocação de vetores. Para tanto, basta especificar o tamanho desse vetor no momento da alocação. Nos exemplos abaixo, apresenta-se a alocação de vetores com **`malloc`** e **`new`**. Após a alocação de uma área com vários elementos, ela pode ser acessada exatamente como se fosse um vetor.

Exemplo em C:

```
#include <stdlib.h>
void main()
{
  int i;
```

```
  int *v;
  v = (int*)malloc(sizeof(int)*10);  // 'v' é um ponteiro para uma área que
                                     // tem 10 inteiros.
                                     // 'v' funciona exatamente como um vetor
  v[0] = 10;
  v[1] = 11;
  v[2] = 12;
  // continua...
  v[9] = 19;

  for(i = 0; i < 10; i++)
    printf("v[%d]: %d\n", i, v[i]);

  printf("Endereço de 'v': %p", v);  // imprime o endereço da área alocada para 'v'
  free(v);
}
```



Exemplo em C++:

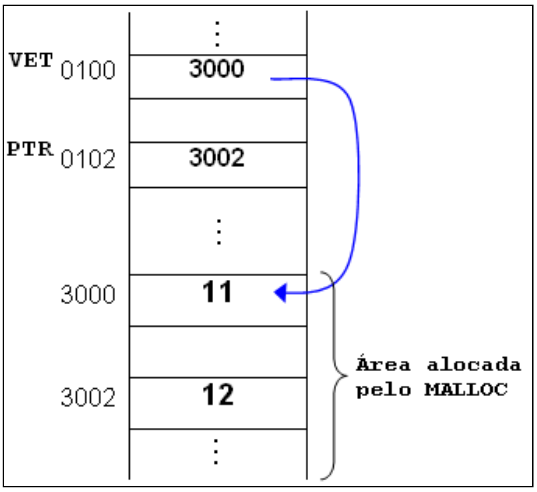
```
#include <iostream>
using namespace std;

void main()
{
  int i;
  int *v;
  v = new int[10];  // 'v' é um ponteiro para uma área que
                    // tem 10 inteiros.
                    // 'v' funciona exatamente como um vetor
  v[0] = 10;
  v[1] = 11;
  v[2] = 12;
  // continua...
  v[9] = 19;

  for(i = 0; i < 10; i++)
    cout << "v[" << i << "]: " << v[i] << endl;

  cout << "Endereço de 'v': " << v << endl; // imprime o endereço da área alocada para 'v'
  delete[] v;
}
```

Aritmética de Ponteiros

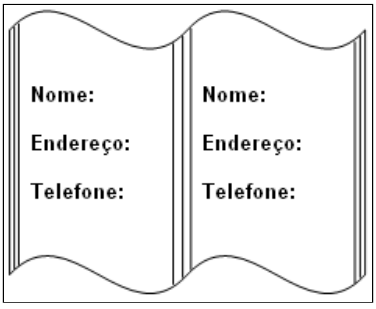
Além de acessar os dados de uma área alocada dinâmicamente como se fosse um vetor, é possível acessá-los através do próprio ponteiro, utilizando a técnica chamada **aritmética de ponteiros.**

```
#include <stdlib.h>
void main()
{
  int *vet;
  int *ptr;
```

```
  vet = (int*)malloc(sizeof(int)*10);
  vet[4] = 44;  // 'vet' funciona como um vetor, depois de malloc

  ptr = vet;  // 'ptr'  aponta para o início da
              // área alocada por 'vet

  *ptr = 11;  // vet[0] = 11
              // coloca 11 na primeira posição da área alocada

  ptr++;      // avança o apontador
  *ptr = 12;  // vet[1] = 12

  printf("%p\n",  ptr);
  printf("%d\n", *ptr);

  // free(ptr);  // Liberar 'ptr' direto causa NULL POINTER ASSIGNMENT
  // Corrigindo, a forma correta é:
  ptr--;
  free(ptr);
}
```



Alocação Dinâmica de *Structs*



Quantos funcionários eu terei?

Para casos em que não se sabe quantos elementos se tem em uma lista, usa-se **alocação dinâmica**.

```
typedef struct P
{
  char nome[30];
  char endereco[40];
  char telefone[10];
} TPESSOA;
```

Realocação de Memória

Se for preciso aumentar o tamanho do vetor, use:

```
realloc(ptr, novo_tam);
```

Exemplo:

```
// código (...)
cadastro = (TPESSOA*)realloc(cadastro, 20);
ptr = cadastro;
// a área de memória é realocada, e todos os dados são copiados da área antiga para a área nova.
```

Testes de Alocação

Para verificar se foi possível alcar a memória solicitada, pode-se testar o valor do ponteito usado para armazenar a área alocada. Se o valor deste ponteiro for NULL, não foi possível alocar a memória.

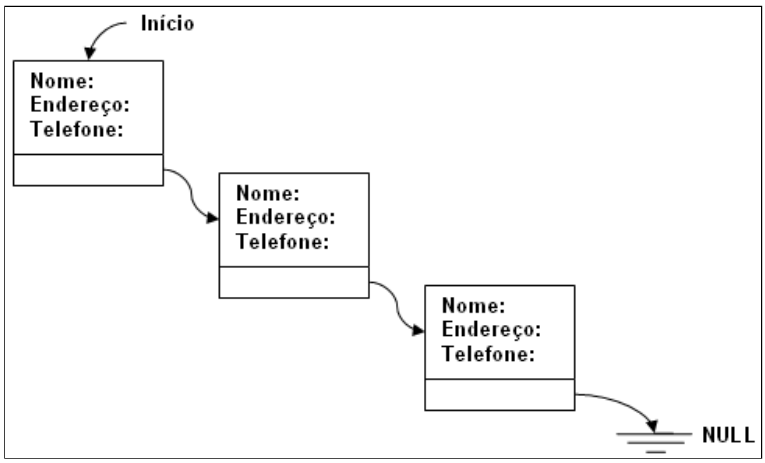
```
int *ptr;
ptr = (int*)malloc(sizeof(int)*10);
if (ptr == NULL)
{
  printf("Não foi possível alocar memória!\n");
  exit(1);
}
```

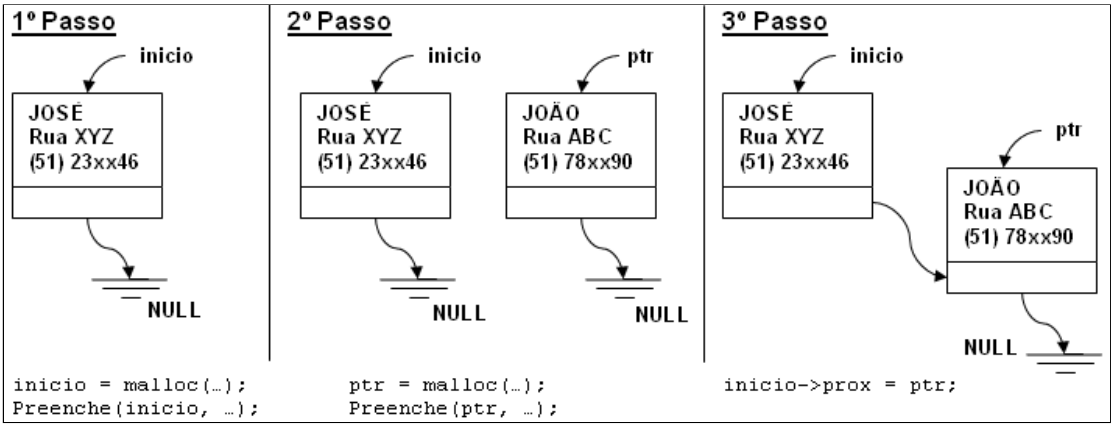
Em operações de realocação de memória, também é necessário realizar este procedimento.

Estruturas Encadeadas

Para crira estruturas encadeadas em C ou C++ costuma-se utilizar variáveis do tipo ponteiro. Em um struct, coloca-se um **ponteiro** para o próximo (ou anterior) elemento da lista da seguinte forma:

```
typedef struct temp
{
    string nome;
    int idade;
    int rg;
    temp *prox; // apontador para o próximo elemento
}TPESSOA;
```

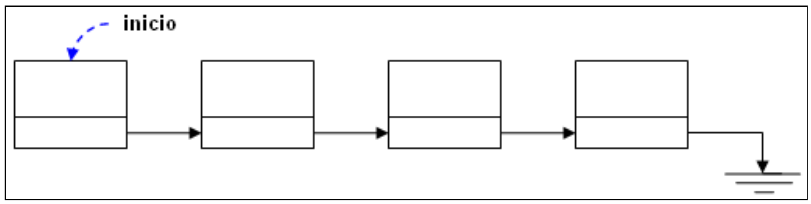




Exemplo 1: Inserção no Início da Lista

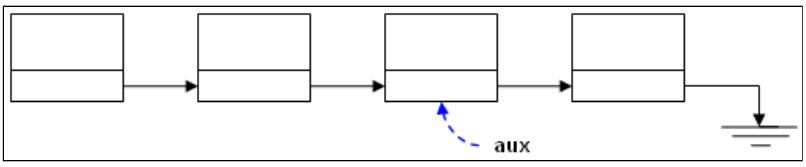
```
// (...)

TPESSOA *inicio, *ptr;

ptr = (TPESSOA*)malloc(sizeof(TPESSOA));
// Teste de alocação
if (ptr == NULL)
{
  printf("Não foi possível alocar memória!\n");
  exit(1);
}

// Lê os dados
cin >>idade;
cin >>nome;

// Coloca os dados lidos na struct
P->idade = idade;
strcpy(P->nome, nome);

// Marca o nodo como último da lista
P->prox = NULL;

//Insere no início da lista
inicio = ptr;
```

Exemplo 2: Busca de um Elemento na Lista




```
if (inicio == NULL)
    return; // lista vazia
aux = inicio;
do
{
  if (aux->nome == dado)
  {
    achou = 1;
  }
  else
  {
    aux = aux->prox;
  }
} while( (achou == 0) && (aux != NULL) );
// 'aux' aponta para o nodo que tem a informação
```

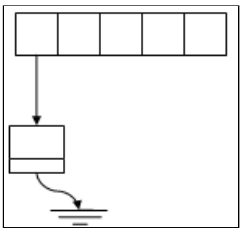
Exemplo 3: Retirada de um Elemento no Meio da Lista




```
// Verifica se o elemento buscado está no início da lista

if(inicio->nome==dado)
    {
        inicio=inicio->prox;
        aux->prox=NULL;
        free (aux);
        return 1;
    }

// Se não estiver no início, busca no restante
// da lista

antes = inicio;
do
{
  aux = antes->prox;
  if (aux->nome == dado)
  {
    achou = 1;
  }
  else
  {
    antes = aux;
    aux = aux->prox;
  }
} while( (achou == 0) && (aux != NULL) );

if (achou == 0)
    return 0;  // não achou ....

// Se achou == 1, então 'aux' aponta para o nodo que tem a informação
// e antes, aponta para o nodo anterior
antes->prox = aux->prox; // retira o aux da lista
free(aux);
return 1;
```

Vetor de Ponteiros

```
typedef struct temp
{
  int dado;
  struct temp *prox;
} TNODO;
```



```
TNODO *listas[5];

listas[0] = new TNODO;
listas[0]->prox = NULL;
```

```
aux = listas[0];
aux->prox = NULL;
```

Ponteiros para Objetos

Consulte a página sobre Orientação a Objetos, para maiores detalhes.

```
class CPessoa
{
  // ...
  void setNome(char *dado);
 // ...
}

CPessoa *Fulano;

Fulano = new CPessoa;
Fulano->setNome("Fulano de Tal");
```