

Aula 23

MC 102 - Algoritmos e Programação de Computadores

Apontadores e Alocação Dinâmica.

Variáveis

Ao declararmos uma variável x como a abaixo:

```
int x = 100;
```

Temos associados a ela os seguintes elementos:

Um nome: x ;

Um endereço de memória ou referência: `0xbf267c4`;

Um valor: 100

Para acessarmos o endereço dessa variável usamos o operador `&`

```
int x = 100;
```

```
printf("Valor de x = %d\n", x);
```

```
printf("Endereco de x = 0x%x\n", &x);
```

Apontadores

Nós já vimos que existem tipos de dados para armazenar o endereço de uma variável. Eles usam o operador `*`.

Uma variável declarada como um destes tipos é chamada de **apontador**.

Ao atribuir o endereço de uma variável a um apontador, dizemos que o mesmo **aponta** para a variável.

```
int x;
```

```
int *ap_x; /* apontador para inteiros */
```

```
ap_x = &x; /* ap_x aponta para x */
```

Apontadores

Para declarar uma variável do tipo apontador utilizamos o operador unário `*`.

```
int *ap_int;
```

```
char *ap_char;
```

```
float *ap_float;
```

```
double *ap_double;
```

Apontadores

Cuidado ao declarar vários apontadores em uma única linha. O operador * deve preceder o nome da variável e não suceder o tipo que o apontador apontará.

```
int *ap1, *ap_2, *ap_3;
```

A declaração abaixo declara quantos inteiros e quantos apontadores para inteiro?

```
int *ap1, ap_ou_int1, ap_ou_int2;
```

Fazendo acesso aos valores das variáveis referenciadas

Um endereço de variável por si só não é muito útil. Para acessarmos o valor de uma variável apontada por um endereço, também usamos o operador *:

Ao precedermos um apontador com este operador, obtemos o equivalente a variável armazenada no endereço em questão:

*ap_x pode ser usado em qualquer contexto que a variável x seria.

```
int x;
int *ap_x = &x;
*ap_x = 3;
```

Apontadores para registros

Para acessar os elementos de um registro usando um apontador, devemos primeiro acessar o registro e depois acessar o campo desejado.

Os parênteses são necessários pois o operador * tem prioridade menor que o operador .

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;
Ponto *ap_p, r;
ap_p = &r;
(*ap_p).x = 4.0;
(*ap_p).y = 5.0;
```

Apontadores para registros

Para simplificar o acesso aos campos de um registro usando apontadores, foi criado o operador ->.

Usando este operador acessamos os campos de um registro diretamente pelo apontador.

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;
Ponto *ap_p, r;
ap_p = &r;
ap_p->x = 4.0;
ap_p->y = 5.0;
```


Alocação Dinâmica de Memória

Já vimos que as matrizes e vetores possuem tamanhos pré-definidos, ocupando a memória como outra variável qualquer. Isso gera o incômodo de definirmos um tamanho muito maior que aquele que normalmente será usado em tempo de execução.

Mas como definir um tamanho dinamicamente, ou seja em tempo de execução, conforme a necessidade do usuário, desde que haja memória disponível?

Alocação Dinâmica de Memória

Isso é feito utilizando a chamada memória dinâmica, oferecida pelo sistema operacional, durante a execução do programa.

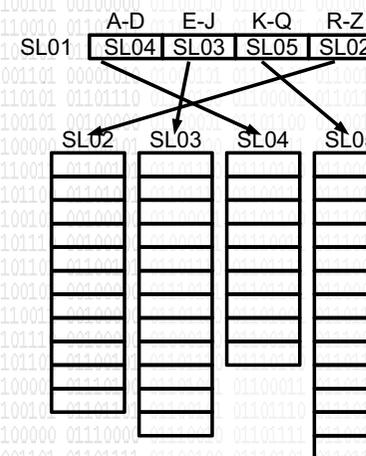
As variáveis globais e locais são armazenadas na **Memória Estática**. A **Memória Dinâmica** é utilizada para o armazenamento de dados, “desafogando” a memória estática, mas não de variáveis.

Memória Estática x Dinâmica

Para podermos usar a memória dinâmica, precisamos saber o **endereço** de memória em que os dados serão armazenados.

É como se uma variável, em vez de armazenar os dados propriamente ditos, possui apenas o endereço da localização dos dados (uma caixa postal para as correspondências, por exemplo).

Memória Estática x Dinâmica



Vamos fazer uma analogia. Imagine que temos poucos lugares em uma sala para atender a todos os alunos que para uma prova de vestibular. Essa sala será reservada para informações aos alunos da localização da correta sala na qual ele fará a prova. Essas “novas salas” foram alocadas após as inscrições, quando o número total de vestibulandos já era conhecido.

Biblioteca stdlib.h - Alocação 1

Contém as funções para alocação e liberação dinâmica de espaços de memória:

Alocar memória:

```
void *malloc(int tamanho_em_bytes);
```

void * significa que o retorno é um endereço de memória genérico, não importante o tipo de dado.

Exemplo:

```
int *x; // vetor criado dinamicamente
Obrigatório
x = (int *) malloc(sizeof(int) * 10000);
x[9876] = 1234;
...
```

Biblioteca stdlib.h - Alocação 2

Alocar memória e zerar o conteúdo alocado:

```
void *calloc(int num_elem, int tamanho_elem);
```

num_elem é o numero de elementos que serão criados.

tamanho_elem é o tamanho de cada elemento que será criado.

Exemplo:

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
...
```

Biblioteca stdlib.h - Alocação 3

Realocar memória (para mais ou para menos):

```
void *realloc((void *) apontador, int novo_tam);
```

apontador é a variável apontador que será alterada. Note que (void*) deve ser usado obrigatoriamente, independente do tipo de dado.

novo_tam é o novo tamanho alocado, em bytes. Se menor, o restante será perdido.

Exemplo:

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
x = (int *) realloc((void *) x, 1000 * sizeof(int));
...
```

Biblioteca stdlib.h - Liberação

Liberar memória alocada:

```
void free(void * apontador);
```

apontador é a variável apontador que contem o endereço para ser liberado.

É muito importante liberar a memória que não será mais usada. Isso evita o esgotamento da memória dinâmica oferecida pelo sistema operacional.

Exemplo:

```
int *x; // vetor criado dinamicamente
x = (int *) calloc(10000, sizeof(int));
x[9876] = 1234;
...
free(x); // libera todo espaço alocado
```

Resumo

- Definição de ponteiros para alocação dinâmica de memória.
- Funções da `stdlib.h` para alocação dinâmica de memória: `malloc`, `calloc`, `realloc` e `free`.

Importante!

- Não se pode usar um ponteiro sem primeiro alocar memória para ele.
- Nunca esquecer de liberar a memória que não será mais usada, pois uma hora, ela pode acabar...

Exercício

Faça um programa que leia um vetor de n inteiros e imprima sua média. Agora, o vetor deverá ser alocado dinamicamente para qualquer quantidade inteira n que o usuário informar (com $n > 0$ e $n < 50$).