Alocação Dinâmica

Ainda não terminamos nosso estudo sobre vetores, combinado? Vamos falar sobre algumas limitações que eles podem nos trazer? Bom, imagine a situação: Você alocou um vetor x de tamanho 50 e no meio da execução do seu programa é necessário que este vetor passe a ter um tamanho maior, por exemplo, 100. Ou imagine, que dado uma situação, seu vetor deva ser menor, e você quer economizar a memória que já havia alocado antes. Imagine uma terceira situação, queremos alocar um vetor de tamanho informado pelo usuário.

O QUE FAZER? *Vetores têm tamanhos estáticos. Seu tamanho é declarado no ato da compilação.*

Temos uma solução! Fazemos uso da **alocação dinâmica de memória**. Que, como o próprio nome já revela, a alocação é realizada dinamicamente, na execução do programa. Nos permitindo alocar ou realocar espaços à medida que for necessário. Desta forma, o processo se tornar mais otimizado.

Então, vamos lá conhecer algumas funções importantes na alocação dinâmica.

sizeof

Revela a quantidade de bytes ocupados por uma variável. Possui duas sintaxes:

```
sizeof (nome_variável)
ou
sizeof nome_variável
```

Então, podemos tanto obter o tamanho em bytes de uma variável simples:

```
int main()
{
        int x;
        printf("%d", sizeof (x));
}
```

Indicará 4 bytes, pois 1 inteiro equivale a 4 bytes

De um vetor:

```
#define T 5
int main()
{
        int vetor[T]={};
        printf("%d", sizeof vetor);
}
```

Indicará 20 bytes, pois o vetor possui tamanho 5

Como também de um tipo de dado:

```
int main()
{
        printf("%d", sizeof (double));
}
```

Indicará 8 bytes, pois 1 double equivale a 8 bytes

Malloc

É uma função pertencente a biblioteca <stdlib.h>, tem como finalidade alocar dinamicamente um número de bytes de memória e retorna um ponteiro genérico (do tipo void) para o início do espaço de memória alocado. Sua assinatura (ou protótipo) é dada por:

```
void *malloc (unsigned int num)
```

O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro, por exemplo: **int***, ou **float***, ou **double***, entre outros. Porém, nem sempre é possível alocar o espaço de memória que desejamos, para estes casos a função malloc retorna o valor nulo (**NULL**).

Por exemplo, vamos tentar alocar um espaço de memória para um inteiro. Quantos bytes tem um inteiro? Precisamos decorar? Não né, conhecemos a função sizeof. Então basta fazer: sizeof(int).

Vamos tentar alocar um espaço de memória, sabendo que malloc retorna um ponteiro e que recebe o tamanho em bytes da memória a ser alocada, temos:

```
int *p;
p = malloc (sizeof(int));
printf("Endereço de p = %x", p);
```

Mas isso "roda"? Bom, segundo as regras da linguagem C, não. E porque não? Ora, porque na definição de malloc ela retorna um ponteiro genérico (**void***), lembra? E no exemplo anterior queremos que o espaço de memória seja para armazenar um inteiro (**int***), o que fazer?

Vamos "dar um cast". Você pode perguntar: "um o quê?"

O cast é um modelador, que quando aplicado a uma expressão, "força" (quando possível) ela a ser de um tipo específico.
Sua sintaxe é dada por:

(tipo) expressão

Ou seja, no exemplo anterior queremos que **p** seja **inteiro**, então vamos *"dar um cast"*:

```
int *p;
p = (int*) malloc (sizeof(int));
printf("Endereço de p = %x", p);
```

Opa! Mas malloc pode retornar um valor nulo, caso não consiga alocar a memória, né? Então devemos verificar se p é nulo antes de imprimir o valor do endereço alocado:

```
int main()
{
	int *p;
	p = (int*) malloc (sizeof(int));
	if(p!= NULL)
		printf("Endereço de p = %x", p);
	else
		printf("Memoria nao alocada");
}
```

Free

Bom, aprendemos alocar um espaço de memória dinamicamente, mas e quando não queremos mais de um espaço de memória? Bom, basta utilizar a função free, também de <stdlib.h> para liberar.

Sua assinatura (ou protótipo) é dado por:

```
void free (void *p)
```

Um exemplo de seu uso pode ser visto, como:

```
int main()
{
	int *p;
	p = (int*) malloc (sizeof(int));

	if(p!= NULL){
		free(p);
		printf("Memoria liberada");
	}
	else
		printf("Memoria nao alocada");
}
```

Alocando mais de um espaço de memória. De forma homogênea e sequencial

Bom, aprendemos alocar dinamicamente um espaço de memória, mas será que conseguimos alocar mais um espaço? Se sim, será que eles ficar 'lado a lado' após a alocação? *Sim, é possível e sim, eles ficam lado a lado*. Logo, isso nos faz lembrar dos vetores. Não é mesmo? Show! Como fazemos?

Ora, como era a sintaxe de malloc para alocar só um espaço?

```
void *malloc (unsigned int num)
```

Basta multiplicarmos o número de bytes que queremos alocar, ou seja:

```
void *malloc (n * num)
```

Vamos utilizar um exemplo?

Imagine alocar dinamicamente um espaço de 50 inteiros para um vetor, utilizando um ponteiro x:

```
int main()
{
        int *x;
        x = (int*) malloc (50*sizeof(int));

        if(x!= NULL){
                printf("Vetor Alocado em x");
        }
        else
                printf("Memoria nao alocada");
}
```

Indicando a quantidade de 50 espaços sequenciais para alocação de memória dinâmica

Para utilizá-lo, basta manipular o espaço alocado em x como um vetor:

```
int main()
{
        int *x;
        int i;

        x = (int*) malloc (50*sizeof(int));

        if(x!= NULL){
                printf("Vetor Alocado em x\n");
                printf("Preechendo x:\n");
                for(i=0; i<50; i++){
                        printf("Informe um valor para o indice %d: ", i);
                        scanf("%d", &x[i]);
                }
                printf("Imprimindo x:");
                for(i=0; i<50; i++)
                        printf("%d ",x[i]);
        }
```

Manipulando o espaço de memória alocado para o ponteiro x, fazendo uso dos índices de memória (vetor)

```
        else
                printf("Memoria nao alocada");}
```

E como vimos, podemos tentar alocar espaços de memória conforme requisição do usuário:

```
int main(){
        int *v;
        int tamanho;

        printf("Informe o tamanho do vetor desejado: ");
        scanf("%d", &tamanho);
        if(tamanho >1){
                v = (int*) malloc (tamanho*sizeof(int));

                if(v!= NULL){
                        printf("Vetor Alocado em v com tamanho = %d\n", tamanho);
                }
                else
                        printf("Memoria nao alocada");
        }
        else
                printf("Tamanho invalido");

}
```

Alocando dinamicamente um espaço de memória informado pelo usuário em tempo de execução

Mas e se alocarmos muito ou pouco espaço de memória, será que podemos “redimensionar” o tamanho de espaços alocados? *Sim, através da função realloc*.

Realloc

A função realloc, possui a responsabilidade de modificar o tamanho da memória previamente alocada para o ponteiro **p***, para mais ou para menos, especificando o novo tamanho na variável **num**. Conforme sua assinatura (ou protótipo):

```
void *realloc (void *p, unsigned int num)
```

Seu retorno também é genérico, igual a função malloc, e para especifica-los devemos *"dar um cast".* Mas atenção para as situações:

- Caso o retorno da função não seja nulo (NULL) e o tamanho da nova alocação seja maior, o conteúdo do antigo espaço será copiado no novo espaço de memória, e nenhuma informação será perdida.
- Caso o retorno da função seja nulo (NULL), um novo espaço de memória não será alocado, o antigo espaço de memória é preservado e o retorno da função será o endereço original.

Vamos a um exemplo:

```
int main(){
        int *v, *p;
        v = (int*) malloc (50*sizeof(int));

        if(x!= NULL){
                printf("Vetor Alocado em x\n");
                printf("Realocando para um espaço maior\n");
                p = (int*) realloc (v, 100*sizeof(int));
                if(p == NULL)
                        printf("Espaço não realocado, mantido
                else
                        printf("Espaço maior alocado, vetor de tamanho 100\n");
        }
        else
                printf("Memoria nao alocada");
}
```

Espaço alocado dinamicamente, com tamanho 50

Tentativa de realocar novo espaço de tamanho maior (100)

se p == NULL, v e suas as informações são preservadas e p será v

Caso p não seja nulo, as informações de v são copiadas para p

Super legal, né? Então, vamos praticar, exercitar e trocar várias ideias no nosso fórum.