# 13. ALOCAÇÃO DINÂMICA DE MEMÓRIA

- Sempre é necessário declarar o número de elementos do array. Mas e se ele for desconhecido? E se ele variar?
- Pode haver desperdício de memória definir um tamanho grande no qual muitos campos não serão utilizados.
- Durante a gerência do processo de sua tarefa pelo sistema operacional, a transferência do programa do disco para a memória e vice-versa (processo conhecido como swapping) irá demorar mais.
- Para permitir que o espaço para estruturas de dados de um programa possa ser alocado durante a execução do mesmo, a linguagem C define funções de biblioteca chamadas funções de alocação dinâmica de memória.

Estas funções são:

```
#include <stdlib.h>     /* necessária para uso das funções */

void *malloc (size_t nrbytes)
void *calloc (int numero_elementos, size_t tamanho)
void *realloc (void *area_alocada, size-t novo_tamanho)
void free (void *area_alocada)
```

- As funções malloc() e calloc() são usadas para alocar espaço para seus dados uma vez que você determinou quanto espaço você precisa. E se sua estimativa se revela muito grande ou muito pequena, você pode mudar o tamanho deste espaço alocado com a função realloc.
- Uma vez que o programa terminou de usar o espaço alocado, a função free deve ser usada para liberar o espaço previamente alocado.

## *13..1 malloc e calloc*

- malloc recebe como argumento o número de bytes a ser alocado. Em muitos computadores, size_t é equivalente a unsigned int., na maioria dos compiladores
- calloc recebe como primeiro argumento o número de elementos a ser alocado. O segundo argumento indica o tamanho em bytes de cada elemento.
- calloc garante que o espaço alocado é inicializado com zeros, enquanto que malloc não; malloc mantém o valor da mémoria reservada, o chamado lixo de buffer.
- Estas duas funções retornam um ponteiro para a nova área alocada do tipo "genérico" void *. Em muitos compiladores o default pode ser uma apontador para char.
- O programador armazena esse endereço num ponteiro de tipo apropriado. Um cast de acordo com o tipo desejado deve ser usado para modificar o tipo do valor retornado.
- Quando não há sucesso na alocação de memória, as funções malloc e calloc retornam um ponteiro nulo, representado pela constante NULL. Assim, toda vez que um programa usa estas funções, deve-se testar o valor retornado para verificar se houve sucesso na alocação.

**Ex1:** alocamos um char, 1 byte.

```
char *ptr;
ptr = malloc(1);      /* fica mais legível usar malloc (sizeof(char)) */
scanf( "%c", ptr);
```

**Ex2** : para sabermos quantos bytes ocupam nosso tipo podemos usar o operador sizeof.

```
typedef struct{
     int RA;  char [10] nome;
  } aluno;
aluno *a;
a = (aluno *) malloc(sizeof (aluno));
/* ou  a = (aluno *) calloc (1, sizeof (aluno);  */
a -> RA = 19;
gets (&a->nome);
```

Pode-se declarar e alocar ao mesmo tempo

```
aluno * a = (aluno *) malloc (sizeof (aluno));
struct ponto *p = (struct ponto *) malloc (sizeof (struct ponto));
```

**Ex3:** alocação  (e depois desaloção) de um vetor de inteiros durante a execução de um programa:

```
int *v;
int n, i;
scanf( "%d", &n);
v = (int) * malloc (n* sizeof (int));
/*  v = (int) * calloc (n, sizeof (int)); */
if (!v){
   printf ("\n\nErro: Memoria Insuficiente\n\n **");
   exit ();
}
for (i = 0; i < n; ++i)
   scanf ( "%d", &v[i]);
for (i = n; i > 0; --i)
   printf( "%d ", v[i-1]);
free(v);
```

**Ex4**: com matrizes

```
   int **A;   /* declarado um ponteiro para um ponteiro de int */
   int i, n,m;

 /* alocado um ponteiro para ponteiro de inteiros, m vezes (m linhas) */
  A = (int **) malloc (m * sizeof (int *));

 /* A = (int **) calloc (m , sizeof (int *));  */
  for (i = 0; i < m; ++i)
     /* alocacao para cada linha que é também um apontador de int */
     A[i] = (int *) malloc (n * sizeof (int));

     /* A = (int *) calloc (n, sizeof (int ));  */
```

OBS: Cada invocação de malloc e calloc aloca além do solicitado bytes adicionais usados para administrar o bloco de memória e usado pela função free. O número não depende do número de bytes solicitado, por isto, não é recomendável usar malloc repetidas vezes com argumento para tamanho muito pequeno. É preferível alocar um grande bloco de bytes e retirar pequenas porções desse bloco na medida do necessário.

## *13.2 free*

- Libera a posição de memória apontada por area_alocada.
- Se a area_alocada for uma apontador null nada será realizado.
- Costuma-se também atribuir o valor null ao ponteiro correspondente a área desalocada após o comando free.

### *13.3 realloc*

- A função realloc é usada para redimensionar um espaço alocado previamente com malloc ou calloc. Seus argumentos são um ponteiro para o INÍCIO de uma área previamente alocada, e o novo tamanho, que pode ser maior ou menor que o tamanho original.
- realloc retorna um ponteiro para a nova área alocada. Este ponteiro pode ser igual ao ponteiro original se o novo tamanho for menor que o tamanho original, e diferente do ponteiro original se o novo tamanho for maior que o tamanho original. Neste último caso, realloc copia os dados da área original para a nova área.
- Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

**Ex. 5:**

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

 int main(void)
 {  int *p, i,k, n;
    printf ("\nDigite a quantidade de numeros que serao digitados: ");
    scanf ("%d",&i);


    /* aloca i espaços para inteiro */
    p=(int*) malloc (i*sizeof(int));
    if (p==NULL)
    {
       printf ("\n Memoria insuficiente");
       exit(1);
    }
    for (k=0;k<i;k++)
    {
       printf ("\nDigite o valor %d ",k+1);
       scanf ("%d",&p[k]);
    }

    for  (k=0;k<i;k++)
        printf ("%d ",p[k]);
```

```
    printf ("\n\nSeu vetor possui %d elementos.",i);
    printf ("\nDigite um valor positivo para aumentar seu vetor.");
    printf ("\nDigite um valor negativo para diminuir seu vetor.");
    scanf ("%d",&n);
    if (!(i+n))
    {  printf ("\nSeu vetor possui 0 elementos.\n\n");
       free(p);
       exit(0);
    }
    else if ((i+n)<0)
    {
       printf ("\n Numero negativo de elementos”);
       free(p);
       exist (0);
    }

    /* a função realloc aumenta (numero positivo) ou diminui
     (numero negativo), o tamanho do vetor dinamicamente. */
    p=(int*) (realloc(p, (i+n)*sizeof(int)));
    if (p==NULL)
    {
       printf ("\n Memoria insuficiente");
       exit(1);
    }
    for (k=0;k<(n+i);k++)
    {
       printf ("\nDigite o %do valor do vetor: ",k+1);
       scanf ("%d",&p[k]);
    }
    for (k=0;k<(n+i);k++)
    printf ("%d\t",p[k]);
    free(p);
}
```

**Ex 6:**

```
#include <stdlib.h>
 …..
char buf[30],   *memchar;
int *memint, i;
      ....
memchar = malloc (sizeof(buf));   /* retorna apontador para char */
if (memchar == NULL) {
   printf ("Erro em alocacao de memoria\n");
   exit (1);
}
else
   memcpy (memchar, buf, sizeof(buf));  /* copia o conteúdo de
                                          * buf para memchar  */
free (memchar);
…..

memint = (int ) calloc (1, sizeof(int));
                          /* uso de cast, quero 1 apontador para int*/
 if (memint == NULL) {
        printf ("Erro em alocacao de inteiros\n");
        exit (1);
 }
  else {
      i = 0;
      do {
             scanf("%d", memint + i);
             if (memint[i]) {
               memint = (int *) realloc (memint, (i + 2) * sizeof(int));
                /* aumenta de 1 (um) o tamanho do array dinâmico
                   representado por memint */

               if (memint == NULL)
                  break;
             }
       } while (memint[i++]);
   }

…..
free (memint);
```

### *13.4 Exemplo com lista encadeada*

```
#include <stdio.h>

typedef struct  {
       int chave;
} tipoitem;
typedef struct tipono {
      tipoitem item;
      struct tipono * prox;
}tipono;
typedef struct tipono * apontador;
typedef struct {
     apontador inicio, fim;
} tipolista;

inicia(tipolista * lista)
{
 lista->inicio = lista->fim = NULL;
}

int vazia ( tipolista lista)
{
 return(lista.inicio == NULL);
}

imprime(tipolista lista)
{ apontador aux;
   for (aux=lista.inicio; aux != NULL; aux=aux->prox)
      printf("%d\n", aux->item.chave);
}
```

```
int insere (tipoitem x, tipolista * lista)
{ apontador novo;
  novo = (apontador) malloc (sizeof(tipono));
  if (novo != NULL)
  {  novo->item = x;
     novo->prox = NULL;
     if (lista->inicio == NULL)
       lista->inicio = novo;
     else
        lista->fim->prox = novo;
     lista->fim=novo;
     return (1);
   }
   else {
       printf("Memoria Insuficiente");
      return(0);
     }
}
int main()
{ tipolista lista;
  int i;
  tipoitem x;
  inicia(&lista);
  for (i=10; i >0 ; i--)
   {  x.chave = i;
     insere(x, &lista);
  }
  imprime (lista);
  retira (x, &lista);
}
```

```
int retira(tipoitem x, tipolista * lista)
{  apontador aux, ant;
  /* se lista está vazia nem entra no for */
   for (aux=lista->inicio, ant = aux; aux != NULL;  ant = aux;
           aux=aux->prox)
    {    if (aux->item.chave == x.chave)
           {
              if (aux == ant)     /* x é primeiro elemento */
              {
                lista->inicio = aux->prox;  /* mudar valor inicio */
                 if (lista->inicio == NULL)  /* se x era único na lista */
                    lista->fim = NULL;
                }
               else
               {
                 ant->prox = aux->prox;
                 if (aux == lista->fim)    /* x era o ultimo elemento */
                    lista->fim = ant;
                 }
               free(aux);
               return(1);
             }
       }
  return(0);
}
```