Memória dinâmica

Para trabalharmos com vetores ou outro tipo de dados, mais ou menos complexos, era absolutamente necessário que soubéssemos, *a priori*, qual o número de elementos de que iríamos necessitar.
Por exemplo, para criarmos duas strings iguais com um tamanho a introduzir pelo utilizador, declarávamos as duas strings com uma dimensão suficientemente grande, de modo a que a string introduzida pudesse ser copiada para outro vetor.
É possível criar memória apenas quando a necessitamos, libertando-a assim que esta não seja precisa.
Poupa-se memória ao evitar estar a reservar conjuntos elevados da mesma, que só seriam novamente libertados, quando o programa ou a função, onde foram declarados, terminasse.
Todas as funções que tratam da alocação de memória dinâmica encontram-se acessíveis através da biblioteca stdlib.h.
As funções malloc e free são as mais básicas para a gestão de memória. malloc é responsável pela alocação/reserva de um pedaço de memória, e free é responsável por libertar esse pedaço de memória.

Função malloc

Sintaxe:

```
void *malloc (size_t n_bytes)
```

onde:

- **size_t** está normalmente definido como sendo typedef unsigned int size_t (inteiro sem sinal);

O objetivo da função é criar um bloco constituído por n_bytes bytes e devolver o endereço desse bloco.
Permite alocar/reservar o conjunto de bytes indicados pelo programador, devolvendo um apontador para o bloco de bytes criado (mais especificamente para o 1.º byte do bloco), ou NULL, caso a alocação/reserva de memória falhe (se não houver memória suficiente).

- **void*** significa que a função devolve um apontador para qualquer tipo, isto é, devolve um endereço de memória.
- Para saber o tamanho do bloco a alocar, poderemos ter que usar o operador sizeof.

Função **free**

Sintaxe:

```
void  free (void *n_bytes)
```

- sendo n_bytes um apontador para o bloco de memória reservado

A função free permite libertar a memória existente no apontador. Este fica com o mesmo valor mas aponta para uma zona de memória que já não lhe pertence.

Exemplo 1:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
char s[200], *ptr;                  /*apontador para um conjunto de char */
printf(”String?”); gets(s);
/*reservar/alocar a memória necessária*/
ptr= (char*) malloc(strlen(s)+1);          /* +1 para o carater terminador */
if(ptr==NULL)
      puts(“Problemas de alocação de memória!”);
else
   {
      /*copia de s para outra string*/
      strcpy(ptr,s); // ou ptr=s;
      printf(“String original: %s\n Cópia: %s\n”, s, ptr);
/*libertar a memória existente em ptr */
      free(ptr);
      ptr=NULL;
    }
}
```

Exemplo 2:

```
#include <stdio.h>
 #include <stdlib.h>
int main()
{
  int *p, *q;
  p = (int *) malloc(sizeof(int));
  q = p;
  *p = 10;
  printf("%d\n", *q);
  printf("%d\n", *q);
 free(p); free(q);
}
```

- O compilador aceita p=q porque ambos são ponteiros e apontam para o mesmo tipo.
- Podemos simplificar p =(int*) malloc(sizeof(int)); por p =(int*) malloc(4); mas como temos sistemas operativos de 32, 64 bits a primeira declaração torna as coisas mais portáveis.

Exemplo 3:

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

  main()
{
  /* ponteiro para memória que será reservada*/
  int *p;
  int i;
  /* reservar 10 elementos inteiros, ou seja, ( sizeof (int) * 10 ) */
  p = (int *) malloc ( sizeof (int) * 10);

  if ( p == NULL ) {
     printf ("Erro: Não foi possível reservar memória\n");
     exit(1);
  }

for(i = 0; i < 10; i++) {
    p[i] = i * 2;
    printf ("%d\n", p[i]);
  }
  /* liberta a memória alocada por malloc */
  free (p);}
```

A função exit

A função exit da biblioteca stdlib interrompe a execução do programa e fecha todos os ficheiros que o programa tenha aberto. Se o argumento da função for 0, o sistema operativo é informado de que o programa terminou com sucesso; caso contrário, o sistema operativo é informado de que o programa terminou de maneira excecional.
O argumento da função é tipicamente a constante EXIT_FAILURE, que vale 1, ou a constante EXIT_SUCCESS, que vale 0.

Exercícios

1- Escreve a função char *Repete (char *string, int n) que cria dinamicamente uma nova string com n "cópias" da string original, separadas por espaço, exceto a última ocorrência.

2- Implementa a função char *metade (char *s) que cria dinamicamente uma nova string, contendo apenas metade da string. Nota: Usa a função **strcnpy**.

3- Implementa a função char *Inverte (char *s) que cria dinamicamente uma nova string, contendo a string invertida. Nota: Usa a função strrev.