Aula 24

MC 102 - Algoritmos e Programação de Computadores

Apontadores e Alocação Dinâmica II.

Alocação Dinâmica

Para alocar memória, já sabemos quais comandos utilizar e a diferença entre eles. Mas como saber se a quantidade de memória solicitada foi realmente disponibilizada para uso?

Devemos utilizar a constante **NULL**, definida em **`stdlib.h`**, que representa um apontador **nulo**, sem valor.

Isso vale para as funções **`malloc()`**, **`calloc()`** e **`realloc()`**.

Alocação Dinâmica

O seguinte programa aloca memória até que o sistema operacional não consiga mais reservar a quantidade de memória desejada.

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

int main(int argc, char **argv) {
   int cont=0, *ap;
   do { /* Cuidado, salve tudo antes de rodar isso */
    printf("Alocado até agora %10dMb\n", cont++);
    ap = (int *) malloc(1048576); /*1Mb de memória*/
   } while (ap != NULL);
   printf("Não foi possível alocar mais memória.");
   return 0;
}
```

>> Critique o programa acima <<

Alocação Dinâmica

Apesar do programa anterior não fazer algo interessante com a memória alocada, ele nos leva à seguinte questão:

O que acontece se mais memória for alocada e retornada para um mesmo apontador, que possui um endereço de memória de uma quantidade alocada anteriormente? Isto é, o que acontece se `malloc()` for utilizado duas ou mais vezes para a mesma variável sem liberar a memória anterior?

Alocação Dinâmica

É importante lembrar que toda memória alocada deverá ser liberada, mas também que um apontador deverá ser tratado com muito cuidado para não se perder a referência da memória.

Aritmética de Apontadores

Existe alguma diferencia nesses comandos?

```
int ap[10] = {1,2,3,4,5,6,7,8,9,0};
printf("%d\n", ap[5]);
printf("%d\n", *(ap + 5));
```

Aritmética de Apontadores

Uma forma de trabalhar com apontadores é realizar "deslocamentos" no endereço de referência do mesmo. Vejamos um exemplo para o uso de vetores:

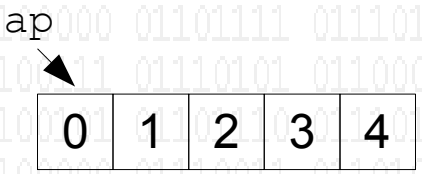
```
int i,*v, *ap;
v = (int *) malloc(5 * sizeof(int));
ap = v;
for (i=0;i<5;i++) {
   scanf("%d", ap);
   ap++;
}
```

Vamos entender como isso funciona!

Aritmética de Ponteiros

Os endereços de mémória também são numéricos.

Comando	**Equilavente a**	
`ap[0]`	`*ap ou *(ap+0)`	1o. elemento
`ap[1]`	`*(ap+1)`	2o. elemento
`ap[2]`	`*(ap+2)`	3o. elemento
`&ap[3]`	`(ap+3)`	End. do 4o. elemento



Os deslocamentos dos endereços na memória estão relacionados com o tamanho dos elementos alocados.

Aritmética de Apontadores

Utilizando apontadores para strings:

```
int strcpy(char *d, char *s, int max) {
  while ((max > 1) && (*s != '\0')) {
    *(d++) = *(s++);
    max--;
  }
  *d = '\0';
  return (*s == '\0');
}
```

O incremento dos apontadores elimina a necessidade de uma variável contador para percorrer os caracteres da string.

Aritmética de Apontadores

Para trabalharmos com matrizes, podemos pensar em indexar os elementos de uma matriz da seguinte forma:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Deste modo, a alocação da matriz pode ser realizada considerando a dispoção de seus elementos em um vetor sequencial, facilitando o acesso aos índices da forma:

`m[i][j]` é equivalente a **`v[i*colunas + j]`**

ou simplesmente **`*(v + i*colunas + j)`**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Aritmética de Apontadores

Lendo e imprimindo os elementos de uma matriz:

```
int i,j,*m;

m = (int *) malloc(25 * sizeof(int));
for (i=0; i<5; i++)
   for (j=0; j<5; j++)
      scanf("%d", m + i*5 + j);
/* Imprimindo a matriz */
for (i=0; i<5; i++) {
   for (j=0; j<5; j++)
      printf("%d", *(m + i*5 + j));
   printf("\n");
}
free(m);
```

Apontadores em Funções

Apontadores podem ser utilizados como resultado de uma função. Por exemplo, a seguinte função retorna a primeira palavra de uma string qualquer.

```
char *firstword(char *str) {
  int i,n=0;
  char *res;

  while ((*(str+n)!='\0')&&(*(str+n)!=' ')) {
    n++;
  }
  res = (char *) malloc((n+1) * sizeof(char));
  res[n] = '\0';
  for(i=0;i<n;i++)
    *(res + i) = *(str + i);
  return res;
}
```

Observações Importantes!

Considere a declaração abaixo:

```
int v[10], *r, *t;
```

Algumas operações são permitidas, outras não. Quais são permitidas e quais não são?

```
a) r = v;
b) t = v + 5;
c) r = r + 1;
d) t = t - 2;
e) *r = *(t +4);
f) *(v +1) = 5;
g) v = v + 1;
```

Observações Importantes!

Considere a declaração abaixo:

```
int v[10], *r, *t;
```

Algumas operações são permitidas, outras não. Quais são permitidas e quais não são?

```
a) r = v;          /* r = &v[0]      */
b) t = v + 5;      /* t = &v[5]      */
c) r = r + 1;      /* r = &v[1]      */
d) t = t - 2;      /* t = &v[3]      */
e) *r = *(t +4);   /* v[1]=v[7]      */
f) *(v +1) = 5;    /* v[1]=5         */
g) v = v + 1;      /* Não permitido */
```

Exercício 1

Faça uma função que receba dois vetores de inteiros e seus tamanhos como parâmetro e retorne um vetor contendo a união dos elementos desses vetores.

Dado A e B, retornar A **U** *B*
usando aritmética de ponteiros.
Os vetores estão ordenados.

Exercício 1 - Programa

União dos elementos de 2 vetores ordenados

```
int *uniao(int *vA, int *vB, int m, int n, int *nelem) {
  int *vAB, *p;
  vAB = (int *) malloc((m + n) * sizeof(int));
  p = vAB;
  while (m + n > 0) {
    if (n == 0) { /* Nao tem mais elementos em B */
      while (m >0) {
        *(p++) = *(vA++); m--;
      }
    } else if (m == 0) { /* Nao tem mais elementos em A */
      while (n >0) {
        *(p++) = *(vB++); n--;
      }
    } else {
      if (*vA < *vB) {
        *(p++) = *(vA++); m--;
      } else if (*vB > *vA) {
        *(p++) = *(vB++); n--;
      } else {
        *(p++) = *(vA++); vB++; m--; n--;
      }
    }
  }
  *nelem = (p - vAB); /* Numero de elementos no vetor vAB */
  vAB = (int *) realloc(vAB, (*nelem) * sizeof(int));
  return vAB;
}
```

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). Use aritimética de ponteiros.