Fundamentos de Programação

Aula 11 - Alocação Dinâmica

Arnaldo Barreto Vila Nova

Sumário

- Introdução
- malloc
- free
- calloc
- realloc

Introdução

- **Problema da Alocação Estática número 1**
  - Quaisquer variáveis declaradas durante um código vão continuar a ocupar o espaço de memória durante todo seu escopo.
  - Por exemplo, um vetor de 500.000 inteiros declarado no início do main irá ocupar 2.000.000 de bytes durante todo o programa, independente se todos os espaços serão utilizados ou não.
- A Alocação Dinâmica permite alocar e liberar memória de acordo com o necessário e quando for necessário.

Introdução

- **Problema da Alocação Estática número 2**
  - A alocação estática é sempre sequencial. Na utilização de uma grande quantidade de dados é possível que não haja disponível toda a quantidade necessária sequencialmente.
  - Por exemplo, uma matriz de 50mil x 50mil inteiros precisará de 10 bilhões de bytes livres (pouco mais de 9 gigabytes) seguidos, sequenciados, na memória. Mesmo para um computador com 16GB de RAM isso pode não ser possível.
- A Alocação Dinâmica permite criar estruturas de dados especiais que ocupem espaços fracionados na memória

Introdução

- **Problema da Alocação Estática número 2**
  - A alocação estática é sempre sequencial. Na utilização de uma grande quantidade de dados é possível que não haja disponível toda a quantidade necessária sequencialmente.
  - Por exemplo, uma matriz de 50mil x 50mil inteiros precisará de 10 bilhões de bytes livres (pouco mais de 9 gigabytes) seguidos, sequenciados, na memória. Mesmo para um computador com 16GB de RAM isso pode não ser possível.

A disciplina de Estrutura de Dados trata apenas disso

- A Alocação Dinâmica permite criar estruturas de dados especiais que ocupem espaços fracionados na memória

Introdução

- A Alocação Dinâmica da Memória visa ter um controle de um ou mais espaços da memória, indicando quando vão ser usados ou liberados
- Reservamos um espaço de memória através da função **malloc** vinculando este espaço a um ponteiro.
- A função **free** libera o bloco de memória quando ele não for mais utilizado
- Ambos da biblioteca <stdlib.h>

malloc (unsigned int tam)

- O parâmetro é o tamanho em bytes do bloco da memória a ser reservado

```
int main()
{
     char *ptr;
     ptr = (char*) malloc(1);
     scanf("%c", ptr);
     printf("%c", *ptr);
     return 0;
}
```

Se você sabe o número exato de bytes a ser usado, pode colocar diretamente o valor

malloc (unsigned int tam)

- O parâmetro é o tamanho em bytes do bloco da memória a ser reservado

```
int main()
{
     int *p;
     p = (int*) malloc(sizeof(int));
     scanf("%d", p);
     printf("%d", *p);
     return 0;
}
```

Se não souber a quantidade de bytes pode utilizar a função **sizeof** para descobrir

malloc (unsigned int tam)

- Caso tenha algum erro de alocação (memória cheia) o **malloc** retorna NULL

```
int main()
{
     long int *p = malloc(5000*sizeof(long int));
     if (p == NULL)
     {
          printf("Memoria cheia");
          return 1;
     }
     ...
}
```

É sempre bom verificar se a alocação deu certo para lidar com o problema de alguma forma

free(void* ptr)

- A função free recebe um ponteiro de um bloco de memória alocado previamente com **malloc** e libera (desaloca) este espaço de memória.
- Próxima chamada de **malloc** ou declaração de variável pode usar esse espaço liberado
- Ele não apaga o valor guardado naquele espaço (se tinha um 5 ou um ‘a’ naquele espaço, ele continuará lá)

free(void* ptr)

- Não deixe ponteiros “soltos” apontando para um espaço de memória já liberado
  - Isso pode ser usado como vulnerabilidade do sistema

```
void main()
{
     int *p;
     p = (int*) malloc(sizeof(int));
     scanf(“%d”, p);
     printf(“%d”, *p);
     free(p);
     p = NULL;
     ...
}
```

É uma boa prática de programação colocar o ponteiro para **NULL** logo depois de liberá-lo se não for alocar outro espaço nele logo

Lixo de memória

- Quando alocamos um espaço de memória, é comum ter valores previamente utilizados naquele espaço que não foram limpos ou zerados, o que chamamos de lixo de memória
- É uma boa prática de programação fazer essa limpeza antes de utilizar o **free** para liberá-lo
- Imagine por exemplo o problema que pode ser um espaço de memória guardando uma senha do sistema for liberada sem apagar essa informação antes

calloc(unsigned int num, unsigned int tam)

- Normalmente usado para usar arrays de tamanho dinâmico, o **calloc** reserva um número **num** de elementos, cada um com tamanho **tam**.
- O valor 0 (zero) é atribuído a cada elemento na alocação, evitando o uso de lixo de memória

calloc(unsigned int num, unsigned int tam)

```
int i, n = 10, *vetor;
vetor = calloc(n, sizeof(int));
for (i=0; i<n; i+=2)
{
     scanf("%d", &vetor[i]);
}
printf("\nEm ordem inversa: ");
for (i=n-1; i>=0; i--)
{
     printf("%d\t", vetor[i]);
}
free(vetor);
vetor = NULL;
```

Estou preenchendo somente as posições pares do vetor

Note que as posições ímpares vão estar preenchidas com o valor **0**. Isso pode não acontecer se você utilizar o **malloc** ao invés do **calloc**

realloc(void* ptr, unsigned int tam)

- Modifica o tamanho de uma alocação anterior
- O conteúdo na memória permanece inalterado
  - Mas você perderá vínculo com alguns valores caso o novo tamanho seja menor que o tamanho do conteúdo
- Se o endereço de memória onde o ponteiro se encontra não tiver espaço sequencial suficiente para o novo tamanho, ele irá procurar um outro endereço com espaço suficiente e transferir os dados.
- Se **ptr** for NULL, o **realloc** funciona como um **malloc**

realloc(void* ptr, unsigned int tam)

```
int i, *v;
v = calloc(5, sizeof(int));

for (i = 0; i<5; i++)
{
    v[i] = i;
}

v = realloc(v, 10 * sizeof(int));

for (i=5; i<10; i++)
{
    v[i] = i;
}
```

Aqui estamos alocando um vetor de inteiros de tamanho 5

Aqui o vetor está sendo aumentado de tamanho para 10 inteiros

realloc(void* ptr, unsigned int tam)

```
int i, *v;
v = calloc(10, sizeof(int));

for (i = 0; i<10; i++)
{
    v[i] = i;
}

v = realloc(v, 5 * sizeof(int));
```

Ao reduzir o tamanho, você perde o contato com as últimas posições. O endereço de memória dessas posições já estão liberadas para outras alocações como se você tivesse usado o **free**.

realloc(void* ptr, unsigned int tam)

```
int i, *v;
v = calloc(10, sizeof(int));

for (i = 0; i<10; i++)
{
    v[i] = i;
}

v = realloc(v, 5000 * sizeof(int));
```

É possível que a realocação de tamanho não seja possível por falta de memória. Caso isso aconteça o **realloc** irá ter resultado **NULL** e podemos perder vínculo com os dados que já estavam no vetor.

realloc(void* ptr, unsigned int tam)

```
int * v2;
v2 = realloc(v, 5000 * sizeof(int));
if (v2 == NULL)
{
    //Deu ruim. Não tem memória disponível pra aumentar, mas os dados continuam em v
}
else
{
    //Deu bom. Vamos liberar o v e usar o endereço em v2.
    free(v);
    v = v2;
    v2 = NULL;
}
```

Para este tipo de situação e evitar a perda de informações, é bom ter um ponteiro extra de segurança para usar o **realloc**

Exercícios

- Usando Alocação Dinâmica armazene 5 números inteiros aleatórios entre 1 e 6.

- Com os números sorteados na questão anterior, imprima quantos valores foram repetidos. (ex.: se foram sorteados 2, 3, 2, 5 e 3, teve dois números que foram repetidos).

- Adicione aos 5 números anteriores mais 5 números aleatórios e imprima novamente quantos números foram repetidos após a adição.

Exercícios

- Usando Alocação Dinâmica, receba valores inteiros do usuário e os armazene em um vetor. O programa deverá ir modificando o tamanho do vetor a cada número lido e deverá para quando o usuário digitar o valor 0 ou não ter mais memória disponível para aumentar o tamanho do vetor.