Programação Estruturada

Aula 21 - Alocação Dinâmica

Yuri Malheiros (yuri@ci.ufpb.br)

Introdução

- Estruturas de dados em C normalmente tem um tamanho fixo
- Isso é um problema, pois precisamos escolher o seu tamanho previamente e não podemos mudá-lo durante a execução do programa
- Imagine um programa que controla o estoque de uma loja
    - Temos que fixar o tamanho do estoque, mas em algum momento esse tamanho pode não ser suficiente

Introdução

- Para resolver esse problema, vamos usar a alocação dinâmica de memória
- Com ela, podemos alocar memória durante a execução do programa
- Então, podemos ter estruturas que crescem ou diminuem de acordo com o necessário

Funções para alocação de memória

- Existem três funções para alocação de memória
- Elas são declaradas em `stdlib.h`
- `malloc` - aloca um bloco de memória
- `calloc` - aloca um bloco de memória e o limpa
- `realloc` - redimensiona um bloco de memória previamente alocado

Funções para alocação de memória

- `malloc` é a função mais usada, ela é mais eficiente que o `calloc`
- Quando alocamos um bloco de memória, a função não sabe que tipo de dado vamos guardar nesse bloco
- Por isso, não é possível retornar um ponteiro para um tipo determinado
- Assim, a função retorna um ponteiro para void `void *`
- Interpretamos isso como um ponteiro genérico para a memória

Ponteiros nulos

- Quando usamos uma função para alocar memória, pode não ser possível alocar um bloco de memória muito grande
- Quando isso acontece, um ponteiro nulo é retornado
- Sempre que pegarmos o retorno de uma função de alocação de memória, devemos testar se esse valor não é um ponteiro nulo
- O ponteiro nulo é representado pela macro `NULL`

Ponteiros nulos

```
p = malloc(1000000);

if (p == NULL) { /* a alocação falhou */ }
```

- No teste condicional, todo ponteiro não-nulo é verdadeiro e um ponteiro nulo é falso, então poderíamos fazer o teste:

```
p = malloc(1000000);

if (!p) { /* a alocação falhou */ }
```

Alocando strings dinamicamente

- A alocação dinâmica é muito útil para trabalhar com strings
- Quando declaramos um string, precisamos definir o seu tamanho máximo
- Mas pode ser difícil antecipar qual valor é adequado

Alocando strings dinamicamente

- `void *malloc(size_t size);`
- A função `malloc` aloca um bloco de `size` bytes e retorna um ponteiro para ele
- `size_t` representa um tipo unsigned int

Alocando strings dinamicamente

- Como um `char` possui 1 byte, então para alocar um string de `n` caracteres, nós usamos:
  - `p = malloc(n+1);`
- Onde `p` é do tipo `char *`
- Somamos `1` a `n` para termos o espaço do caractere nulo no fim do string
- O ponteiro genérico retornado pelo `malloc` é convertido automaticamente para `char *` na atribuição

Alocando strings dinamicamente

- A memória alocada usando `malloc` não é inicializada, então `p` vai apontar para um array não inicializado de `n+1` caracteres
- Podemos usar `strcpy` para inicializar o array

Alocando strings dinamicamente

- Com a alocação dinâmica de memória, é possível escrever função que retornam um ponteiro para um novo string que não existia antes da chamada da função
- Vamos escrever uma função que concatena dois strings sem modificar os strings passados para função
- Nessa função, o string concatenado será retornado

Alocando strings dinamicamente

```c
char *concat(const char *s1, const char *s2) {
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Erro: falha no malloc\n");
    exit(1);
  }

  strcpy(result, s1);
  strcat(result, s2);

  return result;
}
```

Alocando arrays dinamicamente

- Alocar arrays dinamicamente tem as mesmas vantagens de alocar strings dinamicamente
- Prever o tamanho de um array durante a escrita de um programa é difícil
- É mais conveniente esperar até que programa sendo executado decida qual tamanho um array deve ter

Alocando arrays dinamicamente

- Podemos usar o `malloc` da mesma maneira que usamos para os strings
- A principal diferença é que os elementos de um array não necessariamente têm 1 byte de tamanho
- Podemos usar `sizeof` para calcular o espaço necessário para cada elemento

Alocando arrays dinamicamente

- Suponha que vamos alocar um array de `n` inteiros

```
int *a;

a = malloc(n * sizeof(int));
```

- Sempre use o `sizeof` ao invés do número como tamanho em bytes do inteiro, pois esse tamanho pode variar em computadores diferentes

Alocando arrays dinamicamente

- Depois de alocado, podemos ignorar o fato de que `a` é um ponteiro
- Podemos usá-lo como o nome de um array

```
for (i=0; i<n; i++)
  a[i] = 0;
```

Função calloc

- `void *calloc(size_t nmemb, size_t size);`
- `calloc` aloca espaço para um array com `nmemb` elementos, sendo cada elemento com tamanho `size`
- Depois de alocar o espaço, `calloc` inicializa todos os bits como 0

Função realloc

- Depois que um bloco foi alocado, pode ser necessário mudar o seu tamanho
- A função `realloc` permite redimensionar o bloco alocado
- `void *realloc(void *ptr, size_t size);`
- `ptr` deve apontar para um bloco de memória alocado com `malloc`, `calloc` ou `realloc`
- O parâmetro `size` é o novo tamanho do bloco

Desalocando memória

- Um programa pode alocar vários blocos de memória e depois perder o controle sobre o que foi alocado e o que está sendo usado ou não
- Assim, o programa pode desperdiçar espaço de memória

```
p = malloc(...);
q = malloc(...);
p = q;
```

- O primeiro bloco de memória não possui ponteiros e você não vai conseguir usá-lo novamente
- Blocos de memória não mais acessíveis são chamados de garbage
- Algumas linguagem possuem garbage collector, mas C não, você precisa limpar o seu próprio lixo

Função free

- A declaração da função `free` se encontra em `stdlib.h`
- `void free(void *ptr);`
- Basta passar o ponteiro para o bloco de memória que não precisamos mais que ele vai ser liberado e estará disponível para novas alocações

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Função free

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

- Cuidado, nesse exemplo, `p` não aponta mais para um bloco válido na memória
- Modificar `p` pode causar erros inesperados