

C

Guilherme P. Telles

IC

30 de março de 2023

Alocação dinâmica de memória

Alocação dinâmica de memória

- Mecanismo que permite usar posições de memória além das variáveis declaradas no programa.
- Usando alocação dinâmica de memória temos mais flexibilidade para escrever programas sem conhecer antecipadamente a quantidade de dados que serão armazenados.
- A combinação de apontadores e alocação dinâmica de memória permite organizar dados de formas não-lineares (como jagged arrays, árvores etc). Essas organizações podem ser muito eficientes.

Alocação dinâmica de memória em C

- A alocação de memória é feita através de funções da biblioteca-padrão.
- As funções são genéricas e retornam `void*`, que deve ser convertido implícita ou explicitamente.
- É responsabilidade do programador liberar a memória alocada dinamicamente quando ela não for mais necessária.

stdlib.h

- Declara funções para reservar espaço na memória (alocar), realocar e liberar regiões de memória em tempo de execução.

- `void* malloc(size_t n)`

Aloca `n` bytes consecutivamente na memória.

Retorna um apontador `void` para o início da região alocada ou `NULL` se não for possível alocá-la.

- `void* calloc(size_t n, size_t t)`

Aloca `n` elementos de `t` bytes consecutivamente na memória e inicializa todos os bits da região com zero.

Retorna um apontador `void` para o início da região alocada ou `NULL` se não for possível alocá-la.

- `void free(void *ptr)`

Recebe um apontador para o início de um região alocada anteriormente por `calloc` ou `malloc` e libera essa região.

Chamadas subsequentes de `{c,m}alloc` poderão usar esse mesmo espaço.

Liberar uma região que já foi liberada pode produzir erros interessantes.

- Alocação de memória para um inteiro:

```
int* p;  
  
p = malloc(sizeof(int));  
  
*p = 13;  
  
free(p);
```

- Alocação de memória para dois inteiro, “perdendo” o primeiro na memória:

```
int* p = malloc(sizeof(int));  
  
p = malloc(sizeof(int));  
  
*p = 51;  
  
free(p);
```

- Alocação de memória para n inteiros:

```
int* p;  
  
p = (int*) malloc(n * sizeof(int));  
free(p);  
  
p = (int*) calloc(n, sizeof(int));  
free(p);
```

- Alocação de n apontadores e depois n double, n vezes:

```
double **A = malloc(n*sizeof(double*));  
  
for (int i=0; i<n; i++)  
    A[i] = malloc(n*sizeof(double));  
  
...  
  
for (int i=0; i<n; i++)  
    free(A[i]);  
  
free(A);
```

- Alocação de n apontadores e depois n double, n vezes, testando se malloc falhou:

```
int fail = 0;

double **A = malloc(n * sizeof(double*));
if (!A) // se falhou, termina o programa.
    exit(errno);

for (int i=0; i<n; i++) {
    A[i] = malloc(n * sizeof(double));
    if (!A[i]) // se falhou, termina o programa.
        exit(errno);
}

...

for (int i=0; i<n; i++)
    free(A[i]);
free(A);
```

- Alocação de um vetor de n strings com tamanho até 50, inicialmente vazias:

```
char** data = malloc(n*sizeof(char*));  
  
for (int i=0; i<n; i++) {  
    data[i] = malloc(51*sizeof(char));  
    data[i][0] = '\\0';  
}
```

- Alocação de um vetor de n strings com tamanho até 50, inicialmente vazias, testando se malloc falhou:

```
int fail = 0;

char** data = malloc(n * sizeof(char*));
if (!data) // se falhou, seta um flag.
    fail = 1
else
    for (int i=0; i<n; i++) {
        data[i] = malloc(51 * sizeof(char));
        if (!data[i]) { // se falhou, libera e seta um flag.
            while (i)
                free(data[--i]);
            free(data);
            fail = 1;
        }
        data[i][0] = '\\0';
    }
```

- `void* realloc(void *ptr, size_t n)`

Realoca uma região de memória previamente alocada para que fique com `n` bytes.

Retorna um apontador `void` para o início da região com `n` bytes consecutivos alocada na memória. Ou retorna `NULL` se não for possível alocar, e nesse caso a região original permanece inalterada.

A nova região pode estar em uma posição diferente na memória.

Preserva o conteúdo da região até o mínimo entre os tamanhos da original e da nova regiões. Faz cópia dos dados se for necessário.

Exemplo

- Realocação de uma região, dobrando o tamanho:

```
double* A = calloc(n, sizeof(double));  
  
...  
  
double* RA = realloc(A, 2*n);  
  
if (RA != NULL) {  
    // sucesso, |RA|=2|A| e se RA!=A entao A foi liberada.  
    // Podemos fazer A = RA:  
    A = RA;  
}  
else {  
    // falhou, ainda temos A.  
}
```

Arrays de tamanho definido por variável

- Em C um array de tamanho variável não é considerado alocação dinâmica.
- Embora o tamanho seja definido em tempo de execução as variáveis que definem o tamanho têm que estar em um posições de memória fixas.
- A maioria dos compiladores por aí armazena VLAs na stack. Alocação dinâmica é sempre na heap.

```
int n = ...;  
int m = ...;  
...  
int A[n][m];
```

stack e heap

- Stack e heap são porções da memória usadas durante a execução de um programa.

stack

- É alocada usando uma política de pilha (last-in-first-out).
- Quando uma função é chamada, os parâmetros, variáveis locais e dados de controle são armazenados no topo da região stack.
- Quando uma função retorna, a memória com os dados associados à função é liberada e a posição do topo da pilha é atualizada.
- Mais rápida porque o mecanismo de gerenciamento é trivial.
- Há uma localidade maior nos acessos à stack e então essa memória tende a ser mapeada em cache com mais frequência.
- Uma por thread. Quando a thread termina, a stack é liberada.
- O tamanho é definido na criação.

heap

- Não há um padrão de alocação ou liberação de memória na região heap.
- A heap pode ficar fragmentada em blocos livres e ocupados.
- O SO mantém um mecanismo para gerenciar a região heap e otimizar seu uso. Esse mecanismo consome tempo.
- Em geral uma por processo. Quando o processo termina, a heap é liberada.
- A memória usada pela heap aumenta à medida que é alocada, até o limite imposto pelo SO.
- Alocação dinâmica de memória é alocação na heap de um processo.

Encadeamento

- Encadeamento é o uso de apontadores para ligar regiões de memória.
- Regiões de memória encadeadas podem assumir várias configurações.

Exemplo

```
struct aluno {  
    int nusp;  
    struct aluno *prox;  
};  
  
typedef struct aluno aluno;  
  
aluno *p = (aluno*) malloc(sizeof(aluno));  
if (p) {  
    p->nusp = 542317;  
    p->prox = NULL;  
}  
  
aluno *q = (aluno*) malloc(sizeof(aluno));  
if (q) {  
    q->nusp = 654123;  
    q->prox = p;  
}
```



