

# TP1: Implementação das funções de alocação de memória sobre um vetor estático

## 1. Introdução

As funções malloc, realloc e free são utilizadas para gerenciamento de memória, um recurso crucial para qualquer programa, e cuja manipulação na linguagem C deve ser feita diretamente. É importante entender o funcionamento dessas funções para compreender suas limitações e os processos internos de gerenciamento de memória.

Nesse trabalho, o objetivo é implementar as funções de gerenciamento de memória sobre um vetor estático de memória. É necessário garantir que elas funcionem de acordo com suas funções homólogas de alocação dinâmica em C, inclusive em casos especiais, conforme pode ser visto em [1], [2] e [3]. Também devem ser criadas funções para inicialização e finalização do gerenciamento de memória, e para impressão dos dados da memória. É dado um programa principal main.c, que usa as funções criadas no TP para utilizar a memória estática. Essas funções e estruturas são criadas num arquivo valloc.c, e deve ser criado um arquivo header valloc.h para que seja possível a utilização dessas funções no programa principal. O programa main.c recebe um arquivo de entrada com instruções de alocação da memória, e interpreta-o, chamando as funções da biblioteca de acordo.

Pretende-se com esse trabalho adquirir um conhecimento mais apurado dos processos internos da linguagem C para gerenciamento de memória.

## 2. Implementação

### Estrutura de Dados

A memória é representada simplesmente como um vetor estático de tamanho de 1024576 bytes (um vetor de unsigned char). O tamanho máximo 1024576 é dado pela constante MAX\_MEM. Os dados dos blocos são representados pelo TAD Bloco, que representa um bloco livre ou ocupado na memória. Ele é representado da seguinte forma:

```
typedef struct bloco {  
    unsigned char *endereco;  
    int posicao;  
    size_t tamanho;  
    int ocupado;  
    struct bloco *proximo;  
} Bloco;
```

Assim, armazenamos os dados de posição do início do bloco e tamanho do bloco, caso ele esteja ocupado temos ocupado igual a 1, do contrário ocupado é igual a 0. Armazenamos o endereço do bloco, e também armazenamos um ponteiro para o bloco de memória seguinte. Caso esse bloco não tenha nenhum bloco seguinte, proximo é um ponteiro NULL.

## Funções e procedimentos

O TAD de Bloco possui as seguintes funções:

**Bloco \*inicializaLista(size\_t tamanhoMemoria):** Essa função aloca o primeiro bloco da lista usando a função malloc, dando a esse bloco a posição 0, e o tamanho igual a tamanhoMemoria passado. Determina o bloco como vazio, e sem próximo elemento. Por fim, retorna esse objeto.

**Bloco \*recebeBloco(Bloco \*inicio, unsigned char \*endereço):** Essa função percorre a lista procurando o bloco cujo endereço inicial é igual ao endereço passado, retornando esse bloco assim que ele é encontrado. Caso o bloco não seja encontrado, retorna um ponteiro NULL.

**Bloco \*insereBloco(Bloco \*inicio, Bloco novo):** Essa função percorre a lista procurando um bloco que esteja vazio e seja grande o suficiente para que seja possível alocar o bloco recebido. Caso não seja encontrado, a função retorna NULL. Do contrário, o bloco é alocado, e se for necessário, é dividido em dois, alocando o bloco novo na primeira parte.

**void insereDepois(Bloco \*local, Bloco \*novo):** Função que insere o bloco novo logo após o bloco local.

**int removeBloco(Bloco \*inicio, unsigned char \*endereço):** Função que desaloca o bloco que possui o endereço recebido, juntando com os blocos adjacentes que também estiverem vazios. Retorna -1 caso o endereço não esteja na lista de blocos, ou 0 caso a operação seja bem sucedida.

O módulo valloc.c (a biblioteca de alocação de memória), possui as seguintes funções:

**void inicializa\_gerencia(void):** Cria o primeiro item da lista de blocos, usando a função inicializaLista, e passa para o primeiro bloco o endereço inicial da memória.

**void finaliza\_gerencia(void):** Libera os blocos de memória existentes. Percorre a lista usando a função removeBloco em todos os blocos, até sobrar apenas um bloco. Por fim, libera esse bloco.

**void imprime\_status\_memoria(void):** Percorre a lista de blocos, imprimindo na tela as informações de cada bloco (tamanho, posição e status).

**void \*valloc(size\_t size):** Recebe um tamanho de bloco e cria-o utilizando a função insereBloco. Caso seja possível fazer a inserção, define o endereço do início do bloco e retorna esse endereço. Caso não seja possível, retorna NULL.

**void vfree(void \*p):** Caso receba um ponteiro NULL, não executa nenhuma operação. Do contrário, usa a função removeBloco para liberar a memória ocupada pelo bloco de endereço p.

**void \*vcalloc(size\_t nitems, size\_t size):** Recebe o número de blocos que deve alocar e o tamanho de cada bloco. Essa função chama valloc utilizando o produto de nitems e size como parâmetro. Caso valloc retorne NULL, retorna NULL. Do contrário, preenche a memória recém-alocada com zeros e retorna um ponteiro para o início da memória alocada.

**void \*vrealloc(void \*p, size\_t size):** Recebe um ponteiro para o início do bloco que deseja realocar e o novo tamanho desse bloco. Caso o ponteiro seja NULL, apenas aloca um novo bloco de memória usando valloc e retorna o endereço do início desse bloco na memória. Caso o tamanho seja 0, libera a memória ocupada por esse bloco usando vfree, e retorna NULL. Caso contrário, se o tamanho novo for menor que o tamanho atual, reduz o tamanho do bloco atual, dividindo o bloco atual em dois, e unindo com o bloco seguinte caso ele esteja vazio, e retorna p. Caso o tamanho novo seja igual ao anterior, não faz nada e retorna p. Por fim, se o tamanho novo é maior que o anterior, copiamos os dados existentes na região de memória reservada pelo bloco atual em um vetor de dados. Em seguida, a função descobre se o bloco seguinte tem espaço o suficiente para a realocação. Se tiver, estende o bloco atual e retorna p. Do contrário, descobrimos se a região posterior e anterior ao bloco juntas com o bloco atual tem espaço o suficiente. Se tem, podemos ter certeza que há espaço para realocar, e portanto podemos fazer a realocação sem preocupações. Assim, usamos a função vfree para liberar a memória atual e usamos valloc para alocar o bloco com o tamanho novo, e copiamos os dados do bloco anterior na memória do bloco novo, retornando o endereço do início da memória do bloco novo. Caso o espaço adjacente não seja o suficiente, usamos valloc para alocar outra porção de memória para o bloco. Caso valloc retorne NULL, a função retorna NULL. Por fim, se o valloc foi bem sucedido, usamos vfree no bloco de memória anterior, e copiamos os dados do bloco anterior na memória do bloco novo, e por fim, retornamos o endereço do início do novo bloco alocado.

### **Organização do Código, Decisões de Implementação e Detalhes Técnicos**

O TP possui 2 módulos divididos em 4 módulos principais: bloco.c e bloco.h que implementam o TAD de bloco usado nas funções de alocação, e valloc.c e valloc.h que implementam a biblioteca para alocação de memória em vetor estático.

Foi utilizado o tipo int para determinar se um bloco de memória está ocupado ou não, sendo 0 igual a bloco vazio e 1 igual a bloco cheio, pois como a linguagem C entende o valor 0 como falso, podemos usar esse valor em ifs e ele funcionará corretamente.

## **3. Análise de complexidade**

Funções do TAD:

A análise será feita em função de n, sendo n o número de blocos na lista.

**Função inicializaLista:** Apenas aloca um bloco e configura-o, **O(1)**.

**Função recebeBloco:** Entra em um loop que percorre a lista procurando um elemento, retornando-o assim que ele é encontrado. No pior caso ela passa por todos os elementos, portanto **O(n)**.

**Função insereBloco:** Entra em loop que percorre a lista procurando um bloco em que possamos fazer a alocação, e caso encontre-o, faz algumas operações **O(1)** para garantir que o bloco seja alocado corretamente. No pior caso passa por todos os elementos, portanto **O(n)**.

**Função insereDepois:** Faz uma inserção direta após o elemento recebido, portanto **O(1)**.

**Função removeBloco:** Entra em loop que percorre a lista procurando pelo bloco que deve ser removido e pelo bloco anterior. Após isso, caso encontre-os, executa algumas operações **O(1)** para garantir que o bloco seja removido e os blocos adjacentes ajustados de acordo. No pior caso passa por todos os elementos, portanto **O(n)**.

Funções do gerenciamento de memória:

A análise será feita em função de  $n$ , sendo  $n$  o número de blocos na lista.

**Função inicializa\_gerencia:** Apenas chama inicializaLista (que é  $O(1)$ ) e define o endereço do bloco inicial, portanto  **$O(1)$** .

**Função finaliza\_gerencia:** Percorre  $n-1$  elementos da lista, chamando removeBloco (que é  $O(n)$ ) em cada um para desalocá-los, e em seguida chama free no bloco inicial. Portanto,  $(n-1)*O(n) + O(1) = n*O(n) = O(n^2)$ .

**Função imprime\_status\_memoria:** Percorre todos os elementos da lista e mostra seus dados, portanto  **$O(n)$** .

**Função valloc:** Executa algumas operações  $O(1)$  e chama insereBloco, que é  $O(n)$ , portanto  **$O(n)$** .

**Função vfree:** Caso o vetor não seja nulo, chama a função removeBloco (que é  $O(n)$ ), portanto,  **$O(n)$** .

**Função vcalloc:** Executa operações  $O(1)$  para calcular o tamanho do bloco e para preencher a memória com zeros, e chama a função valloc (que é  $O(n)$ ), portanto  **$O(n)$** .

**Função vrealloc:** Se o ponteiro for nulo, chama valloc (que é  $O(n)$ ), se o tamanho for 0, chama vfree (que também é  $O(n)$ ). Do contrário, chama recebeBloco (que é  $O(n)$ ), e se o tamanho novo for menor que o atual, chama insereDepois (que é  $O(1)$ ). Se o tamanho novo for igual ao antigo, faz uma operação  $O(1)$ . Do contrário, se o espaço à sua frente for o suficiente, executa operações  $O(1)$ . Se o espaço à sua frente não for suficiente, executará um vfree, um valloc, e um recebeBloco ( $O(n) + O(n) + O(n) = O(n)$ ). Portanto, é  **$O(n)$** .

## 4. Testes

Foram realizados os testes recomendados na especificação e nos fóruns da disciplina. Os testes foram realizados em um Intel Core i5, com 4GB de memória em um Ubuntu 14.04 LTS (Trusty Tahr).

### 1.tst.i:

```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
lucas@Lucas-Ubuntu:~/Desktop/TP1$ ./tp1 1.tst.i
VAR 2
Status agora:
Pos: 0, Size: 1024576, Status: FREE

VALLOC 10 V0 OK
Status agora:
Pos: 0, Size: 10, Status: USED
Pos: 10, Size: 1024566, Status: FREE

VCALLOC 10 2 V1 OK
Status agora:
Pos: 0, Size: 10, Status: USED
Pos: 10, Size: 20, Status: USED
Pos: 30, Size: 1024546, Status: FREE

VREALLOC 20 V0 OK
Status agora:
Pos: 0, Size: 10, Status: FREE
Pos: 10, Size: 20, Status: USED
Pos: 30, Size: 20, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VFREE 0

VFREE 1
lucas@Lucas-Ubuntu:~/Desktop/TP1$
```

11.tst.i:

```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
lucas@Lucas-Ubuntu:~/Desktop/TP1$ ./tp1 11.tst.i
VAR 2
Status agora:
Pos: 0, Size: 1024576, Status: FREE

VALLOC 20 V0 OK
Status agora:
Pos: 0, Size: 20, Status: USED
Pos: 20, Size: 1024556, Status: FREE

VCALLOC 10 2 V1 OK
Status agora:
Pos: 0, Size: 20, Status: USED
Pos: 20, Size: 20, Status: USED
Pos: 40, Size: 1024536, Status: FREE

VREALLOC 30 V0 OK
Status agora:
Pos: 0, Size: 20, Status: FREE
Pos: 20, Size: 20, Status: USED
Pos: 40, Size: 30, Status: USED
Pos: 70, Size: 1024506, Status: FREE

VFREE 1
Status agora:
Pos: 0, Size: 40, Status: FREE
Pos: 40, Size: 30, Status: USED
```

```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
Pos: 70, Size: 1024506, Status: FREE

VALLOC 40 V1 OK
Status agora:
Pos: 0, Size: 40, Status: USED
Pos: 40, Size: 30, Status: USED
Pos: 70, Size: 1024506, Status: FREE

VFREE 0
Status agora:
Pos: 0, Size: 40, Status: USED
Pos: 40, Size: 1024536, Status: FREE

VCALLOC 15 2 V0 OK
Status agora:
Pos: 0, Size: 40, Status: USED
Pos: 40, Size: 30, Status: USED
Pos: 70, Size: 1024506, Status: FREE

VREALLOC 60 V0 OK
Status agora:
Pos: 0, Size: 40, Status: USED
Pos: 40, Size: 60, Status: USED
Pos: 100, Size: 1024476, Status: FREE

VREALLOC 10 V0 OK
Status agora:
```

```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
Pos: 0, Size: 40, Status: USED
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VREALLOC 1 V1 OK
Status agora:
Pos: 0, Size: 1, Status: USED
Pos: 1, Size: 39, Status: FREE
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VREALLOC 10 V1 OK
Status agora:
Pos: 0, Size: 10, Status: USED
Pos: 10, Size: 30, Status: FREE
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VREALLOC 0 V1 NULL
Status agora:
Pos: 0, Size: 40, Status: FREE
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VFREE 1
Status agora:
Pos: 0, Size: 40, Status: FREE
```

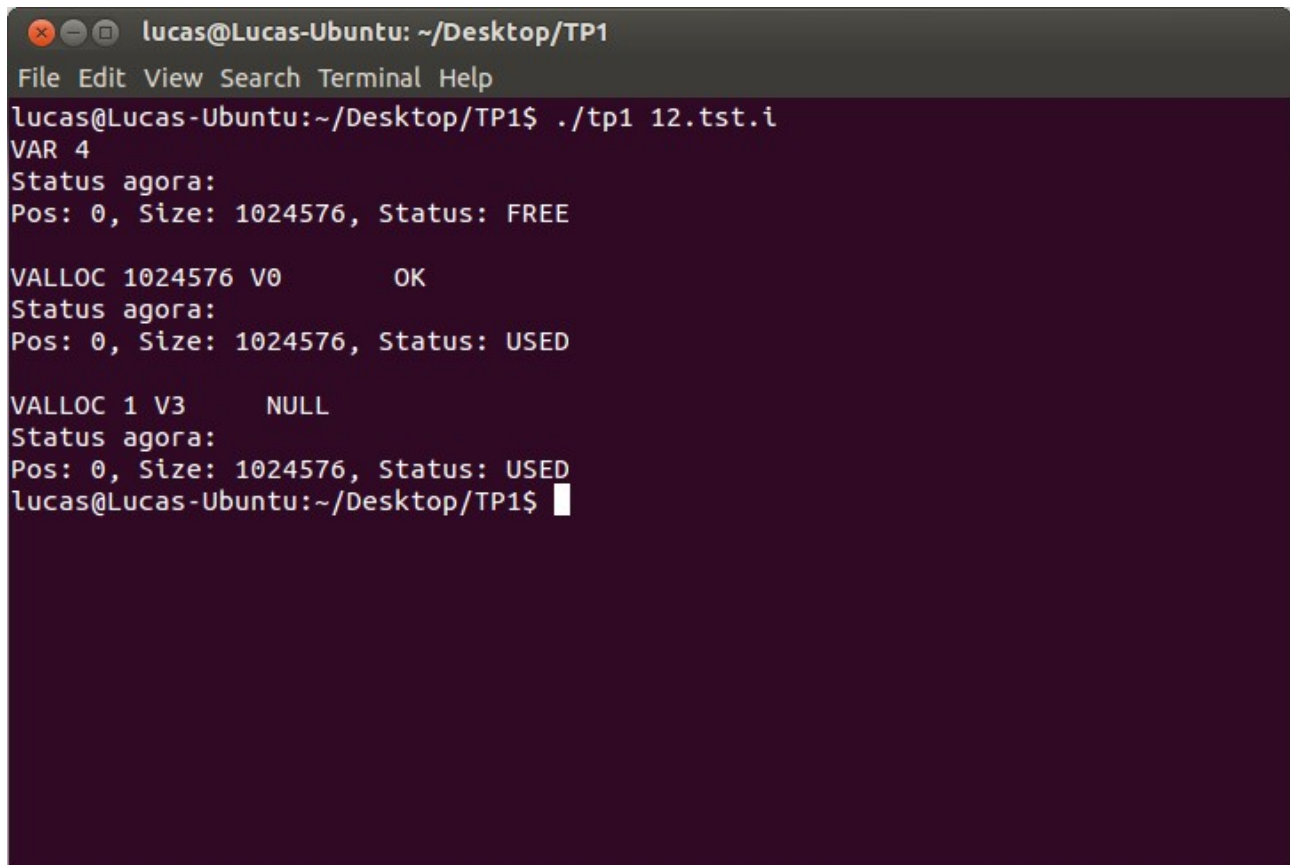
```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VALLOC 1024577 V1 NULL
Status agora:
Pos: 0, Size: 40, Status: FREE
Pos: 40, Size: 10, Status: USED
Pos: 50, Size: 1024526, Status: FREE

VFREE 0
Status agora:
Pos: 0, Size: 1024576, Status: FREE

VFREE 1
Status agora:
Pos: 0, Size: 1024576, Status: FREE
lucas@Lucas-Ubuntu:~/Desktop/TP1$
```

## 12.tst.i

A terminal window titled 'lucas@Lucas-Ubuntu: ~/Desktop/TP1' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of a program '12.tst.i'. The program prints 'VAR 4', then 'Status agora:', and 'Pos: 0, Size: 1024576, Status: FREE'. It then calls 'VALLOC 1024576 V0' which returns 'OK'. It prints 'Status agora:' and 'Pos: 0, Size: 1024576, Status: USED'. Next, it calls 'VALLOC 1 V3' which returns 'NULL'. It prints 'Status agora:' and 'Pos: 0, Size: 1024576, Status: USED'. The prompt 'lucas@Lucas-Ubuntu:~/Desktop/TP1\$' is shown at the end with a cursor.

```
lucas@Lucas-Ubuntu: ~/Desktop/TP1
File Edit View Search Terminal Help
lucas@Lucas-Ubuntu:~/Desktop/TP1$ ./tp1 12.tst.i
VAR 4
Status agora:
Pos: 0, Size: 1024576, Status: FREE

VALLOC 1024576 V0      OK
Status agora:
Pos: 0, Size: 1024576, Status: USED

VALLOC 1 V3      NULL
Status agora:
Pos: 0, Size: 1024576, Status: USED
lucas@Lucas-Ubuntu:~/Desktop/TP1$
```

## 5. Conclusão

O trabalho foi um ótimo exercício de modularização, e levou a um maior entendimento do gerenciamento de memória em C. O desenvolvimento do código ocorreu sem grandes problemas, mas o processo de debug foi longo, para tratar todos os casos variados de chamada das funções de alocação, além de todos os casos em que a memória pode se encontrar, mas a presença dos testes permitiu encontrar os problemas rapidamente, e o uso das ferramentas gdb e valgrind foi crucial para a agilização desse processo.

A maior dificuldade foi descobrir todos os casos possíveis para a chamada das funções e todos os estados da memória na qual devemos operar, os "corner cases" que podem causar um comportamento inesperado. Uma vez encontrados esses casos, resolver o problema era simples. Foi fácil escrever a primeira versão do código em si.

## Referências

[1] **C++ Reference – malloc:** Detalhes do funcionamento do malloc para várias situações.  
<http://www.cplusplus.com/reference/cstdlib/malloc/>

[2] **C++ Reference – realloc:** Detalhes do funcionamento do realloc para várias situações.  
<http://www.cplusplus.com/reference/cstdlib/realloc/>

[3] **C++ Reference – free:** Detalhes do funcionamento do free para várias situações.  
<http://www.cplusplus.com/reference/cstdlib/free/>

## Anexos

Listagem dos programas:

- bloco.c
- bloco.h
- valloc.c
- valloc.h