# Gerenciamento de memória[1][2]

## 1 Visão crítica geral

O uso correto da memória de computadores é requisito fundamental de sistemas bem projetados. Para isso, é importante entender alguns princípios básicos sobre gerenciamento de memória:

***Overhead* de utilização.** O processamento de dados é efetuado na CPU, que contém memórias internas para minimizar a sobrecarga computacional (*overhead*) de solicitar dados a outros dispositivos (RAM, disco, etc.). No entanto, tecnologias de armazenamento de alto desempenho apresentam elevado custo, limitando a capacidade de armazenamento destes dispositivos. Assim, o gerenciamento de memória em um computador é feito de maneira hierárquica, com unidades mais velozes apresentando menor capacidade de armazenamento e vice-versa. Sistemas de alta performance devem tentar minimizar o nível de estruturas de memória envolvidas em sua execução, devendo, portanto, otimizar seu uso de memória.

**Organização da memória.** Aplicações escritas em C/C++ organizam o espaço de memória disponibilizado pelo sistema operacional para sua execução de maneira segmentada. Primeiro, o código da aplicação é armazenado em um sub-bloco denominado **segmento de código**. Em seguida, as variáveis alocadas de maneira estática são armazenadas em um sub-bloco denominado **segmento de dados**. O bloco formado pela união do segmento de dados com o segmento de código é denominado *espaço de alocação estática*.

O espaço restante é denominado *espaço de alocação dinâmica*, sendo reservado para alocações em tempo de execução. Este espaço é subdividido em ***stack*** (pilha) e ***heap***, que não apresentam divisão fixa entre elas. Especificamente, a pilha é utilizada para armazenamento de contexto (variáveis de escopo), crescendo de forma significativa em procedimentos recursivos. Por sua vez, a *heap* é utilizada para armazenar variáveis alocadas dinamicamente por instruções da aplicação. Uma vez que crescem em direções opostas compartilhando o mesmo espaço, é possível ter estouros de pilha (*stack overflow*) ou de *heap* (*heap overflow*) devido a um procedimento recursivo que se expanda demasiadamente (estouro de pilha) ou a uma aplicação que solicite memória em excesso (estouro de *heap*).

**Alocação dinâmica.** A possibilidade de criar variáveis em tempo de execução ou definir espaços de armazenamento de tamanho variável oferece grande expressividade e flexibilidade aos sistemas de computação. No entanto, instruções de alocação dinâmica são processadas pelo sistema operacional, o que gera um overhead computacional a cada solicitação. Ademais, nem todas as plataformas fornecem a possibilidade de alocação dinâmica, como muitas plataformas de sistemas embarcados. Por fim, linguagens que permitem o controle de baixo-nível dos espaços de endereçamento, como C/C++, delegam uma séria responsabilidade aos programadores.

Dadas as considerações acima, a decisão entre o uso ou não de alocação dinâmica costuma variar em função do tipo de sistema que se desenvolve:

---

[1] Roteiro de estudo fornecido na disciplina de Linguagem de Programação 1 (LP1) da Universidade Federal do Rio Grande do Norte (UFRN).

[2] Autor: Leonardo Bezerra (`leobezerra@imd.ufrn.br`).

- **Sistemas embarcados** – em geral, deve-se evitar alocação dinâmica neste tipo de sistema. As exceções são aplicações desenvolvidas para plataformas que oferecem alocação dinâmica e não são utilizadas para processamento de informações em tempo-real.
- **Sistemas tradicionais** – em geral, deve-se adotar alocação dinâmica neste tipo de sistema. Seu uso beneficia o sistema em diversas maneiras, sendo tolerável o *overhead* de alocação.
- **Sistemas de alta performance** – em geral, deve-se usar a alocação dinâmica somente quando inevitável. No entanto, o correto uso de alocação estática para substituir a alocação dinâmica requer um bom nível técnico da equipe de desenvolvimento.

# 2 Alocação estática em C++

Tanto C como C++ fornecem dois tipos de acesso a memória quando se trabalha com alocação estática. No **acesso direto**, a variável declarada representa um espaço de memória. No **acesso indireto**, o espaço de memória é acessado através de seu endereço, que por sua vez pode ser armazenado em uma variável especial para endereços, denominada *ponteiro*. Além disso, o C++ permite que um espaço de memória seja associado a múltiplas variáveis, utilizando acesso direto através de *referências*. Por fim, tanto C como C++ permitem ambos os tipos de acesso para vetores, o que nesta disciplina será denominado **acesso misto**.

## 2.1 Exemplos

A seguir, alguns exemplos das diferentes possibilidades oferecidas pela linguagem C++ (os códigos estão disponíveis através dos links ou no apêndice adicionado no SIGAA):

1. **Acesso direto** (sem referências):
   - *Variáveis escalares:* `http://cpp.sh/33n3i`
   - *Vetores:* `http://cpp.sh/2fkt`
2. **Acesso direto** (com referências[3]): para declarar uma variável como referência para outra, é necessário utilizar o símbolo & logo após o tipo de dado contido na variável.
   - *No mesmo escopo:* `http://cpp.sh/5owne`
   - *Como parâmetro:* `http://cpp.sh/7rkej`
3. **Acesso indireto** (com ponteiros): para declarar uma variável como ponteiro para outra, é necessário utilizar o símbolo * logo após o tipo de dado contido na variável. Note que, como ponteiros devem armazenar apenas endereços de memória, é necessário utilizar operadores auxiliares para obter e acessar endereços. Especificamente, o operador & retorna o endereço de memória de uma variável, enquanto o operador * acessa o endereço especificado por um ponteiro[4].
   - *No mesmo escopo, com variáveis escalares:* `http://cpp.sh/7phkr`
   - *No mesmo escopo, com vetores:* `http://cpp.sh/8opxh`
   - *Como parâmetro, com variável escalar:* `http://cpp.sh/8eqp3`
   - *Como parâmetro, com vetores:* `http://cpp.sh/4daq5`

## 2.2 Laboratório

*Palíndromos* são palavras ou frases que podem ser lidas em ambos os sentidos (ex.: ana, ovo, osso, radar, reviver, rodador)[5]. Uma possível forma de verificar se uma palavra/frase é um palíndromo é invertê-la e testar se a versão invertida é idêntica à original. Neste exercício de implementação, você deve implementar um verificador de palíndromos, restrito ao caso de palavras[6]. O código modelo está disponível no SIGAA (*modelo-palindromo.zip*).

[3] Referências também são conhecidas como *aliases*, ou apelidos, já que na prática são nomes de variáveis diferentes para endereçar um mesmo espaço de memória.

[4] Note que o léxico do C++ é confuso neste aspecto, uma vez que os símbolos * e & são utilizados tanto para denotar que uma variável é, respectivamente, um ponteiro ou uma referência, como para obter e acessar endereços. Neste último caso, o símbolo & denota o operador de *referenciamento* (obtenção de um endereço de memória), enquanto o símbolo * denota o operador de *desreferenciamento* (acesso a um endereço de memória).

[5] Da esquerda para a direita ou da direita para a esquerda.

[6] Para a verificação de frases, é necessário desconsiderar espaços e pontuações. Este será o tema de outro exercício no futuro.

# 3 Alocação dinâmica em C++

Quando lançado, o C++ trouxe grandes melhorias em relação ao C no que diz respeito ao gerenciamento de memória com alocação dinâmica. Mais precisamente, o C++ oferece os operadores `new` e `delete`, detalhadas abaixo:

**`new`, `new[]`** o operador `new` permite a alocação e inicialização de variáveis escalares, em substituição à função `malloc` do C. Para alocação de vetores, utiliza-se o operador `new[]`. As principais vantagens dos operadores `new` e `new[]` são a segurança de tipos e a possibilidade de inicializar tipos abstratos de dados de forma customizada.

```
1 int * var1 = new int;
2 int * v1 = new int[5];
3 int n = 3;
4 int * v2 = new int[n];
```

**`delete`, `delete[]`** o operador `delete` permite a liberação de variáveis escalares alocadas dinamicamente com o operador `new`. Por sua vez, o operador `delete[]` deve ser usado para liberar vetores alocados dinamicamente com o operador `new[]`. A principal vantagem dos operadores `delete` e `delete[]` é a possibilidade de liberar memória de forma customizada no caso de tipos abstratos de dados.

```
1 delete var1;
2 delete [] v1;
3 delete [] v2;
```

**Correspondência de operadores.** Uma vez que C++ ainda permite o uso das funções `malloc` e `free`, é importante entender que cada método de liberação de memória só pode ser usado para liberar um espaço alocado por seu método de alocação correspondente. Assim, uma variável alocada com `new` só pode ser liberada com `delete`. Por sua vez, vetores alocados com `new[]` devem ser liberados com `delete[]`. Por fim, qualquer espaço de memória alocado com a função `malloc` (ou `calloc`) devem ser liberados com a função `free`.

## 3.1 Exemplos

Além dos operadores `new` e `delete`, o C++ oferece uma outra forma de acesso a espaços de memória alocados dinamicamente. Além do acesso indireto a variáveis escalares e vetores, C++ também permite acesso misto a vetores, como exemplificado abaixo[7].

1. **Acesso indireto** (com ponteiros):
   - *No mesmo escopo, com variáveis escalares:* `http://cpp.sh/85yq3`
   - *No mesmo escopo, com vetores:* `http://cpp.sh/2c7do`
   - *Como parâmetro, com variável escalar:* `http://cpp.sh/5awj`
   - *Como parâmetro, com vetores:* `http://cpp.sh/9isn2`
2. **Acesso misto**: aplica-se quando se deseja alocar um vetor cujo tamanho será determinado pelo conteúdo de uma variável[8].
   - *No mesmo escopo:* `http://cpp.sh/9yqno`
   - *Como parâmetro:* `http://cpp.sh/473qm`

## 3.2 Laboratório

No exemplo do código com palíndromos, implemente uma interface com o usuário solicitando a palavra que deverá ser testada. Note que esta implementação corresponde à `main` do programa, devendo ser implementada no arquivo `application/lab01.cpp`. Além disso, o uso de alocação dinâmica neste trecho da aplicação exigirá o uso de alocação dinâmica também no arquivo source `src/palindromo.cpp`.

[7] É importante salientar que o acesso misto só é recomendado no caso de tipos primitivos. Seu uso com tipos abstratos de dados deve ser cauteloso, uma vez que é possível ter vazamentos de memória quando o tipo abstrato apresenta alguma variável interna alocada dinamicamente.

[8] Note que é possível usar a função `free` para liberar o espaço alocado para um vetor neste caso, mas isto não é necessário, uma vez que neste tipo de alocação o compilador já adiciona operações de liberação de memória ao término da execução do programa. No entanto, esta é uma opção que pode ser útil caso seu programa necessite de uma quantidade excessiva de memória

# 4 Matrizes

A alocação de matrizes em C++ ocorre de forma similar à alocação em C. De fato, a única diferença entre as duas linguagens refere-se ao uso dos operadores `new []` e `delete []`.

## 4.1 Exemplos

- *Alocação estática:* `http://cpp.sh/9ehce`
- *Alocação dinâmica, no mesmo escopo:* `http://cpp.sh/5bemn`
- *Alocação dinâmica, como parâmetro:* `http://cpp.sh/2kbe5`

# 5 Depuração

Inevitavelmente, a possibilidade de gerenciar memória manualmente em C/C++ leva a erros por parte de programadores com pouca experiência nestas linguagens. Para auxiliar a depuração do seu código, duas ferramentas principais costumam ser utilizadas:

**gdb** é o depurador oficial distribuído junto a compiladores GCC. Seu uso é recomendável na análise da lógica de programação de sistemas, uma vez que permite a análise interativa do código da aplicação.

**Extra!** Explique o funcionamento do `gdb`. Especificamente, explique quais flags de compilação devem ser ativadas para que o `gdb` possa ser utilizado, bem como os principais comandos oferecidos pelo `gdb`. Por fim, demonstre seu uso em um código de aplicação concreto.

**valgrind** é uma coleção de ferramentas tradicionalmente utilizada para a verificação de erros relacionados à memória em uma aplicação[9]. Diferentes funcionalidades são oferecidas por este software, controladas por flags de ativação. Em geral, as flags mais utilizadas são:

- `-v`, que torna as mensagens do `valgrind` mais explícitas (mais verboso).
- `--leak-check=full`, que solicita ao `memcheck` que identifique possíveis vazamentos de memória.
- `--show-reachable=yes`, que inclui no diagnóstico de vazamentos de memória regiões que poderiam ser recuperada pela a aplicação.

O arquivo `depuracao.zip` contém exemplos dos principais tipos de erros e como o `valgrind` os relata. Para compilar cada exemplo, use `make termo`, substituindo termo por uma das opções em negrito abaixo. Para executar o valgrind, use `valgrind ./termo -v --leak-check=full --show-reachable=yes`.

- **leak**: representa os casos em que houve vazamento de memória (*memory leak*). Isto acontece quando o programador esquece de liberar uma região de memória alocada dinamicamente.
- **overflow**: representa os casos em que se faz um acesso ou escrita além da área alocada. um estouro de heap (*heap buffer overflow*). Como discutido anteriormente, isto ocorre quando se tenta alocar dinamicamente mais memória do que o permitido pelo sistema operacional.
- **uninit**: representa os casos em que se tenta utilizar uma variável não inicializada (*unitialized*).
- **free**: representa os casos em que se tenta liberar uma região já liberada anteriormente (*double memory free*).

# 6 Laboratório

Para finalizar este roteiro, você deverá implementar uma aplicação denominada `matrizes`, em um projeto de nome *lab02*. Sua aplicação deverá seguir o mesmo modelo de organização utilizado para trabalhos anteriores, e deverá conter três funcionalidades principais:

1. `void print(int **, int, int);` – imprime uma matriz n x m.
2. `void transpose(int **, int, int);` – transpõe uma matriz n x m.
3. `int ** diagonal(int **, int, int);` – reduz o espaço necessário para o armazenamento de uma matriz simétrica.

[9] Esta análise é feita implicitamente pela ferramenta `memcheck` da coleção `valgrind`.