Alocação de Memória
Aplicação com Linguagem C++

Rafael Junqueira

Sumário

Público Alvo

Público Alvo.

O presente documento é direcionado ao público de desenvolvedores que atuam com foco em desempenho e o gerenciamento de memória.

Aos profissionais atuantes como cientistas computacionais, programadores, desenvolvedores e engenheiros computacionais.

1. Introdução

Memória é a parte essencial de um computador. Não existe 'computação' sem o entendimento de memória e sua parte unicamente vital. Sem Memória Ram, não há como executar rotinas e assim nenhuma ação, nem de ligar o computador.

Sem memória de armazenamento não há como alocar dados, nem de recuperá-lo. Para que um conhecedor se torne técnico na matéria tecnólogica especificamente computacional, precisa sucintamente compreender que operações computacionais envolvem processos aritméticos.

Tudo em computador é matemática. Soma, subtração, multiplicação e divisão. E cálculos sofisticados são usados para operar em níveis mais elaborados. Assim nasceu arquiteturas mais sofisticadas que a **RISC** (Reduced Instruction Set Computer) , as **CISC** (Complex Instruction Set Computer).

A elaboração delas foi possível quando a memória se tornou um pivô central na indústria que viabilizou o entendimento que ela é na prática a energia e o combústivel dos sistemas. As antigas linguagens lidavam diretamente com registradores, muitas delas ainda existem hoje, como o **Assembly**.

A comunicação era tão direta, que esses registradores eram específicos, a depender da arquitetura, eles podem ser gerais ou ainda mais complexos. Quando foi necessário que as aplicações

evoluíssem em um processo lógico mais 'legível' a leitura do ser humano (Interface homem-máquina).

O Assembly que era prioritamente cálculo-hardware, a abstração se fez e linguagens com uma roupagem mais legível foram criadas. Como o Basic, C, C++, Java.

E outras que distanciam a necessidade do técnico se preocupar com o arranjo dos endereços de memória, as chamadas línguagens de alto nível.

Os três tipos de níveis:

* Baixo nível - Linguagem de máquina;

* Médio Nível - Linguagem hibrída de máquina e humana;

* Alto Nível - Linguagem Humana.

E o resumo mais direto é que Baixo e Médio Nível possibilita manusear com a memória de forma direta, enquanto que a alto nível não. Essas últimas possuem coletores, garbage collection ou tratamento de alocação de memória feita por baixo dos panos sem a devida interação direta do programador.

Por um lado, isso possibilitou menos "perigos" pois que mexer com a memória também promove, sem o cuidado tomado, vazamentos, quebras, congelamentos e surtos.

Por outro lado também permite que você "comande" os céus, já que como foi dito no começo dessa introdução, memória é unicamente a parte vital de um sistema computacional, sem ela, nada existe.

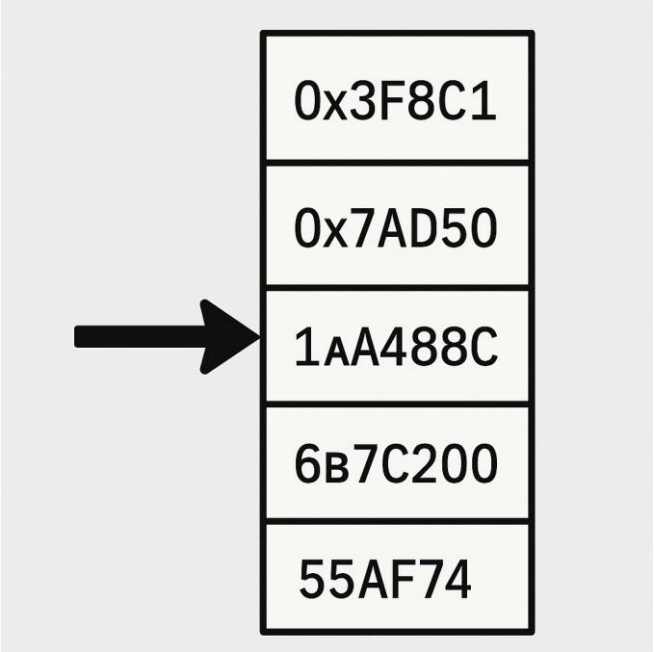
Por isso neste artigo, iremos usar o C++ para entender melhor esse conceito de alocação de memória. E a parte que, a adesão a línguagens de baixo e médio nível, tem por objetivo exatamente de nos proporcionar o controle do "universo".

2. Conceito de Memória

Nós temos três tipos de memórias básicas que são a flexível ou **RAM** (Random Acess Memory) e a rígida ou **SSD** (Solid State Drive \ Hard Memory) e a memória **ROM** (Read-Only Memory) que são dados de apenas leitura, armazenado na **BIOS** (Basic Input/Output System).

A memória RAM que podemos em nosso computador identificar como 8gb, 16gb e 32 gb e que muitos aplicativos exigem um tanto mínimo, é a responsável por executar ordens em nosso sistema. Sem ela, torna-se impossível que uma ação seja executada, acessada ou mesmo finalizada.

Portanto quanto mais memória RAM tivermos, mais rápido os processos são. Existe uma relação direta com a memória ram e a mémoria ridiga. Ela funciona da seguinte forma.



A figura acima demonstra como uma memória RAM 'seta' o endereço, sempre representado por uma sequência de base 16 (hexadecimal). Essa informação visual é onde iremos armazenar comandos.

Curiosidade.

Quer saber porque hexadecimal é uma convenção de identificação do endereço de memória?

Nós sabemos que o sistema computacional entende sequências binárias (100011101), mas para facilitar a leitura, a convenção permite que 'compactar' para valores mais legíveis como o hexadecimal.

Pois que um número hexadecimal equivale a 4 bits e dois equivale a 8 bits (1 byte). Se queremos por exemplo armazenar uma sequência vetorial (array) e ela dá digamos 32 bytes. Sabe como ficaria no sistema 'original' **binário**?

Endereço de memória (00100000)

E em Hexadecimal?

Endereço de memória (0x20)

Quando um jogo exige um determinado valor de memória RAM, ele pressupõe que precisa daquela quantidade para executar ações no aplicativo. É como se você tivesse como requisito ter 16 litros de gasolina para pecorrer uma distância, se não tiver, você é impedido.

Diferente da gasolina, da qual o suficiente impedirá de você chegar no destino e não de rodar até um certo limite, sistemas desenvolvidos definem o quanto de memória são necessárias minimamente para que uma ação seja executada, impedindo que

você 'burle' a exigência de 32gb por uma de 16gb, alegando metade do caminho.

Os endereços que uma memória ram podem acessar são proporcionais a sua execução. Por exemplo, você precisa de endereços de memória para executar ações, porque seria equivalente 'pontos'. Se não tiver pontos, não pode executar uma ação.

Em 16gb ram temos 17,1 bilhões de endereços de memória. E se um jogo exige 17,100,000,001...você já não tem como rodar. Parece injusto. Mas um endereço de memória aceita tantos bytes por dado específico. E isso pode ser o necessário para executar uma ação que faz o jogo iniciar.

Tudo no computador é memória. É como o 'dinheiro' do sistema. Cada endereço de memória é alocado um valor. Quando nós tratamos de programação, aqueles tipos básicos (Data Type) que usamos para declarar variáveis, aqueles valores ali atribuídos vão para esses endereços de memória. Onde serão tratados, operados e executados.

Independente se você tem acesso a memória ou não, é assim que funciona qualquer ação ou programação que você use. Em memória rígida, nós temos a memória geral do computador, que permite armazenar dados, inclusive de executar o que a memória RAM precisa.

Sim porque sem a memória rígida ou pouca dela, a memória RAM não executa nada também. Experimente, deixar o seu computador com pouco espaço em disco. E note que sua memória RAM terá uma leve dificuldade em operar.

16gb RAM - 15 gb de memória em disco, note o lag? Notou? Pois é, isso acontece porque a memória RAM é uma memória que é

usada para executar o que está armazenado. Sem a memória rígida, ela não 'pega nada' e 'não executa nada'.

E a memória ROM (Read-Only Memory) possui dados de leitura e possui não volatilidade, os dados não ficam comprometidos no desligamento do sistema, ela é usada para registrar rotinas de firmware e essencialidades para o funcionamento.

Para ligar o computador, a memória RAM acessa o ROM. Aqui como é apenas leitura, ela não acessa isso no SSD. Portanto ela consegue acessar o computador e ativá-lo sem que a mesma tenha memória rígida.

Vamos resumir:

RAM - Executa rotinas, ações e faz o computador funcionar.

SSD - Armazena dados, operações e rotinas.

ROM - Armazena informações e instruções de firmware.

3. Gerenciamento e Alocação de Memória com C++

O entendimento do capítulo 2 se faz muito presente para compreender como iremos trabalhar agora na linguagem C++ a definição de alocação de memória, aplicação e funcionamento. Porque aqui essa parte da memória será um pedaço abstrato de nosso raciocínio.

Nós normalmente associados a pilha da memória como aquela que destaquei como figura. E nós usamos a imagem de um 'array' (vetor) para compreender como cada índice é um endereço de memória, cada iteração é um acesso ou armazenamento da memória e cada comando é uma rotina calculada, vinculada e dispensada.

Por isso é fundamental entender aqui conceitos básicos da Linguagem C++. Vamos aos pontos que tornam a alocação de memória uma ferramenta poderosa para controlar renderizações, efeitos visuais,

processamento no limite e performance ideais. E a razão que o torna uma das tecnologias mais aplicadas em games.

Nós definimos em C++ um conceito chamado referência e ponteiro. Ela costuma ser um bicho de sete cabeças. Mas se a leitura desse artigo está em dia, não terá problemas em compreender. Iremos devagar, esse é o recurso mais poderoso e precioso dessa linguagem.

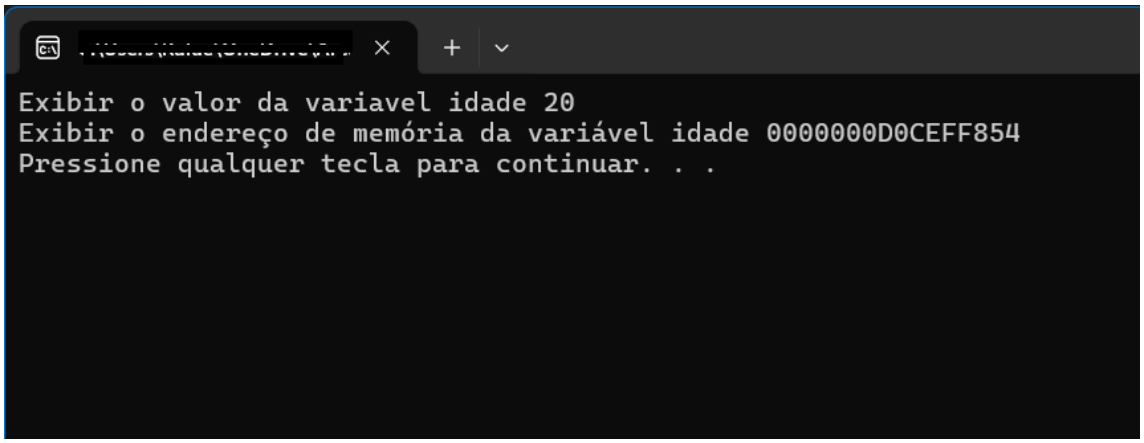
Referência é uma ação dentro da programação que identifica o endereço de memória. Ele infere, e podemos lançar isso em uma impressão de tela (std::cout) ou atribuir isso a uma variavel (int variavel =). O que acontece que essa "informação" é apenas um dado de leitura e não ainda um endereço de memória.

É como se fosse uma string, é apenas uma informação. Sem valor, sem poder e nem nada. É como atribuir std::string nome = "Rafael". Apenas isso.

E isso faz toda a diferença em entender o próximo tópico, que é ponteiro. Mas antes, vamos ver como isso acontece no código.

```
#include <iostream>
#include <locale>

//Criando referência de uma variável
// Toda vez que executar o código, será um endereço diferente
// Usamos o símbolo & para instruir que variável deverá exibir o endereço

int main()
{
	setlocale(LC_ALL,"Portuguese_Brazil");
```

```
        //Uma declaração simples, atribuido o valor.
        int idade = 20;
        std::cout << "Exibir o valor da variavel idade " << idade << std::endl;
        std::cout << "Exibir o endereço de memória da variável idade " << &idade << std::endl;
        system("Pause");
        return 0;
}
```



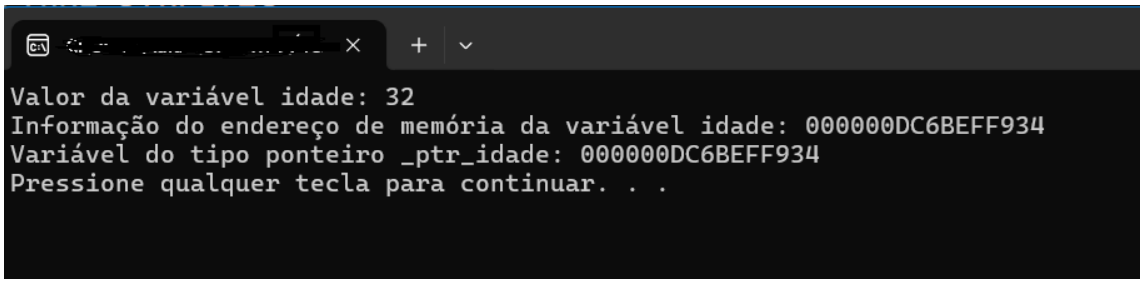
Lembrando que o endereço de memória não seria o mesmo que esse resultado, uma vez que a cada execução do programa, ele aloca em um novo endereço.

Vamos nos certificar de compreender que a informação que estamos exibindo aqui não é o endereço de memória, mas sim 'uma nota' dele. Se você tentar fazer algo com essa informação, neste caso ele apontará para o tipo int * __ptr64.

Neste caso ele passa uma informação que se trata de um ponteiro do tipo inteiro em uma arquitetura de 64 bits. Mas neste exato momento referência não "opera" como um ponteiro devidamente.

Para isso acontecer é o que vamos aprender a fazer agora.

```
#include <iostream>

#include <locale>


int main()

{

	setlocale(LC_ALL, "Portuguese_Brazil");

	//Declaração simples de uma variável com valor inicializado 32

	int idade = 32;


	//Aqui declaramos uma variável do tipo ponteiro que recebe o endereço de memória de idade

	// Pode ser também int *_ptr_idade

	int* _ptr_idade = &idade;


	std::cout << "Valor da variável idade: " << idade << std::endl;

	std::cout << "Informação do endereço de memória da variável idade: " << &idade << std::endl;

	std::cout << "Variável do tipo ponteiro _ptr_idade: " << _ptr_idade << std::endl;

	system("Pause");

	return 0;

}
```



Parece a mesma coisa quando a gente imprime Informação do endereço de memória da variável e o variável do tipo ponteiro. Mas a natureza mudou aqui. No primeiro caso é apenas uma informação registrada na variável &idade com o símbolo de referência, é como se consultássemos.

No segundo caso estamos definimos uma variável que chamamos de _ptr_idade e poderia ter sido qualquer nome (importante porque existem nomes pré-definidos para estes casos também) e temos um símbolo que é o * (asterisco) que agora vai definir essa variável do tipo ponteiro e não apenas inteiro.

Ela vai se comportar agora como uma SETA que irá apontar uma endereço de memória, podendo manipulá-la, gerencia-la e liberá-la. Por isso que é chamado de ponteiro, por ser uma SETA na sua operação prática.

Existe um importante adendo, nem sempre nós usarmos a alocação dinâmica de memória. Ela tem alguns requisitos. E para ser usado precisa entender que isso é que torna a linguagem poderosa e essencial para programas que fazem uso de processamento e tempo de execução dinâmico, como games.

Toda ação dinâmica tem a possibilidade de redimensionar índices. Ou seja, imagine um surgimento de (respawn) de inimigos em uma cena. Para que eles aparecem, um vetor (array) é usado para alocá-los. Então isso é feito em tempo real.

Um vetor que é alocado em tempo real sem pré-definição 'calcula' memória (RAM) em tempo real de processamento, será que ele vai conseguir alocar corretamente ou vai estourar a pilha? (Vazamento de memória).

Se não fizer certo, pode dar lag, congelamento ou saída inesperada do sistema.

Como definimos que algo 'crie um espaço de memória', libere e invalide para que não seja usado indevidamente durante o processo? Antes de passar para o próximo, certifique-se de reler o que vimos antes de passar para não acumular dúvidas.

```
#include <iostream>

#include <vector>


//Usar o vector nos permite entender alocação dinâmica

//Redimensionamento em tempo de execução



int main()

{

        // Aqui criamos o ponteiro e o alocamos o espaço de memória

        // Uso do new std::vector<int>();

        std::vector<int>* idade = new std::vector<int>();

        //É usado -> no lugar de .método, pois estamos usando ponteiro

        idade->resize(200);

        // Aqui liberamos o espaço de memória

        //Uso do delete

        delete idade;

        // Aqui anulamos esse variável como ponteiro

        idade = nullptr;
```

```
    system("Pause");
    return 0;
}
```

Por que não Malloc e free? Que são métodos que fazem exatamente o mesmo que New e Delete. Aqui precisamos entender a ambiguidade. É muito comum, porque há duas distinções definições porque não usarmos. O primeiro motivo é que Malloc e Free é pertencente a linguagem C e segundo que é mais apontado para lidar com sistemas legados (antigos).

E aproveitamos que tantos que confundem C com C++ atribuindo a junção de C/C++. Compreendam que apesar da aproximidade das duas linguagens, encarem com um C e a outra um update. Como acontece com as versões de C++. Não faz muito sentido em usar recursos do C11 com a versão C24, faz? Não. C e C++ são duas linguagens diferentes entre si, compartilham similaridades na sintaxe.

Mas Malloc e Free apesar de 'poder' funcionar no C++ e vocês irão de fato achar em livros ou internet informações contradizendo o que eu falei aqui. Mas é porque são métodos 'funcionais' em C++, mas elas não são recomendas por mais um motivo, além daqueles que eu disse ali em cima.

Malloc e Free não tem construtores, por fazerem parte de uma linguagem que não é de classes e sim procedural. Construtores presentes em Classes (do C++) garante integridade de funcionamento. New e Delete possuem, isso garante que o gerenciamento de memória não sai pela culatra, uma das razões que C é considerado "um desastre" nisso.

Além do New e Delete no código tem uma coisa nova. O que é aquilo de atribuir nullptr a variável idade? Não bastaria só aplicar o delete, porque apesar do comando sugerir "apagar", ela se refere a liberar a memória do ponteiro. Sabe aquele valor hexadecimal? Não está mais atribuído a variável idade.

Mas isso não significa que está seguro. Idade continua sendo um ponteiro. E se ele ainda pode SETAR, ele pode 'sofre bug' ou mesmo apontar por algum bug no processamento. Então o que se faz atribue a variável idade a remoção de status de ponteiro. Ou seja o std::vector<int> idade passa a não ser mais ponteiro. É uma estrutura de dados vector comum sem 'acesso a memória'.

Por isso o nullptr (Anular o ponteiro). Invalida.

Vamos resumir:

***** Defina o tipo um ponteiro;

New - Aloca a memória (seta);

delete - Libera a memória (mas a variável continua ponteiro);

nullptr - Remove o status de ponteiro, é uma variável segura.

Quando usar ponteiros?

E mais um apontamento muito importante. Não se usa ponteiro o tempo inteiro. Vamos entender quando usamos, pois que esse tipo de dado costuma ser 'volátil' usá-lo sem um compromisso adequado costuma gerar problemas desnecessários.

Considere sempre em "Tempo de execução" (runtime):

* Redimensionar tamanhos de arrays;

* Definirmos ou redefinirmos a natureza;

* Renderização de imagem;

* Execução de partícula e efeitos visuais;

* Geração de dados 'subestimado'.

É muito comum ver códigos onde o redimensionamento de array como o std::vector<> que permite exatamente essa mudança de tamanho em processamento. Mas quando se faz isso, pensamos que talvez a

estrutura do vector o faça com ressalvas, certo? Mas não é bem assim que funciona.

Um redimensionamento de 200 posições pode não ser um impacto grande. Mas quando falamos de 100 mil posições isso muda da água para o vinho. E estamos falando de valores numéricos. Ou acionarmos valores de double, que fazem dobrar o tamanho de cada índice. Isso seria o tamanho em megabytes x 100 mil posições.

Agora pegue frames, pixels, modelos 3d, partículas de um game. Que tende a aumentar de tamanho em tempo real. Sim, é indispensável que estejamos trabalhando com ponteiros.

Agora para programas em que tudo é definido antes da compilação, operações de iteração (pecorrer vetor), cálculo, ou leitura de arquivo (depende também se isso vai mudar natureza de configurações no programa, senão não precisa) não tem necessidade usar ponteiro.

4. **Conclusão.**

Vimos neste artigo um resumo rápido de arquiteturas CISC e RISC, do papel de registradores e uma citação de Assembly muito curta.

Falamos sobre os três tipos de memórias SSD, RAM e ROM. Mencionamos sobre como os computadores funcionam basicamente. A introdução nos permite compreender com alocação de memória seria gerenciada por códigos.

Depois podemos ver alguns exemplos de como referenciar um endereço de memória sem que ainda seja possível gerenciá-lo. Mas que é o primeiro passo para começara definir ponteiros.

Em seguida vimos o que eram ponteiros e porque usamos muitas vezes a palavra 'Seta ou Setar', porque ele faz isso, ele aponta para um endereço de memória.

Notamos como criar um ponteiro (int* ptr / int *ptr), como acessar uma memória (new), como liberar memória (delete) e como remover o

status e ponteiro de uma variável (nullptr), que ainda pode ser usar no sistema para outros fins. Entendemos porque não usamos Malloc e Free, enumerando três motivos:

São métodos de alocação de memória dinâmica da linguagem C, é aplicado para sistemas legados (antigos) e não possuem segurança de construtores o que pode causar vazamento de memória ao menor erro.

Depois fizemos uma diferenciação da linguagem C da C++. E por fim definimos quando nós usamos os ponteiros de fato.

Este artigo viabiliza um guia básico de como os ponteiros devem ou não serem usados e como usando C++. De uma forma didática com algum nível de ensino básico de como os computadores funcionam.

Não entramos é óbvio no processo completo, nem por citação coloquei a CPU (ULA) e o microcontroladores nesta explicação, pois que até possuem uma finalidade completa, mas estaríamos nos prolongando muito em uma explicação que seria demasiadamente cansativa e talvez até confusa para explicar algo que já é um pouco complicada.

5. Bibliografia

TANENBAUM, Andrew. S. Organização Estrutura de Computadores. Edição 6ª, 2013. Pearson Universidades.

STROUTSTRUP, Bjarne. Princípios e Práticas de Programação com C++. Edição 3, 2025. Bookman.

ZHIRKOV, Igor. Programação em Baixo Nível - C, Assembly e execução de porgramas na arquitetura Intel 64. Edição 1, 2018. Novatec.

ROCHA, Antônio Adrego da. Análise da Complexidade de Algoritmos. Edição 1, 2014. FCA.

6. Autoria

Criação em julho de 2025.

Sites:

https://mundopauta.com.br/
https://www.youtube.com/@JuncaGames