Módulo 14 - Memória e alocação dinâmica

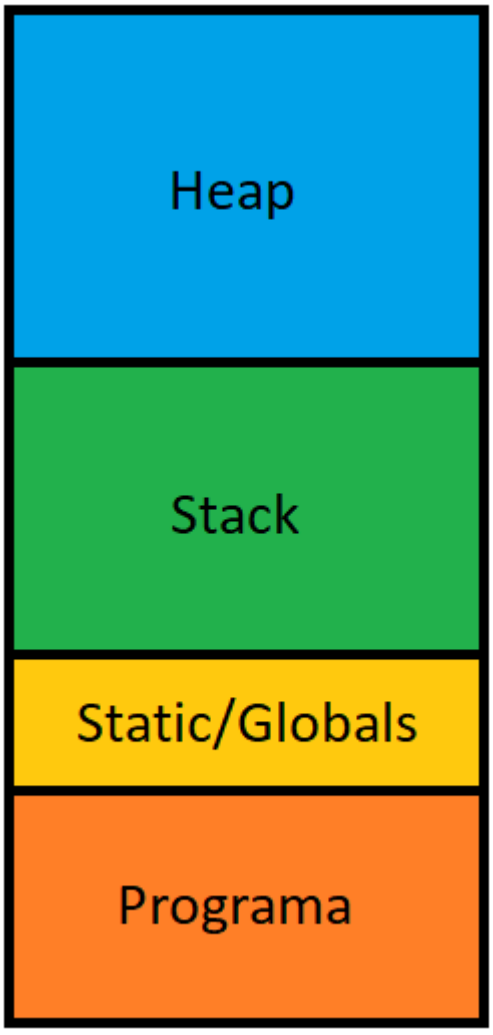
Já ficou claro que em C temos um acesso direto à memória do sistema que estamos trabalhando. Não é atoa que sistemas que necessitam de altíssima eficiência de processamento, como sistemas embarcados, utilizam C como sua principal linguagem de programação.

Por isso, é extremamente necessário conhecer a arquitetura de memória no processo de computação. Até agora, consideramos a memória do computador como um enorme array de endereços, onde variáveis são alocadas de forma “aleatória”. Tudo isso era verdade, menos pelo “aleatório”.

A memória é de fato um array enorme de endereços, porém, sua utilização não ocorre de forma aleatória. Existe uma arquitetura “virtual” envolvida na utilização da memória que a separa em setores de heap, stack, program memory, entre outros. A arquitetura é dita virtual pois existe apenas como conceito, não existe literalmente espaços físicos na memória chamados de heap, stack, etc. Mas o sistema operacional segue esta arquitetura a risca para alocar a memória de seus programas.

Mesmo em sistemas de mais baixo nível que não possuem Sistema Operacional embutido, como microcontroladores, já trazem em sua arquitetura este modelo de manipulação de memória. Por isso, este conteúdo é crucial para qualquer tipo de programador que quer desenvolver sistemas.

Memória da aplicação



Seu programa, seja ele qual for, irá consumir memória na arquitetura indicada acima. A parte **programa** é onde ficara armazenado todo o executável de seu programa. Portanto, aquele binário que vemos com `Format-Hex` é o que será armazenado aí.

Em **Static/Globals** é onde ficarão armazenadas suas variáveis globais e as definidas como static. Veja no código abaixo onde eu printo no console o endereço de uma variável static e duas variáveis comuns:

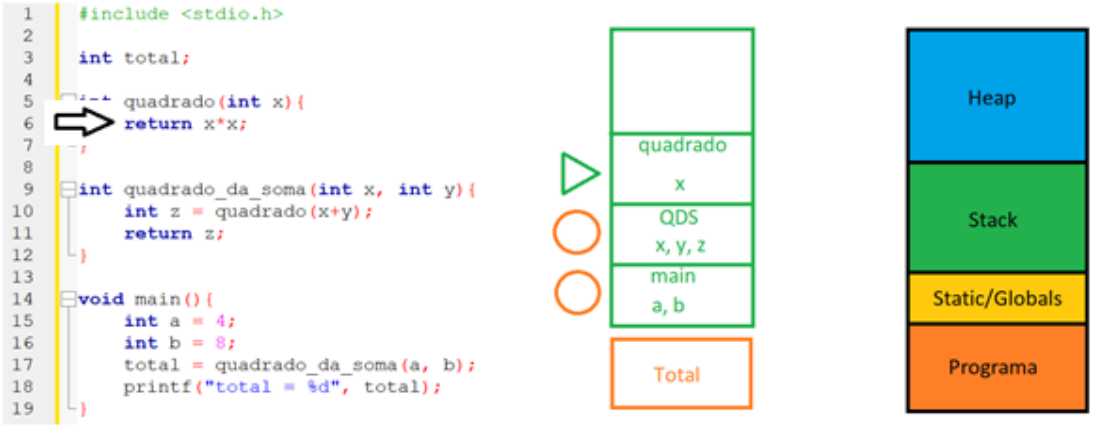
```
static int x;
int y;
int z;
printf("X-> %x\n", &x);
printf("Y-> %x\n", &y);
printf("Z-> %x", &z);

// --> Saida
// X-> 407020
```

```
// Y-> 61ff1c
// Z-> 61ff18
```

Veja que X (variável static) ficou armazenada mais distante das outras, em um endereço mais abaixo. Isso não é por acaso, as variáveis Y e Z são variáveis de escopo (no caso escopo de main) e ficam armazenadas na **stack**, um pouco acima.

Stack

A **stack,** também conhecida como **pilha**, é uma porção importante da memória. Além das variáveis de escopo, esta região também armazena todas as chamadas de funções.



Aparentemente, a stack é uma porção de memória que cresce conforme a aplicação está rodando, mas isso não é verdade. A porção de stack é reservada logo no início da aplicação. Seu programa não tem direito de consumir mais memória de stack do que a pré-definida, lembre-se que há sempre outros programas rodando em seu computador que precisam de memória.

Se seu programa estourar este limite, ocorre o **estouro de pilha**, ou **stackoverflow**. Um erro como esse corrompe a memória de seu sistema causando erros graves. Imagine um programa escrito em um microcontrolador que controla o piloto automático de um carro por exemplo, se este programa sofrer um stackoverflow, gerará um erro que pode colocar vidas em risco.

Recursão

Chamadas recursivas é a ideia de chamar uma função dentro dela mesma várias vezes, causando um consumo acelerado de stack. O conceito é permitido dentro da programação mas deve ser feito com cuidado. A função fatorial abaixo chama ela mesma em seu escopo até que o fator seja 1.

```
// Função fatorial utilizando a recursão
#include <stdio.h>

int fatorial(int n);

int main()
{
  int res;
  res = fatorial(5);
  printf("%d", res);
}

int fatorial(int n) {
  int res;

  if (n == 1) return 1;
  res = fatorial(n - 1) * n; // Chamada recursiva

  return res;
}
```

Caso ocorra o estouro da pilha na execução de seu programa em algum sistema operacional como windows, mac ou linux, o sistema automaticamente encerra seu programa. Já em sistemas mais simples, envolvendo microcontroladores, não há essa proteção, mas o compilador pode avisar o estouro da pilha.

Heap

O heap (amontoado) é uma porção de memória mais flexível que a stack, pois permite manipulação. O programa decide quando, onde e por quanto tempo deseja alocar memória. Resumindo, se trata de uma grande porção de memória que podemos utilizar da forma que quisermos.

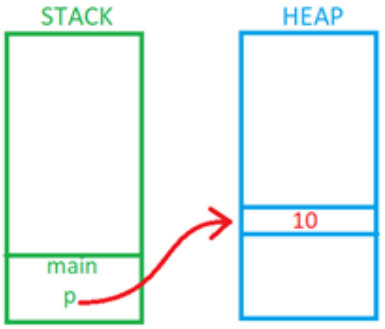
É no heap que conhecemos o conceito de alocação dinâmica de memória. Para alocar memória dinamicamente no programa, utilizaremos 4 funções

- malloc
- calloc
- realloc

- free

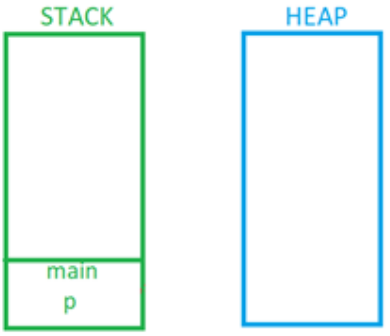
Malloc recebe um tamanho de memória que se deseja alocar e retorna um ponteiro para esta memória. Abaixo, por exemplo, estamos alocando o espaço de um inteiro no heap.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a; // stack
    int *p; // stack

    p = malloc(sizeof(int)*4);
    if(p == NULL) return 1;

    *p = 10;
```



Mas malloc não libera o endereço automaticamente, criando um problema chamado vazamento de memória, ou **Memory Leak**. Para resolver este problema, utilizamos a função free.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a; // stack
    int *p; // stack

    p = malloc(sizeof(int)*4);
    if(p == NULL) return 1;

    *p = 10;

    free(p);

    p = malloc(sizeof(int)*4);
    if(p == NULL) return 1;

    *p = 20;

    free(p);

    return 0;
}
```



Utilizando o comando de linux `valgrind --leak-check=full`, veja o vazamento de memória criado ao executar um comando que utiliza o heap sem free.

Alocar arrays no heap torna a stack mais leve, com menos comsumo de memória.

É importante ressaltar que a memória total do sistema não é literalmente organizada da forma como vimos nos sildes. Ela é cheia de buracos e pode ser segmentada em várias posições, mas isso não altera sua arquitetura de alocação. Sempre existirá uma stack, um heap e uma memória de programa.

Uma variável criada por uma função e tendo seu endereço retornado causa um erro no compilador. Aquele endereço foi criado em uma porção da stack que foi desalocada no momento que a execução do programa retornou à main. Isso causa um problema de “não garantia” de que aquele endereço vai preservar seu valor, já que pode ser alocado por outra variável.

É por isso que não retornamos arrays. Se você tentar retornar um array, estará retornando o endereço para o seu primeiro elemento. E isso nos coloca no mesmo problema.

Malloc aloca de forma segura um endereço fora do escopo, endereço este que estará sempre alocado a não ser que você libere ele. Por isso pode ser retornado tranquilamente.

Jeito errado:

```
int * aloca_array(int size){
    int p[size];
    return &p;
}
```

Jeito certo

```
char * retorna_array(int size){
    char *p = malloc(size);
    return p;
}
```

Calloc e realloc

O que é o calloc: `calloc` é uma função em C usada para alocar um bloco de memória contígua e inicializá-lo com zeros. A função recebe dois argumentos: o número de elementos alocados e o tamanho de cada elemento. O resultado é um ponteiro para a memória alocada, ou NULL se a alocação falhar.

```
int *array = (int *)calloc(10, sizeof(int));
```

Neste exemplo, `calloc` aloca espaço para um array de 10 inteiros e inicializa todos os elementos com zero.

O que é o realloc: `realloc` é uma função em C usada para alterar o tamanho de um bloco de memória previamente alocado dinamicamente, possivelmente redimensionando-o para um tamanho maior ou menor. A função recebe um ponteiro para o bloco de memória original, o novo tamanho desejado e retorna um novo ponteiro para o bloco realocado. É importante notar que o conteúdo do bloco original é preservado o máximo possível durante a realocação.

```
int *array = (int *)malloc(5 * sizeof(int)); // Aloca espaço para 5 inteiros
// ...
array = (int *)realloc(array, 10 * sizeof(int)); // Redimensiona para 10 inteiros
```

Neste exemplo, `realloc` é usado para redimensionar o bloco de memória original alocado com `malloc` para acomodar 10 inteiros em vez de 5.