ALOCAÇÃO DINÂMICA

Todas as operações que fizemos até agora com ponteiros ocorreram na `stack`. Nessa parte, os ponteiros não brilham tanto. Sua real utilidade está quando mexemos na `heap`.
Como vimos, ponteiros guardam endereços de memória e a `heap` é uma região da memória onde os dados persistem até que alguém vá e diga que não são mais necessários... que tal se os ponteiros criados na `stack` ficassem responsáveis por guardar os endereços desses dados na `heap` e fossem usados pra acessar essa região de forma fácil? Essa prática é chamada de **alocação dinâmica de memória**.

A alocação dinâmica tem algumas vantagens por usar a `heap`:

- Por armazenar dados na `heap`, permite um volume de informações muito maior do que as suportadas na `stack`;
- Pode persistir dados através de funções;
- Pode ter vetores como retornos de funções!

Vamos ver agora como fazer pra utilizarmos alocação dinâmica no nosso código.

PRINCIPAIS FUNÇÕES

O C++, por ser uma linguagem derivada de C, herdou diversos de seus mecanismos para gerenciar memória dinamicamente. Todas as funções de C estão disponíveis na biblioteca `<cstdlib>`, mas também vou mostrar posteriormente como fazer a alocação pelos mecanismos nativos e mais modernos do C++.

MALLOC

`malloc` é a função mais básica que aloca memória na `heap`. Ela recebe como parâmetro a quantidade de bytes a ser alocada e retorna ou 1) o endereço de memória de onde foi alocada ou 2) um ponteiro nulo, caso não haja espaço disponível na `heap` para alocar. Exemplo de uso:

```
int *vetor1 = (int *) malloc(sizeof(int) * 100);
```

Na linha acima, estamos fazendo exatamente a mesma coisa que `int vetor1[100]` com um detalhe: estamos alocando o espaço na `heap`, não na `stack`.

Exemplo mais comum: alocar um vetor de tamanho variável.

```
int n;
cin >> n;
int *v = (int *) malloc(sizeof(int) * n);
```

Isso é equivalente sintaticamente a `int v[n]`, mas alocamos na `heap` o espaço.

Uma utilização muito comum é na criação de strings!

```
char *str = (char *) malloc(sizeof(char) * 100);
```

Aqui, estamos aloacndo um espaço na heap para comportar até 100 caracteres (100 bytes).

Note o uso do `sizeof`!
Essa função nativa do C/C++ recebe um tipo e retorna a quantidade de bytes que ela ocupa na memória. Por exemplo, `sizeof(char) = 1`, `sizeof(int) = 4` e outros. É muito usada na chamada das funções de alocação, onde o programador informa a quantidade de elementos a serem alocados e multiplica pela quantidade de bytes do tipo desejado.

É fundamental usarmos o `sizeof` e não calcularmos diretamente a quantidade de bytes alocados, pois isso é uma informação que pode variar de arquitetura para arquitetura (um `int` deve ter no mínimo 2 bytes, mas em muitos computadores tem 4). Para realmente conseguirmos fixar uma quantidade de bytes pro dado, poderiamos usar tipos fornecidos por `<cstdint>`, como `int8_t`, `int32_t`, etc..

CALLOC

`calloc` tem exatamente a mesma finalidade do `malloc` com um detalhe: o `malloc` aloca memória mas não inicializa os dados, ou seja, os dados no espaço são valores-lixo, enquanto o `calloc` aloca E inicializa todas as posições com 0. Ela recebe como parâmetro a) o número de elementos a ser alocado e b) o tamanho em bytes de cada elemento (use o `sizeof`!) e retorna ou 1) o endereço de memória de onde foi alocada ou 2) um ponteiro nulo, caso não haja espaço disponível na `heap` para alocar.

Exemplo de uso:

```
int *vetor2 = (int *) calloc(20, sizeof(int));
```

Aqui, alocamos um vetor de até 20 inteiros e inicializamos todos as posições com 0.

BOAS PRÁTICAS COM ALLOC

Como as funções de alocação retornam sempre `void *`, é fundamental que façamos o cast para o tipo do nosso ponteiro, como visto anteriormente. O C++ nos obriga a fazer isso, enquanto o C não.
Além disso, nem sempre as funções de alocação vão conseguir reservar espaço na memória devido à falta de espaço mesmo. Por isso, após usarmos uma função de alocação, é importante verificar se o ponteiro é nulo ou não, antes de prosseguirmos!

Exemplo de uso:

```
int *v = (int *) malloc(sizeof(int) * 20);
if (v == nullptr) {
  return 1; // Para o programa, não há mais espaço na heap
}
// Continua normalmente.
```

`nullptr` é uma keyword do C++ que indica um ponteiro nulo (bem expressiva).

REALLOC

`realloc` tem o objetivo de permitir que um vetor seja redimensionado mesmo após a sua declaração. Ela recebe como parâmetro a) o ponteiro para o endereço-base do vetor e b) a nova quantidade em bytes do vetor e retorna ou 1) o endereço de memória de onde foi alocada ou 2) um ponteiro nulo, caso não haja espaço disponível na `heap` para alocar.

A função `realloc` funcionará da seguinte forma: ela buscará uma área de memória com o novo tamanho indicado, alocará essa área, copiará os valores existentes previamente para essa nova área; e por fim liberará a memória utilizada anteriormente. Exemplo de uso:

```
char *p1 = (char*) malloc (sizeof(char) * 150);
// operações
p1 = realloc(p1, sizeof(char) * 300);
```

Aqui, primeiro criamos uma string de até 150 caracteres e depois, por alguma necessidade, aumentamos o tamanho dela para 300.

Devemos ter um cuidado, porém. Veja o exemplo:

```
int *p1, *p2;

p1 = (int *) malloc(sizeof(int) * 10);

p2 = p1;

p2 = (int *) realloc(p2, sizeof(int) * 100);
```

O que há de errado?

Como o `realloc` desaloca a memória previamente ocupada e aloca em uma nova região, o ponteiro `p1` aponta para uma região desalocada de memória. Logo, ao usar o `p1`, qualquer valor pode estar ali dentro e explodir seu programa. Logo, deveriamos desalocar o `p1` e não usá-lo mais.
"Mas como faz pra desalocar memória? Já sei demais como alocar..."

FREE

`free` é a função de desalocar memória na `heap`! Ela é bem simples: recebe somente o endereço de memória a ser desalocado e não retorna nada. Exemplo de uso:

```
int *p = (int *) malloc(sizeof(int) * 10);
// diversas operações com o p depois...
free(p);
// Continua o programa com esse espaço disponível para uso :)
```

Simples demais, né?

Cuidado: você deve usar o `free` somente com espaços de memória que estão na `heap`, ou seja, que foram alocados com `malloc`/`calloc`; caso passe um endereço na `stack`, talvez seu computador exploda...

PONTEIRO COMO RETORNO

Anteriormente sem os ponteiros, quando precisávamos preencher um vetor dentro de uma função, nós o passávamos por referência, lembra? Isso era necessário pois não podemos declarar uma função como `int[] gerar_numeros()` no C++.

Para eliminarmos a necessidade de passar um parâmetro adicional como referência, basta retornarmos então o ponteiro para o início do vetor que foi criado na função! Exemplo de uso:

```
int * gerar_numeros(int tamanho) {
  int *v = calloc(tamanho, sizeof(int));
  for (int i = 0; i < tamanho; i++) {
    v[i] = gerar_numero(); // Suponha que essa função existe em algum lug
  }
  return v; // Permitido!
}

int main() {
  int *numeros = gerar_numeros(10);
}
```

Pergunta: o que aconteceria se eu não tivesse usado o `calloc` e sim somente `int v[tamanho]`?

EXERCÍCIOS

PROGRAMA SIMPLES

- Leia o nome do usuário e aloque dinamicamente em um espaço de até 20 caracteres;
- Leia um inteiro n, depois leia e aloque um vetor de até n strings, onde cada string representa uma música favorita do usuário;
- Realoque o tamanho do vetor para que tenha metade do tamanho do vetor original;
- Por fim, imprima o nome do usuário e as músicas presentes no vetor após a realocação.

POSIÇÃO DA STRING ENCONTRADA

Implemente um programa que leia uma frase do usuário e, depois, uma palavra a ser encontrada dentro da frase. Retorne o índice em que a palavra se encontra presente na frase, ou uma mensagem de erro avisando que a palavra não faz parte da frase.
Dica: use a função `strstr` da `cstring` (**https://cplusplus.com/reference/cstring/strstr/**). Veja o que a função retorna e como, por meio desse retorno, você pode calcular a posição em que a palavra inicia!

GERENCIADOR DE MEMÓRIA PARA STRINGS

Implemente um programa que simule um gerenciador de memória para strings. O programa deve ter como funcionalidades:

1 - INICIALIZAR O VETOR DE STRINGS

- O usuário deve informar quantas strings deseja armazenar.
- Para cada string, o programa deve alocar dinamicamente espaço na memória para armazená-la, até um limite de 200 caracteres!
- O usuário deve preencher cada uma das strings pertencente ao vetor.

2 - CONCATENAR STRINGS

- O programa deve permitir que o usuário escolha duas strings do vetor e as concatene em uma nova string alocada dinamicamente. A nova string resultante da concatenação deve ser armazenada em uma **nova** posição do vetor.
- Se o vetor de strings estiver vazio ou for nulo, retorne um erro e peça ao usuário para primeiro inicializar.

3 - BUSCAR SUBSTRING

- O programa deve permitir que o usuário busque uma substring em todas as strings do vetor. Para cada string que contiver a substring, o programa deve exibir a string e a posição onde a substring foi encontrada.
- Se o vetor de strings estiver vazio ou for nulo, retorne um erro e peça ao usuário para primeiro inicializar.

4 - RECRIAR VETOR

- O programa deve permitir que o usuário exclua o vetor atual e inicie outro do zero, seguindo os passos da operação 1.
- Se o vetor de strings estiver vazio ou for nulo, retorne um erro e peça ao usuário para primeiro inicializar.

5 - SAIR

- O programa deve permitir que o usuário interrompa sua execução, liberando a memória alocada até o momento.

Speaker notes