MINISTÉRIO DA EDUCAÇÃO
Universidade Federal de Ouro Preto
Departamento de Computação e Sistemas
Campus João Monlevade

ALOCAÇÃO DINÂMICA DE MEMÓRIA

Curso: Engenharia de Computação — 1º semestre de 2023
Disciplina: Algoritmos e Estruturas de Dados I — Data: 21/06/23
Professor: Alexandre Magno de Sousa — **Data de Entrega: 05/07/23**

1 Problemas com Ponteiros

1.1 Ponteiros Não Inicializados

Ponteiros não inicializados são ponteiros que não foram referenciados para uma variável ou não receberam uma célula criada via alocação dinâmica de memória (por meio das funções `malloc` ou `calloc`).

O uso de ponteiros não inicializados gera um erro de memória durante a execução. Isso ocorre porque durante a tentativa de acessar o conteúdo do endereço apontado pelo ponteiro, esse endereço não existe, pois não há como acessar um endereço que não existe.

Por exemplo, seja `ptr` um ponteiro para inteiro, logo deve ser inicializado por uma das seguintes formas:

(a) `ptr = &nome_de_uma_variavel`, neste caso, o ponteiro recebe o endereço de memória de uma variável;

(b) `ptr = (int*) malloc(sizeof(int))`, nesta opção, o ponteiro recebe o endereço de memória de uma célula de memória criada via alocação dinâmica de memória por meio da função `malloc`;

(c) `ptr = (int*) calloc(n, sizeof(int))`, aqui, o ponteiro recebe o endereço de memória de uma célula de memória criada via alocação dinâmica de memória por meio da função `calloc`, ressalta-se que, para essas função, `n` é uma variável que indica o número de células que serão criadas.

Ao trabalhar com um ponteiro via alocação dinâmica de memória, ao final do programa, é necessário liberar a região de memória criada pela alocação por meio da função `free(nome_do_ponteiro)`, essa função recebe como parâmetro o ponteiro e por meio dele libera a região de memória criada.

NOTA: também é possível atribuir um endereço nulo ao ponteiro por meio da palavra reservada `NULL` como se segue: `ponteiro = NULL`. O endereço nulo não é um endereço válido, serve apenas para "anular" um ponteiro. Dessa forma, tentar utilizar um ponteiro que acabou de ser anulado gera um erro de memória.

1.2 Ponteiros Soltos

Um ponteiro solto é um ponteiro que contém um endereço de uma célula de memória (região de memória criada via alocação dinâmica de memória – `malloc` ou `calloc`) que já foi liberada.

Seja o seguinte trecho de código:

```
1. int *ptr1, *ptr2, numero;
2. ptr1 = (int*) malloc(sizeof(int)); // alocação dinâmica
3. *ptr1 = 7;
4. ptr2 = ptr1;
5. free(ptr1);
6. numero = *ptr2;
```

No trecho de código apresentado, um erro aconteceria na linha 6 quando o conteúdo apontado pelo endereço no ponteiro `ptr2` é acessado para ser atribuído à variável número. Analisando o código, veja que são declarados dois ponteiros para inteiro na linha 1, `ptr1` e `ptr2`, além da variável inteira numero.

Na linha 2, por meio do uso da função malloc, é alocada uma célula de memória para o ponteiro `ptr1`. Em seguida, na linha 3, é atribuído o valor 7 ao conteúdo do endereço de memória apontado pelo ponteiro `ptr1`.

Depois, na linha 4 o ponteiro `ptr2` recebe o endereço de memória contido em `ptr1`. Porém, na linha 5 a célula de memória que estava alocada para `ptr1` é liberada da memória. Lembre-se que os ponteiros `ptr1` e `ptr2` referenciavam a mesma célula de memória, depois que a célula de memória foi liberada os dois ponteiros não mais referenciam um endereço de memória. Isso faz com que qualquer tentativa de uso dos ponteiros depois da liberação de memória realizada na linha 5 gere um erro de memória, o que impossibilita a execução do código.

1.3 Células de Memória Perdidas

É uma célula de memória que foi criada via alocação dinâmica de memória e que seu endereço não está mais acessível. Essa célula é considerada lixo de memória, pois nenhum ponteiro pode mais acessá-la na memória.

Isso acontece porque, uma vez que uma célula de memória é criada via alocação dinâmica, ela continuará a existir na memória até que seja liberada com a função `free`. Neste caso, se um ponteiro que continha seu endereço a perder, não será possível liberá-la da memória, a não ser que o computador seja reiniciado para que o conteúdo da memória RAM seja apagado.

Um exemplo de célula perdida é apresentado a seguir:

```
1. int numero, *ptr;
2. ptr = (int*) malloc(sizeof(int)); //alocação dinâmica
3. numero = 51;
4. *ptr = 13;
5. ptr = &numero;
```

Nesse trecho de código não há nenhum erro de sintaxe, de memória ou de execução aparente. Na verdade o código executaria normalmente. Entretanto, um problema ocorre

quando na linha 5 o ponteiro `ptr` recebe o endereço de memória da variável `numero`, o que faz com que o endereço da célula de memória alocada seja sobrescrito pela atribuição do endereço de memória da variável `numero`. A partir daí a célula criada fica perdida na memória RAM e não é mais possível acessá-la. Nesse caso, somente é possível liberar essa célula de memória perdida reiniciando o computador.

2 Ponteiros, Estruturas e Alocação Dinâmica

Leia, estude e execute o programa descrito no trecho de código a seguir para compreender como ele funciona. Como dica de estudo, represente por meio de desenhos as alocações dinâmicas de memória para melhor entendimento dos ponteiros.

```
# include <stdio.h>
# include <stdlib.h>
# define TAM 10 // Definição de uma constante
typedef struct livro{
    char titulo[100];
    int ISBN;
    int volume;
    int edicao;
    char local_publicacao[100];
    char editora[100];
    int ano;
}TLivro;

int main(){
 TLivro *ptr; //Declaração um ponteiro para o tipo TipoProduto
 ptr = (TLivro*) calloc(TAM,sizeof(TLivro));//Alocação de memória para 10 células
 int i;
 // Realiza a leitura de dados para 10 células por meio de aritmética de ponteiros
 for(i = 0; i < TAM; i++){
    printf(''Digite o titulo do livro:'');
    fflush(stdin);
    fgets((ptr + i)->titulo, 100, stdin); // ou (*(ptr + i)).titulo
    printf(''Digite o ISBN do livro: '');
    fflush(stdin);
    scanf(''%d'', &(ptr + i)->ISBN); // ou (*(p + i)).ISBN
    printf(''Digite o volume do livro: '');
    fflush(stdin);
    scanf(''%f'', &(ptr + i)->volume); // ou (*(p + i)).volume
    printf(''Digite a edicao do livro: '');
    fflush(stdin);
    scanf(''%f'', &(ptr + i)->edicao); // ou (*(p + i)).edicao
    printf(''Digite o local de publicacao do livro:'');
    fflush(stdin);
    fgets((ptr + i)->local_publicacao,100,stdin);//ou (*(ptr+i)).local_publicacao
    printf(''Digite a editora do livro:'');
    fflush(stdin);
    fgets((ptr + i)->editora, 100, stdin); // ou (*(ptr + i)).editora
    printf(''Digite o ano de publicacao do livro: '');
    fflush(stdin);
    scanf(''%f'', &(ptr + i)->ano); // ou (*(p + i)).ano
 }
 //Exibe os dados armazenados
 for(i = 0; i < TAM; i++){
    printf(''TITULO: %s'', (ptr + i)->nome);
    printf(''ISBN: %d'',(ptr + i)->quantidade);
    printf(''VOLUME: %df'',(ptr + i)->preco);
    printf(''EDICAO: %df'',(ptr + i)->preco);
    printf(''LOCAL DE PUBLICACAO: %s'',(ptr + i)->preco);
    printf(''EDITORA: %s'',(ptr + i)->preco);
    printf(''ANO: t%d'',(ptr + i)->preco);
 }
```

```
 //Liberando as três células de memória alocadas
 free(ptr);
 system(''PAUSE'');
 return 0;
}
```

3 Exercícios de Fixação

1. Utilizando aritmética de ponteiros, crie as funções:

   (a) `StrCat(char *destino, char *origem);`

   (b) `StrLen(char *string);`

   (c) `StrCmp(char *string1, char *string2).`

   Essas funções devem funcionar exatamente como as funções `strlen`, `strcat` e `strcmp` da biblioteca `<string.h>`.

2. Desenvolva a função

   ```
   int StrEnd(char *s, char *t)
   ```

   que retorna 1 (um) se a cadeia de caracteres `t` ocorrer no final da cadeia `s`, e 0 (zero) caso contrário.

3. Seja o seguinte trecho de código em C:

   ```
   typedef struct adress {
       char *avenue;
       int number;
       char *country;
   } TAdress;
   typedef struct record {
       char *lastName;
       char  *firstName;
       char *middleName;
       TAdress adr;
   } TNameAndAdress;
   typedef struct recordAll {
       TNameAndAdress name;
       int  employeeNumber;
       float hoursWorked;
   } TInputRecord;
   TInputRecord *ptrDataInput; // declaração do ponteiro
   ```

   Responda as seguintes questões acerca do registro e da variável declarada:

   (a) Escreva a linha de instrução para alocar uma célula de memória ao ponteiro `ptrDataInput` por meio da função `calloc`.

   (b) Escreva a linha de instrução para alocar cem células de memória para o campo `avenue`, o que equivale a uma string de 100 caracteres.

(c) Escreva a linha de instrução para alocar cinquenta células de memória para o campo `country`.

(d) Escreva a linha de instrução para alocar dez células de memória para o campo `lastName`.

(e) Escreva a linha de instrução para alocar cem células de memória para o campo `firstName`.

(f) Escreva a linha de instrução para alocar cinquenta células de memória para o campo `middleName`.

(g) Escreva o código de acesso ao campo `avenue`.

(h) Escreva a instrução de leitura de uma string para o campo `avenue`.

(i) Escreva a instrução de leitura de uma string para o campo `firstName`.

(j) Escreva as linhas de instrução para imprimir na tela os campos `firstName` e `avenue`.

(k) Escreva a linha de instrução para liberar a memória do campo ponteiro `firstName` e do campo `avenue`.

(l) Escreva a linha de instrução para liberar a memória do ponteiro `ptrDataInput`.

4. Seja o trecho de código a seguir:

```
1.  # include <stdio.h>
2.  # include <stdlib.h>

3.  typedef struct _record_{
4.      float account;
5.      int count;
6.      char character;
7.  }TStore;

8.  TStore vp;

9.  int main( ) {
10.     TStore *p, *pt, *ptr;
11.     p = NULL;
12.     p =(TStore*) calloc(1, sizeof(TStore));
13.     p->account = 12.0F;
14.     pt = p;
15.     vp = *pt;
16.     free(p);
17.     p =(TStore*) malloc(sizeof(TStore));
18.     p = &vp;
19.     (*ptr).account = 0.988f;
20.     return 0;
21. }
```

Faça o que se pede:

(a) Represente por meio de desenhos as linhas que realizam alocação dinâmica de memória.

(b) Identifique os possíveis problemas quanto ao uso de ponteiros, caso existam, apresente o número da linha para cada problema.