



LINGUAGEM DE PROGRAMAÇÃO

AULA 4



Prof. Sandro de Araújo



CONVERSA INICIAL

Esta aula teve como base os livros: *Lógica de programação e estruturas de dados*, *Lógica de programação algorítmica* e *Treinamento em Linguagem C*. Em caso de dúvidas ou se desejar aprofundamento, consulte-os em nossa Biblioteca Virtual Pearson.

O objetivo desta aula é conhecer os principais conceitos e aplicações de ponteiros em struct, struct de ponteiros, struct com funções e alocação dinâmica de memória com as funções `calloc()`, `free()`, `malloc()` e `realloc()`, na linguagem C, para resolver problemas computacionais.

TEMA 1 – PONTEIRO: STRUCT

Na linguagem C, uma struct é uma coleção de variáveis referenciada pelo mesmo nome, conhecida também como tipo de dado agregado. Uma vez que variáveis do tipo estrutura são tratadas exatamente da mesma forma que variáveis de tipos básicos, é possível definir variáveis do tipo ponteiro para estruturas (Mizrahi, 2008), conforme a seguinte sintaxe:

```
1. struct <nome_da_struct> *<nome_do_ponteiro>;
```

Um componente ou membro de uma estrutura é uma variável, e pode ser usado para guardar o endereço de outra variável. Um exemplo disso são as listas ligadas, que têm em cada nó um ponteiro para o próximo nó.

Para passar um endereço de uma variável, basta colocar o símbolo '*' antes da definição do seu ponteiro e o operador '&' – operador unário que retorna o endereço na memória de seu operando, na chamada da struct. O uso do asterisco indica o componente da struct pode ser modificado diretamente na memória (Mizrahi, 2008).

Uma vez que variáveis do tipo struct são tratadas exatamente da mesma forma que variáveis simples, é possível definir variáveis do tipo ponteiro para struct, conforme mostrado no exemplo abaixo:

```
1. struct calendario{  
2. int dia;  
3. int mes;  
4. int ano;
```



5. `}; struct calendario agora, *depois; // declara o ponteiro 'depois'`
6. `depois = &agora; // Coloca o endereço no ponteiro 'depois'`

Os componentes individuais de uma struct podem ser acessados como qualquer variável, desde que se use o operador '.' ponto, entre o nome da estrutura (instância) e o nome do componente. Portanto, para acessar os componentes da **struct calendario**, basta usar o ponteiro `*depois`, entre parênteses, junto com a variável de referência da struct, separado por um '.' (Mizrahi, 2008):

1. `(*depois).dia = 28;`
2. `(*depois).mes = 09;`
3. `(*depois).ano = 2018;`

Os componentes da struct podem ser impressos com a mesma sintaxe, conforme o exemplo:

1. `printf("%i/%i/%i\n\n", (*depois).dia, (*depois).mes, (*depois).ano);`

A linguagem de programação C, também usa uma notação simplificada para substituir a forma do exemplo acima, substituímos **(*depois).dia** por **depois->dia**. O operador '->' (seta) é usado para substituir o operador '.' (ponto), eliminando também a necessidade de usar o ponteiro entre os parênteses. A Figura 1 apresenta um algoritmo com a junção dos exemplos acima apresentados.




Figura 1 – Ponteiro para uma Struct

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(){
5
6  struct calendario{
7      int dia;
8      int mes;
9      int ano;
10 }; struct calendario agora, *depois;
11
12 depois = &agora;
13
14 /* (*depois).dia = 28;
15    (*depois).mes = 09;
16    (*depois).ano = 2018;
17    */
18
19 depois->dia = 28;
20 depois->mes = 9;
21 depois->ano = 2018;
22
23 //printf("%i/%i/%i\n\n", (*depois).dia, (*depois).mes, (*depois).ano);
24
25 printf("%i/%i/%i\n\n", depois->dia, depois->mes, depois->ano);
26
27 system("pause");
28 return 0;
29 }
```

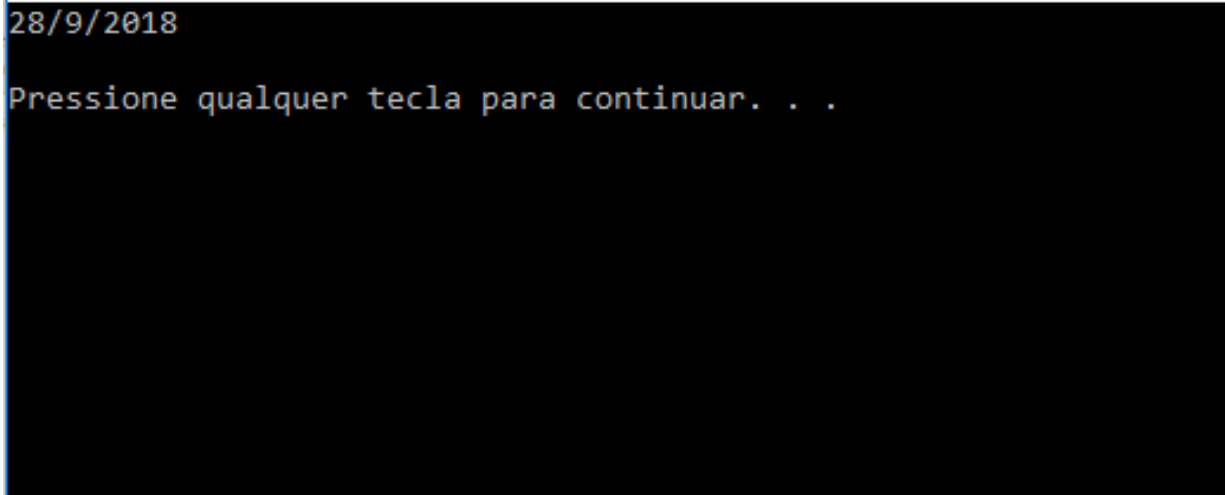
O operador seta ‘->’ facilita, resume e evita erros na compilação no acesso aos componentes da struct. Pois, para essa lógica, a expressão “*depois.dia”, sem o uso de parênteses, apresenta um erro sintático, porque o compilador dá prioridade de execução ao operador ‘.’. O compilador primeiro executa o operador ‘.’ e depois o operador ‘*’. O algoritmo da Figura 1 apresenta os seguintes passos:

1. Entre as linhas seis e dez, criamos uma struct “**calendário**” com três componentes do tipo inteiro “**dia**”, “**mês**” e “**ano**”;
2. Na linha seis, criamos a variável “agora”, juntamente com o ponteiro “*depois”;
3. Na linha 12, é atribuído o endereço de endereço de “agora” para o ponteiro “depois”;
4. Nas linhas 19, 20 e 21, os componentes da struct são instanciados com o uso de ponteiros;

- 
5. Na linha 25, os componentes são impressos na tela do usuário e os acessos aos dados se dá por meio dos ponteiros.

A Figura 2 mostra a saída do algoritmo do algoritmo mostrado na Figura 1, após a sua execução.

Figura 2 – Saída do algoritmo



```
28/9/2018
Pressione qualquer tecla para continuar. . .
```

TEMA 2 – STRUCT DE PONTEIROS

Ponteiros também podem ser definidos como componentes de estruturas. Basta colocar o operador “*” asterisco antes dos componentes de uma struct (Mizrahi, 2008):

1. struct calendario{ //Struct “calendário”
2. int *dia; //Ponteiro “dia”
3. int *mes; //Ponteiro “mês”
4. int *ano; //Ponteiro “ano”
5. }; struct calendario atual; //Variável de referência “atual”

As instruções acima declaram uma struct com o nome **calendario** que apresenta três componentes ponteiros. Além disso, também foi declarada uma variável com o nome **atual** para referenciar a struct **calendario**.

As inicializações dos ponteiros podem ser declaradas conforme as instruções mostradas abaixo:

1. atual.dia = &diaSetembro;
2. atual.mes = &mesSetembro;



3. `atual.ano = &anoSetembro;`

Para imprimir as variáveis, usando os componentes da struct, que por sua vez são os ponteiros que as referenciam, procede-se conforme o exemplo:

1. `printf("Dia = %i\n", *atual.dia);`

A Figura 3 mostra um algoritmo com a junção dos exemplos acima apresentados.

Figura 3 – Struct de ponteiros

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(){
5
6      struct calendario{
7          int *dia;
8          int *mes;
9          int *ano;
10     }; struct calendario atual;
11
12     int diaSetembro = 28;
13     int mesSetembro = 9;
14     int anoSetembro = 2018;
15
16     atual.dia = &diaSetembro; // Atribui o endereço de diaSetembro
17     atual.mes = &mesSetembro; // Atribui o endereço de mesSetembro
18     atual.ano = &anoSetembro; // Atribui o endereço de anoSetembro
19
20     printf("Endereco da variavel diaSetembro      = %p\n", &diaSetembro);
21     printf("Ponteiro *dia aponta para o endereco = %p\n\n", atual.dia);
22     printf("Endereco da variavel mesSetembro     = %p\n", &mesSetembro);
23     printf("Ponteiro *mes aponta para o endereco = %p\n\n", atual.mes);
24     printf("Endereco da variavel anoSetembro     = %p\n", &anoSetembro);
25     printf("Ponteiro *ano aponta para o endereco = %p\n\n", atual.ano);
26
27     printf("Dia: %d Mes: %d Ano: %d\n\n", *atual.dia,*atual.mes,*atual.ano);
28
29     system("pause");
30     return 0;
31 }
```

O algoritmo da Figura 3 apresenta os seguintes passos:

1. Entre as linhas seis e dez. criamos uma struct “**calendário**” com três ponteiros “***dia**”, “***mês**” e “***ano**”;
2. Nas linhas 12, 13 e 14, criamos três variáveis com suas instancias;
3. Nas linhas 16, 17 e 18, atribui-se os endereços das variáveis para os ponteiros:



4. Nas linhas 20, 21, 22, 23, 24 e 25, são impressos os endereços das variáveis e o apontamento dos ponteiros;
5. Na linha 27, são impressos os conteúdos das variáveis usando os ponteiros da struct.

A Figura 4 mostra a saída do algoritmo apresentado na Figura 3, após a sua execução:

Figura 4 – Saída do algoritmo

```
Endereco da variavel diaSetembro      = 000000000062FE2C
Ponteiro *dia aponta para o endereco = 000000000062FE2C

Endereco da variavel mesSetembro      = 000000000062FE28
Ponteiro *mes aponta para o endereco = 000000000062FE28

Endereco da variavel anoSetembro      = 000000000062FE24
Ponteiro *ano aponta para o endereco = 000000000062FE24

Dia: 28 Mes: 9 Ano: 2018

Pressione qualquer tecla para continuar. . . _
```

TEMA 3 – STRUCT: FUNÇÃO

Assim como uma variável, uma struct também pode ser passada como parâmetro para uma função; essa passagem é feita de duas formas: por valor e por referência (Mizrahi, 2008).

3.1 Passagem de uma struct por valor

Uma struct é tratada com uma variável comum, e a passagem por valor é feita por meio da passagem de uma cópia do seu componente para uma função. Na passagem por valor, uma cópia do componente da struct é usada e alterado dentro da função sem afetar a variável da estrutura, na memória da qual ela foi gerada (Mizrahi, 2008).

Figura 5 – Passagem de struct por valor

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct p_valor{
5      int a,b;
6  };
7
8  void imprimir_valor(int num);
9
10 int main(){
11
12     struct p_valor pont1 = {201,302};
13
14     printf("VALORES ANTES DA EXECUCAO DA FUNCAO");
15     printf("\nValor de 'a' = %d", pont1.a);
16     printf("\nValor de 'b' = %d \n\n", pont1.b);
17
18     printf("\nVALORES APOS A EXECUCAO DA FUNCAO\n");
19     imprimir_valor(pont1.a); //Faz uma cópia de a
20     imprimir_valor(pont1.b); //Faz uma cópia de b
21
22     printf("\nVALORES NA MEMORIA APOS A EXECUCAO DA FUNCAO");
23     printf("\nValor de 'a' nao sofreu alteracao na memoria = %d", pont1.a);
24     printf("\nValor de 'b' nao sofreu alteracao na memoria = %d \n\n", pont1.b);
25
26     system("pause");
27     return 0;
28 }
29
30 void imprimir_valor(int num){
31     int num1;
32     num1 = num + 3;
33     printf("Valor %d na execucao da funcao = %d\n",num, num1);
34 }
```

O algoritmo da Figura 5 apresenta os seguintes passos:

1. Entre as linhas 4 e 6, criamos uma struct “**p_valor**” com dois componentes ‘a’ e ‘b’;
2. Na linha 12, os componentes da struct são instanciados;
3. Nas linhas 15 e 16, são impressos os valores dos componentes antes da execução da função “**imprimir_valor()**”;
4. Nas linhas 19 e 20, são feitas duas chamadas para a função “**imprimir_valor**”, que traz como resultados a impressão dos valores da struct processados;
5. Nas linhas 23 e 24, são impressos os valores dos componentes depois da execução da função “**imprimir_valor()**”;
6. Entre as linhas 30 e 34, é criada a função “**imprimir_valor()**”.

A Figura 6 mostra a saída do algoritmo da Figura 5 após a sua execução.



Figura 6 – Saída do algoritmo

```
VALORES ANTES DA EXECUCAO DA FUNCAO
Valor de 'a' = 201
Valor de 'b' = 302

VALORES APOS A EXECUCAO DA FUNCAO
Valor 201 na execucao da funcao = 204
Valor 302 na execucao da funcao = 305

VALORES NA MEMORIA APOS A EXECUCAO DA FUNCAO
Valor de 'a' nao sofreu alteracao na memoria = 201
Valor de 'b' nao sofreu alteracao na memoria = 302

Pressione qualquer tecla para continuar. . . _
```

No exemplo acima, os componentes da struct foram instanciados na **linha 12** com a instrução “**struct p_valor pont1 = {201,302};**”, alterados com a função “**void imprimir_valor(int num)**”. Após a execução da função, os valores antes instanciados continuaram os mesmos, sem alterações na memória.

3.2 Passagem de uma struct por referência

Para que a passagem de um parâmetro seja por referência, basta colocar o símbolo “*” antes da sua definição dos parâmetros formais e o operador “&”¹, na chamada do parâmetro (Mizrahi, 2008). O uso do asterisco indica que esses parâmetros podem ser modificados dentro da função, ou seja, as alterações dos parâmetros sofridas dentro da função também serão sentidas fora dela. Esses efeitos não ocorrem quando os parâmetros são passados por valor (sem o uso do asterisco (*)). Veja o exemplo na Figura 7, com a passagem por referência dos valores de uma struct para uma função.

¹ O operador unário & retorna o endereço na memória de seu operando (MIZRAHI, 2008).



Figura 7 – Passagem de struct por referência

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void imprimir_soma_valor(int *num);
5
6  int main(){
7      struct p_valor{
8          int a,b;
9      }; struct p_valor x, *px;
10
11     px = &x;
12
13     px->a=201;
14     px->b=302;
15
16     printf("VALORES ANTES DA EXECUCAO DA FUNCAO\n");
17     printf("Valor de 'a'= %d \n", px->a);
18     printf("Valor de 'b'= %d \n\n", px->b);
19
20     printf("VALORES APOS DA EXECUCAO DA FUNCAO\n");
21     imprimir_soma_valor(&x.a); //Passa o endereco de 'a' para a função
22     imprimir_soma_valor(&x.b); //Passa o endereco de 'b' para a função
23
24     printf("\nVALORES NA MEMORIA APOS DA EXECUCAO DA FUNCAO");
25     printf("\nValor de 'a' foi alterado diretamente na memoria = %d \n", x.a);
26     printf("Valor de 'b' foi alterado diretamente na memoria = %d \n\n", x.b);
27     //printf("\nValor de 'a' foi alterado diretamente na memoria = %d \n", px->a);
28     //printf("Valor de 'a' foi alterado diretamente na memoria = %d \n\n", px->b);
29
30     system("pause");
31     return 0;
32 }
33 void imprimir_soma_valor(int *num){
34     *num = *num + 3;
35     printf("Valor = %d\n",*num);
36 }
```

O algoritmo da Figura 7 apresenta os seguintes passos:

1. Entre as linhas 7 e 9, criamos uma struct “**p_valor**” com dois componentes ‘a’ e ‘b’;
2. Na linha 9, criamos a variável “x”, que vai referenciar a struct “**p_valor**”, juntamente com o ponteiro “***px**”;
3. Na linha 11, o ponteiro “***px**” recebe o endereço de ‘x’;
4. Nas linhas 13 e 14, os componentes da struct são instanciados;
5. Nas linhas 17 e 18, são impressos os valores dos componentes antes da execução da função “**imprimir_soma_valor()**”;
6. Nas linhas 20 e 22, são feitas duas chamadas para a função “**imprimir_soma_valor**”; como parâmetros, são passados os endereços dos componentes da struct, que traz como resultado a impressão dos valores processados;



7. Nas linhas 25 e 26, são impressos os valores dos componentes depois da execução da função “**imprimir_soma_valor()**”;
8. Entre as linhas 33 e 36, é criada a função “**imprimir_soma_valor()**”.

A figura 8 mostra a saída do algoritmo após execução.

Figura 8 – Saída do algoritmo

```
VALORES ANTES DA EXECUCAO DA FUNCAO
Valor de 'a'= 201
Valor de 'b'= 302

VALORES APOS DA EXECUCAO DA FUNCAO
Valor = 204
Valor = 305

VALORES NA MEMORIA APOS DA EXECUCAO DA FUNCAO
Valor de 'a' foi alterado diretamente na memoria = 204
Valor de 'b' foi alterado diretamente na memoria = 305

Pressione qualquer tecla para continuar... .
```

Uma das vantagens de criar um ponteiro para uma struct é a possibilidade de passar o seu endereço como um parâmetro para uma função. Por manipular apenas o endereço do componente, a passagem por referência promove um ganho de desempenho considerável no acesso à memória.

3.3 Ponteiro do tipo void

Um ponteiro também pode ser declarado como *void*, ou seja, um ponteiro sem tipo definido. Um ponteiro *void* tem como objetivo armazenar endereço de memória (Mizrahi, 2008). Um ponteiro do tipo *void* tem a seguinte sintaxe:

1. `void *nome_do_ponteiro;`

Se em determinado momento existir a necessidade de apenas guardar um endereço, utiliza-se um ponteiro *void*, conforme mostra o algoritmo da Figura 9.



Figura 9 – Ponteiro do tipo *void*


```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int x = 3;
7      void *y = &x;
8      int *z;
9
10     z = y;
11     *z = 9;
12
13     printf("Valor de x: %d\n",x);
14     printf("Endereco de x: %p\n",&x);
15     printf("y aponta para x: %p\n",y);
16     printf("z aponta para y: %p\n\n",z);
17
18     system("pause");
19     return 0;
20 }
```

O algoritmo da Figura 9 apresenta os seguintes passos:

1. Na linha 6, declaramos a variável 'x' com a instância 3;
2. Na linha 7, criamos um ponteiro "y" do tipo *void* que vai receber o endereço de 'x';
3. Na linha 10, 'z' recebe o apontamento de 'y';
4. Na linha 11, z altera a instância de x;
5. Na linha 13, imprimimos a atual instância de 'x';
6. E nas linhas 14, 15 e 16, os endereços da variável e apontamentos, respectivamente.

A Figura 10 mostra a saída do algoritmo após execução:

Figura 10 – Saída do algoritmo



```
Valor de x: 9
Endereco de x: 000000000062FE3C
y aponta para x: 000000000062FE3C
z aponta para y: 000000000062FE3C

Pressione qualquer tecla para continuar. . .
```

TEMA 4 – ALOCAÇÃO DINÂMICA: CALLOC() E FREE()

A alocação dinâmica de memória é um mecanismo que reserva uma quantidade de memória, em região conhecida como *heap*, durante a execução de um programa (Mizrahi, 2008). Na biblioteca *stdlib* da linguagem C, temos quatro funções para trabalhar com alocação dinâmica de memória:

1. `calloc`;
2. `free`;
3. `malloc`;
4. `realloc`.

A função **`calloc()`** tem como objetivo criar um vetor com tamanho dinâmico. Essa função serve para alocar memória durante a execução do programa. Ela faz o pedido de memória ao computador e retorna um ponteiro com o endereço do início do espaço alocado. Um fato interessante dessa função é que, após alocar a memória, ela preenche com zero todos os bits (Mizrahi, 2008). Para declarar uma função `calloc()`, usa-se a seguinte sintaxe:

1. `void* calloc(unsigned int nElementos, unsigned int tElemento)`

Nela, `nElementos` é a quantidade de posições que queremos no vetor, `tElemento` é o tamanho de cada posição de memória do nosso vetor, e `unsigned int` é números inteiros sem sinais (só números inteiros positivos). Na alocação da memória, devemos considerar o tamanho do tipo alocado. Vejamos um exemplo na Figura 11 de um vetor de tamanho 40.



Figura 11 – Exemplo com a função calloc()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6
7      int *x = (int*) calloc(40,4);
8      char *y = (char*) calloc(40,1);
9
10
11     system("pause");
12     return 0;
13 }
```

A principal dificuldade no exemplo da Figura 11 está em decorar o tamanho de cada tipo enquanto estamos criando um algoritmo. Para isso, podemos usar o operador `sizeof`, conforme o exemplo da Figura 12.

Figura 12 – Exemplo com a função calloc()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /* No momento em que precisa-se alocar
7      um espaço na memória, deve-se considerar
8      o tamanho do tipo que será usado.*/
9
10     int *x = (int*) calloc(40,4);
11     char *y = (char*) calloc(40,1);
12
13     /*Solução para saber o tamanho é usar o
14     sizeof(). */
15
16     /* Nesse exemplo temos um vetor de 40 posições
17     de tamanho? */
18
19     int *x = (int*) calloc(40,sizeof(int));
20     char *y = (char*) calloc(40,sizeof(char));
21
22     system("pause");
23     return 0;
24 }
```

Caso a memória não tenha espaço suficiente para a alocação de espaço retornará NULL, vejamos um exemplo na Figura 13 com mais detalhes do uso da função `calloc()`.

Figura 13 – Implementando a função calloc()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /* Se a memória não tiver espaços suficiente
7       para a alocação, a função calloc()
8       retorna NULL*/
9
10     int *px;
11     px = (int*) calloc(4, sizeof(int));
12
13     if(px == NULL){
14         printf("ERRO! Não tem memória suficiente.");
15         exit(1); // finaliza o programa.
16     }
17
18     int i;
19     for(i=0; i<4; i++){
20         printf("Digite px[%d]: ", i);
21         scanf("%d", &px[i]);
22     }
23     printf("\n");
24
25     for(i=0; i<4; i++){
26         printf("Posicao px[%d]= %d\n", i, px[i]);
27     }
28
29     printf("\n");
30
31     system("pause");
32     return 0;
33 }
```

A função **free()** libera o espaço de memória que foi previamente alocado. Essa função recebe um ponteiro, que foi usado para receber o endereço do bloco de memória alocada, e não retorna nada (Mizrahi, 2008). Para declarar uma função **free()**, usa-se a sintaxe mostrada abaixo:

1. `void free(void *nomePonteiro);`

Se um projeto for mais simples, vai precisar liberar ao final de sua execução. É uma boa prática de programação liberar o que foi alocado antes da aplicação terminar sua execução, conforme mostra a Figura 14.

Figura 14 – Função free()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /* Se a memória não tiver espaços suficiente
7       para a alocação, a função calloc()
8       retorna NULL*/
9
10     int *px;
11     px = (int*) calloc(4, sizeof(int));
12
13     if(px == NULL){
14         printf("ERRO! Não tem memória suficiente.");
15         exit(1); // finaliza o programa.
16     }
17
18     int i;
19     for(i=0; i<4; i++){
20         printf("Digite px[%d]: ", i);
21         scanf("%d", &px[i]);
22     }
23     printf("\n");
24
25     for(i=0; i<4; i++){
26         printf("Posicao px[%d]= %d\n", i, px[i]);
27     }
28
29     printf("\n");
30
31     free(px); /* free(x) vai liberar memória onde foi
32               alocado o ponteiro x */
33     system("pause");
34     return 0;
35 }
```

Na Figura 15 temos esse mesmo exemplo, explicando cada etapa do processo no algoritmo.

Figura 15 – Função free()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /* Se a memória não tiver espaços suficiente
7       para a alocação, a função calloc()
8       retorna NULL*/
9
10     int *px;
11     px = (int*) calloc(4,sizeof(int)); } Aloca o espaço
12
13     if(px == NULL){
14         printf("ERRO! Não tem memória suficiente.");
15         exit(1); // finaliza o programa.
16     }
17
18     int i;
19     for(i=0; i<4; i++){
20         printf("Digite px[%d]: ",i);
21         scanf("%d",&px[i]);
22     }
23     printf("\n");
24
25     for(i=0; i<4; i++){
26         printf("Posicao px[%d]= %d\n", i ,px[i]);
27     }
28
29     printf("\n");
30
31     free(px); /* free(x) vai liberar memória onde foi
32               alocado o ponteiro px */ } Limpa o espaço
33     system("pause");
34     return 0;
35 }
```

TEMA 5 – ALOCAÇÃO DINÂMICA: MALLOC() E REALLOC ()

A função malloc() é bem parecida com a função calloc(). Ela também aloca um espaço de memória e retorna um ponteiro do tipo *void* para o início do espaço de memória alocado. Mas há uma grande diferença: essa função não coloca zero nos bits do espaço alocado (Mizrahi, 2008). Para declarar uma função malloc(), usa-se a sintaxe mostrada abaixo:

1. *void* malloc(unsigned int nElementos);*

A função malloc() recebe por parâmetro a quantidade de bytes que será alocada na memória, tendo também o valor NULL, caso apresente algum erro, ou o ponteiro para a primeira posição do vetor, caso tenha sucesso na locação. Vejamos um exemplo na Figura 15.

Figura 16 – Função malloc().

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /*Para criar um vetor inteiro com 25 posições
7       vamos precisar de 100 Bytes. 1 Byte = 4,
8       então 25 * 4 = 100 */
9      int *px = malloc(100);
10
11     /*Para criar um vetor de caracteres com 100 posições
12     cada char tem 1 Byte, 100 * 1 = 100 */
13     char *py = malloc(100);
14
15     system("pause");
16     return 0;
17 }
```

Do mesmo modo, fica complicado toda vez ter que ficar contando Bytes para cada tipo e para resolver esse problema implementamos o operador `sizeof`. Vejamos o exemplo da Figura 17.

Figura 17 – Função malloc() com `sizeof`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /*Para criar um vetor inteiro com 25 posições
7       vamos precisar de 100 Bytes. 1 Byte = 4,
8       então 25 * 4 = 100 */
9      int *px = malloc(100);
10
11     /*Para criar um vetor de caracteres com 100 posições
12     cada char tem 1 Byte, 100 * 1 = 100 */
13     char *py = malloc(100);
14
15     int *px = (int*) malloc(25 * sizeof(int)); // = 100
16     char *py = (char*) malloc(100 * sizeof(char)); // = 100
17
18     system("pause");
19     return 0;
20 }
```

Agora vejamos um exemplo, na Figura 17, tratando o retorno NULL da função `malloc()`.



Figura 18 – Função malloc() com tratamento do retorno NULL

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *px;
7
8      px = (int*) malloc(4 * sizeof(int));
9      if(px == NULL){
10         printf("ERRO! Não tem memória suficiente.");
11         exit(1); // finaliza o programa.
12     }
13
14     int i;
15     for(i=0; i<4; i++){
16         printf("Digite px[%d]: ", i);
17         scanf("%d", &px[i]);
18     }
19     printf("\n");
20
21     for(i=0; i<4; i++){
22         printf("Posicao px[%d]= %d\n", i, px[i]);
23     }
24
25     printf("\n");
26
27     free(px); /* free(x) vai liberar memória onde foi
28               alocado o ponteiro px */
29     system("pause");
30     return 0;
31 }
```

A função `realloc()` aloca e realoca um espaço na memória durante a execução do programa. Essa função realiza um pedido de memória e retorna um ponteiro com o endereço do início do espaço de memória alocado (Mizrahi, 2008).

Para declarar uma função `realloc()`, usa-se a sintaxe mostrada abaixo:

1. `void* realloc(void* nomePonteiro, unsigned int nElementos);`

Essa função recebe por parâmetro um ponteiro para um bloco de memória já alocada, na nossa sintaxe o `*nomeponteiro`, e a quantidade de Bytes a ser alocada, `nElementos` e retorna NULL em caso de erros ou um ponteiro para a primeira posição do vetor em caso de sucesso. Vejamos um exemplo na Figura 19 com mais detalhes do uso da função `realloc()`.



Figura 19 – Função realloc()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      //Criamos um vetor de 25 inteiros.
7      int *px = (int*) malloc(100);
8
9      /* Agora preciso almentar a memória alocada
10     para 50 inteiros (100 Bites). */
11     px = (int*) realloc(px, 200);
12
13     system("pause");
14     return 0;
15 }
```

A realocação da memória também leva em conta o tamanho do tipo. Assim como nas funções anteriores, usamos o operador *sizeof* para calcular o tamanho no momento da execução da função (Mizrahi, 2008), conforme mostrado no exemplo da Figura 20.

Figura 20 – Função realloc() com sizeof

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      // int *px = (int*) malloc(100);
7      // px = (int*) realloc(px, 200);
8
9      //Solução com sizeof().
10
11     int *px = (int*) malloc(25 * sizeof(int));
12     px = (int*) realloc(px, 50 * sizeof(int));
13
14     system("pause");
15     return 0;
16 }
```

FINALIZANDO

Nesta aula, aprendemos os principais conceitos e aplicações de ponteiros em struct, struct de ponteiros, struct com funções e alocação dinâmica de memória com as funções `calloc()`, `free()`, `malloc()` e `realloc()`, na linguagem C, para resolver problemas computacionais. Aproveite a disciplina e bons estudos!



REFERÊNCIAS

MIZRAHI, V. V. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson, 2008.