



# LINGUAGEM DE PROGRAMAÇÃO

AULA 5



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 dessa aula é conhecer os principais conceitos e aplicações de recursividade, iteração, função recursiva, função recursiva com vetor e função macro com a diretiva `#define`. Vamos também utilizar linguagem C para a criação de algoritmos.

## TEMA 1 – RECURSIVIDADE

Recursividade ou recursão é quando uma função chama ela mesma para resolver um problema. E como funciona a recursividade?

De um modo geral, a recursividade é considerada como um processo repetitivo de uma rotina (procedimento ou função), que faz uma chamada para ela mesma. Por conseguinte, se essa função realiza essa chamada inúmeras vezes, é necessário tomar muito cuidado com a quantidade de repetições no processo. Quando não controlada, será executada de forma infinita, que é o que conhecemos como *loop eterno*. Um loop eterno ou infinito ocorre quando um bloco do código repete a instrução descontroladamente, sobrecarregando a memória e ocasionando o travamento de todo o sistema.

Para evitar que um loop seja executado de uma infinitamente, é necessário definir uma condição que vai parar o processo. Para definir a condição de terminação, são necessárias análise e avaliação detalhadas do problema que será resolvido de forma recursiva, entendendo como a rotina deverá terminar.

Uma rotina recursiva pode ser escrita de duas formas:

- Recursão Direta – É uma rotina composta por um conjunto de instruções, e uma dessas instruções faz a chamada para a rotina. A rotina X chama a própria rotina X;
- Recursão Indireta – É uma rotina que contém uma chamada a outra rotina que tem uma chamada a outra rotina e assim sucessivamente. A rotina X chama uma rotina Y, que por sua vez chama X.

Uma rotina recursiva é chamada para resolver um problema. Ela sabe como resolver somente a "parte" mais simples, o "caso" mais trivial. Portanto, a



solução para um problema recursivo normalmente pode ser dividida em uma solução que é trivial, e outra solução mais geral.

Dessa forma, a recursão aplica uma técnica chamada *dividir para conquistar*. Isto é, dividir uma ou mais versões menores do mesmo problema. Exemplo: se o problema é muito grande, mesmo dividindo-o em dois, ainda continuará grande. Podemos dividir as partes em dois novamente; até chegar em algo mais simples e fácil de resolver.

No caso da recursão, o processo vai sendo dividido em partes, até chegar em uma fração que ela vai saber resolver. Dessa forma, temos:

- Caso Trivial ou Condição de Parada – O problema é facilmente resolvível e retorna um resultado; normalmente é o caso mais simples, ou básico, do problema, inclusive nesse caso não haverá a necessidade de aplicar recursão.
- Caso Geral – O problema é, em sua essência, igual ao problema original, porém deve ser uma versão mais genérica. Como esse novo problema é parecido com o original, a rotina chama uma nova cópia de si mesma para trabalhar no problema menor.

Enquanto for necessário dividir o problema em problemas menores, a rotina recursiva continuará chamando a si mesma, para continuar dividindo, até chegar no caso mais básico. Quando isso ocorre, a função para de dividir e começa a gerar os resultados. Todos os dados de todas as variáveis envolvidas na função recursiva devem ser guardados a cada chamada. Isso significa que uma pilha de chamadas da função deve ser criada.

Exemplos de recursividade em nossa vida:

- Caracóis;
- Girassóis;
- As folhas de algumas árvores;
- Dois espelhos quando apontados um para o outro.

O algoritmo passo a passo abaixo ilustra o processo recursivo descrito até aqui:

- Terminou?
  - Se sim:
    - Retorne o resultado.
  - Se não:



- Divida o problema;
- Resolva o(s) problema(s);
- Monitore os resultados na solução do problema original;
- Retorne à solução.

## TEMA 2 – RECURSÃO X ITERAÇÃO

A grande dúvida está em saber quando devemos usar recursão ou iteração em nosso algoritmo. A regra é: se dá pra resolver o problema com iteração, é possível resolver o mesmo problema com recursão. Porém, é necessário, antes de qualquer coisa, compreender bem a diferença entre recursão e iteração, para depois decidir qual delas utilizar em seu código.

A repetição é uma característica tanto da recursão quanto da iteração. A iteração utiliza a repetição em forma de laços ou estruturas de repetição (*for*, *while*, *do-while*), enquanto a recursão utiliza a repetição na forma de chamadas para ela mesma. As duas formas precisam de uma condição para terminar o ciclo repetitivo – um teste de terminação. A iteração se encerra quando a condição de teste falha e a recursão se encerra quando se alcança o caso trivial. As duas formas podem ingressar em loop infinito; no contexto da iteração, se o teste jamais se tornar falso, o laço vai se repetir eternamente, enquanto no contexto da recursão, se o problema não for reduzido de forma que se converta para o caso trivial, e não tenha a condição de parada definida, o laço vai se repetir até sobrecarregar a memória.

As funções recursivas são muito utilizadas em robótica, automação e inteligência artificial. Estruturas de Dados Dinâmicas, como Árvores, Filas, Pilhas e Listas, também fazem uso da recursão.

A partir do momento em que fique clara a diferença e o emprego dessas duas formas de resolver problemas algorítmicos, teremos mais clareza na hora de decidir qual melhor se enquadra na lógica que está sendo desenvolvida. Portanto, analise o tempo computacional que será gasto com uma função recursiva e veja se vale ou não a pena – se o gasto for muito alto, implemente a mesma função de forma iterativa.



### TEMA 3 – FUNÇÃO RECURSIVA

As funções recursivas geralmente tornam o programa mais legível. Elas são definidas como na Matemática para evitar a retribuição de valores a variáveis. Essas funções podem substituir trechos de código que envolvem laços de repetição, como os laços de repetições *while* e *for*.

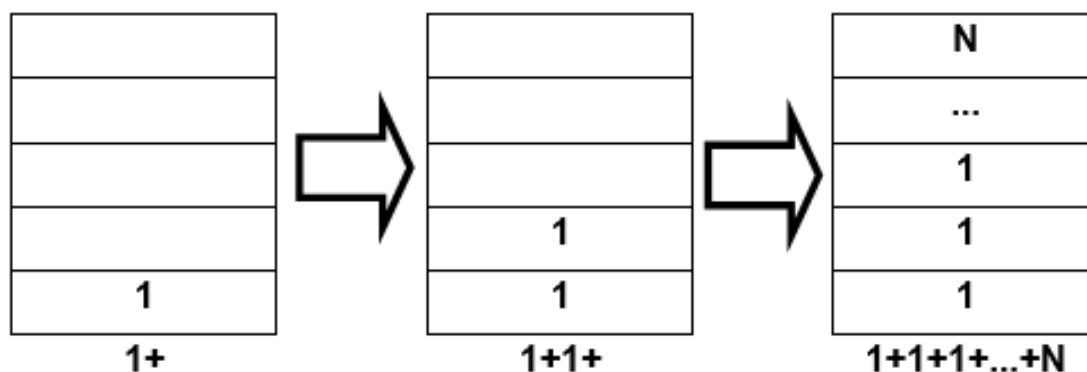
Dizemos que uma função é recursiva quando, dentro do corpo de uma função, se faz uma chamada para a própria função, conforme mostra a Figura 1.

Figura 1 – Exemplo de função recursiva

```
1 void funcao_recursiva(int n)
2 {
3     ...
4     funcao_recursiva(n-1);
5     ...
6 }
7
```

Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um Registro de Ativação na Pilha de Execução do programa. Esse registro de ativação armazena os parâmetros e as variáveis locais da função, bem como o “ponto de retorno” no programa ou subprograma que chamou essa função. Ao final da execução da função, o registro é desempilhado e a execução volta ao subprograma que chamou a função, conforme a Figura 2.

Figura 2 – Armazenamento da função recursiva em uma pilha de função recursiva



Quando a rotina começa a retornar os resultados das chamadas, começa o desempilhamento das chamadas da função na pilha, que retornarão ao



solicitante o agrupamento das camadas, até que se forme o resultado final. À medida que são desempilhados, libera-se os valores da memória.

Quando optamos por usar uma função recursiva para resolver um problema, precisamos saber como fazê-la parar e evitar o loop infinito que estudamos na aula anterior.

Para um melhor entendimento, vejamos alguns exemplos de algoritmos sem recursividade, e na sequência o mesmo algoritmo com recursividade.

Vejamos o Exemplo 1, sem recursividade. O algoritmo apresentado na Figura 3 vai imprimir os números de 1 a 20 usando uma função iterativa com o laço de repetição *for*.

Figura 3 – Algoritmo sem função recursiva

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int inicio, limite;
7
8      inicio = 1;
9      limite = 21;
10
11     for(int i = inicio; i<limite; i++){
12         printf("%d ", i);
13     }
14     printf("\n\n");
15     system("pause");
16     return 0;
17 }
```

A Figura 4 mostra a saída do algoritmo mostrado na Figura 3, após a sua execução.



Figura 4 – Saída do algoritmo

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Pressione qualquer tecla para continuar. . .
```

O algoritmo mostrado na Figura 5 vai imprimir os números de 1 a 20 usando o laço de repetição *for* e a função recursiva *imprimeN()*.

Figura 5 – Algoritmo com função recursiva

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int imprimeN (int inicio, int sfim);
5
6  int main()
7  {
8      int comeco, sfim, i;
9
10     comeco = 1;
11     sfim = 21;
12
13     printf("FUNCAO ITERATIVA\n");
14     for(i = comeco; i < sfim; i++){    // impressao com for
15         printf("%d ", i);    // imprime os numeros
16     }
17     printf("\n\n");
18
19     printf("FUNCAO RECURSIVA\n");
20     imprimeN(comeco, sfim);    //chamada da funcao recursiva
21
22     printf("\n\n");
23
24     system("pause");
25     return 0;
26 }
27 int imprimeN (int comeco, int sfim){    //Função recursiva
28     if(comeco < sfim){
29         printf("%d ", comeco);    // imprime os numeros
30         imprimeN(comeco+1, sfim);    //chamada recursiva
31     }
32 }
```

A Figura 6 mostra a saída do algoritmo mostrado na Figura 5, após a sua execução.



Figura 6 – Saída do algoritmo

```
FUNCAO ITERATIVA
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

FUNCAO RECURSIVA
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

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

Um algoritmo com recursividade torna a escrita mais simples e elegante, tornando-o fácil de entender e de manter. Porém, se o loop recursivo for muito grande, o consumo de recursos será proporcional, podendo sobrecarregar a memória, pois cada chamada recursiva aloca espaço na memória para as variáveis e parâmetros. Na maioria dos casos, com uma solução iterativa (usando laço de repetição) gasta-se menos memória, tornando o algoritmo mais eficiente, na performance do código, do que usar recursividade.

#### TEMA 4 – FUNÇÃO RECURSIVA COM VETOR

Na recursividade com vetor, usamos uma chamada de uma função, passando um elemento de um *array* como parâmetro; depois, dentro dessa função, fazemos uma nova chamada para ela mesma. Isso é, funciona do mesmo modo que uma função recursiva simples, só que, em vez do parâmetro ser uma variável, agora será um vetor. A Figura 7 traz um algoritmo sem função recursiva usando vetor e a Figura 9 traz o mesmo exemplo com função recursiva também usando vetor.





Figura 7 – Algoritmo sem função recursiva

```
1  #include <stdio.h>
2  #include <conio.h>
3  #define max 5 //definindo uma constante
4
5
6  main(){
7      int vet[max];
8      int i, j;
9
10     for (i=0; i<max; i++){
11         .....
12         printf("Digite o numero para o vetor [%d]: ", i);
13         scanf("%d",&vet[i]); //preenchendo o vetor com dados
14     }
15
16     printf("\n\n");
17
18     for(j = 0; j< max; j++){
19         .....
20         printf("Voce digitou o numero %d para o vetor [%d]\n",vet[j],j);
21     }
22
23     printf("\n\n");
24     system("pause");
25 }
```

A Figura 8 mostra a saída do algoritmo mostrado na Figura 7, após a sua execução.

Figura 8 – Saída do algoritmo

```
Digite o numero para o vetor [0]: 11
Digite o numero para o vetor [1]: 12
Digite o numero para o vetor [2]: 13
Digite o numero para o vetor [3]: 14
Digite o numero para o vetor [4]: 15

Voce digitou o numero 11 para o vetor [0]
Voce digitou o numero 12 para o vetor [1]
Voce digitou o numero 13 para o vetor [2]
Voce digitou o numero 14 para o vetor [3]
Voce digitou o numero 15 para o vetor [4]

Pressione qualquer tecla para continuar. . .
```

O algoritmo mostrado na Figura 9 traz a função recursiva *exibir()*.



Figura 9 – Algoritmo com função recursiva

```
1  #include <stdio.h>
2  #include <conio.h>
3  # define max 5 //definindo uma constante
4
5  int j=-1;
6  void exibir(int vetor[]);
7
8  main(){
9      int vet[max];
10     int i;
11
12     for (i=0;i<max;i++){
13         printf("Digite o numero para o vetor [%d]: ", i);
14         scanf("%d",&vet[i]); //preenchendo o vetor com dados
15     }
16
17     printf("\n\n");
18
19     exibir(vet); //chamada da função recursiva
20
21     printf("\n\n");
22     system("pause");
23 }
24
25
26 void exibir(int vetor[]){
27     j++;
28     if (j < max){ // condição de parada
29         //impressão do resultado da função.
30         printf("Voce digitou o numero %d para o vetor [%d]\n",vetor[j],j);
31         exibir(vetor); //chamada recursiva
32     }
33 }
```

A Figura 10 mostra a saída do algoritmo mostrado na Figura 9, após a sua execução.

Figura 10 – Saída do algoritmo

```
Digite o numero para o vetor [0]: 11
Digite o numero para o vetor [1]: 12
Digite o numero para o vetor [2]: 13
Digite o numero para o vetor [3]: 14
Digite o numero para o vetor [4]: 15

Voce digitou o numero 11 para o vetor [0]
Voce digitou o numero 12 para o vetor [1]
Voce digitou o numero 13 para o vetor [2]
Voce digitou o numero 14 para o vetor [3]
Voce digitou o numero 15 para o vetor [4]

Pressione qualquer tecla para continuar. . .
```



Passamos agora ao Exemplo 2, sem recursividade.

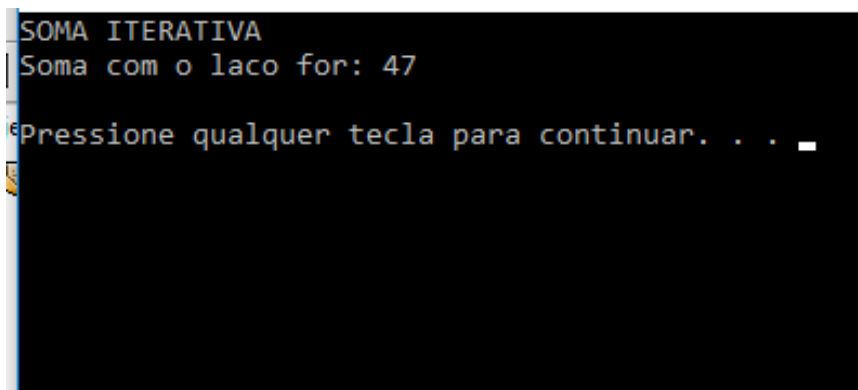
O algoritmo mostrado na Figura 11 vai imprimir a soma dos elementos do vetor usando o laço de repetição *for*.

Figura 11 – Algoritmo sem função recursiva

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define NVETOR 5 //Declaracao de constante
5
6  int main()
7  {
8      int vetor[NVETOR] = {22, 5, 11, 3, 6};
9      int i, soma = 0;
10
11      printf("SOMA ITERATIVA");
12      for(i = 0; i < NVETOR; i++){ //Laço FOR
13          soma = soma + vetor[i]; //Soma dos elementos do vetor
14      }
15      printf("\nSoma com o laço for: %d\n\n", soma); //Imprime resultado do FOR
16      system("pause");
17      return 0;
18  }
19
```

A Figura 12 mostra a saída do algoritmo mostrado na Figura 11, após a sua execução.

Figura 12 – Saída do algoritmo



```
SOMA ITERATIVA
Soma com o laço for: 47
Pressione qualquer tecla para continuar. . . _
```

O algoritmo mostrado na Figura 13 vai imprimir a soma dos elementos do vetor usando o laço de repetição *for* e usando a função recursiva *somaVetor()*.



Figura 13 – Algoritmo com função recursiva

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define NVETOR 5 //Declaracao de constante
5
6  int somaVetor(int vetor[], int tamanho);
7  int main()
8  {
9      int vetor[NVETOR] = {22, 5, 11, 3, 6}; // declaração e inicialização do vetor
10     int i, soma = 0;
11     int resultado = somaVetor(vetor, NVETOR); //chamada da função recursiva
12
13     printf("FUNCAO RECURSIVA\n");
14     //impressão do resultado da função recursiva
15     printf("Soma com a funcao recursiva: %i\n", resultado);
16
17     printf("\n");
18
19     for(i = 0; i<NVETOR; i++){ //laço de repetição for
20         soma = soma + vetor[i]; //soma dos elementos do vetor
21     }
22     printf("FUNCAO ITERATIVA\n");
23     printf("Soma com laco for: %d\n\n", soma); //impressão do resultado do for
24     system("pause");
25     return 0;
26 }
27 int somaVetor(int vetor[], int tamanho){ //função recursiva
28     if(tamanho == 1)
29         return vetor[0];
30     else
31         //soma dos elementos do vetor
32         return vetor[tamanho - 1] + somaVetor(vetor, tamanho - 1);
33 }
34 }
```

A Figura 14 mostra a saída do algoritmo mostrado na Figura 13, após a sua execução.

Figura 14 – Saída do algoritmo

```
FUNCAO RECURSIVA
Soma com a funcao recursiva: 47

FUNCAO ITERATIVA
Soma com laco for: 47

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



## TEMA 5 – FUNÇÕES MACRO – DIRETIVA #DEFINE

Neste capítulo, estudaremos os diversos usos da diretiva `#define`. Basicamente, uma diretiva avisa para o compilador que ele deve procurar todos os eventos de determinada expressão e substituir por outra na compilação do programa. Isso permite criar o que chamamos de *funções macro*.

Uma função macro é um tipo de declaração de função em que são informados o nome e os parâmetros da função como sendo o nome da macro e o trecho de código semelhante a ser aplicado na substituição.

### 5.1 Diretiva `#define`

A diretiva `#define` associa um identificador a uma cadeia de caracteres de token. Após a definição da macro, o compilador pode substituir a cadeia de caracteres de token em cada ocorrência do identificador no arquivo de origem.

A diretiva `#define` permite três sintaxes.

A primeira sintaxe **`#define nome_da_macro`**. Nela, define-se um nome que será ser usado em alguma estrutura no código. Exemplo: nome para ser testado em estruturas condicionais.

Para associar a diretiva `#define` com uma estrutura condicional, usamos outra diretiva chamada **`#ifdef`**. Nos próximos exemplos, Figura 15 e Figura 17, os algoritmos vão checar se o a definição **`status`** realmente existe. Caso exista a definição **`status`** os algoritmos executa a instrução **`printf("O Status existeeeee!!!! Uhuuuuu!!!!\n\n");`**, caso contrário, ele executará a instrução **`printf("Status NAO definido. O #define FOI DECLARADO?\n\n");`**. Vejamos exemplos apresentados na Figura 15 e Figura 17, sem e com **`#define`** respectivamente.

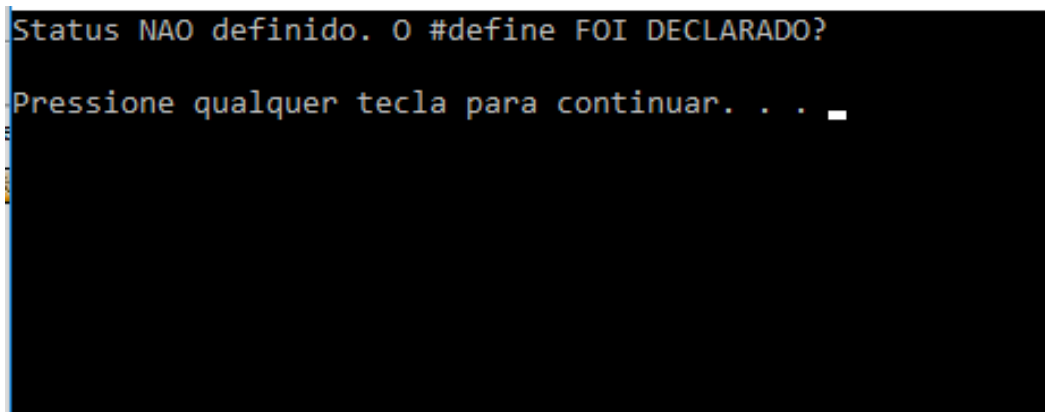


Figura 15 – Algoritmo sem diretiva #define

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      #ifdef status // ifdef diretiva de compilação condicional
6          printf("O Status existeeeee!!!! Uhuuuuu!!!!\n\n");
7      #else
8          printf("Status NAO definido. O #define FOI DECLARADO?\n\n");
9      #endif
10
11     system("pause");
12     return 0;
13 }
14
```

A Figura 16 mostra a saída do algoritmo mostrado na Figura 15, após a sua execução.

Figura 16 – Saída do algoritmo



```
Status NAO definido. O #define FOI DECLARADO?

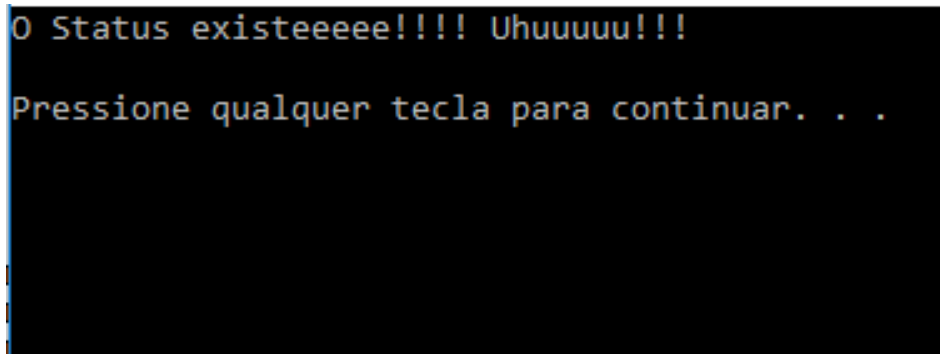
Pressione qualquer tecla para continuar. . . _
```

Figura 17 – Algoritmo com diretiva #define

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define status
5
6  int main() {
7      #ifdef status // ifdef diretiva de compilação condicional
8          printf("O Status existeeeee!!!! Uhuuuuu!!!!\n\n");
9      #else
10         printf("Status NAO definido. O #define FOI DECLARADO?\n\n");
11     #endif
12
13     system("pause");
14     return 0;
15 }
```

A Figura 18 mostra a saída do algoritmo mostrado na Figura 17, após a sua execução.

Figura 18 – Saída do algoritmo

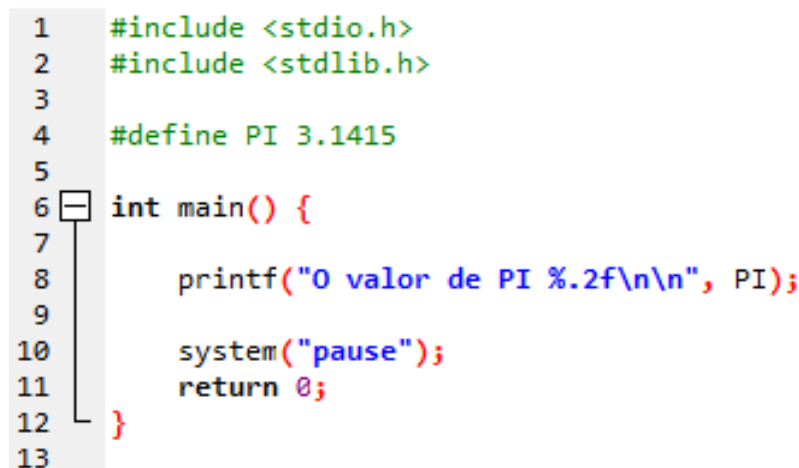


```
O Status existeeeee!!!! Uhuuuuu!!!  
Pressione qualquer tecla para continuar. . .
```

A segunda sintaxe **#define nomeConstante valorConstante**. Nela, define-se um valor para uma constante que será usada ao longo do desenvolvimento do algoritmo.

Essa sintaxe informa ao compilador que ele deve procurar todas as ocorrências de “**nomeConstante**” e substituir por “**valorConstante**” no momento da compilação do programa, conforme exemplo apresentado na Figura 19.

Figura 19 – Algoritmo com diretiva #define



```
1  #include <stdio.h>  
2  #include <stdlib.h>  
3  
4  #define PI 3.1415  
5  
6  int main() {  
7  
8      printf("O valor de PI %.2f\n\n", PI);  
9  
10     system("pause");  
11     return 0;  
12 }  
13
```

A Figura 20 mostra a saída do algoritmo mostrado na Figura 19, após a sua execução.



Figura 20 – Saída do algoritmo

```
O valor de PI 3.14
Pressione qualquer tecla para continuar. . .
```

A terceira sintaxe **#define nome\_da\_macro(PARÂMETROS) expressão**. Ela é uma função macro, ou seja, um pedaço de código pelo qual foi atribuído a um nome. Vejamos Figura 21.

Figura 21 – Algoritmo #define com função com expressão

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define maior(x,y) x>y?x:y
5
6  int main() {
7      int a = 12;
8      int b = 6;
9      int c = maior(a,b);
10
11     printf("Maior valor = %d\n\n",c);
12
13     system("pause");
14     return 0;
15 }
16
```

A Figura 22 mostra a saída do algoritmo mostrado na Figura 21, após a sua execução.

Figura 22 – Saída do algoritmo

```
Maior valor = 12
Pressione qualquer tecla para continuar. . .
```





## 5.2 Boas práticas da diretiva #define

A ideia é não apenas escrever um código funcional, mas sim usar as boas práticas para ter um código limpo e de fácil entendimento. Um código mal escrito dificulta a sua alteração por outros programadores, e é suscetível a erros graves de compilação. Um código malfeito provoca um retrabalho desnecessário e frustração para quem vai ter que reprogramá-lo.

Qualquer tecnologia pode ser usada para ajudar ou piorar de vez a situação. Essa afirmação também se aplica quando usamos macro. Se bem utilizada, pode facilitar muito a lógica do nosso algoritmo, melhorar a leitura do código e otimizá-lo. Só que, se for mal utilizada, pode nos causar muita dor de cabeça, na busca interminável pelos erros do código.

Cada recurso da linguagem de programação tem sua particularidade. As boas práticas existem para que todos tomem os cuidados necessários, evitando-se certos erros no desenvolvimento do algoritmo. Quando usamos macro, é aconselhável sempre colocar, na sequência de substituição, os parâmetros da macro entre parênteses. Isso serve para preservar a “precedência dos operadores”, conforme exemplo apresentado na Figura 23.

Figura 23 – Exemplo de presidência na expressão

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define exprod1(x,y) x*y;
5  #define exprod2(x,y) (x)*(y);
6
7  int main() {
8      int a = 12;
9      int b = 6;
10     int c = exprod1(a+2,b); // a + 2 * b
11     int d = exprod2(a+2,b); // (a + 2) * (b)
12
13     printf("O valor de c = %d\n\n",c);
14     printf("O valor de d = %d\n\n",d);
15
16     system("pause");
17     return 0;
18 }
```



A Figura 24 mostra a saída do algoritmo mostrado na Figura 23, após a sua execução.

Figura 24 – Saída do algoritmo

```
O valor de c = 24
O valor de d = 84
Pressione qualquer tecla para continuar. . . _
```

Deve-se sempre usar letras maiúsculas para declarar sua macro, seja ela uma constante ou uma função. Esse padrão é conhecido pelos programadores experientes e facilita a leitura e a manutenção do código. Vejamos o exemplo apresentado na Figura 25.

Figura 25 – Exemplo de diretiva com letra maiúscula

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define ERRADO(msg) printf("ERRO:%s\n", msg)
5
6  int main() {
7
8      int x;
9
10     printf("Digite um numero entre 0 e 10: ");
11     scanf("%d", &x);
12
13     if((x>0)&&(x<10)){
14         ERRADO("NAO CONTEM ERRO!");
15     }
16     else{
17
18         ERRADO("ATENCAO No DIGITADO INVALIDO!");
19     }
20     system("pause");
21     return 0;
22 }
```



A Figura 26 mostra a saída do algoritmo mostrado na Figura 25, após a sua execução.

Figura 26 – Saída do algoritmo

```
Digite um numero entre 0 e 10: 6
ERRO:NAO CONTEM ERRO!
Pressione qualquer tecla para continuar. . .
```

Preste muita atenção quando for usar operadores unários dentro de macros. Chamar a macros os operadores de incremento/decremento podem ser um problema. Observe o exemplo na Figura 27.

Figura 27 – Exemplo de diretiva com operadores unitários

```
1  #define MENOR(a,b) ((a)>(b)?(b):(a))
2  min = MENOR(a++, b);
```

O compilador irá executar a macro conforme mostrado na Figura 28.

Figura 28 – Exemplo de diretiva com operadores unitários.

```
1  #define MENOR(a,b) ((a++)>(b)?(b):(a++))
2
```

Se o “a” for maior que o “b”, o incrementado será executado mais de uma vez.

É preciso atentar também à expansão de linhas em macro dentro de um laço. A barra “\” em macro representa a quebra de linha e indica para o compilador que a macro continuará na linha abaixo. O exemplo na Figura 29 representa um erro nesse contexto.



Figura 29 – Exemplo de um algoritmo sem as chaves no *if*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define A 13
5  #define SAN printf("primeira linha\n"); \
6             printf("segunda linha\n");    \
7             printf("terceira linha\n\n");
8
9  int main() {
10
11     if (A == 13)
12         SAN
13     else
14         printf("NAO CONFERE!");
15
16     return 0;
17 }
```

O código apresentado na Figura 29 não vai compilar porque a macro possui mais de uma linha; quando executada no “if”, o compilador apresentará erro. Se você colocar o bloco do “if” entre chaves “{ }”, resolverá o problema, conforme mostrado na Figura 30.

Figura 30 – Exemplo de um algoritmo com as chaves no *if*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define A 13
5  #define SAN printf("primeira linha\n"); \
6             printf("segunda linha\n");    \
7             printf("terceira linha\n\n");
8
9  int main() {
10
11     if (A == 13){
12         SAN
13     }
14     else
15         printf("NAO CONFERE!");
16
17     return 0;
18 }
```

A Figura 31 mostra a saída do algoritmo mostrado na Figura 29, após a sua execução.

Figura 31 – Saída do algoritmo

```
primeira linha
segunda linha
terceira linha

-----
Process exited after 0.2399 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

### 5.3 Diretiva #undef

É utilizada sempre que desejamos apagar a definição de uma macro da tabela interna que as guarda. Em outras palavras, ela remove a definição de uma macro para que ela possa ser redefinida.

A diretiva #undef segue a seguinte sintaxe: **#undef nome\_da\_macro**. Vejamos exemplo mostrado na Figura 32.

Figura 32 – Exemplo de diretiva #undef

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define VALOR 235
5
6  int main() {
7
8      printf("Valor = %d\n\n", VALOR);
9
10     #undef VALOR
11     #define VALOR 20
12
13     printf("Novo valor = %d\n\n", VALOR);
14
15     system("pause");
16     return 0;
17 }
```

A Figura 33 mostra a saída do algoritmo mostrado na Figura 32, após a sua execução.



Figura 33 – Saída do algoritmo

```
Valor = 235  
Novo valor = 20  
Pressione qualquer tecla para continuar. . . _
```

## FINALIZANDO

Nesta aula, aprendemos os principais conceitos que envolvem as aplicações de recursividade, iteração, função recursiva, função recursiva com vetor e função macro com a diretiva `#define`. Trabalhamos também com a linguagem C para a criação de algoritmos. Bons estudos!



---

## REFERÊNCIAS

ASCENCIO, A. F. G. **Fundamentos da Programação de Computadores:** Algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

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

PUGA, S.; RISSETTI, G. **Lógica de programação e estruturas de dados.** São Paulo: Pearson, 2016.