Paradigmas de Programação

**Paradigma Procedural com C**

## Introdução

Este material contém notas de aula sobre o *Paradigma Procedural*. O objetivo é apresentar os conceitos deste paradigma utilizando a linguagem de programação C e o compilador *GNU Compiler Collection* – GCC[[1]](#footnote-0). O GCC foi escolhido por ser um dos compiladores mais utilizados, por ser um software livre e estar disponível em diversas plataformas de computação (GREGG, 2024).

## Revisão

Paradigma Imperativo:

* Do latim: *imperare*. Imperar, reinar, governar, dominar, mandar, ordenar.
* Modelo em que a programação é feita com comandos que mudam o estado do programa.
* A programação descreve o fluxo de controle, e é baseada na arquitetura de von Neumann.
* Dados e programas são armazenados na mesma memória.
* CPU executa instruções.
* Exemplos de linguagens: C, C++, C#, Java, Python.
* Caracteriza-se por 3 construções: variável, atribuição e sequência.
* Dividido em estruturado e não estruturado.

Paradigma estruturado:

* Reduzido a três estruturas: sequência, decisão e repetição.
* Essas estruturas especificam os passos necessários para que o programa atinja o resultado esperado.
* Surgiu na década de 50, com ALGOL.
* Representa a forma "natural" de pensar na solução de problemas.
* Exemplos de linguagens: ALGOL, C, Java e Python.

## Paradigma Procedural

* O código contém as três estruturas e tem como unidade básica a função.
* O código é organizável, reusável e modular.
* A função é um bloco de código "autocontido", reusável, com instruções que realizam uma tarefa específica.
* Uma vez que uma função é escrita, ela pode ser usada continuamente.
* Uma função pode ser chamada de dentro de outras funções, e até de dentro dela mesma.
* Em geral, a função possui uma lista de parâmetros (que pode ser vazia) e um tipo de retorno.
* Exemplos de linguagens: Basic, C, Fortran, Perl, PHP e Python.

## A Linguagem C

A linguagem de programação C foi criada por Dennis Ritche, em 1972. Ela é baseada em uma linguagem de programação da Bell Laboratories, cujo nome era "B" (daí o nome "C").

Diversas aplicações conhecidas foram programadas, no todo ou em parte, usando C. Exemplos:

* Sistemas operacionais como Unix (SEBESTA, 2012), Linux e nuttx (THE APACHE SOFTWARE FOUNDATION, 2024);
* Sistemas de gerenciamento de banco de dados como SQLite (SQLITE, 2024) e MySQL (MYSQL-SERVER, 2024);
* Navegadores web como Google Chrome e Chromium (CHROMIUM, 2024);
* Jogos (CHATTERJEE, 2024);
* Ferramentas de desenvolvimento, como GCC (FREE SOFTWARE FOUNDATION, 2024a) e o GDB (FREE SOFTWARE FOUNDATION, 2024b).

Um código em linguagem C pode conter:

* Diretivas de pré-processamento
* Definições de tipos e estruturas de dados
* Declarações de variáveis
* Definições de funções
* Instruções
* Comentários[[2]](#footnote-1)

A função main é sempre o ponto de partida do programa. De acordo com o padrão ANSI C, as cinco opções abaixo são as assinaturas válidas para esta função:

**int main()**

**int main(void)**

**int main(int argc, char \*argv[])**

**int main(int argc, char \*argv[], char \*envp[])**

**int main(int argc, char \*\*argv)**

O arquivo [exemplo-printf.c](https://drive.google.com/file/d/1bw7JSbxxfiE4NBdHkJDYlVLoXxjnk8zZ/view?usp=drive_link) traz um primeiro exemplo de código, contendo:

* Uma diretiva de pré-processamento que instrui o compilador a incluir o conteúdo do arquivo de cabeçalho stdio.h no programa. Este arquivo de cabeçalho contém declarações para funções de entrada e saída padrão. Pré-processador é a ferramenta que converte diretivas antes de ocorrer o processamento do código.
* A função main, sem parâmetros, contendo:
  + A instrução printf(, que exibe o conteúdo na tela (a mensagem “Olá\n”)
  + A instrução return, que realiza o retorno da função.

O código do arquivo citado está disponível no quadro abaixo:

| **// Bibliotecas utilizadas no código**  **#include <stdio.h>**  **// Função principal**  **int main() {**  **printf("Olá!\n");**  **return 0;**  **}** |
| --- |

### Tipos

A linguagem C possui tipagem estática e fraca, embora exista muita discussão sobre a "fraqueza" do seu sistema de tipos.

A linguagem C reconhece alguns tipos básicos de dados. Entre eles estão:

* **char**: usado para armazenar caracteres. É armazenado como um número inteiro de 1 *byte* que representa um valor ASCII.
* **int**: é usado para armazenar números inteiros. Pode representar números positivos e negativos dentro de uma faixa específica, dependendo da arquitetura do sistema. Em sistemas de 32 *bits*, um int geralmente varia de -2.147.483.648 a 2.147.483.647.
* **unsigned int**: semelhante ao tipo int, mas só representa valores não negativos. Aceita valores de 0 a 4.294.967.295 em sistemas de 32 *bits*, e pode ocupar de 2 a 4 *bytes*.
* **short**: é usado para representar números em uma faixa mais restrita que o tipo int. Ocupa 2 *bytes* e aceita valores entre -32.768 e 32.767 (HEXSEL, 2024).
* **float**: é usado para armazenar números de ponto flutuante de precisão simples. Geralmente tem 4 *bytes* de tamanho e pode representar valores na faixa de 1,175494e-38 e 3,402823e+38.
* **double**: é usado para armazenar números de ponto flutuante de precisão dupla. Oferece uma precisão maior do que float, permitindo armazenar valores decimais com mais dígitos. Geralmente tem 8 *bytes* de tamanho e pode representar valores na faixa de aproximadamente 2,225074e-308 a 1,797693e+308.
* **void**: é usado para indicar a ausência de tipo. Em outras palavras, ele é usado quando nenhum tipo é aplicável ou quando nenhum valor é retornado.

A declaração de variáveis é feita informando o tipo e uma lista de identificadores daquele tipo, separados por vírgula. Opcionalmente, uma atribuição de valor inicial pode ser feita. As declarações abaixo seriam válidas em C:

**int a, b, c;**

**int d = 10, e, f;**

**int g, h, i = 10;**

**int j;**

**int k;**

**int l;**

O quadro a seguir apresenta um exemplo de código em C utilizando variáveis dos tipos citados. O código também está disponível no arquivo [exemplo-declaracao-variaveis.c](https://drive.google.com/file/d/1xrbMjwgLleEmNuQtaawmObcJLSEz24fo/view?usp=sharing).

| **// Bibliotecas utilizadas no código**  **#include <stdio.h>**  **// Função principal, agora com retorno inteiro**  **int main() {**  **// Variável**  **unsigned int numeroSemSinal = 9000000;**  **int numeroComSinal = -8000000;**  **short valor = 3;**  **float salario = 1.79769e+308; // Não cabe, pois excede o limite do tipo float! Troque por 1.175**  **double precisao = 1.79769e+308;**  **char caractere = 'L';**  **// Comandos de escrita, imprimindo o valor de cada variável.**  **// O especificador de cada formato é escrito com %[letra]:**  **// %u para inteiro sem sinal**  **// %i para inteiro**  **// %hi para inteiro pequeno**  **// %f para número real de ponto flutuante**  **// %lf para número real de ponto flutuante longo ou com dupla precisão**  **// %c para caractere**  **printf("Valor: %u\n", numeroSemSinal);**  **printf("Valor: %i\n", numeroComSinal);**  **printf("Valor: %hi\n", valor);**  **printf("Valor: %f\n", salario);**  **printf("Valor: %lf\n", precisao);**  **printf("Caractere: %c\n", caractere);**    **return 0;**  **}** |
| --- |

A seguir, outro exemplo de código é apresentado. O código está disponível também no arquivo [exemplo-tipagem-fraca.c](https://drive.google.com/file/d/1Z_iv6vITOQmtnc7eJul5RMRtf4CFnVeo/view?usp=drive_link) e elucida a tipagem fraca:

| **#include <stdio.h>**  **int main() {**  **int i = 92;**  **void \*v = &i;**  **char \*c = v;**    **printf("%c", \*c);**    **return 0;**  **}** |
| --- |

Este código mostra que C tem tipagem fraca porque permite que um ponteiro para void (void\*) seja implicitamente convertido para um ponteiro para char (char\*), sem a necessidade de um *cast* explícito. Na linha **char \*c = v;**, o ponteiro v, que originalmente aponta para um inteiro (**int i**), é atribuído a c, que é um ponteiro para um caractere (char). Essa conversão implícita é possível em C devido à sua natureza de tipagem fraca, que oferece flexibilidade ao programador, mas também pode levar a comportamentos inesperados e erros difíceis de detectar, especialmente em programas grandes e complexos.

Ao tentar compilar o mesmo código com um compilador para C++, por exemplo, a ferramenta g++, nota-se que um erro será exibido, impedindo a criação do executável por entender que a conversão de void\* para char\* é inválida. Alguns compiladores C exibem um aviso (*warning*), mas, em geral, permitem esse tipo de conversão.

### Estruturas de Decisão

As estruturas de decisão permitem controlar o fluxo do programa com base em uma condição. A condição é uma expressão lógica que, em C, é avaliada para um valor inteiro. Isso significa dizer que, ao avaliar uma expressão lógica, esta retorna zero, caso a expressão seja falsa, e qualquer valor diferente de zero, caso a expressão seja verdadeira.

As estruturas de decisão em C são o if e o switch.

Exemplos de códigos utilizando estruturas de decisão:

* [exemplo-if.c](https://drive.google.com/file/d/1cNaajTr1OVSRusaWF3Lg4c0jUYJb35Yv/view?usp=sharing)
* [exemplo-switch.c](https://drive.google.com/file/d/1cqN4EWZbAS8SAPlEpNkBv6eZATuq-SpC/view?usp=sharing)

O operador ternário também é uma forma de realizar decisões em C. Ele é uma forma compacta de expressar uma operação condicional. O operador ternário é composto por três operandos e tem a seguinte sintaxe: (condição) ? expressão1 : expressão2

A condição é seguida por uma interrogação (?) e dois valores separados por dois pontos (:). Se a condição for verdadeira, a expressão1 é avaliada e seu valor é retornado. Se a condição for falsa, a expressão2 é avaliada e seu valor é retornado.

### Estruturas de Repetição

As estruturas de repetição disponíveis em C são: while, do…while e for. Exemplos de códigos utilizando essas estruturas estão disponíveis nos arquivos:

* [exemplo-while.c](https://drive.google.com/file/d/19yTf_t68ZhqIZH8xCQ3nma6WzML1rhjJ/view?usp=drive_link)
* [exemplo-do-while.c](https://drive.google.com/file/d/1-5TqtebmGWBAyxSXW8fL7AV3wMB1Yw9y/view?usp=drive_link)
* [exemplo-for.c](https://drive.google.com/file/d/1yxnoMOaK3tGZQvFVw_qH-n3whryQt_G8/view?usp=drive_link)

Estruturas de decisão e repetição também podem ser implementadas com goto. O arquivo [exemplo-switch-goto.c](https://drive.google.com/file/d/1d3pdS96g9uK04oOJeXKypdb9HkkjeEWv/view?usp=drive_link) exemplifica a criação de um loop utilizando goto e switch.

### Principais Bibliotecas

As principais bibliotecas em C são aquelas que fornecem funcionalidades fundamentais para a linguagem, incluindo manipulação de entrada e saída, operações matemáticas, gerenciamento de memória, ordenação de dados e manipulação de strings. Algumas dessas bibliotecas são:

* **stdio.h**: Fornece funções para entrada e saída.
* **stdlib.h**: Contém funções para alocação de memória dinâmica (malloc, calloc, realloc), controle de processos (exit) e ordenação de dados.
* **math.h**: Contém funções matemáticas como seno, cosseno, exponenciação e logaritmos.
* **string.h**: Fornece funções para manipulação de strings, como cópia, concatenação e comparação.

Um exemplo utilizando as bibliotecas stdio.h e math.h está disponível no quadro abaixo e no arquivo [exemplo-pow-math.c](https://drive.google.com/file/d/1KKUWJaqI2YshhtEy5S0FOiciowyeCqu5/view?usp=drive_link):

| // Bibliotecas utilizadas no código. Pode ser necessário utilizar o parâmetro -lm para compilar.  #include <stdio.h>  #include <math.h>  int main() {  double n;    // Comandos de escrita e leitura  printf("Digite o número: ");  scanf("%lf", &n);  // Imprime a saída formatada com .2lf, assim serão exibidos somente dois dígitos após o ponto (a vírgula).  printf("O quadrado de %.2lf é %.2lf\n", n, pow(n, 2));  return 0;  } |
| --- |

### Entrada e Saída

Entrada e Saída (E/S) é uma parte fundamental de qualquer linguagem de programação, pois permite que os programas interajam com o mundo exterior. Em C, a E/S é tratada através da biblioteca padrão stdio.h.

Algumas funções definidas em stdio.h:

* Escrita:
  + **printf()**: usada para exibir um texto formatado na saída padrão.
  + **fprintf()**: similar à função printf(), mas escreve a saída formatada em um arquivo ao invés da saída padrão.
  + **puts()**: exibe uma *string* na saída padrão seguida por uma nova linha.
* Leitura:
  + **scanf()**: usada para ler dados formatados da entrada padrão.
  + **fscanf()**: similar à função scanf(), mas lê os dados de um arquivo ao invés da entrada padrão.
  + **gets()**: lê uma linha de texto da entrada padrão e a armazena em uma *string*.
  + **fgets()**: lê uma linha de texto da entrada padrão e a armazena em uma *string*. É mais segura que gets() porque permite especificar o tamanho máximo da *string*, evitando estouro de *buffer*.

**Observação**: a entrada padrão normalmente é o console.

### Operadores

Os principais operadores em C são:

* Aritméticos
  + Adição (+)
  + Subtração (-)
  + Multiplicação (\*)
  + Divisão (/)
  + Módulo (%)
  + Pré/Pós-incremento (++)
  + Pré/Pós-decremento (--)
* Relacionais
  + Igual a (==)
  + Diferente de (!=)
  + Maior que (>)
  + Menor que (<)
  + Maior ou igual a (>=)
  + Menor ou igual a (<=)
* Lógicos:
  + E lógico (&&)
  + OU lógico (||)
  + NÃO lógico (!)
* *Bit* a *bit* (*Bitwise*):
  + E *bit* a *bit* (&)
  + OU *bit* a *bit* (|)
  + XOU (XOR) *bit* a *bit* (^)
  + Complemento *bit* a *bit* (~)
  + Deslocamento à esquerda (<<)
  + Deslocamento à direita (>>)
* Atribuição:
  + Atribuição simples (=)
  + Atribuição com adição (+=)
  + Atribuição com subtração (-=)
  + Atribuição com multiplicação (\*=)
  + Atribuição com divisão (/=)
  + Atribuição com módulo (%=)
* Outros:
  + Operador de endereço de memória (&)
  + Operador de ponteiro de indireção (\*)
  + Operador de seleção de membro (.)
  + Operador de seleção de membro através de ponteiro (->)
  + Operador ternário condicional (?:)

O arquivo [exemplo-operadores-bitwise-decremento-incremento-relacional.c](https://drive.google.com/file/d/1JG5_VHMBK3QKbAy_XvRCSTKWaC4RSKaR/view?usp=drive_link) traz um exemplo de código que utiliza alguns dos operadores citados.

Alguns operadores não estão explicados neste material. O leitor é encorajado a buscar mais informações sobre esses operadores.

### *Arrays*

Em C, um *array* é uma coleção de elementos do mesmo tipo, organizados em sequência contígua na memória. Essa estrutura tem tamanho fixo, definido na sua declaração. Os elementos de um *array* são acessados por meio de um índice, um número inteiro que indica sua posição dentro do *array*, iniciando na posição zero.

Os arquivos [exemplo-array.c](https://drive.google.com/file/d/1825oxWKaNAsYHvfLxHQKo98rS0ZxIzAX/view?usp=drive_link) e [exemplo-array-funcao.c](https://drive.google.com/file/d/1mPvWcjtmLyvttrB34i-IpygcwAXFwLaw/view?usp=drive_link) demonstram o uso de *arrays* em C. O primeiro exemplo cria *arrays* para armazenar notas de estudantes, de forma que cada estudante tenha duas notas, armazenadas no *array* notas, e uma média final calculada a partir dessas notas e armazenada no *array* mediaFinal. Neste exemplo, cada aluno acaba sendo representado por um índice, de forma que o primeiro aluno seja o índice 0, o segundo aluno seja o índice 1 e assim por diante. Para acessar a média final do 14º aluno, bastaria acessar mediaFinal[13], pois os índices começam em zero. Para o *array* notas, como cada aluno tem duas notas, o acesso é feito por indice \* 2, para a primeira nota, e indice \* 2 + 1 para a segunda nota.

O arquivo [exemplo-array-funcao.c](https://drive.google.com/file/d/1mPvWcjtmLyvttrB34i-IpygcwAXFwLaw/view?usp=drive_link) contém um código semelhante ao do exemplo anterior, mas contendo uma função leNota para ler e validar cada nota lida e uma função para calcular a média final. A principal vantagem do segundo arquivo está na reutilização do código e na redução da repetição.

### Matrizes

Uma matriz é uma estrutura de dados que permite armazenar elementos de forma organizada em uma grade multidimensional. Cada elemento de uma matriz é acessado por meio de índices correspondentes ao número de dimensões da matriz. Por exemplo, uma matriz bidimensional requer dois índices, enquanto uma matriz tridimensional requer três. Uma matriz unidimensional é equivalente a um *array*.

As matrizes são declaradas especificando o tipo de dado dos elementos, seguido pelo nome da matriz e o número de elementos em cada dimensão, entre colchetes. Por exemplo, uma declaração válida seria:

int matriz[3][3];

A instrução acima declara matriz bidimensional de inteiros, com dimensões 3x3. As matrizes em C são armazenadas de forma contígua na memória, facilitando o acesso eficiente aos elementos.

O arquivo [exemplo-matriz.c](https://drive.google.com/file/d/1eVBeqS5tdXKjaW93Y7lotki5yy__8YZh/view?usp=drive_link) traz um exemplo de código que utiliza duas matrizes: uma para armazenar os nomes dos alunos e outra para armazenar as presenças deles nos períodos de aula.

### Strings

Em C, *strings* são implementadas como sequências de char, o que pode ser feito por meio de ponteiros e *arrays*. O arquivo [exemplo-string.c](https://drive.google.com/file/d/1TYCp3Q2ywK0IQovrNRM1sebF_aekKGC1/view?usp=drive_link) e o quadro abaixo apresentam um código que faz uso de *string*. É possível notar que existem diferentes formatações para ler os caracteres.

| #include <stdio.h>  #include <stdlib.h>  int main() {  char\* nome;  // Aloca espaço para 99 caracteres, mais um para o caractere nulo de terminação  nome = (char \*)malloc(100 \* sizeof(char));  // A declaração abaixo é equivalente às duas linhas acima e não exige o uso de stdlib.h (pois não usa malloc):  // char nome[100];  int idade;    // Comandos de saída e entrada  // %s = string  // %i = inteiro  // %\*c = ponteiro para caractere(s)  printf("Digite seu nome: ");  // Se o nome não tiver espaço, a leitura pode ser feita com esse comando:  //scanf("%s", nome);  // %s lê uma sequência de caracteres até encontrar um espaço em branco, tabulação ou nova linha.  // %[^\n] lê todos os caracteres da entrada até encontrar um caractere de nova linha (\n).  scanf ("%[^\n]%\*c", nome);  printf("Digite sua idade: ");  scanf("%i", &idade);  printf("%s tem %i anos!\n", nome, idade);  return 0;  } |
| --- |

Toda sequência de caracteres precisa ter um caractere nulo de terminação (\0) para indicar o seu fim. Este caractere nulo é importante porque permite que as funções que manipulam *strings*, como printf, scanf, strlen[[3]](#footnote-2), entre outras, identifiquem onde a *string* termina na memória. Por exemplo, o texto “IFC” é armazenado como ‘I’, ‘F’, ‘C’, ‘\0’.

### Funções

O elemento principal do paradigma procedural é a função. Uma função é um bloco de código coeso que executa uma tarefa específica. Isso significa que uma função realiza uma operação bem definida, e seu código está focado em resolver esse problema particular.

Usar funções oferece diversas vantagens:

* Promove a modularidade, dividindo o código em partes menores e mais gerenciáveis, o que facilita a compreensão, a manutenção e a depuração do código.
* Permite reutilizar o código, evitando a duplicação e promovendo a consistência em todo o programa.
* Facilitam a colaboração entre os membros da equipe, pois as funções bem definidas fornecem uma interface clara para a interação entre diferentes partes do código.
* Promovem abstração, permitindo que os desenvolvedores se concentrem nos aspectos de alto nível do problema, sem se preocupar com os detalhes de implementação.
* São fáceis de testar, pois realizam uma única tarefa claramente definida. Isso torna mais fácil identificar e corrigir problemas no código.
* Facilitam a organização, uma vez que ao agrupar tarefas relacionadas em funções separadas, o código tem estrutura e legibilidade melhoradas.

As funções em C são declaradas usando a seguinte sintaxe:

tipo\_de\_retorno identificador\_da\_função(parâmetros) {

// corpo da função

}

O tipo de retorno é o tipo do(s) dado(s) retornado pela função. O identificador da função é a expressão que dá nome à função. Parâmetros são valores que são passados para uma função quando ela é chamada, contendo um tipo e um identificador, separados por vírgula, assim como as variáveis. Eles representam informações que a função precisa para realizar sua tarefa. Já o corpo da função, delimitado por chaves, contém o conjunto de instruções que realizam a tarefa da função.

Para funções que retornam um valor específico (como int, float, char, etc.), o corpo da função terá uma instrução return, obrigatória, que deve ser usada para especificar o valor a ser retornado. Por exemplo, uma função que calcula a soma de dois números deve incluir uma instrução return para retornar o resultado desse cálculo. No entanto, para funções do tipo void, que não retornam valor, o return é opcional e pode ser omitido. Em geral, coloca-se return em funções void se o programador quiser indicar explicitamente que a função chegou ao fim.

Existe uma particularidade na declaração de funções em C. Quando uma função for definida no código após o seu uso, é necessário fazer sua declaração, o que também é chamado de protótipo de função. Quando a função é definida antes de ser chamada em outra função, o protótipo não é necessário. Os códigos presentes nos arquivos [exemplo-funcao.c](https://drive.google.com/file/d/1LpNzJDMkabSVIpPIBFxZeNxglj8Rm7us/view?usp=drive_link) e [exemplo-funcao-prototipo.c](https://drive.google.com/file/d/1Dvl577xbtZ-Y18FxjaJ3cUr7KGJXTejL/view?usp=drive_link) demonstram, respectivamente, o uso da função somaValores sem e com protótipo.

#### Recursividade

Um tipo especial de função possível de ser implementado em C é a **função recursiva**. Uma função recursiva é uma função que chama a si mesma durante sua execução. Esse tipo de função é comumente usado para resolver problemas que podem ser divididos em problemas menores e idênticos ao original.

Na recursão, o problema é dividido em

* **caso(s) base**: representa(m) a(s) condição(ões) em que a recursão termina; e
* **caso(s) recursivo(s)**: representa(m) a subdivisão do problema original em problemas menores.

Uma função recursiva continua chamando a si mesma com parâmetros modificados até que atinja um caso base, momento em que a recursão para e os resultados são retornados.

O exemplo disponível no arquivo [exemplo-funcao-recursiva.c](https://drive.google.com/file/d/1lQncPQ3akzm5yEvNj-zRDTIWahsZAd2Q/view?usp=drive_link) mostra o mesmo problema abordado nos dois exemplos anteriores (exemplo-funcao.c e exemplo-funcao-prototipo.c) sendo resolvido com recursão. Os exercícios funcionam de forma similar, no entanto, ao testá-los com valores muito grandes, tal como o valor 900000, é possível que a solução recursiva apresente o problema de falha de segmentação.

A falha de segmentação é uma situação em que o programa tenta acessar uma região de memória que ele não possui permissão para acessar. Como as chamadas recursivas são empilhadas na pilha de chamadas[[4]](#footnote-3) (*call stack*) do programa, o número de chamadas recursivas para valores grandes pode exceder o tamanho máximo da pilha disponível. Isso resulta em um estouro de pilha (*stack overflow*), o que leva ao erro de falha de segmentação.

A escolha entre uma abordagem iterativa e uma abordagem recursiva depende das características do problema a ser resolvido, das restrições de desempenho e da memória do ambiente de execução.

A abordagem iterativa costuma ser preferível quando o problema pode ser expresso em termos de *loops* e iterações. Ela é mais adequada para problemas que não exigem a repetição de chamadas de função. Além disso, pode ser mais eficiente em termos de desempenho e uso de memória, especialmente para problemas com grandes conjuntos de dados ou que farão muitas chamadas de função (no caso de funções recursivas), correndo risco de haver estouro de pilha, como no caso do exemplo citado anteriormente.

Exemplos de situações em que a iterativa pode ser preferível: cálculo do fatorial e busca em largura em grafos.

A abordagem recursiva é preferível quando a estrutura do problema se alinha naturalmente com a recursão, como problemas que envolvem subdivisão recursiva. Ela pode ser mais expressiva e legível em certos casos, especialmente para algoritmos complexos ou problemas que podem ser simplificados por meio de recursão. Outra situação em que a recursividade é vantajosa ocorre quando a eficiência do código é menos crítica do que a clareza e a simplicidade da solução.

Exemplos de situações em que a recursiva pode ser preferível: travessia de árvores (pré-ordem, pós-ordem e em-ordem) e o problema da torre de Hanói.

### Diretivas

Diretivas em C são instruções que são processadas pelo pré-processador antes da compilação do código. Elas começam com o caractere # e fornecem instruções ao compilador sobre como processar o código-fonte. As diretivas em C são:

* **#define**: Define uma macro. Uma macro é uma sequência de instruções que é definida uma vez e pode ser usada em várias partes do código. São frequentemente usadas para definir constantes e funções em linha.
* **#endif**: Marca o fim de um bloco condicional iniciado por **#if**, **#ifdef** ou **#ifndef**.
* **#error**: Emite uma mensagem de erro durante a compilação.
* **#include**: Inclui o conteúdo de um arquivo de cabeçalho no código fonte.
* **#if**, **#elif** e **#else**: Condições de compilação condicional com base em expressões
* **#ifdef** e **#ifndef**: Testa se uma macro está definida ou não.
* constantes.
* **#line**: Controla a numeração da linha do arquivo fonte durante a compilação.
* **#pragma**: Fornece instruções ao compilador, como otimizações específicas ou comportamentos de compilação.
* **#undef**: Remove a definição de uma macro.
* **#warning**: Emite uma mensagem de aviso durante a compilação.

Exemplos de definições de macros com a diretiva #define:

**#define MAX 12**

**#define PI 3.14159**

**#define MULTIPLICAR(x, y) ((x) \* (y))**

**#define QUADRADO(n) ((n) \* (n))**

O arquivo [exemplo-diretivas.c](https://drive.google.com/file/d/10S0S5whAaTZ1DGW26uA811e5r2Qa5HtF/view?usp=sharing) exemplifica de uso das diretivas #define e #warning.

### Compilação

Como já visto, um arquivo em C é compilado com o comando>

gcc nome-do-arquivo.c

Isso gera um arquivo executável, por padrão, chamado a.out. Para executá-lo em sistemas *Unix-like*, basta digitar ./a.out.

Uma forma mais completa pode reunir os comandos de compilação e execução em uma linha, como no exemplo abaixo:

gcc arquivo.c -o arquivo -lm && ./arquivo

O parâmetro -l realiza a ligação de uma biblioteca com os códigos gerados no projeto. Assim as funções da biblioteca vinculada que estiverem sendo chamadas no código serão incluídas no arquivo executável final. Ao utilizar -lm, liga-se a biblioteca matemática ao código. Em exercícios que não a usam, este parâmetro é opcional.

Há outras formas de usar o gcc. Para conhecê-las, recomenda-se a leitura da [documentação do GCC](https://gcc.gnu.org/onlinedocs/). Por exemplo, acrescentando -Werror=implicit-function-declaration no comando de compilação, trata-se como erro o aviso sobre declaração implícita de função. Inserindo apenas o parâmetro -Werror, trata-se todo *warning* como erro.

Serviço(s) *online* para programar e compilar códigos C:

* <https://www.onlinegdb.com/online_c_compiler>
* <https://www.programiz.com/c-programming/online-compiler/>

### Operador de Referência

Em C, usa-se o operador & (e comercial ou *ampersand*) para referenciar o endereço de memória de uma variável. O quadro a seguir traz um exemplo de código que utiliza o operador & (disponível também no arquivo [exemplo-operador-referencia.c](https://drive.google.com/file/d/1vzooqvxJpgRNGfWXOvjGbzw0RjTqvZwg/view?usp=drive_link)):

| **#include <stdio.h>**  **int main() {**  **double salario = 9050.17;**    **printf("%.2lf\n", salario);**  **printf("%p\n", &salario);**    **return 0;**  **}** |
| --- |

Neste exemplo, a formatação da saída foi feita com %p porque salario é um ponteiro para um endereço de memória. Ponteiro, portanto, é uma variável que armazena um endereço de memória de outra variável. Uma variável de ponteiro aponta para um tipo de dados (como int) do mesmo tipo e é criada com o operador asterisco (\*). O endereço da variável com a qual se está trabalhando é atribuído ao ponteiro. O exemplo a seguir, disponível também no arquivo [exemplo-ponteiro-e-operador-referencia.c](https://drive.google.com/file/d/1FwjnCfdFxRRiXLsnoWRwkv5iFba8fcbS/view?usp=drive_link), altera o código anterior e atribui o endereço de memória da variável salario a uma variável de ponteiro:

| **#include <stdio.h>**  **int main() {**  **double salario = 9050.17;**  **double \*ponteiroSalario = &salario;**    **printf("%.2lf\n", salario);**  **printf("%p\n", &salario);**  **printf("%p\n", ponteiroSalario);**    **return 0;**  **}** |
| --- |

No exemplo, uma variável de ponteiro foi criada (ponteiroSalario). Ela aponta para uma variável do tipo double (salario). É possível notar que o tipo de ponteiro tem que ser do mesmo tipo da variável que ele pretende apontar. O operador & foi utilizado para armazenar o endereço de memória da variável salario, sendo atribuído ao ponteiro.

Na declaração de variáveis, o operador asterisco (\*) tem a função de declarar uma variável de ponteiro. Por isso, ele também é chamado de *operador de indireção* ou *ponteiro de indireção* Ao utilizar este operador no meio do código, é dito que ele tem função de desreferência. Um exemplo de código em que o \* é utilizado como operador de desreferência está disponível no arquivo [exemplo-operador-desreferencia.c](https://drive.google.com/file/d/1U6wTmTXDVPoy5Wiq_YsQd9fDpcIIOrfZ/view?usp=drive_link) e no quadro a seguir:

| **#include <stdio.h>**  **int main() {**  **double salario = 9050.17;**  **double \*ponteiroSalario = &salario;**    **printf("%.2lf\n", salario);**  **printf("%p\n", &salario);**  **printf("%.2lf\n", \*ponteiroSalario);**    **return 0;**  **}** |
| --- |

## Conclusão

Este material abordou os conceitos fundamentais do paradigma procedural e da linguagem de programação C. No paradigma procedural, as funções são utilizadas como blocos de código reutilizáveis para decompor problemas maiores em subproblemas, facilitando a modularização e a organização do código.

A linguagem de programação C é um exemplo desse paradigma e é amplamente utilizada em diversas aplicações. C oferece uma variedade de recursos, incluindo tipos de dados básicos, estruturas de controle como decisões e repetições, além de suporte para ponteiros, *arrays*, matrizes e funções. O texto também apresenta exemplos de código em C para demonstrar os recursos da linguagem.

## Exercícios

1. O programa deve ser feito de forma descendente, com a decomposição do problema inicial em módulos, de modo a dividir as ações complexas em uma sequência de ações mais simples. Essa técnica de programação é chamada de programação:
   1. Abstrata.
   2. Interna.
   3. Declarativa.
   4. Sequencial.
   5. Modular
2. Considere o código em Linguagem C, abaixo. Suponha que ele está no corpo de uma função que foi chamada:

| **int x = 3, y = 1, z = -5, w = -2;**  **int i;**  **for (i = 1; i < x; i++) {**  **while (z < 0) {**  **++i;**  **z++;**  **do {**  **x = x + 3;**  **w++;**  **} while (w < 0);**  **}**  **}**  **printf("%d %d %d %d", x, y, z, w);** |
| --- |

Assinale a alternativa que corresponde à saída impressa na tela.

* 1. 21 1 0 4
  2. 3 1 -5 -2
  3. 2 0 -4 -1
  4. 9 0 -3 -2
  5. 1 0 9 2

1. Considere as afirmações abaixo sobre comandos em linguagens de programação:
2. Uma declaração de variável associa um nome a um valor que, geralmente, não pode ser alterado durante a execução do programa.
3. Expressões aritméticas são expressões cujos resultados são valores numéricos, inteiros ou fracionários.
4. Expressões lógicas são aquelas que têm como resultado um dos dois valores, verdadeiro ou falso.

Quais estão corretas?

* 1. Apenas I.
  2. Apenas II.
  3. Apenas III.
  4. Apenas II e III.
  5. I, II e III.

1. Resolva os exercícios da [Lista #2](https://docs.google.com/document/d/1fzGKMgX1UyGxDF711EK3Uh7cg1mgTyBSJ-9vsYMYzns/edit?usp=sharing).

## Referências

CHATTERJEE, Sumalya. R3DHULK. Github. Disponível em: <https://github.com/R3DHULK/C-For-Gamers>. Acesso em: 28 fev. 2024.

CHROMIUM. GitHub. Disponível em: <https://github.com/chromium/chromium/tree/main>. Acesso em: 27 fev. 2024.

FREE SOFTWARE FOUNDATION. GNU Compiler Collection (GCC). Disponível em: [https://gcc.gnu.org/](https://gcc.gnu.org). Acesso em: 28 fev. 2024.

FREE SOFTWARE FOUNDATION. GNU Debugger (GDB). Disponível em: <https://www.gnu.org/software/gdb/>. Acesso em: 28 fev. 2024.

GREGG, Chris. Compiling C Programs with GCC. Written by Chris Gregg, with modifications by Nick Troccoli. Disponível em: <https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/gcc>. Acesso em: 29 fev. 2024.

HEXSEL, Roberto A. Tipos de dados. Disponível em: <https://www.inf.ufpr.br/roberto/ci067/02_tipos.html>. Acesos em: 28 fev. 2024.

MYSQL-SERVER. GitHub. Disponível em: <https://github.com/mysql/mysql-server/tree/trunk>. Acesso em: 27 fev. 2024.

SEBESTA, Robert W. Concepts of programming languages. Pearson. 10th ed. 2012.

SQLITE. SQLite Source Repository. Disponível em: <https://sqlite.org/src/doc/trunk/README.md>. Acesso em: 27 fev. 2024.

THE APACHE SOFTWARE FOUNDATION. nuttx. GitHub. Disponível em: <https://github.com/apache/nuttx>. Acesso em: 28 fev. 2024.

1. É possível usar outros compiladores como o Clang, o CC, o TCC etc. [↑](#footnote-ref-0)
2. Comentários de uma linha são precedidos de // (duas barras), e comentários de múltiplas linhas iniciam com /\* (barra e asterisco) e terminal com \*/ (asterisco e barra). [↑](#footnote-ref-1)
3. Função que obtém o tamanho da *string*. Está definida na biblioteca string.h. [↑](#footnote-ref-2)
4. A pilha de chamadas é uma região de memória que é específica para cada *thread* de um programa. Ela é alocada na memória RAM e gerenciada pelo sistema operacional. [↑](#footnote-ref-3)