

Lógica de Programação: notas de aula

Prof. Jonatha Costa

2025

Organização

1 Bibliotecas

Biblioteca `<stdio.h>`

Bibliotecas próprias

2 Exemplos-bibliotecas

3 Comandos especiais

4 Geração de Números Pseudoaleatórios em C

Objetivo da aula

- Apresentar comandos de biblioteca `<stdio.h>`;
- Apresentar comandos especiais;
- Apresentar bibliotecas da linguagem C.

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Biblioteca <stdio.h>

A biblioteca padrão **stdio.h** do C é responsável por funções de entrada e saída (I/O), como leitura e escrita de dados, dentre outras funções (comandos).

Principais funções da Biblioteca<stdio.h>

Principais funções de entrada e saída padrão presentes em **stdio.h**:

- `printf()`: Escreve dados formatados na saída padrão (geralmente a tela).
- `scanf()`: Lê dados formatados da entrada padrão (geralmente o teclado).
- `putchar()`: Escreve um único caractere na saída padrão.
- `getchar()`: Lê um único caractere da entrada padrão.
- `puts()`: Escreve uma string na saída padrão seguida por uma nova linha.
- `gets()`: Lê uma linha de texto da entrada padrão (descontinuada nas versões mais recentes de C devido a problemas de segurança).
- `fgets()`: Lê uma string de um fluxo de entrada (utilizado como alternativa segura a `gets()`).
- `fputs()`: Escreve uma string em um fluxo de saída.

Principais funções da Biblioteca<stdio.h>

Principais funções de manipulação de arquivo presentes em **stdio.h**:

- `fopen()`: Abre um arquivo e retorna um ponteiro para o arquivo.
- `fclose()`: Fecha um arquivo aberto.
- `fread()`: Lê dados de um arquivo para a memória.
- `fwrite()`: Escreve dados da memória para um arquivo.
- `fseek()`: Move o ponteiro do arquivo para uma posição específica.
- `ftell()`: Retorna a posição atual do ponteiro do arquivo.
- `rewind()`: Move o ponteiro do arquivo para o início do arquivo.
- `fprintf()`: Escreve dados formatados em um fluxo de saída (arquivo ou outro).
- `fscanf()`: Lê dados formatados de um fluxo de entrada.
- `feof()`: Verifica se o final do arquivo foi alcançado.
- `ferror()`: Verifica se houve um erro no arquivo.
- `fflush()`: Limpa (flush) o buffer de saída de um arquivo.

Principais funções da Biblioteca<stdio.h>

- Principais funções de manipulação de caractere presentes em **stdio.h**:
 - `ungetc()`: Devolve um caractere lido de volta ao fluxo de entrada;
 - `putc()`: Escreve um caractere em um fluxo de saída;
 - `getc()`: Lê um caractere de um fluxo de entrada.
- Principais funções de erro presentes em **stdio.h**:
 - `perror()`: Imprime uma mensagem de erro para a saída padrão com base no código de erro fornecido.
 - `clearerr()`: Limpa os indicadores de erro e fim de arquivo (EOF) associados ao fluxo.
- Principal função de redirecionamento presente em **stdio.h**:
 - `freopen()`: Redireciona um fluxo de entrada ou saída (útil para redirecionar a saída padrão para um arquivo).

Organização

1 Bibliotecas

Biblioteca `<stdio.h>`

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca `math`

Biblioteca `string.h`

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Bibliotecas próprias da linguagem

A linguagem C possui várias bibliotecas padrão que fornecem funcionalidades essenciais para diversas operações, como manipulação de strings, operações matemáticas, controle de tempo, dentre outras.

- **<stdio.h>**
 - Propósito: Funções de entrada e saída padrão;
 - Principais Funções: 'printf()', 'scanf()', 'fopen()', 'fclose()', 'fread()', 'fwrite()', 'getchar()', 'putchar()';
- **<math.h>**
 - Propósito: Funções matemáticas básicas e avançadas;
 - Principais Funções: 'pow()', 'sqrt()', 'sin()', 'cos()', 'tan()', 'log()', 'exp()';
- **<string.h>**
 - Propósito: Manipulação de strings e arrays de caracteres;
 - Principais Funções: 'strlen()', 'strcpy()', 'strcat()', 'strcmp()', 'memcpy()', 'memset()';

Bibliotecas próprias da linguagem

- **<stdlib.h>**
 - Propósito: Funções utilitárias, alocação de memória, controle de processos, conversões de variáveis e geração de números aleatórios;
 - Principais Funções: 'malloc()', 'free()', 'exit()', 'atoi()', 'rand()', 'srand()', 'system()';
- **<ctype.h>**
 - Propósito: Manipulação de caracteres (como verificação de tipos de caracteres e conversões);
 - Principais Funções: 'isalpha()', 'isdigit()', 'isspace()', 'toupper()', 'tolower()';
- **<time.h>**
 - Propósito: Manipulação de tempo e data;
 - Principais Funções: ** 'time()', 'clock()', 'difftime()', 'strftime()', 'mktime()';
- **<limits.h>**
 - Propósito: Define constantes relacionadas aos limites dos tipos de dados primitivos;
 - Exemplos: 'INT_MAX', 'CHAR_MIN', 'LONG_MAX';

Bibliotecas próprias da linguagem

- **<float.h>**
 - Propósito: Define constantes relacionadas aos limites dos tipos de dados de ponto flutuante;
 - Exemplos: 'FLT_MAX', 'DBL_MIN', 'LDBL_EPSILON';
- **<stdbool.h>**
 - Propósito: Define o tipo 'bool' para representar valores booleanos ('true' e 'false');
 - Principais Macros: 'true', 'false';
- **<stddef.h>**
 - Propósito: Define tipos e macros comuns, como 'size_t', 'ptrdiff_t', 'NULL';
 - Principais Definições: 'NULL', 'offsetof()', 'size_t';
- **<stdint.h>**
 - Propósito: Define tipos inteiros de tamanho fixo, como 'int8_t', 'int16_t', 'uint32_t';
 - Exemplos: 'int8_t', 'uint16_t', 'int32_t', 'uint64_t';

Bibliotecas próprias da linguagem

- **<errno.h>**
 - Propósito: Manipulação de códigos de erro retornados por funções do sistema;
 - Principais Definições: 'errno', 'EDOM', 'ERANGE', 'EFAULT';
- **<assert.h>**
 - Propósito: Fornece a macro 'assert()' para fazer verificações em tempo de execução, normalmente usada para depuração;
 - Principal Função: 'assert()';
- **<signal.h>**
 - Propósito: Manipulação de sinais, que são notificações que um processo pode receber de outras partes do sistema operacional.
 - Principais Funções: 'signal()', 'raise()', 'sigaction()'.
- **<locale.h>**
 - Propósito: Manipulação de localidade, permitindo ajustar o comportamento de funções para diferentes regiões geográficas;
 - Principais Funções: 'setlocale()', 'localeconv()';

Bibliotecas próprias da linguagem

- **<setjmp.h>**
 - Propósito: Fornece suporte para saltos não locais no código, permitindo pular entre diferentes partes do código (normalmente usado em tratamentos de erro);
 - Principais Funções: 'setjmp()', 'longjmp()';
- **<stdarg.h>**
 - Propósito: Manipulação de listas de argumentos de tamanho variável em funções;
 - Principais Funções: 'va_start()', 'va_arg()', 'va_end()';
- **<complex.h>**
 - Propósito: Fornece suporte para operações com números complexos (adicionado no padrão C99);
 - Principais Funções: 'cabs()', 'creal()', 'cimag()', 'cexp()';
- **<tgmath.h>**
 - Propósito: Proporciona macros genéricas que funcionam com números inteiros, de ponto flutuante e complexos (adicionado no padrão C99);
 - Principais Funções: Macros genéricas para operações matemáticas, como 'tgamma()'.

Organização

1 Bibliotecas

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

4 Geração de Números Pseudoaleatórios em C

Organização

1 Bibliotecas

Biblioteca `<stdio.h>`

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca `math`

Biblioteca `string.h`

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Exemplo de aplicação de `<math.h>`

```
1  #include <stdio.h>
2  #include <math.h>
3  int main() {
4      int num,pot;
5      printf("Digite um número: "); scanf("%d",&num);
6      printf("Digite um valor de potência para o número: "); scanf("%d",&pot);
7      // Função pow(): Calcula base elevada ao expoente
8      double potencia = pow(num, pot);
9      printf("%.2f elevado a %.2f é: %.2f/n", num, pot, potencia);
10     // Função sqrt(): Calcula a raiz quadrada de um número double
11     raiz = sqrt(num);
12     printf("A raiz quadrada de %.2f é: %.2f /n", num, raiz);
13     return 0;
14 }
```

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Exemplo de aplicação de `<string.h>`

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

3  struct Estudante {
4      char nome[50];
5      int idade;
6      float nota; };
7  int main() {
8      struct Estudante aluno;
9      strcpy(aluno.nome, "Ana"); //“strcpy” escreve caracteres no struct
10     aluno.idade = 20; aluno.nota = 8.5;
11     printf("Nome: %s /n", aluno.nome);
12     printf("Idade: %d /n", aluno.idade);
13     printf("Nota: %.2f /n", aluno.nota);
14     return 0;
15 }
```

Exemplos de códigos executáveis

- Veja a lista de códigos em: <https://github.com/jonathacosta/PL>

Organização

1 Bibliotecas

2 Exemplos-bibliotecas

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Comandos especiais para compactação de códigos em C

- **#define** - A diretiva `#define` é utilizada para criar macros, que são substituições de texto, podendo serem utilizadas para definir constantes ou expressões;
- **#undef** - A diretiva `#undef` é utilizada para desfazer a definição de uma macro feita com `#define`;
- **operador ternário** - (`condição ? expressão1 : expressão2;`)
condição: Uma expressão que será avaliada como verdadeira (não-zero) ou falsa (zero).
expressão1: Será executada se a condição for verdadeira.
expressão2: Será executada se a condição for falsa.

Exemplo de aplicação de #DEF

```
1  #include <stdio.h>
2  #define PI 3.14159
3  #define QUADRADO(x) ((x) * (x))

4  int main() {
5      double raio = 5.0;
6      double area = PI * QUADRADO(raio);
7      printf("Raio: %.2f", raio);
8      printf("Área do círculo: %.2f", area);
9      return 0;
10 }
```

Código Conciso:

Versão 01

```
1  #include <stdio.h>
2  int main() {
3  int x=3, y;
4  if (x<5){y=11;}
5  else{y=10;}
6  printf("%d",y);
7  return 0;
8  }
```

Versão 02

```
1  #include <stdio.h>
2  int main() {
3  int x=3, y;
4  y=(x<5)?11:10;
5  printf("%d",y);
6  return 0;
7  }
```

Versão 03

```
1  #include <stdio.h>
2  int main() {
3  int x=3;
4  printf("%d",(x<5)?11:10);
5  return 0;
6  }
```

Preferências?

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Problema no Código?

Considere o *script* a seguir e analise se há erros!

Código contém erros?

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 5;
5     char b;
6     b = (a > 7) ? "Aprovado!": "Não aprovado!";
7     printf("%c", b);
8     return 0;
9 }
```

Problema no Código?

Considere o *script* a seguir e analise se há erros!

Código contém erros?

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 5;
5     char b;
6     b = (a > 7) ? "Aprovado!": "Não aprovado!";
7     printf("%c", b);
8     return 0;
9 }
```

Erro:

- `b` é `char`, mas o ternário retorna strings.
- `%c` espera um único caractere, causando erro.

Solução 1

Código Corrigido

```
#include <stdio.h>
int main()
{
    int a = 5;
    char *b;
    b = (a > 7) ? "Aprovado!": "Não aprovado!";
    printf("%s", b);
    return 0;
}
```

Mudanças:

- **b** alterado para um **ponteiro char ***
- Utilizando **%s** no **printf** para imprimir strings.

O que é um Ponteiro?

Definição:

- Um ponteiro em C é uma variável que armazena o endereço de memória de outra variável. Em vez de armazenar um valor diretamente, ele aponta para a localização na memória onde o valor está armazenado.

Exemplo de ponteiro

```
int x = 10; // Variável inteira  
int *p = &x; // Ponteiro que armazena o endereço de x
```

p é declarado como `int *`, ou seja, um ponteiro para um inteiro.

O operador `&` (endereço de) é utilizado para obter o endereço de x , que é atribuído ao ponteiro p .

O operador `*` é usado para acessar o valor armazenado no endereço ao qual o ponteiro aponta. Por exemplo:

```
printf("%d", *p); // Imprime o valor de x, ou seja, 10.
```

Ponteiro com *string*?

Definição:

- Uma string é, na verdade, um *array* de caracteres.
Seja o trecho de código:
`char *str = "Hello"; // str armazena o endereço do 'H'`
- O ponteiro `char` pode apontar para:
 - ① O primeiro caractere da string:
`printf("%c", *str); // Imprime 'H'`
 - ② Um caractere de interesse do programador:
`printf("%c", *(str + 1)); // Imprime 'e'`
 - ③ A string completa:
`printf("%s", str); // Imprime 'Hello'`

Código Corrigido com Ponteiro: Solução 01

Código corrigido com ponteiro

```
#include <stdio.h>
int main()
{
    int a = 5;
    char *b;
    b = (a > 7) ? "Aprovado": "Não aprovado";
    printf("%s", b);
}
```

Como funciona no compilador?

- 1 As *strings* literais "Aprovado" e "Não Aprovado" são armazenadas em uma área especial da memória, chamada seção de texto ou seção de dados constantes.
- 2 Essas strings são armazenadas de forma estática, ou seja, são criadas no momento da compilação e permanecem na memória durante a execução do programa.
- 3 O ponteiro *b* então recebe o endereço da *string* "Aprovado" (no caso da condição ser verdadeira). O endereço de memória dessa *string* é atribuído a *b*, e o ponteiro pode ser usado para acessar o conteúdo dessa *string*.
- 4 Quando a função `printf("%s", b)` é chamada, ela imprime o conteúdo da *string* que o ponteiro *b* aponta. Como *b* aponta para a string "Aprovado", o comando `printf` imprimirá "Aprovado".

Código corrigido com Ponteiro: Solução 02

Código Corrigido sem declarar o ponteiro

```
#include <stdio.h>
int main()
{
    int a = 5;
    printf("%s", (a > 7) ? "Aprovado":
    "Não aprovado");
    return 0; }
```

Como funciona no compilador?

- 1 As *strings* literais, como "Aprovado!" e "Não aprovado!", são tratadas pelo compilador como *arrays* de caracteres constantes, mas, **internamente**, elas são representadas como **ponteiros** para o primeiro caractere da *string*.
- 2 Quando o compilador encontra "Aprovado!", ele sabe que isso é um ponteiro para o primeiro caractere da *string* ('A'), e esse ponteiro é o que é utilizado quando a *string* é passada para funções como *printf*.
- 3 A função *printf* espera um argumento do tipo *char** para o especificador de formato *%s*, que indica uma *string*. A expressão ternária $(a > 7) ? \text{"Aprovado!"} : \text{"Não aprovado!"}$ retorna diretamente um ponteiro para a *string* apropriada.

Portanto, não é necessário declarar um ponteiro adicional, porque o tipo de "Aprovado!" ou "Não aprovado!" já é *char**, ou seja, um ponteiro para o primeiro caractere da *string*. Quando a expressão ternária é avaliada, ela simplesmente retorna esse ponteiro, que é então passado para o *printf*.

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

Exercícios de ternários e ponteiros: bloco 1

- ❶ **Operador Ternário com Números:** Escreva um programa que utilize o operador ternário para verificar se um número é positivo, negativo ou zero. O programa deve imprimir a mensagem correspondente:
- Se o número for positivo, deve imprimir “Número positivo”.
 - Se o número for negativo, deve imprimir “Número negativo”.
 - Se o número for zero, deve imprimir “Número zero”.

- ❷ **Uso de `#define` para Definir Constantes:** Utilize a diretiva `#define` para definir uma constante para o valor de PI e calcule a área de um círculo de raio 5. A fórmula para calcular a área de um círculo é:

$$A = \pi \times r^2$$

Em que `r` é o raio do círculo.

- ❸ **Ponteiro para String:** Implemente um programa que utilize um ponteiro para armazenar e imprimir uma string. O programa deve armazenar a string “Bem-vindo ao C!” em um ponteiro de caractere e imprimi-la utilizando `printf`.
- ❹ **Operador Ternário com Ponteiros:** Escreva um programa que, utilizando o operador ternário, decida qual das duas variáveis ponteiro `ptr1` ou `ptr2` deve ser utilizada com base no valor de um número inteiro `a`. Se `a > 10`, o programa deve usar `ptr1`, caso contrário, `ptr2`. Ambas as variáveis ponteiro devem apontar para um valor inteiro.

Exercícios de ternários e ponteiros: bloco 2

- 5 **Uso de `#define` para Função de Cálculo:** Utilizando `#define`, crie uma macro chamada `SQUARE(x)` que calcula o quadrado de um número `x`. Use essa macro para calcular o quadrado de um número inserido pelo usuário e imprima o resultado.
- 6 **Uso de Ponteiros para Funções:** Escreva uma função que receba um ponteiro para um número inteiro e altere seu valor para 100. No programa principal, crie uma variável inteira, passe seu ponteiro para a função e imprima o valor alterado.
- 7 **Operador Ternário e Arrays:** Dado um array de inteiros, escreva um programa que utilize o operador ternário para verificar se o primeiro elemento é maior que 10. Se for, imprima "Maior que 10", caso contrário, imprima "Menor ou igual a 10".
- 8 **Manipulação de Ponteiros em Arrays:** Crie um programa que utilize ponteiros para manipular um array de inteiros. O programa deve imprimir os elementos do array, acessando-os através de ponteiros.
- 9 **Estrutura com Ponteiros:** Defina uma estrutura chamada `Pessoa` com os campos `nome` e `idade`. Crie um ponteiro para uma variável do tipo `Pessoa`, atribua valores a esses campos e imprima as informações.
- 10 **Uso de `#define` para Definir Tipos:** Utilize a diretiva `#define` para criar um tipo de dado `float32`, que seja equivalente a `float`. Em seguida, crie uma variável desse tipo e imprima seu valor.

Organização

- 1 Bibliotecas
- 2 Exemplos-bibliotecas
- 3 Comandos especiais
- 4 Geração de Números Pseudoaleatórios em C
 - Números aleatórios
 - Gerador Linear Congruente

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

O que são Números Aleatórios?

- Números aleatórios são valores gerados sem padrão aparente;
- Computadores geram na prática números **pseudoaleatórios**, utilizando algoritmos determinísticos;
- A geração depende de um ponto de partida - uma **semente** (*seed*) - que inicializa a sequência de números;
- Uma função muito comum nesse contexto é a função *rand()*, que gera números pseudoaleatórios.

Principais Aplicações dos Números Aleatórios

- **Simulações e Modelagem:** Exemplo — métodos Monte Carlo para previsão financeira e física.
- **Jogos:** Decisão imprevisível de adversários, geração de mapas, sorteios.
- **Segurança e Criptografia:** Geração de chaves, senhas e números para protocolos seguros.
- **Testes de Software:** Criação de dados de teste variados para validar comportamento.
- **Algoritmos Probabilísticos:** Otimização, aprendizado de máquina e inteligência artificial.

Exemplos Práticos em C: código não ajustado

- Geração de um número aleatório entre 0 e 9:

```
#include <stdio.h>          // Biblioteca de basilar de input/output
#include <stdlib.h>         // Biblioteca do rand()
int main() {
    int num = rand() % 10;  // sintaxe para gerar valor entre 0 a 9
    printf("Valor gerado: %d\n", num);
    return 0;
}
```


Exemplos Práticos em C: código não ajustado

- Geração de um número aleatório entre 0 e 9:

```
#include <stdio.h>          // Biblioteca de basilar de input/output
#include <stdlib.h>         // Biblioteca do rand()
int main() {
    int num = rand() % 10;  // sintaxe para gerar valor entre 0 a 9
    printf("Valor gerado: %d\n", num);
    return 0;
}
```

Proposição inicial:

Qual o valor gerado pelo código acima?

Os valores diferentes a cada chamada?

Quais as considerações aplicáveis?

Como funciona a função `rand()`

- A função `rand()` gera números inteiros **pseudoaleatórios**.
- São chamados assim porque, apesar de parecerem aleatórios, seguem uma lógica **determinística**.
- A sequência gerada depende de uma **semente inicial** (seed), definida com `srand(seed)`.
- Se a mesma semente for usada, a sequência será exatamente a mesma.
- Internamente, a maioria das bibliotecas C implementa `rand()` por meio de um **Gerador Linear Congruente (LCG)**.

O que é esse Gerador Linear Congruente?

Organização

1 Bibliotecas

Biblioteca <stdio.h>

Bibliotecas próprias

2 Exemplos-bibliotecas

Biblioteca math

Biblioteca string.h

3 Comandos especiais

Um aparente erro e solução

Exercícios com ternários e ponteiros

4 Geração de Números Pseudoaleatórios em C

Números aleatórios

Gerador Linear Congruente

O que a função `rand()` executa?

A função `rand()` é amplamente utilizada para gerar números com aparência aleatória. No entanto, ela não se baseia em fenômenos físicos imprevisíveis, mas sim em um processo **determinístico** e repetível.

Esse processo é implementado por algoritmos matemáticos que produzem sequências de números denominadas **pseudoaleatórias**. Um dos algoritmos mais comuns para isso é o **Gerador Linear Congruente (LCG)**.

Entender o funcionamento interno do LCG nos ajuda a compreender as limitações e a **confiabilidade** da função `rand()`.

Como funciona o Gerador Linear Congruente (LCG)

O Gerador Linear Congruente (LCG) é um algoritmo simples e eficiente para gerar números pseudoaleatórios. Ele calcula o próximo valor da sequência com base na fórmula:

$$X_{n+1} = (a \times X_n + c) \bmod m$$

- X_n : valor atual da sequência (chamado de estado interno).
- a : constante multiplicadora — influencia a distribuição dos números.
- c : incremento — evita ciclos curtos e padrões repetitivos.
- m : módulo — define o intervalo total dos valores possíveis.
- X_0 : semente inicial, definida por `srand()` — determina o ponto de partida da sequência.

O LCG é amplamente adotado por seu bom desempenho e facilidade de implementação em diversas linguagens e plataformas.

Exemplo do GLC com módulo $m = 2^{32}$ e resultado após mod6

Parâmetros utilizando: $a = 1664525$, $c = 1013904223$, $X_0 = 1$, $m = 2^{32}$.

Iteração n	$X_n = (aX_{n-1} + c) \bmod 2^{32}$	$X_n \bmod 6$
1	1015568748	2
2	1586005467	5
3	2165702440	4
4	3577892423	5
5	1928097746	2
6	2671302351	1
7	3047009734	0
8	3055605773	3
9	2846464776	4
10	1734378495	1

Note que, mesmo utilizando um módulo muito grande no gerador, a operação mod 6 (isto é, o operador % 6 em linguagens de programação) atua como uma forma de restringir os valores gerados ao intervalo $[0, 5]$. Isso ocorre porque a expressão $X_n \bmod 6$ retorna o **resto da divisão inteira de X_n por 6**. Assim, mesmo que X_n seja um número grande, o valor resultante da operação será sempre um número entre 0 e 5. Essa técnica é útil para produzir números aleatórios dentro de uma faixa limitada, mantendo a distribuição pseudoaleatória da sequência original.

Por que escolher esses valores no LCG?

Os parâmetros utilizando no LCG não são arbitrários — eles seguem critérios matemáticos que garantem a qualidade da sequência gerada. Um conjunto comum de parâmetros é:

- $a = 1664525$:
 - Tradicionalmente utilizado em bibliotecas padrão (como GCC e Visual C++).
 - Garante boa dispersão dos bits na sequência.
- $c = 1013904223$:
 - Satisfaz a condição de Hull-Dobell para máxima periodicidade.¹
 - Ajuda a evitar padrões e ciclos curtos.
- $m = 2^{32}$:
 - Tamanho ideal para sistemas com inteiros de 32 bits (`unsigned int`).
 - Facilita cálculos modulares com desempenho otimizado.

Esses valores são testados e amplamente utilizados por fornecerem boa aleatoriedade em contextos gerais.

¹A condição de Hull-Dobell estabelece que o gerador linear congruente atinge o período máximo m se, e somente se: (1) c e m são primos entre si; (2) $a - 1$ é divisível por todos os fatores primos de m ; e (3) se m é múltiplo de 4, então $a - 1$ também deve ser múltiplo de 4.

A importância da função `srand(seed)`

Para que a sequência de números pseudoaleatórios varie a cada execução, é necessário definir uma **semente inicial** personalizada.

- `srand(seed)` inicializa o estado interno do LCG com o valor `seed`.
- Se `srand()` não for chamada, a semente padrão (geralmente 1) é utilizada.
- Resultado: a sequência gerada será sempre a mesma em todas as execuções.

Para maior variabilidade, é comum utilizar o tempo atual como semente:

```
srand(time(NULL));
```

Assim, a cada execução, o ponto de partida muda, o que torna a sequência mais próxima do comportamento aleatório desejado.

Por que é chamado **pseudoaleatório**?

- Sequência gerada por fórmula matemática determinística.
- Sequência depende da semente inicial.
- Repetível: mesma semente gera a mesma sequência.
- Rápido e útil para simulações e jogos.
- Não seguro para aplicações criptográficas!

Exemplos Práticos em C - Código ajustado!

- Geração de um número aleatório entre 0 e 9:

```
#include <stdio.h>           // Biblioteca de basilar de input/output
#include <stdlib.h>           // Biblioteca do rand()
#include <time.h>             // Biblioteca de tempo

int main() {
    srand(time(NULL));        // Inicia semente
    int num = rand() % 10;    // sintaxe para gerar valor entre 0 a 9
    printf("Valor gerado: %d\n", num);
    return 0;
}
```

Resumo

- `rand()` gera números pseudoaleatórios.
- Algoritmo base: Gerador Linear Congruente (LCG).
- Semente (via `srand()`) define a sequência.
- Usar `srand(time(NULL))` para variar sequência.
- Adequado para usos gerais, mas não para segurança.

Exercícios de Fixação — Utilização de `rand()`

Resolva os exercícios abaixo utilizando a função `rand()` em linguagem C:

- 1 Gere e imprima 10 números aleatórios entre 1 e 100.
- 2 Simule o lançamento de um dado (números entre 1 e 6) e exiba o resultado.
- 3 Crie um vetor com 20 posições e preencha com números aleatórios entre 0 e 9.
- 4 Simule uma moeda (cara ou coroa) utilizando o `rand()` e repita a simulação 50 vezes, contando o número de ocorrências de cada lado.
- 5 Simule 3 partidas entre Ceará e Fortaleza. Para cada jogo, gere dois números aleatórios (0 a 5), representando os gols de cada time, e exiba o resultado.

Dica: Utilize `rand() % intervalo` para limitar aos valores de interesse.

Exercícios

- **Veja material auxiliar em:** <https://github.com/jonathacosta-IA/PL>
 - *Slides* em: https://github.com/JonathaCosta-IA/PL/tree/main/A-PL_Slides
 - *Códigos* em: https://github.com/JonathaCosta-IA/PL/tree/main/B_PL_Codes