

Aula 05: Lógica de programação com a Linguagem C



por João Luiz Borges Morais

Introdução ao C

Nesta aula iremos nos aprofundar na **lógica de programação junto a linguagem C**, no qual possui uma **sintaxe um pouco menos amigável do que o Python** abordado na nossa terceira aula pois ela se trata de uma **linguagem de baixo nível**, ou seja, uma linguagem **mais próxima da linguagem de máquina**.

Porem, a linguagem de programação C se destaca por ser uma **linguagem estruturada de propósito geral**, frequentemente considerada a "**mãe**" de muitas outras linguagens, como C++, Java, e C#. É conhecida por ser **versátil, permitindo tanto o desenvolvimento de sistemas de baixo nível** (como drivers e sistemas operacionais) **quanto aplicações de alto nível** (como processamento de texto e manipulação de dados)

```
#include <stdio.h>
```

```
int main(void) {
```

```
    printf("Hello, World!\n");
```

```
}
```

História

A linguagem C foi criada em **1972** por **Dennis Ritchie**, nos **Laboratórios Bell**, como uma evolução da linguagem **B**, que por sua vez veio da **BCPL**. Seu principal objetivo era reescrever o sistema operacional **Unix**, que antes era feito em Assembly. A reescrita do Unix em C marcou um avanço importante, pois tornou o sistema mais portátil entre diferentes computadores.

Em **1978**, o livro "**The C Programming Language**", de **Kernighan e Ritchie**, ajudou a popularizar a linguagem. Esse livro ficou conhecido como **K&R C**. Mais tarde, em **1989**, o padrão **ANSI C** foi criado para unificar a linguagem. Ele foi seguido por outras versões, como **C99**, **C11** e **C17**, cada uma trazendo melhorias.

SECOND EDITION

THE



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

```
40
41
42 @classmethod
43     def from_settings(cls, settings):
44         debug = settings.getbool('DEBUG', False)
45         return cls(job_dir(settings), debug)
46
47     def request_seen(self, request):
48         self.request_fingerprint(request)
```

Qualidades necessárias ao escrever código

Ao escrever código, esperamos que uma pessoa utilizará um software baseado neste código, ela espera que este software possa resolver seus problemas e otimizar sua rotina. Se o software apresentar muitos problemas, a pessoa tende a perder a confiança nele e optar por um concorrente, podemos considerar as seguintes qualidades ao escrevê-lo:

Correção

Se nosso código funciona corretamente, conforme o planejado.

Design

Uma medida subjetiva de quão bem escrito nosso código é, com base em quão eficiente, elegante ou logicamente legível ele é, sem repetição desnecessária.

Estilo

Quão esteticamente formatado nosso código é, em termos de indentação consistente e outra colocação de símbolos.

As diferenças de estilo não afetam a exatidão ou o significado do nosso código, mas afetam o quão legível é visualmente

Visual Studio Code

Para começar a escrever nosso código rapidamente, usaremos uma ferramenta que será utilizada nesta aula e muitas outras diversas, o **Visual Studio Code** (VScode), um **ambiente de desenvolvimento integrado** que inclui programas e recursos para escrever código

Instalação

Windows

1. Acesse o site:
<https://code.visualstudio.com>
2. Clique em **Download for Windows** (.exe)
3. Execute o instalador:
 - Aceite os termos
 - Escolha o local de instalação
 - Marque a opção "**Add to PATH**" (recomendado)
 - Finalize clicando em **Install**
4. Após instalar, abra o VS Code pelo menu iniciar

Linux

1. Abra o terminal
2. Execute os comandos:

```
sudo apt update  
sudo apt install wget gpg  
wget -qO-  
https://packages.microsoft.com/keys/microsoft.asc |  
gpg --dearmor > microsoft.gpg  
sudo install -o root -g root -m 644 microsoft.gpg  
/etc/apt/trusted.gpg.d/  
sudo sh -c 'echo "deb [arch=amd64]  
https://packages.microsoft.com/repos/vscode stable  
main" > /etc/apt/sources.list.d/vscode.list'  
sudo apt update  
sudo apt install code
```

Após instalar, execute com:

code

Compilação

No nosso **terminal**, no painel inferior do nosso IDE, iremos **compilar nosso código antes de podemos executá-lo**. Os computadores só entendem **código binário**, que também é usado para **representar instruções** como imprimir algo na tela

Nosso código-fonte foi escrito em caracteres que podemos ler, mas precisam ser **compilados: convertido em código de máquina, padrões de zeros e uns que nosso computador passa a entender diretamente**

Um programa chamado **Compilador pegará o código fonte** como **entrada** e **produzirá o código de máquina** como **saída**

No IDE, Temos acesso ao **compilador da linguagem de programação C** chamado **GCC**, por meio de um **comando** chamado **GCC arquivo.c**

- Este comando possui outra alternativa chamada **GCC arquivo.c output/arquivo**, onde chamamos o **compilador GCC, depois chamamos o arquivo que queremos compilar e o nome do nosso arquivo compilado junto com o caminho onde ele irá ficar**, que nesse caso dentro de uma pasta chamada **output**.

Depois disso, digite o comando **./arquivo** para executá o **arquivo com seu código já compilado**

Funções e Argumentos

Funções são pequenas ações ou verbos que podemos usar em nosso programa para fazer algo, e as **entradas para funções** são chamadas de **argumentos**.

- Por exemplo: Em C, a função de imprimir algo na tela é chamada de **printf** (com f significando texto formatado).

E em C **passamos os argumentos com parênteses**, como em **printf("Hello world");**

As **aspas duplas** indicam que queremos **imprimir as letras Hello world** literalmente, e o **ponto-e-vírgula no final** indica o **fim da nossa linha de código**.

Tipos de funções

As funções também podem ter dois tipos de **saída**:

- **Efeitos Colaterais:** Como algo impresso na tela
- **Valores de retorno:** Um valor que é passado de volta ao nosso programa que podemos usar ou armazenar para mais tarde

Biblioteca Cs50

Resumidamente as **Bibliotecas** são coleções de código pré-escrito (funções, classes, módulos) que **estendem as funcionalidades básicas** de uma linguagem, elas são reutilizáveis, economizam tempo e esforço e evitam reinventar a roda.

Utilizaremos durante nosso percurso uma usada para **meios acadêmicos** com o **objetivo de mostrar conceitos de forma didática** com **foco na lógica de programação**. A biblioteca cs50 foi criada como parte do curso **CS50 – "Introduction to Computer Science"** da **Universidade de Harvard**, um dos cursos de ciência da computação mais populares do mundo.

Instalação da Biblioteca CS50

Windows (usando WSL – Ubuntu)

1. Instale o WSL com Ubuntu:

- No PowerShell (como administrador):

```
wsl --install
```

2. Abra o Ubuntu no WSL e atualize os pacotes:

```
sudo apt update
```

```
sudo apt upgrade
```

3. Instale as dependências:

```
sudo apt install gcc libcs50-dev
```

4. Compile com a CS50:

```
gcc programa.c -lcs50 -o programa
```



YouTube
GET_STRING FUNCTION – CS50 ...
BIBLIOTECA CS50 no VS CODE –
Como instalar e Configurar no...

Linux (Ubuntu, Debian, Arch, etc.)

Para Ubuntu/Debian:

```
sudo apt update
```

```
sudo apt install gcc libcs50-dev
```

Para Manjaro/Arch (via AUR):

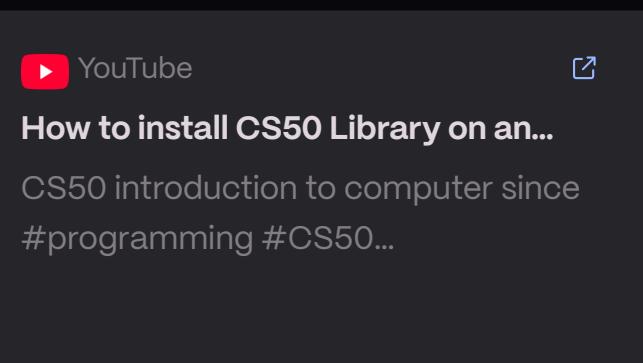
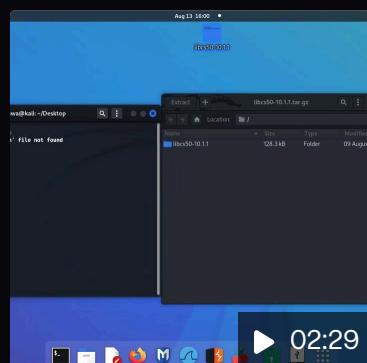
```
yay -S libcs50
```

Ou use pamac:

```
pamac build libcs50
```

Compilação:

```
gcc programa.c -lcs50 -o programa
```



Função get_string

A Biblioteca CS50 incluirá algumas **funções básicas e simples** que podemos usar imediatamente.

Por exemplo, **get_string**, pedirá ao usuário uma string, ou alguma **sequência de texto**, e o **retornará ao nosso programa**.

get_string recebe algum **input** e o usa como **prompt** para o usuário, como "**Qual seu nome?**", e nós teremos que **salvá-lo em uma variável**:

```
string answer = get_string("Qual seu nome?");
```

- Em C, o = indica **atribuição** ou **configuração do valor à direita para a variável à esquerda**. E o programa chamará a função **get_string** primeiro para então **obter seu output**.
- E também precisaremos que nossa **variável chamada answer** é do tipo é do tipo **String**, então nosso programa saberá **interpretar os zeros e uns como texto**.
- Finalmente precisaremos nos lembrar de **adicionar um ponto e vírgula** para **encerrar nossa linha de código**

Agora, iremos utilizar a função **print** para **receber a saída da nossa variável answer**:

```
printf("olá %s", answer);
```

- O **%s** é chamado de **código de formatação**, o que significa que queremos que a **função printf substitua uma variável aonde está o marcador %s**

E a variável que queremos usar é **answer**, que passamos para **printf como outro argumento**, separando do primeiro por uma **vírgula** (**printf(" ola, aswer")** iria imprimir ola, answer para sempre).

De volta ao IDE CS50, nos implementaremos o que descobrimos:

```
#include <cs50.h>
#include <stdio.h>
int main(void)
{
    string answer = get_string("Qual é o seu nome?");
    printf("Olá, %s!\n", answer);
}
```

- Precisamos dizer ao computador para **incluir a biblioteca CS50**, com `#include <cs50.h>` para que possamos usar a **função get_string**.
- Também temos a oportunidade de **escrever o código** usando um "estilo" que **favoreça intuitivamente**, já que poderíamos **nomear nossa variável de resposta com qualquer coisa** mas um **nome mais descriptivo** nos **ajudará a entender** sua **finalidade** melhor do que um nome mais curto como "a" ou "x".

Depois de salvar o arquivo, precisamos **recompilar nosso programa** com `gcc arquivo.c -lcs50 -o arquivo` pois a opção `-lcs50` diz ao compilador para "linkar" (ligar) seu programa com a **biblioteca CS50**, já que **alteramos apenas o código fonte**, mas **não o código de máquina compilado**.

Outras **linguagens ou IDEs** podem **não exigir que recompilemos manualmente nosso código depois de alterá-lo**. Mas aqui temos a **oportunidade de ter mais controle e compreensão** do que está acontecendo nos bastidores.

```
gcc arquivo.c -lcs50 -o arquivo
```

Agora, `./arquivo` **executará nosso programa** e **solicitará nosso nome** conforme o esperado.

Podemos notar que o próximo prompt é impresso imediatamente após a saída do nosso programa, como em `olá, joao luiz`

```
Qual é o seu nome? Olá, joao luiz!
```

Podemos **adicionar uma nova linha após a saída** de nosso programa, de modo que o **próximo prompt esteja em sua própria linha**, com `\n`:

```
printf("olá, %s\n", answer);
```

- `\n` é um exemplo de **sequência de escape**, ou **algum texto que na verdade, representa algum outro texto**

```
Olá, joao luiz!
```

Função principal (main) e Arquivos de cabeçalho

Em C, a **primeira linha que inicia** o que consideramos ser o **programa principal** é o **int main(void)**, sobre o que aprenderemos mais, seguida por uma **chave aberta {** e uma **chave fechada }** envolvendo **tudo o que deveria estar em nosso programa.**

```
int main(void)
```

```
{
```

```
//código que você escreverá
```

```
}
```

Arquivos de cabeçalhos que **terminam com .h** refere-se a algum outro **conjunto de código**, como uma **biblioteca**, que podemos usar no nosso programa. Nós o incluímos com linhas como **#include <stdio.h>** por exemplo, para a **biblioteca de entrada / saída padrão**, que contém a função **printf**

```
#include <stdio.h>
```

Tipos e Códigos de Formato

Existem muitos tipos de dados que podemos usar para nossas **variáveis**, que indicam ao computador que **tipo de dados eles representam**:

- **bool** → uma expressão booleana, verdadeiro ou falsa (true e false)
- **Char** → Um único caractere ASCII como a ou 2
- **double** → Um valor de vírgula flutuante com mais dígitos do que um float
- **float** → Um valor de vírgula flutuante ou número real com um valor decimal
- **int** → Inteiros até um certo tamanho ou números de bits
- **long** → Inteiros com mais bits, para que possam contar mais do que um int
- **string** → Uma linha de caracteres

E a biblioteca CS50 tem funções correspondentes para obter entrada de vários tipos:

- **get_char**
- **get_double**
- **get_float**
- **get_int**
- **get_long**
- **get_string**

Para printf também, existem diferentes marcadores de posições para cada tipo:

- **%c** → para caracteres
- **%f** → para flutuantes, duplos (float, double)
- **%i** → para ints
- **%li** → para longos (long)
- **%s** → para strings

Operadores, Limitadores, Truncamento

Existem vários **operadores matemáticos** que podemos usar também:

- `+` → para adição
- `*` → para multiplicação
- `-` → para subtração
- `/` → para divisão
- `%` → para calcular todo o resto

Faremos um novo programa, **addition.c**:

```
#include <stdio.h>
```

```
#include <cs50.h>
```

```
int main(void)
```

```
{
```

```
    int x = get_int("x: ");
```

```
    int y = get_int("y: ");
```

```
    printf("%i\n", x + y);
```

```
}
```

- Vamos incluir **arquivos de cabeçalho para as bibliotecas** que iremos usar, então vamos chamar **get_init** para **obter inteiros do usuário, armazenando-os em variáveis nomeadas x e y**.
- Em seguida em **printf**, **imprimiremos um espaço reservado** para um **inteiro %i**, seguido por uma **nova linha** com `\n`. Já que queremos **imprimir a soma de x e y**, vamos passar em `x + y` para **printf** para **substituir na string**
- Vamos salvar, executando `gcc addition.c -lcs50 -o output/addition` no terminal e depois `./output/addition` para ver nosso programa funcionando.

Se digitarmos algo que **não seja inteiro**, veremos **get_int pedindo um inteiro novamente** e se digitarmos um **numero muito grande**, como `4000000000000`, **get_int** nos alertará novamente.

Isso ocorre porque, como em muitos sistemas de computador, um **int** em qualquer IDE será de **32 bits**, que pode conter apenas certa de **quatro bilhões de valores diferentes**.

E uma vez que os **inteiros podem ser positivos ou negativos, o maior valor positivo para um int só pode ser cerca de dois bilhões**, com um **valor negativo mais baixo de cerca de dois bilhões negativos**, para um total de cerca de **quatro bilhões de valores totais**

Podemos **mudar** nosso programa para usar o tipo **long**:

```
#include <stdio.h>
```

```
#include <cs50.h>
```

```
int main(void)
```

```
{
```

```
    int x = get_long("x: ");
```

```
    int y = get_long("y: ");
```

```
    printf("%li\n", x + y);
```

```
}
```

- Agora, podemos **digitar inteiros maiores** e ver um resultado correto conforme o esperado

Variáveis e Açúcar Sintético (Boas práticas)

Vamos definir uma variável para algum valor por exemplo `int contador = 0;`

Podemos **aumentar o valor de uma variável** com `contador = contador + 1;` onde olhamos primeiro para o **lado direito, pegando o valor original do contador, adicionando 1 e em seguida, armazenando-o no lado esquerdo** (de volta ao contador, neste caso).

O C também suporta **açúcar sintético** ou **expressões abreviadas para a mesma funcionalidade.** Nesse caso, poderíamos dizer de maneira equivalente: `contador += 1;` para **adicionar um ao contador antes de armazená-lo novamente.**

Também poderíamos escrever `contador++;` e podemos aprender isso (e outros exemplos) examinando a documentação ou outras referências online.

Condições

Podemos traduzir condições, ou blocos "se", com:

```
if(x < y)
{
    printf("x é menor que y\n");
}
```

Observe que em C, usamos { e } (bem como indentação) para **indicar como as linhas do código devem ser aninhadas.**

Podemos ter **condições if e else**:

```
if(x < y)
{
    printf("x é menor que y\n");
}
else
{
    printf("x não é menor que y\n");
}
```

E até mesmo **se não, else if**:

```
if(x < y)
{
    printf("x é menor que y\n");
}
else if (x > y)
{
    printf("x é maior que y\n");
}
else if (x == y)
{
    printf("x é igual a y\n");
}
```

- Observe que para **comparar dois valores em C**, usamos **==**, **dois sinais de igual**
- E logicamente, **não precisamos de if (x == y) na condição final**, já que esse é o **único caso restante**, então podemos apenas dizer o contrário com **else**:

```
if(x < y)
{
    printf("x é menor que y\n");
}
else if (x > y)
{
    printf("x é maior que y\n");
}
else
{
    printf("x é igual a y\n");
}
```

Vamos dar uma olhada em outro exemplo, **conditions.c**:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    //Usuário entra com valor de x
    int x = get_int("x: ");
    //Usuário entra com valor de y
    int y = get_int("y: ");
    //Compara x e y
    if (x < y)
    {
        printf("x é menor que y\n");
    }
    else if (x > y)
    {
        printf("x é maior que y\n");
    }
    else
    {
        printf("x é igual a y\n");
    }
}
```

- Nós incluímos as condições que acabamos de ver, justamente com **duas "chamadas"**; ou usos, de **get_int** para **obter x e y do usuário**
- Vamos compilar e executar nosso programa para ver se ele realmente funciona conforme o planejado

Em **agree.c**, podemos **pedir ao usuário para confirmar ou negar algo**:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    //Solicita um caractere para o usuário
    char c = get_char("você concorda?");
    //Verifica se concordou
    if (c == 'S' || c == 's')
    {
        printf("concordou\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("não concordo\n");
    }
}
```

- Com **get_char**, podemos **obter um único caractere** e, como só temos um em nosso programa, parece razoável chamá-lo de c
- Usamos duas **barras verticais**, **||**, para indicar um **"ou" lógico (matemático)**, onde **qualquer uma das expressões pode ser verdadeira para que a condição seja seguida**. (&& por sua vez, indica "e" lógico, onde ambas as condições deveriam ser verdadeiras).
- E observe que usamos **dois sinais igual**, **==**, para **comparar dois valores**, bem como **aspas simples**, **' '**, para **envolver nossos valores de caracteres únicos**
- Se nenhuma das **expressões for verdadeira, nada acontecerá**, pois nosso programa **não tem um loop**.

Expressões booleanas, loops

Podemos traduzir um bloco "para sempre" no C:

```
While (true)
```

```
{
```

```
printf("Oi mundo!\n");
```

```
}
```

- A palavra chave **while (enquanto)** requer uma **condição**, então usamos **true** como a **expressão booleana** para garantir que nosso **loop seja executado para sempre**.

While dirá ao computador para **verificar se a expressão é avaliada como true (verdadeira)** e, em seguida, **executar as linhas dentro das chaves**.

Em seguida, ele **repetirá isso até que a expressão não seja mais verdadeira**. Nesse caso, **true sempre será true**, então nosso loop é um **loop infinito** ou que será executado para sempre.

Poderíamos fazer algo um **certo número de vezes**, com **while**:

```
int i = 0;
```

```
while (i < 50)
```

```
{
```

```
printf("Oi mundo!\n");
```

```
i++;
```

```
}
```

- Criamos uma **variável** **i**, e a **definimos como 0**. Então, **enquanto i é menor que 50, executamos algumas linhas de código**, incluindo **um em que adicionamos 1 a i a cada passagem** (**i++**). Dessa forma, nosso **loop acabará eventualmente, quando i atingir um valor de 50**
- Nesse caso, estamos usando a variável **i** como **contador**, mas como ela **não tem nenhum propósito adicional**, podemos simplesmente chamá-la de **i**.

Mesmo que possamos **iniciar a contagem em 1**, como demonstrado abaixo, por **convenção devemos começar em 0**:

```
int i = 1;
```

```
while (i <= 50)
```

```
{
```

```
printf("Oi mundo!\n");
```

```
i++;
```

```
}
```

Outra solução correta, mas possivelmente **menos bem projetada**, pode começar com o **contador em 50 e contar para trás**:

```
int i = 50;
```

```
while (i > 0);
```

```
{
```

```
printf("Oi mundo!\n");
```

```
i--;
```

```
}
```

- Nesse caso, a **lógica do nosso loop é mais difícil de raciocinar** sem servir nenhum propósito adicional e pode até mesmo confundir os leitores

Finalmente, mais comumente, podemos usar a palavra chave **for**:

```
int i = 0;
```

```
for (int i = 0; i < 50; i++)
```

```
{
```

```
printf("Oi mundo!\n");
```

```
}
```

- Novamente, primeiro **criamos uma variável** chamada **i** e a **definimos como 0**. Em seguida, verificamos que **i < 50** todas vez que **alcançamos o topo do loop**, antes de **executar qualquer código interno**. Se essa **expressão for verdadeira, executamos o código interno**. Finalmente, depois de executar o código inteiro, usamos **i++** para **adicionar 1 a i**, e o loop se repete.
- O **loop do tipo for é mais elegante do que o loop do tipo while** nesse caso, uma vez que **tudo relacionado ao loop está na mesma linha e somente o código que desejamos realmente executar múltiplas vezes está dentro do loop**

Observe que para muitas dessas linhas de código, como **Condições do tipo if e loops do tipo for, não colocamos um ponto e vírgula no final**

E assim que a linguagem C foi projetada, muitos anos atrás, e uma regra geral é que **apenas as linhas para ações ou verbos têm ponto e vírgula no final**.

Abstração

Podemos escrever um programa que imprime miau (meow) três vezes:

```
#include <stdio.h>

int main (void)
{
    printf("miau\n");
    printf("miau\n");
    printf("miau\n");
}
```

Poderíamos usar um loop-for, para não ter que copiar e colar tantas vezes:

```
#include <stdio.h>

int main (void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("miau\n");
        printf("miau\n");
        printf("miau\n");
    }
}
```

Podemos mover a linha `printf` para **sua própria função**, como nossa própria peça de quebra cabeça:

```
#include <stdio.h>

void miau (void)
{
    printf("miau\n");
}

int main (void)
{
    for (int i = 0; i < 3; i++)
    {
        miau();
    }
}
```

- Definimos uma função, **miau**, acima de nossa função principal (**main**)

Mas convenientemente, nossa **função principal (main)** deve ser a **primeira função em nosso programa**, então precisamos de mais algumas linhas.

```
#include <stdio.h>

void miau(void);

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        miau();
    }
}
```

- Acontece que precisamos **declarar nossa função miau primeiro** com um **protótipo**, antes de usá-lo em miau, e realmente **defini-lo depois**

O **compilador lê nosso código-fonte de cima para baixo**, então ele **precisa saber que o miau existirá posteriormente no arquivo**

Podemos até mesmo **alterar nossa função miau para obter alguma entrada, n e miau n vezes**:

```
#include <stdio.h>
```

```
void miau(int n);
```

```
int main(void)
```

```
{
```

```
    miau(3);
```

```
}
```

```
void miau(int n)
```

```
{
```

```
    for (int i = 0; i < n; i++)
    {

```

```
        miau();
    }
}
```

```
    printf("miau\n");
}
```

```
}
```

```
    printf("
```

Memória imprecisão e estouro

Nosso computador tem **memória em chips de hardware** chamados **RAM, memória de acesso aleatório**.

Nossos programas usam essa **RAM** para **armazenar dados enquanto estão em execução**, mas essa **memória é finita**

Com `imprecision.c`, podemos ver o que acontece quando usamos **valores flutuantes**:

```
#include <cs50.h>
```

```
#include<stdio.h>
```

```
int main(void)
```

{

```
float x = get_float("x: ");
```

```
float y = get_float("y: ");
```

```
printf("%..50f\n", x/y);
```

}

- Com `%.50f`, podemos especificar o número de casas decimais específicas.

hmmmmmm, agora nós temos:

x: 1

y: 10

- Acontece que, isso é chamado de **imprecisão de vírgula flutuante**, em que **não temos bits suficientes para armazenar todos os valores possíveis**.

Com um número **finito de bits para um float**, não podemos representar todos os números reais possíveis (dos quais existe um número infinito de), então o computador **tem que armazenar o valor mais próximo que puder**.

E isso pode levar a problemas em que mesmo pequenas diferenças no valor de soma, a menos que o programador use alguma outra maneira para representar os valores decimais com precisão necessária

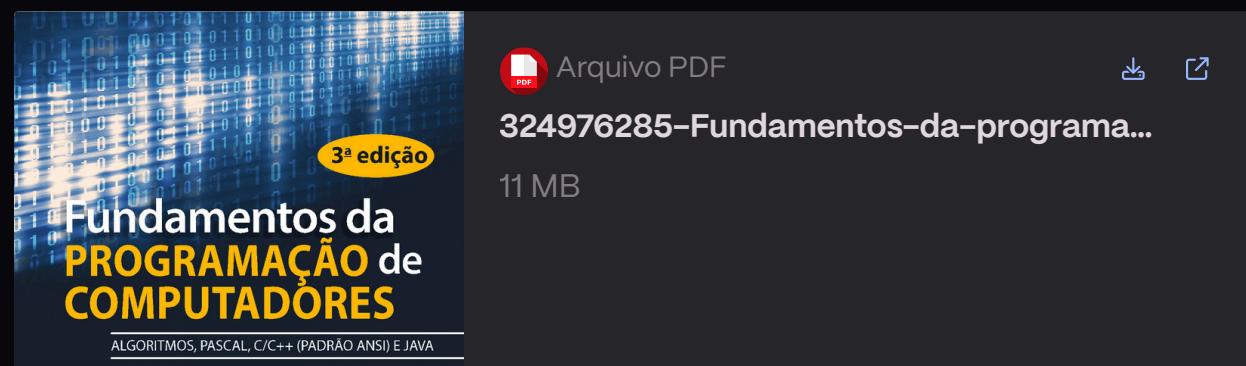
Na nossa primeira aula, quando tínhamos **três bits e precisávamos catar mais do que sete (ou 111), adicionamos outro bit para oito, 1000**. Mas se tivéssemos apenas três bits disponíveis, não teríamos lugar para o 1 extra. Ele desapareceria e estaríamos de volta ao 000.

Esse problema é chamado de **overflow (vazamento) de inteiro**, pois um inteiro só pode atingir um tamanho específico antes de ficar sem bits.

Livro da Semana

Fundamentos da Programação de Computadores

- **Essência:** Introduz a lógica de programação através de **algoritmos** (descrição narrativa, fluxograma, pseudocódigo) e conceitos de **variáveis e tipos de dados** (numéricos, caracteres, lógicos, string).
- **Estruturas Chave:** Aborda **testes condicionais** (`if`, `else`, `switch`) para decisões e **estruturas de repetição** (`for`, `while`, `do-while`) para tarefas repetitivas, além de **funções/sub-rotinas** para modularização do código.
- **Organização de Dados:** Explora o uso de **vetores (arrays)** para armazenar coleções de dados.
- **Linguagens Foco:** Apresenta exemplos práticos e conceitos em **Pascal, C, C++ e Java**, destacando suas características e aplicações para construir uma base sólida no universo da programação.



Ana Fernanda Gomes Ascencio
Edilene Aparecida Veneruchi de Campos

Fundamentos da PROGRAMAÇÃO de COMPUTADORES

ALGORITMOS, PASCAL, C/C++ (PADRÃO ANSI)

arson

Obrigado por assistir!

Até Próxima aula

Professor: João Luiz B. Morais