
NOTAS DE AULA

INTRODUÇÃO À PROGRAMAÇÃO COM C

16 de abril de 2025

Prof. Maurício Massaru Arimoto
UENP

Sumário

1	Primeiros Passos	3
1.1	Estrutura de um Programa em C	3
1.1.1	Inclusão de Bibliotecas	3
1.1.2	Função Main()	3
1.2	Indentação	4
1.3	Comentários	4
1.4	Compilando o Programa	5
1.5	Ambientes de Desenvolvimento Integrado	6
1.6	Depurando o Programa	7
2	Conceitos Básicos da Linguagem	12
2.1	Variáveis	12
2.1.1	Declaração de Variáveis	13
2.1.2	Tipos de Dados	14
2.1.3	Modificadores de Tipo	15
2.1.4	Identificadores de Variáveis	16
2.1.5	Constantes	17
2.2	Entrada de Dados	18
2.2.1	Função scanf	18
2.3	Expressões Aritméticas	19
2.4	Operadores Relacionais e Lógicos	20
2.5	Atribuição	22
2.6	Saída de Dados	24
2.6.1	Função printf	24
2.7	Exercícios Resolvidos	27
2.8	Exercícios Propostos	29

1 PRIMEIROS PASSOS

1.1 Estrutura de um Programa em C

Todo programa escrito em linguagem C deve seguir a estrutura geral da linguagem, conforme exemplificado na listagem do Código 1.

```
1 # include <stdio.h> //biblioteca
2 int main() //função principal
3 { //inicia o corpo da funcao
4     ...
5     printf("Hello world!");
6     ...
7     return 0; //função main retorna um inteiro
8 } //termina corpo da função
```

Código 1: Estrutura geral de um programa em C

A estrutura geral da linguagem C é discutida brevemente nas seções subsequentes.

1.1.1 Inclusão de Bibliotecas

O comando `#include` permite declarar bibliotecas predefinidas que serão utilizadas pelo programa. Por exemplo, a linha abaixo incluir a biblioteca `<stdio.h>`, uma biblioteca padrão em C para funções de leitura (entrada) e escrita (saída) de dados.

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

O nome da biblioteca é bem sugestivo `<stdio.h>` (i/o input/output – entrada e saída). Nela, estão implementadas por exemplo, as funções: `printf(...)` que faz a saída do programa, ou seja, exibe algo para o usuário; e `scanf(...)` que faz a leitura dos dados, ou seja, recebe a entrada do teclado.

1.1.2 Função Main()

Um programa é composto por um conjunto de instruções, e a primeira coisa que um computador vai “perguntar” é qual instrução será executada primeiro. A linguagem C define a função `main()` que sempre será executada primeiro:

```
int main(){...}
```

A função `main()` deve existir em algum lugar do programa e marca o ponto de início de execução do programa. Os caracteres “{” e “}” delimitam o escopo da função, ou seja, tudo que estiver dentro faz parte da função principal. Ela retorna um valor inteiro (`int`), neste caso, o valor ‘0’, indicando que tudo ocorreu bem, sem nenhum erro. Em computadores antigos (16 bits) usava-se `void`, que será tratado em seções subsequentes.

```
return(0);
```

1.2 Indentação

A indentação é o espaçamento (ou tabulação) colocado antes de começar a escrever o código na linha. Ela tem como objetivo indicar a hierarquia dos elementos. No nosso exemplo, os comandos `printf()`, `system()` e `return` possuem a mesma hierarquia (portanto, o mesmo espaçamento) e estão todos contidos dentro do comando `main()` (daí o porquê do espaçamento).

1.3 Comentários

Comentários são recursos oferecidos pelas linguagens que permitem, por exemplo, a inclusão de esclarecimentos do que foi feito no algoritmo e como foi feito. Dessa forma, eles servem para documentar e facilitar o entendimento do algoritmo. Os comentários são identificados e delimitados por símbolos especiais. Na linguagem C, os comentários podem ser adicionados de duas maneiras distintas. Para comentar uma única linha do código, basta adicionar barra dupla invertida (`//`) na frente da linha. O que vier após o `//` será considerado comentário e ignorado pelo compilador. Exemplo de comentário de uma única linha:

```
//Exemplo de comentário de uma única linha
```

Para comentar mais de uma linha do código, basta adicionar barra invertida mais asterisco (`/*`) no começo da primeira linha de comentário e `*/` no final da última linha de comentário. O que vier depois do símbolo de `/*` e antes do `*/` será considerado comentário e ignorado pelo compilador. Exemplo de comentário em blocos de linha:

```
/* Exemplo de comentário  
   com blocos de linha */
```

1.4 Compilando o Programa

Existem vários compiladores voltados para a linguagem C. Um dos mais conhecidos é o compilador de código aberto *GCC – GNU Compiler Collection* ¹. Em ambientes Linux o gcc já vem instalado, e para confirmar basta digitar **gcc** no prompt de comando. Se a mensagem “gcc: no input files” for exibida significa que ele está presente, indicando apenas que nenhum arquivo foi informado para ser compilado.

Caso necessite da instalação do compilador **gcc**, ela pode ser feita com o seguinte comando (ambientes Debian e derivados):

```
sudo apt-get install gcc
```

Para usuários do Windows, o **gcc** também tem uma versão chamada MinGW que precisa ser necessariamente instalada.

Para compilar um arquivo em C com o **gcc** utilizamos a seguinte sintaxe:

```
gcc <nomeArquivo.c> -o <nomeArquivoExecutavel>
```

Por exemplo, no prompt de comando:

```
mauricio@linux-note:~/c-programming$ gcc exemplo.c -o exemplo
```

No exemplo acima o arquivo a ser compilado é chamado de **exemplo1** (com a extensão **.c**). O parâmetro **-o** indica ao compilador que o arquivo executável a ser criado será chamado de exemplo (poderia ser qualquer nome).

Para executar o programa basta digitar o nome do arquivo executável criado:

```
mauricio@linux-note:~/c-programming$ ./exemplo
```

Os símbolos “./” indicam que o programa está no diretório (pasta) atual.

Outra alternativa interessante de compilador de código aberto para C é o *Clang – a C language family frontend for LLVM* ². Sua sintaxe de uso é a mesma do GCC, no entanto, geralmente é necessário fazer sua instalação. No Linux, basta o seguinte comando no prompt (ambientes Debian e derivados):

```
mauricio@linux-note:~$ sudo apt-get install clang
```

¹gcc.gnu.org/

²clang.llvm.org/

1.5 Ambientes de Desenvolvimento Integrado

Para facilitar a tarefa do desenvolvedor de software, existem diversos ambientes de desenvolvimento integrado ou *IDEs* – *Integrated Development Environment* para a programação em C. Um deles é o Sublime Text³, uma IDE que se destaca pela simplicidade e rapidez, além de ser amplamente customizável.

Para compilar um programa escrito em C no Sublime Text, é preciso alguns “ajustes”. Considerando que o Sublime Text esteja instalado, vá no menu **Tools** -> **Build System** -> **New Build System**. Para usuários do Linux, basta copiar o código abaixo no arquivo aberto:

```
1 {  
2     "cmd":["gcc $file_name -o ${file_base_name} && ./${file_base_name}"],  
3     "selector": "source.c",  
4     "Shell": true,  
5     "working_dir" : "$file_path"  
6 }
```

Salve o arquivo com o nome **C.sublime-build** na pasta **config** – pasta pessoal do Linux, por exemplo. Depois, Mude o BuildSystem para o C no Sublime em: **Tools** > **BuildSystem** > **C**. Pronto! Agora podemos compilar e executar o programa através da opção **Tool** -> **Build**, ou simplesmente usar a tecla de atalho “**Ctrl + B**” (Figura 1):



```
File Edit Selection Find View Goto Tools Project Preferences Help  
exemplo1.c  
1 #include <stdio.h>  
2  
3 int main(){  
4  
5     printf("Hello world! \n");  
6  
7     return 0;  
8 }  
9  
Hello world!  
[Finished in 47ms]  
Line 9, Column 1 Tab Size: 4 C++
```

Figura 1: Compilando programa em C com Sublime Text

³sublimetext.com/

1.6 Depurando o Programa

À medida que os programas vão ficando mais complexos, fica difícil encontrar erros e problemas de implementação. Para facilitar esta tarefa, existe o depurador (*debugger*), uma ferramenta útil para testar e depurar programas em busca de erros ou defeitos.

Um depurador permite a execução de um programa passo a passo (linha por linha), além da possibilidade de verificar o estado atual das variáveis em pontos predefinidos do programa.

O GDB⁴, do Projeto GNU, é o depurador padrão para a linguagem C no Linux, permitindo que o desenvolvedor monitore o que está acontecendo “dentro” de outro programa enquanto ele é executado.

Para exemplificar, vamos usar um programa exemplo da listagem 2 cujo objetivo é calcular e mostrar o fatorial de um número dado. O programa contém erros para fins de depuração.

⁴gnu.org/software/gdb/

```

1 # include <stdio.h>
2
3 int main() {
4     int i, n, fat;
5     printf ("Digite um numero: ");
6     scanf ("%d", &n);
7
8     for (i=1; i<n; i++)
9         fat *= i;
10
11     printf("O fatorial de %d e' %d\n",n,fat);
12     return 0;
13 }

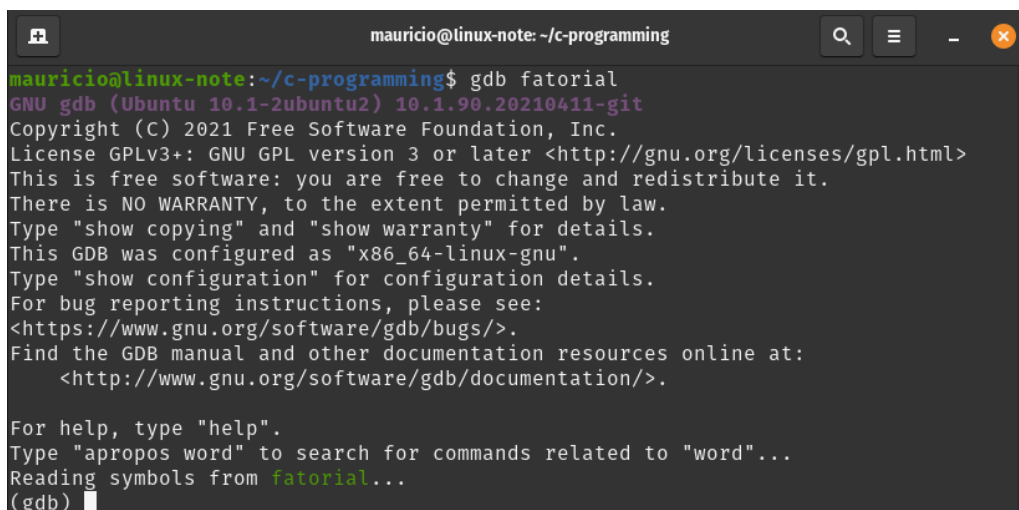
```

Código 2: Problema exemplo - fatorial(n)

Antes de usar o *debugger* é necessário compilar o programa alvo para incluir no executável informações necessárias ao processo de depuração. Usando o compilador `gcc`, basta inserir a flag `-g` ao comando de compilação, como exemplo a seguir:

```
mauricio@linux-note:~/c-programming$ gcc -g fatorial.c -o fatorial
```

O comando mostrado adiciona ao processo de compilação a instrumentação do depurador `gdb` ao executável criado, neste caso, chamado de `fatorial`. Feito isso, podemos iniciar GDB via prompt de comando (Figura 2).



```

mauricio@linux-note: ~/c-programming
mauricio@linux-note:~/c-programming$ gdb fatorial
GNU gdb (Ubuntu 10.1-2ubuntu2) 10.1.90.20210411-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from fatorial...
(gdb)

```

Figura 2: Iniciando o GDB

Na Tabela 1 são sintetizados os principais comandos aceitos no prompt do `gdb`. Alguns desses comandos serão exemplificados a seguir.

Tabela 1: Principais comandos do gdb debugger

Comando	Ação	Exemplo
run/r	Inicia a execução do programa	run
break/b	Insere um 'breakpoint' (ponto de parada) no programa	b main
continue/c	Continua a execução do programa até o próximo breakpoint	c
next/n	Executa a próxima linha de código como uma única instrução	n
step/s	Similar ao next, mas não trata a função como uma única instrução (executa o conteúdo da função linha por linha)	s
print/p	Imprime o conteúdo de uma variável ou expressão	p x
display/d	Mostra o valor de uma variável ou expressão a cada parada	d x
quit/q	Aborda a execução do programa em definitivo	q
kill/k	Força o término da execução do programa	k

Para rodar o programa executável basta usar o comando `run` (ou simplesmente `r`) que indica ao `gdb` para iniciar a execução do programa (Figura 3). Neste caso, o programa será executado até o final pois não foi definido nenhum ponto de parada (`breakpoint`).

```
(gdb) run
Starting program: /home/mauricio/c-programming/fatorial
Digite um numero: 5
0 fatorial de 5 e' 786408
[Inferior 1 (process 3499) exited normally]
(gdb)
```

Figura 3: Executando o programa com o GDB

Observe que, o programa informa o valor errado (propositalmente) do fatorial do número fornecido pelo usuário. O cálculo do fatorial de um número n , inteiro positivo, é realizado por sucessivas multiplicações. Assim, de modo geral o fatorial de um número n é dado por:

$$\bullet \text{ fat}(n) = n * n-1 * n-2 * \dots * 1$$

Então, o fatorial de 5 deveria ser:

- $\text{fat}(5) = 3 * 2 * 1 = 120$

O próximo passo é inserir um ponto de parada (**breakpoint**) dentro do programa. Ao executar o programa, o **gdb** vai parar no **breakpoint** definido à espera de comandos para depurar, por exemplo, imprimir o conteúdo das variáveis ou executar linha por linha do código para verificar comportamentos indesejados. Para definir um **breakpoint** usamos comando **break**:

- **break <local>**, em que o local pode ser uma linha ou função específica do programa.

Para exemplificar, vamos inserir um **breakpoint** em um ponto com suspeita de erro de implementação no programa fatorial. A Figura 4 mostra a inserção de um **breakpoint** na linha 8 do programa.

```
(gdb) break 8
Ponto de parada 1 at 0x555555551d6: file fatorial.c, line 8.
(gdb)
```

Figura 4: Inserindo um breakpoint em uma linha específica

Neste momento, podemos executar o programa até **gdb** encontrar o primeiro **breakpoint** definido, conforme mostrado na Figura 5. Observe que o programa parou a execução na linha 8 à espera de algum comando do usuário para depuração.

```
(gdb) run
Starting program: /home/mauricio/c-programming/fatorial
Digite um numero: 5

Breakpoint 1, main () at fatorial.c:8
8          fat *= i;
(gdb)
```

Figura 5: Executando o programa até o primeiro breakpoint

Conforme mostrado na Tabela 1, podemos utilizar vários comandos para depurar o programa. Por exemplo, na Figura 6 exibimos o valor das variáveis **i**, **fat** e **n** através do comando **print/p(variável)**. A variável **fat** é utilizada para guardar o resultado do fatorial. Como ela não foi inicializada, o comando **p** mostra algum lixo da memória, resultado em cálculo errado do fatorial.

Vamos corrigir o problema inicializando a variável corretamente e depois compilar e executar novamente programa do fatorial. A Figura 7 mostra o resultado da execução. Perceba que, embora o erro tenha sido corrigido, o resultado obtido do cálculo do fatorial continua incorreto.

```
(gdb) p i
$1 = 1
(gdb) p fat
$2 = 32767
(gdb) p n
$3 = 5
(gdb) █
```

Figura 6: Mostrando o conteúdo das variáveis

```
(gdb) run
Starting program: /home/mauricio/c-programming/fatorial
Digite um numero: 5
O fatorial de 5 e' 24
[Inferior 1 (process 3988) exited normally]
(gdb) █
```

Figura 7: Executando o programa após a correção

Para encontrar o outro problema no código é necessário prosseguir com o processo de depuração. Para continuar a depuração, tem-se os comandos **continue**, **step** ou **next**. A Figura 8 exemplifica o uso do **next/n**, o qual avança para a próxima linha/instrução do programa, no caso voltando para o laço **for**. Continuando com a depuração, podemos chegar ao problema que tem relação com a expressão avaliada dentro do laço **for** (**i < n**). Pegando como exemplo a entrada fornecida pelo usuário, cujo valor a ser calculado é o **fat(5)**, o programa irá calcular o fatorial até o número 4. Neste caso, basta corrigir a expressão **i <= n** para **i <= n**.

```
(gdb) n
7      for (i=1; i<n; i++)
(gdb) █
```

Figura 8: Executando a próxima instrução do programa com o comando **next**

2 CONCEITOS BÁSICOS DA LINGUAGEM

No capítulo anterior, discutimos a estrutura básica pra criar um programa em C e alguns conceitos que ajudam na legibilidade e entendimento do programa. Neste capítulo, vamos discutir os conceitos elementares da linguagem C, que se compreendidos, facilitam sua aplicação em outras linguagens de programação como Java e #C.

Para exemplificar sua aplicação, vamos criar um simples programa à medida que os esses conceitos são apresentados. Nosso desafio é criar um programa que leia quatro notas correspondendo às médias bimestrais de um aluno, e depois calcular e exibir a média final desse aluno na tela.

2.1 Variáveis

Durante o processamento de um programa, todas informações manipuladas são armazenadas na memória do computador. Para que a CPU possa buscar essas informações de forma mas rápida e eficiente, cada posição da memória é associada a um endereço exclusivo, o que facilita seu acesso.

Para utilizarmos uma determinada posição de memória usamos o conceito de variável. Uma variável representa um espaço de memória identificado e reservado para guardar um valor durante o processamento.

Para efeito didático, imagine a memória como um tabela (Figura 9) contendo várias posições. A medida que definimos uma variável inteira (por exemplo, X) e atribuímos um valor a ela (por exemplo, X = 15), o computador reserva um espaço para armazenar o valor e associa a um endereço físico de memória (por exemplo, E2), o que permite a sua identificação de forma unequivoca.

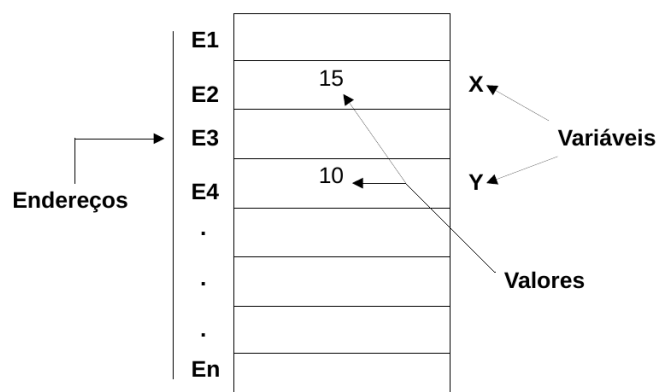


Figura 9: Variáveis na memória

2.1.1 Declaração de Variáveis

Nenhuma variável pode ser usada ao longo de um programa sem antes ter sido declarada. Na linguagem C, toda vez que declaramos uma variável devemos especificar para o compilador o tipo de dado que ela vai armazenar. Isso porque C é uma linguagem de programação considerada tipada ou tipificada, diferentemente da linguagem Python, por exemplo.

A declaração geral de uma variável em C tem a seguinte sintaxe:

- **<tipo de dados> <identificador>**

Em que:

- **<tipo de dado>**: corresponde ao tipo de dado a ser armazenado na variável, por exemplo, um número inteiro, um número real ou uma letra.
- **<identificador>**: corresponde ao nome que vai identificar de forma inequívoca a variável declarada.

A seguir, temos alguns exemplos de declaração de variáveis:

```
int x, y, z; //declara 3 variaveis do tipo inteiro
```

```
double valor; //declara variavel do tipo ponto flutuante com maior precisão
```

```
char letra; //declara variavel do tipo caractere
```

Observe que para declarar mais de uma variável do mesmo tipo basta separá-las por vírgula (,). Observe ainda que toda declaração de variável termina com operador de ponto e vírgula (;), servindo para separar as instruções que compõem o programa.

Para o nosso exemplo, precisamos declarar quatro variáveis para armazenar os valores das notas bimestrais do aluno. A declaração dessas variáveis é apresentada no trecho de código apresentado a seguir.

```
1 #include <stdio.h>
2
3 int main() {
4     //declara 4 váriaveis para guardar as notas bimestrais
5     float notaB1, notaB2, notaB3, notaB4;
6     return 0;
7 }
```

2.1.2 Tipos de Dados

Vimos que o tipo de uma variável especifica os valores permitidos para àquela variável. Na Tabela 2 são sintetizados os tipos básicos utilizados na linguagem C, incluindo o intervalo de valores permitidos para cada tipo e o espaço em bytes ocupado em memória.

Tabela 2: Tipos Básicos

Tipo em C	Valores Válidos	Espaço Ocupado
int	-32767 a 32767	4 bytes
char	Qualquer caractere	1 byte
float	2^{-37} a 2^{37}	4 bytes
double (+ precisão)	2^{-37} a 2^{37}	8 bytes

Para checar o tamanho em bytes que um determinado tipo de dado ocupa na memória da nossa máquina, podemos usar o operador `sizeof` da linguagem C, conforme demonstrado pelo Código 3.

O operador `sizeof` será abordado com mais detalhes no Capítulo sobre alocação dinâmica de memória. Nesse exemplo, também usamos a função `printf` para mostrar na tela os tamanhos de cada tipo de dado. A função `printf` será discutida em seções subsequentes.

```
1 #include <stdio.h>
2
3 int main() {
4     printf(" --- TIPO ---|--- BYTES ---\n");
5     printf(" char .....: %d bytes\n", sizeof(char));
6     printf(" int.....: %d bytes\n", sizeof(int));
7     %printf(" long.....: %d bytes\n", sizeof(long));
8     printf(" float .....: %d bytes\n", sizeof(float));
9     printf(" double.....: %d bytes\n", sizeof(double));
10    return 0;
11 }
```

Código 3: Checando o tamanho de diferentes tipos de dado

Tipo int

O tipo de dado **int** é utilizado somente para armazenar valores numéricos inteiros, conforme exemplos apresentados a seguir:

```
int x = 100; //x recebe o valor 100

int idade = 18; //idade recebe o valor 18
```

Tipo char

O tipo de dado **char** é utilizado para armazenar somente um caractere alfanumérico, utilizando a codificação de caracteres ASCII, que representa qualquer caractere em 8 bits. O caractere deve ser inserido entre aspas simples (' '), conforme exemplos a seguir:

```
char x = 'a'; //x recebe o caractere a

char y = '1'; //y recebe o valor 1
```

Observe que o tipo **char** pode armazenar valores pequenos de números, uma vez que ele trabalha com a tabela ASCII que aceita números pequenos.

Tipo float

O tipo de dado **float** é utilizado para armazenar valores reais fracionários ou em ponto flutuante. Do mesmo modo, o tipo de dado **double** também armazena valores reais, mas com maior precisão. Na linguagem C a parte decimal utiliza ponto (.) e não vírgula (,), conforme exemplos a seguir:

```
float x = 1.5; //x recebe o valor 1.5

double y = 150.245; //y recebe o valor 150.245
```

Percebam que, nos exemplos apresentados utilizamos o operador “=” para atribuir um determinado valor a uma variável. Os operadores de atribuição serão descritos com mais detalhes na Seção 5.2.

2.1.3 Modificadores de Tipo

Os modificadores de tipo são utilizados para alterar os tipos básicos da linguagem, gerando novos tipos. Na Tabela 3 são sintetizados os modificadores de tipos da linguagem C, o intervalo de valores permitidos e seu tamanho em bytes.

A linguagem C permite a utilização de mais de um modificador de tipo apresentado na Tabela 3 sobre um mesmo tipo de dado. Desse modo, podemos declarar um inteiro grande (long int) sem sinal (unsigned). Essa combinação permite aumentar em muito o intervalo de valores possíveis para aquela variável:

```
unsigned long int m;
```

Tabela 3: Modificadores de Tipo

Tipo em C	Valores Válidos	Espaço Ocupado
signed int	-2147483648 a 2147483647	4 bytes
unsigned int	0 a 4294967295	4 bytes
short int/signed short int	-32768 a 32767	2 bytes
unsigned short int	0 a 65535	2 bytes
long int/signed long int	-2147483648 a 2147483647	4 bytes
unsigned long int	0 a 4294967295	4 bytes

2.1.4 Identificadores de Variáveis

Toda linguagem de programação define regras específicas para a formação de identificadores, para que eles possam ser reconhecidos pelo computador. Isso serve não somente para nomes de variáveis, mas também para denominar outros elementos do programa.

Os identificadores válidos na linguagem C devem sempre começar com:

- Letras combinadas com dígitos, e o caractere underscore “_”. Espaços, pontos ou outros símbolos não são permitidos.

Já os identificadores inválidos na linguagem C incluem:

- Caracteres especiais, tais como \, *, #, etc.
- Palavras reservadas da linguagem C, tais como if, else, while, for, case, void, etc.

Assim como qualquer linguagem de programação, a linguagem C possui um conjunto de palavras reservadas que compõem a sua sintaxe. Ao todo, são 32 palavras-chaves (Tabela 4), representando comandos e operadores específicos da linguagem e, portanto, não podem ser utilizadas pelo programador para dar nomes a variáveis ou a funções criadas por ele.

Tabela 4: Palavras reservadas à sintaxe da linguagem C

auto	break	case	continue	const	char	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	signed	sizeof	short	static
struct	switch	typeof	union	unsigned	void	volatile	while

A seguir, segue alguns exemplos de identificadores válidos na linguagem C:

- `numero`, `_numero`, `numero2`
- `mediaNotas`, `media_notas`
- `m10a5`

É importante destacar que a linguagem C é case-sensitive, ou seja, ela diferencia palavras minúsculas de palavras maiúsculas. Portanto, `divida`, `Divida` e `DIVIDA` são três identificadores de variáveis distintos na linguagem C.

Embora diferentes combinações sejam possíveis para os identificadores, é importante utilizar nomes para variáveis que revelem seu verdadeiro propósito. Por exemplo:

- `valorTotalDivida` é melhor que `vTotal`
- `data_recebimento` é melhor que `data`
- `numeroDependentes` é melhor que `num`

2.1.5 Constantes

Uma variável constante permite armazenar um determinado dado na memória sem que ele seja alterado durante a execução do programa. O valor continuará o mesmo, ou seja, constante. Na linguagem C podemos declarar constantes usando o comando `#define`, cuja sintaxe é a seguinte:

- `<#define> <nomeConstante> <valorConstante>`

Exemplos de declaração de constante através do comando `#define`:

```
#define TAM 100 //constante representando valor de TAM
```

```
<#define PI 3.1415 //constante representando valor de  $\pi$ 
```

Uma outra forma de declarar uma constante é através do comando `const`, cuja sintaxe é a seguinte:

- `<const tipoConstante> <nomeConstante> = <valorConstante>;`

Exemplos de declaração de constante através do comando `const`:

```
const int i = 100; //constante inteira que armazena o número 100
```

```
const double pi = 3.1415; //constante real que armazena o valor de  $\pi$ 
```

Perceba que a declaração de uma constante é muito similar à declaração de uma variável. Basicamente, o prefixo **const** apenas informa ao programa que a variável declarada não poderá ter seu valor alterado ao longo do programa.

2.2 Entrada de Dados

Comandos de entrada permitem que um ou mais dados sejam obtidos (lidos) pelo computador a partir de um dispositivo de entrada como, por exemplo, o teclado. Os valores lidos devem ser armazenados em variáveis na memória para que esses possam ser utilizados pelo programa. Para isso, o comando de entrada de dados deve, além de solicitar a operação de leitura, informar os nomes das variáveis que irão armazenar os valores lidos.

2.2.1 Função scanf

Na linguagem C a leitura de dados através da entrada padrão, ou seja, via teclado é feita com a função **scanf**, cuja sintaxe é a seguinte:

- `scanf("tipoEntrada", listaVariaveis);`

Em que:

- **tipoEntrada**: tipos de valores que serão lidos da entrada.
- **listaVariaveis**: variáveis em que serão armazenados os valores lidos.

Os tipo de entrada especificam o formato de entrada dos dados que serão lidos pela função `scanf()`. Cada tipo de entrada é precedido por um sinal de %, e um tipo de entrada deve ser especificado para cada variável a ser lida. A Tabela 5 resume os tipos de entrada básicos da linguagem C.

Alguns exemplos de leitura de dados usando a função `scanf()`:

```
scanf("%d", &x);
```

```
scanf("%f", &y);
```

```
scanf("%c", &z);
```

Tabela 5: Tipo de Entrada - indicando quais são os tipos de valores lidos

Tipo	Formato	Descrição
int	%d, %i	leitura de números inteiros
long int	%ld	leitura de números inteiros longos
char	%c	leitura de um caractere
float	%f	leitura de números reais
double	%lf	leitura de números reais com maior precisão

```
scanf("%d%f%c", &x, &y, &z);
```

Observe que usamos o operador & que retorna o endereço de memória da variável. Vamos explorar mais o operador & quando falarmos de ponteiros.

Agora que já sabemos como ler as quatro notas bimestrais do aluno via teclado, podemos seguir com a solução do nosso problema. No trecho código a seguir, acrescentamos a leitura das notas. Isso é feito na linha 8, na qual a função `scanf` lê as quatro notas pelo teclado e armazena nas respectivas variáveis os valores lidos.

O próximo passo para resolver o nosso problema é calcular a média final das notas lidas pelo teclado. Na seção a seguir, são discutidas as expressões aritméticas.

```
1 #include <stdio.h>
2
3 int main() {
4     //declara 4 variaveis para guardar as notas
5     float notaB1, notaB2, notaB3, notaB4;
6
7     //lê as notas via teclado
8     scanf("%f%f%f%f", &notaB1, &notaB2, &notaB3, &notaB4);
9     return 0;
10 }
```

2.3 Expressões Aritméticas

Expressões aritméticas são expressões que geram resultados numéricos, sejam números inteiros ou fracionários. A declaração geral de uma expressão aritmética tem a seguinte sintaxe:

<operando> <operador aritmético> <operando>.

Os operadores utilizados em expressões aritméticas escritos na linguagem C são os mesmos utilizados em expressões aritméticas comuns. Porém, em linguagens de programação o símbolo utilizado para a multiplicação é o asterisco (*), enquanto na divisão é utilizado a barra (/). A Tabela 6 mostra os operadores que podem ser utilizados em expressões aritméticas, na forma adotada pelo pseudocódigo.

Assim como na Matemática, os operadores aritméticos têm diferentes precedências na execução das operações: primeiro são calculada as potências, depois as multiplicações e as divisões e, no final, as somas e as subtrações. Expressões com operadores de mesma precedência justapostos são avaliadas da esquerda para a direita. Essa ordem de precedência pode ser alterada através do uso de parênteses.

Tabela 6: Expressões Aritméticas

Operação	Símbolo	Exemplo	Descrição
Adição	+	$x + y$	Resulta na soma de x e y
Subtração	-	$x - y$	Resulta na subtração de x por y
Multiplicação	*	$x * y$	Resulta na multiplicação de x e y
Divisão	/	x / y	Resulta na divisão de x por y
Resto	%	$x \% y$	Resulta no resto da divisão de x por y

Voltando ao nosso problema, para calcular a média das notas basta somá-las e em seguida dividir pelo total de notas, ou seja:

- $(\text{notaB1} + \text{notaB2} + \text{notaB3} + \text{notaB4}) / 4;$

O próximo passo é guardar o resultado da média das notas e depois mostrar esse resultado. Para armazenar o resultado usamos a atribuição.

2.4 Operadores Relacionais e Lógicos

Os operadores relacionais comparam dois valores ou expressões, e o resultado será sempre 1 (verdadeiro) ou 0 (falso). Por exemplo, x é maior que y? é representado na linguagem por $x > y$? Na Tabela 7 são listados os operadores relacionais disponíveis.

Os operadores lógicos, por sua vez, comparam duas ou mais expressões executadas em conjunto, resultando também sempre um resultado 1 (verdadeiro) ou 0 (falso). A Tabela 8 sintetiza os operadores lógicos utilizados na linguagem C.

A tabela verdade representada na Tabela 9 sintetiza os resultados produzidos por cada operador lógico de acordo com os resultados das expressões lógicas x e y, sendo 1 ou 0.

Tabela 7: Operadores Relacionais em C

Operador	Significado	Exemplo
==	Igual	x == y
!=	Diferente	x != y
>	Maior	x > y
<	Menor	x < y
>=	Maior ou igual	x >= y
<=	Menor ou igual	x <= y

Tabela 8: Operadores lógicos

Operador	Em C	Significado
E	&&	Verdadeiro se somente ambos os operandos são verdadeiros
OU		Verdadeiro se um dos operandos for verdadeiro
NOU	^	Verdadeiro se apenas um dos operandos for verdadeiro
NÃO	!	Verdadeiro se o operando for falso, falso se o operando for verdadeiro

Tabela 9: Tabela verdade operadores lógicos

x	y	x && y	x y	x ^ y	!x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Como ocorre nas expressões aritméticas, existe precedência na execução dos operadores relacionais e lógicos. A ordem de precedência de execução é a seguinte: !, >, >=, <, <=, ==, !=, ^, && e ||. Da mesma forma, podemos alterar a ordem de precedência na execução dos operadores usando o recurso de parênteses.

Independentemente da ordem de precedência ou não, o uso de parênteses é útil pois, além de garantir correteza na avaliação das expressões, também tende a melhorar o entendimento da expressão como um todo.

Os operadores relacionais e lógicos, são frequentemente utilizados em comandos de seleção e de repetição, os quais são temas de discussão nos capítulos seguintes.

2.5 Atribuição

Na linguagem C o símbolo “=” é utilizado para representar uma atribuição de um valor ou resultado de uma expressão a uma variável. Caso já exista um valor na variável, o valor é sobrescrito pelo novo valor, sendo valor anterior perdido.

A atribuição tem à esquerda o nome da variável que vai receber o valor, seguido à direita pela expressão cujo valor será utilizado na atribuição:

- <variável> = <expressão>

Somente um nome de variável pode ser colocado à esquerda em uma atribuição. A execução inicia avaliando a expressão à direita, colocando depois o seu resultado na variável à esquerda.

O valor da expressão à direita pode ser uma constante, uma variável ou qualquer combinação válida de operandos e operadores. Alguns exemplos de atribuições:

```
nota = 10; //variável nota recebe o valor 10

letra = 'a';

x = 5.25; //variável x recebe o valor 5,24

y = y + 2; //variável y recebe o seu conteúdo mais o valor 2

z = 2 * 8 + 15; //variável z recebe o resultado da expressão à direita

y = z + 1; //variável y recebe o conteúdo de z mais o valor 1
```

Também é possível fazer a atribuição de um mesmo valor a várias variáveis por meio do encadeamento de atribuições em um único comando:

```
x = y = z = 5.1;
```

No trecho do código a seguir, usamos o comando de atribuição para o cálculo da média final das notas bimestrais. Inicialmente, na linha 6, criamos a variável `mediaFinal` para guardar o resultado da média final. Depois, na linha 12, atribuímos o resultado da média final calculada à variável média final.

```
1 #include <stdio.h>
2
3 int main() {
4     //declara 4 variaveis para guardar as notas bimesrais
```

```

5  float notaB1, notaB2, notaB3, notaB4;
6  float mediaFinal;
7
8  //le as notas via teclado
9  scanf("%f%f%f%f", &notaB1, &notaB2, &notaB3, &notaB4);
10
11 //calcula media das notas
12 mediaFinal = (notaB1 + notaB2 + notaB3 + notaB4) / 4;
13 return 0;
14 }

```

Um uso comum da atribuição é para aumentar ou diminuir em uma ou mais unidade o valor contido em uma variável. A Tabela 10 resume algumas atribuições válidas na linguagem C.

Tabela 10: Incremento e Decremento de Variáveis

Operação	Valor de y após	Valor de x antes	Valor de x após
y = 2 + x++	7	5	6
y = 2 + ++x	8	7	6
y = 2 + x--	7	5	4
y = 2 + --x	6	5	4

Além disso, para tornar os comandos de atribuição mais simples, a linguagem C permite representá-los de uma maneira simplificada. Por exemplo, as atribuições a seguir:

```

i = i + 1;

j = j * (k - 5);

mediaFinal = resultado / numeroItens;

```

Podem ser representadas na forma simplificada como:

```

i += 1;

j *= (k - 5);

resultado /= numeroItens;

```

2.6 Saída de Dados

Comandos de saída são usados para mostrar os resultados que foram solicitados a fim de que esses sejam vistos pelo usuário, por exemplo, na tela do computador ou utilizados em futuro processamento.

2.6.1 Função `printf`

Na linguagem C a exibição dos dados através da saída padrão, ou seja, via monitor/tela é feita com a função **`printf`**, cuja sintaxe é a seguinte:

- `printf("expressaoSaida", listaArgumentos);`

Em que:

- `expressaoSaida`: podem incluir mensagens que serão exibidas como saída.
- `listaArgumentos`: podem incluir identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes.

Alguns exemplos de saída de dados através da função **`printf`**:

```
printf("%f %f %f %f \n", notaB1, notaB2, notaB3, notaB4);  
  
printf("%f \n", a * b + 1 / (c * c));  
  
printf("Nota 1 %f: \n", notaB1);
```

No primeiro exemplo, será transferido para a saída o valor contido em `n1`, depois o valor contido em `n2`, e assim por diante. No segundo exemplo, será avaliado a expressão fornecida $(a * b + 1 / (c * c))$, transferindo apenas o seu resultado para a saída. Já no terceiro, usamos strings para serem mostrados como saída. No exemplo, `n1` é o nome da variável e `Nota 1` a string a ser exibida. Supondo que o valor contido na variável `n1` seja 10, a saída produzida será:

```
Nota 1: 10
```

Note que no exemplo de uso da função **`printf`** foram adicionados os caracteres “`\n`”, os quais provocam uma mudança de linha na tela. A Tabela 11 resume outros caracteres que podem ser utilizados em conjunto com o símbolo barra invertida (`\`) para produzir um efeito específico.

A solução final do nosso problema é apresentada no Código 4. Na linha 15, usamos o comando **`printf`** para mostrar na tela a média final das notas calculadas.

Tabela 11: Formatações de saída

\	Função
\b	Backspace
\n	mudança de linha
\t	tabulação horizontal
\\	imprime o próprio caractere \
\'	imprime aspas simples
\"	imprime aspas duplas

```
1 #include <stdio.h>
2
3 int main() {
4     //declara 4 variaveis para guardar as notas bimestrais
5     float notaB1, notaB2, notaB3, notaB4;
6     float mediaFinal;
7
8     //lê as notas bimestrais via teclado
9     scanf("%f%f%f%f", &notaB1, &notaB2, &notaB3, &notaB4);
10
11     //calcula a média final das notas
12     mediaFinal = (notaB1 + notaB2 + notaB3 + notaB4) / 4;
13
14     //mostra na tela a média final calculada
15     printf("Média Final: %f \n", mediaFinal);
16     return 0;
17 }
```

Código 4: Solução final: média final das notas do aluno

Para por em prática os elementos básicos apresentados neste capítulo, vamos discutir a implementação de dois problemas retirados de [Paes 2016].

Exercício: Consumo Médio – O programa deve calcular e exibir o consumo médio de um automóvel, sendo fornecidos a distância total percorrida (em km) e o total de combustível gasto (em litros) [Paes 2016]. Como entrada, o programa recebe dois valores inteiros, o primeiro representado a distância total percorrida, e o segundo representado o total de combustível gasto. Em seguida, o programa calcula e exibe o consumo médio do automóvel.

Para o problema proposto, se temos a distância percorrida e o total gasto com combustível, basta dividi-los. Por exemplo, se o automóvel percorreu 100 km com 10 litros, significa que ele fez 100/10 kml/l, ou seja, 10 kml/.

Assim, precisamos ler os valores, fazer o cálculo e mostrar o resultado. O Código 5 apresenta a solução para este problema. Declaramos uma variável para valores inteiros para a distância percorrida, e duas variáveis para valores reais para guardar o total gasto com o combustível e o consumo médio do automóvel (linhas 5 e 6). Na linha 9 fazemos a leitura dos dados necessários e calculamos o consumo médio na linha 11. Por fim, exibimos o resultado na linha 14. Observe que o valor é representado com duas casas após a vírgula (%.2f).

```
1 #include <stdio.h>
2
3 int main() {
4     //guarda a distancia percorrida
5     int distanciaPercorrida;
6     float totalGasto, consumoMedio; //guarda o total gasto e consumo medio
7
8     //le a distancia percorrida e o total gasto
9     scanf("%d%f", &distanciaPercorrida, &totalGasto);
10    //calcula o consumo medio
11    consumoMedio = distanciaPercorrida / totalGasto;
12
13    //exibe o consumo medio calculado
14    printf("Consumo medio = %.2f\n", consumoMedio);
15    return 0;
16 }
```

Código 5: Consumo médio de um automóvel

Exercício: Troco – O programa deve calcular quanto um cliente deixa de receber de troco de uma máquina automática de café. O café custa R\$ 7 , sendo que a máquina só aceita notas de R\$ 5 e nunca dá troco. Por exemplo, se o cliente pedir apenas um café, ele vai perder R\$ 3, pois terá inserir R\$ 10, e a máquina não devolverá o troco.

O cliente pode comprar mais de um café em uma só compra. Logo, se o cliente inserir R\$ 15, ele receberá dois cafés e perderá R\$ 1.

O programa recebe como entrada a quantia em dinheiro que o cliente inseriu na máquina (sempre múltiplo de 5) e, em seguida, calcula e exibe quanto o cliente perdeu no troco.

O problema especifica que o cliente só pode inserir notas em múltiplos de R\$ 5, ou seja, 5, 10, 15, 20, 25, e assim por diante. Especifica também que o cliente pode comprar mais de um café. Por exemplo, se o cliente inserir R\$ 25, significa que ele vai querer três cafés ($3 * 7 = 21$) e, portanto, perderá R\$ 4.

Supondo que o cliente inseriu R\$ 25. Nesse caso, podemos multiplicar 7 por no máximo 3, obtendo como produto 21. Assim, $25 - 21 = 4$, que corresponde o quanto o cliente deixará de receber. Perceba que, o resto de uma divisão inteira nos dá exatamente o que estamos procurando, pois $25 / 7 = 3$, e resto = 4. O código da listagem 6 mostra a solução. O cálculo do troco é feito na linha 10. Conforme vimos na Tabela 6 da Seção 5.3, na linguagem C o operador do resto é o %.

```
1 #include <stdio.h>
2
3 int main() {
4     //guarda a quantia em dinheiro e o troco
5     int quantiaDinheiro, troco;
6
7     //le a quantia em dinheiro inserida pelo cliente
8     scanf("%d", &quantiaDinheiro);
9
10    //calcula o quanto o cliente perdera no troco
11    troco = quantiaDinheiro % 7;
12
13    //exibe o troco que o cliente deixou de receber
14    printf("%d\n", troco);
15    return 0;
16 }
```

Código 6: Troco perdido pelo cliente

2.7 Exercícios Resolvidos

1. Faça um programa que recebe os lados de um triângulo, calcule o mostre o seu perímetro.

```
1 #include <stdio.h>
2
3 int main() {
4     float a, b, c, perimetro;
5     scanf("%f%f%f", &a, &b, &c);
6 }
```

```

7 //calcula do perimetro
8 perimetro = a + b + c;
9
10 printf("Perimetro: %.2f\n", perimetro);
11 return 0;
12 }

```

Código 7: Calculo de perímetro

2. Faça um programa que recebe a base e a altura de um triângulo, calcule e mostre a sua área.

```

1 #include <stdio.h>
2
3 int main() {
4     float base, altura, area;
5     scanf("%f%f", &base, &altura);
6
7     //calcula da area do triangulo
8     area = (base * altura) / 2;
9
10    printf("Area = %f\n", area);
11    return 0;
12 }

```

Código 8: Calculo da área

3. Faça um programa que, dada a massa e altura de uma pessoa, calcule e mostre o seu IMC.

```

1 #include <stdio.h>
2
3 int main() {
4     float massa, altura, imc;
5     scanf("%f%f", &massa, &altura);
6
7     //calcula do imc
8     imc = massa / (altura * altura);
9     printf("IMC = %.2f\n", imc);
10    return 0;
11 }

```

Código 9: Calculo da imc

5. Faça um algoritmo que recebe o valor de um depósito e o valor da taxa de juros, calcule e mostre o valor do rendimento e o valor total depois do rendimento.

```

1 #include <stdio.h>
2
3 int main() {
4     float deposito, taxaJuros;
5     float rendimento, valorTotal;
6     scanf("%f%f", &deposito, &taxaJuros);
7
8     rendimento = deposito * (taxaJuros / 100); // rendimento
9     valorTotal = deposito + rendimento; //valor total apos rendimento
10
11     printf("Valor do rendimento: %.2f\n", rendimento);
12     printf("Valor total apos o rendimento: %.2f\n", valorTotal);
13     return 0;
14 }

```

Código 10: Calculo do rendimento

6. Faça um algoritmo que recebe o salário-base de um funcionário, calcule e mostre seu salário a receber, sabendo-se que ele tem gratificação de R\$ 50,00 e paga imposto de 12% sobre o salário-base.

```

1 #include <stdio.h>
2
3 int main() {
4     float salarioBase, imposto, salarioReceber;
5     scanf("%f", &salarioBase);
6
7     //calcula o imposto
8     imposto = salarioBase * 0.12;
9     //calcula o salario a receber
10    salarioReceber = (salarioBase - imposto) + 50;
11
12    printf("Salario a receber: %.2f\n", salarioReceber);
13    return 0;
14 }

```

Código 11: Calculo do rendimento

2.8 Exercícios Propostos

1. Faça um programa que receba uma determinada hora expressa no formato de horas, minutos e segundos e, em seguida, transforme no valor correspondente em segundos.

2. Faça um programa que receba as coordenadas de dois pontos (A e B) no plano cartesiano, calcule e mostre distância entre esses dois pontos. Para calcular da distância entre dois pontos (A e B), temos:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

1. Faça um programa que receba a quantidade de minutos e imprima o equivalente em segundos.
2. Faça um programa que leia as variáveis a, b e c e imprima os valores das quatro fórmulas seguintes:

a) $\frac{(a * b)}{c}$

b) $a^2 + b + 5c$

c) $a * b * c + b + \frac{c}{3} * 5 - 1$

d) $\frac{(a * b * c)^3}{2}$

3. Faça um programa que receba um número inteiro e imprima o seu antecessor (inteiro anterior) e o seu sucessor (inteiro posterior).
4. Faça um programa que receba um valor representando o gasto realizado por um cliente de um restaurante e exiba o valor total a ser pago, considerando os 10% do garçom.
5. Faça um programa que receba quatro valores inteiros a, b, c, d. A seguir, calcule e mostre a diferença do produto de a e b pelo produto de c e d.
6. Faça um programa que receba o número de um funcionário, seu número de horas trabalhadas, o valor que recebe por hora e calcula o salário desse funcionário. A seguir, mostre o número e o salário do funcionário.
7. Uma locadora de carros está fazendo uma promoção e alugando seus carros por R\$ 30,00 a diária. Além disso, a locadora cobra R\$ 0,01 por quilômetro rodado. Para fidelizar os clientes, a locadora está dando 10% de desconto no valor total do aluguel de qualquer carro.

Faça um programa que leia quantos dias a pessoa ficou com o carro: $[1; 30]$ e quantos quilômetros ela rodou $[1; 1000]$, calcule e mostre o valor total que a pessoa deve pagar pelo aluguel do carro.

8. Faça um programa que leia um número real que representa um valor em dólares e realize a conversão de dólar para real: para cada valor lido em dólar, será exibido o correspondente em reais ($1 \text{ dólar} = 4.64 \text{ reais}$). Ao final, mostre o valor convertido.

REFERÊNCIAS

[Paes 2016] Paes, R. B. (2016). *Introdução à programação com a linguagem C*. Novatec.