INTRODUÇÃO À PROGRAMAÇÃO

PROGRAMAÇÃO 101

1 Olá mundo!

Com este livro você aprenderá aspectos básicos de computação, algoritmos e, principalmente, aprenderá a expressá-los utilizando a linguagem C, comumente utilizada como primeira linguagem ensinada em cursos de programação.

O contéudo deste documento aborda todo o conteúdo ministrado durante as aulas do Programação 101. Futuramente, este livro cobrirá tudo o que é ensinado nas disciplinas Programação e Desenvolvimento de Software I e II (PDS1 e PDS2) e Estruturas de Dados, matérias do Departamento de Ciência da Computação da UFMG.

Este material é **gratuito** e não pode ser vendido sob hipótese alguma.

1.1 Por quê C?

A linguagem C é utilizada nas disciplinas ofertadas pelo DCC-UFMG (além de C++ e Python). A linguagem pode ser um pouco mais difícil do que outras quando é a primeira a ser aprendida, mas garante um bom entendimento do funcionamento de um computador e escrita de programas mais eficientes. Com C você será disciplinado a ser um desenvolvedor melhor.

Além disso, C é uma das linguagens mais utilizadas no mundo inteiro. Programas escritos em C podem ser executados em diversas máquinas e em diversos sistemas operacionais, facilitando a criação de um software que funcione em várias plataformas.

A linguagem pode ser utilizada em áreas base, como desenvolvimento de sistemas operacionais, drivers e aplicações para sistemas embarcados e também pode ser utilizada em desenvolvimento de jogos, navegadores, implementação de novas linguagens... O universo de coisas que você pode fazer com C é enorme!

1.2 O que aprenderei com este livro?

Com este documento você aprenderá desde o mais básico de programação, escrevendo seu primeiro programa, entenderá o que é uma variável (e aprofundará mais tarde), aprenderá a definir "comportamentos"para seu programa a aspectos mais avançados, como gerenciamento de memória de forma segura, estruturas de dados e algoritmos para otimizar seu programa.

Algumas dessas coisas podem ainda não estar nesta revisão, pois o livro não foi completamente escrito ainda por falta de tempo do autor.

1.3 Preciso instalar algo em meu computador para acompanhar este livro?

Se você deseja praticar o conteúdo deste livro, sim. Existem algumas maneiras que você pode escrever um código em C e executar na sua máquina, mas para ter consistência com o curso, recomendamos que utilize o programa Code::Blocks. É um programa gratuito e disponível para Windows, macOS e Linux. Se quiser alguma alternativa, sinta-se livre para conversar com a equipe do Programação 101.

1.4 Preciso saber alguma coisa para entender o conteúdo?

Não. Este livro foi escrito para que qualquer pessoa consiga aprender a desenvolver o programa que quiser (dentro dos recursos disponíveis), não importa o quanto já sabia antes sobre

qualquer coisa na área.

1.5 Sugestões?

Este livro é editado a cada edição do Programação 101 visando sempre melhorar a forma como o conteúdo é abordado e, por isso, está aberto a críticas e sugestões, que podem ser feitas nos contatando.

1.6 Nota final

Estudar programação pode ser difícil pela primeira vez e provavelmente será estressante. Quando algo não funciona, tente rever a lógica do programa. Teste-o manualmente, tente pensar numa nova solução, peça a ajuda de alguém. Algumas vezes, respire fundo, faça pausas e descanse. Lembre-se que existem muitas maneiras de resolver um problema e não necessariamente porque um problema foi resolvido de duas formas diferentes significa que uma é melhor que a outra.

É importante notar que o mais importante de tudo é a prática. Algumas vezes pode parecer que seu professor ou sua professora teve uma inspiração divina ao resolver um problema, mas a verdade é que essa pessoa praticou durante muito tempo e já resolveu muitos exercícios, e por isso pareceu fácil, quando na verdade não foi.

Desejamos a você boa sorte nesta nova jornada!

2 Breves histórias

Serão apresentadas nesta seção pequenas histórias e curiosidades sobre computação. Não é de forma alguma necessária ler esta seção para continuar sua leitura.

2.1 O Engenho Analítico

Durante o século XIX um matemático chamado Charles Babbage inventou uma calculadora mecânica automática para resolver funções polinomiais, uma máquina cujo objetivo era principalmente evitar erros humanos.

Enquanto fazia esta calculadora, Babbage percebeu que era possível criar uma máquina que não fosse aplicada somente à calculos, mas sim genérica. Então, em 1833 começou a projetar um computador que era Turing-completo antes mesmo deste conceito existir. Esta máquina se chama Engenho Analítico.

O matemático morreu em 1871 enquanto ainda fazia sua máquina. Mesmo não finalizada, Ada Lovelace, filha do Lord Byron, escreveu o primeiro algoritmo para ser executado numa máquina. Este algoritmo executava funções matemáticas. Assim, Ada Lovelace se tornou a primeira programadora do mundo.

O Engenho Analítico era um computador semelhante aos primeiros computadores criados, porém era completamente mecânico e não trabalhava com números na base binária e sim decimal. A proposta de máquina ficou perdida com o tempo e não influenciou de forma alguma a criação dos primeiros computadores eletrônicos do século XX e, assim, computadores surgiram pelo menos duas vezes na história da humanidade.

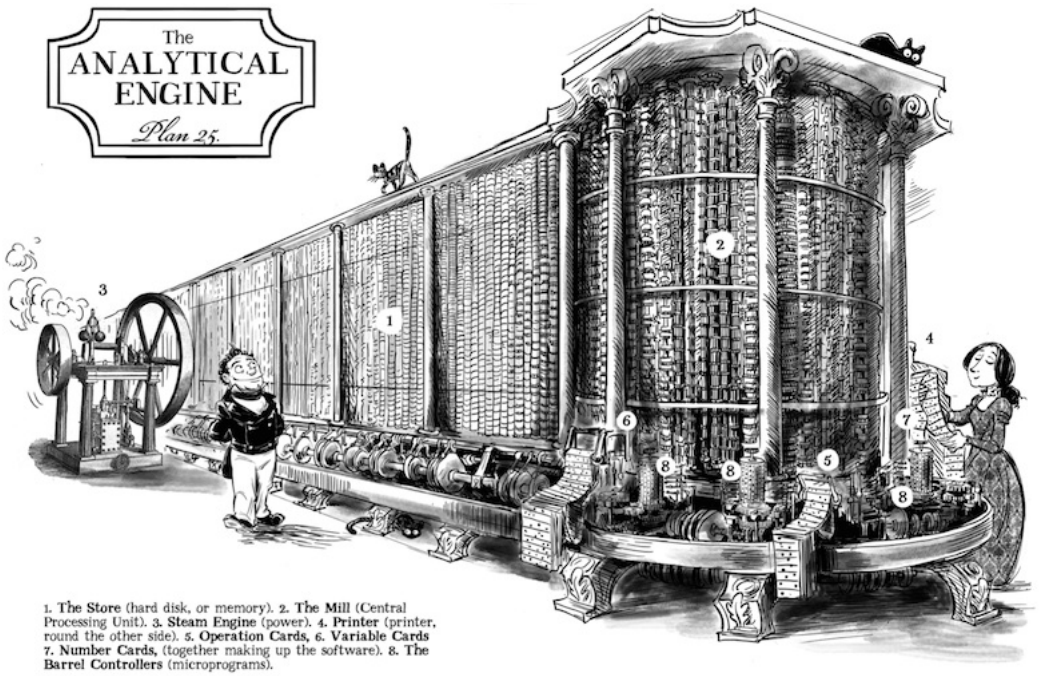

Figura 1: O Engenho Analítico.

2.2 ENIAC

O ***E****lectronic* ***N****umerical* ***I****ntegrator* ***a****nd* ***C****omputer* foi um computador que começou a ser desenvolvido durante a segunda guerra mundial mas foi finalizado somente em 1946. Uma vez criado, o poderosíssimo computador reduziu cálculos balísticos que duravam cerca de 12 horas para serem feitos a meros 30 segundos.

Antes do ENIAC esse trabalho era feito por uma equipe de 80 mulheres. A função delas no trabalho era chamado de computador. Como a máquina substituiu o trabalho de boa parte delas, foi chamada de computador.

Ao final 6 mulheres foram escolhidas para programar esta máquina enorme que ocupava uma sala inteira. Essa história está hoje documentada pelo projeto ENIAC PROGRAMMERS.

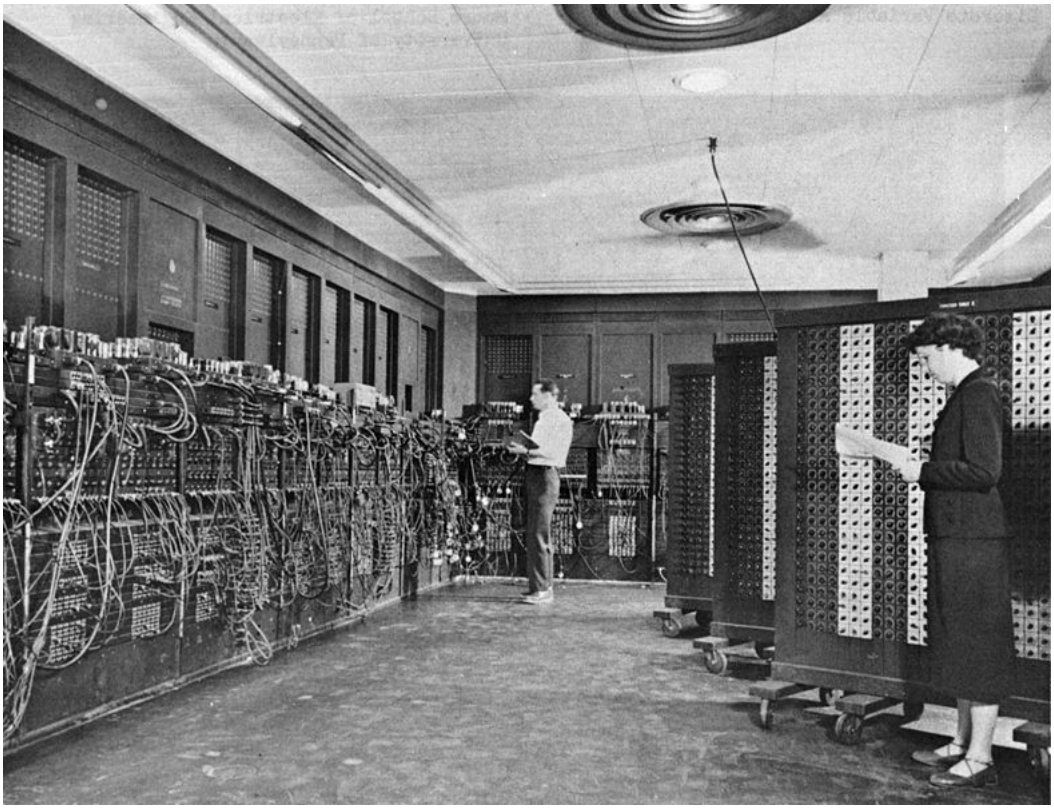

Figura 2: O ENIAC.

3 Preparando seu computador

Nesta seção iremos instalar as ferramentas necessárias para escrever nossos primeiros códigos e executar em nossas máquinas. Assim como no curso Programação 101, recomendo que instale o programa Code::Blocks.

3.1 Instalando os programas necessários

Entre no link do parágrafo acima, baixe a versão da **IDE** do seu sistema operacional e instale o programa. Se você usa Windows, faça o download do executável com "*mingw*" no nome. O arquivo deve ter um nome parecido com "*codeblocks-17.12mingw-setup.exe*".

3.2 Utilizando o programa

Na primeira vez que abrir o Code::Blocks o programa provavelmente te perguntará qual compilador usar. Se você usa Windows e não baixou a versão com MinGW, provavelmente não terá nenhum compilador disponível. Nesta hora, escolha o *gcc* (ou algum outro de sua preferência).

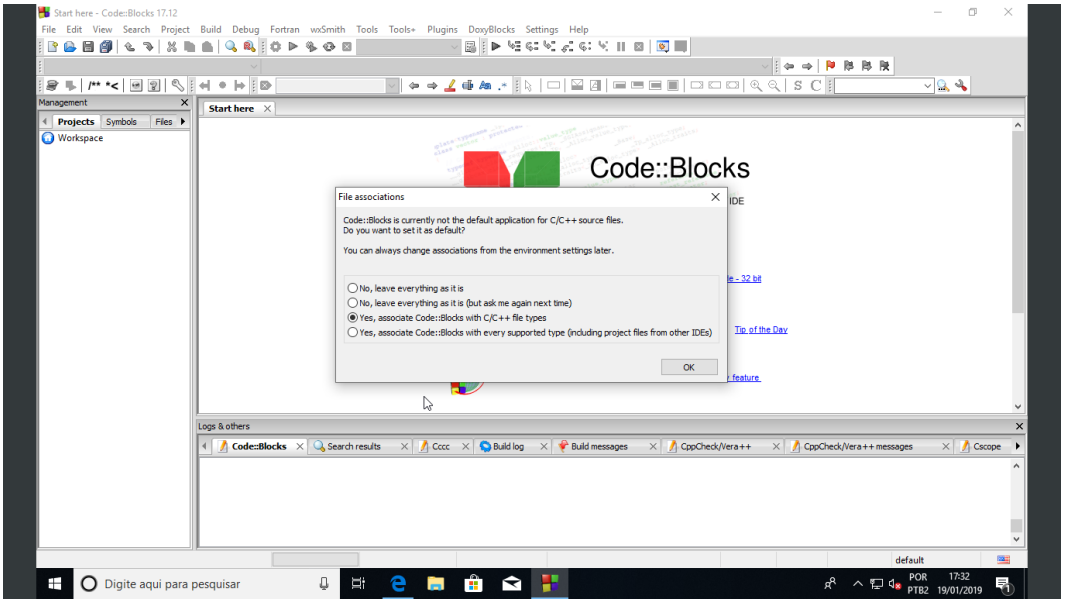

Figura 3: Escolha de compilador.

Para escrever algum código, clique em *File* no canto superior esquerdo, depois em *New* e então *Empty file*:

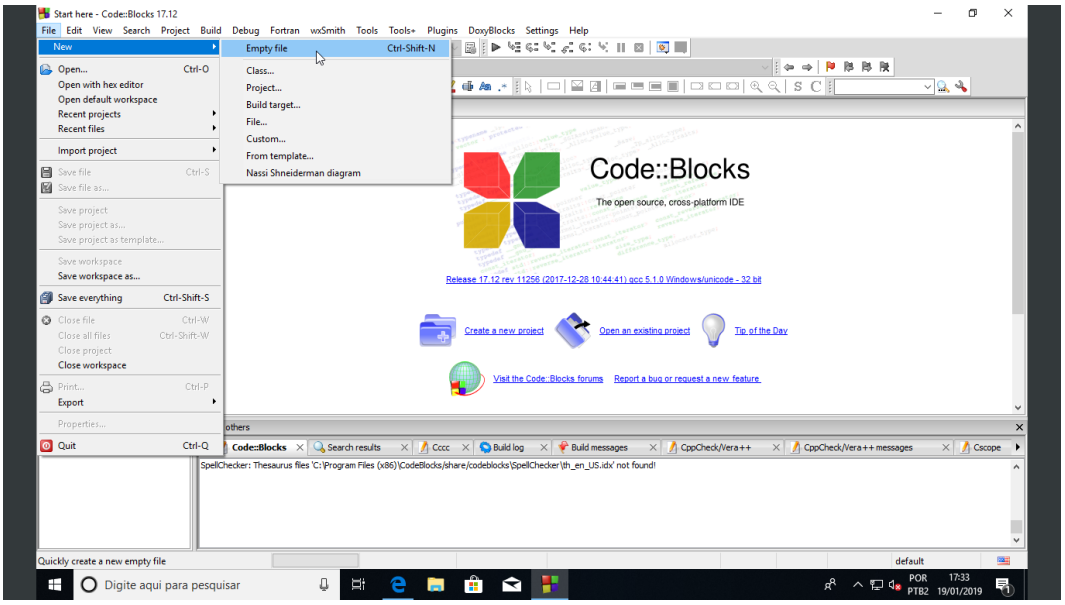

Figura 4: Criando um novo arquivo.

Então escreva o código na área em branco, salve o arquivo onde quiser e clique no botão com uma engrenagem e um "*play*"em cima:

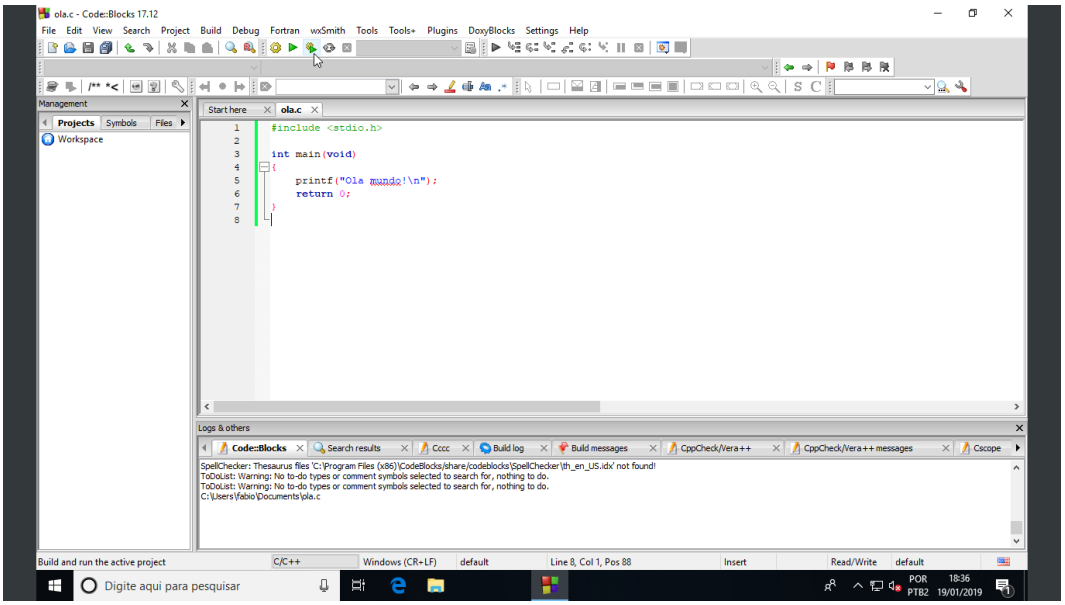

Figura 5: Compilando o arquivo.

Então seu programa será executado:

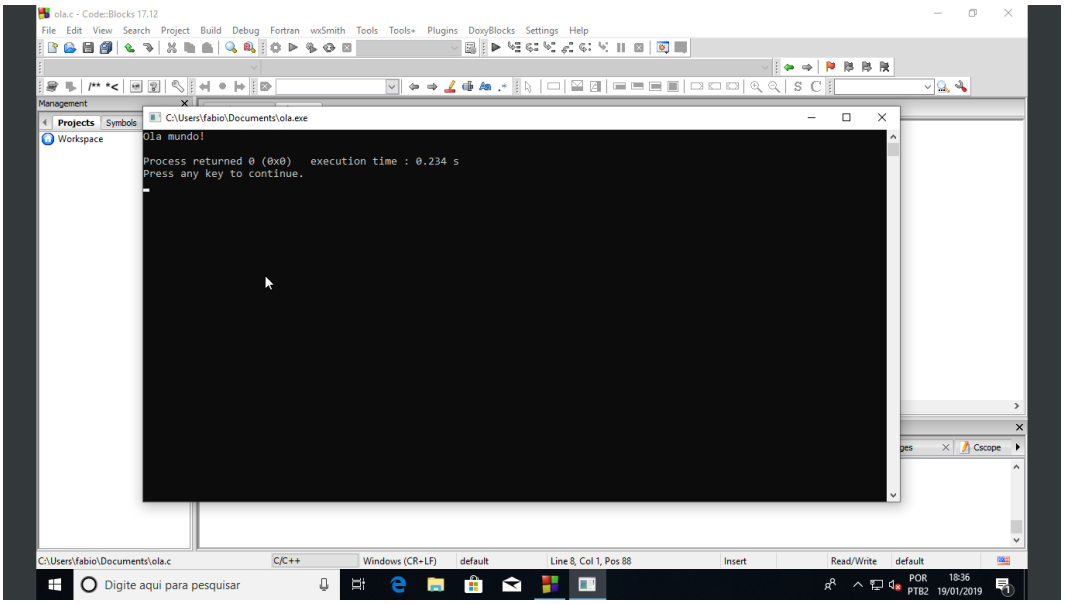

Figura 6: Executando seu programa.

4 Introdução

Computação se trata da resolução de problemas matemáticos computáveis. Antes de aprender qualquer coisa específica da área, é necessário conhecer o que é um algoritmo. Depois disso iremos escrever nosso primeiro programa utilizando a linguagem C.

4.1 Algoritmos

Algoritmo não é um conceito unicamente de computação e seu significado atual foi definido no século XIX. Trata-se de uma sequência de instruções bem definidas que transformam alguns dados (entrada) em outros dados (saída).

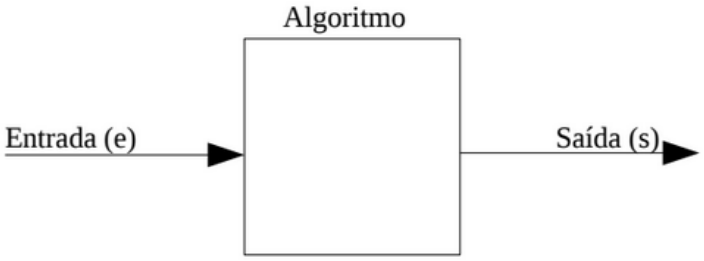

Figura 7: Abstração do conceito de algoritmo.

A figura 7 é uma demonstração prática do que é um algoritmo. Pode-se imaginar como uma "caixa mágica"que, dada uma entrada, produz uma saída. Por exemplo, a equação matemática $f(x) = x^2 + 4x$ pode ser tomada como um algoritmo: sua entrada é x e sua saída é seu resultado, e a "caixa mágica"é o algoritmo que aplica esta equação à entrada, produzindo sua saída.

Assim como uma equação matemática, um algoritmo pode ter mais de uma entrada, como funções de várias variáveis (como $f(x, y) = 2x + xy + y^2$), ou mais de uma saída, como a famosa *Fórmula de Bhaskara* e também ter várias entradas e saídas simultaneamente.

Para exemplificar o conceito apresentado nesta subseção, definimos um simples algoritmo. Nele, teremos dois números quaisquer (que serão as entradas e_1 e e_2) e compararemos os dois para determinar o maior deles. O maior valor será a saída s do algoritmo.

Algoritmo 1: Comparador de valores

Entrada: dois valores numéricos
Saída: o maior dos dois valores

```
1  início
2  |  se e1 > e2 então
3  |  |  s ← e1
4  |  fim
5  |  se e2 > e1 então
6  |  |  s ← e2
7  |  fim
8  |  se e1 = e2 então
9  |  |  s ← e1
10 |  fim
11 fim
```

No algoritmo acima, o símbolo ← faz com que o dado à esquerda receba o valor do dado à direita. Por exemplo, a linha $s \leftarrow e_1$ significa "coloque o valor de e_1 em s".

Para finalizar esta subseção, podemos comparar um algoritmo como uma receita culinária: adicione alguns ingredientes (entradas), faça algo com estas (aplique o algoritmo) e tenha em mãos o produto final (saída). Assim como uma receita, existe consistência num algoritmo: não se coloca leite e chocolate em pó para se obter frango à parmegiana (mas muitos *bugs* fazem parecer que isso acontece).

4.2 Olá mundo!

Conhecida a definição de algoritmo, podemos finalmente escrever nosso primeiro programa em C. Para isso, é necessário utilizar o programa mencionado na seção 3 e seguir o passo-a-passo de criar um arquivo. Feito isso, podemos escrever o seguinte código:

Algoritmo 2: Olá mundo!

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Ola mundo!\n");
6   return 0;
7 }
```

e então compilar e executar nosso software. Assim, veremos o seu nascimento com um texto escrito em nossa tela: *"Ola mundo!"*

Parabéns! Você criou seu primeiro programa. Antes de continuar para a próxima seção, tente manipular o código e observar resultados. Por exemplo, escreva outra coisa entre as aspas; retire o "$\backslash n$"; remova uma linha, palavra ou símbolo e tente executar o novo código. Faça tudo o que couber em sua imaginação!

Note que alguns caracteres do programa acima, como chaves ("{"), ponto-e-vírgula, aspas duplas ("") e algumas palavras (*int*, *void*) fazem parte da linguagem C e não podem ser modificados ou esquecidos. São tão importantes quanto a estrutura de uma frase em linguagens naturais como Português, assim como pontuação. Com a prática ficará claro como escrever programas na linguagem.

5 Dados

Quando escrevemos um programa, geralmente estamos manipulando dados. O algoritmo 1 mostra, em pseudocódigo, a comparação de dois valores, assinalando na sua saída o maior deles. Neste caso, os dados tratados eram numéricos, mas não são os únicos tipos de dado que podemos tratar em programação. Demonstrando que podemos manipular mais do que números, a subseção 4.2 mostra um código escrito em C no qual o dado manipulado era uma frase e, de fato, ela era escrita na tela do nosso computador.

5.1 Variáveis

Para a manipulação de dados, podemos usar o conceito de variáveis. Para demonstrar o que são variáveis, podemos fazer uma analogia a um conceito matemático aprendido no ensino fundamental: incógnitas. Aprendemos que podemos escrever $x = 3$ e, nisso, temos uma incógnita x e esta tem valor 3.

Analogamente, em computação, temos as variáveis. Elas têm um nome, assim como chamamos a incógnita de x. Elas também têm um valor (da mesma maneira que x tinha o valor 3), ou melhor dizendo, um estado. Se diferenciam no fato de que variáveis também têm **tipo**. O tipo de uma variável define se ela é um número, se ela é um caractere (letra, número, símbolo) e entre vários outros tipos, que serão vistos neste resumo (não necessariamente nesta seção).

Por exemplo, podemos definir uma variável com o nome "saudacao"e podemos dizer que seu valor é "Ola mundo!". Também podemos definir uma outra variável com o nome "tempo"e valor $1,4 \cdot 10^{10}$. Pode-se observar que a variável "saudacao"poderia ser do tipo *frase* e "tempo"poderia ser do tipo *número*, já que seus valores são, respectivamente, uma frase e um número.

Na linguagem C (assim como em outras linguagens), a declaração de variáveis se dá da seguinte forma: primeiro, se escreve o tipo de dado (que serão abordados na subseção seguinte). Logo após, separado por um espaço em branco, se escreve o nome. Em seguida, pode-se definir ou não o valor desta variável. Uma variável pode ter seu valor alterado a qualquer momento no código, desde que ela tenha sido declarada.

```
tipo nome = valor; // declaramos uma variavel com valor (inicializacao)
```

O exemplo dado nesta subseção poderia ser escrito da seguinte maneira:

```
frase saudacao = "Ola mundo!";
numero tempo = 14000000000;
tempo = tempo + 1; // mudando valor da variavel
```

Veja que podemos alterar o valor de uma variável para qualquer outro valor, desde que ela já exista. Note também que, para a criação da variável é necessário declarar seu tipo mas, ao modificar seu valor, não se escreve seu tipo à sua esquerda. Por exemplo:

```
numero contador = 1;
contador = 2;
```

é um código valido, mas

```
numero contador = 1;
numero contador = 2;
```

não é válido, pois está declarando a mesma variável duas vezes. E o seguinte código:

```
contador = 1;
contador = 2;
```

também não é válido, pois a variável não foi declarada em lugar nenhum.

Os tipos "frase"e "numero"não existem em C. Foram utilizados somente para efeitos de demonstração.

Existe uma regra para a criação de nomes de variáveis em C: seu nome só pode conter caracteres alfanuméricos (letras e números) e o caractere "_". Além disso, seu nome não pode ter como primeiro caractere um número! Assim, no seguinte código:

```
numero curso = 101;
numero _idade = 20;
numero nota_aluno = 100;
numero IdadeEmDias = _idade * 365;
```

todas as variáveis tem nome válidos, mas no próximo:

```
numero 1a = 30;
frase pergunta-nome = "Qual o seu nome?";
```

não são válidos, pois ou começam com número ou tem um caractere inválido no seu nome.

É importante observar que o valor de uma variável criada com base em outra depende do valor desta outra no momento de criação. O código a seguir pode simplificar a ideia:

Algoritmo 3: Estado de variáveis

```
#include <stdio.h>

int main(void)
{
  int x = 13;
  int f_x = 2 * x * x + x / 2 + 3; // f_x = 347
  x = 12;
  // f_x continua sendo 347, enquanto x agora vale 12
}
```

O programa não guarda a fórmula para cálculo da variável f_x mas sim o valor calculado no instante em que foi criada.

5.2 Tipos de dado

Como dito na subseção anterior, não tratamos unicamente de números. Abordaremos nesta seção 4 tipos de dados simples da linguagem C. Lembre-se que os tipos apresentados nesta subseção **não são** os únicos existentes (é até possível criar os seus próprios tipos de dado - mas não se preocupe com isso agora).

Os tipos de dados que apresentaremos agora são: **int**, **float**, **double** e **char**. A tabela abaixo mostra suas propriedades:

Tipo	Valor representável	Tamanho (em bytes)
int	Valores numéricos **inteiros**	4
float	Valores numéricos **reais**	4
double	Valores numéricos **reais**	8
char	Símbolos, caracteres	1

Tabela 1: Tipos de dados simples da linguagem C.

Veja que esta tabela mostra 3 tipos de dado usados para representação de números, sendo que dois desses tipos são para números reais! Mas existe uma diferença neles: o tamanho. Este define o intervalo de valores representáveis. Como **double** tem tamanho maior do que **float**, esse pode representar valores num intervalo maior que este.

Todos os tipos ocupam um certo número de *bytes* na memória, exatamente o seu tamanho. Isso cria uma limitação, como dito antes. Por exemplo, se tivéssemos um tipo de dados para número de tamanho 1B, teríamos 00000000_2 (0 na base 2) como menor valor e 11111111_2 (oito 1's na base 2) como maior valor (que é 255 na base decimal). Dentro do intervalo de valores entre 0 e 255 estão os valores representáveis por este tipo de dado de tamanho 1 *byte*.

Abaixo temos exemplos de códigos em C explorando variáveis e tipos de dado:

Algoritmo 4: Somador de dois valores inteiros

```c
#include <stdio.h>

int main(void)
{
  int v1 = 10;
  int v2 = 15;
  int soma = v1 + v2;
  printf("A soma dos valores %d e %d e' %d\n", v1, v2, soma);
  return 0;
}
```

Algoritmo 5: Divisão de dois valores reais

```c
#include <stdio.h>

int main(void)
{
  float v1 = 15.5;
  float v2 = 5.0;
  float divisao = v1 / v2;
  printf("A divisao de %f e %f e' %f\n", v1, v2, divisao);
  return 0;
}
```

Algoritmo 6: Divisão de dois valores reais

```c
#include <stdio.h>

int main(void)
{
  char c = 'A';
  printf("O caractere %c tem valor decimal %d\n", c, (int) c);
  return 0;
}
```

Perceba que os algoritmos acima mostram coisas que ainda não foram ensinadas, como "%d"e "%c"dentro do que está sendo impresso. No entanto, se você tentar executar o algoritmo acima, perceberá o que significam estas coisas, e, mais tarde neste livro, também exploraremos o que são.

Assim como dito no final da seção anterior, é recomendado que você mexa com o código acima tentanto extrapolar o que foi passado até agora. Mas não se preocupe com os possíveis erros, pois entenderá o que está acontecendo na continuação deste documento (ou com um de seus professores!)

5.3 Literais

A última seção de dados, literais, aborda como devem ser escritos os valores para cada tipo de dado. Por exemplo, não se delcara uma variável do tipo char como

```c
char letra_a = a; // INCORRETO!
```

pois o valor assimilado à variável (o conteúdo à direita do sinal de igualdade) não é um valor válido para a linguagem.

Para cada tipo de dado existe mais de um tipo de literal que pode ser assimilado. Por exemplo, podemos escrever um valor hexadecimal (da base de contagem 16) num int:

```
int valor = 0xFD09A;
```

0x ao início denota um valor hexadecimal.

Também pode-se escrever um valor na base octal:

```
int valor = 0071263;
```

00 ao início denota-se um valor octal.

Como já deve ter percebido, os valores de char são escritos entre aspas simples e devem ter somente 1 caractere:

```
char letra_A = 'A'
```

E de valores reais, sejam to tipo *float* ou *double*, separa-se a parte inteira da parte decimal com um ponto:

```
float valor1 = 14.3;
double valor2 = 127632139.1123;
```

Note que estamos tratando de um computador, então tudo isso está na memória na notação binária. Por exemplo, o caractere A tem valor 65_{10} (ou 01000001_2) (este valor é uma **convenção** e é definido pela tabela ASCII). E a notação binária não está presa aos caracteres, mas todos os outros tipos de dado. Assim, você pode escrever uma variável do tipo *int* e imprimí-la como um caractere:

Algoritmo 7: Imprimindo um *int* como um *char*

```
#include <stdio.h>

int main(void)
{
  int valor_A = 65;
  printf("%c\n", valor_A);
  return 0;
}
```

e também vice-versa:

Algoritmo 8: Imprimindo um *char* como um *int*

```
#include <stdio.h>

int main(void)
{
  char letra_A = 'A';
  printf("%d\n", letra_A);
  return 0;
}
```

6 Interação com o usuário

Até então tudo o que foi visto neste resumo foi mostrar mensagens na tela do usuário, através de uma função (conceito ainda não explicado neste documento) chamada *printf*. Sabemos que a partir desta podemos escrever uma mensagem na tela do usuário, e inclusive apresentar dados do próprio programa!

Para entedermos mais sobre interação com o usuário, exploraremos as ideias de entrada e saída de dados. Já conhecemos parte da saída de dados, então, começaremos por ela.

6.1 Saída de dados

Todos os programas apresentados até então utilizam uma função chamada *printf*. Seu nome significa *print formatted output* ou "imprima saída formatada", em português. Sabemos que se escrevermos a linha

```
1 printf("Ola mundo!\n");
```

veremos a frase "Ola mundo!"na nossa tela. Percebemos então que o conteúdo escrito entre estas aspas duplas é aquele que veremos na tela.

Quando se deseja escrever um dado do programa na tela do usuário, utilizamos um tipo de sequência de caracteres entre as aspas duplas no lugar onde queremos que este dado apareça e, após a última aspa, separamos por vírgula cada um dos dados, na mesma sequência colocada na frase.

Descobrimos na subseção 5.2 que esta sequência de caracteres começa com o caractere % e, logo em seguida, um outro caractere (ou mais) que indicam que tipo de dado é aquele que deve ser colocado naquela posição. Esta sequência de caracteres se chama ***placeholder***.

A tabela seguinte mostra os placeholders existentes para os tipos de dado já apresentados. Lembre-se que **existem outros tipos de dado** e, por conseguinte, existem placeholders para estes. Alguns tipos não apresentados aparecem mais tarde no resumo.

Placeholder	**Tipo de dado**
%d ou **%i**	int
%c	char
%f	float
%lf	double

Tabela 2: *Placeholders* para os tipos de dado já apresentados.

Note que às vezes um *placeholder* pode ser intuitivo, como %c de char, mas pode também não ser, como é o caso de %lf de double (lf significa "*long float*").

Tão importantes quanto os *placeholders* são os **caracteres escapados**. Um deles já foi visto, o $'\backslash n'$. Este caractere indica uma quebra de linha (ou nova linha, se preferir). Entenda-o como tendo o mesmo efeito que a tecla *Enter* do seu computador.

O caractere $'\backslash n'$ não é o único caractere escapado existente. Também existe o $'\backslash t'$ que cria espaço de tabulação. Na verdade, este tópico é bastante extenso, pois pode cobrir também **sequências de caracteres escapados**, que permitem uso de cores de fundo e de letras, negrito, itálico, desde que o terminal tenha suporte.

Pesquise *ANSI Escape Sequences* se estiver interessado em aprender sobre isso.

Um outro uso de caractere escapado seria para inserir o caractere de aspas duplas $'"'$ no texto que é impresso pela função *printf*. Como o texto desta função é denotado por este

caractere no início e fim, se o caractere '"' não for escapado (tiver uma barra invertida (\) atrás) estaríamos denotando o fim desta *string*.

Algoritmo 9: Utilizando caracteres escapados

```c
#include <stdio.h>

int main(void)
{
  printf("\"Tenho tres filhos e nenhum dinheiro... Por que nao posso ter
    nenhum filho e tres dinheiros?\" \n");
  printf(" - Homer Simpson\n");
  return 0;
}
```

6.2 Entrada de dados

Se na subseção 6.1 foi apresentado o tema saída de dados como apresentação de informação, seja de algo pré-escrito ou de uma informação interna do programa, veremos entrada de dados como a requisição de informação do usuário do programa que escreveremos.

O conteúdo da subseção 6.1 já havia sido apresentado, embora sem explicações, em todo conteúdo deste resumo, pois estávamos sempre mostrando informações ao usuário através da função *printf*. A entrada de dados vai utilizar recursos semelhantes da entrada de dados, como os *placeholders*.

Desta vez, faremos uso da função *scanf* para ler dados do usuário. Para ler dados do usuário, devemos ter um variável na qual possamos armazenar o valor.

Algoritmo 10: Lendo um dado do usuário

```c
#include <stdio.h>

int main(void)
{
  int idade;
  int idade_segundos;
  printf("Oi! Qual a sua idade?\n");
  scanf("%d", &idade);
  idade_segundos = 3600 * 24 * 365 * idade;
  printf("Voce ja viveu %d segundos!\n", idade_segundos);
  return 0;
}
```

Note o caractere &. Ele não é um erro e **deve** ser utilizado. Mais adiante neste material será abordado o que é e significa.

Perceba também que não há $'\backslash n'$ ao final do conteúdo entre aspas duplas na função scanf. Cuidado com isso!

Funciona da mesma maneira para os outros tipos de dado: se queremos ler um caractere, utilizamos o *placeholder* de um caractere e declaramos uma variável do tipo *char*. E o mesmo para float e double.

Pode ser que queira ler um conjunto de caracteres, como o nome de uma pessoa, afinal, pode parecer mais interessante ler frases do que apenas um caractere. Não faremos isso agora porque o conteúdo apresentado até aqui não é suficiente para que se entenda como criar

variáveis de frases (*strings*) nem como manipulá-las.

Algoritmo 11: Conversor de temperatura

```c
#include <stdio.h>

int main(void)
{
  float temp_celsius;
  float temp_fahrenheit;

  printf("Digite a temperatura em graus celsius: ");
  scanf("%f", &temp_celsius);

  temp_fahrenheit = temp_celsius * 1.8 + 32.0;

  printf("%f graus celsius = %f graus fahrenheit\n", temp_celsius,
    temp_fahrenheit);
}
```

É recomendado que faça os exercícios dados pelo curso Programação 101 para reforçar o que foi aprendido até então.

7 Condições e repetições

A partir desta seção podemos começar a desenvolver programas mais interessantes (e também um pouco mais difíceis), pois não estaremos mostrando somente informações, começando a definir "comportamentos"para nossos códigos de acordo com o valor destas informações, através do uso de **comandos** de controle, sendo eles de repetições e condições.

Primeiramente, exploraremos as condições e alguns exemplos serão dados. Logo após, veremos as repetições.

7.1 Condições

Como dito anteriormente, poderemos definir o comportamento do software com o uso de condições. Todo programa que desenvolvermos, a partir de agora, poderá fazer alguma coisa (ou não fazer alguma coisa) dependendo de dados no programa, isto é, valores das variáveis nos nossos programas.

Para fazermos isso, utilizaremos o comando ***if***. O seguinte código será usado como exemplificação do comando:

Algoritmo 12: Condição

```
#include <stdio.h>

int main(void)
{
  int idade;

  printf("Digite sua idade: ");
  scanf("%d", &idade);

  if(idade < 18)
  {
    printf("Voce e menor de idade!\n");
  }
  else
  {
    printf("Voce e maior de idade!\n");
  }

  return 0;
}
```

Temos no código acima uma variável *idade* do tipo *int*. Pedimos ao usuário sua idade e armazenamos esta na variável mencionada. Então, temos o comando condicional que segue a seguinte regra: *"idade é menor do que 18?"*. Este valor é avaliado e, se a resposta para a pergunta for verdadeira (for sim), somente a primeira frase é vista na tela do usuário. Mas, se a resposta for falsa (for não), esta parte do código é ignorada e o que é executado está após o comando ***else*** e, assim, somente a segunda frase é exibida ao usuário.

Este tipo de comando nos permite controlar o **fluxo de execução** dos nossos programas. Veja que, no exemplo dado, o programa ou executa um dos *printf*s ou o outro, mas nunca os dois ao mesmo tempo. E qual destas opções acontece, claro, depende diretamente do valor de uma variável, que, nesse caso, foi dado pelo usuário.

No caso do código dado acima, a condição foi $idade < 18$, mas existem outros operadores relacionais.

Operador relacional	Significado
==	Verifica se os dois valores são iguais
!=	Verifica se os dois valores são diferentes
<	Verifica se o valor à esquerda do operador é menor do que o valor à direita
>	Verifica se o valor à esquerda do operador é maior do que o valor à direita
<=	Verifica se o valor à esquerda do operador é menor ou igual ao valor à direita
>=	Verifica se o valor à esquerda do operador é maior ou igual ao valor à direita

Tabela 3: Operadores relacionais da linguagem C.

Veja que a tabela 3 só fala sobre a comparação de dois valores ao mesmo tempo, para todos os operadores relacionais. Isso acontece porque só se pode comparar dois valores ao mesmo tempo. Vejamos o seguinte programa:

Algoritmo 13: Comparação mal-escrita

```c
#include <stdio.h>

int main(void)
{
  int valor;

  printf("Digite o valor: ");
  scanf("%d", &valor);

  if(1 < valor < 3) // INCORRETO!
  {
    printf("OK!\n");
  }
  else
  {
    printf("Invalido!\n");
  }

  return 0;
}
```

Podemos rapidamente perceber que o único valor inteiro que faz com que a condição $1 < valor < 3$ seja satisfeita é 2. Testando o programa, colocamos o valor 2 e veremos a frase "*OK*!" na tela. Testando mais um pouco, colocamos qualquer outro valor e continuamos a ver a mesma resposta. Por quê isso acontece?

Para entendermos isso, vamos usar uma analogia com uma expressão aritmética. Se tivéssemos de fazer a conta $(3+4)*2$, sabemos que existe uma ordem de prioridade na qual a conta deve ser feita. Neste caso, fazemos $(3+4)$ primeiro, e substituímos isto pelo resultando, obtendo a conta $(7)*2$. Agora, pode-se fazer o último passo, encontrando o resultado 14.

Na expressão lógica $1 < valor < 3$, também existem prioridades a serem seguidas. Como dito anteriormente, os operadores condicionais comparam dois valores ao mesmo tempo. Então, a expressão condicional $1 < valor < 3$ é a mesma que $(1 < valor) < 3$. Agora, pode parecer estranho colocar parêntese no que parece uma inequação, mas fará sentido.

A avaliação de uma expressão condicional, que segue uma determinada ordem, é feita de maneira parecida como a expressão aritmética usada como exemplo: primeiro, avaliamos a primeira expressão: $(1 < valor)$. Esta avaliação pode ter dois resultados: **verdadeiro** ou

falso.

Quando uma expressão é avaliada como verdadeira, em C, ela é substituída pelo valor 1 e, quando falsa, por 0. Assim, destes dois valores possíveis, a expressão $(1 < valor)$ será substituída ou pelo valor 0 ou por 1, ambos menores que 3. Chegando ao final da avaliação, $(1 < valor) < 3$ é substituído por $(1) < 3$ ou $(0) < 3$, e ambas as condições são verdadeiras. Assim, Sempre veremos a frase "*OK*!" na tela.

Embora nós entendamos muito bem o que queríamos dizer com a expressão $1 < valor < 3$, não devemos escrever uma condição desta maneira. A expressão pode ser dividida em duas: $1 < valor$ e $valor < 3$. As duas podem ser combinadas utilizando-se **operadores lógicos**. Neste caso, para que a combinação das duas tenha o mesmo efeito da original utilizamos o operador **e**.

Qualquer valor maior do que 1 satisfaz $1 < valor$ e qualquer valor menor do que 3 satisfaz $valor < 3$, mas não é qualquer valor que satisfaz ambas ao mesmo tempo. Somente o valor 2 satisfaz ambas simultâneamente, assim como na inequação original. Seguindo o raciocínio, o programa estaria corretamente escrito da seguinte maneira:

Algoritmo 14: Comparação bem-escrita

```c
#include <stdio.h>

int main(void)
{
  int valor;

  printf("Digite o valor: ");
  scanf("%d", &valor);

  if(1 < valor && valor < 3)
  {
    printf("OK!\n");
  }
  else
  {
    printf("Invalido!\n");
  }

  return 0;
}
```

A correção foi feita apresentando o operador lógico **e** (&&). A condição acima só é avaliada como verdadeira se tanto a condição $1 < valor$ e a condição $valor < 3$ forem verdadeiras. Este é o efeito deste operador lógico.

Operador lógico	Significado	Efeito
&&	e (*and*)	Seu resultado é verdadeiro quando as duas condições são verdadeiras.
||	ou (*or*)	Seu resultado é verdadeiro quando qualquer uma das duas condições for verdadeira.
!	Inversão	Este operador lógico inverte o resultado da avaliação da expressão condicional.

Tabela 4: Tabela de operadores lógicos da linguagem C.

A tabela 4 mostra os operadores lógicos da linguagem C. Pode-se fazer combinações de

expressões condicionais através destes. Os seguintes programas demonstram isso:

Algoritmo 15: Utilizando operador lógico e

```c
#include <stdio.h>

int main(void) // prototipo de batalha naval
{
  char letra = 'B';
  int numero = 5;

  char coordenada1;
  int coordenada2;

  printf("Digite as coordenadas: ");
  scanf("%c %d", &coordenada1, &coordenada1);

  if(letra == coordenada1 && coordenada2 == numero)
  {
    printf("Acertou!\n");
  }
  else
  {
    printf("Errou!\n");
  }

  return 0;
}
```

Algoritmo 16: Utilizando dois operadores lógicos simultâneamente

```c
#include <stdio.h>

int main(void) // verifica se um caractere e alfabetico
{
  char c;

  printf("Digite um caractere: ");
  scanf("%c", &c);

  if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
  {
    printf("O caractere '%c' e alfabetico.\n", c);
  }
  else
  {
    printf("O caractere '%c' nao e alfabetico.\n", c);
  }

  return 0;
}
```

O algoritmo 16 explora também um conceito apresentado sobre caracteres serem armazenados como uma sequência de bits, que também podem ser tratados como números (todos os dados são bits, afinal). Para uma maior abstração, o caractere 'A' tem valor binário 01000001_2, que é 65_{10} e 'Z' tem valor 01011010_2, que é 90_{10}. Todos os caracteres maiúsculos estão neste intervalo, assim, pode-se verificar se um caractere está no

intervalo 'A' $\leq$ caractere $\leq$ 'Z'.

Algoritmo 17: Estruturas if - else if - else

```
#include <stdio.h>

int main(void)
{
  int nota;

  printf("Digite a nota: ");
  scanf("%d", &nota);

  if(nota < 40) printf("Conceito F\n");
  else if(nota < 60) printf("Conceito E\n");
  else if(nota < 70) printf("Conceito D\n");
  else if(nota < 80) printf("Conceito C\n");
  else if(nota < 90) printf("Conceito B\n");
  else printf("Conceito A\n");

  return 0;
}
```

No algoritmo 17 vemos uma corrente de condições que são excludentes. Se o mesmo programa não fosse feito desta maneira, usando somente *if*s, com qualquer nota ≥ 90 todas as mensagens seriam vistas na tela do usuário. No mesmo algoritmo, pode ter percebido a falta dos caracteres { e } para o conteúdo de cada *if*. Quando somente um comando é executado pela condição, o uso das chaves é opcional.

Esta subseção apresentou muitos conteúdos e antes de continuar com o material é recomendado praticar o que foi aprendido através dos exercícios dados para fixar o conteúdo e não acumular dúvidas.

7.2 Estruturas de repetição

Assim como o conteúdo anterior, com as estruturas de repetição definiremos comportamentos para nossos programas. E, da mesma maneira, o comportamento dependerá da validação de condições, as mesmas aprendidas anteriormente.

As estruturas de repetição, como o nome indica, definem comportamentos repetitivos nos programas que desenvolvemos. Embora seja um conceito simples, assim como o anterior, é fundamental e provavelmente fará parte de todo projeto que desenvolva.

O primeiro que veremos (e o mais básico) será feito através do comando ***while***. Vejamos o

seguinte código:

Algoritmo 18: Comando *while*

```c
#include <stdio.h>

int main(void)
{
  int contador = 1;

  while(contador <= 3)
  {
    printf("%d\n", contador);
    contador = contador + 1;
  }

  printf("Fim\n");

  return 0;
}
```

Neste, temos uma variável contador inicializada em 1. Em sequência, já encontramos o comando mencionado, com uma expressão condicional entre parênteses. Para que aquele *bloco* de código abaixo do *while* seja executado, aquela condição deve ser **verdadeira**.

Diferentemente do comando *if*, sempre que o bloco de um comando de repetição é finalizado, a expressão condicional é avaliada novamente e, se a mesma for **verdadeira** mais uma vez, o bloco é executado novamente. Resumidamente, aquele pedaço do nosso programa será executado **enquanto** aquela expressão condicional for **verdadeira**.

Podemos abstrair mais ainda este conceito, analisando o algoritmo 18 passo a passo:

1. Criamos uma variável chamada *contador* e a inicializamos com o valor 1;
2. Iniciamos uma estrutura de repetição *while* com a condição *contador* ≤ 3;
3. A expressão condicional *contador* ≤ 3 é verdadeira para *contador* $= 1$, o bloco é executado;
4. *contador* é impressa na tela;
5. *contador* tem seu valor acrescido de 1;
6. A expressão condicional *contador* ≤ 3 é verdadeira para *contador* $= 2$, o bloco é executado;
7. *contador* é impressa na tela;
8. *contador* tem seu valor acrescido de 1;
9. A expressão condicional *contador* ≤ 3 é verdadeira para *contador* $= 3$, o bloco é executado;
10. *contador* é impressa na tela;
11. *contador* tem seu valor acrescido de 1;
12. A expressão condicional *contador* ≤ 3 é falsa para *contador* $= 4$, o fluxo do programa sai do bloco *while*;
13. "Fim"é impresso na tela.

E é apresentado na tela do usuário:

```
1
2
3
```

```
4 Fim
```

Listing 1: Saída do algoritmo 18

Embora seja um exemplo simples, demonstra bem como funciona a estrutura *while* e como é o *fluxo de execução* (o "comportamento") do programa.

Uma outra estrutura de repetição que utiliza o comando *while* é a *do while*. Nela, o bloco é executado e, em seguida, a condição é avaliada (e, se verdadeira, o processo se repete). É escrita da seguinte maneira:

Algoritmo 19: Estrutura *do while*

```c
#include <stdio.h>

int main(void)
{
  // ...

  do
  {
    // bloco
  }
  while(/* condicao */);

  return 0;
}
```

Um exemplo de aplicação deste é verificando se a entrada do usuário é válida:

Algoritmo 20: Validação de entrada

```c
#include <stdio.h>

int main(void)
{
  int numero;

  do
  {
    printf("Digite um numero entre 1 e 100 (inclusive): ");
    scanf("%d", &numero);
  }
  while(numero < 1 || numero > 100);

  return 0;
}
```

O programa 21 pedirá ao usuário que entre com um valor enquanto este valor não for válido.

Dentro de blocos de repetição dois comandos para o controle do *fluxo de execução* podem ser utilizados: ***break*** e ***continue***. O primeiro é utilizado para sair do bloco de repetição mesmo

que a expressão condicional ainda seja verdadeira.

Algoritmo 21: Validação de entrada

```c
#include <stdio.h>

int main(void)
{
  int numero;
  int erro = 0;

  while(1) // loop eterno, pois 1 e verdadeiro
  {
    // ...

    if(erro == 1)
    {
      break;
    }
  }

  return 0;
}
```

O exemplo acima é uma demonstração. Nela, onde há as reticências poderia haver um trecho de código que modifica a variável erro na ocorrência de algo indesejado.

"Mas você não poderia ter feito um bloco *while(erro == 0)*?"Claro que sim. Mas existem várias maneiras de se fazer a mesma coisa quando se trata de algoritmos. Também existe a maneira mais **legível** de se escrever um algoritmo, algo muito importante principalmente se outra pessoa lerá seu código.

Também pode-se considerar como legível um código que deixa explícito o que está fazendo através de comentários (embora não abordados, foram demonstrados em códigos marcados pelos caracteres //) e nomes concisos. Por exemplo, se uma variável tem propósito de contador, nomeie-a contador (ou algo parecido), e não João ou catioro.

Outro comando de repetição é o ***for***. Sua estrutura é a seguinte:

Algoritmo 22: Comando ***for***

```c
#include <stdio.h>

int main(void)
{
  int i;

  for(i = 1; i <= 10; i++)
  {
    printf("%d\n", i);
  }

  return 0;
}
```

Este mesmo código pode ser escrito com o comando *while*, obtendo-se o mesmo efeito:

Algoritmo 23: Comando *while* comparado com *for*

```c
#include <stdio.h>

int main(void)
{
  int i = 0;

  while(i <= 10)
  {
    printf("%d\n", i);
    i++; // neste caso especifico (e no algoritmo anterior), e o mesmo que i = i + 1
  }

  return 0;
}
```

O $i++$ será abordado na seção 8.

No comando *for*, o primeiro "campo"entre os parênteses (separados por ponto e vírgula) ocorre somente uma vez. Em seguida, é feita a verificação da expressão condicional. Se avaliada como **verdadeira**, o bloco de código do comando é executado e, ao finalizar, o último campo é executado, voltando para o passo de avaliação da expressão condicional. O algoritmo 23 mostra exatamente isso.

Por último, mais um exemplo de código de estruturas de repetição:

Algoritmo 24: Primeiros 10 números da sequência de Fibonacci

```c
#include <stdio.h>

int main(void)
{
  int a = 1;
  int b = 0;
  int temp;

  int i;

  for(i = 0; i < 10; i++)
  {
    printf("%d\n", a);
    temp = a + b;
    b = a;
    a = temp;
  }

  return 0;
}
```

O conteúdo apresentado por esta seção é muito denso e **deve ser praticado**. Portanto, procure exercícios nas listas dadas e explore os conceitos apresentados por si mesmo.

8 Arranjos / Vetores

Nesta seção estudaremos o que são arranjos, como utilizá-los e alguns tipos particulares de arranjos, como strings e matrizes.

8.1 Arranjos

Com o conteúdo abordado até então, conseguimos fazer grandes e complexos programas, mas ainda com uma grande dificuldade. Podemos, com certa facilidade, fazer programas que armazenam informações de alunos, como seus nomes, notas e outras informações. O código a seguir mostra um programa simples que trata de 3 alunos:

Algoritmo 25: Gerenciador de 3 alunos

```c
#include <stdio.h>

int main(void)
{
  int mediaAluno1, mediaAluno2, mediaAluno3;

  printf("Digite a media do aluno 1: ");
  scanf("%d", &mediaAluno1);

  printf("Digite a media do aluno 2: ");
  scanf("%d", &mediaAluno2);

  printf("Digite a media do aluno 3: ");
  scanf("%d", &mediaAluno3);

  // ...

  return 0;
}
```

Mas o que acontece quando nos é pedido para criar um programa para gerenciar dados de até 100 alunos? Ou até mesmo 1000? Teremos que criar uma variável para cada um deles? Ler os dados manualmente para cada um deles?

Esta é a principal motivação do uso de arranjos. Podemos criar uma coleção de dados de um determinado tipo, acessando cada um deles pelo seu índice. O programa seguinte ilustra esta ideia:

Algoritmo 26: Gerenciador de 3 alunos

```c
#include <stdio.h>

int main(void)
{
  int mediaAlunos[1000]; // um arranjo de 1000 inteiros

  for (int i = 0; i < 1000; i++)
  {
      printf("Digite a media do aluno %d: ", i + 1);
      scanf("%d", &mediaAluno[i]);
  }

  // ...

  return 0;
}
```

Neste programa, criamos um vetor com capacidade de armazenar 1000 inteiros. Em seguida, utilizamos uma estrutura de repetição para ler cada um dos 1000 valores e armazená-los.

Perceba que, na primeira vez que este loop é executado, o valor lido é guardado no índice 0 de mediaAluno. Depois, no índice 1, e então no 2, 3... Até que se chegue no índice 999. Os índices de um arranjo de tamanho n vão de 0 até $n - 1$.

A definição de um arranjo, então, se dá da seguinte forma:

```
tipo nome[TAMANHO];
```

Seus valores podem ser modificados:

```
nome[0] = 10;
```

E podem ser utilizados em outras partes do código:

```
int a = nome[0] + nome[1];
printf("a = %d\n, nome[0] = %d\n", a, nome[0]);
```

Além disso, podem ter valores iniciais:

```
int valores[] = {456, 12, 9856, -2634, 84, 0, -13, 12, 1};
```

Perceba que, desta vez, não foi colocado o tamanho do vetor de forma explícita. Isto porque é possível inferir seu tamanho através da quantidade de valores na inicialização. Neste caso específico, o arranjo valores tem tamanho 9.

É importante observar que o tamanho de um vetor deve ser definido em tempo de compilação, e não em tempo de execução. Isto significa que o programa a seguir não é válido:

Algoritmo 27: Arranjo com tamanho definido em tempo de execução, da forma errada

```
#include <stdio.h>

int main(void)
{
  int tamanho;

  printf("Digite a quantidade de valores: ");
  scanf("%d", &tamanho);

  int valores[tamanho];

  // ...

  return 0;
}
```

É possível obter o efeito desejado com técnicas que serão abordadas algumas seções depois desta. Por enquanto, o que deve ser feito é criar um arranjo que seja grande o suficiente para

comportar diversos valores e não permitir que o usuário dê um valor maior do que o máximo:

Algoritmo 28: Arranjo com tamanho definido em tempo de execução, da forma certa

```c
#include <stdio.h>

int main(void)
{
  int tamanho;

  do
  {
      printf("Digite a quantidade de valores (minimo 1, maximo 1000): ");
      scanf("%d", &tamanho);
  }
  while (tamanho < 1 || tamanho > 1000);
  int valores[1000];

  for (int i = 0; i < tamanho; i++)
  {
      printf("Digite o %do valor: ", i + 1);
      scanf("%d", &valores[i]);
  }
  // ...

  return 0;
}
```