INTRODUÇÃO À PROGRAMAÇÃO

PROGRAMAÇÃO 101

Sobre este documento

Este livro é um material auxiliar para a primeira metade da disciplina Programação e Desenvolvimento de Software I (antigamente Algoritmos e Estruturas de Dados I) da Universidade Federal de Minas Gerais. É um material **gratuito** e não pode ser vendido em hipótese alguma!

O material apresentado neste documento tem caráter didático, tendo como foco desde pessoas que nunca tiveram contato com computação até pessoas que já programam uma linguagem de programação que não seja C.

O conteúdo é neste livro é resumido e deve ser usado como um **complemento às aulas**. Um livro completo abordando as matérias Programação e Desenvolvimento de Software I e Estruturas de Dados ainda será lançado, também gratuitamente.

O material ainda não está pronto, sujeito a alterações. Se qualquer modificação for feita, será feito um aviso durante a próxima aula. Sugestões e críticas são bem-vindas!

Desejamos a você bons estudos!

1 Brevíssima história

O início da computação deu-se na década de 1940, com a criação do primeiro computador eletrônico digital. Batizado de *Electronic Numerical Integrator and Computer* (ou *ENIAC*), foi principalmente utilizado para cálculos balísticos, reduzindo trabalhos manuais que podiam durar dúzias de horas para meros trinta segundos.

Como qualquer computador, o *ENIAC* também deveria ser programado, e este trabalho coube às 6 primeiras programadoras da história. Suas histórias não são amplamente divulgadas, tendo sido documentadas só recentemente, pelo *ENIAC Programmers Project*. O documentário está disponível gratuitamente, em inglês.

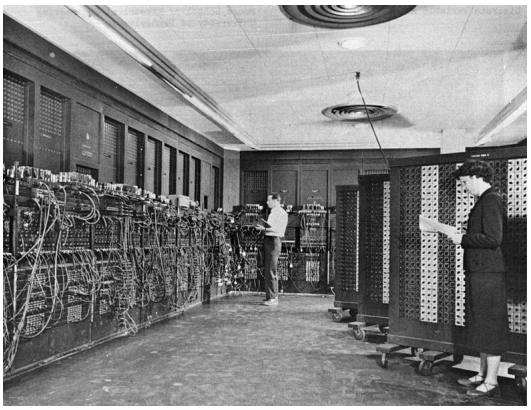

Figura 1: Parte do *ENIAC*.

O *ENIAC* era um computador enorme, de baixíssimo poder computacional (comparado com computadores atuais), mas nada disso tira o mérito de ter sido uma grande invenção. Como muitas tecnologias, foi criado por propósitos de guerra e, assim como várias dessas invenções, tornou-se parte do cotidiano de muitas pessoas.

Também durante a mesma guerra, desta vez na Inglaterra, Alan Turing trabalhava num projeto que deveria descriptografar mensagens alemãs enviadas por meios sem fio. Este era um trabalho muito difícil para ser feito manualmente, visto a quantidade de configurações diferentes possíveis para utilizadas pelos alemães para se criptografar uma mensagem.

A máquina construída por Turing recebeu o nome de *Bombe* e obteve sucesso na quebra das mensagens, ao descobrir um padrão nas mensagens alemãs. Por esta e outras criações, Turing é conhecido como o Pai da Computação. Seu nome é utilizado no principal prêmio da área, o Prêmio Turing, o "Nobel da computação".

Embora o sucesso de Alan tenha sido fundamental para os países Aliados, foi condenado à injeções de hormônio pela sua orientação sexual. Não muito tempo

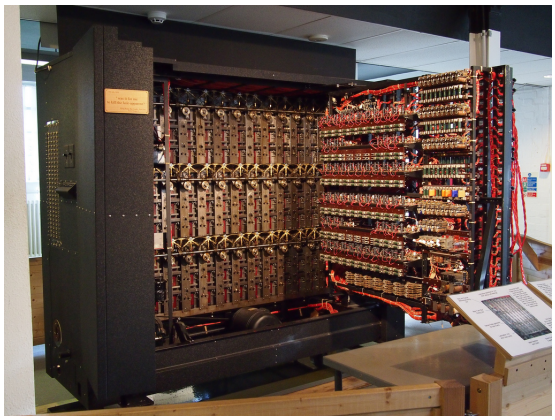

Figura 2: Bomba eletromecânica (*Bombe*).

depois, morreu envenenado, supostamente um suicídio, com cianeto. Esta pequena parte de sua vida se tornou um filme, *O Jogo da Imitação*, em 2014. Embora romantizado, é um ótimo filme!

A história da computação é recente, mas tem vários episódios interessantes. Conhecê-la não é fundamental, mas pode ajudar a entender o desenvolvimento do ferramental e pode fazer com que você se interesse ainda mais por esta maravilhosa área de conhecimento!

2 Introdução

Computação muito se trata da resolução de problemas do dia-a-dia. Para tanto, é necessário conhecimento de alguns conceitos. Trataremos primeiramente de conhecer o que é um algoritmo e, após entendermos a ideia, vamos escrever um pequeno algoritmo utilizando a linguagem C, que é a linguagem utilizada na matéria AEDs I, II e III.

2.1 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).

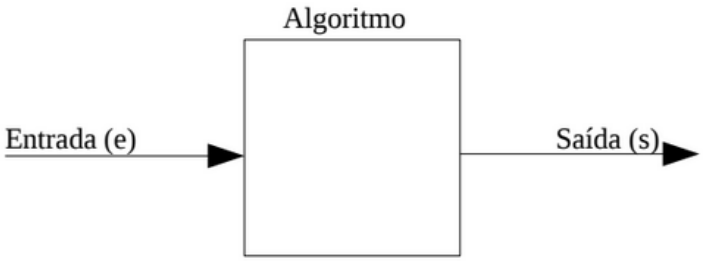

Figura 3: Abstração do conceito de algoritmo.

A figura 3 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).

2.2 Olá mundo!

Conhecida a definição de algoritmo, podemos escrever nosso primeiro programa em C. Para escrevermos o programa, precisamos de um **editor de código** e um **compilador** ou uma **IDE**, para que possamos transformar o código que escrevemos na linguagem C nos 0's e 1's que nossas máquinas entendem. Por termos de facilidade, recomendamos que use uma **IDE**, mas usar um **editor de código** e **compilador** pode ser melhor para o aprendizado.

Se você não sabe o que é um **editor de código**, **compilador** ou **IDE**, vá para a seção 7 (Definições).

O seguinte código, após compilado e executado, mostrará a mensagem *Ola mundo!*

Algoritmo 2: Olá mundo!

```c
#include <stdio.h>

int main(void)
{
  printf("Ola mundo!\n");
  return 0;
}
```

Existe uma lenda que diz que quem nunca praticou o código acima nunca dominará a linguagem!

3 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 2.2 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.

3.1 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). 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.

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

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

```
1 frase saudacao = "Ola mundo!";
2 numero tempo = 14000000000;
3 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.

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!

3.2 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.

Foram apresentados poucos tipos de dado e de maneira simplificada nesta tabela com propósito de não complicar o conteúdo apresentado. Uma tabela completa está presente na seção **??**.

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*.

Veja a seção 7 (Definições) se o nome *byte* for desconhecido.

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

Algoritmo 3: 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 4: 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 5: 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;
}
```

Nos algoritmos acima vemos algo que ainda não conhecemos, como estes "%d"ou "%c"dentro do que está impresso. Pode ser fácil presumir que no lugar destes caracteres serão colocados os valores designados. Isso será explicado na próxima seção.

É recomendado que o leitor pratique os conceitos demonstrados acima para aumentar seu domínio sobre o conteúdo apresentado e possa avançar sem grandes dificuldades. Explore os códigos acima, modifique-os, tente causar erros e entendê-los (pesquise!).

3.3 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:

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

0x ao início denota um valor hexadecimal.

Também pode-se escrever um valor na base octal:

```c
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 6: 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 7: Imprimindo um *char* como um *int*

```
#include <stdio.h>

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

4 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.

4.1 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 3.2 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 8: Utilizando caracteres escapados

```
#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;
}
```

4.2 Entrada de dados

Se na subseção 4.1 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 4.1 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 9: Lendo um dado do usuário

```
#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 10: 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 Pré-AEDS para reforçar o que foi aprendido até então.

5 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.

5.1 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 11: Condição

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int idade;
6
7    printf("Digite sua idade: ");
8    scanf("%d", &idade);
9
10   if(idade < 18)
11   {
12     printf("Voce e menor de idade!\n");
13   }
14   else
15   {
16     printf("Voce e maior de idade!\n");
17   }
18
19   return 0;
20 }
```

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 12: Comparação mal-escrita

```
#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 13: 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 14: 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 15: 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 15 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 16: Estruturas if - else if - else

```c
#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 16 vemos uma corrente de condições que são excludentes. Se o mesmo programa não fosse feito desta maneira, usando somente *if*s, com qualquer nota $\geq$ 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.

5.2 Estruturas de repetição

ESTA SEÇÃO SERÁ REVISADA, E NÃO CONSTA COMO VERSÃO FINAL

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 17: 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 17 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 $contador \leq 3$;
3. A expressão condicional $contador \leq 3$ 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 $contador \leq 3$ 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 $contador \leq 3$ 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
Fim
```

Listing 1: Saída do algoritmo 17

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 18: 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 19: 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 20 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 20: 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 21: 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 22: 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 6.

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 22 mostra exatamente isso.

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

Algoritmo 23: 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.

6 Arranjos (vetores)

Esta seção apresenta o conceito de arranjos (também chamados de vetores por alguns). Ele expande o conceito de variáveis e utiliza o ferramental aprendido na seção anterior. Não se espera um domínio completo dos dois conceitos, mas noção e prática dos dois.

O que mostraremos agora é simples, comparado ao conteúdo anterior, mas simplificaremos o conteúdo que está sendo passado agora para melhor entendimento e mais tarde, na seção **??**, traremos de volta o conceito e explicaremos-no de maneira mais aprofundada.

6.1 Estendendo o conceito de variáveis

Para entendermos arranjo, expandiremos nosso conceito de variáveis. Sabemos, até então, que variáveis tem **nome, tipo e valor**. Mais profundamente, o valor desta variável está na memória de seu computador, codificada com 0's e 1's.

Para entendermos este primeiro novo conceito, imaginemos uma memória com capacidade de 16B (sim, é muito pouco, mas é para a demonstração). Podemos dividir esta memória em 16 partes de 1B cada e, por conveniência, precisamos de uma maneira de identificar cada uma destas partes. Assim, surgem os endereços.

Imagine uma rua, e nesta rua existem 16 casas, numeradas de 0 a 15. Em cada uma destas casas, existem 8 cômodos, lado a lado, nos quais pode ou não ter uma única pessoa, e este conjunto de pessoas forma um número na base binária de contagem: se existe alguém em um cômodo, é 1 e, senão, 0.

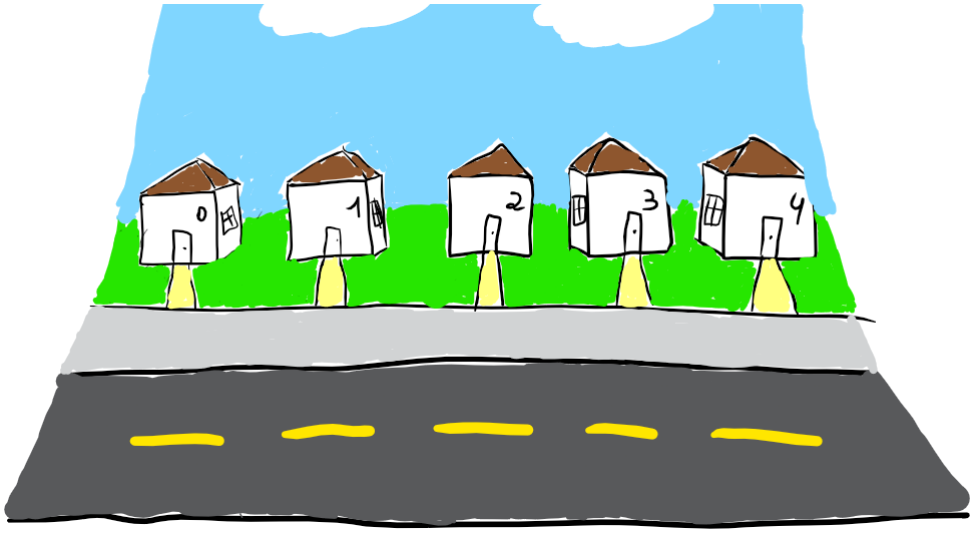

Figura 4: Asbtração da "rua memória RAM".

A sua memória funciona de maneira semelhante: se queremos acessar uma das

casas para descobrir ou modificar seu valor, acessamos seu endereço e realizamos a operação desejada. Então, para acessá-las, precisamos de seu endereço. No nosso caso, a "rua"é a nossa memória RAM e seu endereço é denotado unicamente pelo seu número. Assim, se desejamos obter o conteúdo da primeira "casa", acessamos o endereço 0. Se desejamos modificar o conteúdo da "casa"13, acessamos o endereço 12.

O número de casas nesta "rua"é pequeno comparado aos computadores atuais. Se hoje temos um computador com, digamos, 4GB de memória RAM, esta memória seria como uma única rua muito, muito longa, com 2^{32} casas, numeradas de 0 a $2^{32} - 1$.

COLOCAR IMAGEM DE ABSTRAÇÃO DA MEMÓRIA AQUI

Como uma forma de abstração, ao invés de acessarmos diretamente um endereço, damos nome a este endereço. Por exemplo, se criamos uma variável *contador*, ela possui um endereço na memória, mas decorar este endereço pode ser uma tarefa difícil e então temos acesso à este conteúdo utilizando nomes.

COLOCAR IMAGEM DE ABSTRAÇÃO DE NOMES AQUI

Uma outra abstração que aprendemos para variáveis é que elas têm tipo, mas seu tipo não está escrito na memória, somente seu valor. Seu tipo é, na verdade, a maneira como devemos interpretar aqueles dados na memória. Isso foi apresentado na subseção 3.3, mas superficialmente.

Por isso, se temos escrito na memória o valor 01000001_2 (65_{10}) podemos tanto interpretá-lo como o valor inteiro 65 ou como o caractere 'A', como definido pela tabela ASCII. Por isso explicitamos o tipo da variável, pois assim sempre saberemos como interpretar este conjunto de 8 bits (1 byte). Mas como ao final ainda temos um conjunto de bits, podemos interpretar da maneira que quisermos, convertendo de um tipo para outro.

Veja também que o tipo interfere também em quantos bytes são lidos. Pela tabela 1 sabemos que uma variável do tipo *int* ocupa 4B na memória, enquanto uma do tipo *char* ocupa somente 1B. Assim, o tipo interfere em quantos bytes serão lidos (ou quantas "casas"serão "abertas").

6.2 Arranjos

Agora que o conceito de variáveis foi expandido e sabemos melhor como isto funciona na memória, pode ser mais fácil entender este conceito, embora simples, é melhor que se entenda como funciona em seu computador, e não superficialmente apenas.

Imagine que você precisa escrever um programa, e este terá um número grande de dados, e você precisa armazenar todos! Da maneira como aprendemos até agora, podemos criar uma variável para cada um dos dados, mas rapidamente perde-se o controle do que está acontecendo no programa, tornando extremamente complicado mantê-lo.

Para facilitar o trabalho, criamos um arranjo. Digamos que estamos resolvendo um problema que precise armazenar 100 valores inteiros. Agora, em vez de

criar 100 variáveis, uma a uma, criaremos "uma"variável capaz de armazenar 100 valores. Escreveremos da seguinte maneira:

```
int valores[100];
```

Assim como uma variável, escrevemos seu tipo e seu nome. Em seguida, entre colchetes ([e]) explicitamos o **tamanho** deste arranjo. Uma vez definido, este tamanho não pode ser alterado em momento algum. Outra limitação é que o tamanho deve ser conhecido, isto é, não pode ser obtido do usuário. Então, se o tamanho depender do usuário, definimos um tamanho máximo para o arranjo e criamos um arranjo com este tamanho máximo, sem usar toda sua capacidade na maioria das vezes.

Semelhante aos endereços de memória, se desejamos fazer algo com o primeiro dado deste arranjo, devemos acessá-lo na posição 0:

```
int valores[100];
valores[0] = 10;
```

Se desejamos imprimí-lo:

```
int valores[100];
valores[0] = 10;
printf("%d\n", valores[0]);
```

7 Definições

Nesta seção encontra-se definições breves de termos abordados neste documento que não possuem seção ou subseção para os mesmos. Os itens desta seção estão dispostos alfabeticamente.

7.1 Bit

Um *bit* (ou ***b**inary dig**it***) é uma unidade binária que pode assumir os valores 0 ou 1. Pode-se escrever qualquer valor numérico inteiro com bits, utilizando-se da base binária de contagem.

7.2 Bloco

Neste documento, um **bloco de código** é um algoritmo denotado entre os caracteres { e }.

Algoritmo 24: Bloco de código

```
#include <stdio.h>

int main(void)
{
  int valor;

  {
    valor = 1;
    int numero = 2;
    printf("%d, %d\n", valor, numero);
  }

  printf("%d\n", valor);

  return 0;
}
```

Tudo o que existe no *escopo* externo ao bloco existe também no bloco, mas aquilo que só existe dentro do bloco (dentro do *escopo*) do bloco só existe dentro do bloco.

7.3 Byte

Um *byte* é um conjunto de 8 bits (veja bit nesta seção). É uma unidade de contagem de memória. O valor 50 (da base decimal) pode ser escrito como o valor de 1 *byte* 00110010_2 (notação da base 2).

7.4 Compilador

Um **compilador** é um programa que, em suma, traduz o seu programa escrito numa linguagem de programação para o código de máquina. São exemplos de

compiladores:

1. GCC
2. Clang
3. MinGW

7.5 Editor de código

Um **editor de código** é um programa feito para a escrita de código. Destaca a sintaxe do código, mostra macações, entre várias outras funcionalidades. Embora não seja necessário, pode ajudar bastante no desenvolvimento de alguma aplicação. São exemplos de editores de código:

1. Atom
2. Sublime Text 3
3. Vim

7.6 Escopo

TODOVariáveis existem dentro de um determinado *escopo*. TODO

7.7 IDE

Um **IDE** ou *Integrated Development Environment* é um programa de desenvolvimento que combina um editor de código, compilador e depuradores. Facilita o desenvolvimento de aplicações e auxilia no encontro de erros. São exemplos:

1. Dev-C++
2. Eclipse