Migrando de *Pascal* para *C*

Marcos Castilho

Versão 1.1
Setembro de 2024

Sobre o autor:

Marcos Alexandre Castilho é Professor Titular da Universidade Federal do Paraná. É Bacharel em Matemática pela Universidade Estadual Paulista Júlio de Mesquita Filho (1987), mestre em Ciência da Computação pela Universidade Federal de Minas Gerais (1991) e doutor em Informática pela Université Paul Sabatier (Toulouse 3, França, 1998). É professor do Departamento de Informática da UFPR deste 1992.

Este livro foi escrito usando exclusivamente software livre: Sistema Operacional GNU/Linux Mint, compilador de texto LaTeX, editor vim, compiladores *Free Pascal* e *GCC*, dentre outros.

Este texto está em construção.

Sumário

1 Introdução

Este texto é um material complementar à disciplina *CI1001 - Programação I*, ministrada nos cursos de Bacharelado em Informática Biomédica e Bacharelado em Ciência da Computação da Universidade Federal do Paraná.

Para contextualizar a motivação deste livro é importante esclarecer que a disciplina CI1001 é ministrada no segundo período destes cursos e seu pré-requisito é CI1055 - Algoritmos e Estruturas de Dados I. Nesta última adota-se a linguagem *Pascal* para a parte prática da disciplina.

Já no segundo período, a disciplina CI1001 adota a linguagem *C* para as implementações de alguns tipos abstratos de dados básicos, tais como listas, pilhas e filas. Por isso nossos estudantes devem fazer a transição de uma linguagem para a outra. Acreditamos que o aprendizado supervisionado da mundança para uma segunda linguagem é benéfica e facilita o aprendizado de futuras outras linguagens.

Este preâmbulo justifica este texto, cujo principal objetivo é facilitar a migração para a linguagem *C* para quem já sabe *Pascal*.

A intenção deste texto não é fornecer uma referência completa para a linguagem *C*, para isto existe farta literatura. A principal referência é o famoso livro dos criadores da linguagem, Brian Kernighan e Dennis Ritchie [KR88].

A ideia é discutirmos algumas diferenças fundamentais entre as duas linguagens para que possa servir para uma transição mais tranquila entre elas.

Ambas linguagens são estruturadas, mas foram concebidas para fins diferentes. Enquanto a linguagem *Pascal* foi projetada para ser uma linguagem didática, voltada para o aprendizado de programação de computadores, a linguagem *C* foi criada para que o sistema Unix pudesse existir. Uma mesma equipe de pesquisadores projetou e desenvolveu na mesma época tanto Unix quanto *C*. *Pascal* foi projetada por Niklaus Wirth em 1970, mais ou menos na mesma época em que *C* também foi criada.

Cada linguagem de programação tem suas características próprias, elas diferem em aspectos tanto sintáticos quanto semânticos. No entanto, os fundamentos de construção de algoritmos são os mesmos. Todas elas possuem maneiras de receber entradas, produzirem saídas, desviarem ou repetirem código.

Por este motivo, neste material consideramos que os leitores ou leitoras dominam a arte de desenvolvimento de códigos estruturados em *Pascal* e não nos preocuparemos com esta problemática. Nos concentraremos somente nas diferenças básicas entre as duas linguagens.

A maneira de apresentar os conceitos será comparar diversas estruturas em ambas linguagens e focar nas diferenças não apenas sintáticas, mas sobretudo nas semânticas, pois são bastante diferentes.

As duas linguagens fazem parte do mesmo paradigma de linguagem de programação, linguagens imperativas e estruturadas. Por esta razão, as duas linguagens contêm comandos com funcionalidades semelhantes, porém com sintaxe e semântica diferente.

Consideramos importante entender estas diferenças, pois uma vez que esta etapa está vencida de uma primeira linguagem para uma segunda linguagem, no caso aqui *Pascal* e *C*, o aprendizado de qualquer outra linguagem de programação será muito mais produtivo.

Omitiremos muitos detalhes da linguagem *C*, pois foge do escopo deste material. A apresentação terá foco nas novidades e por isso usaremos frequentemente exemplos de problemas simples e clássicos. Pretendemos escrever até onde paramos na disciplina de Algoritmos e Estruturas de Dados I, a saber: até vetores e registros, incluindo funções também.

Uma das principais diferenças é que, por ter fins didáticos, a linguagem *Pascal* "esconde" deliberadamente uma série de coisas do aprendiz, enquanto que a linguagem *C* torna tudo explícito, o que é útil no desenvolvimento de um sistema operacional como o Unix. O compilador *C* tem por filosofia dar total liberdade para o programador, por isso este deve ter muita responsabilidade e saber muito bem o que está fazendo para não produzir códigos incorretos.

Desta forma, recomenda-se consulta à farta literatura que apresenta em profundidade aspectos semânticos dos elementos das linguagens aqui abordadas. Os e as estudantes devem ter por costume buscar outras fontes e conquistar autonomia para enfrentar situações das mais diversas em programação de computadores.

2 Hello world!

Praticamente todos os textos introdutórios de linguagens de programação iniciam com o clássico *hello world* e não fugiremos deste padrão. Os leitores e leitoras poderão perceber que as diferenças entre *Pascal* e *C* já aparecem, não apenas do ponto de vista puramente sintático, mas sobretudo, semântico.

A figura 1-(a) contém um programa *hello world* em *Pascal* e na figura 1-(b) mostramos o mesmo programa codificado em *C*.

```
1 program helloworld;
2
3 begin
4     writeln ('Hello world!');
5 end.
```

(a)

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

(b)

Figura 1: Hello world em *Pascal* e em *C*.

Em *Pascal* um programa é constituído de um cabeçalho e um programa principal. Em *C* não existe a noção propriamente dita de programa principal. Esta linguagem consiste de funções e variáveis. As funções contêm sentenças que por sua vez alteram os valores das variáveis.

Em *C*, *main()* é uma função como qualquer outra função da linguagem. Ela apenas difere das outras pois, ao se invocar um programa executável, ele inicia a execução por ela. Por este motivo, todo programa em *C* deve obrigatoriamente possuir uma função *main()* implementada.

No nosso exemplo, a função *main()* não recebe nenhum argumento, o que pode ser observado pelos parênteses (abre e fecha), e retorna um valor do tipo `int`, que é o equivalente de `longint` nas implementações recentes do *Free Pascal*.[1]

Uma pergunta interessante é: para quem a função *main()* retorna um valor? Retorna para o sistema operacional. Nos sistemas baseados em Unix, e aqui vamos usar sempre o *Linux*, isto é percebido como o *status de saída* do programa executado. Neste exemplo optamos por retornar um zero, pois costumeiramente nos sistemas Unix este valor indica que o programa executou a contento.

Do ponto de vista prático, *printf* exerce a mesma função do comando *write* do *Pascal*. Mas existe uma diferença relevante, a saber:

- Em *Pascal*, *write* e *writeln* são comandos nativos da linguagem.[2] Em *C*, *printf* é uma função, que por sinal não é nativa. Ela está definida em uma biblioteca da

[1] A bem da verdade, a quantidade de bytes que um determinado tipo usa é dependente da versão do compilador, do sistema operacional e do barramento da máquina. Os PC's atuais usam barramento de 64 bits, enquanto que máquinas mais antigas, ou até mesmo dispositivos embarcados, podem ter barramentos mais limitados. Por isso é sempre bom consultar estas informações para seu sistema em particular.

[2] Originalmente faziam parte das bibliotecas "input" e "output".

linguagem, denominada `stdio.h`. Por este motivo a primeira linha do programa informa ao compilador *C* que deve incluir esta biblioteca para que a função *printf* seja conhecida.

No caso, *printf* significa imprimir formatado. Neste programa o formato é bem simples e consiste somente da mensagem. Notem o uso do `\n` no final dela, significa uma mundaça de linha. Em termos práticos, tem o mesmo efeito de um comando *writeln*. Falaremos mais sobre comandos de entrada e saída na seção 8.

Outra diferença semântica relevante é que um programa em *Pascal* executa até o seu final. Em *C*, o programa termina tão logo um comando *return* seja encontrado. Neste caso não faz diferença pois o *return* está na última linha do programa.

As demais diferenças são meramente sintáticas. Um bloco em *Pascal* é definido entre as palavras *begin* e *end*. Para o programa principal, o *end* deve ser seguido de um ponto final para indicar o término do programa. Para funções e procedimentos, o *end* deve vir seguido de um ponto-e-vírgula. Em *C*, os blocos são definidos entre os símbolos de abre chave `{` e fecha chave `}`.

Notem também que em *C* os textos que aparecem nas estruturas de impressão são delimitados por aspas duplas, enquanto que em *Pascal* são delimitados por aspas simples.

As diferenças por enquanto param aqui. O restante dos conceitos são os mesmos para estes programas:

- Os dois compiladores (*Pascal* e *C*) ignoram espaços em branco, tabulações e linhas em branco;
- O estilo de programação, seja qual for, deve usar adequadamente indentação para facilitar a leitura do código;
- O fluxo de execução de um programa é o mesmo para as duas linguagens: de cima para baixo e da esquerda para a direita, no caso de haver mais de um comando na mesma linha;
- Evidentemente, algumas estruturas podem fazer com que este fluxo seja alterado, notadamente os comandos de desvio, bem como comandos de repetição e chamadas de funções.
- Todos os comandos são terminados por um ponto-e-vírgula.

3 Compilando programas simples

Partimos do princípio que os leitores e as leitoras sabem compilar um programa em *Free Pascal* no Linux. O programa da figura 1-(a) é compilado assim:

```
$ fpc helloworld.pas
Free Pascal Compiler version 3.2.0+dfsg-12 [2021/01/25] for x86_64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling helloworld.pas
Linking helloworld
5 lines compiled, 0.2 sec
```

O nome do executável tem o mesmo nome do programa fonte sem a extensão `.pas`. Logo, para executar, basta digitar `./helloworld`.

Em *C* usaremos o compilador livre do projeto GNU, o `gcc`. Para compilar este programa do modo mais simples faz-se assim:

```
$ gcc helloworld.c
$ ls
a.out  helloworld.c  helloworld.o
```

Para este programa particular, o compilador atuou silenciosamente e não produziu nenhuma saída visível. Mas ele gerou um arquivo de nome `a.out`. Ignorem por enquanto o arquivo `helloworld.o`, que aliás também é produzido pelo *Pascal*.

Este nome, `a.out`, é histórico, procurem saber a respeito procurando na wikipedia, por exemplo. Para executar o programa fazemos assim, evidentemente:

```
$ ./a.out
Hello world!
```

Para gerar um arquivo executável com um nome diferente pode-se usar uma opção do compilador, que é a opção `-o` seguido do nome desejado. Outras opções ajudam a melhorar a interação com o `gcc`. Uma boa opção é `-Wall`, que ativa o modo de avisos (*warnings*) para que o compilador seja um tanto "chato" e avise o programador de qualquer problema que ele detecte automaticamente. Desta forma poderíamos fazer conforme abaixo e em seguida executar `./helloworld`:

```
$ gcc -o helloworld -Wall helloworld.c
$ ls
helloworld  helloworld.c  helloworld.o
```

É interessante observar que a linguagem *C* é muito menos exigente do que a linguagem *Pascal* em alguns sentidos, conforme veremos no decorrer deste texto. Por isso, ativar a opção `-Wall` é bastante útil para aprendizes, mas também para programadores experientes.

Isto se deve às diferentes filosofias das linguagens. Em geral, os compiladores *C* tendem a "confiar" nos programadores, aceitando a princípio construções que, em *Pascal*, não seria aceito. Esta diferença é importante para novatos em *C*, pode-se produzir erros graves de lógica pelo uso errado de uma construção sintaticamente aceita. Logo veremos exemplos disso.

4 Variáveis e tipos

O termo "variável" indica um símbolo que representa um espaço na memória onde é possível armazenar valores. A quantidade de espaço na memória utilizada por uma variável depende do seu tipo. Por exemplo, variáveis do tipo inteiro podem utilizar 8, 16, 32 ou 64 bits, enquanto que variáveis do tipo caractere utilizam 8 bits.

Além de indicar a quantidade de bytes necessária para armazenar uma variável, o tipo desta variável também indica o conjunto de operações válidas para ela. Por exemplo, para uma variável do tipo inteiro, o conjunto de operações válidas incluem soma, subtração, divisão e multiplicação (entre outros).

Tanto na linguagem *Pascal* quanto na linguagem *C* é necessário declarar uma variável indicando o seu tipo, por isso dizemos que as linguagens são tipadas.

Uma diferença muito importante entre *Pascal* e *C* é que a primeira é uma linguagem fortemente tipada, ou seja, as variáveis são restritas às operações do tipo daquela variável. Por exemplo, a operação de soma de inteiros usa a adição tradicional na matemática, enquanto que uma soma de caracteres pode variar de uma linguagem para outra. Neste sentido, em *Pascal* somar caracteres resulta na concatenação dos caracteres, enquanto que na linguagem *C* resulta na soma dos valores respectivos da tabela ASCII.

Em todo caso, em *Pascal* não é possível misturar tipos indiscriminadamente, sendo bastante rígida quando se faz conversões de tipo. Veremos isso em breve. Já na linguagem *C* uma variável nem sempre é restrita às operações do seu tipo. Isto permite operações muito confusas como por exemplo somar um a uma variável do tipo caractere.

Os tipos em linguagens de programação servem para indicar como os bytes serão interpretados pelo compilador. O conceito é o mesmo em ambas linguagens. Por isso, os tipos variam de representações de inteiros, reais ou caracteres apenas pela maneira em que os bytes são vistos pelo compilador: quantidade de bytes e uso ou não do sinal, se a representação é de complemento de dois, de ponto flutuante ou outras.

A linguagem *Pascal* tem como tipos básicos:

- Ordinais (`char, integer, longint, smallint, int64, qword,` ...);
- Reais (`real, double,` ...);
- Palavras (`string`);
- Booleanos (`boolean`);
- Arrays (`array`) e registros (`record`).

Em *C* a nomenclatura difere, e para os tipos elementares ordinais e reais a diferença básica com relação ao *Pascal* está no nome:

- Ordinais (`char, int, short, unsigned int,` ...);
- Reais (`float, double, long double,` ...);

Trataremos de arrays, strings e registros em seções mais a frente neste material. Mas enfatizamos que o tipo booleano não é nativo da linguagem *C*[3].

Notem acima que o tipo `char` é tratado como um tipo inteiro, tanto em *Pascal* quanto em *C*.

Assim, pode-se declarar variáveis, repectivamente em *Pascal* e em *C* como nos seguintes exemplos (notem a diferença sintática de se fazer comentários no código em cada uma delas):

```
1 (* declarando variaveis em Pascal *)
2 var
3     dia, ano, mes: integer;
4     preco, media: real;
5     letra: char;
```

```
1 /* declarando variaveis em C */
2     int dia, ano, mes;
3     float preco, media;
4     char letra;
```

Percebam que em *C* não é necessário o uso da palavra reservada `var`.

Mais a frente falaremos da noção de escopo de uma variável. Em *Pascal* elas podem ser globais ou locais a uma função. Em *C* também, mas em versões recentes do compilador elas podem ter o escopo limitado até mesmo a um determinado bloco.

Uma diferença importante nas duas linguagens é que em versões recentes de *C* pode-se declarar variáveis em qualquer parte de um programa, o que é proibido em *Pascal*, na qual as variáveis devem ser declaradas antes do `begin ... end`.

Um detalhe sintático adicional é que, contrariamente ao *Pascal*, na linguagem *C* é feita a distinção entre identificadores maiúsculos e minúsculos (a linguagem é "*case sensitive*"). Desta forma, em *C*, duas variáveis `n` e `N` são diferentes, enquanto que em *Pascal* seriam a mesma variável.

[3]Este tipo existe desde a versão C99, mas não é como em *Pascal*. Escreveremos mais sobre isto em breve.

5 Atribuição de valores para variáveis

Atribuir valores para variáveis é uma operação básica em programação. Porém, os aspectos semânticos são diferentes nas duas linguagens *Pascal* e *C*. Um aprendiz tende a perceber como uma mera variação sintática, mas não é.

Primeiramente, em *Pascal*, uma atribuição é feita com o uso do `:=`, enquanto que em *C* ela é feita com o uso de um `=`, sem os dois pontos (`:`). Mas a diferença é mais do que aparenta.

Em *Pascal* uma atribuição é um comando. O código `x:= <expressao>;` resulta na alteração do valor da variável `x` após a expressão ter sido completamente avaliada. Em *C* o mesmo ocorre, porém com uma diferença semântica importante.

Em *C* os operadores retornam valores e o `=` é um operador de atribuição (e não um comando) que portanto retorna um valor. Desta maneira, em uma atribuição tal como `x = 2`, além da variável `x` receber o valor 2, a expressão `x = 2` retorna o valor 2.

A questão é que em *C* este valor retornado pode não ser utilizado. No caso acima ele não foi. Mas poderia ter sido utilizado, por exemplo, para atribuir o valor retornado para outra atribuição, tal como `y = x = 2`. As operações ocorrem da direita para a esquerda, assim primeiro se resolve `x = 2`, que resulta no valor 2, que por sua vez é atribuído a `y`, que naturalmente retorna o valor 2 também, o qual não foi utilizado.

Este fato pode gerar problemas para iniciantes, como será visto na seção 10.

Em *C* pode-se inicializar uma variável no momento de sua declaração.[4] Desta maneira podemos escrever códigos como este aqui:

```
1    int dia = 30, mes = 12, ano = 2023;
2    float temperatura = 27.5, media = 5.5;
```

Outros operadores de atribuição existem em *C* para facilitar a redação de códigos. A tabela a seguir mostra alguns destes operadores e sua semântica:

`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

O símbolo `%` é o operador de resto de divisão inteira equivalente ao `mod` em *Pascal*.

Ainda, a linguagem *C* também possui alguns operadores especiais tais como estes:

`x++`	`x = x + 1`
`++x`	`x = x + 1`
`x--`	`x = x - 1`
`--x`	`x = x - 1`

[4]Na verdade, o compilador *Free Pascal* também permite, embora isso não faça parte do *Pascal* padrão tal como definido originalmente.

Embora possam parecer redundantes, as construções `x++` e `++x` têm semânticas diferentes. Nos dois casos o retono é o mesmo valor, isto é, `x+1`. A diferença está no momento em que o valor do retorno é utilizado em uma outra operação tal como `y = x++` ou em `y = ++x`.

Para exemplificar, considere o seguinte trecho de código:

```
1    int x, y, z;
2
3    x = 2;
4    y = x++;
5    z = ++x;
```

No primeiro caso, `y` recebe o valor de x e em seguida o valor de `x` é incrementado de 1. No segundo caso, primeiramente o valor de `x` é incrementado de 1 e em seguida este valor define o retorno da expressão. Por este motivo, `y = x++` resulta no valor 2 para `y`, enquanto que no segundo caso, `z = ++x`, `z` recebe o valor 3.

O valor de `y` é 2, enquanto que o valor de `z` é 3. Mas, observem atentamente, o valor de `x` será sempre 3, tanto no final da linha 4 quanto no final da linha 5.

Um ponto importante sobre variáveis é relacionado com o escopo delas, mas para poder explicar melhor o assunto temos que falar antes a organização da memória, o que será feito na seção 17. Por isso a explicação sobre escopo ficará para a seção 18.

6 Const e Define

É comum as linguagens de programação permitirem declarações de constantes, e isso ocorre tanto em *Pascal* quanto em *C*.

Embora utilizemos o termo "constante" o conceito é um pouco diferente. Vejamos primeiramente como era feito em *Pascal*.

```
1  const
2       MAX = 10;
3       PI = 3.14159;
4       CINCO = 2 + 3; (* conta feita pelo compilador *)
5
6  begin
7       writeln (MAX + 1,' ', PI * PI,' ', CINCO);
8  end.
```

A regra em *Pascal* é que o compilador deve ser capaz de avaliar a expressão na declaração da constante em tempo de compilação. Quando uma declaração de constante foi previamente declarada e posteriormente é usada no código, o compilador insere o valor atual da constante ao invés do nome dela.

Em *C*, existe o conceito de *define* que permite obter o mesmo resultado, mas como o funcionamento é diferente, também aceita algumas coisas mais sofisticadas. Vejamos o básico primeiro reescrevendo em *C* o código acima.

```
1  #define MAX 10
2  #define PI 3.14159
3  #define CINCO = 2 + 3 /* conta sera feita na execucao */
4
5  int main (){
6       printf ("%d %f %d\n", MAX + 1, PI * PI, CINCO);
7  }
```

O compilador *C* faz a substituição literal de uma *string* por outra, por exemplo, em todo lugar que aparece `MAX` este nome é substituído por `10`. Vejam o caso de `CINCO` ser substituído por `2 + 3` na linha 3 do código acima.

Isto permite fazer *defines* de coisas tais como no exemplo abaixo.

```
1  #include <stdio.h>
2
3  #define MAX(a, b) ((a > b) ? a : b)
4
5  int main (){
6       int a, b;
7
8       scanf ("%d %d", &a, &b);
9       printf ("%d\n", MAX (a, b));
10 }
```

O pré-processador trocou `MAX (a, b)` por `((a > b) ? a : b)`, então o que o compilador compila na verdade é esta última expressão.

Este programa imprime o maior valor dentre os números lidos, usando um operador ternário válido na linguagem *C*. Este operador funciona como uma espécie de *if/else* e é constituído por três elementos: um teste (`a > b`), que se for verdadeiro retorna o valor de `a`, mas se for falso retorna o valor de `b`.

No exemplo acima, poderíamos ter optado por construir uma função que faria a mesma coisa, mas optamos pelo uso do *define* para mostrar o conceito. Neste caso, a escolha do uso do *define* mostra a flexibilidade da linguagem, pois torna o código ligeiramente mais eficiente, mantendo a legibilidade, uma vez que o uso do nome facilita a leitura e ao mesmo tempo evita uma chamada de função. Isto é útil para códigos simples tal como o apresentado.

A principal diferença é que o compilador *C* faz um pré-processamento antes do processo de compilação propriamente dito. O pré-processador processa "macros" antes da compilação, fazendo inclusão de arquivos de cabeçalho (*headers*), expansões de macros, tais como no exemplo acima, permite compilação condicional, dentre outras coisas. Não faz parte deste texto comentar mais sobre ele, interessados devem procurar a respeito na literatura.

O interessante é que muitos programas podem fazer uso de macros permitindo a construção de programas de maneira bem interessante. O compilador *Free Pascal* não faz pré-compilação.

7 Seção de declaração de tipos

As duas linguagens permitem definir novos tipos de dados. Em *Pascal* isto é feito pela instrução *type*. Abaixo relembramos como era feito nesta linguagem.

```
type
    vetor = array [1..20] of real;
    inteiro = longint;
    ponto = record
            x: integer;
            y: integer;
    end;

var
    v: vetor;
    n: inteiro;
    p: ponto;
```

Em *C* a instrução é um *typedef*, que pode ser usado tal como no exemplo a seguir.[5]

```
typedef int inteiro;
typedef vetor int[20];
typedef struct ponto {
    int x;
    int y;
} ponto;

int main (){
    vetor v;
    inteiro n;
    ponto p;

    return 0;
}
```

Existem muitas formas de usar um *typedef*, os interessados podem procurar saber mais a respeito na literatura. Mas, no exemplo acima, usamos a sintaxe
`typedef <nome> tipo`.

[5] Vetores serão vistos na seção 19 e structs na seção 20.

8 Entrada e saída em *C*

Conforme já mencionamos, o compilador básico da linguagem *C* não dispõe de comandos de entrada e saída como na linguagem *Pascal*. É preciso incluir a biblioteca de entrada e saída (*Standard I/O*) através da instrução `#include <stdio.h>`.

Esta biblioteca disponibiliza uma família de funções que realizam entrada e saída, as duas de maior interesse por enquanto são: *scanf* (entrada) e *printf* (saída). Elas fazem o papel que o *read* e *write* fazem em *Pascal*. Vamos nos concentrar apenas nestas duas por enquanto.

A função *printf* (saída) significa saída formatada, razão da letra 'f' no nome *printf*. Inicialmente vale dizer que é uma função que tem um número variável de parâmetros. Ela produz a saída de acordo com o formato indicado pelo programador na saída padrão (terminal). As outras funções da família diferem, entre outras coisas, ao direcionar a saída não necessariamente para a saída padrão.

A função escreve a saída sob o controle de uma string que define o formato específico na qual os argumentos subsequentes são convertidos e em seguida impressos. Para exemplificar usaremos poucos exemplos com inteiros e floats, outros formatos podem ser posteriomente consultados e explorados em função da necessidade do programador.

Desta forma, considere o seguinte trecho de um programa codificado em *C*:

```
1 float a = 4;
2 int b = 2;
3
4 printf ("O valor de %f dividido por %d resulta em %f\n", a, b, a / b);
```

Observem que a função *printf* deste código tem 4 argumentos. O primeiro é a string de formatação, ela aparece entre as aspas duplas. O segundo argumento é a variável `a` (float), o terceiro é a variável `b` (int) e o quarto é a divisão de `a` por `b` (resulta em *float*, pois `a` é um *float*).

A string de formatação contém um texto terminado em `\n` que é um *newline* e significa mudar de linha. A mensagem em si contém textos que são literalmente impressos, mas contém também alguns símbolos precedidos de um porcento (`%`). A ordem em que estes aparecem é relevante, o primeiro é um `%f` o segundo é um `%d` e o terceiro é um `%f`. Eles são associados pela ordem com os parâmetros que sucedem a string de formatação, respectivamente `a`, `b` e `a/b`. Os símbolos contendo os `%` neste caso significam:

`%d`	uma das maneiras de imprimir um inteiro
`%f`	uma das maneiras de imprimir um *float*

Desta forma, ao executar a função *printf*, os símbolos com `%` são substituídos pelo valor da variável respectiva que aparece a partir do segundo parâmetro. Mas, o valor desta variável é impresso segundo o formato indicado pela tabela acima, isto é, o valor de `a` é impresso como um *float*, o valor de `b` como um inteiro e finalmente o valor de `a / b` como um *float*.

O resultado na saída padrão é:

```
O valor de 4.000000 dividido por 2 resulta em 2.000000
```

O programdor decide o formato de saída. Se ele quiser imprimir um valor float como um inteiro, basta trocar o `%f` por um `%d`. Mesmo que a variável continue sendo float, na saída aparece um inteiro. Ou vice-e-versa.

Em outras palavras, é preciso que o programador tenha responsabilidade e saiba o que está fazendo. O compilador *C* permite que o programador possa fazer praticamente tudo o que quiser, o que é uma filosofia totalmente diferente daquela da linguagem *Pascal*.

Uma vez apresentado o *printf* podemos mostrar a função básica para a leitura de dados do teclado, que é a *scanf*, entrada formatada. Assim como a primeira, ela também é uma função que tem um número variável de parâmetros e também usa a noção do formato controlado por uma string de formatação.

Desta forma, considere o seguinte programa codificado em *C*:

```
1 int main(){
2     float a;
3     int b;
4
5     scanf ("%f %d", &a, &b);
6     printf ("%f dividido por %d resulta em %f\n", a, b, a / b);
7
8     return 0;
9 }
```

A linha de interesse é evidentemente a que contém o *scanf* (linha 5). Notem que temos três argumentos, primeiro uma string de formatação (entre as aspas duplas) e em seguida dois parâmetros que são os endereços das variáveis `a` e `b`. Maiores detalhes na seção 14.

A string de formatação indica que o programador deseja fazer a leitura de um float (`%f`) e em seguida de um inteiro (`%d`).

A parte complicada é explicar os símbolos de e-comercial (`&`) que antecedem os identificadores das variáveis. Para explicar é necessário lembrar que *scanf* é uma função e, como explicaremos mais a frente neste texto (seção 13), esta função recebe parâmetros por cópia.

Voltando às explicações sobre passagem de parâmetros vistas na disciplina de Algoritmos e Estruturas de Dados I, no caso para a linguagem *Pascal*, é preciso lembrar que haviam duas maneiras de fazer passagem de parâmetros: por cópia ou por referência.

A linguagem *C* não possui a noção de passagem de parâmetros por referência tal como em *Pascal*. Todas as passagens de parâmetros são por cópia. A linguagem *Pascal*, como foi projetada para ser utilizada com fins didáticos, optou por "esconder" o real significado do que é uma referência para não complicar a vida do aprendiz.

O leitor ou a leitora deve recordar que quando se quer alterar o valor de uma variável dentro de uma função é necessário a passagem por referência. No caso de *Pascal*, ao se declarar o parâmetro na função, é necessário usar a palavra `var` assim: `procedure f (var a: real; var b: longint)`. Pois bem, a referência nada mais é do que a referência ao endereço da variável em questão.

O operador e-comercial (`&`) significa acessar o endereço da variável, e não o conteúdo dela. Desta forma, a função *scanf* recebe cópias dos endereços de `a` e de `b`, o que é visto quando se escreveu `&a` e `&b` no programa acima. A partir dos endereços, uma vez que o usuário digita valores na entrada padrão, estes valores são colocados nos endereços especificados, isto é, no lugar da memória onde estão as variáveis `a` e `b` da função *main*.

Isso não deveria ser uma surpresa ou algo mágico, pois sabemos que variáveis nada mais são do que nomes para endereços de memória. Elas servem para o programador poder abstrair os endereços e usar nomes apropriados para que um ser humano possa ler e escrever códigos.

Finalmente, relembre que a linguagem *C* não foi desenvolvida para fins didáticos, ela foi criada para se construir um sistema operacional, como já foi dito neste texto.

Portanto, em *C*, o programador deve deixar de lado a abstração fornecida na linguagem *Pascal* e explicitar o endereço das variáveis a serem lidas, ou qualquer outra que ele queira modificar dentro das suas funções.

Na seção 13 discutiremos mais sobre passagem de parâmetros em *C*. Por enquanto temos o suficiente para continuarmos a entender o básico da nova linguagem através da comparação de duas linguagens diferentes.

9 Expressões e operadores

A linguagem *C*, assim como praticamente todas as linguagens de programação, também possui operadores aritméticos e lógicos. Porém, em *C* não há o conceito de expressão booleana no mesmo sentido que existe em *Pascal*.

A propósito, a linguagem *C* não possui nativamente o tipo boolean[6]. Logo abaixo explicaremos melhor este ponto.

Os principais operadores são os mesmos do *Pascal*: os básicos são adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`) e resto de divisão inteira (`%`).

Algumas diferenças existem, contudo. Uma delas é que não existe um operador específico para a divisão inteira, como havia o `div` em *Pascal*. O tipo do resultado de uma divisão é definido pelos operandos. Vejamos em uma tabela simples:

operando	operando	resultado
int	int	int
int	float	float
float	int	float
float	float	float

Assim podemos afirmar que `4 / 2` resulta em 2 (inteiro), mas `4 / 2.` resulta em 2.0000 (*float*), pois na última expressão existe um ponto (`.`) após o número 2.

Alguns dos operadores relacionais básicos também são os mesmos, como nos casos das desigualdades: `<, <=, >, >=`. Uma diferença importante é que o operador de igualdade é um duplo símbolo de igual (`==`). enquanto que o operador de diferença é grafado assim: `!=`, isto é, um ponto de exclamação antes do símbolo de igualdade. Reforçando: um único símbolo de igualdade (um `=` isolado) é um operador de atribuição.

A linguagem também dispões de operadores lógicos que fazem o mesmo papel que, em *Pascal*, fazem o `AND, OR, NOT`. Eles são grafados diferentemente, no caso assim, respectivamente: `&&, ||, !`.

A ordem de precedência entre os operadores é a mesma da linguagem *Pascal*.

Uma importante diferença semântica entre as duas linguagens é referente ao que já foi explicado na seção 5, ou seja, os operadores em *C* retornam valores, ainda que eles não sejam utilizados. Isto fará diferença para um aprendiz quando ele for escrever códigos com desvios ou repetições, por exemplo. Mostraremos isso na seção 10.

A tabela a seguir mostra o valor resultando para os diferentes operadores, lembrando que em *Pascal* o retorno é booleano (`true` ou `false`):

operação	resultado
`a < b`	1 caso a seja menor do que b, 0 caso contrário
`a <= b`	1 caso a seja menor ou igual a b, 0 caso contrário
`a > b`	1 caso a seja maior do que b, 0 caso contrário
`a >= b`	1 caso a seja maior ou igual a b, 0 caso contrário
`a == b`	1 caso a seja igual a b, 0 caso contrário
`a != b`	1 caso a seja diferente de b, 0 caso contrário

[6] Já comentamos sobre isso na seção 4

No caso dos operadores lógicos, a tabela é a seguinte:

operação	resultado
`a && b`	1 caso ambos a e b tenham valores diferentes de 0, 0 caso contrário
`a || b`	1 caso um dentre a e b tenha valor diferente de 0, 0 caso contrário
`! a`	1 caso a seja 0, 0 caso a tenha valor diferente de 0

Por exemplo, considere a seguinte expressão matemática: $0 \leq x \leq 10$; Para representá-la em *C* é assim:

```
0 <= x && x <= 10
```

Em *Pascal* seria necessário o uso de alguns parênteses para cuidar dos booleanos. Ficaria assim:

```
(0 <= x) AND (x <= 10)
```

O compilador analisa da esquerda para a direita, desta forma `0 <= x` resulta em 0 ou 1, em seguida `x <= 10` também resulta em 0 ou 1. Finalmente o operador `&&` decide e produz também ou 0 ou 1.

A título de exemplo, podemos querer representar a negação da expressão acima, isto é: $\neg(0 \leq x \leq 10)$, que ficaria assim:

```
! (0 <= x && x <= 10)
```

Pelas leis de De Morgan esta expressão é equivalente a $(x < 0) \vee (x > 10)$, a qual pode ser representada em *C* assim:

```
x < 0 || x > 10
```

Agora sim uma breve explicação da diferença entre booleanos nas duas linguagens. Em *Pascal*, o tipo boolean é nativo e resulta sempre em *true* ou *false*, enquanto que em *C*, *false* é 0 e qualquer coisa diferente de zero é considerado *true*, inclusive valores negativos.

Então, considerando que as variáveis `OK` e `NotOK` são do tipo *boolean*, uma expressão do tipo `OK:= 2 > 1` resulta em *true* enquanto que `NotOK:= 1 > 2` resulta em *false*. Consequentemente, a expressão `(2 > 1) AND (1 > 2)` resulta em *false*.

Pois bem, em *C*, o resultado de uma expressão deste tipo é zero ou um. Desta forma, `OK = 2 > 1` resulta em 1, enquanto que `NotOK:= 1 > 2` resulta em 0. Naturalmente, a expressão `2 > 1 && 1 > 2` resulta em 0.

Mas lembrem-se que as expressões em *C* retornam valores, por isso, nesta linguagem, é possível somar os valores de retorno. Considere o código abaixo.

```c
#include <stdio.h>

int main(){
    int cont;

    cont = (2 > 1) + (4 > 2);
    printf ("%d\n", cont);
}
```

Uma vez compilado e executado, vejam que o programa imprime como saída o valor 2.

```
$ gcc teste.c
$ ./a.out
2
```

Pode parecer estranho, mas basta lembrar o que escrevemos na seção 5: as expressões em *C* retornam valores. Por isso, `(2 > 1)` retorna 1 e `(4 > 2)` retorna 1 também e consequentemente podemos somar estes valores, e como `1 + 1` é 2, a saída é 2. Isto não pode ser feito em *Pascal*.

Finalmente vale esclarecer que as duas linguagens fazem o chamado "curto-circuito" em expressões que usam os operadores lógicos `AND` (`&&`) e `OR` (`||`). Para exemplificar, vejamos as duas situações abaixo.

```
(4 > 2) OR (1 > 2) (em Pascal)
4 > 2 || 1 > 2 (em C)
```

Neste caso, como 4 é maior que 2, a primeira expressão resulta em verdadeiro e o valor de toda a expressão já está definido, pois verdadeiro OR qualquer outra coisa resulta em verdadeiro. Por isso a avaliação `1 > 2` não é feita.

Vejamos o caso do outro operador.

```
(1 > 2) AND (4 > 2) (em Pascal)
1 > 2 && 4 > 2 (em C)
```

Como 1 não é maior do que 2, a primeira expressão resulta em falso e o valor de toda a expressão já está definido, pois falso AND qualquer outra coisa resulta em falso. Por isso a avaliação `4 > 2` não é feita.

10 Comando condicional

Toda linguagem de programação possui um comando de desvio condicional, no caso de *C* ele também é o *if* ou na versão *if/else*.

A sintaxe é simples, mas diferente de *Pascal*:

```
1 if (expressao)
2     comando1;
3 else
4     comando2;
```

Sintaticamente ainda, os parênteses são obrigatórios em *C*, bem como o ponto-e-vírgula antes do *else*, exatamente ao contrário do *Pascal*.

A principal diferença semântica é o que foi mencionado na seção anterior, a linguagem *C* não possui nativamente o tipo booleano. O que é testado no *if* é se o valor sendo avaliado é zero ou se é diferente de zero.

De fato, `if (expressao)` avalia se `expressao` é 0, neste caso é o que chamaríamos de `false` em *Pascal*. Se `expressao` for qualquer valor diferente de zero (positivo ou negativo), então chamaríamos em *Pascal* de `true`.

Por exemplo, no código abaixo, será impresso sempre `oi` e nunca `tchau`.

```
1 if (2023)
2     printf ("oi\n");
3 else
4     printf ("tchau\n");
```

Pode-se trocar 2023 por qualquer valor positivo ou negativo, desde que não seja 0, a saída será sempre `oi`.

Por este motivo, é comum encontrar códigos em *C* que contém linhas como estas aqui:

```
1 int x;
2
3 /* algumas linhas de codigo */
4
5 if (x)
6     printf ("oi\n");
7 else
8     printf ("tchau\n");
```

A única maneira de entrar na parte *else* é x conter o valor 0.

Embora seja comum, pode-se argumentar se isto é ou não é uma boa prática de programação em geral e se não seria o caso de explicitar o teste sendo feito, tal como na figura abaixo.

```c
int x;

/* algumas linhas de codigo */

if (x == 0)
    printf ("oi\n");
else
    printf ("tchau\n");
```

De qualquer maneira, no caso particular da linguagem *C*, é amplamente aceito usar `if (x)`.

Finalmente, da mesma maneira que em *Pascal*, caso se queira executar mais de um comando, seja na parte então, seja na parte senão, basta abrir um bloco com um par de chaves. Também pode-se aninhar desvios condicionais.

Vejamos um exemplo mais comum deste comando.

```c
int x;

scanf ("%d", &x);
if (x % 2 == 0)
    printf ("%d eh par\n", x);
else
    printf ("%d eh impar\n", x);
```

Algo um pouco mais complexo poderia ser como abaixo.

```c
int x;

scanf ("%d", &x);
if (0 <= x && x <= 10)
    printf ("%d esta no intervalo [0..10]\n", x);
else
    printf ("%d esta fora do intervalo [0..10]\n", x);
```

11 Repetições

A linguagem *C* oferece algumas alternativas para repetição de código. Em todas elas não há necessidade de abrir um bloco com as chaves no caso de haver um único comando. Caso contrário, se houver mais de um comando a ser repetido, basta abrir o bloco.

As construções mais parecidas com *Pascal* são estas abaixo:

```
1 /* comando while */
2 while (expressao)
3     comando;
4
5 /* comando do/while */
6 do
7     comando;
8 while (expressao)
```

Assim como no caso do *if/else* explicado na seção anterior estas formas testam o valor inteiro da expressão. Igualmente, o valor 0 significa um valor falso e qualquer outro valor, positivo ou negativo, significa um valor verdadeiro.

O interessante de se avaliar um número inteiro e não uma expressão booleana é que podemos avaliar qualquer função que retorne um inteiro. Vejam este exemplo abaixo no qual o laço vai terminar ou quando `x` for maior ou igual a `y`, ou se o usuário digitar um valor que não corresponde ao tipo esperado, pois a função *scanf* retorna o número de itens corretamente lidos, compatíveis com os tipos e corretamente atribuídos.

```
1 int x, y;
2
3 while (scanf("%d %d", &x, &y) == 2 && x < y)
4     printf ("%d %d\n", x, y);
```

Não é difícil perceber a similaridade do *while* entre as duas linguagens. No caso do comando *do/while* é um pouco diferente, pois o *repeat/until* do *Pascal* termina quando a condição for verdadeira. Já em *C*, o comando é repetido enquanto a condição é verdadeira. Isto produz expressões negadas uma com relação à outra.

Por outro lado, o comando *for* em *C* é semanticamente bastante diferente do mesmo comando em *Pascal*. Na linguagem *Pascal* o *for* é rigidamente controlado pelo compilador. Vejamos um trecho de código em *Pascal*:

```
1 /* comando for em pascal */
2 for i:= 1 to n do
3     comando;
```

O comando interno do *for* será executado *sempre n* vezes, independentemente do programador tentar alterar os valores das variáveis `i` e `n` no comando interno.

O compilador *Pascal*, antes do laço iniciar, armazena os valores das variáveis e não permite que o programador altere o número de vezes que o comando será executado.

Ainda, em *Pascal*, o *for* só permite incrementar de 1 em 1 ou decrementar também de 1 em 1.

Vejamos dois exemplos para entendermos o comportamento do *for* em *Pascal*.

Neste primeiro exemplo, tentamos incrementar o `i` no corpo do *for* (linha 9) para tentarmos enganar o comando e ver se ele faz o incremento de `i` de 2 em 2. Vejam o resultado logo abaixo, dá erro de compilação.

```
1  program for_em_pascal;
2  var i, n: longint;
3
4  begin
5      n:= 5;
6      for i:= 1 to n do
7      begin
8          writeln (i);
9          i:= i + 1;
10     end;
11 end.
```

Aqui o erro de compilação, que proibe alterar o valor de `i`:

```
$ fpc for_em_pascal.pas
Free Pascal Compiler version 3.2.0+dfsg-12 [2021/01/25] for x86_64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling for_em_pascal.pas
for_em_pascal.pas(9,10) Error: Illegal assignment to for-loop variable "i"
for_em_pascal.pas(12) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

No próximo exemplo tentamos novamente enganar o compilador alterando na linha 8 o valor da variável `n`. Desta vez compila, mas o *for* continua a ser executado 5 vezes. Apesar do valor da variável `n` ter sido alterado, não muda o número de iterações do *for*.

```
program for_em_pascal;
var i, n: longint;

begin
    n:= 5;
    for i:= 1 to n do
    begin
        n:= -1;
        writeln (i,' ',n);
    end;
end.
```

Aqui o resultado da compilação e da execução do programa, onde se precebe que `n` teve seu valor alterado mas isso não impediu que o laço tenha rodado cinco vezes.

```
$ fpc for_em_pascal.pas
Free Pascal Compiler version 3.2.0+dfsg-12 [2021/01/25] for x86_64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling for_em_pascal.pas
Linking for_em_pascal
11 lines compiled, 0.1 sec
$ ./for_em_pascal
1 -1
2 -1
3 -1
4 -1
5 -1
```

O compilador *Pascal* cuida do processo iterativo separadamente, provavelmente em alguma variável interna.

Desta forma, está comprovado que *Pascal* é uma linguagem rígida com relação ao comando *for*.

Em *C*, o comando *for* é quase uma maneira simplificada, ou econômica, de escrever um *while*. A tradução do programa *Pascal* acima para *C* é assim:

```
/* comando for em C */
for (i = 1; i <= n; i++)
    comando;
```

A sintaxe geral é assim:

```
/* comando for em C */
for (inicializacao; teste de termino; comando_final)
    comando;
```

Em suma, o *for* tem três partes separadas por pontos-e-virgulas: um ou mais comandos de inicialização, um teste para verificação de término do laço e finalmente um ou mais comandos que são executados ao final do bloco do laço. Nenhuma destas partes é obrigatória!

A forma geral do *for* acima pode ser feita com um *while* da seguinte forma:

```
1 inicializacao;
2 while (teste de termino){
3     comando;
4     comando_final;
5 }
```

O uso padrão do comando no entanto é muito parecido com o que se faz em *Pascal*, isto é, algo como imprimir os números de 1 a 10 pode ser feito assim:

```
1 for (i = 1; i <= 10; i++)
2     printf ("%d ", i);
3 printf ("\n");
```

Mas em *C* o for pode ser utilizado de algumas formas alternativas e completamente diferentes do que *Pascal* permite. Neste primeiro exemplo o valor inicial de i foi definido antes do *for* e o comando de inicialização do *for* ficou vazio. Observem o ponto-e-vírgula imediatamente seguindo o abre parênteses na linha 2.

```
1 i = 1;
2 for (; i <= 10; i++)
3     printf ("%d ", i);
4 printf ("\n");
```

Neste outro exemplo, mostramos que é possível ter coisas mais complexas nas três partes do comando, como por exemplo, para alterar o valor do incremento de duas variáveis de uma só vez. Basta fazer uso de uma vírgula, que significa executar um comando e depois o outro. Assim, inicialmente, tanto `i` quanto `j` têm seus valores iniciais definidos respectivamente para 1 e 0. O teste permaneceu o mesmo, mas o comando final permite incrementar `i` de 1 em 1 e também decrementar `j` de 2 em 2.

```
1 for (i = 1, j = 0; i <= 10; i++, j-=2)
2     printf ("%d %d\n", i, j);
```

Outra forma bizarra é esta abaixo, na qual nenhuma das três partes aparece e resulta em um laço infinito que vai imprimir "Hello world" eternamente. Vejamos:

```
1 for (;;)
2     printf ("Hello world\n");
```

Se estas coisas fazem sentido ou não depende do que o programador deseja fazer. Como já falamos, em *C*, o compilador permite praticamente total liberdade para o programador fazer o que bem entender. Se ele entende de fato ou não o que está fazendo não é problema do compilador. Como já dito, isto é filosoficamente totalmente diferente de *Pascal*, o que resulta no fato de que o programador deve ter mais responsabilidade.

Por exemplo, o programador pode conhecer a existência do *break*, que faz com que o laço termine. Neste caso, o programa acima poderia fazer sentido se fosse escrito assim:

```
1  for (;;){
2      printf ("Hello world\n", i);
3      if (algum_teste ())
4          break;
5  }
6  outro_comando;
```

É feio, mas funciona corretamente. Se o teste do *if* na linha 3 for verdadeiro então o *for* termina e o `outro_comando` será executado.

Este exemplo mostra a grande diferença semântica do *for* do *Pascal* com relação ao *C*, pois é evidente a possibilidade de terminar o laço a qualquer momento em *C*.

Outra possibilidade é usar um *continue*, que faz com que os comandos subsequentes sejam ignorados e o fluxo do programa volte para o início do laço. Por exemplo assim:

```
1  for (i = 0; i < n; i++){
2      if (algum_teste ())
3          continue;
4      printf ("Hello world\n", i);
5  }
```

Neste caso, se o teste do *if* na linha 2 for verdadeiro e o *continue* for executado, então o *printf* não será executado e o fluxo volta para a avaliação `i < n`. Observem que o `i++` será executado. Desta forma o programa abaixo vai imprimir na saída `1 3 4`.

```
1  #include <stdio.h>
2
3  int main(){
4      int i;
5
6      for (i=1; i<5; i++){
7          if (i == 2)
8              continue;
9          printf ("%d ", i);
10     }
11     printf ("\n");
12 }
```

Uma observação final sobre os comandos *break* e *continue*: ambos são formas "estruturadas" de um *goto*, que sabidamente é proibido em programação estruturada. Contudo, estas duas formas são aceitas, pois são consideradas "seguras". A bem da verdade, a versão atual do compilador (*free*) *Pascal* aceita estes dois comandos também, embora não estivessem no *Pascal* padrão original.

Sobre este assunto seguem três referências que podem interessar:[7]

- Structured programming with GOTO statements
  `https://dl.acm.org/doi/pdf/10.1145/356635.356640`;
- GOTO considered harmful
  `https://dl.acm.org/doi/pdf/10.1145/362929.362947`;
- GOTO considered harmful considered harmful
  `https://dl.acm.org/doi/pdf/10.1145/214748.315722`.

[7]Gentilmente apontadas pelo prof. Bruno Müller, do DInf/UFPR.

12 Switch/case

A linguagem *C* tem o equivalente do *case* do *Pascal*, é o *switch/case*. Como de costume, existem diferenças semânticas importantes.

Vamos usar como exemplo um menu simples, inicialmente feito em *Pascal*, ilustrado na figura 2.

```
1  program menu;
2  var x, y, opcao: longint;
3  begin
4      writeln ('Digite dois inteiros');
5      read (x, y);
6      writeln ('Escolha:');
7      writeln ('  1 - soma');
8      writeln ('  2 - subtracao');
9      writeln ('  3 - multiplicacao');
10     writeln ('  4 - divisao');
11     read (opcao);
12     case opcao of
13         1: writeln (x + y);
14         2: writeln (x - y);
15         3: writeln (x * y);
16         4: writeln (x div y);
17     else
18         writeln ('opcao invalida');
19     end;
20 end.
```

Figura 2: Um menu em *Pascal*.

A implementação do *case* em *Pascal* é somente uma maneira econômica de se escrever uma sequência de *if/else*'s aninhados. O importante a destacar é que quando se cai em uma regra nenhuma outra é executada.

O equivalente deste programa em *C* é mostrado na figura 3.

```c
#include <stdio.h>

int main(){
    int x, y, opcao;

    printf ("Digite dois inteiros: ");
    scanf ("%d %d", &x, &y);
    printf ("Escolha:\n");
    printf ("  1 - soma\n");
    printf ("  2 - subtracao\n");
    printf ("  3 - multiplicacao\n");
    printf ("  4 - divisao\n");
    scanf ("%d", &opcao);
    switch (opcao){
        case 1: printf ("%d\n", x + y); break;
        case 2: printf ("%d\n", x - y); break;
        case 3: printf ("%d\n", x * y); break;
        case 4: printf ("%d\n", x / y); break;
        default: printf ("opcao invalida\n");
    }

    return 0;
}
```

Figura 3: Um menu em *C*.

Contrariamente ao comportamento do *case* em *Pascal*, na linguagem *C* o fluxo é desviado a partir da opção escolhida para o primeiro ponto em que a condição *case* é válida, mas segue a partir deste ponto até o final do comando. Por isso é necessário, caso se queira, usar o *break*, caso contrário todos os comandos subsequentes serão executados. Dependendo do programa, pode ser útil não usar o *break*.

O próximo exemplo da figura 5 mostra que, em determinadas situação nas quais o uso do *case* em *Pascal* seria natural, pode não ser o caso em *C*. Nestes casos talvez seja melhor manter uma sequência de *if/else*'s aninhados ou usar outras estruturas tais como os vetores, que serão apresentados na seção 19.

O exemplo é simples, mas mostra que se existem muitos casos para serem analisados, a lista pode crescer muito e tornar o código difícil de escrever e de se dar manutenção, além de prejudicar muito a legibilidade.

```c
#include <stdio.h>

int main(){
    int x;

    scanf ("%d", &x);
    switch (x){
        case 1: case 2: case 3: case 4: case 5:
            printf ("[1..5]\n");
            break;
        case 6: case 7: case 8: case 9: case 10:
            printf ("[6..10]\n");
            break;
        default: printf ("maior do que 10\n");
    }

    return 0;
}
```

Figura 4: Um uso ruim do switch/case em *C*.

Em *Pascal* é bem fácil escrever este código usando o `case`.

```pascal
program exemplo_case;
var x: integer;

begin
    read (x);
    case x of
        1..5: writeln ('[1..5]');
        6..10: writeln ('[6..10]');
    end;
end.
```

Figura 5: O mesmo programa em *Pascal*.

13 Funções, parte 1

O conceito de funções em *C* é o mesmo de *Pascal*, mas como sempre existem diferenças semânticas significativas. Em primeiro lugar, em *C* não existem procedures, somente funções.

A sintaxe de uma função é a seguinte:

```
1  tipo_retorno identificador (lista de parametros)
2  {
3      corpo da funcao
4  }
```

A função cujo nome é `identificador` recebe uma lista de parâmetros e realiza as instruções contidas no corpo dela e retorna um dado de um certo tipo `tipo_retorno`.

Se o *tipo_retorno* for `void` significa que não haverá valor de retorno, e podemos pensar que seria o equivalente a uma procedure em *Pascal*. Neste caso, existem duas maneiras de retornar o fluxo de execução para quem chamou a função: ou quando a função termina ou usando-se um comando *return* em qualquer ponto do código e a função terminará neste ponto.

Se o *tipo_retorno* for qualquer outro tipo então o valor de retorno é definido em uma expressão *return expressão*, onde *expressão* deve ter o mesmo tipo *tipo_retorno*. Isto é, o *return* é obrigatório neste caso.

A lista de parâmetros pode ser vazia, mas os parênteses são obrigatórios. Mas caso não seja vazia, ela é constituída por uma lista de parâmetros separados por vírgula, cada parâmetro deve ter seu tipo definido. Vejamos um exemplo simples.

```
1  /* retorna o MDC entre dois inteiros pelo metodo de Euclides */
2  /* se um dos parametros for nulo, retorna -1                 */
3  int mdc (int a, int b){
4      int resto;
5
6      if (a == 0 || b == 0)
7          return -1;
8
9      resto = a % b;
10     while (resto != 0){
11         a = b;
12         b = resto;
13         resto = a % b;
14     }
15     return b;
16 }
```

Esta função recebe dois inteiros `a` e `b` e retorna um outro inteiro. A variável `resto` é local à função (ver seção 18). Caso um dos parâmetros seja nulo então a função retorna o valor -1 para quem a chamou e imediatamente termina. Por isso não é necessário o *else* neste *if*. O restante do código calcula o MDC pelo método de Euclides e retorna o valor no *return* da linha 15.

Uma das principais diferenças com relação à *Pascal* é que, em *C*, a função termina quando encontra um *return* enquanto que em *Pascal* ela só termina quando encontra o *end*. Os demais conceitos são similares.

O próximo exemplo mostra um caso no qual o tipo do retorno é *void*. Observem que não há necessidade do *return*.

```
1 void imprime_se_par (int a){
2
3     if (a % 2 == 0)
4         printf ("%d eh par", a);
5 }
```

Agora mostramos um caso cujo retorno é *void* mas a função contém um *return*. Neste caso, a função só imprime alguma coisa se `b` for diferente de zero.

```
1 void imprime_divisao (int a, int b){
2
3     if (b == 0)
4         return;
5     printf ("%d / %d = %d", a, b, a/b);
6 }
```

A outra diferença significativa é que não existe o conceito de passagem de parâmetros por referência como em *Pascal*, em *C* isto deve ser explicitado passando o endereço da variável (também conhecido como um *ponteiro*). Falamos um pouco sobre isso na seção 8 quando apresentamos o *scanf*.

Para melhor explicar o conceito é necessário antes falarmos de ponteiros.

14 Ponteiros

Embora o conceito de ponteiros seja simples, ele não é costumeiramente fácil de ser absorvido por estudantes. Por isso esta seção terá algumas subseções antes de explicarmos o conceito propriamente dito.

14.1 Motivação

Qual é o motivo de ponteiros serem necessários ou úteis para programadores? Algumas respostas estão listadas a seguir.

- Manipulação direta da memória: ponteiros permitem acessar diretamente regiões de memória, o que é útil em operações de baixo nível, como manipulação de buffers, gerenciamento de memória ou interação com hardware;
- Eficiência: passar grandes estruturas de dados para funções (como vetores ou objetos complexos) pode ser ineficiente se feito por cópia. Usar ponteiros para passar referências para esses dados é mais eficiente e evita cópias desnecessárias;
- Alocação de Memória Dinâmica: em linguagens como *C*, mas também *Pascal*, ponteiros são essenciais para a alocação dinâmica de memória, que por sua vez permite que os programas determinem a quantidade de memória necessária em tempo de execução;
- Estruturas de Dados Complexas: ponteiros são fundamentais na implementação de estruturas de dados complexas, como listas ligadas, árvores e grafos, onde elementos precisam referenciar outros elementos. Estas estruturas permitem elaborar algoritmos altamente eficientes para, por exemplo, fazer buscas, inserções e remoções em tempo logarítmico.

14.2 Por quê não estudamos ponteiros em algoritmos 1

Quando estudamos *Pascal* em Algoritmos e Estruturas de Dados 1 não foi necessário explicar o conceito de ponteiros, embora exista nesta linguagem.

Na ocasião, omitimos a existência do tipo *pointer*, que é justamente o conceito que explicaremos nesta seção. Como não foi visto em *Pascal*, explicaremos aqui somente na linguagem *C*.

Não há como evitar explicar ponteiros quando estudamos a linguagem *C*, basicamente porquê não existe passagem de parâmetros por referência. Como este é um curso de algoritmos e também porquê os programas devem ser estruturados modularmente com o uso de funções, é inevitável abordar o assunto antes de continuarmos a estudar funções, o que será feito na seção 15.

14.3 Variáveis e a memória do computador

No capítulo 4 do livro de Algoritmos 1 [CSW20] vimos que os computadores modernos usam a chamada arquitetura Von Neumann, na qual os dados e o programa ficam em

memória. O hardware provê instruções que podem ser utilizadas quando os códigos-fonte das linguagens de programação são compilados e os programas executáveis são gerados.

Alguns conceitos devem ser explicados antes de iniciarmos o estudo sobre ponteiros.

O que é a memória do computador?

A memória do computador pode ser ilustradamente vista como uma matriz de duas colunas, na qual a primeira coluna são os endereços de memória e a segunda coluna são os dados armazenados nestes endereços.

O que são endereços de memória?

Cada região de memória tem um endereço único. Após a compilação, uma variável passa a ser referenciada por este endereço e não pelo seu identificador.

Esquematicamente podemos ver isso como na figura abaixo.

Endereço	Conteúdo
0x7fff00da0000	
0x7fff00da0004	
0x7fff00da0008	
0x7fff00da0012	
0x7fff00db0016	
0x7fff00db0020	
0x7fff00db0024	
⋮	

Na figura, os endereços são mostrados na forma hexadecimal, por isso iniciam com `0x`, e são mostrados de 4 em 4 bytes por questões didáticas neste momento.

O que é uma região de memória?

É uma ou mais posições na memória RAM para armazenar o valor de uma variável.

Quando declaramos uma variável em *Pascal* ou em *C*, por se tratarem de linguagens tipadas, devemos declarar um identificador (o nome da variável) e seu tipo, conforme vimos na seção 4 deste texto.

O que é um identificador?

Cada variável possui um nome, que é o seu identificador. Este nome não é armazenado na memória, mas serve como uma referência simbólica para um endereço específico desta memória. No código fonte dos programas utilizamos o identificador para nos referimos à uma variável. Durante a compilação, esse identificador é mapeado

para um endereço específico na memória.

O que são tipos e seus tamanhos?

O tipo da variável (`int`, `float`, `char`, ...) determina o tamanho da região de memória que é necessário para que o dado possa ser corretamente armazenado bem como os bits naquela região de memória são interpretados.

Por exemplo, no programa abaixo, declaramos duas variáveis do tipo `int` (`a` e `b`) e outras duas do tipo `float` (`x` e `y`). Atribuímos alguns valores para melhor exemplificar.

```
1  int main (){
2      int a, b;
3      float x, y;
4
5      a = 13;
6      b = 5;
7      x = 1.57;
8      y = -9.84;
9
10     return 0;
11 }
```

Vamos assumir neste caso que estes dois tipos ocupam 4 bytes de memória (32 bits), mas o real tamanho de cada tipo depende do hardware, do sistema operacional e do compilador específico.

Essas variáveis são alocadas estaticamente quando o programa executável inicia, e, de alguma forma, elas devem ocupar alguma posição na memória.

Também vamos assumir, para fins de exemplo, que `a`, `b`, `x` e `y` ocuparam as quatro primeiras posições da matriz abaixo. Assumiremos também que o computador tem o controle de qual posição de memória cada variável usa, desta forma iremos considerar que existe uma matriz assim:

Identificador	Endereço
a	0x7fff00da0000
b	0x7fff00da0004
x	0x7fff00da0008
y	0x7fff00da0012

A partir desta matriz, o programador pode abstrair o endereço e usar os identificadores em seus programas, pois é mais confortável para o ser humano usar nomes ao invés de endereços. Mas, importante destacar, os identificadores só existem no código fonte.

Portanto, podemos visualizar a memória como na figura abaixo[8].

[8]Lembrando que os números na coluna de conteúdo são na verdade bits (zeros e uns) e são interpretados pelo compilador a partir dos tipos das variáveis.

Endereço	Conteúdo
0x7fff00da0000	13
0x7fff00da0004	5
0x7fff00da0008	1.57
0x7fff00da0012	-9.84
0x7fff00db0016	
0x7fff00db0020	
0x7fff00db0024	
⋮	

Também podemos ter um pouco mais de conforto visual e imaginar que a matriz agora tem três colunas, unificando as duas matrizes acima. Considere que os espaços não utilizados na coluna de conteúdo contêm lixo de memória. A coluna do identificador não faz parte da grade, para ilustrar que é meramente um conforto visual para o leitor ou leitora.

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	13
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
	0x7fff00db0016	
	0x7fff00db0020	
	0x7fff00db0024	
	⋮	

Para deixar bem claro, variáveis são identificadores que correspondem aos endereços de memória e o que ocorre ao se atribuir valores para as variáveis (ou consultar seus valores), estes valores são armazenados nos respectivos endereços associados aos seus identificadores. Tudo isso é transparente para o programador.

14.4 Variáveis e tipos

Já apresentamos as variáveis e tipos na seção 4, mas vale alguma explicação adicional neste ponto, mesmo que seja um pouco redundante. Isto será feito por questões de completude do texto desta seção.

Nas linguagens tipadas, as variáveis têm tipos. Os tipos servem para o código do programa seja gerado e tem relação com a interpretação dos bytes acessados na forma que o tipo especifica. Eles interessam para as operações ou para o desreferenciamento, que será apresentado em breve.

Basta saber que a memória do computador contém bytes e que os tipos da linguagem são somente um facilitador para o ser humano, evitando que nós humanos tenhamos que ficar lidando com bytes.

É mais confortável para nós que o compilador assuma para ele, através do conceito de tipos de dados, a conversão de bytes em conceitos tais como int's, float's ou

char's. Quando escrevemos `int n` estamos dizendo para o compilador olhar os 4 bytes acessados correspondentes ao identificador `n` e os interpretá-los na representação de inteiros com sinal.

O mesmo ocorre quando escrevemos `float x`. Estamos dizendo para o compilador olhar os 4 bytes acessados pelo identificador `x` e interpretá-los como um número real em notação de ponto flutuante. Quando escrevemos `char c`, o compilador olha apenas 1 byte naquela posição de memória e faz a conversão baseado na tabela *ASCII* ou outra tabela de conversão.

Ao declaramos uma variável como sendo de um certo tipo, o compilador reserva espaços de memória para aquele tipo. Nos computadores recentes com arquitetura de 64 bits, normalmente os tipos `int` e `float` ocupam 4 bytes, enquanto que o tipo `char` ocupa um único byte e um `doble` ocupa 8 bytes, e assim por diante.

O tamanho da memória reservada depende do tipo da variável, a qual depende da arquitetura da sua máquina, do seu sistema operacional e do compilador utilizado. Nesta seção iremos assumir que inteiros e float's usam 4 bytes cada.

14.5 Finalmente os ponteiros

Um ponteiro é uma variável que armazena um endereço de memória. Nas arquiteturas atuais de 64 bits, isto significa normalmente um espaço de 8 bytes.

É sempre necessário especificar o tipo do ponteiro, isto é, o tipo do dado que está no endereço armazenado pelo ponteiro, mesmo que seja do tipo `void`, que é um tipo especial, vazio, ou indefinido.

Para declarar um ponteiro, basta usar o símbolo de asterisco entre o tipo do dado e o identificador da variável ponteiro. Vejamos na figura abaixo as mesmas variáveis `a`, `b`, `x` e `y` vistas acima e outras duas novas variáveis, as quais daremos os identificadores `ptr_int` e `ptr_float`. Observem atentamente o asterisco (`*`) logo antes dos identificadores nas linhas 4 e 5.

```
1  int main (){
2      int a, b;
3      float x, y;
4      int *ptr_int;       /* indica um ponteiro para int   */
5      float *ptr_float;   /* indica um ponteiro para float */
6
7      a = 13;
8      b = 5;
9      x = 1.57;
10     y = -9.84;
11
12     return 0;
13 }
```

As variáveis de nosso interesse foram declaradas nas linhas 4 e 5 do programa acima. A primeira, `ptr_int`, é do tipo *ponteiro para int* e a segunda, `ptr_float` é do tipo *ponteiro para float*. Ambas terão como conteúdo endereços de memória. A primeira, `ptr_int`, conterá um endereço de uma região de memória que por sua vez

contém uma região de memória que acessa um `int`, enquanto que a segunda conterá um endereço de uma região de memória que por sua vez acessa um `float`.

Notem que os ponteiros (variáveis ponteiros) foram declarados mas não tiveram ainda valor atribuído, por isso contêm lixo de memória. Mas é evidente que, por serem variáveis, estão em algum lugar da memória. Vamos considerar a seguinte tabela de identificadores versus endereços:

Identificador	Endereço
a	0x7fff00da0000
b	0x7fff00da0004
c	0x7fff00da0008
d	0x7fff00da0012
ptr_int	0x7fff00da0016
ptr_float	0x7fff00db0024

Novamente, mostraremos as variáveis com seus conteúdos em uma visualização em três colunas, para facilitar a compreensão (observem que as novas variáveis ocupam 8 bytes cada):

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	13
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
ptr_int	0x7fff00db0016	???
ptr_float	0x7fff00db0024	???
	0x7fff00db0032	
	⋮	

Conforme já mencionado, observem que as variáveis ponteiros foram declaradas mas até o momento possuem lixo de memória. Como colocar valores nelas?

14.6 Atribuição de valores para ponteiros

Os valores dos ponteiros devem ser endereços, pois são endereços que os ponteiros armazenam. A linguagem *C* dispõe de um operador específico para capturar o endereço de uma variável, trata-se do operador & (e comercial). Este operador opera sobre uma variável e resulta no endereço dela.

Então podemos fazer um programa como este abaixo, no qual definimos conteúdos para as variáveis ponteiros. A primeira variável, `ptr_int`, é do tipo ponteiro para `int`, portanto, o programador deve atribuir um endereço o qual contenha um `int`. No nosso caso tanto pode ser `a` quanto `b`. Similarmente, `ptr_float` pode receber tanto o endereço de `x` quanto o de `y`, pois esta variável é do tipo ponteiro para `float`.

```
1  int main (){
2      int a, b;
3      float x, y;
4      int *ptr_int;
5      float *ptr_float;
6
7      a = 13;
8      b = 5;
9      x = 1.57;
10     y = -9.84;
11     ptr_int = &a;
12     ptr_int = &x;
13
14     return 0;
15 }
```

Agora vamos observar na tabela de endereços abaixo o que ocorreu após as duas atribuições nas linhas 11 e 12, isto é, `ptr_int` recebeu o endereço de `a` e `ptr_float` recebeu o endereço de `x`.

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	13
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
ptr_int	0x7fff00db0016	0x7fff00da0000
ptr_float	0x7fff00db0024	0x7fff00da0008
	0x7fff00db0032	
	⋮	

Observem atentamente na tabela acima que o conteúdo de `ptr_int` é o endereço de `a` e também que o conteúdo de `ptr_float` é o endereço de `x`.

Como qualquer outra variável, as variáveis ponteiros também têm seus próprios endereços, mas seus conteúdos são também endereços. Ademais, podemos alterar seus valores também, como ocorre com qualquer variável.

Por exemplo, podemos executar comandos tais como estes:

```
1      ptr_int = &b;
2      ptr_float = &y;
```

Da mesma forma como ocorre com qualquer variável, os conteúdos de `ptr_int` e `ptr_float` agora foram alterados. Nossa tabela fica assim agora:

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	13
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
ptr_int	0x7fff00db0016	**0x7fff00da0004**
ptr_float	0x7fff00db0024	**0x7fff00da0012**
	0x7fff00db0032	
	⋮	

Observem atentamente na tabela acima que o conteúdo de `ptr_int` mudou e agora contém o endereço de `b` e também que o conteúdo de `ptr_float` mudou e agora contém o endereço de `y` (destaque em negrito).

É exatamente a mesma coisa que se tivéssemos executado `a = -19`, a tabela ficaria assim (destaque em negrito):

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	**-19**
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
ptr_int	0x7fff00db0016	0x7fff00da0004
ptr_float	0x7fff00db0024	0x7fff00da0012
	0x7fff00db0032	
	⋮	

14.7 Imprimindo valores de ponteiros

Naturalmente, pode-se imprimir os valores de qualquer variável, basta usar a formatação certa no comando `printf`. Um endereço, quando é impresso com a formatação `%p`, mostra o endereço na sua forma hexadecimal. Vejamos um programa exemplo e sua saída:[9]

```
1  #include <stdio.h>
2
3  int main (){
4      int a, b;
5      float x, y;
6      int *ptr_int;
7      float *ptr_float;
8
9      a = 13;
10     b = 5;
11     x = 1.57;
12     y = -9.84;
13     ptr_int = &a;
14     ptr_float = &x;
15
16     printf ("o valor de a eh: %d\n", a);
17     printf ("o endereco de a eh: %p\n", &a);
18     printf ("\n");
19
20     printf ("o valor de x eh: %d\n", x);
21     printf ("o endereco de x eh: %p\n", &x);
22     printf ("\n");
23
24     printf ("o valor de ptr_int eh: %d\n", ptr_int);
25     printf ("o endereco de ptr_int eh: %p\n", &ptr_int);
26     printf ("\n");
27
28     return 0;
29 }
```

A saída deste programa, no nosso exemplo, é a seguinte:

```
o valor de a eh: 13
o endereço de a eh: 0x7fff00da0000

o valor de x eh: 1.57
o endereço de x eh: 0x7fff00da0008

o valor de ptr_int eh: 0x7fff00da0000
o endereço de ptr_int eh: 0x7fff00db0016
```

[9] Os endereços das variáveis podem mudar de uma execução para outra. Neste texto usamos endereços fictícios, meramente ilustrativos.

Observem nas linhas 17, 21 e 25 o uso de `%p`, que significa a formatação para imprimir um endereço e também `&a`, `&x` e `ptr_int` que usa o operador `&` para obter os endereços das respectivas variáveis `a`, `x` e `ptr_int`.

14.8 Desreferenciamento

O título desta seção tem um nome feio, mas é relacionado com a seguinte ideia: ao se atribuir à um ponteiro o endereço de outra variável, faz-se uma referência a esta outra variável. Desreferenciar significa seguir a referencia e acessar o conteúdo daquele segundo endereço.

Parece complicado, mas não é.

Um ponteiro referencia uma região de memória da mesma maneira que uma variável "normal". Lembrem-se que uma variável é somente um identificador para o ser humano abstrair o endereço de uma região de memória, a partir de uma tabela de símbolos.

No programa acima, por exemplo, a variável `a`, a qual tem o endereço `0x7fff00da0000`, tem o conteúdo `13`. É fato conhecido do leitor ou da leitora que para acessar este conteúdo pode-se usar tanto uma atribuição quanto, por exemplo, imprimir este valor, assim:

```
1  #include <stdio.h>
2
3  int main (){
4      int a, b;
5
6      a = 13;
7      b = a;                    /* acesso via atribuicao */
8      printf ("a = %d\n", a); /* acesso via impressao  */
9
10     return 0;
11 }
```

Agora a mesma informação, isto é, o valor `13`, também pode ser acessado a partir da variável `ptr_int`, pois embora seu próprio endereço seja outro, seu conteúdo é o mesmo endereço da variável `a`.

Como usar esta informação para acessar o valor `13` usando `ptr_int`?

Aqui entra o segundo operador de interesse, que é o operador de desreferenciamento, denotado (novamente) por um asterisco (`*`)

Não confundam os dois asteriscos, depende do contexto. Ao se declarar uma variável, ele serve para indicar que é uma variável ponteiro. Aqui no caso, este operador serve para desreferenciar o ponteiro, ou seja, acessar o conteúdo do endereço referenciado.

O programa abaixo deve esclarecer melhor o conceito:

```
1  #include <stdio.h>
2
3  int main (){
4      int a;
5      int *ptr_int; /* declaracao de um ponteiro para int */
6
7      a = 13;
8      ptr_int = &a; /* um ponteiro recebe um endereco */
9
10     printf ("o valor de a eh: %d\n", a);
11     printf ("o endereco de a eh: %p\n", &a);
12     printf ("o conteudo de ptr_int eh: %p\n", ptr_int);
13     printf ("o conteudo de *ptr_int eh: %d\n", *ptr_int);
14
15     return 0;
16 }
```

A saída ilustrativa deste programa é a seguinte:

```
o valor de a eh: 13
o endereco de a eh: 0x7fff00da0000
o conteudo de ptr_int eh: 0x7fff00da0000
o conteudo de *ptr_int eh: 13
```

Na linha 13 usamos o operador de desreferenciamento para acessar o conteúdo do endereço contido em `ptr_int`, que evidentemente é `13`, pois ele contém o mesmo endereço da variável `a`.

A diferença de contexto mencionada acima é que quando o símbolo `*` aparece na declaração de um tipo, significa que o tipo é um ponteiro para algum outro tipo, no caso acima, um `int`. Quando o símbolo `*` aparece na linha 13 como argumento do *printf*, ele significa desreferenciar o ponteiro, isto é, acessar o conteúdo daquele endereço.

Vejamos um outro código para mostrar que os conteúdos agora podem ser alterados não apenas por uma variável tal como vocês conheciam até agora, mas também usando os ponteiros.

```
1  #include <stdio.h>
2
3  int main (){
4      int a;
5      int *ptr_int;
6
7      a = 13;
8      ptr_int = &a;
9
10     printf ("o valor de a eh: %d\n", a);
11     printf ("o endereco de a eh: %p\n", &a);
12     printf ("o conteudo de ptr_int eh: %p\n", ptr_int);
13     printf ("o conteudo de *ptr_int eh: %d\n", *ptr_int);
14
15     *ptr_int = 0; /* preste atencao aqui ! */
16     printf ("o valor de a eh: %d\n", a);
17
18     return 0;
19 }
```

O que ocorreu na linha 15 foi uma atribuição do valor zero na posição de memória indicada por `*ptr_int`, ou seja, pelo conteúdo de `ptr_int`, que é exatamente o mesmo endereço de memória da variável `a`. A consequência é que o valor de `a` foi alterado para 0, naturalmente, porque `a` acessa exatamente o mesmo endereço. Vejamos a saída do programa para comprovarmos isto.

```
o valor de a eh: 13
o endereco de a eh: 0x7fff00da0000
o conteudo de ptr_int eh: 0x7fff00da0000
o conteudo de *ptr_int eh: 13
o valor de a eh: 0
```

Copiamos nossa ilustração da memória para facilitar a visualização:

Identificador	Endereço	Conteúdo
a	0x7fff00da0000	13
b	0x7fff00da0004	5
x	0x7fff00da0008	1.57
y	0x7fff00da0012	-9.84
ptr_int	0x7fff00db0016	0x7fff00da0004
ptr_float	0x7fff00db0024	0x7fff00da0012

14.9 Observações finais

O conceito de ponteiros nos será útil já na próxima seção, para podermos explicar como passar endereços nos parâmetros. Futuramente servirá também para lidarmos com alocação dinâmica, assunto que, por enquanto, não está neste texto.

15 Funções, parte 2

Agora que já temos a noção de ponteiros podemos explicar a passagem de parâmetros por endereços. Conforme veremos, os ponteiros serão fundamentais, pois *na linguagem C só existe passagem de parâmetros por cópia* e não tem o conceito de passagem de parâmetros por referência como em *Pascal*.

O leitor ou a leitora pode rever estes conceitos no livro de Algoritmos e Estruturas de Dados I ([CSW20]).

O problema clássico é a troca dos valores de duas variáveis, no qual se quer usar uma procedure para realizar a operação. Em uma primeira versão, usaremos incorretamente a passagem de parâmetros por cópia, conforme o programa abaixo.

```
1  program parametros_por_copia_errado;
2
3  procedure troca (a, b: integer);
4  var temp: integer;
5  begin
6      temp:= a;
7      a:= b;
8      b:= temp;
9  end;
10
11 var x, y: integer;
12 begin
13     x:= 2;
14     y:= 9;
15     troca (x, y);
16     writeln (x,' ', y);
17 end.
```

O importante a destacar é que os parâmetros `x` e `y` usados na declaração da procedure `troca` foram passados por cópia, o que sintaticamente é indicado pela ausência da palavra `var` precedendo os identificadores `a` e `b` dos parâmetros.

Podemos ver isso na memória da mesma forma como fizemos na seção anterior. Na figura abaixo, os endereços são meramente ilustrativos.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	2
y	0x7fff00da0004	9
	0x7fff00da0008	
	0x7fff00da0012	
	0x7fff00db0016	
	0x7fff00db0020	

No momento da execução do programa principal, a memória está como na figura acima. Ao se ativar a função *troca*, os parâmetros `a`, `b` e a variável local `temp` são alocadas na memória (na pilha/stack, ver seção 17).

O parâmetro `a` recebe uma cópia do valor de `x` e o parâmetro `b` recebe uma cópia do valor de `y`. `temp` ainda tem lixo de memória.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	2
y	0x7fff00da0004	9
a	0x7fff00da0008	2
b	0x7fff00da0012	9
temp	0x7fff00db0016	???
	0x7fff00db0020	

Durante a execução do programa, os valores de `a` e `b` são trocados, resultando no estado de memória abaixo (destaques em negrito).

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	2
y	0x7fff00da0004	9
a	0x7fff00da0008	**9**
b	0x7fff00da0012	**2**
temp	0x7fff00db0016	**2**
	0x7fff00db0020	

Ao término da função, os parâmetros e a variável local deixam de existir e o fluxo de execução volta para o programa principal, conforme a ilustração abaixo, onde se percebe claramente que os valores de `x` e `y` ficaram inalterados.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	2
y	0x7fff00da0004	9
	0x7fff00da0008	
	0x7fff00da0012	
	0x7fff00db0016	
	0x7fff00db0020	

A solução para este problema, em *Pascal*, é passar os parâmetros por referência, usando-se a palavra `var` antes dos parâmetros, conforme segue (ver detalhes em [CSW20]).

```
1  program parametros_por_referencia_certo;
2
3  procedure troca (var a, b: integer);
4  var temp: integer;
5  begin
6      temp:= a;
7      a:= b;
8      b:= temp;
9  end;
10
11 var x, y: integer;
12 begin
13     x:= 2;
14     y:= 9;
15     troca (x, y);
16     writeln (x,' ', y);
17 end.
```

Em *C* não há passagem de parâmetros por referência, então, como resolver o problema?

O programa abaixo é a versão incorreta do uso da função *troca*, pois os parâmetros serão passados por cópia e a mesma situação explicada acima ocorrerá.

```
1  #include <stdio.h>
2
3  void troca (int a, int b){
4      int temp;
5
6      temp = a;
7      a = b;
8      b = temp;
9  }
10
11 int main (){
12     int x, y;
13
14     x = 2;
15     y = 9;
16     troca (x, y);
17     printf ("%d %d\n", x, y);
18 }
```

A única maneira de fazer um programa correto em *C* é explicitar o que *Pascal* esconde, que é o fato de que, na realidade, o que é passado é uma referência (um ponteiro) ao endereço das variáveis envolvidas.

Agora que já temos o conhecimento da existência de ponteiros, e também dos operadores `&` e `*`, conforme vimos na seção anterior, podemos fazer o programa da maneira correta, conforme o programa abaixo.

```
1  #include <stdio.h>
2
3  void troca (int *a, int *b){
4      int temp;
5
6      temp = *a;
7      *a = *b;
8      *b = temp;
9  }
10
11 int main (){
12     int x, y;
13
14     x = 2;
15     y = 9;
16     troca (&x, &y);
17     printf ("%d %d\n", x, y);
18 }
```

Observem na linha 16, a chamada à função `troca` usou o operador `&` para passar *os endereços* de `x` e `y`. Por outro lado, na linha 3, os parâmetros `*a` e `*b` receberam respectivamente cópias *dos endereços* da variáveis `x` e `y` da função `main`. Isto é, receberam cópias dos valores, que são endereços, e portanto continuam acessando os locais corretos na memória.

A figura abaixo ilustra novamente o estado da memória quando a função *main* inicia.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	2
y	0x7fff00da0004	9
	0x7fff00da0008	
	0x7fff00da0012	
	0x7fff00db0016	
	0x7fff00db0020	

No momento da ativação da função *troca*, da mesma forma como vimos acima, os parâmetros e a variável local da função são alocados na pilha/stack.

Identificador	Endereço	Conteúdo
x	**0x7fff00da0000**	2
y	**0x7fff00da0004**	9
a	0x7fff00da0008	**0x7fff00da0000**
b	0x7fff00da0016	**0x7fff00da0004**
temp	0x7fff00db0024	???

Notem no destaque, em negrito, que desta vez os parêmetros `*a` e `*b` receberam cópias dos endereços das respectivas variáveis `x` e `y` da função *main*, pois estes parâmetros são *ponteiros para int* e por isso recebem endereços.

Sabemos que a lista de parâmetros define os tipos dos parâmetros, portanto quando escrevemos `int *a` e `int *b` na linha 3, isto indica que `a` e `b` são ponteiros.

Por outro lado, `*a` e `*b`, no corpo da função, nas linhas 6, 7 e 8, significa que estamos usando o operador de desreferenciamento para acessar os conteúdos dos endereços, que são os mesmos das variáveis `x` e `y` da função *main*.

A figura abaixo mostra, com destaques em negrito, que as trocas ocorreram efetivamente nos conteúdos dos endereços de memória armazenados nos ponteiros, isto é, tem como efeito que a troca ocorreu nos endereços das variáveis `x` e `y` da função *main* e por este motivo essas variáveis tiveram seus valores alterados.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	**2**
y	0x7fff00da0004	**9**
a	0x7fff00da0008	0x7fff00da0000
b	0x7fff00da0012	0x7fff00da0004
temp	0x7fff00db0016	2
	0x7fff00db0020	

Observem o que ocorreu:

1. `temp = *a`: `temp` recebeu o conteúdo de `*a`, isto é, do valor 2, pois `a = 0x7fff00da0000` e consequentemente `*a = 2`;

2. `*a = *b`: `*a` recebeu o valor 9, pois `b = 0x7fff00da0004` e consequentemente `*b = 9`, logo, a atribuição foi feita em `*a`, que modifica o valor do endereço `0x7fff00da0000`, que é o mesmo endereço da variável `x` da função *main*;

3. `*b = temp`: `*b` recebeu o valor 2, claramente pois `temp` tem este valor. Notem que `b = 0x7fff00da0004` e consequentemente `*b` recebeu o valor 2 pois a atribuição foi feita em `*b`, que modifica o valor do endereço `0x7fff00da0004`, que é o mesmo endereço da variável `y` da função *main*.

Quando a função termina, o fluxo de execução volta para a função *main* e os parâmetros e a variável local são liberados da memória, resultando no estado de memória da figura abaixo, que mostra claramente que a função *troca* funcionou conforme desejado.

Identificador	Endereço	Conteúdo
x	0x7fff00da0000	**2**
y	0x7fff00da0004	**9**
	0x7fff00da0008	
	0x7fff00da0012	
	0x7fff00db0016	
	0x7fff00db0020	

É por este motivo que a função *scanf* exige um `&` precedendo os identificadores das variáveis que se deseja ler. Em *C*, a única maneira de se ler um valor, ou de maneira

geral, de se *modificar* o valor de uma variável em uma função é passar um ponteiro para ela.

Uma última observação, o fato de que a linguagem *C* só passa parâmetros por cópia vai impactar no caso de, por exemplo, passar vetores nos parâmetros. Vejam a seção 19 para mais detalhes sobre isso.

16 Considerações sobre o retorno de funções

Um aspecto sobre o retorno de valores de função em *Pascal* e em *C* é tão importante que mereceu uma seção independente neste texto.[10]

Em *Pascal* o retono de uma função é de uso obrigatório e em *C* não é!

De fato, em *Pascal*, ao se definir uma função qualquer, quem chamou a função deve obrigatoriamente utilizar este valor. Vejamos um exemplo bem simples no qual o retorno da função `soma_um` é atribuído para a variável `n` na linha 11.

```
1  program exemplo_retorno;
2  var n: integer;
3
4  function soma_um (n: integer): integer;
5  begin
6      soma_um:= n + 1;
7  end;
8
9  begin
10     read (n);
11     n:= soma_um (n);
12     writeln (n);
13 end;
```

O fato de ser obrigatório usar o valor de retorno em *Pascal* faz com que uma função só possa ocorrer em uma expressão aritmética ou booleana, em qualquer outro caso o compilador acusa erro e aborta a compilação.

Isso caracteriza a diferença semântica entre uma função e um procedimento em *Pascal*. As funções fazem parte de expressões e portanto só podem ocorrer no lado direito de uma atribuição ou como parte de uma expressão contida em um comando *write*. Os procedimentos, por sua vez, como não têm valor de retorno, tem um papel semântico de um novo comando.

Por outro lado, na linguagem *C*, o uso do valor de retorno por quem chamou a função não é obrigatório e por este motivo uma função em *C* pode tanto estar no lado direito de uma atribuição ou em um comando de impressão, por exemplo, mas também podem ocorrer como um novo comando. Vamos ver um exemplo e depois analisaremos as consequências.

[10] Esta seção nasceu em uma conversa muito interessante com o professor Fabiano Silva do DInf/UFPR.

```
1  #include <stdio.h>
2
3  int soma_um (int n){
4      return n + 1;
5  }
6
7  int main(){
8      int n;
9
10     scanf ("%d", &n);
11     soma_um (n);        /* aceito em C, nao seria em Pascal */
12     printf ("%d", n);
13
14     return 0;
15 }
```

O problema apontado na linha 11 não causa erro de compilação. Trata-se de um erro de lógica, pois o fato do parâmetro `n` ter sido passado por cópia implica que a função recebeu um valor, digamos 3, retornou o valor 4, o qual por sua vez não foi utilizado no programa, portanto a variável `n` declarada em *main* permaneceu com o valor 3, que foi impresso em seguida. O correto seria ter escrito `n = soma_um (n)`. Uma outra solução teria sido passar o parâmetro por endereços, como vimos na última seção.

Embora o exemplo acima tenha apresentado uma situação de erro, muitas vezes é normal que os programas não usem o valor de retorno, por exemplo, quando elas retornam algum código de erro. É exatamente o caso das funções *scanf* e *printf* usados no mesmo programa, nas linhas 10 e 12, respectivamente. Estas duas funções retornam valores, conforme vimos em uma nota de rodapé na seção 11.

A primeira função retorna o número de variáveis corretamente lidas do teclado, enquanto que a segunda retorna o número de caracteres impressos, excluindo o byte nulo usado para terminar as strings.

No caso do *scanf*, na maior parte das vezes, o programador está interessado somente na leitura da variável e simplesmente despreza o valor do retorno. Isto não provoca de maneira alguma nenhum erro de lógica, muito menos de compilação.

Isto provoca uma brutal diferença semântica entre as duas linguagens. Começa a ficar mais clara a diferença *conceitual* no projeto das duas linguagens e as motivações destes projetos.

Como já dissemos, *Pascal* foi criada para fins didáticos, embora seja uma linguagem poderosa[11], enquanto que *C* foi desenvolvida para se construir um sistema operacional.

Vejam a flexibilidade proporcionada pelo projeto da linguagem *C*, esta diferença acima explicada, a qual foi tomada pelos projetistas da linguagem, é compatível e coerente com, por exemplo, os operadores terem valor de retorno, tal como mostrado na seção 5 que `x = 1` retorna o valor 1 e portanto pode ser usada em `y = x = 1`.

[11] O compilador *Free Pascal* foi feito em *Pascal*.

Mas este valor de retorno também está presente, por exemplo, em `x + 1`. Se `x` vale 1, então o valor do retorno é 2.

E por tudo o que foi escrito acima, é possível reescrever o exemplo anterior assim:

```
1  #include <stdio.h>
2
3  int main(){
4      int n;
5
6      scanf ("%d", &n);
7      n + 1;              /* errado de novo! */
8      printf ("%d", n);
9
10     return 0;
11 }
```

Neste caso, o compilador não acusa *warning*, não aborta a compilação e gera o executável. Novamente o programa está errado, mas o compilador aceita, pois ele confia no programador.

Para vermos o *warning* é necessário invocar o compilador com a opção `-Wall`. Vejamos o *warning* abaixo (*statement with no effect*), mas é certo que o executável foi gerado:

```
$ gcc -Wall lixo.c
lixo.c: In function ‘main’:
lixo.c:7:7: warning: statement with no effect [-Wunused-value]
    7 |     n + 1;            /* errado de novo! */
      |     ~~~~~
```

O próximo programa mostra um programa inútil que é aceito pelo compilador *C*:

```
1  int main(){
2      1;
3
4      return 0;
5  }
```

O mesmo código, quando escrito em *Pascal*, gera erro de compilação. Vejamos o programa e o erro abaixo (*Illegal expression*).

```
1  program lixo;
2
3  begin
4      1;
5  end.
```

```
$ fpc lixo.pas
Free Pascal Compiler version 3.2.0+dfsg-12 [2021/01/25] for x86_64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling lixo.pas
lixo.pas(4,6) Error: Illegal expression
lixo.pas(6) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

17 Organização da memória

Quando se faz um programa de maior porte pode ser necessário conhecer e entender a noção de escopo de variáveis. Isto está relacionado com a maneira como a memória do computador é utilizada. Este conhecimento também é necessário para entender algumas diferenças importantes entre o uso de variáveis em *Pascal* e em *C*. Descreveremos brevemente como isso é feito. A explicação será simplificada, pois os detalhes de como é exatamente feito foge do escopo deste texto.

A memória do computador possui uma organização mais refinada do que pensamos. Vejamos o esquema de como a memória é segmentada em alguns trechos que são utilizados pelos compiladores de maneira diferente para cada tipo de variável e escopo, conforme veremos a seguir na seção 18. Vamos explicar esta divisão de baixo para cima com relação à figura.

STACK	dados de funções
⋮	área livre
HEAP	alocação dinâmica
BSS	variáveis não inicializadas
DATA	variáveis inicializadas
TEXT	código do programa

Vamos também considerar o programa em *Pascal* da figura 6 para melhor explicar os conceitos. Em seguida veremos o mesmo programa codificado em *C* para podermos comparar as diferenças.

```
1  program imprime_media;
2
3  var x: longint;
4      y: longint = 2;
5
6  function media (a, b: longint): real;
7  var temp: real;
8  begin
9      (* temp nao eh necessario            *)
10     (* mas torna o exemplo mais completo *)
11     temp:= (a + b) / 2;
12     media:= temp;
13 end;
14
15 begin
16     read (x);
17     writeln (media (x, y));
18 end.
```

Figura 6: Um programa simples em *Pascal*.

A área TEXT contém o código binário do programa sendo executado. A arquitetura dos computadores, como sabemos, atende ao modelo Von Newmann, no qual

os dados e o programa são armazenados em memória. É este fato que distingue os computadores de outras máquinas que fazem cálculos. Normalmente a area TEXT inicia no primeiro endereço de memória.[12]

A área DATA contém as variáveis globais que têm sua inicialização na declaração delas. É o caso da variável `y` declarada e inicializada com o valor 2 na linha 4 do programa.

A área BSS contém as variáveis globais declaradas porém não inicializadas na declaração delas. É o caso da variável `x` declarada na linha 3 do programa.

A área HEAP contém as variáveis dinamicamente alocadas no programa. Este código não possui nenhuma com esta característica. O conceito de alocação dinâmica será visto posteriormente. Mas vale dizer que, até o momento, todas as variáveis que utilizamos são *estaticamente* alocadas, isto é, o sistema cuida de reservar espaço na memória para caber os bytes necessários para cada tipo de variável, evidentemente dependendo do tipo delas.

No caso, as variáveis `x` e `y`, sendo do tipo `longint` ganham 4 bytes cada uma. Como vimos, usamos 4 bytes para `x` na área BSS e outros 4 bytes para a variável `y` alocados na área DATA. Dizemos que isto é feito "em tempo de compilação", pois a única forma de alterar o espaço alocado é alterar o código fonte e recompilar.

Variáveis dinamicamente alocadas não tem este espaço alocado no momento da compilação. Caberá ao programador realizar esta tarefa "em tempo de execução", ou seja, enquanto o programa está rodando.

Finalmente, a área STACK (pilha, em português) é utilizada para armazenar os dados de funções e procedimentos, a saber: o ponto de retono para quem a chamou, os parâmetros e as variáveis locais da função.

No caso do nosso programa exemplo, teremos na STACK 4 bytes para o retorno da função, pois ela retorna um valor real, além de outros 12 bytes, sendo 8 para os parâmetros `a` e `b` (2 vezes 4 bytes para um longint) e finalmente mais 4 bytes para a variável local `temp` que é do tipo real.

O funcionamento da STACK foi explicado na disciplina Algoritmos e Estruturas de Dados I e não será revisto aqui. O livro [CSW20] pode ser consultado para rever os conceitos. Mas este processo é conhecido como "alocação automática", pois o sistema cuida do processo de alocação e liberação de espaço automaticamente conforme as funções são iniciadas ou encerradas.

Agora veremos na figura 7 o mesmo programa codificado em *C* e mostraremos as diferenças com relação ao que foi explicado para o caso da linguagem *Pascal*.

Na versão em *C* deste programa a principal diferença é que *main* é uma função, portanto as variáveis `x` e `y` são alocadas na STACK e não em DATA ou BSS.

[12] A linguagem *C* tem uma funcionalidade mais avançada que permite acessar os endereços desta área, por exemplo, usando ponteiros para funções.

```c
#include <stdio.h>

float media (int a, int b){
    float temp;

    temp = (a + b) / 2.0;

    return temp;
}

int main(){
    int x, y = 2;

    scanf ("%d", &x);
    printf ("%f\n", media (x, y));
}
```

Figura 7: Um programa simples em *C*.

18 Escopo de variáveis

A noção de *escopo*, o qual se aplica também a outros conceitos que não só variáveis, por exemplo, funções, tem a ver com a validade de uma variável (ou outros conceitos) em determinados trechos de código.

Em *Pascal* os escopos típicos de um programa básico são os escopo global e o local. As variáveis globais são definidas no cabeçalho do programa e valem globalmente em todo o programa, enquanto que as variáveis declaradas em funções tem o escopo definido somente no bloco delimitado pelo `begin ... end;` da função.

Por exemplo, considere este código completo em *Pascal*:

```
1  program escopo;
2  var x, y : real; (* validas entre as linhas 2 e 16 *)
3
4  procedure outro_escopo;
5  var x, n : integer; (* validas entre as linhas 5 e 9 *)
6  begin
7      x:= 10;  (* variavel local declarada na linha 5  *)
8      y:= 2.1; (* variavel global declarada na linha 2 *)
9  end;
10
11 begin
12     x := 2.468;
13     outro_escopo();
14     writeln (x, ' ', y);
15 end.
```

Embora tenham o mesmo nome, as variáveis cujo identificadores são `x`, declaradas nas linhas 2 e 5, são diferentes, pois estão em escopos diferentes. A primeira delas, na linha 2 do código acima, é do tipo real e tem validade no programa todo. O mesmo ocorre para a variável `y` declarada na mesma linha. Elas tem escopo global. Por isso a variável `y` pode ser acessada na procedure, na linha 8[13].

Por outro lado, a variável `x` declarada na linha 5, embora tenha o mesmo nome da outra variável `x`, tem escopo local, ela só é válida ao longo das linhas 5 e 9. Observem também que esta outra variável é do tipo `integer`, o que impediria, por exemplo, uma atribuição como `x:= 2.1` dentro da função.

A partir do momento em que a função é chamada, a variável `x` que vale é a local, declarada na linha 5 do tipo `integer`. Após o término da função, a variável `x` é a global. Isto ocorre porquê as variáveis locais das funções, bem como seus parâmetros, são alocadas na memória na *STACK*, conforme explicado na seção 17.

Em *C* os principais tipos de escopo são:

- Escopo de arquivo;
- Escopo de bloco;

Para melhor explicar, vamos mostrar uma versão alternativa para o programa acima feito em *Pascal* desta vez em *C*. A intenção é mostrar todos os tipos de escopo.

[13]Evidentemente isso deve feito com cuidado.

```
1  #include <stdio.h>
2
3  float x; /* escopo de arquivo, valida entre as linhas 3 e 17 */
4
5  void outro_escopo (int y){ /* escopo de bloco, valida entre as linhas 5
       e 9 */
6      int n;
7
8      x = 10;
9      y = 7;
10 }
11
12 int main(){
13     float y = 0; /* escopo de bloco, valida entre as linhas 12 e 17 */
14
15     x = 2.468;
16     outro_escopo (y);
17     printf ("%f %f", x, y);
18 }
```

Na linguagem *C* é comum fazermos programas que são compilados a partir de vários arquivos. Até o momento vimos somente programas completos contidos em um único arquivo.

Logo, o escopo de arquivo ocorre caso a declaração da variável seja feita fora de qualquer bloco ou parâmetro de função, incluindo as variáveis locais.

No programa em *C* acima, a variável `x` tem escopo de arquivo e é uma variável global neste caso. Ela será alocada na área BSS.

Por outro lado, o escopo de bloco é quando uma variável foi declarada dentro de um bloco ou lista de parâmetros. Neste sentido, a variável `y` declarada na linha 12, tem escopo de bloco. Ela só existe durante a execução da função *main*.

Por sua vez, o parâmetro `y` que aparece como parâmetro na linha 5, bem como a variável `n` declarada na linha 6 (que por sinal não foi utilizada), têm escopo de bloco e só são válidas durante a execução da função `outro_escopo`.

Como `x` tem escopo de arquivo, ela vale também na função e portanto tem seu valor alterado de 2.468 para 10. Por outro lado, o parâmetro `y` que recebeu uma copia da variável de mesmo nome da função *main*, isto é, recebeu o valor 0, mesmo tendo sido alterada para ter o valor 7 na função (linha 9), não resulta na alteração da variável `y` do *main*, pois o escopo da primeira é de bloco, isto é, durante a execução da função. Uma vez terminada a função, o valor de `y` em *main* permanece igual a 0.

Com isso, a saída do programa é: `10 0`.

A questão a ser entendida é: ao se escolher uma variável com um identificador qualquer em uma determinada linha de código, qual é o escopo dela? No caso de haver mais de uma variável diferente, porém com o mesmo identificador, estamos nos referindo a qual variável?

```c
#include <stdio.h>

int n; /* escopo de arquivo, variavel global */

int faz_algo () {
    int i, s; /* escopo de bloco, valem ate terminar a funcao */
              /* este s eh diferente do s do main             */
    i = 2;
    s = 0;

    for (int i = 0; i < n; i++){ /* outro i que vale no for somente */
        int e; /* escopo de bloco, vale somente no for */
               /* a cada iteracao, eh um e diferente   */

        printf ("-> leitura para i (for) = %d\n", i);
        scanf ("%d", &e);
        s = s + e;
        /* na ultima iteracao, i vale n - 1 = 4 */
    }
    /* aqui e nao existe mais e o i volta a ser o primeiro i (= 2) */

    printf ("i (fora do for) = %d\n", i);
    return s / i;
}

int main (){
    int s; /* escopo de bloco, vale somente no main */

    n = 5;
    s = faz_algo ();
    printf ("%d\n", s);


    return 0;
}
```

Figura 8: Exemplificando escopos em *C*.

Estes questionamentos fazem sentido e, para exemplificar, vamos considerar o código da figura 8. Trata-se de um código bizarro e de difícil leitura, ele foi concebido apenas para exemplificar os conceitos e produz como saída o seguinte:

```
-> leitura para i (for) = 0
1
-> leitura para i (for) = 1
2
-> leitura para i (for) = 2
3
-> leitura para i (for) = 3
4
-> leitura para i (for) = 4
5
i (fora do for) = 2
7
```

Como pudemos confirmar, este código é péssimo e não segue praticamente nenhuma das boas práticas de programação. Por vezes declarar variáveis com escopo de bloco no meio do código pode não ser uma boa prática em geral. O conceito existe e deve ser usado com critério, não somente porquê alguém viu um código na Internet.

Por exemplo, imagine que o programador queria que o return final da função fosse o cálculo da média, e não uma divisão por 2. Então ele não poderia ter usado o `for (int i...` e deveria ter optado somente por um `for (i...`, isto é, sem o `int i` no *for*.

19 Vetores

A linguagem *C* também permite usar vetores, mais ou menos nos mesmos moldes da linguagem *Pascal*. Sabemos que um vetor é uma área de posições contíguas de memória. Então na declaração abaixo reservamos 20 espaços para números do tipo *longint*. Sabendo que o tipo *longint* ocupa 4 bytes, então alocamos 80 bytes.

```
1 var v: array [1..20] of longint;
```

Desta maneira, em *Pascal*, garantimos não apenas os 80 bytes em posições contíguas, mas também podemos acessar os conteúdos através da indexação dentro da enumeração `[1..20]`. Neste texto daremos o nome de "rótulo" para os valores desta enumeração.

Como resultado, a primeira posição tem o rótulo 1, a segunda o rótulo 2, e assim sucessivamente até a última posição que tem o rótulo 20. Lembramos que os rótulos devem sempre ser do tipo ordinal (inteiro).

Um exemplo de uso de vetores em *Pascal* é mostrado a seguir.

```
1 program usando_vetores_1;
2 var v: array [1..20] of longint;
3     i: integer;
4
5 begin
6     for i:= 1 to 20 do
7         v[i]:= i;
8 end.
```

Também sabemos que em *Pascal* podemos definir 20 posições de *longints* de outras maneiras, tal como mostrado no código abaixo.

```
1 var v: array [0..19] of longint;
```

O que mudou com relação à versão anterior foram os rótulos, pois como a enumeração mudou para `[0..19]` agora a primeira posição tem rótulo 0, e assim sucessivamente até a vigésima e última posição que tem o rótulo 19. Mas foram alocados os mesmos 80 bytes definindo 20 posições contíguas de memória.

O programa abaixo usa esta definição para que o vetor produzido seja idêntico ao gerado no código anterior.

```
1 program usando_vetores_2;
2 var v: array [0..19] of longint;
3     i: integer;
4
5 begin
6     for i:= 0 to 19 do
7         v[i]:= i + 1;
8 end.
```

Na linguagem *C*, a maneira de declarar um vetor difere sintaticamente e tem como consequência um uso diferente. Segue abaixo um exemplo de um vetor em *C* que usa 20 posições de memória.

```
1 int v[20];
```

Em *C* não existe o mesmo conceito de rótulo, o que temos são as posições que iniciam sempre em zero. Consequentemente, o primeiro elemento do vetor é `v[0]`, o segundo é `v[1]`, e assim sucessivamente até o vigésimo e último que é `v[19]`. Isto não pode ser alterado.

O código em *C* a seguir é equivalente ao anteriormente escrito em *Pascal*. Notem na linha 5 um uso comum para programadores *C* do teste do *for* escrito `i < 20` e não `i <= 19`.

```
1 int main (){
2     int v[20];
3     int i;
4
5     for (i = 0; i < 20; i++)
6         v[i] = i + 1;
7 }
```

Um detalhe importante diferencia os vetores em *Pascal* dos vetores em *C*. A linguagem *C* nunca faz cópia de vetores quando eles são passados como parâmetros, enquanto que *Pascal* faz.

Para provar que em *Pascal* isso funciona, abaixo mostramos um programa exemplo que produz um vetor inicial cujos conteúdos são os números de 1 a 5. Em seguida uma função que recebe o vetor por referência e que zera este vetor.

```
1  program exemplo_que_funciona;
2
3  const MAX = 5;
4  type vetor = array [1..MAX] of integer;
5  var v: vetor;
6      n: integer;
7
8  procedure inicializar_vetor (var v: vetor; n: integer);
9  var i: integer;
10 begin
11     for i:= 1 to n do
12         v[i]:= i;
13 end;
14
15 procedure imprimir_vetor (var v: vetor; n: integer);
16 var i: integer;
17 begin
18     for i:= 1 to n do
19         write (v[i],' ');
20     writeln;
21 end;
22
23 procedure zerar_vetor (var v: vetor; n: integer);
24 var i: integer;
25 begin
26     for i:= 1 to n do
27         v[i]:= 0;
28 end;
29
30 begin
31     read (n);
32     inicializar_vetor (v, n);
33     imprimir_vetor (v, n);
34     zerar_vetor (v, n);
35     imprimir_vetor (v, n);
36 end.
```

Ao executar este programa compilado obtemos a saída esperada, Isto é, vemos que tanto a inicialização do vetor funcionou mas também a procedure que zera o vetor.

```
5
1 2 3 4 5
0 0 0 0 0
```

Muito bem, se trocarmos o parâmetro por referência por outro por cópia na função *zerar_vetor*, conforme ilustrado na figura abaixo, teremos a saída errada. isto é, o compilador *Pascal* faz a cópia de todo o vetor na stack, zera o vetor copiado e ao retornar para o programa principal, o vetor ainda está com os dados inalterados.

```
(* funcao errada que recebe o vetor por copia *)
procedure zerar_vetor (v: vetor; n: integer);
var i: integer;
begin
    for i:= 1 to n do
        v[i]:= 0;
end;
```

A saída é:

```
5
1 2 3 4 5
1 2 3 4 5
```

Em *C* os vetores nunca são copiados e para explicar o motivo, vamos inicialmente mostrar o mesmo programa acima escrito em *C*.

```
#include <stdio.h>
#define MAX 5

void inicializar_vetor (int v[], int n){
    int i;

    for (i = 0; i < n; i++)
        v[i] = i + 1;
}

void imprimir_vetor (int v[], int n){
    int i;

    for (i = 0; i < n; i++)
        printf ("%d ", v[i]);
    printf ("\n");
}

procedure zerar_vetor (int v[], int n){
    int i;

    for (i = 0; i < n; i++)
        v[i] = 0;
}

int main (){
    int v[MAX];
    int n;

    scanf ("%d", &n);
    inicializar_vetor (v, n);
    imprimir_vetor (v, n);
    zerar_vetor (v, n);
    imprimir_vetor (v, n);
}
```

Iniciando a explicação pela função *main*, observamos na linha 27 que para declarar um vetor a sintaxe é esta: `int v[MAX]`, isto é, `MAX` define o tamanho, que no caso é 5, os os elementos do vetor são `v[0]` até `v[4]`.

Passando para explicar como passar vetores nos parâmetros, podemos observar nas linhas 4, 11 e 19 que a sintaxe é simplesmente `int v[]` para o vetor, isto é, não é necessário escrever o tamanho do vetor.

O motivo disso é que o que está sendo passado não é o vetor, mas sim um ponteiro para o início do segmento de memória onde o vetor foi alocado. Em suma, poderíamos perfeitamente termos escrito `int *v` ao invés de `int v[]`. Em *C*, as duas formas são equivalentes, embora a que usamos é mais legível pois torna mais claro que estamos usando um vetor.

Falamos na seção 13 que em *C* os parâmetros são passados sempre por cópia. Mas falamos acima que não existe cópia de vetores. Parece contraditório, mas não é.

Sabemos que a lista de parâmetros especifica basicamente os tipos dos parâmetros. Consequentemente, `int v[]` significa o tipo deste parâmetro. Em *C*, este tipo não é um vetor, mas é um *ponteiro* (ou endereço) para o início do segmento de memória onde o vetor inicia.

Ao usarmos os colchetes contendo um valor inteiro, o que pode ser observado, por exemplo, nas linhas 8, 15 e 23 do programa acima, em uma sintaxe idêntica à da linguagem *Pascal*, estamos na verdade fazendo um deslocamento a partir do início deste segmento (do vetor) e o conteúdo desejado pode ser acessado. Vamos detalhar melhor considerando o código acima.

Quando o *for* da linha 7 inicia, o valor de `i` é zero. Logo, `v[0]` significa um deslocamento de 0 posições a partir do início do segmento de memória apontado por `v`, que naturalmente resulta na primeira posição do vetor, a qual recebe o valor `1`.

Na segunda iteração, `i` vale 1. Logo, `v[1]` significa que há um deslocamento de uma unidade de *tamanho de um inteiro*, isto é, um deslocamento de 4 bytes, e a segunda posição do vetor é acessada, a qual recebe o valor `2`.

Na última iteração, `i` vale 4. Logo, `v[4]` significa que há um deslocamento de 4 unidades de *tamanho de um inteiro*, isto é, 16 bytes, e a última posição do vetor é acessada, a qual recebe o valor `5`.

Por isso é impossível na linguagem *C* obtermos o mesmo efeito do programa exemplo errado que fizemos acima em *Pascal*, isto é, quando conseguimos copiar um vetor para a stack.

Uma vez isto explicado, podemos mostrar também uma função que faz a leitura de um vetor em *C*.

```
1  #include <stdio.h>
2
3  void ler_vetor (int v[], int tam){
4      int i;
5
6      for (i = 0; i < tam; i++)
7          scanf ("%d ", &v[i]); /* observe aqui */
8      printf ("\n");
9  }
10
11 void imprime_vetor (int v[], int tam){
12     int i;
13
14     for (i = 0; i < tam; i++)
15         printf ("%d ", v[i]);
16     printf ("\n");
17 }
18
19 int main (){
20     int v[20];
21     int i;
22
23     ler_vetor (v, 20);
24     imprime_vetor (v, 20);
25
26     return 0;
27 }
```

Observem na linha 7 o uso do `&` que aparece em `&v[i]`. Já explicamos sobre isso.

Esta apresentação dos vetores pode parecer complexa, mas o uso de vetores é igual ao uso em *Pascal*, a menos destes detalhes: (1): vetores sempre iniciam em zero; e (2): vetores sempre são passados nos parâmetros por ponteiros.

A observação (2) tem um efeito óbvio, que seria o equivalente em *Pascal* a uma chamada da função usando parâmetros por referência.

O leitor ou a leitora deste material que cursou Algoritmos e Estruturas de Dados 1 deve lembrar que falamos que vetores em *Pascal* devem sempre ser passados por referência, embora a linguagem permita a passagem por cópia.

Na ocasião, falamos que era boa prática, pois vetores normalmente ocupam muito espaço em memória, além do tempo necessário para a cópia, fora o fato de que esta cópia é feita na stack.

20 Structs

A linguagem *C* também permite a construção de tipos compostos, os registros. Em *Pascal* a denominação é *record*, enquanto que em *C* a denominação é *struct*. Mas o conceito é o mesmo, um registro permite que se tenham variáveis com tipos heterogêneos acessados pelo mesmo nome.

Um exemplo é um tipo que contém informações sobre funcionários, como esta abaixo, inicialmente em *Pascal*.

```
1 var funcionario : record
2         identificacao: integer;
3         salario: real;
4         tempo_de_casa: integer;
5 end;
```

Para acessar os elementos de um record deve-se usar um ponto, tal como ilustrado a seguir.

```
1     var f: funcionario;
2
3         funcionario.identificacaor:= 13;
4         funcionario.salario:= 1234,56;
5         funcionario.tempo_de_casa:= 7;
```

Vejamos agora como o conceito de registo é usado na linguagem *C*. Reescrevemos o programa acima nesta linguagem assim:

```
1 struct funcionario {
2     int identificacao;
3     float salario;
4     int tempo_de_casa;
5 }
```

Neste caso, o nome do tipo é `struct funcionario`. Observem que `funcionario` é o tag da struct e não o nome do tipo. Ele consiste de três campos, um inteiro que armazena alguma identificação do funcionário na empresa, seu salário que é um valor float e o tempo de casa, que é um inteiro.

Para acessar os elementos de uma struct deve-se usar um ponto, da mesma forma como foi feito em *Pascal*.

```
1 struct funcionario f;
2
3     f.identificacao = 13;
4     f.salario = 1234,56;
5     f.tempo_de_casa = 7;
```

A linguagem permite usar structs nos parâmetros de uma função bem como no tipo do retorno, como mostramos na figura 9.

Notem que definimos o tipo `struct funcionario` nas linhas 3–7. Em seguida definimos na linha 9 uma função cujo tipo de retorno é `struct funcionario`, a qual, por sua vez, não tem parâmetros.

Uma outra função foi definida na linha 19, ela não tem retorno, é do tipo `void`, mas recebe uma `struct funcionario` como parâmetro.

Na linha 29, a função `ler_dados_func` foi ativada e a variável `f`, na mesma linha, recebeu o retorno da função contendo os dados do funcionário. Na linha 30 esta *struct* foi passada como parâmetro da função `imprimir_dados_func`.

```
1  #include <stdio.h>
2
3  struct funcionario {
4      int identificacao;
5      float salario;
6      int tempo_de_casa;
7  };
8
9  struct funcionario ler_dados_func(){
10     struct funcionario f;
11
12     scanf ("%d", &f.identificacao);
13     scanf ("%f", &f.salario);
14     scanf ("%d", &f.tempo_de_casa);
15
16     return f;
17 }
18
19 void imprimir_dados_func (struct funcionario f){
20
21     printf ("%d\n", f.identificacao);
22     printf ("%f\n", f.salario);
23     printf ("%d\n", f.tempo_de_casa);
24 }
25
26 int main(){
27     struct funcionario f;
28
29     f = ler_dados_func ();
30     imprimir_dados_func (f);
31
32     return 0;
33 }
```

Figura 9: Um exemplo completo em *C*.

21 Strings

Não por acaso deixamos o conceito de *strings* para a última seção deste material, pelo fato de não termos estudado este tema em Algoritmos e Estruturas de Dados 1. Mas para fins de completude, vale o conteúdo.

A linguagem *Pascal* possui um tipo básico `string`, que no fundo é um vetor de `char`. Porém, como *Pascal* abstrai vários conceitos de aprendizes, ela implementa o conceito de uma maneira que seu uso seja bastante natural para o programador. Vejamos um programa exemplo.

```
1  program strings_em_pascal;
2
3      var s1: string[20]; (* pode ter no maximo 20 caracteres *)
4      s2: string;         (* o tamanho aqui eh 255             *)
5
6  begin
7      read (s1);
8      s2:= 'o maior campeao brasileiro eh o Palmeiras';
9
10     if s1 > s2 then
11         writeln (s1,' eh maior')
12     else
13         writeln (s2,' eh menor ou igual')
14 end.
```

O compilador permite que o programador trate *strings* como qualquer outra variável básica. Notem que o programa acima permite leitura com o comando `read`, atribuição, comparações e impressão. Nas comparações, ele usa ordem lexicográfica. Por exemplo, "abacaxi" é menor do que "manga", pois "a" é menor do que "m".

Um detalhe que importa é que o compilador sabe o tamanho útil da *string* e isso faz muita diferença. No código acima, o tamanho da *string* `s1` é 20 e o de `s2` é 255.

Em *C* não existe o tipo *string* da mesma forma como em *Pascal*. Na linguagem *C* uma *string* é um vetor de caracteres e por este motivo não existe a mesma facilidade na leitura, atribuição, comparação e impressão, pois não se pode ler um vetor em *C*. Mesmo um vetor de inteiros, não importa o tipo, os elementos devem ser lidos um por um, vetores não podem ser comparados ou impressos a não ser pela comparação ou impressão um a um, tipicamente em um comando *for*.

A linguagem *C* não é a melhor linguagem de programação para lidar com *strings*, tudo é complicado e pode levar a programas que não são seguros. Por exemplo, embora seja possível usar a função *scanf* com a opção de formatação `%s`, isto não é recomendado por questões de segurança.

Por este e outros motivos, trabalhar com *strings* em *C* normalmente é feito com o uso de uma biblioteca que contém funções que as manipulam. Esta biblioteca se chama `string.h` e deve ser incluída no código. Ela possui funções tais como as apresentadas na tabela abaixo.

função	o que faz
fgets	lê uma string do teclado
fputs	imprime uma string na tela
strncpy	copia uma string em outra
strncmp	compara duas strings

Um dos principais motivos é que, sendo um vetor de caracteres, é complicado saber onde termina a parte útil que um vetor usa. As falhas de segurança exploram isso provocando acesso à posições de memória que estrapolam os limites do vetor.

A linguagem convencionou que toda *string* deve terminar com um `\0`, que é um byte que significa literalmente zero. Vejamos um esquema de como isso é feito, supondo que a declaração feita foi `char s[10]`, isto é, uma *string* de tamanho máximo 10 (índices de 0 a 9).

0	1	2	3	4	5	6	7	8	9
m	a	r	c	o	s	\0	?	?	?

Como se trata de um vetor, podemos acessar os conteúdos da maneira tradicional. Por exemplo, `v[0]` é `m`, `v[4]` é `o`, e `v[6]` é `\0`.

Reforçando, o uso do `\0` serve para indicar o término da *string* propriamente dita, que evidentemente pode ser menor do que o tamanho total do vetor. Isto é muito diferente de *Pascal*, que como dito anteriormente, "sabe" o tamanho útil da *string*.

Por isso, em uma declaração como `char s[10]`, embora tenhamos espaço para 10 caracteres, um deles deve ser usado pelo `\0`. Consequentemente, toda *string* tem um espaço útil a menos com relação ao tamanho do vetor. Portanto, a maior *string* que pode ser escrita um um vetor de 10 caracteres tem tamanho 9. Cabe ao programador cuidar disso e cuidar atentamente para não escrever a partir da posição em que está o `\0`.

22 Conclusão

Este texto termina aqui. Como dissemos, não é nosso objetivo escrever um texto sobre a linguagem *C*, mas sim prover um material básico comparativo para que a transição inicial de *Pascal* para *C* possa ser feita da melhor maneira possível.

Reforçamos que o leitor e a leitora devem procurar na literatura mais material a respeito desta nova linguagem, em particular, o livro dos criadores da linguagem [KR88].

Com relação à disciplina Programação 1, ela também não trata da linguagem *C*, mas sim de Tipos Abstratos de Dados usando esta linguagem como ferramenta didática.

A vantagem de usar *C* para este assunto é que ele é intimamente ligado ao conceito de alocação dinâmica, assunto para o qual esta linguagem é bastante apropriada e recomendada. Claro que isto também pode ser feito em *Pascal*.

Não escreveremos sobre isto neste material, provavelmente haverá outro sobre o assunto.

Referências

[CSW20] Marcos A. Castilho, Fabiano Silva, and Daniel Weingaertner. *Algoritmos e Estruturas de Dados I.* Universidade Federal do Paraná, 2020. Licença Creative Commons BY-NC-ND.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.