Capítulo 2

O segundo programa em C

2.1 Programas e erros...

Observação: 3 *Regras de trabalho*

- *Vamos sempre começar por lhe apresentar um programa;*
- *você deve digitá-lo*[1] *num editor de textos*[2]*;*
- *depois rode o programa com o compilador que estiver à sua disposição e em seguida leia os comentários que faremos e também os comentários feitos pelo* `compilador`*, sobretudo porque você pode ter esquecido de digitar algum direcionador de memória* & *;*
- *tome a iniciativa de fazer alterações nos programas usando a experiência que for adquirindo, mas grave-os com nomes diferentes; Aqui você vai adquirir experiência sobre um erro muito comum:* perder programas porque gravou por cima algum outro programa*. Esta é uma dor de cabeça comum a todos os programadores. Tenha por hábito fazer* `backup`*, cópia de reserva, dos seus programas. Tenha o cuidado de gravar a alteração de um programa, com outro nome:*

  ```
  prog01.c prog02.c ...prog101.c
  ```

  e você vai sempre encontrar o mais recente, ou algum mais antigo que funcionava tão bem*...*
- *aos poucos deixaremos de transcrever os programas no livro, é mais prático que você leia os programas com um editor de textos, inclusive quando os rodar, é interessante tê-los numa tela ao lado.*

[1] Todos os programas do livro se encontram distribuidos em disco para economisar-lhe a digitação. Basta "carregá-los" para o editor.

[2] grave os programas no modo texto, (sem acentos) Se você estiver usando alguma *IDE* (*ambiente integrado*) o editor é próprio para programação. Se estiver usando algum editor como `word`, tome o cuidado para gravar os programas no modo texto.

Exemplo: 2 *O primeiro programa*

```
/* Programa prog01.c
        Assunto: le uma palavra pelo teclado e a imprime
        por Tarcisio Praciano Pereira - 10 licoes para aprender C
        Sobral, julho de 2002 - UVA
        */
# include < stdio.h > // (1) leitura da biblioteca stdio.h
# include "traducao.h"// (2) leitura da biblioteca traducao.h

inteiro principal() // (3) inicio do programa, tipo palavra
inicio
        palavra coisa; // (4) declaracao de variavel
        imprima("%s\n",
                "escreva alguma coisa pelo teclado, palavra, numero..."); // (5)
        ler("%c",&coisa); // (6)
        imprima("%s%c\n","O primeiro caracter da coisa foi -- > ",coisa); // (7)
        imprima("%s\n","============================");
        voltar(0); // (8)
fim // (9)
```

Comentários do programa.

Os comentários *são parte integrante de um programa e de forma alguma devem ser considerados um "*apêndice extra perfeitamente dispensável*". Um programa é uma peça de abstração, escrito em linguagem técnica, em geral muito conciso, e, consequentemente, difícil de ser lido. Os* comentários *vêm suprir informações complementares e tornam o programa* legível.

Comentários podem ser caracterizados de duas formas:

- *Com duas barras, "//". O compilador ignora o que venha depois até o final da linha.*
- *Entre os sinais '/' e '*' no começo, e revertidos ao final '*' e '/'. Veja logo no inicio do programa.*

Quando se vai escrever um comentario longo, este segundo metodo,é o mais adequado. Pequenos comentários, como acima, depois de um comando, é preferivel usar o primeiro metodo. Mas a decisão e o estilo são seus. Há programadores que usam colunas de asteriscos no início e no final de cada linha de um bloco de comentários para torná-lo mais ostensivo. Vale tudo, desde que você não se atrapalhe (nem atrapelhe os outros) com a poluição visual...

Os comentários servem para explicar o programa (inclusive para o próprio autor...) ou também como parte do planejamento do trabalho. No inicio do programa os comentários dizem o que o programa faz, quem fez o programa, as modificações que se pretendem fazer nele, os defeitos que ainda existam, etc...

Os comentários que fizemos usando "//" tem uma numeração que vai servir de referência para uma seção de observações que costumamos fazer ao final dos programas. Veja, por exemplo, `musica.c`*, não tente compreender o programa agora, veja apenas como estamos usando os comentários numerados. E, claro, você pode rodar e ler o programa, mas observe (leia o programa) ele necessita que no sistema exista um programa chamado* `bell` *que acione o* `alto-falante` *do computador. Troque* `bell` *pelo nome certo. Se este programa não existir nada vai acontecer.*

1. *A linguagem C, como toda linguagem moderna, é expansível, quer dizer, você pode criar novos comandos, são as funções. Cada função é um novo comando. Estes comandos novos ficam com freqüência dentro de arquivos chamados 'bibliotecas'. O programa começa lendo a biblioteca padrão do C para Input/Output - < stdio.h > Entrada/Saida. Leia também a nossa biblioteca,* `traducao.h`*, em que fizemos as traduções dos comandos da linguagem C. As bibliotecas são arquivos com a extensão "*`.h`*" e ficam colocados em diretórios específicos que o* `gcc` *sabe quais são. Quando quisermos incluir uma biblioteca nossa, como "*`traducao.h`*", temos que usar aspas em volta da biblioteca e então* `gcc` *vai procurá-la no diretório de trabalho, ou no diretório indicado pelo caminho que indicarmos:*

   ```
   # include "/home/meu_nome/C/minha_biblioteca.h"
   ```

2. *erro grave concluir da observação anterior que você pode construir uma linguagem C especial para você, com seus próprios comandos. É verdade, mas seria inútil. Linguagens, mesmo de computador, existem para que as pessoas se comuniquem. O conhecimento é social e não individual. Só podemos ser avançados na medida em que o grupo social o seja junto conosco. Não teria sentido criar o* seu *C! Mas tem sentido* pensarmos em programar em Português*, aqui no Brasil, e seguir entendendo programação em Inglês.*

3. *Exercício: Experimente! Apague*

   $$\# \; include < stdio.h >$$

   e rode o programa:

   ```
   gcc -Wall prog01.c -oprog.
   ```

   Como resultado você vai receber a informação que `printf(), scanf()` *estão sendo usados pela primeira:*

   ```
   prog01.c:14:
   warning: implicit declaration of function 'printf'
   ```

   O `gcc` *vai ignorar, pedantemente, a nossa tradução "imprima"e vai lhe falar de "printf"fazendo o mesmo com "scanf". Porque nós não traduzimos o compilador.*

4. *estrutura de um programa Há dois tipos de funções num programa:*

- *main() A função "principal()",*
- *e as outras que a "principal()"chama.*

Todo programa tem que ter uma função `principal()` *e depois deve ter outras funções que executam tarefas específicas, tarefas auxiliáres.*

Neste programa a função "principal()"chama apenas outras funções que se encontram definidas na biblioteca 'stdio.h'. Portanto as outras funções podem já existir em alguma bibilioteca e inclusive podem já ter sido usadas e testadas por outro programa (melhor ainda).

5. *Cada função é um pequeno programa. Neste sentido a linguagem C já nasceu "moderna", no espirito de programação modular. Um programa se constitui essencialmente de sua* função principal *que irá colocar em ação os demais atores, as outras funções, que foram feitas sob-medida para executar pequenas tarefas específicas. Assim, um grupo grande de funções pode existir para compor, quando necessário, um determinado programa. Isto se chama* hoje `reciclagem de programas` *ou em inglês,* `re-use of programs`.

6. *Variáveis, funções, tem que ter um tipo. A sintaxe da declaração de variáveis é*

$$< tipo\_dados >< nome\_da\_variavel >;$$

Pode haver várias variáveis do mesmo tipo na mesma declaração, separadas por vírgulas.

7. *leia acima... A linguagem C não distingue variáveis e funções em primeira instância. Inclusive o compilador, quando encontrar erros, vai se referir às variáveis como funções.*

8. *A função 'imprima()' (printf()) exige que comecemos dizendo que* tipo de dados *lhe fornecemos para imprimir: "%s", quer dizer que vem uma 'frase', (string), para ser impressa. O símbolo "\n" é um 'comando': mudança de linha. Você verá depois que "\n" não é um comando, agora adianta pouco discutir esta diferença semântica.*

9. *'ler()' é a tradução de 'scanf()' que é um comando muito rápido e pode conduzí-lo a erros. Use "leia()"(fgets()) como você verá nos próximos programas. Usamos "ler()"para que o programa ficasse simples, mas é um defeito.*

   Warning, Warning, Warning, Warning....!!!!

   Rode `prog04_2.c`[3] *para ver o risco do* `scanf()`. *Leia os comentários, dentro do programa.*

10. *Aqui 'imprima' tem dois formatadores de dados, %d para inteiros, e %s para frases (strings).*

[3] no diretório `BC` este programa se chama prg04_2.c

11. *Toda função da linguagem C deve terminar com o comando "*`voltar()`*" (return), e,* com frequência*, com um número. Mais a frente você vai ver que isto é falso... Este número pode ser usado para fornecer ao sistema informações sobre o comportamento da função, erros cometidos por ela, (na verdade pelo programador...). Devolvendo zero significa que tudo correu bem.*

12. *Os algoritmos começam com "inicio", "{", e termina com "fim" "}". De forma mais precisa, Os "*blocos lógicos*" começam com "inicio" "{" e terminam com "fim", "}".*

Vocabulário: 2 *Bloco lógico e variável local*

- bloco lógico *é um* conjunto de ações*, de comandos, que tenham um objetivo específico. Uma função é um* bloco lógico*, mas dentro de funções você pode encontrar mais* blocos lógicos*. É um conceito difuso mas que aos poucos você compreenderá.*

  Sempre que você encerrar um conjunto de comandos entre chaves *você terá criado, para a linguagem C, um bloco lógico.*

  Veja a importância deste fato: no início de um bloco lógico *você pode definir variáveis locais que deixam de existir à saída do bloco.*

- variável local *são variáveis criadas dentro*[4] *de um bloco lógico. Elas tem sua existência associadas ao bloco lógico em que forem criadas. O conceito que se opõe a este é o de* variável global*. Podemos dizer que você deve evitar o uso de variáveis globais e se habituar a usar apenas variáveis locais.*

- variável global *São variáveis criadas fora de blocos lógicos e que portanto ficam sendo reconhecidas por distintos blocos lógicos. Algumas vezes somos forçados a criar este tipo de variável, entretanto devemos inclusive deixar indicativos no cabeçalho do programa apontando a existência delas numa tentativa de eliminá-las, se possível. Como estas variáveis tem uma existência ampla, há riscos que no planejamento nos esqueçamos de suas presenças e elas, assim, interfiram nos resultado de forma inexperada. Variáveis globais são um risco a ser evitado. Quando você tiver que definir uma variável global, indique isto no cabeçalho do programa como uma forma de aviso de existe um* problema *no programa.*

No disco que acompanha este livro, há um diretório chamado `BC` em que os programas foram testados em ambiente `Borland, BC ou TC`. Também os nomes dos programas ficam dentro do limite do DOS de oito caracteres. Programas que fujam a este padrão tem seus nomes corrompidos pelo *ambiente* de programação da `Borland`.

[4] no início de um bloco lógico

Exercícios: 4 *Alterações no programa*[5] *prog01.c*

1. *Compile e roda o programa*[6] `prog01.c`.

2. *Se você digitou uma palavra, o programa guardou somente a primeira letra. Experimente digitar números, analise o resultado.*

3. *Altere*[7] *prog01.c, subsitua*

```
palavra coisa;  // (3) declaracao de variavel
```

   por

```
palavra coisa[30];  // (3) declaracao de variavel,
```

   agora compile-o e veja que o resultado foi desconcertante, uma montanha de erros foram anunciados.

   gcc -Wall -oprog prog01.c
   prog, para executar o programa.

   Leia o relatório de erros e veja nos próximos exercícios a saída.

4. *Substitua*

```
ler("%c",&coisa); // (5)
imprima("%s%c\\n","A primeira letra foi -> ",coisa); // (6)
```

   por

```
ler("%s",coisa); // (5)
imprima("%s%s\n","A primeira letra foi -> ",coisa); // (6)
```

   e veja o resultado digitando uma palavra com até 29 letras.

5. *Digite também uma palavra com mais de 30 letras. Digite duas palavras,* quer dizer, duas "strings" separadas por um espaço, *por exemplo. Analise por que dá errado.*

6. *Faça algumas experiências alternado as tres linhas aqui discutidas e veja os resultados. Mas, somente rode os programas se o compilador não apontar erros ou "warnings" porque este programa usa* vetores de caracteres *que são* ponteiros. *Ponteiros devem ser usados com cuidado porque fazem acesso direto à memória da máquina.*

[5] no diretorio `BC` este programa se chama prg01.c
[6] em `BC` prg01.c
[7] no diretorio `BC` este programa se chama prg01.c

Algo pode ter saído errado quando você rodou este programa. Vamos analisar o que pode ter acontecido. Primeiro você pode ter digitado um número. Experimente, se não o fez.

Se você tiver digitado um número, o programa rodou, sem problemas, apesar de que ele não tenha sido feito para isto. Observe que foi um erro[8], porque se você desejasse que o programa lesse um número, você deveria ter dito isto. Foi um erro *do programador*, não do programa. Programas não erram, eles fazem apenas aquilo para o qual foram planejados[9].

Com o programa modificado, se você tiver escrito uma frase de verdade, o programa só imprimiu a primeira palavra. Porque quando ele encontrou o primeiro espaço considerou encerrada a leitura da *variável* `coisa[30]` e, naturalmente, somente imprimiu a primeira palavra.

Experimente colocar a frase entre aspas, veja o resultado.

Lição: 1 *C roda aquilo que não se espera...*

Um dos mitos por traz da linguagem C é que com ela se fazem programas que rodam muito rápido. Isto pode ser verdade, e uma das razões se encontra no fato de que o compilador espera que você não cometa erros e reduz ao mínimo a verificação da lógica do programa.

É comum se dizer que um programa em C sempre faz alguma coisa... mesmo que não seja o que se espera. Isto não é um defeito da linguagem, acontece que C é considerada uma linguagem para programadores profissionais, poristo não tem sido considerada uma linguagem para inciantes...

O programa acima foi feito para escrever frases, mas escreve também números. Num programa grande e complexo isto poderia ser um desastre. Claro, o programa lhe pedia para escrever alguma coisa, isto não se faz! A comunicação usuário-programador *deve ser mais completa e sempre clara. Além disto o próprio programa deve ter recursos de verificação do que o usuário está fazendo e deve então orientá-lo a repetir a operação de forma correta. Na verdade "programas" só fazem aquilo para o qual foram planejados.*

Observação: 4 *Comentários dos exercícios*

- *Há uma diferença fundamental entre*

  ```
  palavra coisa;  // (3) declaracao de variavel
  ```
  e
  ```
  palavra coisa[30];  // (3) declaracao de variavel,
  ```

  No primeiro caso, C entende que "coisa" é um simples caractere, um dos 256 caracteres que você pode produzir com o teclado.

  No segundo caso, C entende que "coisa[30]" é um "vetor" de caracteres.

  C enumera os índices a partir de zero, que dizer que você tem direito de usar

  coisa[0], coisa[1], . . . , coisa[29], coisa[30]

  e se o vetor estiver construido corretamente, coisa[31]='\0' é o `NULL`*, um caracter especial que marca o fim dos vetores.*

  Você não tem o direito de fazer uso deste último espaço de forma diferente sem o risco de erros no seu programa, é como se você guardasse uma garrafa cheia destampada*...*

[8] É preciso desmistificar os erros, errar é natural de quem está aprendendo, simplesmente.
[9] tem gente que diz "desenhados", que horror.

Este último caractere se chama NULL e serve para marcar o fim dos vetores corretamente construidos. Claro que você pode construir vetores incorretamente, correndo riscos de que seus dados se misturem produzindo erros.

- *Veja a diferença entre estas duas linhas:*

```
ler("%c",&coisa); // (5)
imprima("%s%c\\n","A primeira letra foi -> ",coisa); // (6)
```

e

```
ler("%s",coisa); // (5)
imprima("%s%s\n","A primeira letra foi -> ",coisa); // (6)
```

O formatador %s anuncia às funções "imprima()" e "ler()" que virá um "vetor de caracteres" ou string.

Aqui também se encontra uma das dificuldades no uso de 'ler()' (scanf()). Esta função da linguagem C é muito sensível... ela sempre guarda os dados através dos seus endereços. É o que se encontra expresso em

```
ler("%c",&coisa); // (5)
```

que poderíamos traduzir como

> *"guarde o caracter que vem pelo teclado na variavel cujo endereço é &coisa".*

- *Tudo muda quando declaramos "palavra coisa[30];" porque coisa[30], sendo um vetor, é uma sucessão de 30 endereços, para C, um vetor de endereços. Antes chamamos de* vetor de caracteres. *Mais a frente você vai ver que existem outros tipos de vetores. Todo vetor é um* vetor de endereços de um certo tipo de dados. *Aqui estamos com um* vetor de caracteres. *Leia a respeito de* tipos de dados *no capítulo 7, mas faça apenas uma leitura rápida.*
- *Usar o* redirecionador de endereços *num endereço é errado. Vamos dizer isto de outra forma, usando uma linguagem técnica.*
  1. *Existem variáveis do tipo "endereço", são os* `ponteiros`.
  2. *Além disto precisamos declarar que tipo de ponteiro. Leia mais a respeito no capítulo sobre ponteiros, veja no índice remissivo.*
  3. *Quando uma variável for* um ponteiro, *então não será correto usar o direcionador "`&`" na frente desta variável em* `ler() scanf()`. *Seria o mesmo que dizer*
     guarde este dado no endereço endereço X.
- *Isto justifica que deixemos de lado a funcao 'ler()' (scanf()) nos primeiros passos do uso da linguagem. Iremos fazer um uso de um método mais complicado, entretanto mais seguro,evitando esta discussão inicial deixando-a para um momento em que o estudante de C esteja mais a vontade com a linguagem.*

  Rode e leia o programa[10] `prob_scanf.c`. *Sobretudo leia os comentários ao final do programa.*

[10] dentro do `BC` o nome aparece cortado, tem mais de oito caracteres. Troque o nome arquivo para `prbscanf.c`

- *Existe um compilador para a linguagem C, chamado* `checker` *que faz uma verificação do uso da memória pela variáveis do tipo ponteiro e pode alertar para problemas deixados dentro de um programa.*

  Sintaxe: checker -gcc programa.c [comandos do gcc] Não pude encontrar um similar para `DOS`*, não sei se existe.*

Observação: 5 *Programas robustos.*

Agora ficou claro que não se espera que você escreva números. Mas para frente você vai aprender a criar estruturas de controle *de entradas de dados que aconselharão o usuário a re-escrever o que se pede, no caso de que ele tenha respondido com alguma* inconveniência*. São métodos para fazer programas seguros, ou robustos. Está cedo, entretanto, para uma discussão mais aprofundada sobre este assunto.*

Vocabulário: 3 *Dados, variável*

- *dados É complicado discutir o que são dados, um programa todo pode ser um* dado*... mas se tentarmos simplificar as coisas para começar a discutir, programas servem para manipular dados, quer dizer transformar uma* informação bruta *em uma* informação tratada, manipulada, lapidada... "processada"*.*

  O programa prog01.c parece rídiculo, pede um nome e volta a escrevê-lo na tela. Mas ele poderia ter pedido o "seu nome" para comparar com os dados de um banco interno de nomes afim de permitir-lhe ou negar-lhe a entrada no sistema. Então o seu nome é uma informação que, comparada com um banco de clientes, diz se você pode ou não ter acesso às outras informações.

- *variável Para guardar dados se criou um sistema engenhoso que usa tres etapas.*

  - *Uma tabela formada de palavras, chamadas identificadores, associadas aos endereços, uma tabela de alocação que é basicamente o que cada usuário, ou programador, usa. Estas palavras são chamadas "variáveis". Em certas linguagens esta tabela se chama de "espaço de nomes".*
  - *A cada tal variável se associa um endereço inicial no segmento de memória reservado para tal onde se inicia o conteúdo da variável e, pelo seu tipo, se calcula onde deverá terminar reservando-se o próximo endereço inicial de outra variável. Por esta razão você precisa declarar o tipo da variável que você pretende usar.*
  - *Uma associação dos elementos do espaço de nomes com seus respectivos enderecos iniciais, chamada ainda de "alocação". Esta alocação é* dinâmica *porque as variáveis são criadas e destruídas portanto os endereços iniciais mudam durante a execução de um programa. Você, e qualquer outro programador, não precisa se preocupar com isto, está é uma atribuição do* `sistema operacional`*.*

  Em C, o uso do endereço, pelo programador, é uma das características da linguagem havendo um tipo particular de variável que opera sobre o

endereço, as variáveis do tipo `ponteiro`*. Você pode programar em C sem usar ponteiros, mas o atrativo é que podemos acelerar os programas com seu uso, como também torná-los mais perigosos. Como já dissemos, pegar nos fios elétricos de mal jeito pode levar à morte, mas você não prefere viver no escuro... aprenda a usar ponteiros. Mas deixe para fazer uso deles quando tiver uma compreensão segura de como funcionam.*

Veja um exemplo bem simples que mostra a importância do uso de endereços para acelerar a manipulação da informação.

Exemplo: 3 *Uso de ponteiros e a velocidade Pense no seguinte exemplo, um enorme armazem em que uma instituição tem guardados todo o seu acervo, tomemos o caso de um museu.*

Sempre novos itens chegam e seria impossível prever de antemão onde cada um deles seria guardado, inclusive será preciso adquirir de vez em quando uma nova casa para abrigar o acervo sempre crescente do museu, (imagine que os governantes se preocupam com os museus e sempre estão liberando mais verba para enriquecer a instituição...)

Como manter o acervo organizado? As peças vão chegando e simplesmente recebem um número de ordem de chegada*, e um endereço em que se encontram guardados (pode ser a identificação de uma sala em um determinado prédio). No fim do dia a lista que identifica os itens do acerto é novamente ordenada (por ordem alfabética) o que significa que se* re-orientam os ponteiros *entre objetos e endereços. É muito menos pesado trocar a indicação*

$$piano21 \rightarrow casa10 \text{ - } sala\ 30$$

do que manter sempre *o piano21 no mesmo lugar... quando houver um concerto e for preciso levar o piano21 para o auditório, o lugar dele não precisa ficar reservado, outros objetos podem ocupar o seu lugar, e depois ele pode ser guardado n'outro endereço e o cadastro vai simplesmente ser re-organizado com a troca*

$$piano21 \rightarrow casa15 \text{ - } sala\ 3$$

porque seria muito mais difícil estar mudando de lugar pianos.

A exemplificação acima se aplica literalmente a qualquer banco de dados, quer dizer um programa que associe distintas informações como nomes de pessoas objetos, endereços, contas bancárias*, por exemplo um catálogo telefônico.*

Um banco de dados *fica inútil se não estiver* ordenado*, porque então simplesmente os dados ficaram perdidos, imagine um* catálogo telefônico *em que os nomes dos usuários não apareçam em ordem alfabética, seria inútil!*

Mas os bancos de dados *são dinâmicos no sentido de que sempre estamos colocando novos nomes ou retirando nomes ou qualquer outro tipo de dados, consequentemente vivem desordenados. Como os dados podem ocupar muito espaço na memória do computador, (igual pianos no acervo do museu), é preferivel ordenar os endereços que são relativamente pequenos. Aí entram os ponteiros para acelerar o processamento.*

Observação: 6 *Abstração e variável*

Os computadores, atravez de vários sistemas de códigos, podem guardar informações praticamente de quase todo tipo. A palavra "abstração" adquiriu um sentido novo com a ciência da computação, *ela distingue as informações pelo que se entende hoje como de* níveis de abstração. *Antes* abstração *era sinônimo de difícil, hoje caracteriza o nível de complexidade de um conceito no sentido de que ele comporte uma quantidade maior de informações. Por exemplo, um número guarda um tipo de informação que podemos considerar como de primeiro nível de abstração.*

Mas, um par ordenado de números tem um nível maior de abstração porque pode guardar informação não numéricas como endereços de apartamentos de um prédio de vários andares. Em ternos ordenados de números poderiamos guardar a informação de quantos habitantes existe por apartamento... Nos dois últimos casos os "números" deixaram de ser números e passaram ser códigos.

Esta é evolução dos "números", que vistos como códigos, criam novos tipos de dados e sucessivos niveis de abstração. Veja que "número de telefone" não *é número, e sim código... você não somaria dois números de telefone, somaria ?*

A memória de um computador tem um endereçamento "dinâmico" semelhante a de um edifício de apartamentos. Dinâmico *porque a cada instante mudam*

- *os endereços;*
- *os habitantes;*
- *o tamanho dos apartamentos.*

Os habitantes são as "variáveis". Aqui, à diferença com o que ocorre num edifício de apartamentos, o tamanho dos apartamentos se adaptam ao tamanho das variáveis... Ao definir uma variável se estabelece o endereço do ponto inicial de memória que ela vai ocupar e do tipo de dado que ela vai representar, (ou "conter", com se diz comumente), e assim se marca o início de uma próxima "variável". ou o outro endereço inicial. Se o sistema operacional *for bem feito, ele fica monitorando o uso das variáveis para realocar o espaço na memória, levar para o disco, ou trazer de volta do disco,* páginas *de memória.*

Veja a nova formulação de prog01.c → prog02.c.
Primeiro compile[11] e rode `prog02.c`:

gcc -Wall -oprog prog02.c

prog, para executar o programa, ou ./prog

Exemplo: 4 *prog02*

```
/* Programa prog02.c
Assunto:le uma frase pelo teclado e a imprime
   Programa errado, compile e corrija o erro. Ver exercicios.
   por Tarcisio Praciano Pereira - 10 licoes para aprender C
   Sobral, julho de 2000 - UVA
*/

#include <stdio.h>
#include <string.h>
#include "traducao.h"


palavra principal()
```

[11] em `BC` procure prg02.c

```
inicio
palavra coisa1[30], coisa2[30], coisa3[30]; //(0)
   imprima("%s%\n", "escreva uma frase curta pelo teclado, ");// (1)
   imprima("%s\n", "digamos, com tres palavras.. "); // (2)
   imprima("%s\n"," pode ser o seu nome, por exemplo ");//(3)
   ler("%s%s%s",coisa1,coisa2,coisa3); //(4) ainda usa 'scanf'
imprima("%s%s%s\n",coisa1,coisa2,coisa3); //(5)
fim


/* Comentarios:
  0) declaracao de tres variaveis - vetores do tipo string.
1,2,3) mensagens orientando o usuario a fornecer os dados.
em (1) tem um erro que o compilador detecta.
4) Leitura de dados com 'ler' (scanf) observe a ausencia do
direcionador de endereco &, desnecessario porque as
variaveis sao do tipo ponteiro, declaracao implicita.
5) Um unico 'imprima' imprime todos os dados.
*/
```

Rode o programa para ver o que acontece. Programinha ruim, não é? Claro, estamos apenas começando. Vejamos alguns defeitos e como poderíamos corrigí-los.

Exercícios: 5 *Alterando e entendendo prog02.c*

1. *Compile e rode o programa* `prog02.c`.
2. *Quando compilado, o compilador reclama:*

   ```
   prog02.c:16:
   warning: unknown conversion type character 0xa in format
   prog02.c:21:
   warning: control reaches end of non-void function.
   ```

   Verifique que na linha 16 tem % sem o caracter que caracteriza o tipo de dados "conversion type". O outro erro, linha 21, se deve à ausência de um valor a ser devolvido, corrija estes erros. Observe que o programa, mesmo errado, roda. Na maioria das linguagens modernas isto não se dá. Corrija estes erros, (compare com prog02_1.c).
3. *Rode prog02.c, digitando cada um dos nomes em uma linha diferente (separados por "enter").*
4. *Refaça prog02.c para colocar as tres mensagens da entrada de dados num único 'imprima'. Observe que cada mensagem é um parâmetro, veja o último 'imprima' para se inspirar.*

5. *Melhore a* saída de dados *colocando um separador entre cada palavra escrita:*

   ```
   coisa1,'' '',coisa2,'' '',coisa3
   ```

   não se esquecendo de incluir os formatadores %... Veja no exemplo abaixo a solução.

   solução *leia os comentários no programa* `prog02.c`

Depois vamos tornar este programa mais inteligente, deixando que ele mesmo detecte quantas palavras o usuário quer escrever. No momento vamos ser mais imperativos: *Escreva uma frase com tres palavras.*

Exemplo: 5 *prog02_1.c*

```
/* Programa prog02_1.c
Assunto:le uma palavra pelo teclado e a imprime
   por Tarcisio Praciano Pereira - 10 licoes para aprender C
   Sobral, julho de 2000 - UVA
*/
#include <stdio.h>
#include <string.h>
#include "traducao.h"
#include "ambiente.h"


palavra principal()
inicio
   palavra coisa1[30], coisa2[30], coisa3[30]; //(0)
   imprima("%s%s%s\n", "escreva uma frase curta pelo teclado",
  "com tres palavras. ",
  " Pode ser o seu nome, por exemplo "); // (1)
   ler("%s%s%s",coisa1,coisa2,coisa3); //(2)
   imprima("%s %s %s\n ",coisa1,coisa2,coisa3);//(3)
   devolve 0; //(4)
fim

/* Comentarios:
0) Declaracao de variaveis com tamanho adequado para caber nomes.
1) Um unico 'imprima' para as tres frases. Observe que as
     frases podem ser dispostas em tres linhas diferentes, os
  espacos entre os parametros nao tem significado.
2) Ainda usando 'ler' (scanf)
3) Observe os espacos entre os formatadores de dados e veja
  o resultado disto na impressao. Tire os espacos e veja o
resultado.
```

```
4) A ausencia de 'voltar' provoca um erro.
*/
```

2.1.1 Análise do prog02_1.c

Começaremos por discutir os símbolos estranhos $\%s$ etc.. que apareceram nos programas.

Observação: 7 *Formatadores de dados*

- **cabeçalho** *O sinal "/*"marca o início de um comentário que é terminado com o sinal "*/". O programa começa com um comentário que costumamos chamar de "cabeçalho", nele colocamos as informações genéricas sobre o programa. Se, por exemplo, estivermos trabalhando em equipe com colaboradores, eles devem receber programas nossos para alterar, modificar, melhorar. Alguns dos programas até* funcionam, *outros só trazem a idéia daquilo que devem fazer... No cabeçalho colocamos estas informações, não apenas para nós mesmo que escrevemos o programa, como também para os outros que vão trabalhar com os programas.*

  Não duvide, se você for ler um programa uma semana depois que o escreveu, possivelmente não vai mais entendê-lo....

- **comentários** *Os comentários podem ser escritos em diversos lugares dentro dos programas. Se por um lado você deve escrever muitos comentários, também deve ter o cuidado para que eles não causem uma poluição visual que depois atrapalhe a leitura do programa. Guarde a idéia de que um programa deve ser um* texto bonito, *agradável para os olhos e de fácil leitura.*

- **formatação de dados** *O símbolo % informa ao C que se segue uma formatação de saída de dados.*

  - *Se forem frases, (*strings*), então fica: $\%s$.*
  - *Se for um número, depende do tipo de número:*
    - *para números inteiros: $\%d$;*
    - *para números fracionários: $\%f$*

- *O erro, na terceira versão de* `prog02.c`*, consiste em que colocamos poucos "$\%s$" uma vez que os espaços separadores são também* `caracteres`*. O comando que imprime os dados deve ser assim:*

```
imprima("%s%s%s%s%s\n ",coisa1,'' '',coisa2,'' '',coisa3);
```

  se quisermos imprimir tres palavras com espaços entre elas.

  Naturalmente, você deve estar horrorizado! Como ficaria se quisessmos escrever 30 palavras... Se acalme, veremos uma solução mais inteligente depois. Se quiser estudar o assunto agora, veja os programas `texto.c`, `texto01.c`, `texto02.c` `texto03.c`.

Chamamos estes símbolos de formatadores de dados, mas eles tem diversos nomes, porque também têm diversas funções, por exemplo, eles[12] *servem para* "traduzir dados de um tipo para outro".

$\mathcal{C}$ parece ser uma linguagem contraditória. Por um lado relativamente livre, por outro lado contendo restrições de formatação rigorosas. Toda vez que você usar uma *função* de saída de dados, tem que informar a esta *função* que *tipo dados* lhe vão ser passados. Porque, se não o fizer corretamente, $\mathcal{C}$ poderá seguir em frente coletando erros atrás de erros.

Para começar, que é *tipo de dados*? Dedicamos um capítulo a este assunto, veja no índice, e se você quiser pode dar um salto agora para lá, onde esta questão está sendo discutida com mais detalhes. No momento vamos dizer que em computação se distinguem três coisas bem claramente:

- caracteres e palavras;
  - `caracteres`, em princípio, qualquer um dos símbolos que você pode obter apertando uma tecla. Há alguns poucos que não podem ser obtidos desta forma. São caracteres especiais, como o `caracter de fim de linha`.
  - `frases`, ou `aglomerados de caracteres` os `vetores de caracteres`, em inglês, `string`s
- números;
  - número inteiro
  - número fracionário, chamado `real` ou em inglés, `float`.
- vetores;
  - vetores de caracteres (`strings`)
  - vetores de inteiros ou de reais.

Caracteres são qualquer um dos *duzentos e poucos* sinais que você pode produzir com o teclado do computador, como

A, a, / , % ...

existe uma tabela americana chamada *tabela ASCII*[13] que registra umas duas centenas de caracteres que são, no fundo, a base do modo de comunicação escrita que usamos. Esta tabela já foi muito mais importante do que é hoje porque os meios de comunicação evoluiram tanto que hoje já praticamente não mais usamos "caracteres" para nos comunicar. Usamos *cores*... ou mais exatamente `bits`.

Os números são agregados de caracteres tirados da coleção

1,2,3,4,5,6,7,8,9,0, "."

[12] Ver `cast` a este respeito.

[13] `American Standard for Communication and Information Interchange`

que podem ser inteiros, se não usarmos "**.**" e se usarmos o "ponto" representam números fracionários. Esta é uma explicação muito rasteira, mas é mais ou menos a forma como Fibonacci explicou os números decimais no século 11, sem incluir o "ponto". Sem dúvida, seria ótimo que você não ficasse satisfeito com ela e criticasse o autor chamando-o de *superficial...*

Depois podemos combinar estes dois tipos de dados,

`caracteres , números`

para criar *tipos de dados* bem mais complexos. Você pode ver isto a partir do capítulo 5.

Dito isto, o programa começa com `palavra`[14], para indicar que *coisa, coisa1, etc...* são do tipo `caracter`. Em ingles se usam duas palavras para isto, `string, character`. `character` é um caracter, ao passo que `string` é *um vetor de caracteres*, quer dizer um aglomerado de caracteres que pode inclusive conter espaços, (*"espaço" é também um caractere que você gera quando usa a "barra de espaços que nada mais do que uma tecla...*)

Um vetor de caracteres é, por exemplo

"Isto é um vetor de caracteres"

é um conjunto de caracteres delimitado por "aspas". Observe que

`''Isto é um vetor de caracteres''`

é diferente de

`'Isto é um vetor de caracteres'`

A segunda expressão é um erro, porque $\mathcal{C}$ usa `'d`

A primeira linha do programa indica que *coisa1, coisa2, coisa3* são variáveis que devem conter `palavra`s. É uma declaração de tipo de dados.

Observação: 8 *E o que significa* variável *?*

As linguagens de programação são exemplos de linguagens formais. *Quer dizer que elas tentam, e com relativo sucesso, estabelecer uma comunicação entre o homem e a máquina.*

Na verdade entre programadores e aqueles que construiram os compiladores das linguagens...portanto entre homens presentes em frente ao teclado e homens ausentes representados pelo compilador.

Consequentemente elas tem que satisfazer a um conjunto de regras lógicas que vão dar sentido as frases de que se compõem os programas. Da mesma forma como eu não posso me dirigir a você, querendo me referir a uma cadeira, dizer: me dê a mesa. *Você nada vai entender, sobretudo se na nossa frente não houver nenhuma mesa.*

`cadeira` *é uma variável da lingua portuguesa ocupada com um significado bem definido, e naturalmente, imutável. Mas você já ouviu alguém dizer*

esta coisa não serve para escrever

fazendo referência a:

- *um lápis sem ponta;*
- *uma caneta quebrada;*
- *uma velha máquina de escrver.*

coisa é uma variável da lingua portuguesa.

Quer dizer que há palavras livres para assumir distintos valores.

No presente caso temos

- `palavra, imprima, ler` *que são palavras reservadas "*portuguesas"da linguagem C;
- em inglês seriam `char, printf, scanf`;

[14] em inglês seria `char`

- *coisa, coisa1, coisa2, coisa3* que escolhemos para guardar os fonemas que você resolver guardar: *lápis*,*caneta*, *máquina velha* etc...

Examine o arquivo `traducao.h` *onde vai você vai encontrar muitas das palavras reservadas da linguagem C com a respectiva tradução que estou usando nos programas. Aí está o segredo de programar em Português...*

Depois da *declaração de tipos de dados* vieram os comandos `imprime`, `ler`. Estes comandos podem receber uma quantidade indefinida de parâmetros, mas, para cada parâmetro, deve vir indicado o tipo de dado que vai ser lido, é este o significado de $\%s$ para ler ou escrever `palavra`s.

Bom, faltou discutir o que significam as tres primeiras linhas do programa:

`diretiva de compilação`

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

informa ao compilador que ele deve ler a `biblioteca` `stdio.h` que é um arquivo onde se encontram definidas as funções `prinf` e `scanf`[15].

`diretiva de compilação`

A linha

```
#include ''traducao.h''
```

diz ao compilador para ler o arquivo `traducao.h`. A diferença entre aspas ou sinal de desigualdade reside no local onde se encontram os arquivos. Quando o nome se encontra envolto por aspas, isto significa para o compilador que o arquivo se encontra no mesmo diretório em que se está trabalhando com programa, o "diretório corrente". Quando se envolve o nome do arquivo com $<, >$, o compilador sabe que deve procurar o arquivo no diretório padrão em que se encontram todas as bibliotecas da linguagem C.

A linha

```
palavra principal()
```

é o início do programa... e `palavra` indica que a função `principal()` vai produzir uma saída de dados do tipo `palavra`. Isto é, `principal` é do tipo `palavra`.

Todo programa em C tem a estrutura básica dos programas acima.

- Primeiro vêm as diretivas de compilação marcadas pelo símbolo #, como `include`, que significa `incluir`.
- Depois vêm as definições das funções que serão usadas. Discutiremos logo no próximo capítulo o que são funções.
- Depois a vem a `função principal()`, em inglês se chama `main()`,é a função que gerencia o programa.

  Ela é responsável de colocar as coisas para andar, é o maestro que vai comandar o espetáculo.

[15] que traduzimos por `imprima` e `ler`. Se você não gostar destes nomes, use outros...

Antes da palavra `principal` se encontra o tipo de dados que a função vai produzir, na última linha do programa, com a função `voltar, return`. Depois dos parêntesis, que podem conter parâmetros, vem uma *chave-aberta* contendo o algoritmo implementado ao final do qual se *fecha a chave* terminando assim o programa.

- A ordem como as funções vierem dispostas no programa é irrelevante, mas no início deve vir uma lista das funções que vào ser definidas no arquivo e que faz parte integrande do *cabeçalho* do programa. Isto é o que se chama de `protótipos`, é o planejamento do programa.

- Tome como exemplo o programa `integral.c`. Depois dos comentários vem a lista de funções que vão ser definidas mais abaixo:

```
// Declaracao de funcoes ********************
real     Riemann(real ini, real _fim, real deltax);
real     f(real x);
```

- Em C, as chaves servem para definir as unidades lógicas. É o que se chama um `bloco`, uma unidade lógica. Observe que dentro de um *bloco* pode ter outro *bloco*... Aqui estamos usando "`inicio`" e "`fim`", com esta finalidade, como traduções das chaves $\{, \}$.

Os vocábulos que usamos nesta seção foram:

Vocabulário: 4 *bloco lógico, imprima, include, escreva, função, palavra, ler, principal, printf, protótipos, scanf, char, string*

- `bloco lógico` *é uma unidade de programação, conceito difuso que aos poucos você irá dominar. As funções são blocos lógicos. Ao abrir e fechar* chaves *você cria um bloco lógico.*

- `imprima()` *é tradução de* `printf()` *é a função da linguagem C para produzir uma saída de dados. Imprimir no video.*

- `include` *é uma diretiva de compilação para que o compilador veja informações em uma biblioteca. Há várias diretivas de compilação, elas são assim chamadas porque dirigem o compilador para fazer tarefas bastante complicadas antes de criar o programa. Não as discutiremos neste livro.*

- `escreva()` *outra tradução que fizemos de* `printf()`*, para mostrar-lhe que podem ser diversas. Isto poderia ser considerado um defeito, por alguns, um qualidade por outros (a diversidade...)*

- `função` *As funções são as menores unidades lógicas.* `printf(), scanf()` *são funções. Tudo em C é função*[16]

[16] nem tudo, você vai ver depois..., mas quase tudo, digamos.

As funções em C se assemelham um pouco com as funções da Matemática. Depois você vai ver as diferenças, mas agora usemos as semelhanças. Em C definimos uma função f e depois escrevemos $f(a)$; C calcula qual é o resultado de $\underline{f}$ aplicada em $\underline{a}$. Como em Matemática...

Mas frequentemente escrevemos apenas `f();` *o que não se faz em Matemática, porque em C existem funções que não recebem parâmetros.*

- `palavra` *foi uma das traduções que demos para* `char` *que significa caractere, um tipo de dados de C.*
- `ler()` *é tradução de* `scanf()` *é a função da linguagem C para ler dados pelo teclado. Esta função deve ser evitada, usar* `fgets()`*, ver* `prog04_3.c` `prog03_8.c`
- `principal()` *é função* gerente *do programa, obrigatória, em inglês,* `main()`

Observação: 9 *Tradução da linguagem C*

A tradução da linguagem que estamos introduzindo neste livro não é uma brincadeira. Ou a nossa proposta ou a de outro grupo de pessoas um dia virá a ser levada a sério. Faz parte de nossa identidade cultural sabermos programar em nossa lingua. Sem dúvida é uma atitude anti-globalizante *de defesa do desenvolvimento regional.*

Entretanto algum organismo, possivelmente a `SBC`*, deve num certo momento chamar um grupo de pesquisadores e programadores para estabelecer um padrão, porque uma linguagem de programação deve ser padronizada afim de que os programas possam rodar em qualquer lugar e ser entendidos por todos.*