Capítulo 3

Números e Letras

Resumo.
A linguagem *C* não foi feita para trabalhar com números, ela foi feita para trabalhar com caracteres, acessar memória, executar operações aritméticas e operações lógicas[a]. Mesmo assim ela tem uma capacidade numérica limitada em seu formato original. As implementações modernas tornaram esta capacidade bem mais avantajada, porque *C* nasceu dentro do ambiente que hoje podemos caracterizar como de programas livres ou de domínio público, que sempre foi típico dos que programavam em `Unix`[b]. Uma conseqüência disto é que a linguagem *C* cresceu e hoje até poderia ser considerada para processamento numérico porque tem bibliotecas dirigidas para tal.
Neste capítulo vamos explorar um pouco da capacidade numérica da linguagem *C*. O capítulo 8, na verdade, se dedica à Matemática, aqui vamos apenas brincar um pouco com os números. Veremos um pouco de suas limitações, como estas podem ser alteradas, que você tem nas mãos *os meios para produzir estas alterações* e e que você vai aprender como fazê-lo...
Se o seu sistema for LinuX, você tem chances de fazer grandes alterações, mas tenha cuidado, não vá fazer o que você não sabe sem tomar as precauções adequadas. Esta frase não tem o objetivo de amendrontá-lo, nem de inibí-lo.Aprenda *C* a fundo e terá uma ferramenta imponente em suas mãos.

[a] porque *C* foi feita para montar um sistema operacional.

[b] É interessante observar que a própria palavra Unix, é uma marca registrada, o sistema operacional Unix tem um dono. Mesmo assim Unix sempre foi usado com liberalidade. `LinuX` é um clone do Unix, tem dono, Linus Torvalds, mas está colocado sob o GPL

3.1 Brincando com números em C.

Como o título menciona, nesta seção vamos estudar um programa que efetua operações com números.

O programa que lhe vamos propor, `prog03.c`, vem com erros. Algumas correções e *mais outros tantos erros* se encontram na suite de programas

`prog03_X.c`

Estamos absolutamente convencidos que os erros são o instrumento mais profícuo do aprendizado, tantos os nossos como os seus.

Mas, não limite sua imaginação, altere os programas atendendo a todas as possibilidades que lhe vierem à cabeça para "experimentar" outros resultados. Encontre, você mesmo, outras alterações e as teste. Os exercícios propostos são apenas um guia para despertar a sua curiosidade. Nao tema estragar o computador ou o compilador, eles sao muito robustos.

Primeiro compile e rode `prog03.c`:

```
gcc -Wall -oprog prog03.c
digite, prog, para executar o programa.
Possivelmente digite ./prog, se o sistema não estiver
bem instalado, um defeito no path.
```

depois leia o programa para descobrir onde está errado, e o corrija.

Exercícios: 6 *Alterando prog03.c*

As soluções destes exercícios, são os programas `prog03_X.c`.

1. *Altere* `prog03.c` *para escrever a soma dos dois números que lhe forem fornecidos. Veja a solução proposta em prog03_1.c.*

2. *prog03.c imprime os números que você forneceu colados um no outro, feio! Corrija isto. Ver solução em prog03_1.c*

3. *Torne o programa mais "verboso", conversando mais detalhadamente com o usuário, tanto na entrada de dados como na saída de dados.*

4. *Experimente com somas de números cada vez maiores para testar a precisão do sistema que você usa.*

5. *Altere prog03.c para somar números não inteiros. Solução* `prog03_8.c`.

6. *Faça um sistema generoso de mensagens para tornar sua "calculadora" mais atraente, por exemplo, peça os números para somar um a um.*

7. *Altere prog03.c para fazer a multiplicação entre dois números.*

8. *Altere todos os programas da serie prog03*.c substituindo* `ler() (scanf())` *pelo par de funções*

   ```
   leia(), converte_palavra()
   ```

   . Veja `prog03_91.c`. *Você precisa declarar uma variável*

   ```
   palavra deposito[80]
   ```

   para receber dados. O tamanho, "80", porque cabe uma linha.

9. *Existem dois* tipos *de números nas linguagens de programação,* `inteiros` *ou* `reais`*. Do ponto de vista de Matemática isto é uma aberração*[1]*, mas não estamos fazendo Matemática, aqui. Exemplifique o que é número inteiro e o que é número real, em computação.*

   Resposta `prog03_10.c`

10. *** fora do contexto Descreva o que significa dizer-se que um número a está representado na base 8, na base 10 ou na base 16.*

    Resposta `base.c`*, compile este programa assim*

    gcc -Wall -oprog -lm base.c

    a opção −lm instrui o compilador a fazer uso da biblioteca matemática. Experimente omitir esta instrução, o compilador não saberá o que é `pow`*, a função potência.*

Observação: 10 *Comentando os exercícios*

O programa `prog03.c` *sugere que você use números inteiros muito grandes para ver o que pode acontecer. Não sabemos o que "muito grande" representa para você, por exemplo a soma dos custos dos rombos bancários pagos pelo "proer", quer dizer, por nós todos:*

40.000.000.000 dólares !

Se isto já for muito, experimente definir a constante

```
inteiro BANCOS =  40000000000;
```

e a use nas suas alterações de prog03.c. Observe que o programa talvez não suporte este valor[2]*.*

Experimente somar números maiores que "BANCOS". No C que roda em LinuX você precisará experimentar com inteiros maiores que 2 mi para notar alguma coisa extranha. Experimente, portanto. Na versão tradicional do C os inteiros vão de -32768 até 32767. Isto quer dizer que se você pedir para somar

$$n = 32767; m = 1 \quad \Rightarrow \quad n + m = -32768$$

Veja as respostas que obtivemos:

```
A soma de 2147483647 com 1:  -2147483648
A soma de 2147483646 com 1:  2147483647
```

Quer dizer que os inteiros de C não suportam um "BANCOS".

Mesmo assim, sob LinuX os inteiros chegam a casa dos bilhões. Isto nada tem de estranho, nem LinuX é melhor que qualquer outro sistema apenas porque os inteiros em C, sob LinuX, tem um espectro mais amplo. LinuX é melhor porque você pode alterar este espectro, por exemplo. Com números inteiros muito grandes, você pode estar gastando memória à toa, e talvez, para um uso específico seja interessante reduzir a largura deste espectro.

[1] porque os inteiros também são números reais

[2] mas o Brasil ainda continua em pé, apesar deste roubo, e ainda dizem que somos um país pobre...claro, quando pedimos dinheiro para as Universidades, aí o país é pobre... com BANCOS se poderia pagar 8 anos do orçamento minguado das Universidades brasileiras.

Em LinuX você pode fazer facilmente isto, mas não se esqueça que o sistema é multi-usuário e isto pode causar problemas... Observe que, mesmo que você seja o único usuário cadastrado *em uma máquina rodando* `LinuX` *você, ainda assim, não é o único usuário...existe um monte de usuários (do sistema) trabalhando junto com você. Eles podem necessitar (quase certamente necessitam) do C instalado na máquina...*

Aprenda como fazer*, antes de* fazer*. Estudando mais a fundo,* não vai ser ainda aqui*, você irá aprender como alterar a linguagem C para um uso específico O segredo é,* construa uma biblioteca adequada, para não atrapalhar os outros usuários.

Outro: o uso das `diretivas de compilação`*.*

Este programa é muito pequeno para oferecer espaço para muitas mensagens, mas você pode incluir "setas" no local onde o usuário do programa irá digitar números, por exemplo. Se não o tiver feito, faça-o agora.

Observação: 11 *Setas - código ASCII*

Já nos referimos antes à tabela ASCII, consultando-a você pode inserir alguns caracteres diferentes para embelezar seus programas.

Leia a biblioteca `ambiente.h`*. Nela estão definidas as letras acentuadas do alfabeto brasileiro. Se você quiser escrever "corretamente" a frase*

" A acao sistematica do modulo eh "

você deverá[3] *digitar:*

```
imprima(``A a%c%co sistem%ctica do m%cdulo %c'',
cedi,atil,aagu,oagu,eagu);
```

Veja o programa[4] `acentuacao.c` *para uma alternativa de escrita com os sinais diacríticos da lingua portuguesa.*

No lugar de cada `%c` *aparecerá o valor de cada uma das macros* `cedi,atil, aagu, oagu, eagu` *definidas na biblioteca* `ambiente.h`*.*

Se você rodar o programa `ascii.c`*, ele lhe vai imprimir alguns dos valores da tabela ASCII. Nem todos podem ser impressos, um deles é o caracter de fim de linha... outro irá produzir som no "alto-falante" do micro. Também existem variações da tabela ASCII usadas por fabricantes diferentes e o resultado do programa muda de um micro para outro. Consequentemente é difícil ter certeza de que as palavras serão escritas corretamente quando o programa for executado.*

Observe a estrutura da informação abaixo:

```
imprima("A soma de %d com %d eh %d\n",n,m, n+m);
```

A função `imprima` *recebe dois tipos de parâmetros*[5]*:*

- *Um* `vetor de caracteres`*,* `"A soma de %d com %d eh %d\n"`*, na qual se encontram presentes instruções de formatação de dados:* `%d`*, e*

[3] não se assuste, esta é apenas uma forma de resolver o problema

[4] veja também a suite de programas `texto*.c`

[5] tipos, não quantidades

- *os parâmetros que vão preencher o* `vetor de caracteres` *na mesma ordem em que se encontram as instruções de formatação. Se chama a isto de expansão dos formatadores.*

Exercício: 1 *1. Altere o programa* `prog03.c` *para conversar com o usuário pedindo sucessivamente mais números para serem somados num total de cinco parcelas. A função que faz leitura de dados,* `ler()`[6]*, tem uma sintaxe semelhante a da função* `imprima()`*:*

`ler(``%d'',&num)` *em que*

- `''%d''` *é a instrução sobre que tipo de dado será lido.*
- `num` *é a variável que vai guardar o dado.*
- `&` *aponta para o local da memória que estiver associado com a variável* `num`*.Quer dizer que vai ser feita uma atribuição de valores indireta: guardar um dado na memória cujo endereço está sendo referido por* `&num`*.*
- *Posteriormente, nas operações aritméticas, você vai usar* `num` *e não* `&num`*.*

2. *Faça um programa que solicite do usuário cinco números inteiros, e calcule a soma destes números. Ver prog03_2.c*

3. *O perigo de 'ler()' (scanf()) Observamos que é um perigo, mas pode ser usado de forma benigna, para quem já tenha adquirido experiência.*

   Quando rodar `prog03_2.c` *ofereça todos os números de uma só vez, separados por espaços. Veja o que acontece. Rode novamente o programa, porém, forneça os números um por um, separados por* $<$ *enter* $>$*.*

4. *Melhore a versão de* `prog03.c` *usando a função* controladora de fluxo *'*`enquanto()`*'* `while()`*, sintaxe:*

   enquanto (condicao)
   inicio
   ler("%d",numero)
   fim

   faça "condicao" controlar se o usuário fornceu um número. As funções de controle de fluxo são estudadas no capítulo 4, dê um pulo até lá.

5. *Faça um programa que solicite do usuário dois números inteiros, e devolva o produto destes números. Ver o programa* `prog03.c` *A multiplicação em C se faz usando o operador* `*`*.*

[6] `scanf()`

6. *Será que o programa de multiplicação está mesmo fazendo as contas certas? Verifique! Se o resultado for do tamanho de*

   BANCOS,

   provavelmente o programa está errando...

Observação: 12 *Agilidade perigosa do* `scanf()`

Em um dos exercícios acima vimos a agilidade do `scanf()` *que pode ser muito útil, mas que é uma função perigosa. Há autores que dizem "para solucionar os problemas de* `scanf()`*, o melhor método consiste em não usar* `scanf()`*". Preferimos dizer, "deixe o uso de* `scanf()` *para quando você tiver um domínio adequado de todos os seus* efeitos colaterais*".*

Ao ler 5 números digitados, separadamente, com espaços, nos permite que forneçamos todos os números de uma seqüência, de uma única vez. Isto, inclusive, nos permite fazer correções, se tivermos cometido algum erro de digitação. Experimente.

Poristo você deverá aprender a usar 'ler()' (scanf()), mas deve observar que ela oferece riscos. Se você não quiser que ela leia dois valores seguidos, ela ainda assim poderá lê-los. Mas não se esqueça, tudo que o programa fizer é consequência direta do que o programador tiver escrito. Portanto, se você souber usar corretamente a linguagem, seus programas funcionarão corretamente e sem mistérios. Aqui não existem mistérios, pode haver incompetência *apenas.*

Veja os programas[7] *(rode e leia)*

`prog04_2.c, prog04_21.c, prob_scanf.c`

Observação: 13 *Programar bem*

Um dos segredos para ser um bom programador, e isto vale para qualquer linguagem de programação, consiste em ser organizado.

Nunca é pouco insistir que os programas devem ser bem comentados (evitando-se a poluição visual). Um programa deve ser consequência de uma criteriosa modularização e isto vai ficar claro no capítulo 5 do qual você deve fazer agora uma rápida leitura e inclusive usá-lo como um manual complementar na leitura do texto anterior. No capítulo 5 discutimos função, vá até lá rapidamente (e volte logo).

As funções são o método de modularização da linguagem C.

Observação: 14 *Instruções de formatação de dados.*

- *Em Linux Uma lista parcial dos identificadores de tipos de dados em C é:*

tipo de dado	*real*	*inteiro*	*caractere*	*vetor de car.*	*not. cientif.*
	`f`	`d`	`c`	`s`	`e`

Consulte `info libc` *em LinuX para ver uma descrição mais completa, quando sentir necessidade. Procure por* `Formatted Output, Formatted Output Basics` *onde se encontram informações sobre* `printf()`*.*

[7] em `BC` altere o nome do programa `prob_scanf.c` para `prbscanf.c`

O sistema de ajudas dos pacotes computacionais quase que exigem que o usuário já tenha alguma prática. Não desista nas primeira tentativas.

- *Em BC*

  Abra um programa e nele escolha algum comando sobre o qual você deseja informações. Digamos que seja `printf()`*, o noso* `imprima()`*. Escreva na área de textos este comando e coloque o cursor sobre ele.*

  Procure o botão `Help`*. Clicando, cai um menu, escolha* `Topic search` *(busca por tópico).*

  Você será conduzido à família de funções que imprimem:
  cprintf, fprintf, printf, ...

  Passe uma vista. Não tente entender tudo de uma só vez. Volte a ler quando necessário, quando precisar de uma informação específica. Mas, sobre tudo, vá até o final desta página do help, descendo com a seta para baixo*. Desça até o final.*

  Você vai encontrar `Examples`*, os exemplos. Coloque o cursor sobre*
  `printf example`

  dê `enter`*, e você verá um programa completo mostrando como funciona* `printf()`*.*

  Você poderá passar o ratinho sobre o programa, realçá-lo. Vá até até o `Edit` *e escolha* `copy`*. Abra um editor de textos e no editor cole o programa, e grave com o nome que lhe parecer adequado.*

  Você tem um exemplo que funciona. Teste! Alguns exemplos tem excesso de informação. Apague alguma coisa do exemplo e rode e leia. Querendo copie de novo... O `help` *do* `BC` *é um pequeno tutorial sobre a linguagem, você ganhará muito ao usá-lo.*

  O `Help` *do* `BC` *é muito rico, você terá que se acostumar a linguagem concisa e técnica. Mas sempre terá exemplos para rodar e comparar com o que tiver lido. Como dissemos sobre a* ajuda *em* `Linux`*, o sistema de ajuda quase que exige uma prática computacional. É preciso persistir no uso.*

3.1.1 Leitura de dados

Desde o início estivemos alertando o leitor sobre os erros potênciais no uso da ágil e poderosa função `ler()` (`scanf()`) e mesmo assim fizemos uso dela todo o tempo com a desculpa de que ela tornaria os programas mais simples.

É esta razão que pela qual ela é usada com frequência pelos programadores. Apesar dos críticos que chegam a dizer que *a solução para o* `scanf()` *é não usá-la*, ela segue em uso porque é prática.

Vamos mostrar-lhe uma alternativa segura, e ao mesmo tempo prática, ao uso de `scanf()` representada pelo par de funções

- `leia()` (`fgets()`)

- `converte_palavra() (sscanf() )`

Observe que deixamos entre parêntesis as funções originais de $\mathcal{C}$.

Veja os programas `prog03_8.c`,`prog03_10.c`, `prog03_11.c`, com erros, e o `prog03_12.c` em que os erros do anterior se encontram corrigidos.

Este método seguro de leitura de dados consiste no uso seguido das duas funções: `leia()` (`fgets()`) e `converte_palavra()` (`sscanf()`). A sintaxe delas é:

Sintaxe:

```
palavra deposito[80];
leia(deposito, tamanho_de_palavra(deposito), entrada_pdr);
converte_palavra(deposito, "%f", &numero);
```

em que a variável `deposito` deve ser declarada com espaço suficiente para a leitura dos dados.

Depois a função

`converte_palavra()`

irá ao "`deposito`" pegar o que estiver lá contido e *transformar* no tipo de dados que se deseja, observe o formator de dados. No exemplo acima estamos transformando os dados contidos em "`deposito`" em `ponto flutuante` que é um dos tipos de números fracionários em $\mathcal{C}$.

`deposito` é um vetor de caracteres, não se esqueça de que $\mathcal{C}$ tem uma origem humilde e *sabia* apenas mexer com `números` e `caracteres`.

Então iniciamos a entrada de dados colocando num grande *depósito* os caracteres que compõem os dados que nos interessam, depois a função

`converte_palavra`

transforma os dados no tipo adequado.

Observação: 15 *E o prático ?*

Estamos propondo a substituição de uma única função, `scanf()`*, por duas, e dissemos que tudo isto ainda seria prático.*

De fato, como estão as coisas, estamos trocando uma *por* duas*...nada prático.*

Para tornar prático o uso destas duas funções, crie um arquivo

`entrada_dados`

contendo o esqueleto contido no box acima. Este arquivo já existe, ele se chama

`entra_dados.c`

se encontra no disco junto com os demais programas, edite-o agora para ver como foi criado e modifique-o para suas necessidades.

Se habitue a sempre usar a mesma variável `deposito` *para receber dados, o nome é até sugestivo.*

Quando precisar fazer uma entrada de dados, importe para o programa o arquivo `entrada_dados` *e siga programando tranquilamente.*

Não se esqueça de trocar a formação de dados na terceira linha e o nome da variável que vai receber os dados.

Se a variável que for receber os dados for tipo

`ponteiro`

não use o

`redirecionador de endereços`

No exemplo, no box acima, foi preciso usar porque `numero` *não é do tipo* `ponteiro`.

Serão necessários agora dois toques para construir a entrada de dados. Prático, não?

Este método se aplica a programas inteiros. Quando vamos fazer um novo programa, nunca começamos de zero, sempre fazemos uma busca de algum programa que se encontre próximo do nosso objetivo, para isto colocamos palavras-chave nos comentários iniciais dos nossos programas.

Veja o arquivo

`esqueleto.c`

que, no mínimo, é o nosso ponto de partida.

Análise de prog03XX

Vamos olhar de perto a sintaxe de cada uma dessas funções.

Lembre-se que no fragmento de programa acima pode haver mais funções antes de `leia()` e que a declaração de variáveis tem que vir no início do bloco, ver `prog03_10.c`.

Vamos analisar os parâmetros de `leia()` (`fgets()`)

`leia`(deposito, tamanho_de_palavra(deposito), entrada_pdr); tem os seguinte parâmetros:

- `deposito`, é um vetor de caracteres suficientemente grande para receber o que se espera. É sua responsabilidade definir isto tendo em mente não perder espaço nem deixá-lo curto demais. Se o espaço for insuficiente, dados vão se perder e $\mathcal{C}$ segue andando lampeiro. Outras linguagens parariam e emitiriam uma mensagem de erro.

- `tamanho_de_palavra(deposito)` poderia ser considerado uma falta de otimização da linguagem porque o compilador poderia deduzir este número inteiro da declaração de `deposito`. Mas isto tornaria o compilador mais lento.

- `entrada_pdr` informa ao compilador de onde vem os dados. Neste momento estamos lendo os dados via teclado, se você estiver com pressa de ver como se lêem dados de um arquivo em disco, veja os programas da série 20, `prog20_2.c`, por exemplo, em que `entrada_pdr` define um arquivo em disco onde os dados serão gravados e `prog20_3.c` em que se lêem dados contidos num arquivo em disco.

Vamos analisar `converte_palavra(), (sscanf())`:
`converte_palavra`(deposito, "%f", &numero);
Ela exige tres parâmetros:

- `deposito` A variável onde os dados se encontram guardados, no nosso exemplo um vetor de caracteres;
- `o formatador de dados` "%f" que aqui funciona como um *operador para mudança de tipo de dados*;
- `destino definitivo` a variável onde os dados serão guardados, (corrigindo: o endereço da variável onde os dados serão guardados). Ver abaixo.
- observe a possível necessidade de usar o redirecionador de endereço "&". Falamos possível porque se a variável já for do *tipo ponteiro*, o redirecionador de endereço não pode ser usado.

3.2 Brincando com as palavras em C.

Resumo.
Existem dois tipos de dados básicos em C e na maioria das linguagens de processamento de dados: *números* e *caracteres*. Uma *palavra* é uma *seqüência* de *caracteres* entre "aspas", com algumas exceções: há alguns caracteres que tem significado especial e que para serem incluídos nesta definição é preciso algum esforço extra, como o caracter "\"que serve para indicar ao compilador que depois dele vem outro caracter especial. Por exemplo, para fazer uma mudança de linha, como você já viu nos programas anteriores, usamos "$\backslash n$" dentro de uma *mensagem*.
Neste parágrafo vamos aprender a lidar com caracteres e vetores de caracteres.

3.2.1 Palavras, macros, caracteres.

Se é verdade que tudo, dentro do computador é formado de "zeros" e "uns"[8], para nós humanos, o "computador" cria uma facilidade extra e expande este universo estreito com auxílio de uma codificação para *caracteres*, e ainda cria uma partição muito útil:

- Caracteres numéricos: 0,1,2, ..., 9
- Os outros... completando alguma coisa da ordem de $256 = 2^8$ caracteres. O teclado do computador, tem, por exemplo 101 teclas. Mas "a" e "A" são dois caracteres diferentes obtidos apertando ou não "shif" junto com a tecla "a". Com a tecla "ctrl" você pode obter novos caracteres e assim por diante. O que caracteriza um `caracter` é sua escritura entre 'aspas' simples. Assim '3' é um caracter, como 'a' e "3" é um vetor de caracteres.

  Tenha cuidado com estas combinações,

  ctrl-X, alt-X

  alguns destes caracteres de controle podem, simplesmente, travar o programa que você estiver usando...

  `alt`-X parece ser sempre inofensivo, mas não é o caso de `ctrl`-X.

[8] Literalmente falando isto é falso hoje. O que era *zero* hoje é *baixa voltagem*, cerca de 2.5 volts, e o que era *um* hoje é *alta voltagem*, cerca de 5 volts...

- Aglomerados destes dois tipos de dados formam as `macros`, e os `vetores de caracteres`, veja a *observação* a respeito destes dois *tipos de dados*. De forma simplificada, o que estiver entre "aspas" é `vetor de caracteres`.

Alguns destes caracteres têm *efeito especial*, uns você pode usar diretamente, outros ficam escondidos para uso interno do sistema operacional mas podem ser acessados com alguma "sabedoria" desde que isto seja feito "*sabendo-se o que se está fazendo*". Você já sabe que "*ctrl alt del*" serve para desligar a máquina, por exemplo...

Exercícios: 7 *Verificando os caracteres*

Os exercícios desta seção usam conceitos que somente vamos estudar no capítulo 4, salte até este capítulo quando sentir necessidade, ou ignore os conceitos ainda não estudados. Basicamente as funções `para(), se()` *e o conceito de* `laço` *é que precisamos do capítulo 4, aqui.*

1. *Leia a biblioteca* `ambiente.h` *e analise o programa* `ascii_2.c`*, (rode e leia o programa) para ver como se pode acentuar. Logo no começo de* `ambiente.h` *se encontram as macros para definir as nossas letras acentuadas.*

2. *Leia o programa* `ascii.c` *e procure entender o comando de impressão. Talvez você deva rodar o programa e depois voltar a lê-lo. Ignore por enquanto a função* `para`*, ela vai ser estudada no próximo capítulo, ela é responsável pela "evolução" da variável* n *desde* 0 *até 255. Este trecho do programa se chama um* laço*.*

3. *Claro, você não conseguiu ver nada do que o programa imprimiu... 256 linha rodaram em frente aos seus olhos. Solução: faça uma restrição dentro do* `para` *de modo que o programa imprima umas 10 linhas, (ver* `ascii_1.c`*).*

4. *Melhore o programa* `ascii_2.c` *a seu gosto, incluindo mensagens na saída de dados e na entrada de dados. Use muitas palavras acentuadas para aprender a passar parâmetros para "*`imprima()`*".*

Veja que existe uma *complicação*[9]extra. Para nos comunicarmos com a máquina precisamos de uma linguagem, como *C*, ou mesmo antes das linguagens, se encontram os sistemas operacionais, *LinuX*, FreeBSD, DOS etc... Ora as linguagens são feitas de *palavras*[10], quer dizer, "aglomerados" de caracteres aos quais se fez uma associação com uma rotina da linguagem: é isto que você usa quando digita *dir* e *enter* na linha de comandos. O mesmo se dá quando você executa a linha

```
gcc -v numero.c -onumero
```

[9]sem preconceitos, as *complicações* são necesárias, uma vida *simples* é insuportável...
[10]veja observação a respeito

criando uma nova *palavra*, `numero` que agora o sistema operacional reconhece como incluida no seu vocubulário e associada a uma rotina gravada no HD, provavelmente em sua área de trabalho.

Uma "nova"especificação se vem estabelecendo, um modismo. Palavras deste tipo que acabamos de descrever passam a ser chamadas de "macros". As `macros` servem para definir *variáveis*, *nomes de comandos* e *funções*. A palavra `macro` é antiga, passou para o esquecimento, e agora volta a ser usada com frequência como um sinônimo de `variável`.

Vamos usar macros com mais freqüência a partir do capítulo 5 quando começaremos a criar funções. Agora vamos trabalhar com `vetores de caracteres`.

Vocabulário: 5 caracter, palavra, macro, vetor de caracteres, variável, símbolo, identificador

Nós usamos palavra *num sentido pouco usual em computação, e foi proposital.*

Aqui existe uma ambiguidade, "palavra" é usada também como medida da informação junto com bit, byte, *apenas mal definida, existem palavras de 16 bits e de 32 bytes. A idéia era que*

- *um* bit, *seria a menor medida da informação,*
- *um aglomerado de 8 bits formaria um* byte. *Isto pode ser pensado fisicamente como um integrado contendo 8 diódos que possam energisados criando um total de* 2^8 *possibilidades;*
- *e 16 bytes formaria uma* word, *que é* palavra *em inglês.*

Mas a prepotência de certas firmas computacionais anarquizou esta convenção porque lançaram no mercado máquinas, ou "sistemas", arrogantemente dizendo, que a word *era de 32 bytes...*

Uma outra ambiguidade persiste, nas linguagens humanas se tem o conceito de palavra *reservado para um aglomerado de letras e possivelmente números, como "Pentium III", para representar os objetos ou as idéias que nos cercam e com os quais nos comunicamos formando as frases de nossa linguagem.*

Ainda existe o conceito de macro *em computação, também difuso quanto à sua concepção.*

Você pode encontrar o uso de macro, identificador, nome, símbolo *usados como sinônimos.*

- **caracter** *qualquer um dos símbolos que é possivel acionando uma tecla (ou um conjunto de teclas de uma só vez). Como o famoso* `ctrl-alt-del`. *O programa* `ascii.c` *lhe mostra o que são caracteres, leia e rode o programa.*
- **identificador** *é usado para fazer referência ao nome de uma variável e também de funções.*
- **macro** *é usada com freqüência para representar o nome de uma rotina, de um programa.*
- **símbolo** *é usado como sinônimo de* identificador, *mas pode ser encontrado com como sinônimo de* variável.
- **variável** *tem uma concepção mais estável, tem o significado usado em Matemática.*
- **Word** *Uma medida da informação. Ambígua, ora signifca 16 bytes, outras vezes 32 bytes.*

- **vetor de caracteres** *Um aglomerado de caracteres enfeixados entre aspas. Alguns dos caracteres de controle não podem ser incluidos porque eles tem uso especial para comunicação com o sistema operacional, como "\n" que significa "nova linha".*

Leia o arquivo `traducao.h` e verá alí um conjunto de macros. As que se encontram a esquerda são as nossas que tem por definição aquelas que se se encontram à esquerda, usando a macro `define` do $\mathcal{C}$.

Quando você escrever

```
inteiro numero;
```

você informa ao C que está criando uma nova macro, designada pelo aglomerado de letras $\{n, u, m, e, r, o\}$, classificada como `variável`, que deverá ser associada a um endereço de memória e que deve guardar um número inteiro.

Mas quando você escrever "numero", o compilador $\mathcal{C}$ vai identificar este objeto como um vetor de caracteres e ignorar o que se encontra entre as aspas mas fazendo-lhes uma associação de endereços sucessivos de memória se você tiver definido tudo direitinho.... é isto que se chama um *vetor de caracteres* .

Há algumas regras para definir

```
macros, variáveis, símbolos, identificadores
```

,

- a primeira é que elas não devem estar entre "aspas", tudo que estiver entre aspas é um `vetor de caracteres`;

- depois elas não devem começar com um número;

- finalmente os caracteres

  ```
  +  - / * $ % # @ ! ^ \
  ```

  devem ser evitados e algumas vezes são proibidos.

O caracter `-` não é em geral proibido, mas é um mau hábito usá-lo, porque se confunde com a subtração.

O caracter `_` também não é proibido, mas tem em um significado especial que você ainda verá no momento certo, e deve também ser evitado como primeiro elemento ou último.

Em C as dois identificadores `Abracadabra` e `abracadabra` são diferentes, quer dizer que a linguagem é sensível à diferença entre letra maiúscula e minúscula.

Vamos seguir, entretanto, o hábito linguístico, leia "jargão", que prefere a palavra "identificador" à palavra "macro".

Um `identificador` será uma `macro` para nós apenas quando receber uma definição(quando houver um algoritimo associado ao identificador).

3.2.2 Vetores de caracteres.

Usamos `vetores de caracteres` para

- mensagens construir as mensagens de um programa.
- receber dados Ver a variável `deposito` que usamos em conjunto com `leia(),converte_palavra()` . Veja o arquivo `entrada_dados`.

Há programas que se especilizam apenas em manipular mensagens, por exemplo:

1. Um programa que gerencie um pacote computacional, basicamente irá emitir mensagens para informar ao usuário quais são as possibilidades do programa, exibir o *menu*.
2. Um módulo de pesquisa de assuntos numa página da Internet ou num banco de dados especializado em "assuntos", irá receber *palavras* que o usuário queira fornecer para fazer uma busca dentro do programa site ou nos sites que tenham ligações com este em que o usuário estiver conectado.
3. Você pode estar interessado em construir um programa que cadastre senhas. Uma senha seria um *vetor de caracteres*. Observe que senhas não são `macros`. As `macros` devem receber dados ou conter dados.

Originalmente, em $\mathcal{C}$, não havia este tipo de dados, ele foi adicionado posteriormente com a criação da *biblioteca* `string.h`, da qual fazem parte as seguintes funções[11]:

Vocabulário: 6 *Algumas funções definidas em* `string.h` `copia_de_palavra (strcpy()) ;concatena_palavras() (strcat()) ;tamanho_de_palavra() (strlen()) ;compara_palavras() (strcmp()); compara()`

- `copia_de_palavra(), strcpy()` *Sintaxe:*
  copia_de_palavra(var, "palavra");

  Se `var` *tiver sido definida com o tamanho adequado irá receber "palavra".*

  Observe a diferença entre a atribuição numérica e esta entre variáveis do tipo ponteiro. Para números vale

  ```
  x = 3;
  ```

  se `x` *for uma variável de tipo numérico. Você terá feito a atribuição do valor* 3 *à variável* `x`. *Para* cadéias de caracteres *não é assim que se faz atribuição. Temos que usar* `copia_de_palavras()` *ou (*`strcpy()`*).*

- `concatena_palavras(), strcat()` *Sintaxe:*

  ```
  concatena_palavras(primeiro,segundo);
  ```

[11] leia comandos, se quiser

Se a variável `primeiro` *contiver a string "José" e a variável* `segundo` *contiver a string "Maria" então depois da execução de*

```
concatena_palavras(primeiro,segundo);
```

a variável `primeiro` *conterá o valor "JoseMaria". Claro, se você desejar um espaço entre os dois nomes, você deverá executar primeiro:*

```
concatena_palavras(primeiro,'' '');
```

veja abaixo.

Esta função serve para inicializar variáveis também. O trecho de programa seguinte fará com que `primeiro` *contenha "Jose Maria":*

```
palavra primeiro[30];
concatena_palavras(primeiro,''Jose'');
concatena_palavras(primeiro,'' '');
concatena_palavras(primeiro,''Maria'');
```

Ver `prog04_1.c`.

- `tamanho_de_palavra()`, `strlen()` *Sintaxe:*

  `tamanho_de_palavra`*(var)*

  Se a variável `var` *contiver uma string definidas por uma seqüência de n caracteres, será o inteiro positivo n a resposta de*

  `tamanho_de_palavra`*(var)* $-> n \in \mathbf{N}$

- `compara_palavras()`, `strcmp()` *Sintaxe:* `compara_palavras(var1,var2);`

  com duas variáveis do tipo `string`. *Esta função da linguagem* $\mathcal{C}$ *tem um uma sintaxe perversa. O resultado de*

  `compara_palavras(primeiro,segundo)`

  será zero se os `vetores de caracteres` *contidos nas variáveis* `primeiro`, `segundo` *forem iguais. Veja o seguinte pedaço do programa prog04_2.c*

  palavra primeiro[30],segundo[50];

  imprima("A primeira palavra − − − >");

  leia(primeiro, tamanho_do(primeiro), entrada_pdr);

  imprima("A segunda palavra − − − >");

  leia(segundo, tamanho_do(segundo), entrada_pdr);

  imprima("% d\ n", "O resultado da comparação é:",

  compara_palavras(primeiro,segundo));

  Vai resultar em zero se as variáveis `primeiro` *e* `segundo` *contiverem a mesma string, apesar da definição diferente das variáveis.*

- `compara()` *definida em* `ambiente` *corrige a perversão de* `compara_palavras()` `(strcmp())`.

  `compara(primeiro,segundo)` *será zero somente se o conteúdo de primeiro e segundo forem diferentes.*

Aqui podemos aproveitar para sugerir-lhe um método que todo programador experiente:

Observação: 16 *Reutilização*

Todo programador profissional, quando vai escrever um novo programa, começa por examinar entre os seus programas um parecido com aquele que deseja produzir.

Afinal, porque começar tudo do nada!

Se tiver que comparar palavras, vá buscar em alguma biblioteca *programas que comparem palavras algum apropriado porque assim você vai se relembrar qual é sintaxe de* `compara_palavras()` *e sobretudo como foi que fez uso desta função a última vez que precisou. Em geral apagamos quase todo o programa para aproveitar quase que apenas o* `esqueleto`*...*

É isto que se chama reutilização *de programas.*

Deveriamos usar a palavra reciclagem*, embora os americanos venham usando a palavra* reutilização*.*

A reciclagem *de programas que já foram testados e que ainda funcionam bem, economiza tempo de trabalho.*

Cabe relembrar o presença dos comentários *nos programas... eles podem criar condições para reciclagens eficientes de antigos programas.*

Os nossos programas todos foram feitos com este objetivo em vista, inclusive todos os mais recentes tem uma linha

```
Palavras chave
```

para faciliar a busca em LinuX. Se eu quiser procurar um programa que acesse arquivos, na linha de comandos, executo:

```
grep arquivo *.c
```

e vou ter uma lista de programas que mexem com arquivos.

Leia a nossa biblioteca `ambiente.h` *onde está definida a função* `compara()` *que inverte a lógica perversa de* `strcmp()`*. Veja como é fácil corrigir aquilo que não nos agrade na linguagem* $\mathcal{C}$*.*

Quando precisamos de comparar a igualdade *entre palavras, usamos* `compara()` *em vez de* `strcmp()`*.*

Observação: 17 *Utilização correta de* `strcmp()`

Mas tem lógica em `compara_palavras()` `strcmp()`*.*

Veja o seguinte trecho de programa

`primeiro = ''aaa'';`	
segundo = "bbb"	
terceiro = "ccc"	
pri = strcmp(segundo, primeiro)	*"aaa", "bbb"*
seg = strcmp(terceiro, segundo)	*"bbb", "ccc"*
ter = strcmp(primeiro, terceiro)	*"ccc", "aaa"*
imprima("%d",pri);	*1*
imprima("%d",seg);	*1*
imprima("%d",ter);	*-1*

Nas três últimas linhas você tem o valor das variáveis

`pri,seg,ter`

que guardam o resultado das comparações feitas.

Observe a ordem como as comparações são feitas por

`strcmp(maior, menor)`

para que o resultado seja positivo.

Quando você usar com este objetivo, em um programa, deixe um comentário dizendo que é este o uso.

Exercícios: 8 *Vetores de caracteres*

1. *No programa* `prog04_1.c` *a variável* `primeiro` *é definida com um valor inicial, um vetor de caracteres vazio. Se não for assim* C *vai colocar lixo não reciclável em seu interior. Experimente modificar o programa eliminando a inicialização da variável* `primeiro` *e analise o resultado.*

2. *Explique (tente explicar!) onde, exatamente, e por que, no programa* `prog04_1.c`*, o compilador enfia lixo se a variável não for inicializada na declaração.*

3. *Rode o programa* `prog04_2.c` *e analise a diferença entre*

   `tamanho_de_palavra(), strlen()`

   e

   `tamanho_da(), sizeof().`

   Uma, conta o número de caracteres do conteúdo da variável, a outra, diz qual é tamanho (tamanho planejado) da variável.

4. *Análise, ao rodar o programa* `prog04_2.c` *um defeito: pede que você aperte uma tecla para continuar, mas não espera... Responsável:* `ler()` *ou* `scanf()`*. Veja a solução do problema em* `prog04_21.c`*. Leia o comentário também.*

5. *O programa* `prog04_3.c` *compara duas variáveis para determinar se são iguais. Verifique que isto é independente da definição das variáveis, claro, desde que sejam do mesmo tipo.*

6. *Corrija* `prog01.c` *para que ele leia o seu nome completamente, (se ainda não o tiver feito...)*

7. *Leia e rode* `prog04_5.c` *, ele é um* tutorial *sobre o uso de* `strcmp()`, `compara()`. *Você vai ver que* `strcmp()` *é uma importante função para ficar em* background[12]*, como fizemos em* `compara()`.

[12]você está vendo aqui uma forma importante de usar $\mathcal{C}$, construindo outras ferramentas com as poderosas funções da linguagem.