Parte II

Aprofundando os conhecimentos

Você deve ter *visitado* a maioria dos capítulos desta segunda parte por sugestão feitas no texto anterior.

Todos os capítulos desta segunda parte foram escritos como complemento ao texto inicial. Rigorosamente falando, ao terminar a primeira parte, se você tiver feito todos os exercícios, analisados todos os programas sugeridos, o volume de trabalho feito foi grande e você já sabe programar em $\mathcal{C}$.

Se você tiver feito isto, parabens. Certamente, para você, a leitura dos capítulos desta segunda parte devem ser uma segunda leitura. Leia agora com cuidado procurando compreender todos os detalhes. O objetivo, agora, é conduzí-lo à construção de bibliotecas, domínio de técnicas mais avançadas da linguagem e prepará-lo para a produzir sistemas de maior porte. Você encontrará aqui varios esboços de sistemas, *inacabados*, mas você deverá conseguir terminar aqueles que se sintonizarem com seus interesses.

Uma regra importante para aprender qualquer coisa, inclusive programar, se dedique a um *problema* para o qual você esteja motivado. Se

- você for professor, *que tal usar a linguagem para construir um sistema de controle acadêmico para os seus cursos* ?

- se você tem uma grande coleção de músicas, de filmes, de livros, *que tal construir um sistema de controle da sua coleção* ?

- se algum parente ou amigo tem um comêrcio, *que tal fazer o sistema de contabilidade e estoques deste comêrcio* ?

Voê irá encontrar dicas para qualquer um desses projetos entre os exercícios ou programas que lhe apresentaremos. Escolha um *objetivo* e se concentre nele. Use o livro para ir em busca do que você precisar para produzir o seu projeto. Você vai encontrar aqui pelo menos as indicações de onde encontrar o que você poderá precisar para atingir o seu objetivo.

De agora em diante ignore a frase comum "*estes capítulos foram pensados para uma primeira leitura*". Esta frase valia apenas enquanto você se encontrava ainda lendo a primeira parte...

Se você quiser continuar usando os comandos em Português você sabe como fazer: usar e expandir o arquivo `traducao.h` e seguir programando em Português.

Nós o encorajariamos a fazê-lo, é a forma mínima, mais elementar de aprender a escrever outra linguagem de programação... e é tentando modificar que se pode aprender a fazer coisas novas. Entretanto este é um livro sobre a linguagem $\mathcal{C}$ e não seria correto insistir nos programas em Português e manter este título para o livro. Mas você pode tomar o rumo que quiser, e deve.

O objetivo do livro nesta segunda parte é discutir com mais profundidade as estruturas de dados, as estruturas de controle de fluxo, e estudar algumas das bibliotecas padrão do $\mathcal{C}$, enfim, deixá-lo em condição de mergulhar na linguagem em profundidade ou escolher outra linguagem mais propícia para o seu trabalho porque você já deve ser um programador.

Você está entrando na segunda metade do livro.

Como obter informações.

Em LinuX existe um sistema de manuais que podem ser acessados de diversas maneiras. Uma delas consiste em digitar `info`. A primeira vez que você for usar este sistema, digite

info info

para ter ler algumas informações iniciais de como funciona este sistema. Depois apanhe um pouco até descobrir a imensidão de informações que ele guarda. Não desisita ante as primeiras dificuldades.

Você sai do `info` digitando `q` e percorre suas página apertanto a barra de espaços. Pode ser a seta para baixo ou para cima ou `Page up Page down`.

Outro método que se considera em extinção é

`man`

Este segundo sistema é um tradicional sistema de informações de máquinas `Unix` que o `info` deve susbstituir aos poucos em `LinuX`.

Digitando `man man` você vai poder ler algumas páginas sobre este sistema de informações.

Você sai do `man` digitando `q` e percorre suas página apertanto a barra de espaços, como no `info`.

Digitando

`man gcc ; info gcc`

você vai poder ler uma série de páginas com inormação técnica sobre o compilador `gcc`.

Digitando

`info glib`

você vai poder ler as páginas sobre o sistema de bibliotecas do `gcc`.

Se você aprender a usar "emacs" você estará dentro de um poderoso editor de textos, (entre outras coisas) e nele poderá escolher, com o mouse, a opção "Info" que irá colocar na tela as opções do sistema de informações (manuais, tutoriais etc...). O comando "m" abre um diálogo no pé da página onde você pode digitar "Libc" e aí você se encontra no "manual de referência do gcc". ver "info" no índice remissivo ao final do livro

Outra forma de buscar informações, em LinuX, consiste em digitar "info nome_palavra". Por exemplo "info scanf" irá chamar a página do Manual do LinuX em que você poderá ler a sintaxe de uso do "scanf()". ver "info" no índice remissivo ao final do livro

O índice remissivo alfabético

Este livro tem um índice remissivo alfabético que consideramos uma das partes mais importantes do texto. Ele se propõe a fornecer informações e atalhos para que você consiga rapidamente elaborar um pequeno programa. Uma outra função que ele deve ter é a de permitir que você recupere uma informação que ficou vaga. Inclusive o *índice* aponta para outras fontes de consulta. O trabalho na construção de um índice remissivo é quase o mesmo de escrever um livro, nos justificamos assim, se ele não conseguir atender às suas expectativas em toda sua extensão, mas neste caso reclame, bata duro nos autores, e claro, traga suas sugestões também, elas serão sempre benvindas para a próxima edição.

Procure no índice remissivo "informação"para ver onde pode procurar mais informações.

Capítulo 6

Variável global e local

> As linguagens "modernas" de programação[a] tem um conceito especial de *variável local* que se opõe ao conceito de *variável global*.
> Os exemplos vão lhe deixar claro o que significam estes conceitos, e vamos partir da análise de exemplos para construir o conceito.
> As primeiras secções fazem análise do assunto e levantam a poeira, na última fixamos os detalhes, mas sugerimos que a última seja lida numa segunda leitura.
>
> [a] Definição de "moderno"? é aquilo que usamos e atualizamos hoje...

6.1 Variável global e local

Sem tentar definir logo no início, deixe-nos apenas dizer que *antes* todas as variáveis eram *globais*, quer dizer, podiam ser vistas, (pior, poderiam exercer influência) em qualquer parte do programa.

Com a modularização, as variáveis podem agora pertencer ao *interior* de um módulo de programação, e $\mathcal{C}$ joga forte com este conceito:

> *Quando você abrir um par de chaves, criou um módulo e, pode, no começo deste novo bloco lógico, criar variáveis que serão destruidas fora das chaves, (quando este bloco lógico estiver fora de uso).*

Não se trata apenas de *existência* e *destruição*, que são as idéias centrais da frase anterior. Claro isto faz parte do próprio conceito de variável local. Se trata de um mecanismo para evitar que dados fiquem sem controle dentro de um sistema.

Esta "localização" protege o programa contra a existência de variáveis que fiquem zanzando, quais "zumbis", perdidas dentro do programa...

Em suma, mesmo antes de entrarmos no assunto, é preciso convencê-lo de que as variáveis globais podem representar riscos muito grandes e devem ser evitadas

a todo custo. Já dissemos antes, estamos repetindo, quando você precisar definir uma variável global, deixe um comentário a respeito no cabeçalho do programa e ao lado da definição da variável

```
int numero; // variavel global !!!
```

que lhe sirva de alerta para que, se possível, alterar o *status* de variável global.

A figura (fig. 6.1), página 118 mostra a relação entre uma *variável local*, uma *variável global* definidas em um módulo e *um submódulo* do programa.

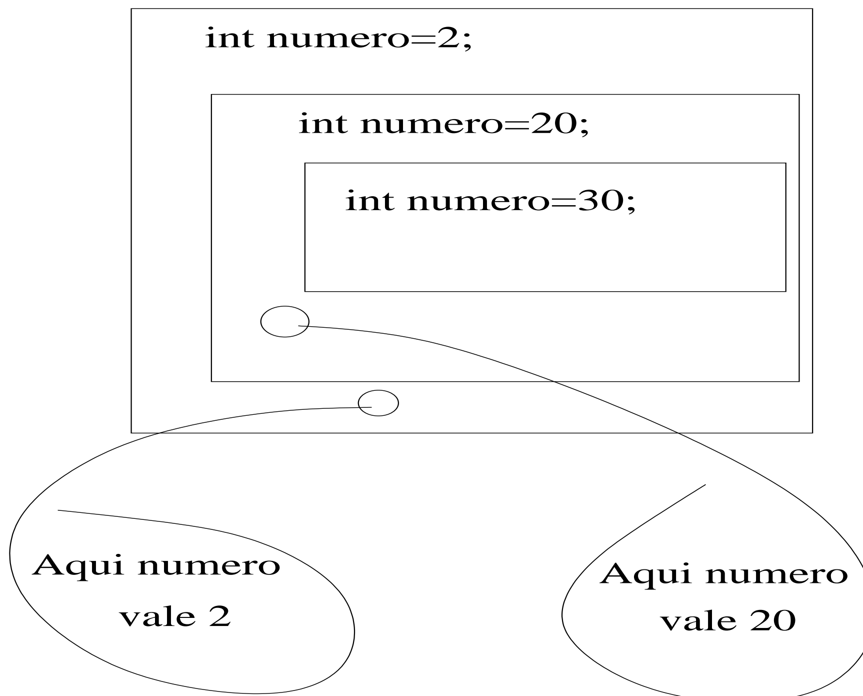

Figura 6.1: Variável global e variável local.

Na figura (fig. 6.1) você pode identificar tres regiões,

- O ambiente do programa todo, designado como *externo*;
- um módulo;
- um submódulo.

O módulo interno é chamado pelo externo, está é uma situação comum, e pode ser uma única "linha" de programa, um `loop`, por exemplo.

Há diversos fatos que podemos salientar com respeito à variável `numero`:

- `numero` no bloco interno, nada tem o que ver com `numero` no bloco intermediário. As duas podem até ter `tipos` diferentes, o que não seria nada aconselhável, neste caso deveriam ter identificadores diferentes.
- O valor que `numero` tem no bloco intermediário, consequentemente, é diferente do que a outra tem no bloco interno.

- Há razões fortes para que você tenha variáveis tão distintas, mas com o mesmo nome. Se você estiver calculando uma soma, gostará, certamente de chamar sempre a variável que acumula os valores de `soma`.

- É preciso ser cuidadoso com esta facilidade para não pecar contra a legibilidade do programa, a sua compreensão. Comentários sempre ajudam. Sufixos permitem que você use o nome adequado adaptado ao módulo em que estiver a variável, por exemplo `soma_interna`, `soma_externa`.

Resumindo, você pode fazer o que a figura (fig. 6.1) sugere, mas não deve. Inclusive o compilador irá advertí-lo dizendo que o valor de `numero` ensobreia[1] o valor de `numero`, quando analisar o sub-módulo. Embora as três variáveis `numero` sejam três variáveis diferentes, evite de fazer isto. Use

`numero1, numero2, numero3`

ou mesmo `nomes` mais sugestivos

`numero_de_fora, numero_do_meio, numero_de_dentro`

nunca esquecendo que a regra é que o programa fique legível. Se `numero` representa a quantidade de grãos de feijão que o programa está contabilizando, porque não chamar esta variável de

`numero_grao_feijao ?`

Veja o seguinte exemplo que podemos etiquetar como grotesco, leia e rode o programa `grotesco.c`.

O programa `grotesco.c` ilustra inclusive os riscos que fazem da linguagem $\mathcal{C}$ um ambiente de programação delicioso...

Rode o programa `grotesco.c` e depois o leia. Faça isto diversas vezes até que fique claro o significado de variável *local* e *global*.

Exercícios: 27 *Tres variáveis com o mesmo nome*

Considere o programa, no disco, chamado `global_01.c`. *Ele mostra como se pode usar de forma abusiva a definição de variáveis locais que é um fato positivo nas linguagens modernas, mas que pode se voltar contra o programador.*

1. *Leia* `global_01.c` *e justifique os comentários (41), (42), (43)*

2. *Altere a variável, escolha os nomes, de modo que o programa saia do loop.*

O exemplo, no exercício anterior é uma representação perfeita da figura (fig. 6.1) em que uma variável designada pelo *nome* `numero` existe em tres ambientes distintos.

Neste exemplo temos três ambientes dentro do macro ambiente representado pelo programa, quero me referir ao arquivo onde está esta função e o cabeçalho do programa. Veja `global_01.c`, a variável `numero` está definida em tres ambientes. Rigorosamente falando o singular está errado, são tres variáveis, definidas em tres *espaços de nomes*, com endereços diferentes na memória.

[1] shadows

Dentro do `while()` a variável tem um valor que se encontra em permanente alteração e **nada tem o que ver com a variável que controla o laço**. Consequentemente este programa não para nunca a não ser com `Ctrl-C`. Rode e veja os valores se alterando **dentro** do laço.

Observe outro fato, no bloco interno a variável `numero` não foi incializada e o compilador lhe atribui um valor sobre o qual o programador não tem controle.

Exercícios: 28 *Variável global e local*

1. *Modifique o programa* `global_01.c` *para que ele pare sozinho, acrecentando uma variável* `contador` *para controlar o* `while()`*.*

2. *Altere* `global_02.c` *para que você possa ver as mensagens fora do laço e verificar assim que as variáveis* `numero` *são diferentes. Solução* `global_03.c`

3. *Por que o 9 é impresso seguidamente por* `global_03.c` *?*

6.1.1 Comentários sobre os exercícios

Em `global_01.c`, no ínicio do ambiente interno, definimos a variável local numero, que se encontra representada pelo mesmo nome que a *variável global* definida no ambiente externo.

Para $\mathcal{C}$ se tratam de duas variáveis diferentes. Quando o compilador inicia o processamento do ambiente interno, cria um novo `espaço de nomes` onde registra a nova variável `numero` que acontece ser designada pelo mesmo nome que a variável definida anteriormente.

Até este momento nenhum problema. A coisa até pode ser assim mesmo. Podemos e até devemos usar o mesmo nome para variáveis que venham a ter a mesma função em um módulo interno de um programa. Isto pode tornar as coisas menos claras, mas tem suas razões, e um comentário resolve este problema.

Esta é uma *elipse* da arte de programar.

Mas não podemos *considerar* esta maneira de fazer como um *método de bem programar*. A maneira de *programar bem* sugere que no bloco interno se acrescente um sufixo ao identificador da variável.

Muito melhor do que esta confusão provocada com o uso do mesmo identificador, seria usar uma pequena variante do nome:

`numero, numero_local, numeroL`

por exemplo. Com a nova variante fica fácil de identificar "quem faz o que" e se guarda o *espirito* do nome: `soma, numero, contador`. Uma variável que deve contar, tem que ser chamada de `contador`, mas uma variável local que for contar pode ser chamada de `contador_local`.

Afinal, as mesmas facilidades que temos nos permitem

- construir um número grande de variáveis;

- com nomes parecidos, mas indicadores da funcionalidade da variável;
- diferenciados por sufixos que indiquem em espaço de nomes as variáveis tem seus endereços.

O objetivo aqui, entretanto, é discutir o conceito de variável local em oposição à variável global.

O exemplo, embora grotesco, ilustra o fato. Ao entrar em um bloco interno, $\mathcal{C}$ permite que novas variáveis sejam criadas. As variáveis criadas em um bloco interno são locais, como dissemos acima, novo espaço de nomes é criado e nova referência é assim feita entre estas variáveis e seus valores.

Assim que o programa abandona o bloco, as variáveis alí criadas são destruidas. Deixam de existir.

> Referências a "nomes" iguais, como no programa acima, vão ter valores diferentes dentro do bloco lógico, onde vale o valor local, ou fora do bloco lógico onde vale o valor global. É o que você pode ver ao rodar `global_01.c`.

Quando você abrir uma chave, num programa em $\mathcal{C}$, uma novo espaço de nomes será criado. Para definir novas variáveis neste bloco, isto tem que acontecer logo no ínicio do bloco, antes de qualquer comando do bloco. Outro local é ilegal e produzirá algum tipo de erro, (nem sempre o mesmo), inclusive `gcc` não lhe vai alertar que as variáveis foram definidas em local errado, ele simplesmente vai se perder apresentando outros tipos de erro.

> **Aliás, adquira um hábito. Sempre que for abrir uma nova chave, se pergunte se não deveria definir uma nova função...**

Claro, o conceito `global` é relativo.

Imagine tres blocos de programa, um inserido no seguinte. A figura (fig. 6.2) página 122, mostra isto.

- Variáveis definidas no bloco médio serão globais relativamente ao bloco "mais interno",
- e locais relativamente ao bloco "mais externo".

Ainda tomando como referência a figura (fig. 6.2), uma variável definida no bloco lógico **A** é visível nos blocos lógicos **B,C,D** a *não ser que haja outra variável com mesmo nome* em alguma destas regiões interiores. Se houver uma variável com o mesmo nome, definida num bloco interno, os valores desta *nova variável* vão se sobrepor ao da variável externa, dentro do bloco.

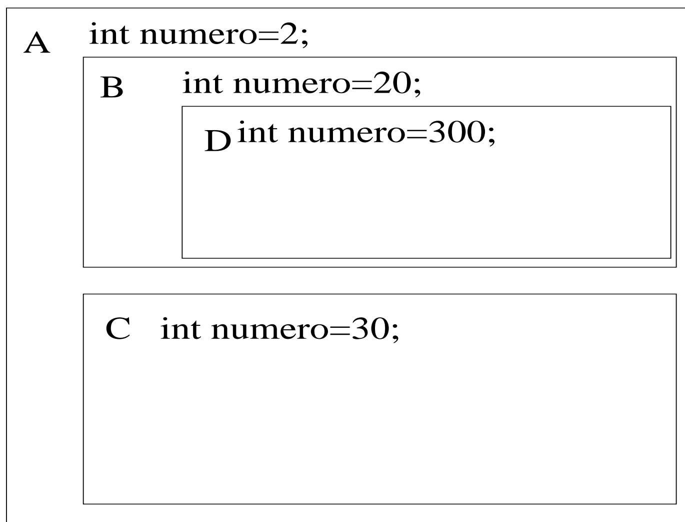

Figura 6.2: Variável global e local

Sublinhando a última frase. Uma variável `numero` definida em **A** será sobreposta por outra, de nome `numero` definida em **B**. O programa `global_02.c` ilustra isto muito bem.

Uma variável definida na bloco lógico **D** não existe nos blocos lógicos **A,B,C**. Ela será destruida quando o programa sair do bloco **D**.

Um exemplo desta situação pode ser vista no programa `global_04.c`. Rode o programa, e o leia para verificar que confere.

A situação descrita no exemplo acima, de variáveis com exatamente o mesmo nome, em módulos encaixados, deve ser evitada. Não conseguimos imaginar nenhuma situação interessante em que este exemplo possa ser usado, *sem riscos*.

A única utilidade deste exemplo, e você deve rodar programa `global_01.c` para ver o que acontece, é alertá-lo para não fazer esta bobagem.

Mas importante do que discutir o exemplo, é compreender a importância do conceito e da metodologia que ele encerra. Vejamos algumas conseqûncias que podemos deduzir do experimento feito acima.

- As variáveis locais têm vida limitada, relativamente curta, produzem, portanto, economia de processamento, importante em grandes programas;

- as variáveis locais tendo vida local, permitem um controle mais efetivo dos seus valores, sabemos exatamente em que módulo, programa, elas se encontram, (ou pelo menos poderemos saber...).

- uma grave confusão pode se originar de variáveis locais com nomes idênticos aos de variáveis globais.

- O uso de sufixos `local` ou `L` nos nomes das variáveis, tornam os nomes diferentes, guardando o sentido que estes nomes tiverem.

Se evitarmos a identidade de nomes entre variáveis locais e globais, podemos concluir que o uso deste tipo de variável é recomendado. Devemos tomar como metodologia, sempre que for possível adotar variáveis locais e evitar o uso de variáveis globais.

Exercícios: 29 *1. Faça um diagrama, como na figura (fig. 6.1) para representar o programas dos exercícios.*

2. *Chamamos* nível de profundidade *de um programa, ao número inteiro que meça a maior quantidade blocos encaixados num programa. A figura (fig. 6.1) mostra um programa cujo nível de profundidade é dois. Cálcule o nível de profundidade dos programas dos exercícios.*

3. *Calcule o nível de profundidade dos programas* `prog*.c` *que você deve ter em disco, (se não tiver, solicite pelo endereço tarcisio@e-math.ams.org).*

O nível de profundidade *é uma medida para analisar a complexidade de programas.*

6.2 Técnicas com o uso de variáveis locais

Na introdução, que por sinal estaria no momento oportuno para ser lida, insistimos numa técnica de bem programar. A seguinte frase pertence a um outro autor e é repetida, com nuances diferentes, em todo livro de programação: "*é tão difícil corrigir-se um programa ruim, que é melhor voltar a aprender a programar e fazer outro programa.*"

Não há nada mais verdadeiro, apenas é preciso não cair no mau hábito de sempre querer estar fazendo tudo de novo. É preciso aprender as fazer coisas corretamente, desde o princípio, em particular na *arte de programar computadores*[2].

Programar é uma arte, inserida na Ciência dos Computadores. Um dos parâmetros para medir se um programa é bom, *indiscutivelmente* é *a beleza.* Observe que não nos refirimos ao resultado do programa na tela, e isto também é importante. O código do programa deve

- ser bonito;
- deve ser fácil de ler;
- deve ser escrito de tal forma que outras pessoas o consigam entender e possam alterá-lo com facilidade.
- deve ser escrito de tal forma que possa ser reciclado.

O contrário desta lista de características seria um programa que somente o autor consiga ler. O resultado, necessáriamente, será um código que nem o autor conseguirá ler algumas semanas depois.

[2] título de um livro de Donald Knutt

Como dissemos acima, na companhia de um autor renomado, programar computadores é uma arte, e consequentemente é difícil ensinar esta arte a outras pessoas. Como *arte*, é uma questão pessoal, no sentido de que duas pessoas que resolvam o mesmo problema, computacionalmente, quase com certeza, vão escrever códigos distintos. Quase com certeza, também, os programas vão coincidir nos aspectos fundamentais.

Você vai ter que desenvolver sua própria arte em programação como cada programador desenvolveu a sua. Isto não nos impede, e muito pelo contrário, é necessário, lermos os programas feitos por outros programadores para com eles dominar os detalhes iniciais e os avançados.

- Quando ainda não soubermos, temos que acompanhar os mestres;
- quando já formos mestres, temos que ficar atentos nas dificuldades dos discípulos para manter acesa a capacidade de crítica de nossos próprios métodos.

A arrogância, sem dúvida, é um indício de corrupção! e tome a palavra *corrupção* no sentido que você quiser. Funciona!

Ao final desta seção, vamos transformar um programa extenso em módulos mostrando-lhe como eliminar as variáveis globais.

Vejamos um exemplo de programa que executa um item de um menu. O programa está incompleto, falta a implemtação das funções

`executa_item_do_menu(inteiro item), apresentacao().`

Abaixo vamos transformá-lo eliminando a variável global.

Exemplo: 8 *Programa com um única variável global*

```
1) tipo principal()
2){
3)      inteiro item;
4)      apresentacao();
5)      item = menu_do_programa();
6)      executa_item_do_menu(item);
7)      volta tipo_de_dados;
8) }
```

Este programa tem apenas uma variável global, `item`*, e se possível, devemos eliminá-la.*

Este é um padrão básico de programas. Digamos que ele representa o planejamento inicial de qualquer projeto. Vamos analisar cada item tendo sempre em mente a questão das variáveis.

- `tipo principal()` *Porque todo programa, todo projeto, falando de forma mais correta, tem uma função principal que gerencia o sistema de programas que formam o projeto. No meio dos que programam em* `C` *é comum a afirmação de que o* tipo *da função principal deve ser* `inteiro`.

- `inteiro item;` *Uma variável global que vai servir de ligação entre o*
  `menu_do_programa()`
  e a função que executa as opções do menu. É a única variável global do sistema. A função
  `menu_do_programa()`
  apresenta as possibilidades do projeto e recebe do usuário a sua intenção sob forma de um número inteiro que vai passar para
  `executa_item_do_menu(item).`

- `apresentacao()` *Faz uma descrição do sistema.*

- `menu_do_programa();` *O menu já descrito.*

- `executa_item_do_menu(item);` *quem faz o trabalho.*

- `volta tipo_de_dados;` *Todo programa deve terminar com este comando. Num "autêntico" programa em C deve ser "sempre" um inteiro, seguindo a tradição. Este número deve informar ao sistema operacional se a execução do programa foi feita corretamente, e, em caso contrário, informar o nível de falha na execução do programa.*

Não precisamos, para um grande programa, mais do que uma variável global, e mesmo assim, uma, é muito. Todos os dados serão gerenciados localmente por funções específicas chamadas a partir do menu que *resolverão* todas as questões com as variáveis enquanto elas estiverem *no ar*, deixarão todos os dados gravados em disco antes de *saírem do ar*, de modo que não haverá nenhum valor *zanzando* pela memória sem controle.

Abaixo as variáveis globais! é a regra.

Vamos agora reformular o "programa"acima mostrando-lhe como podemos eliminar a única variável global. Veja como se faz e acompanhe as justificativas que apresentaremos depois.

Exemplo: 9 *Eliminando a variável global*

Compare o programa do exemplo (ex. 8). Observe que numeramos as linhas e agora algumas linhas vão ficar faltando, porque foram eliminadas de um exemplo para o outro.

```
1) inteira principal()
2){
4)      apresentacao();
6)      executa_item_do_menu(menu_do_programa());
7)      volta 0;
8) }
```

- *Na linha (1), definimos o tipo de dados da função* `principal()` *como* `inteira`. *Observe que na linha (7) agora está retornando 0. Não precisava ser zero, poderia ser um número calculado dentro do programa que indicasse ao sistema operacional o nível de funcionamento do programa, este é um detalhe mais especializado...*

- *Desapareceu a linha 3, porque não precisamos de variável global.*

- *Desapareceu a linha 5, porque a funcao* `menu_do_programa()` *foi parar dentro da área de parâmetros de*

  `executa_item_do_menu()`.

  Desta forma, em vez de guardar a resposta de

  `menu_do_programa()`

  em uma variável global, repassamos esta resposta diretamente para a função

  `executa_item_do_menu()`

  sendo assim desnessário o uso de variáveis globais.

- *É preciso observar que o programa ficou menos legível. Agora estamos usando o conceito "*função composta*" para eliminar variáveis globais. Em computação este assunto é registrado sob a rubrica* passagem de valor .

  Isto deixa a lógica do programa mais difícil de entender. Tem duas formas de resolver este novo problema:

  1. *Comentários explicativos colocados nos pontos críticos do programa*
  2. *Uso de* `identificadores` *mais claros. Isto fizemos no "*`programa`*" acima:* `executa_item_do_menu()`, *diz, com o seu nome, que ela recebe a escolha feita em* `menu_do_programa()` *e vai executá-la.*

  O tamanho dos `identificadores` *é praticamente ilimitado: 256 caracteres (três linhas). Ninguém precisa de algo tão grande, mas podem ser frases inteiras como* `executa_item_do_menu()`.

Este programa existe, veja `pensionato.c`[3], *leia-o !*

Eliminamos a variável global.

Abaixo as variáveis globais! é a regra.

E esta regra se propaga para dentro das funções particulares do sistema. Um bom sistema é feito de uma multidão de pequenas funções cuja redação deve caber na tela e que tenha controle completo de todas as variáveis envolvidas. Se algum dado for enviado para outra função devemos nos arranjar para a função interessada receba este dado diretamente e você está vendo aqui uma das principais funções do "comando" `return` ao final de cada função.

Eis um sistema seguro, eis uma forma avançada de programar.

Vamos discutir na secçào final deste capítulo, como $\mathcal{C}$ transfere valores entre as funções.

[3]no diretório para DOS, procure `pension.c`

Exercícios: 30 *Transformando global em local*

1. *Leia e rode o programa* `padrao.c`*.*
2. *Transforme* `padrao.c` *num programa que faça alguma coisa.*
3. *Faça uma cópia do programa* `contabilidade.c` *em outro arquivo, por exemplo* `teste.c` *e transforme as etapas do programa em funções, pelo menos tres:*
   - `apresentacao()`
   - `menu_contabilidade()`
   - `executa_contabilidade()`

   passe o valor de `opcao` *para uma variável inteira da função principal e passe esta variável como parâmetro para* `executa_contabilidade()`
4. *Elimine a variável que recebe a opção de* `menu_contabilidade` *usando esta função diretamente como parâmetro de* `executa_contabilidade()`*.*

   solução: `pensionato.c`
5. *erro de compilação No programa* `pensionato.c`*, identifique na função "principal()"a linha*

   `fim = executa_pensionato(menu_pensionato());` *e nela troque*

   menu_pensionato()

   por

   menu_pensionato

   rode o programa e analise a mensagem de erro produzida. Tente dar uma explicação para a mensagem de erro.

   solução: ver índice remissivo, "ponteiros,tutorial".

6.3 Passando valores entre funções

Nesta seção vamos trabalhar com o assunto: *passando valores entre funções.* Esperamos que você *rode e leia* os programas na ordem como eles são sugeridos no texto, inclusive os comentários contidos nos programas.
Observe que os programas são parte integrante do texto do livro, embora distribuidos eletronicamente, em separado. Ninguém aprende a programar lendo livros...apenas lendo livros!

Rigorosamente falando, em $\mathcal{C}$ existem apenas dois tipos de dados:

1. *números* e sua variantes, inteiros, fracionários (float), (o que inclue caracteres, short int ...), e

2. *vetores* que são os ponteiros, os objetos diretamente associados com um endereço inicial e outro final (calculado a partir do tipo de dado).

Consequentemente, a passagem de dados entre funções usando os *ponteiros* tem que ser um pouco mais traumática porque na verdade se está passando um endereço e o valor tem que ser lido indiretamente.

O exercício 1, abaixo, merece sua atenção, sem dúvida, mas talvez você deve tentá-lo diversas vezes, algumas vezes deixando-o para depois. É um desafio.

Exercícios: 31 *Passagem de dados*

1. *Desafio Transforme o programa* `162.c` *para fazer a passagem de dados entre* `menu()` *e* `executa()` *usando* `string`, *"d", "h", "c" em vez de* `caracteres`, *como esta projetado atualmente, 'd', 'h', 'c'.*
   Solução programa `163.c`.

2. *Os programas* `163.c` *e* `162.c` *tem uma péssima apresentação. Melhore a apresentação dos programas e envie uma cópia para o autor.*

3. *O programa* `165.c` *é uma alteração de* `163.c` *para que seja impresso o endereço da variável. Analise o programa e idenfique onde se faz isto.*

4. *Desafio, outro? Transforme o programa* `163.c` *para que sucessivamente as funções* `apresentacao(), menu()` *passem dados para* `executa()`.

   Solução programa `164.c`, *veja também o programa* `pensionato.c`

5. *Leia os comentários no programa* `164.c`

Se você conseguir resolver o problema, de forma diferente de `163.c`, me envie uma cópia, eu irei colocar sua solução no livro, com seu nome, mas se você conseguir fazê-lo sem variáveis globais...!

O objetivo do exercício 1 é mostrar a dificuldade de *passar valores* em $\mathcal{C}$ quando estes valores não forem inteiros. Observe também que `switch()` somente aceita parâmetros inteiros, o que forçou que fosse substituido por uma cascata de `if-elses`.

Numa comparação, quando se passam valores usando variáveis do tipo `ponteiro` o que se faz é dizer *em que endereço* XXX o valor se encontra. Veja `164.c`.

Vamos analisar o programa `sexto.c` para produzir uma versão melhorada do mesmo.

Exercícios: 32 1. *Rode o programa* `sexto.c`. *Compile,*

```
gcc -Wall sexto.c -oprog
```

e rode

`prog` *em alguns sistemas,* `./prog`

2. *Leia* `sexto.c` *procurando entender cada uma das suas tres funções. Quantas variáveis globais o programa tem?*

3. *Há uma mensagem de advertência no rótulo do programa* `sexto.c`*, leia e identifique no programa as variáveis a que ela se refere.*

4. *Suite* `sexto01,sexto02,sexto03`

   (a) *Leia e rode o programa* `sexto01.c`

   (b) *Verifique onde é usada a variável* `sinal` *na função*

   `principal()`

   e qual sua razão.

   (c) *Verifique que a variável* `sinal` *é desnecessária, você pode colocar diretamente*

   `verificacao(senha)`

   como parm̂etro do `se()`*. Experimente isto e elimine esta variável global. Solução* `sexto02.c`

   (d) *Verifique onde a variável* `senha` *é utilizada na função*

   `principal()`.

   Torne esta variável local usando-a exclusivamente na função `recepcao()`*.*
   Solução `sexto03.c`

   (e) *Observe que na compilação de* `sexto03.c` *há uma advertência (warning). Conclua que "solução" encontrada no programa* `sexto03.c` *para tornar local a variável* `senha` *não é boa. Ela deve ser global no âmbito do prgrama, "absolutamente" global. Corrija o programa* `sexto03.c`

   Solução `sexto04.c`

   (f) *Leia as observações presentes em todas as versões de* `sextoX.c` *e resolva os problemas propostos.*