Capítulo 5

Criando funções

Em Matemática, a expressão

$$y = f(x)$$

significa "*executar sobre o objeto x as operações "compactadas" em f* ", gerando um novo objeto que chamamos y ou $f(x)$.

$$y \text{ é a imagem de } x \text{ por } f.$$

Resumimos no símbolo f um conjunto de operações[a].

Em $\mathcal{C}$, o signficado de *função* não é exatamente este que temos em Matemática, mas se encontra bem próximo no sentido de *resumir um conjunto de operações* com um símbolo. Em computação, como em Matemática, a definição de funções acrescenta novos *vocábulos* à linguagem, "novos comandos", dizemos em computação.

As funções são algoritmos. Em $\mathcal{C}$, o menor *módulo executável* é uma função e um programa se constitue de uma função `principal()` (`main()`) que pode chamar uma lista de outras funções definidas no mesmo arquivo ou em alguma biblioteca *incluída* no começo do arquivo.

[a] por razões práticas e teóricas, precisamos da função trivial $f(x) = x$, também chamada de identidade.

Nos capítulos anteriores fizemos os nossos primeiros programas, quase todos com o formato:

```
programa em português              programa em inglês
tipo_de_dados principal()          tipo_de_dados main()
      inicio                             {
            comando1;                          comando1;
            ...                                ...
            comandoN;                          comandoN
      fim                                }
```

Programas formados de uma única função. Esta é uma metodologia antiga e perigosa para escrever programas, porque com ela se desenvolvem programas longos. Quando um programa for longo, se desenvolvendo por várias páginas, com dezenas, centenas ou milhões de linhas, é difícil de se verificar a sua correção.

Deve-se fazer o contrário: *escrever programas pequenos que executem uma única tarefa de natureza bem simples.* É o que se entende por *programação modularizada* que se encontra a meio caminho das técnicas mais recentes de programação, como *programação orientada a objeto, POO.* No penúltimo capítulo faremos uma breve apresentação desta técnica.
Em $\mathcal{C}$ isto se faz *construindo funções.*

Estivemos todo o tempo fazendo observações que o prepararam para chegar ao meio do livro.

Você está chegando ao meio do livro.

Neste capítulo vamos aprender a modularizar os programas, de forma sistemática, usando o conceito de função. Usaremos como ponto de partida os diversos exemplos que construimos nos primeiros capítulos e os iremos modificar. Eles serão assim a idéia motivadora na construção das funções ao mesmo tempo que você verá, de uma forma inteiramente natural, como aparecem as funções que precisamos e, inclusive, irá aprender o método inverso de trabalho: *construir primeiro as funções que realizam as pequenas tarefas que resolvem um grande problema.*

A teoria que necessitamos já foi praticamente toda desenvolvida nos 4 capítulos anteriores o que nos resta agora é desenvolver técnicas para construir programas seguros e interligar pequenos programas num projeto maior.

Ainda faremos mais um aprofundamento teórico nos capítulos 6 e 7 que se encontram na outra metade do lívro, seguidos do estudo de alguns problemas aplicados.

Mas as técnicas básicas de programação você as vai adquirir neste capítulo completando a lógica do capítulo anterior.

A tecnologia que precisamos é *surpreendentemente* simples:

como construir funções.

Vamos mostrar-lhe com um exemplo. Considere o seguinte programa:

```
# include <stdlib.h>
# include <stdio.h> // (1) leitura da biblioteca padrao
inteira principal()
{
        char deposito[80];
        inteiro numero;
        imprima("Este programa lhe pede um numero maior do que 10 \n");
        imprima("Testa se o numero eh maior do 10 e o informa \n";
        imprima("se voce acertou. \n");
        imprima("Me de um numero maior do que 10");
        leia(deposito, tamanho_do(deposito), entrada_pdr);
        converte_palavra(deposito, "%d", & numero);
        se (numero > 10) imprima("Voce acertou o combinado \n");
        ou_entao imprima("Voce errou ! \n");
        voltar(0);
}
```

Há três *momentos* no programa:

- Uma mensagem inicial;
- uma entrada de dados;
- um teste com as respectivas mensagens.

Vamos chamar a mensagem inicial de `mensagens()` e criar o bloco lógico

```
inteira mensagens()
{
        imprima("Este programa lhe pede um numero maior do que 10 \n");
        imprima("Testa se o numero eh maior do 10 e o informa \n";
        imprima("se voce acertou. \n");
        voltar(0);
}
```

Vamos criar outro bloco lógico formado pelo teste e suas mensagens sob o nome: `resposta()`. Mas agora vamos colocar uma variável em

```
resposta(inteiro numero)
```

```
inteira resposta(inteiro numero)
{
        se (numero > 10) imprima("Voce acertou o combinado \n");
        ou_entao imprima("Voce errou ! \n");
```

```
        voltar(0);
}
```

Agora a função principal se resume a:

```
# include ¡stdlib.h¿
# include ¡stdio.h¿ // (1) biblioteca padrao
inteira principal()
{
        char deposito[80];
        inteiro numero;
        mensagens();
        imprima("Me de um numero maior do que 10");
        leia(deposito, tamanho_do(deposito), entrada_pdr);
        converte_palavra(deposito, "%d", & numero);
        resposta(numero);
        voltar(0);
}
```

Ao passar por por `mensagens()` o programa vai buscar o conteúdo desta função e o executa. Ao passar por `resposta(numero)` o programa vai buscar o contúdo de `resposta()` e o executa com o valor que a variável `numero` tiver recebido. Dizemos que o valor de `numero` foi passado para `resposta()`.

Este programa se chama `funcao.c`, leia-o.

Este é o objetivo deste capítulo: funções. Vamos repetir o que fizemos com o programa `funcao.c` mais uma vez e você vai ser convidado a fazer uma lista de exercícios em que irá colocar as mãos para trabalhar na modularização de alguns programas.

5.1 Verificador de senhas.

Vamos construir um *verificador de senhas*, por exemplo, para um servidor de internet.
Observe que não estamos estabelendo um plano para montar um *servidor de internet*, porque, para montar um *servidor de internet*, precisa mais do que apenas saber testar senhas... faremos um *pequeno módulo* que será importante na montagem do *grande projeto*.
Infelizmente não podemos garantir, ainda, que o conteúdo deste capítulo possa ser imediatamente aplicado num grande projeto, num servidor de rede, por exemplo, por favor, aguarde mais um pouco, pelo menos até terminar o livro... quando você saberá como encontrar o restante.

A metodologia será a mesma que usamos nos capítulos anteriores.

Antes de discutir o *signficado* de "função" e *qual é formato genérico* de uma função em $\mathcal{C}$, vamos iniciar um processo de metamorfose do programa inicial, incitá-lo a rodar cada uma das novas formas, analisar os comentários ilustrativos e assim conduzindo-o à construção de programas mais completos.

Esta seção está baseada no programa `funcao01.c`[1]e suas transformações: `funcao0X.c`.

Exemplo: 6 *Testando uma senha.*

Veja o programa `funcao01.c`*, compile-o, rode-o*

```
gcc funcao01.c -Wall -oprog
prog
```

[2]

e depois o leia, analisando na tela o texto porque lhe faltam as mensagens explicativas: um defeito do programa que será explorado nos exercícios.

Este programa contém a parte central do que seria um módulo verificador de senhas de um sistema qualquer, de uma rede de computadores, ou de um terminal bancário. Obviamente ele ainda está muito rudimentar e o nosso objetivo aqui é o de torná-lo mais sofisticado.

Depois que você tiver compreendido o seu funcionamento, passaremos à discussão das versões melhoradas*.*

Leia o programa e os comentários que se encontram no arquivo, chame-o para uma tela do computador num editor de textos, enquanto você roda o programa noutra tela.

5.1.1 Metamorfoses do *Leitor de Palavras.*

Evolução `funcao01.c` → `funcao0X.c`

Observe nossa *mudança de comportamento*, não colocamos o programa dentro do texto, porque você tem os programas em um disco ou pode ir buscá-los num site na internet. Porque não tem mesmo sentido você aprender uma linguagem de programação apenas lendo um livro, você tem que estar sentado na frente do micro e ai pode abrir o programa n'outra janela.

Analisando

`funcao01.c`

podemos observar que o algoritmo tem tres momentos:

- apresentação A entrada de dados, quando a senha que vai ser testada é fornecida.
- análise O teste, comparação com o valor memorizado na variável `senha`.
- diagnóstico A saída de dados em que é emitida:
  - mensagem de sucesso,

[1] no diretório do `BC`, procure func0X.c

[2] Veja que trocamos, propositadamente, a ordem dos parâmetros passados ao `gcc`, a ordem é irrelevante. Talvez você precise digitar `./prog`

– ou, alternativamente, a de insucesso.

Quer dizer que o programa poderia ter sido escrito assim:

```
{
        apresentacao();
        analise(sinal);
        diagnostico();
}
```

em que

- `apresentacao()` é conjunto de mensagens iniciais que orientam o usuário sobre o objetivo do programa;
- `analise()` representa tudo que se encontra dentro do "`enquanto()`" e
- `diagnostico()` é a parte final.

O planejamento acima transforma o programa num `script`, um roteiro, como no teatro, o no cinema, o diretor chama os atores, em ordem para que cada um execute o seu papel. Aqui é a função `main()` que vai representar o trabalho do diretor do espetáculo.

A proxima lista de exercícios vai mostrar-lhe como você deve transformar

`funcao01.c` → `funcao02.c`

o programa que finalmente fizemos está ligeiramente diferente do planejamento acima, são coisas do planejamento: entre a *proposta inicial* e a *final* sempre existe uma *diferença*, mas deixamos registradas as duas para que você compreenda que isto pode acontecer, sem maquilagens.

Você está sendo conduzido por trás da cena em vez de ficar sentado na platéia.

Exercícios: 23 *Construindo funções*

1. *Rode e leia o programa* `funcao01.c`. *Como você verificou, faltam mensagens. Coloque as mensagens de entrada e saída de dados. Rode o programa, depois de alterado.*

2. *Separe a* análise de dados *da função principal em* `funcao01.c`, *dentro do editor de textos* corte e cole *a linhas de código, depois do* `fim` *abrindo uma nova função chamada*

   `inteira analise()`

   não se esquecendo do par `inicio,fim` *ou abrindo e fechando chaves. Compile e rode o programa, se o compilador indicar erros, antes de ler a solução procure entender as reclamações do compilador.*

   Solução `funcao01_1.c`

3. *Rode o programa* `funcao01_1.c`. *Leia o programa para entender como ele funciona e volte a rodar e ler até ficar claro o processamento.*

4. alarme.c *Escreva uma função que receba um número inteiro e emita duas possibilidades de mensagem*

   - *Se o número for menor do que* 3 *dê boas vindas ao usuário e imprima o número na tela.*
   - *Se o número for maior ou igual do que* 3*, dê um beep de alarme, uma mensagem de pesar, e imprima o número na tela.*

5. *Use o programa* `alarme.c` *e modifique* `funcao01_1.c`*.: separe uma função* `diagonostico(sinal)` *que faça a análise do que aconteceu em função do número de tentativas para entrar no sistema.*

   Solução `funcao01_2.c`

6. *Melhore a saída de dados de* `funcao01_2.c` *com algumas trocas de linha e espaços entre os dados.*

7. *O programa* `funcao01_2.c` *oferece ao usuário apenas duas possibilidades para "errar", apesar de anunciar que três possibilidades ficam garantidas, não tem uma terceira possibilidade. Corrija isto dando-lhe tres possibilidades de erro.*

8. *Acrescente mensagens no programa* `funcao01_2.c` *informando ao usuário que ele terá até 3 possibilidades para fornecer a senha e que esta é secreta e está gravada no programa (você pode escrever uma mentirazianha, dizer que a senha está em um arquivo secreto...)*

9. *Acrescente mensagens no programa* `funcao01_2.c` *informando ao usuário que, após errar tres vezes, o sistema só lhe dará possibilidade de novas tentativas depois de uma hora. Mas não vamos implementar este aspecto agora...*

   Solução: `funcao02.c`

10. *Rode o programa* `funcao02.c` *e veja como funciona. Depois leia o programa. Repita este processo até entender o programa completamente.*

11. *Melhore o programa* `funcao02.c`*, suas mensagens, lay-out, etc...*

12. *Defina duas funções:* `sucesso()`*,* `insucesso()` *que imprimam as mensagens acima de sucesso ou insucesso. Teste estas funções isoladamente como descrevemos acima, depois inclua as funções em* `ambiente.h`*. Altere* `funcao02.c` *para fazer uso das duas funções criadas agora.*

    Não se esqueça de incluir a linha

    ```
    # include ambiente.h
    ```

13. *Generalize, (aumente o grau de abstração) das funções* `sucesso(), insucesso()` *deixando que elas recebam uma mensagem auxiliar indicando que tipo de sucesso ou insucesso ocorreu.*

    Solução em ambiente.h.

Observação: 22 *Como construir funções*

As funções que foram objeto dos exercícios anteriores, foram testadas antes de produzirmos o programa `funcao02.c` *que é a modularização de* `funcao01.c`.

Leia os comentários em `funcao02.c` *em que explicamos "tecnicamente" como se faz esta modularização. Aqui, mais abaixo, vamos retomar esta discussão em termos mais genéricos e menos "técnica". Leia também os comentários contidos nos módulos de teste,* `funcao021.c, funcao022.c`

O método

Tudo que fizemos foi `copiar e colar` *o conteúdo dos programas primitivo colocando este conteúdo sob tres etiquetas*

`apresentacao(), leitora(), diagnostico().`

Este é o uso mais imediato das funções *na linguagem* $\mathcal{C}$.
Vantagens?
Observe que sim e quais:

1. *trabalho limpo Primeiro uma vantagem psicológica: O programa tem uma função* `principal()` *que é um* resumo limpo do trabalho.

2. *favorece metologias distintas A subdivisão em módulos não é única. Dois programadores diferentes encontrariam certamente duas formas distintas de definir os módulos.*

3. *criar móduloes reutilizáveis Neste caso temos um problema muito simples e forçamos a divisão em tres módulos.*

   Se bem aproveitado, este método deve criar pequenos móduloes reutilizáveis em outros programas. . Reciclagem de programas.

4. *Facilita a verificação dos programas A função* `diagonostico()` *contém um* `se()` *e geralmente é bom fazer um módulo separado para testar o* `se()`, *verificar se seu comportamento é o esperado. As duas outras funções podiam formam um único módulo, mas as dividimos em dois para ilustrar melhor o método.*

5. `main()` *é o planejamento do trabalho Você está diante de um* planejamento vazio... *programamos*

   A função `principal()` *é uma espécie de listagem de operações. Na "prática" as operações mesmo, serão construidas depois. Em outras palavras, a função* `principal()` *pode funcionar como um planejamento do trabalho, em que apenas descrevemos suas etapas, "funções".*

6. *teste de pequenos programas Cada uma dessas etapas pode ser testada separadamente, e veja como:*

- *Escrevemos* `leitora()`*, em um arquivo separado, ver*

  `funcao021.c`

  e lhe demos o nome de `principal()`*, porque seria a única função do programa* `funcao021.c`*. Compilamos e rodamos e pudemos ver os erros que sempre se cometem. Corrigidos os erros, o módulo* `leitora()` *ficou pronto.*

- *criação de bibliotecas Escrevemos* `diagnostico()` *em outro arquivo separado, ver*

  `funcao022.c .`

  Neste caso precisamos de uma outra função "`principal()`*" para passar o valor da variável "*`resultado`*" que a função* `diagnostico()` *analisaria para decidir qual frase seria emitida:*

  `sucesso ou insucesso.`

- *programas, o embelezamento Não testamos o primeiro módulo porque era uma simples lista de "imprima()s", mas o certo seria também fazê-lo porque poderia haver algum erro de formatação de dados, ou,* **mais importante***, poderiamos ter testado a* beleza*, a* estética *das mensagens. Depois que um programa está funcionando podemos incluir nele alguma arte, tendo cuidado para não criar poluição visual.*

Os exercícios abaixo exploram esta técnica. Rode os módulos de teste, também, para ver o efeito que eles têm.

Observação: 23 *Pequenos programas que façam pouca coisa.*

Esta é a técnica onde o uso de funções entra,

`funcao01_1.c, funcao02.c`

rodaram da mesma forma como `funcao01.c`*, porém visualmente mais limpas e todas suas partes podem ser vistas na tela, logo, fáceis para corrigir.*

Esta experiência deve levá-lo a concluir que é vital a programação modularizada. Também deve convencê-lo da importância de escrever pequenas funções que possam ser reutilizadas facilmente, como as funções `sucesso(), insucesso(), apeteco()` *, definidas na biblioteca* `ambiente.h`

A função `leitora()` *mostra bem o que já comentamos anteriormente. Dividimos um problema em pequenas tarefas, cada tarefa executada por uma função específica. A função* `leitora()` *pode ser utilizada em muitos contextos, e você vai ver isto acontecer aqui. Claro, não precisa ser como se encontra acima, com alguma pequena modificação, por exemplo nas mensagens, (e até isto pode ser generalizado...)*[3]*, em vez de*

Leitura de senha com sucesso.

poderiamos ter usado apenas

Sucesso.

[3]esta generalização é o que se chama de *abstração*, quer dizer tornar uma função menos particular para que possa ser melhor reutilizada

que no presente contexto significa "sucesso de leitura da senha" mas em outra contexto significaria outro tipo de sucesso. Ver o programa `sucesso.c`.

Observação: 24 *Programas reutilizáveis*

Esta é uma idéia central da programação, fazer programas simples, tão simples, que possam ser reutilizados em muitas circunstâncias.

Além do mais, um programa simples é pequeno e difícil de ter erros, e se tiver, é fácil de detectá-los.

Se um programa for pequeno e executar uma tarefa bem genêrica, os membros do grupo de trabalho podem encontrar o que fazer com ele, isto é a chave do trabalho em grupo.

Onde escrevemos *programa* leia *função...*

Observação: 25 *Roteiros e o teatro computacional.*

Programar hoje se aproxima muito de tornar vivo *um cenário.*

- *Primeiro construimos funções;*
- *quando você avançar mais em computação verá que em computação tudo gira em torno de* objetos *e dos métodos que processam os dados definidos por estes objetos. Veja o penúltimo capítulo.*
- *As funções, e mais do que elas os objetos, contém as informações sobre um pequeno modelo computacional.*
- *Finalmente escrevemos a função principal, que é o roteiro, no mesmo sentido que se usa em* teatro*, que chama os figurantes,*

  `as outras funções,`

  fazendo rolar a "cena",

  `rodar o programa ...`

- *Muitas vezes escrevemos primeiro a função* `principal()` *que então tem o duplo papel, de planejamento do trabalho e de script ao final.*

É o que você está vendo no verificador de senhas, `funcao02.c`*, construimos as funções que executam cada um dos pedaços isoladamente, depois a função principal contém o script que executa o programa. É assim que se programa bem.*

A expressão linguagens de roteiros[4] *se refere às linguagens relativamente recentes, e bem mais evoluidas, um exemplo é* `Python`*, outro* `Perl`*, mas em qualquer linguagem moderna se pode programar da mesma forma. C++ é um exemplo de linguagem em que se pode programar assim, esta última é uma evolução do C.*

Exercícios: 24 *Melhorando o leitor de senhas*

1. *Visite a nossa biblioteca* `ambiente.h`*, estude as funções alí definidas. Inclua* `apeteco2()`, `apetecof()` *no leitor de senhas.*

[4] em inglês, *script languages.*

2. *Para limpar a tela antes do programa começar, faça uso de*
   `limpa_janela()`.

   Use esta função, junto com `apetecof()` *para limpar a tela quando o programa terminar.*

3. *Altere a função* `mask()` *para que ela apresente os seus programas, (definida em* `ambiente.h`*).*

No espirito com que nos propusemos a escrever este capítulo, baseado em exemplos, vamos agora construir mais outro exemplo. Vamos usar uma outra técnica de trabalho que vai ser desenvolvida ao vivo, (porque você deverá estar rodando os programas).

5.1.2 Sistema de contabilidade geral

metamorfose de `contabilidade.c`

No prefácio deixamos a pergunta sobre uma técnica de programar bem, com uma resposta evasiva. Aqui você vai encontrar uma pequena concretização do método.

Mas, *programar* é uma arte... e você, que está lendo este livro, pretende ser um artista.

É difícil o ensino das artes pelos aspectos subjetivos que elas envolvem. O fato de que *programar seja uma arte* não impede que tenha aspectos técnicos e é preciso desmitificar as artes, também. Podemos mostrar a nossa arte, e você desenvolver a sua. A grande regra é, *seja independente e ético.*

Obviamente, não espere encontrar aqui o programa definitivo para gerenciar a contabilidade geral nem mesmo de uma pequena empresa. Aqui você vai encontrar um pequeno exemplo que pode ser o ponto de partida para o `sistema de contabilidade geral` que você ainda vai construir. Não se esqueça que não somos contadores, portanto, apesar de que tenhamos recebido instruções de um contador sobre este programa, a construção definitiva de um programa nesta área tem que ser supervisionado por um profissional da mesma. Os programadores resolvem problemas acompanhados e supervisionados por pessoal especializado nas áreas de interesse dos programas.

Vamos produzir as funções que completem, pelo menos parcialmente, o programa `logica06.c`, como prometemos anteriormente.

Primeiro vamos fazer o que devemos: criar um outro arquivo com o nome, `contabil.c`, deixando intacto `logica06.c` porque aquele tem uma função específica para o trabalho do livro e não deve ser mexido, apesar de dizermos no título que faremos as metamorfoses dele, vamos deixá-lo em sua forma original.

Exemplo: 7 *Reciclagem de programas*

A reciclagem de programas *economisa tempo de digitação e oferece mais segurança e robustez aos sistemas. Um programa que esteja funcionando bem deve ser re-utilizado (reciclado).*

Para evitar de perder tempo, adquira algumas técnicas:

1. *Só recicle programas que estejam funcionando bem.*
2. *Copie um programa que esteja funcionando bem para outro arquivo. Modifique o novo e não mexa no velho.*
3. *Todo programa deve ter um cabeçalho que identifique o que o programa faz, quem o fez, etc... inclusive as dificuldades falhas e melhoras desejadas.*
4. *Ao reciclar mude o nome do programa internamente no cabeçalho, isto pode ser crucial quando você fizer buscas para encontrar o programa* certo*, que você precisa, para reciclar.*

Esta maneira de agir deve ser uma regra de trabalho relativamente aos arquivos que você recebeu em disco com a observação de que os pode passar em frente desde que não os altere*, a razão é esta descrita aqui. Se você alterar algum programa, deve trocar-lhe o nome, até mesmo porque você não vai querer que ganhemos a fama pelo que você fizer, vai? Entretanto, nós queremos ter o respeito pelo nosso trabalho.*

Como guardar o que esta feito e iniciar uma alteração ?
Em LinuX:

`cp logica06.c contabil.c,`

ou, usando algum gerenciador de arquivos de sua preferência, faça a cópia de `logica06` para o novo arquivo `contabil.c`.

Isto já deve estar feito no disco que você adquiriu com o livro, use outro nome diferente de `contabilidade.c` para não perder as modificações que já fizemos neste último programa, por exemplo, `contabil.c` que também se encontra no disco.

Observação: 26 *Delineando o método de trabalho*

- *Seguindo os princípios da* Contabilidade*, há dois arquivos diferentes que devemos criar e manipular,* `deve, haver` *e os contadores tem suas esquisitices técnicas, tudo que se paga,* está em "haver" *e tudo que se recebe* fica em "deve"*, vamos seguir estes princípios.*
- *Vamos criar a função* `deve()`*, ou melhor, o protótipo da função*[5] `deve()`*, no arquivo* `deve.c`*.*
- *A função* `principal()` *fará o seu papel de controladora de* `deve()`*, e depois de testada e aprovada, será transferida para biblioteca*

  `contabilidade.h`

  para ser chamada pelo programa `contabil.c`*.*
- *Todas as funções que iremos aqui construir, serão transferidas para*

  `contabilidade.h,`

  a nossa biblioteca de contabilidade geral, depois de serem testadas individualmente e estiverem funcionando a contento.

[5] a palavra *protótipo* é usada com um significado especial, discutiremos isto no próximo parágrafo

5.1.3 Como registrar dinheiro

Vamos criar `deve.c`, leia o arquivo no disco. Este arquivo é semelhante ao `prog20_3.c` e se quiser seguir a evolução do acesso ao disco, veja os programas `prog20_x.c` e leia no capítulo 7, mais aprofundadamente, as regras para acesso a arquivos em disco. Aqui nos vamos limitar aos comentários que se encontram dentro dos programas para não desviar a atenção do assunto principal, as funções.

Exercícios: 25 *Alterando o arquivo* `deve.c`
Leia o arquivo `deve.c`

1. *Identifique o local em se dá a leitura de dados da razão social de quem paga, e elimine a variável intermediária "deposito", use diretamente "nome". Solução* `deve02.c`

2. *Leia o arquivo "*`deve.dat`*" onde os dados estão sendo gravados. Altera o registro no arquivo para que os dados fiquem separados por uma linha. Solução* `deve02.c`

3. *Torne possível que o nome do arquivo seja lido pelo teclado. Solução com erro em* `deve03.c`*, ver* `deve04.c`*.*

4. *Substitua a versão do* `deve.c` *na biblioteca* `contabilidade.h` *e rode o programa* `contabil.c`

```
gcc -Wall -oprog contabil.c
```

   o programa agora roda um pouco melhor do que antes.

5.2 Máquina de calcular.

Nesta seção vamos fazer algumas funções que efetuam cálculos e passam os dados para outras funções.
Vamos criar uma biblioteca `maquina.h` deixando-a pronta para ser incluída em uma interface gráfica, (que não será feita aqui) mas inteiramente funcional para ser usada no modo texto.

5.2.1 O menu de opções

A seguinte lista de exercícios vai conduzí-lo a construção do planejamento inicial do trabalho.

Exercícios: 26 *Planejamento da calculadora*

1. *Em LinuX execute*

```
grep menu *.c | more
```

   e você vai ter uma listagem na tela de todos os programas contenham a palavra "menu". A mesma ferramenta existe também no Windows... ela se chama

```
localizar
```

Selecione um deles para iniciar o programa `calculadora.c`.
Solução `calculadora01_1.c`. *Outra solução* `calculadora01.c`

Observe que não vamos começar com `calculadora.c` *mas sim com* `calculadora01.c`.

2. *Tanto* `calculadora01.c`, `calculadora01_1.c` *têm um defeito estético: primeiro permitem a leitura dos números, depois oferecem as opções do sistema, icluindo "parar". Corrija isto.*
Solução `calculadora02.c`

3. `calculadora01_1.c` *representa uma outra linha de programação. Analise a diferença com* `calculadora01.c`. *O uso de* `getchar()` *oferece riscos e sugiro que você evite esta função até que sua prática aumente. Vamos seguir com o método de* `calculadora01.c`.

4. `calculadora03.c` *termina o planejamento, mas está funcionando muito mal, está muito poluida, visualmente, rode, analise, e corrija.*
Solução `calculadora.c`

5. *Melhore o menu de opções em* `calculadora.c`, *se inspire em* `162.c` *inclusive usando um único* `printf` *que torne o programa mais legível.*