Capítulo 4

Controle lógico do fluxo

Aqui começa programação.
Vamos estudar a lógica por trás de *qualquer programa*, em *qualquer linguagem* de programação do *tipo imperativo*.
As palavras chaves deste capítulo são:

`se()`	`if()`
`se()/ou_entao`	`if()/else`
`escolha()`	`switch()`
`enquanto()`	`while()`
`para()`	`for()`
`pare`	`break`
`voltar()`	`return()`

Com estas palavras se fazem programas, portanto com elas podemos construir as "frases" com que vamos instruir um computador a processar os dados que lhe entregamos.

Vamos estudar a sintaxe desta comunicação, neste capítulo, é isto que muitas vezes é chamado de *lógica*.

4.1 O condicional `se()` (`if()`)

Vamos começar com duas estruturas de contrôle do fluxo:

1) se()	if()
se(certo) faça();	if(certo) faça()
2) se() ou_entao	if() else

se(certo)
faça();
ou_entao
faça_outra_coisa();
	if(certo)
	faça();
	else
	faça_outra_coisa();

Observe o uso do `ponto-e-virgula` no caso `if() else`, antes do `else` tem `ponto-e-virgula`. Esta observação é importante para programadores em Pas-

cal porque (no caso do Pascal) não tem `ponto-e-virgula` antes do `else`. Em $\mathcal{C}$, tem !

A função `se()` recebe um parâmetro e o *avalia*. Se esta avaliação resultar em *verdadeiro*[1] o comando associado será executado. Veja, por exemplo, o seguinte bloco

```
imprima(Forneça-me dois numeros );
imprima(O segundo numero deve ser maior do que o primeiro);
se(numero1 < numero2 )
      {
            imprima(OK ! voce obedeceu ao combinado);
      }
```

Este bloco está implementado no programa `prog05.c`. Leia e rode o programa.

Esta comunicação está claramente incompleta. Falta uma alternativa: *... se o usuário do programa tiver fornecido um segundo número* **menor** *do que o primeiro* ?

Aqui entra o `ou_entao` (`else`). Vamos completar o esquema.

```
imprima(Forneça-me dois numeros );
imprima(O segundo numero deve ser maior do que o primeiro);
se(numero1 < numero2 )
      {
            imprima(OK ! voce obedeceu ao combinado);
      }
ou_entao
      imprima(Voce desobedeceu ao combinado !  )
```

Se a condição testada pelo `se()` se verificar falsa, o programa se conecta, imediatamente, a um `ou_entao` que se encontre depois `se()`.

Se você, por engano, colocar algum comando entre o final do `se()` e o `ou_entao` haverá uma mensagem de erro indicando isto. O `prog05_1.c` está preparado com este erro, basta apagar o comentário antes do `ou_entao`. Experimente !

Em geral (nem sempre) usamos um par

`se(condicao) ou_entao`

[1] leia observação sobre verdade e falsidade.

rode o programa prog05_1.c que implementa o se() ou_entao. Depois leia o programa e volte a rodá-lo...

Exercícios: 9 *se()/ou_entao*

1. *Rode, leia, rode o programa* prog05_1.c.
2. *Apague o comentário da linha que precede o* ou_entao *e compile o programa. Ele vai reclamar dizendo que há um erro antes do* else *(*ou_entao*).*
3. *Altere o programa* prog05_1.c *para ele* diga[2] *que sabe calcular o fatorial de n e se o usuário apresentar um número maior do que 13 responda que passou da capaciade da máquina... solução* prog05_2.c
4. *Estude* prog05_2.c *e veja que tem* voltar *duas vezes com valores diferentes. Por que ? Observe que* prog05_2.c *é um exemplo de planejamento vazio... veja o próximo exercício.*
5. *Planeje um programa para calcular a divisão exata de dois números inteiros, o* dividendo *pelo* divisor*. Se o usuário fornecer um* divisor *maior do que o* dividendo *diga que é impossível,* ou_entao *que a o programa ainda não sabe fazer esta conta, pedindo, gentilmente, que volte depois.*

   solução prog05_3.c
6. *Altere o programa* prog05_3.c *pedindo que o usuário forneça mais alguns números e faça mais alguns testes para treinar o uso de*

   se()/ou_entao

   experimente omitir voltar*. Experimente* voltar *com outros valores inteiros.*

Os programas prog05_1.c prog05_2.c prog05_3.c todos tem a estrutura lógica da figura (fig. 4.1) página 74,

Há um teste, indicado na (fig. 4.1) como *condição* e, em qualquer hipótese, Verdadeira ou Falsa, o programa deve parar. Então o comando voltar tem que aparecer **tanto** dentro do se() como dentro do ou_entao. Este é um erro comum de lógica de programação, o programador esquecer de fazer uma análise lógica do fluxo, e consequentemente, se perder nas saídas do programa.

Experimente, apague um voltar nestes programas e compile.

Veja mais este exemplo, com fatorial[3].

```
if(n<10)
      {
            fatorial(n);
      }
```

[2] logo veremos um programa para calcular fatorial, por enquanto faça apenas o planejamento...

[3] precisamos de enquanto() para calcular o fatorial. Por enquanto será apenas *planejamento vazio...*

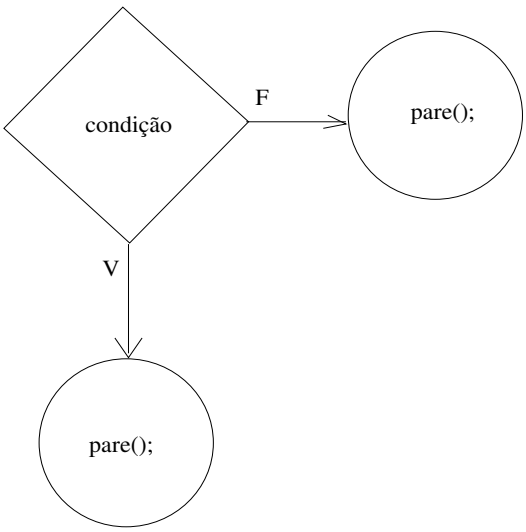

Figura 4.1: `se() ou_entao`

irá calcular o fatorial de n se o número n for menor do que 10. Como não existe o fatorial de números negativos, este algoritmo precisa ser melhor especificado pois *ele não duvidaria* em calcular o fatorial de -1 se o número $n = -1$ lhe fosse apresentado.

Precisamos de mais uma estrutura de controle de fluxo para resolver o problema do fatorial. Se estiver curioso, leia o programa `fatorial.c` mas você pode resolver os exercícioso desta seção sem resolver inteiramente o problema: *planejamentos vazios*. Se habitue a esta idéia que ela é essencial em programação.

Sempre fazemos *planejamentos vazios* numa primeira etapa, e esta seção traz vários exemplos. Sem brincadeira, eles são importantes para uma a construção da lógica do problema.

Exercícios: 10 *Fatorial incompleto Escreva o programa* `fat_inc.c` *que execute o algoritmo incompleto acima. Veja "solução" no disco...*

Vamos agora melhorar o planejamento do fatorial.

```
se(n >= 0)
     inicio
          fatorial(n);
     fim
se(n<0)
     inicio
          imprima(‘‘nao existe o fatorial de %d’’,n);
     fim
```

Exercícios: 11 *Melhorando o fatorial*

Transforme o esquema gráfico acima no programa `fat_inc02.c`*, a segunda versão do fatorial. Veja a solução no disco. O programa* `fat_inc02.c` *faz uso de funções, vá ao capítulo 5 para uma breve leitura sobre funções para entender a solução.*

Existe um esquema gráfico, chamado `fluxograma`, que nos permite uma visualização do *fluxo* lógico.

As figuras (fig. 4.2),(fig. 4.3) (fig. 4.4) mostram a evolução de um programa, ver páginas 75, 76, 77.

A figura 4.2 representa os dois primeiros retângulos acima: um único `se()`. A figura 4.3 contém o planejamento para o caso de $n < 0$ indicando que o processo deve parar, com uma alternativa (`else`) para caso de que $n!$ ultrapasse a capacidade da máquina.

Observe que no primeiro caso o processo para, apenas se $n > 10$, mas a "máquina" tentaria calcular $(-1)!$, o fatorial de número negativo.

A figura 4.4 contempla uma entrada de dados, é um programa completo.

Veja na figura (fig. 4.2) página 75,

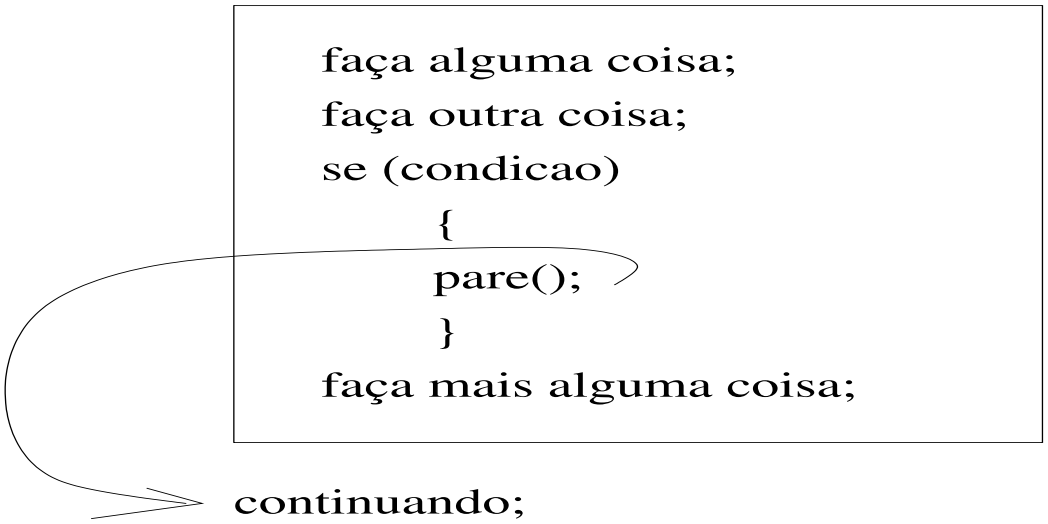

Figura 4.2: Fluxograma do se()

Quando for necessário usar dois `se()`, um completando o outro, então possivelmente é melhor usar `se() ou_entao()`:

```
se(verdadeiro) faça ou_entao  faça_outra_coisa
```

O exemplo do fatorial ficaria agora melhor descrito assim:

```
se(n >= 0)
        imprima("%d \ n ",fatorial(n));
ou_entao
        imprima("%s \n",
                "nao ha fatorial de numeros negativos");
```

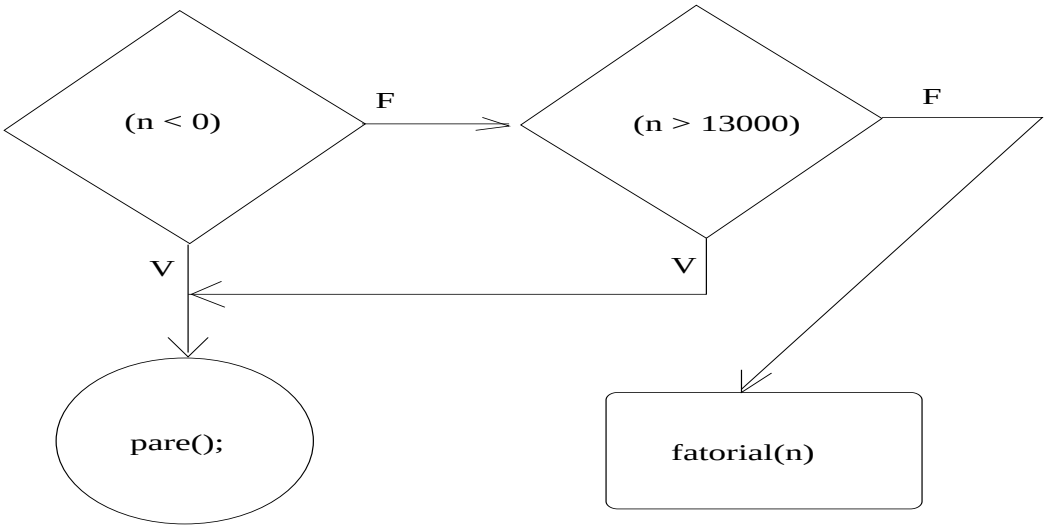

Figura 4.3: Fluxograma com dois se()

supondo que `fatorial()` seja uma função definida em alguma bilioteca.

Os fluxogramas nos trazem uma lição importante para o planejamento de sistemas:

> *rapidamente uma folha de papel será insuficiente para fazer o fluxograma de um problema.*

Isto significa que você deve dar um "*zoom*" no problema... quer dizer, resolver as questões maiores do problema, as grandes decisões, estas sim representadas num *primeiro* fluxograma.

Um exemplo fala mais que mil palavras. Vamos fazer o fluxograma para resolver uma equação do segundo grau.

- Uma equação do 2ºgrau tem tres coeficientes, logo tres entradas de dados; quer dizer tres retângulos no fluxograma.
- Temos que calcular o "delta", um círculo no fluxograma, porque os círculos representam *ações*.
- Pontos de decisão, *se o "delta" for positivo, nulo ou negativo*, tres pontos de decisão (dois diamantes),
- Tres círculos representados as tres possibilidades finais: duas raizes, uma única raiz, nenhuma raiz real.

Total 10 figuras geométricas, um desenho complicado difícil de ser colocado numa folha de papel. Pior, a visualização seria pobre que é o contrário do objetivo.

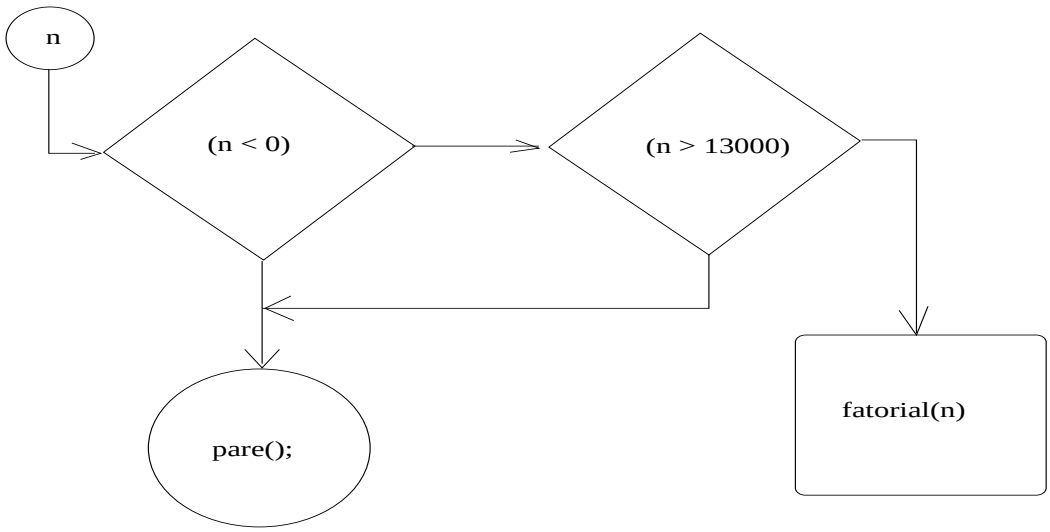

Figura 4.4: Fluxograma com dois se(), uma entrada e uma saída dados

Conclusão: devemos visualizar o problema em grandes linhas com um

`fluxograma-genérico`

e depois fazer

`fluxogramas-locais`

para cada um dos blocos do genérico.
No caso do problema das raizes o estudo genérico seria:

- Entrada de dados Primeiro uma única representação para entrada de dados.
- Decisão uma única, verifica se $\Delta \leq 0$ e conclue se tem raiz, (sim ou não). Remete ao cálculo das raizes ou a uma mensagem de impossibilidade.

Veja o fluxograma desta análise lógica na figura (fig. 4.5) página 78,
O bloco (lógico)

calcule as raízes

esconde vários cálculos, você está diante de um *planejamento vazio...* programamos em equipes. Este é a idéia inicial, depois cada um dos membros da equipe irá tratar do *seu módulo*.

Há funções escondidas dentro do fluxograma, mas a idéia geral do processamento está expressa.

Existe uma função para calcular o Δ *escondida* dentro do ponto de decisão. Tem uma função para calcular as raizes *escondida* dentro da frase "calcula as raizes". Há seis linhas de entrada de dados *escondidas* na frase "entrada de dados".

Isto a gente resolve depois, primeiro vem o planejamento. Claro, tem muito sistema que nunca passa do planejamento e da promessa.

Vamos escrever o planejamento do programa que corresponde a este fluxograma.

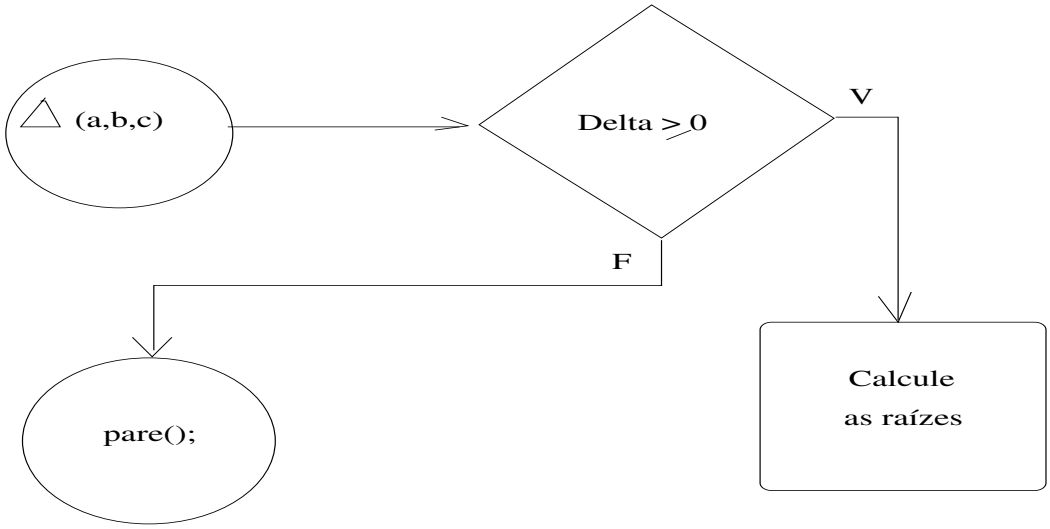

Figura 4.5: Fluxograma da equação do segundo grau.

```
entra_dados();
{
        delta = calcula_delta(a,b,c)
        se(delta >= 0)
                {
                        calcula_raizes(a,b,c);
                }
        ou_entao
                {
                        imprima("equacao impossivel");
                }
        voltar 0;
}
```

Depois iremos escrever as tres funções

- `entra_dados()`,
- `calcula_delta()`,
- `calcula_raizes()`

Quando começamos a produzir os programas fomos encontrando algumas dificuldades o que nos levou a construir tres versões do programa, em cada caso resolvendo uma parte do problema. Veja que em `seg_grau.c` existem três funcoes para entrar os dados, uma para cada um dos coeficientes. Seguramente tem forma melhor de resolver este problema e você será convidade no próximo bloco de exercícios, a ajudar o autor, com uma solução melhor.

Veja os programas

seg_grau01.c, seg_grau02.c, seg_grau03.c

e a *versão final* seg_grau.c. Rode os programas para ver como foi sendo feito planejamento, e depois leia cada um deles.

Exercícios: 12 *Equações do segundo grau*

1. *Faça o programa* equ_seg01.c *para executar o planejamento sugerido acima. Solução no disco.*

2. *Complete o programa* equ_seg01.c *para que ele resolva equações do segundo grau.*

3. *imprimindo o falso Experimente o programa* logica01.c*, rode-o e depois leia-o. (No BC procure logi*.c)*

4. *imprimindo o falso Observe a expressão*

   (primeiro==segundo)

   dada como parâmetro ao imprima*. Substitua* (primeiro==segundo) *por* (primeiro=segundo)*, rode o programa e veja a diferença. Substitua também como parâmetro do* se *e descubra que diferença isto faz.*

5. *Estude o programa* fatorial04.c*. Experimente retirar algumas chaves (pares de chaves) para ver a diferença. Ignore por enquanto as outras versões de* fatorial*.*

6. *Em* logica01.c *temos a função* se() *que irá avaliar uma* expressão *(observe que o parâmetro é abstrato...). Experimente fornecer ao programa dados não numéricos, frases. Em* BC *ver logi01.c.*

7. *"*(primeiro == segundo)*" é uma operação lógica seguida de uma* avaliação*: o parêntesis. Coloque um alguns "*imprimas*" em* logica01.c *para testar os valores de* (primeiro == segundo)*.*

8. *Rode diversas vezes o programa e tente encontrar uma lei de formação para o valor de (primeiro==segundo)*

9. *No programa* logica01_1.c *usamos*

   (primeiro = segundo)

   como parâmetro do if()*. Verifique que o compilador* gcc *vai avisá-lo do risco, mas o programa vai rodar. Se você fornecer ao programa os números*

   1,0

   nesta seqüência, ele responderá que os números são iguais. Por que ? Veja a resposta nos comentários do programa.

10. *erros que enganam o compilador. Rode um programa com o seguinte comando:*

    imprime(primeiro=segundo)

depois de definidas as variáveis, naturalmente, e observe o comentário "absurdo" do compilador.

Objetivo do exercício: prepará-lo para procurar erros em local diferente do apontado pelo compilador...

11. efeitos colaterais. *Escreva um programa que "rapidamente" iguale duas variáveis inteiras, se elas forem diferentes, e nada faça se elas forem iguais.*

Lição: 2 *Erros lógicos e de sintaxe. Num dos exercícios anteriores lhe pedimos que substituisse* `(primeiro==segundo)` *por* `(primeiro=segundo)`. *Aqui temos um exemplo de erro lógico que não é erro de sintaxe.*

A expressão (primeiro=segundo) se constitui de dois operadores:

- *operador atribuição,*
- *operador avaliação*[4].

*No nosso sistema de computadores seqüenciais,(*Von Neuman*) a CPU tem que fazer operações seguidas começando sempre da parte interna de uma expressão no modelo matemático de função composta:*

$$\textit{(entrada) } x : \tag{4.1}$$

$$x \mapsto f(g(h(x))) = f(g(y)) = f(z) = w \tag{4.2}$$

$$\textit{(saída) } w \tag{4.3}$$

A CPU se utiliza de um sistema de memória onde cálculos intermediários são guardados. Se você tiver a curiosidade de ler um trecho de programa em assembler*, verá, uma lista de operações semelhante à sequência de equações acima, apenas usando palavras como* `sto`*, abreviação de* `store`*, do inglês, que significa* guardar*, seguida de um endereço de memória.*

Hoje isto continua, apenas o programador escreve numa linguagem de alto nível *que cria o programa em* `assembler`*, quando compila o código escrito pelo programador.*

Qualquer linguagem de processamento (do tipo imperativo) faz uso deste sistema, de acordo com o modelo acima descrito, no cálculo de w,

w é conseqüência dos cálculos intermediários sobre as variáveis y, z. Sendo w a última, é passada para o "standard output", (saída padrão) que pode ser

- *o próximo comando a ser processado, (observe que x está sendo passado para o próximo comando, assim como y)*
- *o vídeo;*
- *a impressora;*

[4] alguns autores traduzem ao pé da letra do inglês "evaluation", que em português é "avaliação", por "evaluação", palavra inexistente...

- *um arquivo em disco, etc...*

No caso `(primeiro=segundo)`*, o resultado desta operação composta é o valor de* `segundo`*. Porque a operação* mais interna *é a atribuição que, ao ser executada, produz este valor (faz parte da engenharia do compilador* C*).*

Todas as operações ao serem executadas produzem valores e este é o segredo dos efeitos colaterais: nem sempre este valor produzido é claro, muitas vezes sendo um valor secundário efetuado por uma operação primitiva pode dar um ganho no programa, em velocidade[5]*.*

Essencialmente verdadeiro, tudo isto. Muito bonito se não produzisse algoritmos difíceis de serem lidos e interpretados. Esta velocidade pode ser consumida depois durante o trabalho de manutenção de um programa cheio de atalhos turtuosos*.*

Embora "`se (primeiro=segundo)`*" possa ser considerado um erro de sintaxe, o gcc*[6] *não tem condições de detectar isto, uma vez que o resultado desta operação, sendo uma expressão válida para o* `gcc`*, pode ser passada como parâmetro à função* `se()`*.*

Exímios programadores usam estes artifícios para conseguirem melhor desempenho em seus programas e infelizmente não conseguem mais entendê-los duas ou tres semanas depois... (nem eles e nem os seus companheiros de equipe), o que dá existência, provavelmente, a sistemas operacionais *bonitos, mas emperrados, e cheios de erros... praticamente impossíveis de serem encontrados, até porque não são erros, são os chamados* efeitos colaterais *.*

Os efeitos colaterais[7] *são um risco em programação e sem dúvida devem ser evitados a todo custo.*

Programação não precisa mais ser "econômica", como nos primeiros dias de sua existência. A economia a ser feita deve ser no trabalho do programador que, é, digamos assim, o "item"caro nesta área de trabalho. Computadores são relativamente baratos, tem bastante memória e velocidade, o que permite que os programas possam ser mais legíveis e fáceis de serem mantidos. Programação é coisa séria, não é brincadeira, de um programa mal feito podem depender vidas ou eleições.

Não podemos deixar de convidá-lo a fazer experiências diversas para descobrir detalhes sobre os efeitos colaterais *... eles são múltiplos e trazem preciosas lições que podem evitar horas perdidas na correção de programas, muito em particular a troca de* `==` *por* `=`*. São exemplos dos tais* insetos*, (bugs), que infetam programas enormes.*

C é uma linguagem imperativa, quer dizer que nela existem apenas duas opções: Dada uma relação, ela será:

- verdadeira ou, exclusivamente,

[5] mas pode ser lida por uma função como `scanf()`, é preciso saber reconhecer isto ou evitar de usar `scanf()`

[6] vamos escrever sempre "gcc" em vez de "compilador da linguagem C de agora em diante

[7] É difícil se livrar totalmente de efeitos colaterais, mas quando usados eles devem ser documentados.

- falsa.

Para o `gcc`, falso fica caracterizado pelo zero, qualquer outro valor diferente de zero representa o verdadeiro. Este é um ponto que você deve incorporar em sua lógica pessoal ao se tornar um *"C-programmer"*...

Exercícios: 13 *Efeitos colaterais*

Programar sem "efeitos colaterais" é difícil. No momento em que você se tornar um bom programador, dificilmente vai evitar de fazê-lo, entretanto, não se esqueça de que uma boa companhia para um "efeito colateral" é um comentário, que explique o que está acontecendo, inclusive para você mesmo, algum tempo depois...

Nos exercícios abaixo estamos lhe dando exemplos de "efeitos colaterais", habitue-se, aqui nos exercícios, a escrever o código incluindo os comentários explicativos, mesmo que o exercício não lhe peça para fazê-lo.

1. *Escreva um pequeno programa, (reciclagem de* `logica01.c`*), para testar que possíveis combinações de dos operadores* `=, ==`*, tem sentido, como em:*

   `(primeiro==segundo=primeiro)`

   e determine o valor que resulta destas combinações. Use `printf()` *para ver o resultado. Solução* `logica02.c`

2. *Construa uma função que faça uso* útil *dos efeitos colaterais obtidos na questão anterior, não se esquecendo da documentação que deixe claro o que faz o programa. Solução* `logica02.c`

3. *Construa um programa que verifique se dois vetores de caracteres fornecidos (duas senhas) são iguais ou diferentes. Use* `strcmp()` *em vez de* `==`*. Solução* `logica03_1.c`*.*

4. *Faça uma nova versão do programa* `logica03_1.c` *usando a função* `compara()` *definida em* `ambiente.h`*.*

5. *Construa uma função em que sempre o teste do* se *seja zero, (falso), qualquer que sejam os valores atribuidos às duas variáveis que se peçam ao usuário.*

   Estaremos enganando o usuário entao ?

6. *Rode e leia* `sonho.c`*. Faça outros programas conversacionais, como* `sonho.c`*, para adquirir prática com*

   `se()/ou_entao.`

   Veja `sonho01.c`*.*

4.2 Múltiplas escolhas.

Para *escolhas* múltiplas `gcc` tem, a estrutura:
`escolha() - switch()`
que vamos estudar neste parágrafo.

As palavras reservadas, em português e em inglês para construir `escolha()` estão abaixo em correspondência:

escolha(variavel)	inicio	caso	valor	pare	fim
switch(variavel)	{	case	valor	break	}

A função `escolha()` recebe uma variável e testa os seus valores contra uma seqüência planejada pelo programador.

O modelo é o seguinte:

escolha(variavel)
inicio
    caso valor1: comando11;... ;pare;
    caso valor2: comando21;... ;pare;
    padrao: comandop1;... ;pare **atenção**
    caso valorn: comandon1;... ;pare;
fim

A palavra-chave `caso` marca o início de um novo teste e de uma nova seqüência de comandos que devem ser executados caso o teste se revele positivo.

Observação: 18 *Uma cascata de execuções*

A palavra-chave `pare` *é essencial no uso desta estrutura, se ela não estiver presente, os casos seguintes serão executados, depois do primeiro teste positivo.*

Observe que você pode fazer uso desta facilidade *quando quiser que depois que uma* opção verdadeira *for identificada, uma seqüência de outras sejam executadas.*

Neste caso elimine o `pare`*, mas não se esqueça de colocar um comentário indicando que deseja que todas as opções sejam executadas portanto eliminou o* `pare`*.*

Observação: 19 *Sugestão:* Arquivos lembrete

Crie um arquivo chamado "00escolha" ou outro nome qualquer sugestivo, por exemplo, "`00switch`*", para reciclar esta estrutura.*

Este método pode economisar um bom tempo de digitação, e vai ajudá-lo na memorização da sintaxe.

Deixe no diretório de trabalho arquivos com os nomes das estruturas de fluxo, depois chame-os e cole-os no ponto adequado quando estiver redigindo um programa.

Você pode usar este método com programas inteiros, é a reciclagem de programas.

Os dois zeros antes do nome têm o efeito de forçar que os seus arquivos-lembrete *fiquem no topo da árvore de arquivos, quando eles foram listados sem uma ordem específica. Também você pode fazer uma listagem dos* arquivos-lembrete *existentes com*

`ls 00* , ou no DOS dir 00*`

Exercícios: 14 *Estudo da função* `escolha() (switch())`

1. *Rode o programa*[8] `logica04.c`*, ele está errado e os erros estão indicados no próprio programa, corrija o programa.* `logica04_1.c`

2. *Volte a rodar* `logica04.c` *e digite o último caso correto, 3. Verifique que os anteriores não foram executados. Portanto* `escolha()` *age sequencialmente até encontrar uma primeira opção correta. Rode várias vezes* `logica04.c` *até entender como funciona o* `switch()` *do* $\mathcal{C}$.

3. *Melhore o programa* `logica04.c`*, acrescentando espaçamento entre as mensagens, letras maíusculas, etc... dê um jeito neste programa que está muito feio.*

4. *limitação de* `escolha()` *Experimente usar* `escolha()` *de forma mais inteligente, ver* `logica05.c`*. Tente dar aos casos um valor lógico.*
   Solução em `logica05_1.c`

Observação: 20 $\mathcal{C}$ *interpretado*

Espero que os autores de `calc` *jamais leiam este livro... porque estou diminuindo o a importância de* `calc`*, injustamente. Verifique isto.*

Se você estiver estudando $\mathcal{C}$ *em um ambiente LinuX, quase certamente existe no sistema um programa chamado* `calc`*. Este programa é uma linguagem de programação que se assemelha ao* $\mathcal{C}$*, mas é* um interpretador de comandos*. Se você digitar, dentro de* `calc`*,*

(3==2)

terá como resultado, imediatamente

$\rightarrow 0.$

Experimente! Abra uma área[9] *de trabalho, (shell), digite* `calc` $<$ *enter* $>$*, e dentro do* `calc` *digite "(3==2)".*

Por que o resultado é zero?

`calc` *é uma linguagem de programação bastante poderosa, mas exclusivamente voltada para Matemática e usa a estrutura lógica de* $\mathcal{C}$*.* `calc` *usa comandos semelhantes ao da linguagem* $\mathcal{C}$*, mas não é* $\mathcal{C}$*, poristo é injusto o título acima.*

`calc` *não é* $\mathcal{C}$*, certamente foi escrita em* $\mathcal{C}$ *como quase tudo que existe nos computadores, mas* `calc` *é uma linguagem independente, interpretada, semelhante ao* $\mathcal{C}$*, que você pode usar para aprender* $\mathcal{C}$.

[8] em `BC` logi*.c, logi4_1.c

[9] supondo que você tem o `calc` instalado...

*"*Interpretada*" significa que ela lê um comando, faz sua análise e avaliação e imprime o resultado ou uma mensagem de erro.*

Outro exemplo, em `calc`*. Digite a seguinte frase típica de* $\mathcal{C}$:

```
if (3==2) printf("esta certo"); else printf("esta errado");
```

e você vai ver impresso na próxima linha,

esta errado.

Com `calc` *você pode testar as expressões da linguagem* $\mathcal{C}$ *quando tiver dúvidas quanto ao valor delas.*

A função `escolha()` tem um uso limitado à verificação de constantes. É excelente na construção de menus, por exemplos. Ver o programa `logica06.c`, que (por acaso) está errado, como exemplo.

Exercícios: 15 *Construção de um menu*

1. *Leia o programa* `logica06.c`*. É um exemplo de como se pode fazer um menu em* $\mathcal{C}$*.*

2. *Rode* `logica06.c` *e corrija os seus erros.*
   Solução `logica06_1.c`

3. *Melhore o programa* `logica06_1.c` *incluindo espaçamentos entre as frases para tornar sua leitura mais agradável.*

4. *Um defeito técnico, na linha em que se pede para digitar as opções, em* `logica06_1.c`*, não deveria haver "mudança de linha". Corrija isto, se já não o tiver feito.*

O programa `logica06.c` será expandido e completado mais a frente.

Para terminar a discussão do `escolha()`, veja que outros 'casos' podem estar presentes e serem ignorados. Por exemplo, no caso de `logica06.c`, podiamos ter incluido algumas novas rotinas de contabilidade, que planejamos produzir no futuro, mas sem incluir no menu a sua listagem.

Como o usuário não sabe de sua existência, não irá escolhê-las e assim poderiamos evitar que aparecesse a mensagem sobre as rotinas ainda não construídas.

Utilidade: fica registrado no programa o que ainda pretendemos fazer, evitamos de dar explicanos que podem ficar sem a devida compreensão pelo público leigo.

Mas o que interessa mesmo aqui é registrar que `escolhe(opcao)` despreza os 'caso's cujos valores não sejam contemplados pela variável `opcao`. Observe, entretanto, que `gcc` aceitaria uma opção fora do menu e portanto a frase guardada para o planejamento apareceria na tela ... claro que tem meios para evitar isto, veja como nos exercícios.

Exercícios: 16 *Testando a capacidade de* `escolha()`

1. *Veja em* `logica06_2.c` *que ficou um "caso" escondido a título de planejamento, mas que ele pode ser executado se 7 for digitado.*

2. *Experimente colocar um* `se()` *evitando que os casos escondidos do planejamento exponham os programadores ao ridículo...*
   Solução `logica06_3.c`

Observação: 21 *Múltiplas escolhas e seus problemas*

A função `switch()` *é uma das múltiplas situações em que a linguagem C é* frouxa *permitindo que aconteçam situações "inesperadas".*

Elas não são, absolutamente, "inesperadas", são erros de avaliação do programador.

Aqui, como em outras tantas situações, lhe pedimos para reler o último parágrafo da introdução, técnicas para programar bem*. Uma dessas técnicas consiste em nunca fazer* grandes programas*, em vez disto, contrua pequenas funções que executem tarefas específicas e cujo aglomerado seja um grande programa. As pequenas funções são fáceis de serem depuradas e situações, como um item indesejado, ficam na frente dos olhos.*

Além de fazer pequenos programas, se possível constituidos de uma única função, (projeto difícil...), comentários bem claros indicando possíveis cascas de banana *devem ser incluídos.*

Mais! Um programa contstituído de uma única funcao*, certamente, é candidato a virar item de biblioteca.*

Quando você constrói uma função deve cuidadosamente analisar as possíveis situações em que ela seria um risco e deixar isto registrado com um comentário.

Exercícios: 17 `if-else - switch`

1. *Um estacionamento de aeroporto tem as seguintes regras para cobrança:*

   - *A taxa mínima é $2,00 pelas duas primeiras horas, não importanto se o usuário fique menos de duas horas.*
   - *A cada hora que se passar, além das primeiras duas horas, o usuário deve pagar $0,50 por hora, até o limite de 24 horas;*
   - *Para estacionamentos que passem de 24 horas a cobrança passa a ser feita por dia a razão de $13,00 por dia ou fração de dia.*

   Faça o programa para o cálculo das tarifas do estacionamento.
   Solução: `estacionamento.c`

2. *Melhore o programa* `estacionamento.c` *tornando mais justo a cobrança quando o tempo ultrapassa 24 horas levando em consideração fração de dia também.*

4.3 enquanto() while()

Laços são um tipo de estrutura de fluxo de dados em que *alguma coisa acontece* durante *algum tempo*. Por exemplo,

```
enquanto (estiveres na minha frente)
                eu vou te olhar nos olhos;
```

Claro que vou deixar de *lhe olhar nos olhos* quando o *objeto* já não mais estiver *na minha frente*.

Vamos ver alguns exemplos menos sublimes e bem mais práticos.

Aproveitando a oportunidade, `enquanto()` sempre se escreve com letra minúscula, mesmo no início de uma frase... com letra maiúscula `gcc` não entenderia

`enquanto(), while().`

Claro que você pode alterar isto no arquivo `traducao.h`. Mas se, no Brasil, estabelecermos um padrão para programação em português, todos deveremos adotar o mesmo padrão. Por enquanto programar em português é uma experiência.

Dizemos que `gcc` é sensível à diferença entre letra maíscula e mínuscula. Portanto `enquanto()` é diferente de `Enquanto()`, e `Enquanto()` não existe em `traducao.h`.

Mas você pode alterar isto, re-escrevendo o arquivo `traducao.h` e nele alterando

`enquanto ---> Enquanto.`

e neste caso `gcc` só entenderia `Enquanto().`

A função `enquanto()` analisa uma expressão e, se ela for *verdadeira*, executa o comando que vem em seguida. Por exemplo, o seguinte algoritmo calcula a soma dos 10 primeiros numeros inteiros estritamente positivos (a partir de 1). Veja os comentários logo a seguir.

```
(1) soma = 0
(2) i = 1
(3) enquanto( i < 11)
(4)     inicio
(5)                 soma = soma + i
(6)                 i = i + 1
(7)     fim
```

```
(1) soma = 0
(2) i = 1
(3) while( i < 10)
(4)         {
(5)                 soma = soma + i
(6)                 i = i + 1
(7)         }
```

Comentários:

1. Inicializa a variável `soma` com zero; Aqui estamos começando a discutir processamento de dados... `soma` é um `buffer`, um local onde se armazenam dados para uso posterior. Alguns diriam que `soma` é um acumulador. Os valores a serem somados vão ser *acumulados* nesta variável, logo ela tem que ser inicializada com zero. Se fossemos fazer multiplicações sucessivas, a inicialização do `buffer` se faria com 1.

Porque, zero é o elemento neutro da adição, e um é o elemento neutro da multiplicação.

2. inicializa a variável `i` com 1, porque é um contador nós, normalmente, contamos de um em diante. Não se esqueça de que $\mathcal{C}$ conta de zero em diante... dizemos que $\mathcal{C}$ é de base 0. Consequentemente este programa não está otimizado, está perdendo espaço na memória.

3. Teste, verifica se `i` passa de 10, quando isto acontecer encerra o laço. A variável `i` é um contador.

4. `inicio`, marca o começo de um bloco lógico. Em $\mathcal{C}$ é a chave-abrindo: {.

5. Incrementa `soma` com o valor de `i`;

6. Incrementa `i` de uma unidade.

7. `fim`, marca o ponto final de um bloco lógico. Em $\mathcal{C}$ é a chave, }, fechando que termina os blocos.

Uma das dificuldades iniciais em computação está aqui presente nos itens (5) e (6). Não se tratam de igualdades matemáticas ou equações.

O operador "=", em algumas linguagens de computador, significa uma "atribuição". Há linguagens de computador que têm um símbolo diferente para este operador, em algumas é uma seta, em outras, como Pascal, Maple, MuPAD, é o símbolo

:=.

em $\mathcal{C}$	em Pascal
`soma = soma + i;`	`soma := soma + 1;`

Assim

`soma = soma + i`

é "um comando" que se constitue da composição de dois operadores:

- primeira operação a ser executada `soma + i` que adiciona o valor de `i` ao valor de `soma`;

- segunda operação a ser executada `soma = soma + i` que atribue o valor calculado anteriormente, "`soma + i`", à `soma`, alterando, assim, o valor guardado em `soma`.

Quer dizer que o laço descrito acima vai ser executado 10 vezes,

`enquanto( i < 11)`:

O quadro seguinte mostra, a cada passo, qual é o valor de `soma`, do acréscimo e do valor atualizado de `soma`.

no.oper.	soma	acréscimo	soma atualisada	status
1	0	1	1	CONTINUA
2	1	2	3	CONTINUA
3	3	3	6	CONTINUA
4	6	4	10	CONTINUA
5	10	5	15	CONTINUA
6	15	6	21	CONTINUA
7	21	7	28	CONTINUA
8	28	8	36	CONTINUA
9	36	9	45	CONTINUA
10	45	10	55	PARA

Observe que este laço calcula a soma dos termos de uma progressão artimética de razão 1, desde o primeiro termo 1 até o termo 10.

Exercícios: 18 *1. Escreva um programa que imprima os termos de uma p.a. dados o primeiro termo, a razão e o número de termos pelo teclado. Solução* `logica07.c`

2. Melhore o programa `logica07.c` *criando mensagens e diálogos com o usuário. Solução* `logica07_1.c`

3. Melhore programa `logica07.c` *para que ele imprima os dados da p.a. em linhas separadas. Acrescente também linhas separadoras com*

"_____"

4. Use a função `limpa_janela()` *para limpar o terreno e tirar a poluição visual da tela. A função* `limpa_janela()` *está definida na biblioteca* `ambiente.h`.

5. Use a função `apeteco2()` *para provocar paradas na execução do programa e permitir que o usuário leia melhor o conteúdo das mensagens, prosseguindo quando lhe parecer adequado. A função* `apeteco()` *está definida na biblioteca* `ambiente.h`. *Verifique que há vários tipos de* `apeteteco()`.

6. Re-utilize programa `logica07.c` *para calcular os termos de uma p.g.*

7. Re-utilize programa `logica07.c` *para calcular as atualizações de sua poupança. Faça com o programa crie uma tabela para lhe mostrar a defasagem da poupança relativamente ao crescimento do dolar, por exemplo.*

8. Faça um programa que compare o que acontece, se você colocar 100 reais na poupança ou pegar 100 no cheque especial. Use `imprima()` *assim*

```
imprima(''este mes na poupanca %f ou no cheque especial %f\n'',
poupanca, cheque);
```

observe que as variáveis terão que ser do tipo `real` *ou em C,* `float`. *Sobre tudo na poupança em que o incremento é de* 0.05% *ao mes...*

Observe que `logica07_1.c` é segunda versão de `logica07.c`. O primeiro, `logica07.c`, cabe inteiramente dentro da tela, sem os comentários. Ele foi cuidadosamente depurado. Depois, na segunda versão, foram incluidas mensagens explicativas ou auxiliares para conduzir o usuário a fornecer os dados.

Como `logica07.c` é pequeno e cabe inteiramente na tela, facilmente pudemos encontrar os erros cometidos e corrigí-los todos.

Depois que um programa estiver funcionando, sem erros, podemos fazer-lhe uma segunda versão incluindo

- enfeites;
- mensagens de comunicação com o usuário;
- incluir os comentários.

Tudo isto é necessário, mesmo que o programa fique um pouco maior do que a tela.

Agora estamos melhorando um programa que já funciona, mas teremos controle da situação. Muitas vezes é esta hora de incluir comentários porque o programa está funcionando e já concluimos os truques para que ele ficasse menor (os efeitos colaterais).

Exercícios: 19 *Mas, por* `enquanto()`

1. *Observe que se um laço iniciar com*

   `enquanto(1)`

   ele certamente nunca irá parar sem uma ação mais forte (Ctrl-C acionado no teclado...).

   Use isto para alterar o programa `estacionamento.c` *de modo que o operador não precise voltar a acionar o programa a cada cliente que sai. E como o estacionamento é eterno o programa ficaria para sempre no ar.*

   Chamamos isto de loop infinito *porque é um laço que nunca para de ser executado. Um programa que deve ficar sempre no ar, geralmente, é gerenciado por um* loop infinito *que deixa na tela o menu de opções do programa.*

   Em `BC` *evite fazer isto. Primeiro aprenda a ligar nas* `options` *o acesso ao* `Ctrl-C` *que pode estar desligado. Em* `Linux` *é fácil parar um programa que esteja* entalado*... Em* `BC` *veja como fazer:*

   - *Abra um programa no editor, qualquer um, por exemplo*

     `estacionamento.c`

     e escreva `Ctrl` *deixando o cursor sobre esta palavra;*
   - *Clique no* `help` *e escolha* `topic search`*.*
   - *Você vai parar em* `ctrlbrk`*. Dê enter;*
   - *Você será conduzido à pagina das interrupções. Faça uma cópia do exemplo que esta ao final da página -*

- *Copie para* `estacionamento.c` *o trecho*

```
#define ABORT 0

int c_break(void)
{
   printf("Control-Break pressed.  Program aborting ...\n");
   return (ABORT);
}
```

Agora você pode criar um `enquanto(1)` *que o programa irá parar ao você apertar* `ctrl c`.

Solução `estacionamento01.c`. *Fora do contexto. Usa funções de tratamento do tempo. Ignore, inicialmente, a função* `ConverteTempo()`. *Use-a. Este programa deve ser compilado em* `Linux` *com*

gcc -Wall -oprog -lm estacionamento.c

a opção fará com que a biblioteca `math.h` *seja usada.*

2. *Melhore* `estacionamento01.c` *uma vez que há uma tremenda poluição visual. Coloque*

   `apeteco2(), limpa_janela()`

   em locais adequados para melhorar o visual do programa.

3. *Se você já estiver estudando integrais, facilmente pode transformar programa* `logica07.c` *num programa que calcule integrais. Se não estiver estudando integrais, esqueça esta questão, por enquanto...*

   Solução `riemann04.c`

   esqueça por enquanto as outras versões de `riemann.c` *que serão discutidas mais a frente.*

4. *Melhore o programa* `logica06_3.c` *substituindo o* `se()` *por*

   `enquanto(opcao > 6)`

   Solução `logica06_4.c`.

5. *Como* `logica06_4.c` *suja, irremediavelmente, a tela, use a função*

   `limpa_janela()`

   definida em `ambiente.h` *para melhorar a performance do programa, frente aos clientes mais exigentes. Se inspire em* `prog04_2.c` .

   Solução `logica06_5.c`

6. *Estude a função* `limpa_janela()` *definida na biblioteca* `ambiente.h` *para saber o que ela fez no programa anterior. Estude as funções definidas em* `ambiente.h` *para entender o funcionamento de uma biblioteca.*

7. *Coloque uma mensagem* de sucesso *no programa* `logica06_5.c` *usando a função* `sucesso()` *definida em* `ambiente.h`. *Altere a mensagem, como alí se sugere, para atender ao seu gosto.*

Você foi conduzido a estudar a biblioteca `ambiente.h`. Quase todas as funções da linguagem $\mathcal{C}$ se encontram definidas em alguma biblioteca. A linguagem mesmo, sem as bibliotecas é mínima. Experimente apagar `# include <stdio.h>` que se encontra em todos os programas e depois compilá-lo, veja o que acontece.

4.4 Outro método para laços.

> Um método alternativo de fazer laços é o `para()` (`for()`):
> `para(`$i = 0; i < 10; i = i + 1$`)`
> `imprima("o termo %d da sucessao é --> %d",`$i + 1, 2 * i + 3$`);`
> que vamos estudar neste parágrafo.

O efeito da expressão

```
para(i=0; i<10; i=i+1)
imprima("o termo %d da sucessao eh --> %d",i+1,2*i+3);
```

é o de imprimir sucessivamente os termos de uma sucessão, veja o resultado obtido com `calc` .

```
for(i=0; i<10; i=i+1)
printf("o termo %d da sucessao eh  --> %d\n",i+1,2*i+3);
o termo  1 da sucessao eh --> 3
o termo  2 da sucessao eh --> 5
o termo  3 da sucessao eh --> 7
o termo  4 da sucessao eh --> 9
o termo  5 da sucessao eh --> 11
o termo  6 da sucessao eh --> 13
o termo  7 da sucessao eh --> 15
o termo  8 da sucessao eh --> 17
o termo  9 da sucessao eh --> 19
o termo 10 da sucessao eh --> 21
```

Exercícios: 20 *Rode usando* `calc`

1. *Rode a expressão abaixo com* `calc`

   ```
   for(i=0; i<10; i=i+1)
   printf("o termo %d da sucessao eh --> %d\n",i+1,2*i+3);
   ```

Basta chamar `calc`*, marcar o texto acima com o ratinho, colá-lo na tela do* `calc` *e dar um <enter> para ver o resultado que está apresentado acima. Nos fizemos isto mesmo, depois colamos o resultado do* `calc` *aqui.*

2. *Porque imprime até o 10º e não apenas até o 9º?*

3. *Usando* `calc` *apresente os termos de uma progressão aritmética de razão 0.5 com primeiro termo* -3.

   Solução:

```
> for(i=0; i<10; i=i+1)
>> printf("o termo %d da sucessao   --> %f\n",i+1,0.5*i-3);
```

Observe as diferenças entre `enquanto()` e `para()`. Todos os dados necessários ao funcionamento de `para()` devem ser fornecidos como parâmetros:

- o valor inicial do contador;
- o teste para finalizar o processo;
- o método para fazer incrementos.

em suma, `para()` exige tres parâmetros.

Em $\mathcal{C}$ existe um atalho para fazer incrementos do tipo $i = i + 1$:

```
for(i=0; i<10; i++)
printf("o termo %d da sucessao eh --> %d\n",
 i+1,2*i+3);
```

quer dizer que as duas expressões

```
i = i+1 ; i++
```

são equivalentes. Há vários atalhos destes tipo reconhecidos pelo `gcc` que vamos reunir no apêndice. Ao fazê-lo estamos claramente lhe dizendo que o seu uso deve ser feito sob restrição. Obviamente que o atalho será executado com mais rapidez, mas o programa ficará mais dificil de ser lido.

Os exercícios lhe darão mais experiência no uso de `para()` que mil palavras.

Exercícios: 21 *Mais loops*

1. **Alerta** *Antes de fazer este exercício, rodando* $\mathcal{C}$ *dentro do windows, veja no manual como ativar o* `ctrl-c`*. Veja também, no índice remissivo* Ctrl-c*. Nos ambientes integrados existem uma opção para configurar o* $\mathcal{C}$ *e ativar o* `ctrl-c` *que vem desativado para dar maior rapidez ao compilador. Isto pode ser feito com uma instrução de compilação, e esta forma de fazer é melhor porque não altera a configuração do* ambiente integrado*.*

   Experimente compilar e rodar o programa `logica06_6.c`*. Ele não para nunca, a não ser que você o pare com* `ctrl-c`*. Leia o programa e analise porque isto acontece. Verifique o primeiro* `enquanto()`*.*

2. *Altere* `logica06_6.c` *trocando*

   `enquanto(1) --> enquanto (t)`

   e definindo `t` *como inteiro em linha anterior, inicializando com o valor* $t = 1$. *Compile e rode o programa. A única maneira de pará-lo é com* `ctrl-c`, *novamente.*

3. *Agora troque t por opcao e veja que o programa para digitando* 0 *como opção não solicitada no menu. Experimente! Veja* `logica06_61.c` *e leia ao final dos comentários.*

4. *Corrija o programa* `logica06_61.c` *para o menu informar que a saída do programa se dá com*

   `opcao = 0`

   solução `logica06_62.c`

5. *Corrija agora o programa* `logica06_5.c` *para que ele fique no ar indefinidamente até que* `opcao = 0` *seja escolhida.*

   Solução `logica06_63.c`

6. *Corrija agora o programa* `logica06_5.c` *incluindo no menu a opção* 0 *para que o sistema pare.*

   Solução `logica06_71.c`

7. *Porque não teria sentido resolver os exercícios deste bloco usando* `para()`*?*

8. *Traduza* `logica06_71.c` *para C.*

   Solução `logica06_7.c`

4.5 Parando no meio de um bloco.

Vamos estudar as funções
`pare, (break), voltar, (return)`
nesta seção.

Com frequência, dentro de um laço, precisamos de interrompê-lo, abruptamente, sem que as demais funções sejam executadas. Quem faz isto é a função `pare`, (`break`), a mesma que estudamos junto com `escolha()`, (`switch()`).

Ao encontrar esta função, `gcc` encerra o processo que estiver sendo executado dentro de um bloco e passa o controle do fluxo para a próxima função imediatamente após este bloco. É isto que ocorre com `escolha(), switch()`.

A figura (fig. 4.6), página 95 mostra como isto se passa.

Veja uma situação-padrão:

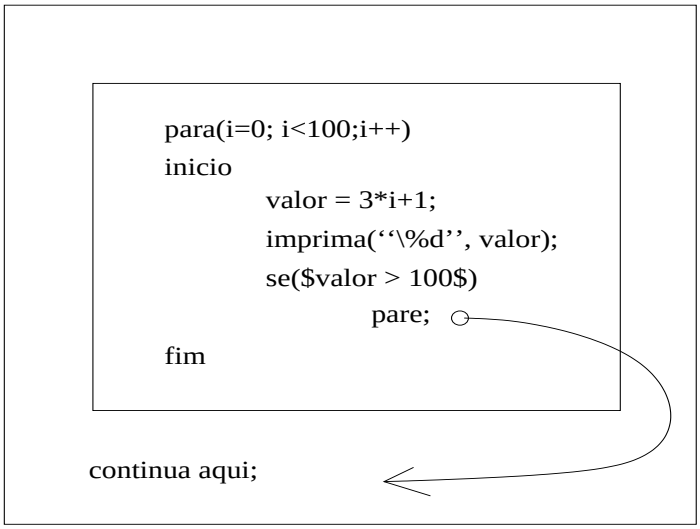

Figura 4.6: Ao encontrar `pare()` o fluxo é desviado para a próxima função externa ao bloco.

```
para(i=0; i < 100;i++)
        inicio
                valor = 3 * i + 1;
                imprima("%d", valor);
                se(valor > 100)
                        pare;
        fim
imprima("%d",i);
```

Vai sair do laço quando `i = 34` sem executar os comandos internos para este valor de `i`. Este laço resolve a equação (desigualdade)

$$3x + 1 > 100.$$

Exercícios: 22 *Parando para resolver desigualdades*

1. *Verifique que laço acima resolve resolve a desigualde:*

   $$3 * i + 1 > 100$$

   no conjunto dos inteiros, imprimindo 34, o primeiro número inteiro `i` *que a torna verdadeira. Solução* `logica08.c`.

2. *Altere o programa que rodou o laço acima para resolver a desigualdade*

   $$3 * i + 1 > 100$$

   num intervalo de números racionais, obviamente sem apresentar todas as soluções, por que?

3. *Melhore* `estacionamento01_1.c` *permitindo que o técnico em computação pare o programa para fazer manutenção no micro.*

   Solução: `estacionamento01_2.c`

4. *Em* `estacionamento01_2.c` *usamos um variável do tipo*

   `vetor de caracteres`

   com tamanho 1. *Talvez fosse mais simples usar variável do tipo caracter,* `estacionamento01_3.c` *que está errado. Rode, veja as mensagens de erro e procure entendere porque não funciona. "Solução" nos comentários de* `estacionamento01_3.c` .

5. *Melhore o programa* `logica08.c` *incluindo mensagens adequadas de comunicação com usuário.*

6. *Escreva todos os comentários necessários à boa compreensão do programa* `logica08.c`.

7. *Resolva as seguintes desigualdades, usando programas em* $\mathcal{C}$.

   (a) $3k + 2 > -100; k \in \mathbf{Z}$;

   (b) $-10 < 3k + 2 < 200; k \in \mathbf{Z}$;

   (c) $-10.5 < 3x + 2 < 20.7; x \in \mathbf{R}$;

   Solução: logica08_1.c; logica08_2.c; logica08_3.c Para resolver a última desigualdade você vai precisar de usar `real` *em vez de* `inteiro` *como tipo de dado. Veja o capítulo 5 a respeito de tipos de dados.*

8. *Melhore os programas* `logica08_X.c` *escrevendo comentários e incluindo os monólogos de comunicação com o usuário, paradas, limpezas de tela etc...*

9. *Melhore os programas* `logica08_X.c` *fazendo que saiam deixando a tela limpa, mas permitindo que o usuário consiga ler os dados, antes da limpeza ser efetuada.*

10. *Altere o programa* `logica06_63.c` *para que o sistema saia do ar quando a opção 0 for escolhida, mas usando* `pare` *(*`break`*), em vez de usar a opção 0 no* `enquanto()`. *Solução* `logica06_64.c`.

No programa `logica08_3.c` fizemos uso dos atalhos `k-=0.01, k+=0.01` que `gcc` entende respectivamente como `k=k-0.01, k=k+0.01`.