Capítulo 8

Matemática em $\mathcal{C}$

Neste capítulo, o primeiro parágrafo é um breve guia de referência da linguagem $\mathcal{C}$ específico para o assunto do capítulo, onde você poderá encontrar rapidamente informações sobre o uso das ferramentas da linguagem.
Aqui operadores aritméticos e lógicos.

Podemos fazer muita matemática com $\mathcal{C}$ e vamos deixar-lhe uma pequena amostra do que é possível fazer.

Em $\mathcal{C}$ podemos fazer programas para determinação de códigos, ou melhor, para validação de códigos.. Os programas que executam estas tarefas são bastante complicados porque fazem operações de comparação com vetores de caracteres, uma pequena amostra disto se encontra no programa `sexto06.c`, usando

```
grep senha *.c
```

você pode encontrar outros programas nossos que dão exemplos de comparação entre palavras. Mas códigos são mais do que senhas, e pode ser uma aritmética muito pesada, não vai ser possível desenvolver tais programas aqui.

Vamos apresentar-lhe uma *matemática* mais simples:

- como calcular aproximadamente integrais;
- o método para calculo de integrais que vamos usar chama-se *somas de Riemann*, não é o melhor nem o mais rápido, tem coisa melhor, mas os outros partem deste, como se tratam de somas, antes aprenderemos a fazer somas;
- para fazer somas, precisamos de varreduras, antes aprenderemos a fazer varreduras que também se encontra no caminho para fazer gráficos.

É o que faremos aqui.

8.1 Operadores aritméticos e lógicos

Este capítulo se dedica à aritmética em $\mathcal{C}$ mas é impossível escrever um programa sem usar operadores lógicos, vamos, portanto, misturar os operadores aritméticos e lógicos neste contexto.

Aritmética e lógica

Falamos de "operadores" na linguagem $\mathcal{C}$ de modo semelhante como se diz em Matemática, "operação":

- em Matemática escrevemos "$a + b$" e dizemos que "a operação de adição entre os números a, b. Você executou uma "operação" matemática.

- em Computação dizemos que o operador "+ se aplica aos números $3, 4$

$$(3, 4) \to 3 + 4;$$

  e escrevemos apenas $3 + 4$.

Mas quando você, num programa, executar

$$x = 3 + 4;$$

não é uma operação matemática que estará sendo executada e sim

- primeiro a operação matemática $3 + 4$, porque as sentenças, em $\mathcal{C}$, são executadas da direita para a esquerda, como em japonês, e de dentro para fora, se houver parenteses;

- depois foi executada uma operação interna de registro de dados, guardando na variável x o valor calculado anteriormente

- Previamente a variável x deve ter sido definida;

- agora o número inteiro 7 está sendo sendo associado à variável x.

Em Matemática, e porque os humanos são inteligentes e conseguem deduzir a *confusão de dados a partir do contexto*, confundimos duas operações:

1. Associação de dados, que é feito em computação em $\mathcal{C}$ com o símbolo

$$=$$

2. Teste lógico, que em Matemática se usa o mesmo símbolo, mas em $\mathcal{C}$ se usa o símbolo ==

Em Matemática, a expressão

$$3x + 4 = 0$$

é uma sentença dita "aberta", e com isto se quer dizer, podemos testar se um determinado valor dado à variável x torna esta sentença *verdadeira* ou *falsa*. Em $\mathcal{C}$ não teria sentido escrever esta expressão sem primeiro atribuir um valor a x. Em suma, em $\mathcal{C}$ podemos fazer:

$$x = 10; \tag{8.1}$$

$$(3x + 4 == 0) \tag{8.2}$$

para verificar se é verdadeira para o valor 10 dado a x. Observe os parenteses, eles são um operador da linguagem $\mathcal{C}$, o operador avaliação, eles forçam o cálculo do seu conteúdo.

Veja o programa `logica08.c` a este respeito. Verifique isto com "calc", por exemplo[1], execute em "calc" as seguintes linhas:

```
x=3
3*x+4==0
```

Depois de digitada a última linha, "calc" responderá com "0" porque terá verificado que à direita e à esquerda de == se encontram objetos diferentes:

$$13 = 3x + 4 \neq 0 \quad \Rightarrow \quad 3x + 4 == 0 \text{ é falso.}$$

Em Matemática estas diferenças sutís têm que ser determinadas a partir do contexto pelos humanos. Em $\mathcal{C}$, cujos programas são escritos para as máquinas, estes detalhes têm que ser cuidadosamente diferenciados.

O exemplo acima mostra que não será possível falarmos de *aritmética* sem mencionar a lógica. Embora este capítulo esteja dedicado à aritmética, teremos que mencionar os operadores lógicos.

8.1.1 Uma lista seca de operadores

quase seca...

Em $\mathcal{C}$ existem alguns "operadores", que vamos descrever detalhadamente abaixo incluindo exemplos. Se você tiver chegado até aqui "linearmente", muitos deles já serão seus conhecidos, porque você os encontrou nos programas.

1. $\boxed{!}$ É operador lógico, "not". O não lógico;
2. $\boxed{!=}$ É negação do operador lógico "igual". O "diferente de";
3. $\boxed{=}$ não é um operador aritmético, é um "comando da linguagem $\mathcal{C}$ algumas vezes identificado como "atribuição". Sobre tudo observe que é diferente de == que aparece mais abaixo.

[1] na ausência do `calc`, faça um pequeno programa contendo somente estas linhas, reserve `teste.c` para isto

4. [%] O operador "mod", é a divisão inteira. As duas formas abaixo são equivalentes':

$$r = mod(a, b) \equiv r = a\%b;$$

tem como resultado o resto na divisão de a por b.

```
int  resto(int a, int b)
{
 printf("O resto na divisao de %d por %d e' %d ",a,b,mod(a,b))
 return 0;
}
```

```
resto(4,6) -> O resto na divisao de 4 por 5 e' 4.
```

Há diversas aplicações para esta função, uma delas na validação de códigos, como CPF em que o "dígito" *pode* ser o resto na divisão do número formado pelos outros algarismos por um quociente escolhido. Assim se pode verificar se um número apresentado como CPF tem legitimidade ou não. O quociente, o segundo parâmetro em `mod(a,b)`, deve ser grande, em geral, para aumentar a segurança, ou diminuir a chance nas tentativas falsas.

5. [/] A divisão comum se faz com o operador / e ainda há uma divisão inteira que com o operador //. Assim, o resultado do seguinte *teste* será 1

6. [//]

$$(8//5) * 5 + mod(8, 5) == 8$$

8//5 calcula o quociente na divisão de 8 por 5 e mod(8,5) calcula o resto nesta divisão. A expressão do *teste* é a equação da divisão euclidiana:

$$divisor * quociente + resto = dividendo.$$

Se você copiar esta expressão para a folha do `calc`, ao dar `enter` terá como resposta:

1

porque a avaliação lógica entre as duas expressões resultou em `verdadeiro`.

7. [*] O operador multiplicação;

Este operador é da multiplicação em $\mathcal{C}$ como na maioria das linguagens de programação.

Como a adição, se parece com a multiplicação da Matemática.

Se parece porque em Matemática, sempre podemos calcular $a * b$ dados dois números. Em $\mathcal{C}$, depende do limite na memória do computador.

Este seria um dos diferenciadores, $\mathcal{C}$ foi projetado para trabalhar com números inteiros dentro de uma faixa de grandeza. Além desta faixa de grandeza os resultados voltam a se repetir ciclicamente.

Por, exemplo, apenas para podermos exemplificar com valores pequenos, suponhamos que o $\mathcal{C}$ com que você estivesse trabalhando se encontrasse limitado aos inteiros de -9 até 9. Neste caso,

$$3 * 2 = 6;$$
$$3 * 5 = (15 - 9) = -4;$$
$$3 * 6 = 0$$

Porque este *seu* $\mathcal{C}$ faria as contas e usaria o "noves fora" para dar a resposta, mas aceitando resultados entre -9 e 8.

8. $\boxed{+}$ O operador adição;

   Este é operador de adição valendo os mesmo comentários que fizemos com respeito à multiplicação comparada com esta operação na Matemática.

9. $\boxed{++}$ É o operador incremento. Fica definido pela seguinte identidade:

   $$(a++) \text{ é idêntico a } (a = a + 1)$$

   Duas variantes deste operador: $=+$, $=-$ que exigem um parâmetro numérico.

   - $\boxed{=+}$ Sintaxe: $x{=}+numero$ equivale a $x = x + numero$
   - $\boxed{=-}$ Sintaxe: $x{=}-numero$ equivale a $x = x - numero$

   Observe que se "float x", $\mathcal{C}$ prossegue somando uma unidade a x.

10. $\boxed{-}$ O operador subtração;

    Este é operador de subtração valendo os mesmo comentários que fizemos com respeito à multiplicação comparada com esta operação na Matemática.

11. $\boxed{--}$ O operador decremento;

    É semelhante ao ++ para fazer subtrações.

    Sintaxe $x--$ equivale a $x = x - 1$ Ver $=-$.

12. $\boxed{->}$ Operador para acessar um elemento de uma classe ou de uma estrutura;

13. $\boxed{/}$ O operador divisão ;

    Este é operador de <u>divisão inteira</u> quer dizer que a/b calcula apenas o quociente na divisão entre os dois números inteiros a, b. Se um dos números for do tipo `float` o operador imediatamente se acomoda ao tipo mais amplo, float tendo como resultado o número racional $\frac{a}{b}$.

    Além disto valem os comentários que fizemos com respeito à multiplicação comparada com esta operação na Matemática.

14. $\boxed{<}$ Operador lógico "menor do que" ;

    É um operador lógico, quando $\mathcal{C}$ avalia $a < b$ o resultado será 0 *ou* 1 conforme esta expressão seja falsa ou verdadeira.

15. $\boxed{<<}$ Operador "shift" à esquerda ; Você vai precisar deste operador quando seus programas se tornarem mais sofisticados. Serve para agilizar operações aritmética e lógicas.

    Se você conhece (viu) aquelas antigas máquinas de calcular com manivela e duas orelhas que permitiam correr o carro para direita ou para esquerda, então você tem aqui um exemplo computacional... << corre o carro para esquerda. Por exemplo, $4 \equiv 100$ à direita a expressão deste número em binário. $4 << 1 \equiv 100 << 1 = 1000 \equiv 8$ aumenta uma casa "positional" no número (em seu formato binário).

    Exemplos:

$$
\begin{aligned}
3 \equiv 11 \;\Rightarrow\; & 3 << 1 \equiv 11 << 1 = 110 \equiv 6 \\
4 \equiv 100 \;\Rightarrow\; & 4 << 1 \equiv 100 << 1 = 1000 \equiv 8 \\
8 \equiv 1000 \;\Rightarrow\; & 8 << 1 \equiv 1000 << 1 = 10000 \equiv 16 \\
7 \equiv 111 \;\Rightarrow\; & 7 << 1 \equiv 111 << 1 = 1110 \equiv 14 \\
5 \equiv 101 \;\Rightarrow\; & 5 << 2 \equiv 101 << 2 = 10100 \equiv 20 \\
7 \equiv 111 \;\Rightarrow\; & 7 << 2 \equiv 111 << 2 = 11100 = \\
& 10000 + 1000 + 100 \equiv 16 + 8 + 4 = 28
\end{aligned}
$$

    Nas equações acima estamos usando $\equiv$ para traduzir, nos dois sentidos, de decimal para binário ou vice-versa.

    Veja que em $a << b$, exatamente, o carro anda b casas "posicionais" para esquerda, sempre na expressão binária do número a.

    Observação: 35 *Para que servem as operações binárias*

    São operações de "baixo nível" e consequentemente muito rápidas. Os dois operadores "shift" (translação) podem ser utilizados para multiplicar ou dividir. São usados em cálculo lógicos também uma vez que eles se aplicam aos caracteres.

16. $\boxed{>>}$ Operador "shift" à direita ; Você vai precisar deste operador quando seus programas se tornarem mais sofisticados. Serve para agilizar operações aritmética e lógicas.

    Aqui você tem o operador que corre o carro para a direita. Consequentemente ele é destrutivo (o que não acontece com as máquinas de balcão em que simplesmente o carro não corre além do limite físico).

$a >> b$ corre o "carro"para direita, o número de casas indicadas pelo parâcimetro b, até "consumir"a.

Por exemplo, $4 \equiv 100$ à direita a expressão deste número em binário. $4 >> 1 \equiv 100 >> 1 = 10 \equiv 2$ aumenta uma casa "positional" no número (em seu formato binário).

Exemplos:

$$\begin{aligned} 3 \equiv 11 &\Rightarrow 3 >> 1 \equiv 11 >> 1 = 1 \equiv 1 \\ 4 \equiv 100 &\Rightarrow 4 >> 1 \equiv 100 >> 1 = 10 \equiv 2 \\ 8 \equiv 1000 &\Rightarrow 8 >> 1 \equiv 1000 >> 1 = 100 \equiv 4 \\ 7 \equiv 111 &\Rightarrow 7 >> 1 \equiv 111 >> 1 = 11 \equiv 3 \\ 5 \equiv 101 &\Rightarrow 5 >> 2 \equiv 101 >> 2 = 1 \equiv 1 \\ 7 \equiv 111 &\Rightarrow 7 >> 2 \equiv 111 >> 2 = 1 \end{aligned}$$

Nas equações acima estamos usando $\equiv$ para traduzir, nos dois sentidos, de decimal para binário ou vice-versa.

Veja que em $a >> b$, exatamente, o carro anda b casas "posicionais" para esquerda, sempre na expressão binária do número a.

17. $\boxed{<=}$ Operador lógico "menor ou igual a";

    Associa à expressão $a <= b$ um dos valores $0, 1$ conforme ela seja verdadeira ou falsa considerando os dados guardados nas variáveis a, b.

    Algumas vezes os programadores são tentados a deixar dentro dos programas "*chaves*" secretas que eles conhecem. É uma forma de garantir que ninguém venha a substituí-lo na manutenção de um programa.

    Veja esta forma *maldosa* de atribuir zero a uma variável

    $$a = (5 <= 3);$$

    Como $5 <= 3$ será avaliada como `falsa`, o resultado, depois de calculados os parênteses, é zero que será atribuido à variável a. Esta linha perdida entre uma centena de linhas, é uma chave secreta violenta.

    Péssimo exemplo, não siga. Mas eventualmente ele pode ser útil, neste caso coloque comentários dizendo o que está fazendo.

18. $\boxed{==}$ Operador lógico "igual";

    Diferente de $=$.

    $a == b$ serve para testar se a é igual a b.

19. $\boxed{>}$ Operador lógico "maior do que" ;

    Releia $<=$ substituindo este por $>$ com a leitura adequada....

20. >= Operador lógico "maior ou igual a";
    Releia <= substituindo este por >= com a leitura adequada....

21. ? : Condicional dentro de uma expressão;
    De mau estilo, torna o programa ilegível. Veja o segmento de programa:

$$a = 3;$$
$$b = 7;$$
$$a > b?\ \text{imprima("verdade"):imprima("falso");}$$
$$falso$$

    ou ainda
    $a > b?$ `imprima("verdade"):imprima("falso");`
    Equivale a "`se() ou_entao`", portanto use o "`se() ou_entao`".

22. ^ O operador bit-lógico "xor" (ou exclusivo);

23. { } Chaves - para delimitar blocos lógicos de um programa;

24. | O operador bit-lógico "or" (ou) ;

25. || O operador lógico "or" (ou) ;

26. ~ O operador bit-lógico "não" (calcula o complemento de todos os bits, onde estiver 0 coloca 1 e vice versa.) ;

8.2 Equação do segundo grau

Vamos analisar a sucessão de programas prog6101.c ... prog6104.c que resolvem equações do segundo grau. O programa prog6101.c é programa que foi feito inicialmente sendo os demais evoluções que ele sofreu. O prog6104.c é o "top de linha"sendo um programa modularizado e com um lay-out agradável para o usuário. A recomendação é que você primeiro rode os programas e depois os estude.

Observação: 36 *Compilando programas matemáticos*
Se você compilar com a linha de comandos:

```
gcc -Wall prog6101.c -oprog
```

o compilador vai emitir uma mensagem de erro dizendo desconhecer a função `pow`*. Isto acontece porque as funçõe matemáticas implementadas pela* `Fundação Gnu` *sofreram uma compilação particular que as torna mais rápidas sendo necessário fazer referência a um tipo particular de acesso à biblioteca matemática:*

```
gcc -Wall -lm prog6101.c -oprog
```

com a instrução de compilação −lm Experimente os dois modos de compilação para ver o resultado.

Vamos discutir aqui o mais simples dos programas apenas, o primeiro deles.

Um programa que resolva equações do segundo grau deve, sequencialmente, efetuar as seguintes operações:

- Diálogo inicial Mostrar o formato da equação para estabelecer uma linguagem de comunicação (qual é o coeficiente do termo do segundo grau, do primeiro grau e do termo independente);

  ```
  imprima("ax^2 + bx + c = 0 \n ");
  imprima("Forneca-me os coeficientes -> a, b, c \n");
  ```

- Entrada de dados Ler os coeficientes que o usuário irá digitar pelo teclado;

  ```
  leia(deposito, tamanho_do(deposito), entrada_pdr);
  converte_palavra(deposito, "%f", &a);
  leia(deposito, tamanho_do(deposito), entrada_pdr);
  converte_palavra(deposito, "%f", &b);
  leia(deposito, tamanho_do(deposito), entrada_pdr);
  converte_palavra(deposito, "%f", &c);
  ```

  A função

  ```
  leia(deposito, tamanho_do(deposito), entrada_pdr);
  ```

  aguarda que algum dado seja fornecido pelo teclado. O usuário pode responder com qualquer coisa, mesmo absurda.

  A função seguinte

  ```
  converte\_palavra(deposito, "%f", &a);
  ```

  toma o conteúdo de `deposito` e vai lançá-lo na variável `a`. Eu já tenho gravado em disco um pequeno arquivo chamado "`entra_dados`" em que estas duas linhas se encontram gravadas e simplesmente eu as copio para o programa que estiver fazendo quando quiser fazer uma entrada de dados. Sempre uso a mesma variável `deposito` com esta finalidade. Se eu me esquecer de fazer a declaração `palavra deposito[80]` o compilador vai me anunciar este erro dizendo-me que "a função `deposito` está sendo definida implicitamente" que significa que ele a desconhece...

- algoritmo Calcular o "delta" para decidir se a equação tem soluções reais e quantas são; `delta = b*b - 4*a*c;`

- algoritmo Separar em tres opções os cálculos, dependendo do valor do "delta" com uma mensagem adequada para cada caso. Veja na figura (fig. 8.1) página 180,

```
se (delta >= 0)
inicio
 delta = pow(delta,2);
 if (delta == 0)
 inicio
   raiz1 = –b/2*a;
   imprima("Ha uma unica raiz: %f \n", raiz1);
   imprima("\n ");
 fim
 ou\_entao
 inicio
   raiz1 = (–b – delta)/2*a;
   raiz2 = (–b + delta)/2*a;
   imprima("As raizes sao \%f e \%f$\backslash$n ", raiz1, raiz2);
 fim
fim
ou\_entao
inicio
 imprima("A equacao eh impossivel no campo real $\backslash$n ");
fim
```

Figura 8.1: Equação do segundo grau

O resultado desta análise é o programa `prog6101.c`. Rode primeiro o programa para ver seu funcionamento, depois leia o programa e verifique que as etapas acima descritas se encontram executadas.

Exercícios: 48 *Equação do segundo grau*

1. *Rode o programa* `prog6101` *e verifique que o código tem um defeito grave, o usuário fica sem saber o que deve fazer, porque o programa não o informa de que se encontra à espera de uma entrada de dados. Corrija isto.*

   Solução `prog6102`

2. *Rode programa* `prog6101` *com uma equação simples (que você conheça as raizes), verifique que o programa calcula errado. Corrija o programa.*

3. *Rode o programa* `prog6102` *em que o defeito de comunicação com o usuário foi corrigido. Mas verifique que o programa continua calculando errado as raizes. Corrija isto.*

   Solução: prog6103.c

   A solução destes exercícios se encontram nos programas

   `prog6102.c prog6103.c prog6104.c`

4. *Corrija o "diálogo" na entrada de dados fazendo aparecer uma mensagem indicando qual o coeficiente que deve ser fornecido. Solução* `prog6102.c`

5. *Estude o programa* `prog6103.c`. *Observe que nele foram incluidas as funções*

   - `mascara()` *logo no inínicio.*
   - `obrigado(), apetecof(), copyleft()` *ao final.*

Rode o programa e veja o efeito destas funções. Elas estão definidas no arquivo `ambiente.h`*. Altere estas funções (em* `ambiente`*) e roda o programa para analisar os efeitos.*

6. *Altere as funções* `obrigado(), apetecof(), copyleft()` *para atender os seus interesses.*

7. *Estude o programa* `prog6104.c`*. Ele traz a seguinte inovação sobre* `prog6103.c`

   modulariza as operações. O "lay-out" das comunicações com o usuário está também melhorado.

8. *Complete o programa* `prog6104.c` *para que ele resolva equações com soluções complexas (não) reais. Acrescente um módulo para fazer isto, mas não se desconcerte se funcionar mal...*

   Solução `prog6105.c`

9. *Modularize mais o programa* `prog6105.c` *separando as rotinas para calcular raizes complexas reais das raizes complexas não reais.*

   Solução `prog6106.c`

Observação: 37 *Modularização dos programas*

*Um melhor título para este texto seria "*nível de abstração nos programas.*"*

Na suite de programas `prog6101.c, ..., prog6106.c` *não há nenhuma inovação essencial fora o caso do programa* `prog6103.c` *em que todos os defeitos principais de* `prog6101.c` *foram sanados.*

Os outros programas, principalmente `prog6105.c, prog6106.c` *são refinamentos dos anteriores.*

Uma pergunta se impõe: trabalho necessário ou inútil.

Aqui vamos retomar, brevemente, uma observação feita na introdução destes livro: "como aprender a programar bem?" Mesmo aqui, perto do fim do livro, é difícil de responder a esta pergunta...

Uma tentativa de resposta viria dentro de uma longa história da computação que obviamente não faremos aqui, até mesmo porque não somos historiadores. Mas vamos deixar uma breve resenha que não será muito clara por si própria, entretanto, de tanto repassar os olhos sobre ela você vai terminar compreendendo qual o seu sentido.

- *Primeiro se "programou" o ENIAC para resolver cada problema individualmente.*

- *Depois as linguas de programação foram criadas. Com uma linguagem de programação resolvemos grupos de problemas, sem mexer no computador. Observe que no item anterior se quer dizer que se alterava a configuração da máquina, literalmente.*

- *Depois as linguagens de programação evoluiram passando a usar bibliotecas em que várias operações comuns a todos os problemas passaram a ficar disponíveis para qualquer programa. Veja o que acontece com o programa* `prog6103.c`*, nele fazemos uso das funções contidas na biblioteca* `ambiente.h`

  - `mask()`*,, que identifica todos os nossos programas. Criamos a nossa marca, a simplesmente a repetimos em todos os programas.*
  - `copyleft()`*, estabelecemos uma vez por todas a questão da titularidade dos programas.*
  - `obrigado()`*, pensamos uma vez em como fazer uma despedida, depois simplesmente a repetimos.*
  - `apeteco()`*, esta é uma função importantíssima, pese sua grande simplicidade. Seu nome significa "APerte uma TEcla para COntinuar" e de uma vez por todas resolvemos este detalhe em qualquer programa.*

E a história ainda continua, nas linguagens de programação montadas durante os anos 90, se aprofundou a capacidade de "abstração" permitindo programas com muito mais alcance. Surgiram as linguagens de programação a objeto *que são módulos mais evoluidos ainda. É o caso de* $C++$, `Python`, `java`*, entre outras menos recentes.*

A "modularização" é apenas um pequeno detalhe, um passo inicial, mas é necessário dominar desde logo este passo.

E repetindo, os módulos de um programa não devem passar do tamanho da tela porque os insetos adoram se esconder nas laterais...

8.3 Somas e integrais em C

Se você não sabe o que é integral, ou se você tem medo de integrais, melhor pular esta seção. Se você quiser aprender um pouco sobre integral, acompanhe o processo.
Vamos estudar uma sucessão de variantes do programa `integral.c`. Começaremos com uma versão simples até alcançarmos uma versão mais "sofistificada". A suite de programas que estudaremos é:

`integral01.c integral02.c integral03.c`

8.3.1 Integral de funções univariadas

Sobre *integral* leia por exemplo [3] onde o assunto está voltado para o *cálculo aproximado de integrais* como é o espirito aqui. Para uma visão teórica mais detalhada veja [4].

A integral de f, simbólicamente:

$$\int_a^b f$$

que se lê

integral de f de a até b,

é área da região limitada pelo gráfico de f, pelo eixo OX entre os pontos a, b.

Esta é uma definição elementar que serve a muitos propósitos. Calcular

$$\int_a^b f$$

aproximadamente pode ser feito de várias maneiras, por exemplo, calculando as áreas de retângulos contidos na região acima descrita. Veja gráficos em [4], ou em qualquer livro de Cálculo.

A idéia do algoritmo está na figura (fig. 8.2) página 183,

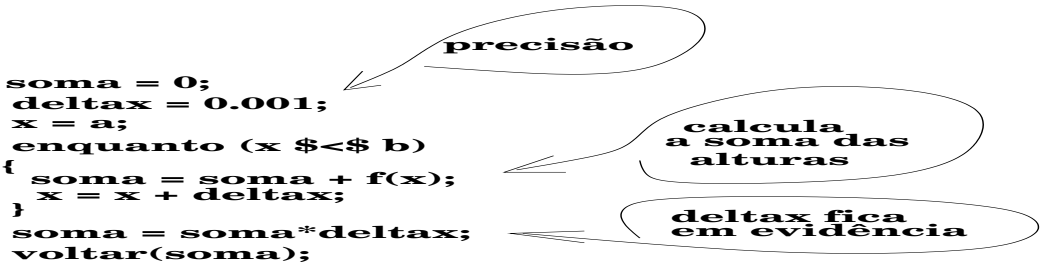

Figura 8.2: Cáculo da integral, aproximadamente.

em que fazemos a variável x "varrer" o intervalo $[a, b]$ com passo $deltax$. Quando x for igual a b, ou ultrapassar este valor, então o programa para devolvendo o valor $soma * deltax$

Dentro do *loop*, os valores de f calculados sobre a malha determinada em $[a, b]$ pelo passo $deltax$, vão sendo acumulados na variável $soma$ que foi inicializada com zero, $soma = 0$, fora do *loop*.

Isto equivale a dizer que somamos todos os valores de f calculados sobre esta malha. A expressão matemática do loop é:

$$\sum_{k=0}^{k=n-1} f(x_k).$$

e, naturalmente, é melhor que você converse com um matemático (ou me envie um e-mail...) se tiver alguma dúvida sobre o que está dito aqui, do contrário este livro se tornaria um livro de Matemática.

Quer dizer que "somas" ou "varreduras" se fazem com *loops*.

Analise o programa `integral01.c`, depois o rode para ver o resultado e volte a lê-lo para compreender melhor como ele funciona.

O algoritmo que apresentamos acima, é o *loop* da função

`Riemann()`

que calcula Somas de Riemann. Ela está etiquetada com o número (4) no programa. Veja o comentário " o algoritmo para calculo da integral".

A função "principal" se ocupa de todos os detalhes.

- Inicialmente "conversa" com o usuário solicitando o o intervalo de integração:

```
imprima("inicio do intervalo [a,b] de integracao");
```

- Se seguem depois tres entradas de dados para receber os valores de

$$a, b, deltax.$$

que se encontram "numeradas" com (3), (3.1).

- Finalmente uma única linha de comando contém a mensagem final e chama `Riemann()`

```
imprima(" de  %f \n",Riemann(_inicio,_fim,deltax)); // (4)
```

A etiqueta (4) marca os pontos em a função `Riemann()` está definida ou é chamada.

Defeitos do programa integral01.c

O diálogo inicial com o usuário é pobre. Deveria ser mais longo dizendo qual o objetivo do programa. As tres entradadas de dados dizem pouco sobre o que pedem.

Um programa precisa ser mais elegante e conversar melhor com o usuário. Claro, numa versão posterior deste programa (noutro livro...), podemos incluir um ambiente gráfico deixando o programa trabalhar no porão e apresentar em janelinhas as mensagens instruindo o usuário sobre as entradas de dados. Mas agora já é possível algo melhor do isto, veja os exercícios.

A saída de dados final está muito lacônica. Ela poderia até mesmo explicar como o programa foi feito. Quando soubermos fazer gráficos, (próxima secção), seria o caso do programa ir mostrando o gráfico do que está fazendo.

Exercícios: 49 *Melhorando integral01.c*

1. *Melhore a entrada de dados de* `integral01.c` *explicando melhor o que o programa se propõe a fazer com cada um dos números solicitados.*

2. *Veja em* `integral02.c` *que espaçamento colocado entre entre as seções do programa não interferem com a lógica e podem ser usados para facilitar sua interpretação. Faça alterações semelhantes para tornar* `integral01.c` *mais legível e mais comunicativo.*

3. *Construa uma função para fazer a entrada de dados. Solução* `integral03.c`

4. *Subdivida o programa em pequenos módulos tornando a função principal apenas uma listagem de tarefas. Solução* `integral03.c`

5. *Melhore a saída de dados tornando-a mais verbosa.*

8.4 Gráficos de funções usando $\mathcal{C}$

Você vai ver aqui como poderia fazer uma pequeno tutorial para ensinar gráficos de funções aos seus alunos ou para você mesmo, para seu aprendizado pessoal.
Nesta seção faremos duas coisas:

- Chamaremos um programa externo
  Dentro do espirito do Unix, e LinuX é Unix: *para que fazer outro programa se já existe um que funciona bem?* e é o caso com gráficos. Existe um programa, de domínio público, chamado GNUPLOT, com versões para DOS e WINDOWS, inteiramente grátis, que trabalha muito bem com gráficos.
  Vamos aqui aprender a chamar o GNUPLOT de dentro de um programa feito em $\mathcal{C}$ para que GNUPLOT faça o gráfico que precisamos.
- Vamos reutilizar um programa
  Para fazer o gráfico de uma função f precisamos "varrer" o domínio de f para construir a matriz dos pontos $(a_i, f(a_i))$ em que $i \in \{1, \ldots n\}$ sendo n a quantidade de pontos que vamos representar, ou a precisão com que faremos o grá fico.
  Esta idéia é a mesma para calcular integrais usando Somas de Riemann, logo vamos reutilizar os programas `integral01.c ...` que fizemos anteriormente.

Este parágrafo está baseado no programa `grafun.c` e o nosso objetivo é conduzí-lo a entender este programa.

Leia rapidamente `grafun.c`, inclusive a avaliação do mesmo que pode ser encontrada no começo: programa primário... há uma lista de coisas que devem ser feitas para que ele se torne um programa respeitável.

Observe que `grafun.c` se encontra no "estágio final" onde a sucessão de programas

`grafun01.c grafun02.c grafun03.c`

o vai levar, logo ele não deve ser fácil de ser entendido. Não se desespere, entretanto... Vamos começar com `grafun01.c`, que se encontra no "estágio inicial".

É sempre assim, não tenha dúvida, que começa a vida de um programa. Primeiro construimos um que funcione, depois passamos a melhorá-lo. Melhor

uma galinha[2] na panela do que duas no terreiro.

Rode o programa `grafun01.c` e depois o leia.

gcc -Wall -lm grafun01.c -o grafun

e rode o executável que foi criado:

`grafun`

ou possivelmente, dependendo da organização de sua máquina, (se a tiver recebido a mensagem de que o programa não existe)

`./grafun.`

8.4.1 Comentando o programa `grafun01.c`

Vamos iniciar o comentário pelo centro (a parte mais importante) do programa. O resto é periférico... [3]

O centro do programa é:

```
dados = fopen("dados","w");  // (5.1)
x = _inicio;
enquanto (x <= _fim)
 inicio
     fprintf(dados, "%f %c %f\n",x,' ',funcao(x));
      fprintf(dados, "%f %c %f\n",x,' ',0.);
      x = x + delta;
   fim
fclose(dados); // (5.2)
```

Isto porque estamos usando um método típico do Unix. Se tem um programa que faz gráficos, eu não vou perder tempo inventando um meio de fazer gráficos, eu vou usar este programa: `Gnuplot`.

`Gnuplot` faz gráficos de distintas maneiras, numa delas ele precisa ler um arquivo contendo os pares de pontos $(x, f(x))$ para fazer o gráfico da função f.

Rode `grafun01.c`, depois edite o arquivo "dados" para ver o que foi escrito alí.

`grafun01.c` abre o arquivo "dados" e nele escreve os pares de pontos

$$(x, f(x)), (x, 0).$$

Depois o programa `Gnuplot` vai ler este arquivo e criar o gráfico.

Mas para fazer isto é preciso definir as variáveis:

- x;
- _inicio;
- _fim;

[2] pensamento imediatista...as galinhas no terreiro põem ovos!

[3] você não deve levar a sério os comentários do autor...

- dados;
- funcao();

Vamos ao começo do programa. Depois dos comentários iniciais, o programa começa chamando as bibliotecas necessárias onde estão definidas as funções que o programa precisa e inclusive nossas traduções.

Depois definimos duas funções auxiliares:

```
inteira  rotulo();
real     funcao(real x); // (3)  equacao cuja grafico se deseja
```

Em rotulo() uma marca simples para o programa dizendo o que ele faz. Em funcao() definimos a equação da função cujo gráfico desejamos.

Definimos as variaveis necessárias ao programa:

- palavra deposito[60], titulo[80];
- real _inicio, _fim, x, delta;
- ARQUIVO *dados; // (5.1) dados para o `Gnuplot` com (x,funcao(x))
- ARQUIVO *transfere; // (5.3) linhas de comando do `Gnuplot`
- palavra programa[18]="gnuplot transfere"; // (6) programa externo

Leia `grafun01.c` vamos apenas observar as linhas:

```
deposito[strlen(deposito)-1]='\0';                    // (7.3)
copia_de_palavra(titulo,"'");                         // (7.4)
concatena_palavras(titulo,deposito);                  // (7.5)
concatena_palavras(titulo,"'");
```

- Em `(7.3)` estamos cortando a variável depósito para ficar com o tamanho exato da expressão do título que o usuário tiver escolhido. Colocando o NULL na última posição vaga.
- Em `(7.4)` estamos colocando aspas simples, necessário para `Gnuplot` que exige que o texto que ele receba fique entre aspas simples ou duplas.
- Em `(7.5)` estamos fechando as aspas.

finalmente chegou o momento de lançar os dados em "dados" como dissemos acima. Leia o arquivo "`dados`". Depois abrimos o arquivo "`transfere`"

```
// **** redireciona output para  'transfere'
transfere = fopen("transfere","w");       // (5.3)
fprintf(transfere, "set title %s \n",titulo);
fprintf(transfere, "set pointsize %f \n", 0.1);
```

```
fprintf(transfere, "set size %f ,%f \n", 1. , 0.7);
fprintf(transfere, "%s %s %s \n","plot","'dados'"," with points");
fprintf(transfere, "%s \n", "pause -2    ");
fclose(transfere);   // fechando o arquivo transfere
```

escrevendo nele todos os comandos que `Gnuplot` precisa para fazer o gráfico. A última função fecha "`transfere`": `fclose(transfere)`. Leia o arquivo depois de rodar o programa para ver o que foi nele escrito, e volte a ler `grafun01.c`.

Não sei se você vai acreditar, mas eu penei um bocado para aprender a escrever corretamente os dados em disco. Não espere entender isto de um tapa... Rode `grafun01.c`, leia os arquivos `dados` e `transfere`, volte a ler e rodar `grafun01.c`, e assim você irá entendendo os comandos.

O programa `grafun01.c` termina chamando `Gnuplot` através da variável "programa":

```
imprima("*** chama um processo externo, Gnuplot *** ");
system(programa);   // (6.1) (6.2) chama programa {\tt Gnuplot}
```

Faça a seguinte experiência que o vai ajudar a entender todo o teatro que acabamos de montar. Execute numa shell do `Linux`, dentro do diretório onde você está rodando os programas o comando:

`gnuplot transfere`

para que você volte a ver o gráfico.

Exercícios: 50 *Entendendo e modificando* `grafun01.c`

1. *Compile* `grafun01.c`*! tem tanto erro que é melhor passar para* `grafun02.c`*. Fica como desafio, para quando você adquirir mais experiência, corrigir este programa. É um bom exercício.*

2. *Melhor talvez abandonar* `grafun01.c`*, tem muito erro. Leia e rode* `grafun02.c`*. O erro está indicado nas primeiras linhas.*

3. *Apague*[4] *as bibliotecas definidas no início do programa*

   ```
   # include <stdio.h>   // (1) **** leitura da biblioteca stdio.h
   # include <stdlib.h>
   # include <string.h>
   # include "traducao.h"  // (2)  **** leitura da biblioteca traducao.h
   ```

   compile o programa e analise os erros (ufa... que quantidade enorme de erros) apague as bibliotecas uma por uma...

4. *Como você já deve ter rodado* `grafun02.c`*, vamos entender como funciona* `Gnuplot`*. Rode:*

   gnuplot transfere

   e veja que o resultado é o mesmo que rodar `grafun02.c`*.*

[4] não obedeça! em vez de apagar, comente...

5. *Edite os arquivos "dados", "transfere". Eles foram criados por* `grafun02.c`. *Também* `grafun02.c` *chamou* `Gnuplot` *e lhe passou os comandos contidos em "transfere"*

```
system(programa);   // (6.1) (6.2) chama programa {\tt Gnuplot}
```

Altere esta linha em `grafun01.c` *para*

```
system("gnuplot transfere");
```

e volte a compilar. Haverá um erro, porque a variável "programa" deixou de ser usada. Corrija este erro apagando a definição desta variável.

6. *Defina mais duas outras funções e faça* `Gnuplot` *colocar os seus gráficos na tela.*

   Solução `grafun02.c`

7. *Defina uma "biblioteca" de funções, veja* `menu_funcao.h` *para que o programa nela vá buscar a equação que lhe interessa.*

   Solução estude `grafun03.c`

8. *Altere* `grafun03.c` *para que lhe seja permitido, primeiro, ver o arquivo* `menu_funcao.h` *de modo que, quando você digitar o título possa escrever a equação da função. Observe que em LinuX você pode facilmente memorizar a equação da função com o rato e colar na tela de execução do programa.*

   Solução: `grafun04.c`

9. *Modularize o programa* `grafun04.c` *de formas que a função principal() seja apenas uma listagem de tarefas definidas pelos módulos (funções)*

   Solução: `grafun05.c`

O programa `grafun.c` é a versão final desta série de estudos.