O Guia do Tio Jedi para a Programação em C!

- Alpha Edition

Bruno Figueira “Jedi” Lourenço

Sumário

Como ler esse livro

O livro foi dividido em três partes: Básico, Intermediário, Avançado. (nota: eu ainda não tenho bem definido o conteúdo de cada parte, portanto eu deixarei para terminar essa seção após ter terminado parte do texto).

Note que a premissa básica é que que o leitor já teve algum contato com *alguma* linguagem de programação, já sujou as mãos com estruturas de dados como pilhas, filas, listas e árvores e possui entendimento dos conceitos básicos ligados ao um computador digital.

Esses são pré-requisitos importantes, pois nos permite concentrar no que é realmente importante e nos pontos que geralmente causam problemas ao programar em C.

Note também, que não possuo nenhum compromisso em manter um tom formal. De nada adianta um texto correto mas que funciona como um sonífero para o leitor, por isso o autor teve um cuidado especial em fazer com que os exemplos sejam claros e que o texto seja leve, interessante e por vezes, irreverente. Naturalmente, nem sempre o que se *deseja* é de fato alcançado, sendo assim, sugestões e críticas serão perpetuamente bem-vindas.

Cada capítulo a partir do terceiro, possui uma série de exemplos, que o autor recomanda que sejam compilados e testados. Além disso, ao final desses capítulos há alguns exercícios. Note que qualquer leitor que queira mandar um exercício para ser incluído em edições futuros é bem-vindo.

Além disso, os exercícios possuem estão marcados com um grau de dificuldade variando de 1 a 10. O leitor, ao final de cada capítulo, deve estar apto fazer todos os exercícios que tenham pelo menos dificuldade 5. Note que isso não significa que *todos* os exercícios devem ser feitos, mas apenas que o leitor deveria fazer ao menos alguns que possuem dificuldade 5 ou mais.

Parte I

Básico

Capítulo 1

Introdução

Naturalmente, você como leitor, deve estar esperando que eu faça uma introdução repleta de elogios ao C e que justifiquem o tempo gasto com a leitura desse texto. Bom, infelizmente, isso não vai acontecer. O objetivo desse texto é mostrar um panorama honesto da linguagem, suas construções mais comuns e problemas relacionados à essas construções.

O C é uma linguagem de uso geral, mas definitivamente não será a solução de todos os seus problemas, porém *em boa parte* dos casos será uma solução muito boa. Se você não possui idéia da linguagem a ser usada em um projeto, há grandes chances de C ser uma boa escolha:

- É uma linguagem portável.(bom... desde de que siga algum padrão: ISO, ANSI)
- Possui aspectos de linguagens de alto e de baixo nível(manipulação de ponteiros e endereços de memória).
- Existem excelentes bibliotecas gratuitas para diversos campos de aplicação.

Os três pontos acima são razoavelmente questionáveis, já que existem outras linguagens que fornecem essas mesmas vantagens ou parte delas. (Java, FORTRAN, Pascal e muitas e muitas outras) Mas, então, por quê, *de verdade*, programar em C?

A resposta não é simples e espero que ao término desse texto, o leitor possua a sua própria resposta. O que eu posso oferecer agora, é a minha versão dos fatos e os *meus* motivos para querer programar e dominar bem o C.

Antes, porém, um pouco de história.

1.1 As origens

Ao contrário das linguages típicas dos anos 70 e 60, o nome "C" não é um acrônimo como no caso das seguintes linguagens:

- COBOL = ***C***ommon ***B***usiness ***O***riented ***L***anguage
- FORTRAN = ***For***mula ***Tran***slating System

- LISP = ***Lis***t ***P***rocessing
- PL/1 = ***P***rogramming ***L***anguage 1.

O leitor pode se perguntar então: "Existiu uma linguagem chamada 'A' e uma linguagem chamada 'B'?". O autor desconhece se existiu (ou existe) uma linguagem chamada "A", mas de fato existiu uma linguagem chamada "B".

Na verdade, o B era uma simplificação de uma linguagem chamada BCPL (***B***asic ***C***ombined ***P***rogramming ***L***anguage) desenvolvida por Martin Richards no longínquo ano de 1966 na Universidade de Cambridge.

O BCPL era uma linguagem compacta, sem muitas restrições de sintaxe. O objetivo era criar uma linguagem que pudesse ser utilizada para a construção de compiladores e de sistemas e que fornecesse um resultado melhor que o FORTRAN sem precisar de recorrer ao assembly.[1]

Quando, Ken Thompson precisou de uma linguagem para programar no novo computador PDP-7 nos laboratórios da Bell, ele não pôde utilizar o BCPL, pois era demasiadamente pesado e PDP-7 era uma máquina muito limitada.

Por isso, ele criou um *subset* do BCPL chamado B.[2] O B retirou tudo que Thompson acreditava que era supérfluo no BCPL.

O B era uma linguagem interessante: conscisa, simples e leve. Porém, ela era demasiadamente de baixo nível. A manipulação de strings era trabalhosa e havia apenas um único tipo de dado: o "computer word"(uma "palavra de computador"). Sendo assim, o mesmo tipo de dado poderia ser uma inteiro, uma string e o programador poderia acidentalmente elevar uma string ao quadrado. O que certamente não era o que ele pretendia. Além disso, não havia operações em ponto-flutuante(embora isso seja mais um problema das limitações do hardware do que da linguagem propriamente).

E que sistema operacional a equipe de Ken Thompson estava desenvolvendo nos laboratórios da Bell? Nada menos do que o **Unix**.[3] Nesse ponto, o leitor deve notar que o desenvolvimento do B e do C estava intimamente ligado com a programação de sistemas operacionais, mais especificamente, o **Unix**.

Quando se programa sistemas operacionais, há um equilíbrio delicado entre *facilidade* e *desempenho*. A linguagem utilizada deve ser "fácil". Note que "fácil" quer dizer "mais fácil do que assembly". Isto é, programas feitos na linguagem devem ser necessariamente menos trabalhosos do que os mesmos programas escritos em assembly. "Facilidade" geralmente implica em "abstração". Isto é, a linguagem "esconde" determinados problemas que não são do interesse do programador para aquela tarefas específicas ao qual a linguagem é destinada.

Se a linguagem oferece um nível muito alto de abstração, o desempenho provavelmente sofrerá, já que o programador perde um pouco o controle sobre alguns aspectos do programa, impedindo que ele faça um ajuste mais "fino".

É visando atingir esse equilíbrio delicado que Dennis Ritchie projeta o C, como um sucessor do B. O objetivo era criar uma linguagem que fornecesse so-

[1] `http://www.lysator.liu.se/c/clive-on-bcpl.html`

[2] Há controvérsias sobre o *porquê* da linguagem se chamar B. Dizem que como o B era uma versão "menor" do BCPL, Ken Thompson apenas tirou CPL e portanto B não significaria nada de especial. Outros dizem que o B seria uma homenagem à sua esposa.

[3] A origem do nome Unix é curiosa. De início, nos laboratórios da Bell, houveram tentativas de criar uma sistema operacional chamado **Multics**. Porém, o projeto não foi para frente e com isso algumas das pessoas que estavam no projeto do **Multics** iniciaram o desenvolvimento do **Unix**, que teoricamente seria uma versão mais simples do sistema anterior. Com isso, eles abondonaram o prefixo *Multi* e substituíram por *Uni*.

luções para algumas características do B que eram indesejáveis como a ausência de um sistema de tipagem.(veja a seção "Leitura Adicional")

Naturalmente, o C criado naquela época (começo dos anos 70) era bastante diferente do que é hoje. Nos anos 80 a linguagem começou a tomar a forma como é conhecida hoje. E o resto, como dizem, é história.

1.2 Os padrões

As linguagem de programação, de modo geral, precisam de um padrão se elas querem ser portáveis. Isto é, precisam de um documento que defina as funcionalidades que uma implementação da linguagem **X** precisa ter para que seja de fato uma implementação padronizada.

Mas porque isso torna a linguagem portável? Na verdade, apenas o padrão não torna a linguagem portável. O que torna a linguagem portável é a *existência* de implementações que sigam o padrão.

E o que isso significa do ponto de vista do programador da linguagem **X**? Se o programador sabe que para diferentes plataformas existem implementações que seguem os padrões, teoricamente bastaria ele programar utilizando *apenas* o que está especificado nesses padrões e com isso obteria-se um programa portável.

Quem são as organizações que criam os padrões? Bom ... Dependendo da linguagem existem várias. O Java por exemplo é "padronizado" pela *Sun Microsystems*, eles fornecem a especificação da *Java Virtual Machine* e se alguém quiser chamar a sua implementação de "Java" é melhor seguir a espeficação senão a *Sun* processa! Foi assim que a Microsoft perdeu $20 milhões de dólares para a Sun em 1997. A Microsoft criou uma implementação do Java que não era 100% compatível e como a licença da Sun prevê que as implementações do Java devem ser 100% compatíveis com as espeficações, então a Sun resolveu processá-la.

Bom, o Java é um caso à parte. Dificilmente uma linguagem possui um padrão que é definido por apenas um empresa. Geralmente, grupos de empresas e de pessoas interessadas formam comitês para elaborar os padrões. Como exemplo, podemos citar a **ISO**(*International Organization for Standardization*) e a **ANSI**(*American National Standards Institute*).

No caso do C, durante muitos anos o "padrão" foi um livro chamado "K&R" escrito em 1978 por Brian Kernigham e Dennis Ritchie. Também chamado de "o livro branco", a primeira edição desse livro descrevia a linguagem C e como utilizá-la para programar no Unix.

Nessa época, se alguém quisesse escrever um compilador para C, com certeza teria ler o "K&R". Naturalmente, uma boa parte das implementações adicionavam à linguagem algumas novidades. Com isso, aos poucos a primeira edição do "K&R" foi deixando de refletir realmente a linguagem C que estava em uso.

O uso da linguagem, que aumentava cada vez mais, trouxe a necessidade de um padrão que viesse corrigir a salada de implementações diferentes. Em 1983, a **ANSI** formou um comitê para estabelecer a especificação padrão do C. Em 1989, esse padrão ficou pronto e foi ratificado como **ANSI** X3.159-1989 "Programming Language C.".

O padrão **ANSI** define o comportamento do pré-processador, o conjunto de funções que constituem a biblioteca padrão, como devem funcionar as expressões

aritméticas, atribuições, *casts*, regras de precendência e todo tipo de detalhe que é de se esperar de um padrão decente.

Hoje(2008), a maior parte das implementações de um compilador de C, segue esse padrão. Naturalmente, a linguagem não fica parada no tempo e em 1999 um comitê da ISO publicou um padrão que tinha o propósito de extender algumas funcionalidades do C: adicionava suportes a números complexos, um tipo booleano explícito, o tipo `long long int` e muitas coisas mais.

Infelizmente, ainda são poucas as implementações que seguem esse novo padrão, também chamado de C99. Então, por enquanto, ainda é mais seguro programar no que é chamado de "**ANSI** C".

1.3 E então? A que conclusão chegamos?

O C foi criado para tornar a programação mais fácil. Mas, "ser fácil" implica que alguma coisa é mais difícil, correto? Bom, o C certamente é mais fácil do que Assembly e é possível obter o mesmo desempenho. Alguns dizem que C *não* é mais fácil do que algumas linguagens que surgiram recentemente como Java, C# e outras. É claro que essas linguagens são orientadas a objetos, mas a comparação se mantém.

Como foi possível observer, o C é uma linguagem que foi desenvolvida por especialistas. Isso não significa que pessoas que não sejam especialistas não possuam condições de aprender a linguagem. Mas, com certeza, elas esbarrarão em alguns cantos do C que são mais complexos como manipulação de ponteiros.

Aprender C, de modo geral, é uma experiência bastante recompensadora. Apesar de ser "difícil" inicialmente, a linguagem possui um combinação única de ferramentas de alto e baixo nível; proporcionando a longo prazo um conhecimento maior do hardware e do sistema operacional.

Para quem é da área de Computação ou Engenharia, cedo ou tarde precisará escrever um *device driver* ou fazer algum outro tipo de codificação em baixo nível. O C, nesse caso, sempre será uma boa opção.

O ponto importante é que o C "não esconde" nada do programador. Embora para o iniciante isso possa ser ruim, para o programador com mais experiência isso significa maior controle e pode ser altamente desejável.

Então... O programador iniciante está em apuros? Não exatamente. Ninguém nasce sabendo C. O único jeito de se aprender C é *programando em C*. [4] Por isso, esse livro contém alguns exercícios ao final de cada capítulo a partir do terceiro para que o leitor possa exercitar os conceitos aprendidos.

1.4 O C resolverá todos os meus problemas?

Não! O programador deve ter capacidade de avaliar qual é a melhor linguagem a ser utilizada para efetuar de forma correta *e* eficiente determinada tarefa. Outros fatores também devem ser levados em consideração como: tempo disponível para o projeto, plataforma-alvo, finalidade da aplicação e outros.

Essa avaliação desse conjunto de fatores é o que se chama popularmente de "bom-senso" e foge ao escopo desse livro.

[4] Pesquisas mostram que aprender por osmose ainda não é uma alternativa viável, apesar de alunos da UnB contestarem essa pesquisa.

Apesar de não aparecer, o autor também programa em outras linguagens que não seja C e acha Python a beleza em forma de linguagem de programação.

O leitor deve ter em mente que é sempre bom aprender novas linguagens, ainda que ele nunca tenha oportunidade de fazer um uso sério delas. Novas linguagens significam novas técnicas que por sua vez significam um arsenal maior de ferramentas a dispor do programador.

1.5 O que o futuro reserva?

De tempos o comitê da ISO se reúne para fazer algumas modificações na linguagem. O C também possui uma descendência própria.

O C++, desenvolvido por Bjarne Stroustrup que na fase de desenvolvimento era chamado de "C com classes", talvez seja o "filho" mais notável do C. A tal ponto, que a maioria dos compiladores de C também compilam para C++, sendo chamados então de compiladores de C/C++.

O "++"no C é o operador de incremento, então C++ significaria algo como "C + 1", ou seja, a linguagem C mais *alguma coisa*, que no caso esse alguma coisa seria a orientação a objetos. Teoricamente o C++ era para ser compatível com o C e vice-versa. Mas, na prática há algumas pequenas diferenças. O C++ é mais fortemente tipado do que o C. De modo geral, os dois são compatíveis.

Mas não é só de C++ que vive a descendência do C. Existe também o D! Essa linguagem não possui o compromisso de ser compatível com C e também acrescenta orientação a objetos. Mais informações podem ser encontradas em `http://www.digitalmars.com/d/` .

Claro que existem vários outros descendentes do C. A própria influência do C se mostra através da adoção do seu estilo de sintaxe nas mais diversas linguagens como Java, C# e até na linguagem para web PHP.

> *"Great, just what I need.. another D in programming."* – Segfault[5]

1.6 Leitura Adicional

A história do C é bastante interessante e felizmente, na Internet há uma série de links com informações adicionais. Esses links foram utilizados pelo autor para escrever as seções anteriores.

- `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html` Artigo escrito por Dennis Ritchie que conta a história do desenvolvimento do C e coloca-o no contexto do B e do BCPL. Além disso, mostra uma panorama geral do Unix que estava sendo desenvolvido na época.

- `http://cm.bell-labs.com/cm/cs/who/dmr/bcpl.html` Para aqueles que ficaram curiosos e querem mais informações sobre o BCPL, essa página contém o manual de referências do BCPL de 1967 para download. Para os corajosos, uma versão "moderna" do compilador de BCPL pode ser encontrada em `http://www.cl.cam.ac.uk/~mr10/BCPL.html`

[5]Retirado de http://www.digitalmars.com/d/index.html

- `http://cm.bell-labs.com/cm/cs/who/dmr/kbman.html` Contém uma espécie de manual do usuário para o B escrito por Ken Thompson, bastante interessante. As semelhanças com o C são facilmente percebidas.

- `http://en.wikipedia.org/wiki/C_%28programming_language%29` A wikipedia possui muitas informações interessantes sobre o C. Na parte "External Links" há links para tutoriais e sites sobre programação em C.

Capítulo 2

Primeiros Passos

Nesse capítulo será mostrado como instalar os softwares necessários para dar início à criação de programas em C. Lembrando que o compilador utilizado será o GCC, que está disponível "de grátis".[1]

Note que o leitor provavelmente precisará de um depurador, portanto o autor sugere a utilização do GDB e se o leitor achar necessário, algum *front-end* gráfico para ele como o DDD.

2.1 Instalando os programas necessários

O GCC é uma coleção de compiladores (*GNU Compiler Collection*), mas também designa um compilador presente nessa coleção(*GNU C Compiler*). Esse compilador já existe há um bom tempo e é um dos mais utilizados atualmente, além de estar presente em várias plataformas.

Se o leitor possui Linux instalado em sua máquina, provavelmente possuirá o GCC instalado também. Caso possua Windows, a história é um pouco diferente. As seções seguintes explicam como ter um ambiente de desenvolvimento funcional para Linux e para Windows.

2.2 Linux

Para verificar se o GCC já está instalado, abra uma janela do terminal (gnome-terminal, kterminal, urxvt ou o que for o emulador de terminal no seu sistema) e digite `gcc`. Se o gcc estiver instalado, deve aparecer a seguinte mensagem: `gcc: no input files`. Caso contrário, deve aparecer uma mensagem parecida com essa: `bash: gcc: comando não encontrado`

Bom... se o GCC não estiver instalado, aí vai depender da distribuição. Para as distribuições baseadas no Debian (como Ubuntu), existe um meta-pacote chamada *build-essential* que instala o GCC. Para instalar esse pacote logue como root e digite `apt-get install build-essential`, caso esteja no Ubuntu, digite `sudo apt-get install build-essential`.

[1] Free as in "free speech", but also free as in "free beer" `http://www.gnu.org/philosophy/free-sw.html`

Para instalar o GDB, apenas digite `apt-get install gdb`. Use `sudo` caso esteja no Ubuntu. Lembrando que as distribuições baseadas no Debian, geralmente vêem com um programa para instalar os pacotes .deb chamado *Synaptic*. A instalação pode ser feita através desse programa também.

O autor não tem experiência em distribuições que não sejam baseadas no Debian, sendo assim se algum leitor quiser mandar instruções para instalação do GCC e do GDB em outras distribuições, elas serão bem-vindas.

2.2.1 Editores de Texto

O leitor provavelmente irá querer digitar o código-fonte em um programa decente. Felizmente, no Linux há excelentes opções:

- gedit Vem com a maior parte das distribuições do Linux. É leve, possui uma arquitetura de plugins e é possível customizá-lo bastante. Combina com quem usa o ambiente GNOME(embora, nada impeça de utilizá-lo no KDE).`http://www.gnome.org/projects/gedit/`
- kate Um editor de texto um pouco mais completo do que o gedit. Possui mais opções de customizações e possui a opções de acoplar um terminal virtual na parte inferior da tela. Combina com quem usa o ambiente KDE.(e de modo análogo, não há nada que impeça de ser utilizado no GNOME) `http://kate-editor.org/`
- geany Esse é o favorito do autor. Embora os desenvolvedores o identifiquem como "Uma IDE rápida e leve", o autor enxerga o geany mais como um editor de texto turbinado. Possui um emulador de terminal, mostra as divisões do programa em funções, além de pequenas facilidades que tornam programar uma tarefa mais divertida. É ver para crer. Altamente recomendado.`http://geany.uvena.de/`
- vi Esse é para o pessoal *l33t*. O autor não possui muita proficiência nesse editor, mas dizem que apesar de difícil de aprender, torna programar muito mais fácil. Existem diversas variantes e boa parte das distribuições do Linux vem com uma versão dele, sendo assim nenhum link em especial será indicado. Para o leitor interessado, uma pesquisa no Google deve resolver os problemas.
- emacs Outro editor clássico presente em boa parte dos sistemas Unix. Ele faz de tudo e para os momentos de desespero funciona até como psicólogo(sem brincadeira!). Existem duas variedades populares: o GNU Emacs, escrito inicialmente por ninguém menos do que Richard Stallman e o XEmacs desenvolvido inicialmente na Lucid Inc.[2]

Existem dezenas(talvez centenas) de outros editores de texto. O leitor deverá utilizar aquele que se sentir mais confortável.

2.3 Windows

Para aquelas pobres almas que utilizam o Windows não há uma saída simples como no caso do Linux. É certo que poderia simplesmente instalar uma IDE

[2]Informação retirada de http://en.wikipedia.org/wiki/Emacs

como o Dev-C++ e pronto, mas o autor tem algumas ponderações a fazer sobre as IDEs e por isso o leitor é aconselhado a ***não*** instalar uma IDE.

Para utilizar o GCC no Windows o autor sugere utilizar o *MinGW* ou o *Cygwin*. Ambos são formas de adicionar funcionalidades que são típicas de Unix ao Ruindow$. Naturalmente, eles possuem suas diferenças, mas para os propósito desse livro, os dois servem bem.

O Cygwin é composto de ferramentas típicas do Unix, o GNU toolchain[3], um conjunto de bibliotecas que implementam uma API que segue o padrão POSIX[4], além de possuir ports de diversos softwares como o X Windows System e o GNOME. Com isso, é possível compilar programas tipicamente feitos para Unix.[5]

Já o MingW possui uma abordagem mais minimalista, possui um foco em performance e portanto, não possui uma API tão extensa quanto o Cygwin. Conforme, dito anteriormente, para os nossos propósitos, os dois são bons o suficiente.

Antes de prosegguir para as próximas seções, que mostram como instalar o Cygwin e o MinGW, gostaria de fazer um apelo emocionado: "LEIAM OS MANUAIS, POR FAVOR!". Esse simples passo evita dores, frustrações e teclados quebrados. Nesse caso em especial, se o leitor não possui familiariedade com Unix, ele deve ler os manuais do cygwin ou do mingw para ter ao menos uma idéia de como se usa o shell.

2.3.1 Cygwin

O primeiro passo é baixar o setup.exe no site `http://www.cygwin.com/`. Esse setup funciona como um gerenciador de pacotes. Com ele é possível baixar e instalar novos programas bem como desinstalar os programas existentes.

- Execute o setup
- Clique em avançar
- Clique em "Install from Internet" e depois clique em avançar
- Escolha o diretório root, se quiser não mexa no padrão(`c:\cygwin`) e clique em avançar
- Clique em avançar novamente
- A menos que o leitor possua alguma configuração de proxy, apenas clique em avançar, de novo
- Nessa etapa, o cygwin está baixando uma lista de mirrors com os pacotes, escolha um dos mirros e clique em avançar. Note que alguns dos mirrors podem estar quebrados. Se isso acontecer apenas volte e escolha outro mirror.

[3] O GNU toolchain é composto do GCC, make, binutils, GDB e do GNU Build System

[4] Por alto, o padrão POSIX define o conjunto de funcionalidades que um sistema deve ter para ser considerado um "Unix"

[5] Lembrando que "Unix" designa, hoje em dia, uma família de sistemas operacionais tais como Linux, FreeBSD e outros

- Quando aparecer a tela de seleção de pacotes vá em "Devel", clique para expandir o menu e procure o pacote gcc, clique onde está escrito "Skip" e deve aparecer o nome da versão a ser instalada. Faça a mesma coisa para o pacote que tem o nome "gdb". Se o leitor quiser instalar mais alguma coisa, basta procurar nas seções que o setup cuida das dependências. Clique em avançar. Nesse ponto, o setup vai baixar os arquivos e o leitor deve ter paciência.

- Quando terminar de baixar e instalar os pacotes aparecerá o opção de criar um ícone na área de trabalho. O autor sugere que o leitor escolha essa opção, pois assim possuirá um acesso rápido ao shell do cygwin.

Seguindo esses passos, o leitor provavelmente terá instalado com sucesso o cygwin. Abrindo o ícone do Cygwin na área de trabalho, um shell se abrirá parecido com o prompt do MS-DOS. Digite `gcc` e confira se aparece a mensagem: `gcc: no input files` que é sinal de que a instalação procedeu de forma correta.

Para ir ao driver C (onde provavelmente estão os seus arquivos) digite: `cd /cygdrive/c/` e pronto, o leitor estará na raiz do driver C. Caso o usuário nunca tenha mexido em uma shell do Unix, segue abaixo alguns comandos simples:

- `ls` Lista os arquivos no diretório atual.

- `cd` Muda de diretório(*Change Dir*). Caso seja digitado `cd ..` muda para o diretório pai("o diretório anterior").

- `pwd` Imprime na tela o nome do diretório atual

- `cat` Geralmente é chamado da seguinte forma: `cat arquivo` Onde "arquivo" é um arquivo de texto qualquer. Imprime na tela o o conteúdo desse arquivo.

- `rm` Utilizado para remover arquivos: `rm arquivo`

Isso deve ser mais do que o suficiente para o leitor conseguir se virar na shell do cygwin.

Caso o leitor queira instalar ou desinstalar algum pacote, basta utilizar o setup. Talvez, mais tarde, o leitor tenha interesse em instalar um front-end gráfico para o GDB como o DDD, nesse caso, é uma boa idéia utilizar o setup para instalar esse programa.

2.3.2 MinGW

Em `http://sourceforge.net/project/showfiles.php?group_id=2435` aparece uma lista de arquivos presentes no MinGW, baixe o arquivo "Automated MingGW Installer". Com esse arquivo, o leitor poderá baixar todos os arquivos necessários da internet.

- Abra o programa baixado e clica em *Next*.

- Clique em "Download and Install" e clique em *Next*.

- Finja que leu a licença e clique em "I Agree"

- Clique em "Current"(se o leitor quiser software mais novo, mas potencialmente menos estável, pode clicar em Candidate) e depois clique em *Next*
- Se o leitor quiser selecionar mais algum programa além do "MinGW" base tools, basta clicar no nome do programa. Quando terminar de selecionar os programas, basta clicar em *Next*.
- Escolha a pasta em que o MinGW será instalado e clique em *Next*.
- Escolha uma pasta no menu "Iniciar" para adicionar uma entrada do MinGW, clique em "Install"
- Agora é só esperar um pouco. Quando terminar de baixar e instalar, clique em *Next* e depois em *Finish*.

Pronto, agora provavelmente o MinGW estará instalado. Para que o leitor possa utilizar o gcc a partir da linha de comando, os seguintes passos serão necessários:

- Vá no Painel de Controle e clique em Sistema
- Clique na aba "Avançado" e clique em "Variáveis do Ambiente"
- Na parte de variáveis do sistema, procure por uma variável que esteja escrito "Path" e clique em Editar.
- Muita calma nessa hora, não delete nada! Dentro do diretório de instalação do MinGW tem um pasta chamada "bin", adicione essa pasta à essa variável. Por exemplo, se o MinGW foi instalado em `c:\MinGW`, o leitor deve adicionar à variável Path a seguinte string: `;c:\MinGW\bin\`, lembrando que o ";" é necessário para separar cada entrada na variável "Path". `Clique`em ok e saia dessa janela. `Abra`um prompt de comando e digite "gcc"(sem aspas). Se aparecer `gcc: no input files`, está tudo ok. Caso contrário, tente reiniciar o computador e verifique se o diretório correto está listado na variável do sistema "Path"

Agora, falta só instalar o GDB. A instalação é um pouquinho mais chata, mas é relativamente simples. Primeiro, o leitor precisa instalar algum descompactador que consiga trabalhar com arquivos `.tar.bz2`, o 7-zip é uma boa opção e é recomendado.

Agora é só seguir os passos:

- Vá em `http://sourceforge.net/project/showfiles.php?group_id=2435` e clique em GNU Source-Level Debugger.
- Clique no arquivo que *não* contém *src* no nome. (a menos que o leitor queira compilar o GDB, nesse caso o leitor estará por conta própria). Baixe o arquivo em uma pasta qualquer
- Descompacte o arquivo em uma pasta qualquer. Note que existem várias pastas: `bin , include, info` e etc. Copie o conteúdo de cada uma dessas pastas para as pastas correspondentes no diretório onde o MinGW foi instalado.

- Abra um prompt de comando e digite o "gdb"(sem aspas). Se tudo deu certo, o gdb será inicializado. Para sair digite `quit`.

O GDB está instalado e pronto para ser utilizado.

2.3.3 Editores de Texto

Nas duas semanas que o autor ficou isolado fazendo os rascunhos iniciais desse texto, infelizmente só teve acesso a máquinas com Windows. Sendo assim, como ele não queria programar no bloco de notas(tarefa que apenas os mais sagazes e masoquistas conseguem), ele optou por utilizar o Notepad++.

O Notepad++ fornece uma interface agradável e tudo o que se espera de um editor de texto para programadores: syntax highlighting e outras coisas mais. Felizmente, o autor ficou isolado apenas durante duas semanas, sendo assim não houve oportunidade de testar mais editores de texto para Windows.

O leitor pode confiar na palavra do autor de que o Notepad++ é uma boa opção, ou pode optar também por olhar na Internet. Nesse caso, a página `http://en.wikipedia.org/wiki/Comparison_of_text_editors` oferece uma comparação de editores de texto.

2.4 Os exemplos do livro

Na pasta "exemplos", que o leitor deve ter recebido junto com este texto, há os exemplos que aparecem durante os capítulos. No início de cada programa aparece a o comando necessário para compilar. Naturalmente, para a compilação ser bem-sucedida, o leitor deve estar com o shell aberto na pasta que contém os exemplos.

Recomenda-se que o leitor leia pelo menos parte do manual do GCC que corresponde à versão instalada. Desse modo, ele passará a entender o significado dos parâmetros de compilação e das capacidades do compilador utilizado. `http://gcc.gnu.org/onlinedocs/`

De qualquer forma, o autor dará uma breve explicação sobre a compilação do primeiro exemplo (Cap3_ex1.c). O comando de compilação sugerido é `gcc Cap3_ex1.c -o Cap3_ex1 -Wall -g`. Vamos analisar por partes.

- `gcc Cap3_ex1.c` Esse trecho apenas está chamando o programa gcc passando como argumento o arquivo a ser compilado

- `-o Cap3_ex1` O `-o` serve para indicar o nome do arquivo executável a ser criado. Nesse caso o arquivo executável criado se chama `Cap3_ex1`, mas se o leitor quiser pode espeficar um outro nome qualquer. Se o usuário não espeficar nenhum nome, ou seja, omitir o `-o`, um arquivo chamado `a.out`(*assembler output*) será criado.

- `-Wall` Isso indica ao compilador que ele deve emitir todas os *warnings* sobre construções dúbias e pontos aonde o compilador acha que o programador possa ter comitido algum erro.

- `-g` Finalmente, essa opção faz com que o executável seja carregado com símbolos de debug e possa ser rodado num depurador como o gdb.

Para executar o programa digite no diretório onde o executável se encontra: `./Nome_do_programa`

Uma observação final sobre os exemplos do livro. Infelizmente, o compilador reclama quando encontra caracteres como "ç" ou "é", então no código em si, não utilize sinais de acentuação e etc. Mas, nos comentários o leitor deve tentar escrever corretamente.

2.5 O perigo das IDEs

IDEs(*Integrated Development Environment*) servem para auxiliar a tarefa de programação. No entanto, o autor recomenda que não se utilize IDEs no aprendizado de uma linguagem.

Geralmente, as IDEs (Dev-C++, Eclipse, Netbeans e muitos outros) criam makefiles e possuem regras próprias de compilação. Para uma linguagem como o C, que a compilação é uma etapa *crucial*, esconder isso do programador iniciante talvez faça com que ele fique dependente da IDE.

Principalmente para os programadores de C, é importante entender os comandos e os parâmetros de compilação, além de entender como se compila um projeto separado em módulos. O autor observe que muitos programadores sabem programar em uma IDE específica e ficam completamente perdidos ao mudar de ambiente.

O objetivo desse texto é tornar o programador apto para programar em C em qualquer ambiente de programação, por isso propositalmente, o autor escolheu um ambiente o mais *espartano* possível. Aprender a utilizar uma IDE depois será fácil.

O leitor deve notar também, que o autor não é contra o uso de editores de texto turbinados que completam a sintaxe das palavras e outras funções que agilizam a tarefa de programar. O autor é contra esconder o processo de compilação do programador iniciante.

Capítulo 3

Tipos de dados primitivos

Como é de praxe, começaremos com o *Hello world!*

Exemplo 1 - Cap3_ex1.c

```
1  /*
2   * Cap3_ex1.c
3   * gcc  Cap3_ex1.c -o Cap3_ex1 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){/*início da função main*/                                    main
9          printf("Hello World!");
10
11         return 0;
12 }/*fim da função main*/
```

Como é de se esperar, ao executar o programa, aparecerá na tela do terminal a frase *Hello World!*. Esse programa, apesar de enxuto, revela muitas coisas interesantes sobre o C, portanto o dissecaremos linha a linha:

- `include <stdio.h>`: Todos os comandos começados por `#` indicam comandos passados para o *pré-processador*. Antes do arquivo-fonte ser compilado, *todas* as diretivas de pré-processamento são resolvidas. A diretiva `#include` serve para incluir arquivos no código-fonte(isso foi razoavelmente óbvio...). Nesse caso, estamos incluindo o *arquivo cabeçalho stdio.h* que corresponde à biblioteca *stdio*(Standard IO) que contém diversas funções para facilitar a entrada/saída e é definida em diversos padrões. Por hora, basta saber que se desejamos utilizar uma função de uma biblioteca é necessário incluir o arquivo (ou arquivos) cabeçalho correspondentes. [1]

- `int main(void)`:Essa é a função principal do programa. Cada programa só deve possuir uma única função main e 1 (um) deve ser o número de funções *main* no programa. Nem 2 ou 3. 4 está fora. [2] Mas, o que a função *main* tem de especial? Quando o programa é carregado em memória, a primeira função a ser chamada é a *main*. Conforme veremos adiante, o formato de uma função é:

[1] Arquivos-cabeçalhos... Em inglês "header files"

[2] Alusão infame à Monty Python.

tipo nome_da_função(*tipo* argumento1, *tipo* argumento2, . . . , *tipo* argumento$N$). Naturalmente, a palavra "tipo" especifica um *tipo de dado*, que é o alvo principal deste capítulo. O "tipo" que é *anterior* ao nome da função representa o retorno dela, isto é, qual o tipo de valor que a função retorna.

Bom... Com essa informação é possível deduzir que a função `main` retorna um inteiro (`int`). E o `void` dentro dos (), indica que a função *não* recebe argumentos.

- `printf("Hello World!");`: Essa linha é razoavelmente intuitiva. É essa a linha que produz o *"Hello World!"* que aparece no terminal. Porém há algumas sutilezas que o leitor não deve deixar de notar:
  - `printf` (*Print Formatted*) é uma função e recebe uma *string* como argumento. Uma *string* em C é delimitada por "" e conforme o leitor verificará bem mais a frente, nada mais é do que um *vetor de caracteres*.
  - Esta função está presente na biblioteca *stdio* e isso justifica a inclusão do arquivo cabeçalho correspondente, `stdio.h`.
- `return 0;`: O leitor deve lembrar que conforme foi dito anteriormente o retorno da função `main` é um inteiro (`int`). Se a função retorna *alguma coisa*, então em *algum lugar* precisa estar especificado esse retorno. É aí que o comando `return` entra. Ele especifica o *retorno* da função. Mas pera lá... A função main está retornando esse valor 0 para *quem*? Para o sistema operacional. É uma convenção que o retorno 0 indique que o programa finalizou corretamente.

Há algumas coisas mais que são dignas de menção:

- Um bloco de código é envolto por { }(e não por `Begin` e `End`, como no Pascal.)
- Todos os comandos(*statements*) são finalizados por ;
- Comentários são iniciados por `/*` e finalizados por `*/`.

Com esse exemplo, conseguimos descobrir muita coisa sobre o C, mas ainda não vimos declaração de variáveis, nem atribuições. O próximo exemplo ilustrará esses conceitos:

Exemplo 2 - Cap3_ex2

```
1  /*
2   * Cap3_ex2.c
3   * gcc  Cap3_ex2.c -o Cap3_ex2 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){/*início da função main*/                                    main
9          int a,b;
10         int resultado;
11         float dinheiro;
12
13         a = 2;
```

```
14          b = 3;
15          resultado = a + b;
16          dinheiro = 9.99;
17          printf("O resultado de a + b eh %d.\n",resultado);
18          printf("Eu tenho %f reais.\n",dinheiro);
19          return 0;
20  }/*fim da função main*/
```

Embora, semanticamente este exemplo não faça o menor sentido, ele é extremamente valioso para explicar diversos conceitos. De modo geral, a declaração de variáveis possui o seguinte formato:
tipo_da_variável nome_da_variável;

Onde o "tipo_da_variável" indica um tipo qualquer de dado. Note que se desejar declarar várias variáveis do mesmo tipo, basta separar cada nome de variável com ",".

A declaração de variáveis no começo do bloco de código de funções reserva um espaço na memória para aquelas variáveis. E esse espaço fica reservado até o *término* da função(o que no caso da função *main* fatalmente implica no término do programa). Note que nesse exemplo, estamos alocando variáveis de forma *estática*. Alocação dinâmica é um tópico um pouco mais avançado que será discutido em uma etapa posterior.

Depois que uma variável estiver corretamente declarada, é possível atribuir valores à essa variável utilizando o "=". Embora, não tenhamos discutido expressões aritméticas, elas são razoavelmente intuitivas, como mostra o exemplo.

Alguns detalhes importantes que o leitor não deve deixar de perceber

- Note que utilizamos o tipo *float*, que nada mais é do que um número em ponto flutuante com precisão simples. Note que a casa decimal, na atribuição, é separada por "."

- Um cuidado simples, mas que seguido evitará horas de frustração depurando o programa, é **atribuir à variáveis valores que correspondam ao seu tipo**. O autor gostaria de deixar extremamente claro a **importância** dessa medida. Isso não significa que não se possa atribuir uma variável `int` a um float . Se o programador realmente quiser fazer isso, faça um *cast* explícito. Ainda nesse capítulo o leitor aprenderá mais sobre *casts*.

- O `\n` que aparece dentro das strings utilizadas nas duas ocorrências de `printf()` indicam "new line", ou seja, uma nova linha. Isso insere uma nova linha na saída. Ao processar a string, a função "enxerga" isso como um código para pular para a próxima linha.

Além desses dois detalhes há uma sutileza que talvez o leitor não tenha percebido e por isso a próxima seção tratará disso.

3.1 lolz, printf is f0r the l33t!

Imagino que todo autor de um livro que sirva ao propósito de *ensinar* o C se depara com um dilema. `printf()` é uma função que possui diversos conceitos intricados e não é uma função comum. Para explicar 100% o funcionamento do `printf()`, seria necessário que o leitor já tivesse algum conhecimento sobre C,

principalmento no que tange as funções *vararg*. Mas, se o leitor está lendo um livro desses, provavelmente é porque *não* possui tal conhecimento. Por outro lado, se o autor não mostrar o `printf()`, até o leitor terminar de aprender funções *varargs*, nenhum programa dele possuirá saída na tela! Fato que pode ser bastante frustrante para o leitor.

Bom... o que a maioria faz? Simplesmente *não* explica e o leitor fica com a impressão que tem algo mágico por de trás da função ou que é alguma bruxaria ou outra coisa igualmente escabrosa. Sendo assim, eu proponho ao leitor uma explicação. Naturalmente eu não entrarei em detalhes profundos, mas espero que seja um explicação *satisfatória* de todo modo.

O leitor deve recordar do primeiro exemplo. Nele, a função `printf()` foi chamada com um *único* argumento. Esse argumento foi uma string. Mas no segundo exemplo, em cada uma das ocorrências a função `printf()` foi chamada com *2* argumentos! Isso indica que a função `printf()` é uma função *vararg*, isto é, possui uma número variável de argumentos.

Na primeira ocorrência do `printf()` ao invés de aparecer `%d` na tela, apareceu o valor da variável `resultado`. E na segunda ocorrência, ao invés de aparecer `%f` na tela, apareceu o valor da variável `dinheiro`.

A essa altura o leitor deve ter percebido que `%d` e `%f` são códigos que indicam o tipo de variável a ser impressa. Quando a função `printf()` processa o primeiro argumento (que sempre deve ser uma vetor de caracteres, ou seja, uma string) se algum token(palavra) for um %, o *próximo* argumento passado à função é processado e ao invés do código (% + uma letra), o valor da variável é impresso.

Note que o programador deve zelar para que cada variável esteja corretamente associado ao código. Vamos supor que no Exemplo 2, no segundo `printf(),` o programador tivesse substituído o `%f` por um `%d`. O que aconteceria? Bom... Ao processar a string passada como primeiro argumento, a função iria encontrar o `%d` . Então, a função iria buscar o segundo argumento e iria "lê-lo" como um *inteiro* e o resultado seria imprevisível.

Os exemplos abaixo servem para ilustrar o uso do `printf()` e aproveitar para ilustrar mais alguns aspectos da declaração de variáveis.

Exemplo 3 - Cap3_ex3

```
1  /*
2   * Cap3_ex3.c
3   * gcc Cap3_ex3.c -o Cap3_ex3 -Wall -g
4   *
5   */
6  int main(void){/*início da função main*/                                    main
7          int foo1=2,foo2;
8          float dinheiro=10.0;
9          int idade = 18;
10
11         foo2=3;
12
13         printf("%d + %d eh igual a %d\n",
14                 foo1,foo2,foo1 + foo2);
15         printf("Eu tenho %f reais e eu tenho %d anos\n",
16                     dinheiro,idade);
17         return 0;
18 }/*fim da função main*/
```

Nesse exemplo, o autor gostaria de chamar atenção para a declaração de variáveis. Note que as variáveis `foo1` e `dinheiro` foram inicializadas na própria declaração de variáveis. Para isso, basta colocar o "=" seguido do valor da inicialização logo após o nome da variável.

Em relação aos dois comandos `printf()` que aparecem no programa, o primeiro possui **4** argumentos e o segundo possui **3** argumentos. Ao executar o programa, provavelmente aparecerá na tela:
2 + 3 eh igual a 5
Eu tenho 10.000000 reais e eu tenho 18 anos

Como foi o dito anteriormente, cada "código" é substituído por um argumento passado para a função. Note que "variável" e "argumento" foi usado de forma indistinta, mas o argumento não necessariamente precisa ser uma variável. O quarto argumento do primeiro `printf()`é uma expressão aritmética. Na verdade, o argumento pode ser um valor numérico também, como o próximo exemplo ilustra.

Exemplo 4 - Cap3_ex4

```
1  /*
2   * Cap3_ex4.c
3   * gcc  Cap3_ex4.c -o Cap3_ex4 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9
10         printf("O C pode ser utilizado como uma calculadora\n");
11         printf("Uma calculadora bem barata. . .\n");
12         printf("660 dividido por 12 = %f\n",660.0/12.0);
13
14         return 0;
15 }
```

Note que foram passados para a função `printf()` valores em ponto flutuante e portanto, foi utilizado o código correspondente `%f`.

3.1.1 Colocando a cabeça para funcionar

Imagine que o leitor esteja projetando uma nova linguagem de programação chamada **XYZ** e que quisesse dar a opção para o programador de criar funções que possam receber um número variável de argumentos. Certamente, a linguagem iria precisar de um meio de informar ao programador o número de argumentos passados às funções, não é? Bom, não necessariamente.

Na verdade, a única coisa que o programador *precisa* é de um meio de de "recuperar" cada argumento. Mas cada argumento pode ser de *tipos* e *tamanhos* diferentes! Então, um procedimento que apenas leia um número *fixo* de bytes não serve.

Uma saída seria deixar a cargo do programador descobrir o *tipo* de cada argumento e a quantidade e fornecer um procedimento que leia uma quantidade de bytes especificada pelo programador. Note que o tamanho de um tipo de dado é um informação que está sempre disponível, então esse procedimento não precisa nem receber a quantidade de bytes, poderia receber como argumento o

próprio tipo do dado. Tendo o tipo de dados, o procedimento teria meios de descobrir a quantidade de bytes a serem lidos.

É basicamente isso que o C faz. O programador da função *vararg* deve descobrir a quantidade e o tipo de cada argumento por si só.

Vamos mudar o enfoque agora. Vamos supor que o leitor seja o programador da função vararg. Ele tem o procedimento para "ler" os argumentos, mas ele precisa de um meio de descobrir quantos são e de que tipo são.

O leitor deve lembrar que a função printf sempre recebia *pelo menos* uma string. É aí que está o segredo. Essa string identifica o *número* e o tipo de cada argumento através dos códigos começados por %.

Em C as funções *varargs* devem receber pelo menos 1 argumento fixo e preferencialmente, esse argumento deve fornecer meios de identificar os outros argumentos.

3.2 Alguns tipos de dados

Espero que nesse ponto o leitor já esteja familiarizados com a declaração de variáveis pelo menos do tipo `float` e `int`. Na verdade, a declaração para todos os outros tipos funciona basicamente da mesma forma.

3.2.1 Inteiros

O que normalmente se chama de "inteiros" equivale, em C, às variáveis do tipo `int`. Como foi dito, anteriormente, o `int` possui o tamanho de uma "palavra" do computador. Para um processador de 32-bits, uma "palavra" possui 4 bytes.

Como descobrir, então, os limites de um `int` de 32-bits? Um dos bits representa o sinal, então sobram 31-bits para representar o número[3]. Então, como cada bit pode assumir apenas 2 valores, o número total de combinações é $2^{31} = 2147483648$. Então o maior `int` possível seria $+2147483648$ e o menor seria -2147483648? Não exatamente, o menor `int` é esse mesmo, mas não devemos esquecer que precisamos considerar o 0, portanto subtraímos 1 do maior `int` possível.
Sumarizando os resultados:
Maior `int` possível em uma máquina de 32-bits: $+2147483647$
Menor `int` possível em uma máquina de 32-bits: -2147483648

Se em algum momento uma variável `int` receber um valor *maior* do que o máximo, fala-se em *overflow* e se receber um valor menor do que o mínimo, fala-se em *underflow*. Para os dois casos, os resultados são imprevisíveis. Então o autor aconselha que não se utilizem variáveis do tipo `int` caso haja um risco razoável *overflow* ou *underflow*.

"Holy moly, e se eu precisar utilizar um valor maior do que o máximo?", indaga o leitor. Nesse caso, há algumas possibilidades:

- Utilizar o tipo `long int`
- Utilizar o tipo `long long int` (apenas se for programar em ISO C99)

[3] A rigor, a forma de se representar números inteiros utilizada é a chamada *complemento de dois*. E o bit de sinal, na verdade também faz parte do cálculo do valor do número. Veja `http://en.wikipedia.org/wiki/Two's_complement`

- Utilizar uma biblioteca que ofereça números com precisão arbitrária como o **GMP** (*Gnu Multi-Precision Library*). [4]

O leitor já viu nos exemplos anteriores como declara inteiros, como fazer atribuições e etc. Porém, um exemplo a mais nunca faz mal a ninguém.

Exemplo 5 - Cap3_ex5

```
1  /*
2   * Cap3_ex5.c
3   * gcc Cap3_ex5.c -o Cap3_ex5 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          /*valor foi inicializado com 2*/
10         int valor = 2;
11
12         printf("O valor da variável valor eh %d\n",valor);
13
14         /*atribui-se 15 à variável valor*/
15         valor = 15;
16
17         printf("O valor da variável valor agora eh %d\n",valor);
18
19         /*imprime-se o valor de valor+1*/
20         printf("valor + 1 = %d\n",valor+1);
21
22         return 0;
23 }
```

Esse exemplo teve como objetivo apenas reforçar os conceitos já ilustrados. Note que a variável `valor` foi inicializada com 2, depois foi atribuído o valor de 15 e depois efetuou-se a operação de `valor` + 1. Note que essa operação *não* afeta o conteúdo da variável valor.

De modo geral, inicializar variáveis é uma boa prática. O programador ***não*** deve assumir que as variáveis são inicializadas por padrão. Assumir isso pode ser um fonte de erros. O exemplo abaixo, ilustra isso.

Exemplo 6 - Cap3_ex6

```
1  /*
2   * Cap3_ex6.c
3   * gcc Cap3_ex6.c -o Cap3_ex6 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int teste;
10
11         printf("O valor da variavel valor eh: %d\n",teste);
12
13         return 0;
14 }
```

[4] `http://gmplib.org/`

Ao rodar esse exemplo da máquina do autor o resultado foi o seguinte: `O valor da variavel valor eh: -1208924160`. Muitos programadores iniciantes pensam que as variáveis são inicializadas com 0 ou algo do tipo mas na verdade no C as variáveis são "inicializadas" com que estiver ocupado aquele espaço da memória que o sistema reservou para a variável.

3.2.2 Números reais

No C, existem dois tipos de dados primitivos que representam os números reais: `float` e `double`. No padrão ANSI, as operações em ponto-flutuante devem seguir o padrão IEEE 754. Bom, eu não vou reinventar a roda, se o leitor desejar entender como funcionam os números em ponto flutuante nesse padrão basta entrar em `http://en.wikipedia.org/wiki/IEEE_754`. De todo jeito, uma breve explicação pode ser encontrada na seção 4.3.1

O importante, por agora, é que o `float` é um número com precisão simples(seis dígitos de precisão) e o `double` com precisão dupla(dez dígitos de precisão).

O exemplo abaixo ilustra o uso do tipo `double` e do tipo `float`.

Exemplo 7 - Cap3_ex7

```
1  /*
2   * Cap3_ex7.c
3   * gcc  Cap3_ex7.c -o Cap3_ex7 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          float teste_float;
10         double teste_double;
11
12         teste_float = 5.5;
13         /*utilizando notação científica para atribuir valores*/
14         teste_double = 1.0E5;
15
16         printf("teste_float = %f\n",teste_float);
17         printf("teste_double = %f\n",teste_double);
18
19         return 0;
20 }
```

Note que o código `%f` serve tanto para `double` quanto para o `float`. Note também, que pode-se utilizar o estilo de notação científica para atribuir valores às variáveis em ponto flutuantes. Nesse caso, o número após o `E` indica o expoente da base 10.

Assim como com o tipo `int` é possível efetuar as operações aritméticas básicas com os tipos em ponto flutuante de modo análogo, como ilustra o exemplo abaixo.

Exemplo 8 - Cap3_ex8

```
1  /*
2   * Cap3_ex8.c
3   * gcc  Cap3_ex8.c -o Cap3_ex8 -Wall -g
4   *
5   */
```

```
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          float a,b,c;
10         double num1,num2,resultado;
11
12         a = 2.0;
13         b = -5.15;
14         c = a/b;
15
16         num1 = 17.0;
17         num2 = -3.512E1;
18         resultado = num1*num2;
19
20         printf("c = %g\n",c);
21         printf("resultado = %g\n",resultado);
22
23         c = c - b;
24         resultado = resultado + num1;
25
26         printf("c = %g\n",c);
27         printf("resultado = %g\n",resultado);
28
29         return 0;
30 }
```

Note que dessa vez utilizou o operador `%g` ao invés de `%f`. Ambos funcionam apenas para números e ponto flutuante. A diferença é que o `%g` imprime apenas a quantidade mínima de casas decimais necessárias, enquanto o `%f` imprime uma quantidade *fixa* de casas decimais. Conforme veremos numa etapa posterior, é possível controlar a quantidade de casas e a formatação dos números impressos com `printf`.

Outro detalhe interessante é que se o leitor deseja atribuir números que não possuam casas decimais a uma variável que seja em ponto flutuante recomenda-se que utilize o ".0" para deixar claro que é um número decimal. Ou seja, faça assim `double a = 5.0;` e não assim `double a = 5;`.

3.2.3 Caracteres

Curiosamente, no C, os caracteres também são tipos numéricos. Na verdade, o que um caracter guarda é o valor que corresponde àquele caracter na tabela ASCII.

Os caracteres são representados pelo tipo `char`. E embora tenha sido dito anteriormente que o tipo char é um tipo numérico, a menos que se tenha uma razão muito boa não se deve efetuar operações aritméticas com caracteres.

Exemplo 9 - Cap3_ex9

```
1  /*
2   * Cap3_ex9.c
3   * gcc  Cap3_ex9.c -o  Cap3_ex9 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          char c;
10
```

```
11          c = 'D';
12
13          printf("Imprimindo o caracter c: %c\n",c);
14          printf("Imprimindo o codigo ASCII correspondente: %d\n",c);
15          /*putchar é uma outra forma de imprimir um caracter*/
16          putchar(c);
17
18          return 0;
19  }
```

A declaração de um `char` segue tudo que nós já vimos até agora. Para um char ser atribuído ele deve estar entre aspas simples (‘’) e não aspas duplas. Aspas duplas, como veremos mais tarde, delimitam **strings**.

De modo análogo se o programador quisesse inicializar a variável na declaração bastaria ter feito `char c = 'D';` A grande novidade aqui pode ser a utilização de uma função nova a `putchar`. Essa função recebe um `char` e imprime na saída padrão esse caracter.

Note que as duas ocorrências do `printf` receberam como segundo argumento a variável `c`. Na primeira, o código passado foi `%c`, que é código que corresponde aos caracteres normais. E na segunda, o código passado foi `%c`, que é código que corresponde a números inteiros. O resultado disso foi que na segunda ocorrência, apareceu na tela 68 que é justamente o código na tabela ASCII para o caracter “D”.

O leitor deve recordar do `\n` utilizado para pular uma linha. Apesar de possuir “dois” caracteres, é na verdade um único caracter e possui um código na tabela ASCII. Além do `\n` existem outros caracteres que servem ao propósito de formatação como o `\t` que aplica uma tabulação horizontal.

Exemplo 10 - Cap3_ex10

```
1   /*
2    * Cap3_ex10.c
3    * gcc  Cap3_ex10.c -o  Cap3_ex10 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           char nova_linha = '\n';
10          char tab = '\t';
11
12          printf("Observe este printf\n");
13          printf("\tEste ficou mais deslocado\n");
14
15          /* podemos obter o mesmo resultado com mais trabalho da seguinte
16           * forma
17           */
18          printf("Observe este printf%c",nova_linha);
19          printf("%cEste ficou mais deslocado%c",tab,nova_linha);
20
21          return 0;
22  }
```

Note que os dois pares de `printf` produzem o mesmo resultado. Por enquanto, manipular apenas caracteres pode não parecer muito útil, no entando, a verdadeira utilidade será verificada quando o autor mostrar *strings*, que são apenas *vetores* de caracteres.

3.2.4 Outros tipos de dados

Na verdade, esses não são os únicos tipos de dados primitivos que a linguagem possui, mas por ora esses tipos bastam para o leitor conseguir efetuar um número grande de tarefas. Os outros tipos serão introduzidos aos poucos, já que muitas vezes eles são utilizados para tarefas muito específicas.

3.3 Alguns operadores e expressões aritméticas

O C é muito rico em operadores. O objetivo por agora não é discutir *todos* os operadores e sim alguns que o leitor ache útil. As 4 operações aritméticas estão representadas através de `+ - / *`. O exemplo abaixo ilustra alguns aspectos das expressões aritméticas.

Exemplo 11 - Cap3_ex11

```
1  /*
2   * Cap3_ex11.c
3   * gcc Cap3_ex11.c -o Cap3_ex11 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a = 4;
10         int b = 2;
11         int c;
12
13         /*soma a e b e guarda o resultado em c*/
14         c = a + b;
15         printf("%d + %d = %d\n",a,b,c);
16
17         /*subtrai a de b e guarda o resultado em c*/
18         c = a - b;
19         printf("%d - %d = %d\n",a,b,c);
20
21         /*multiplica a por b e guarda o resultado em c*/
22         c = a * b;
23         printf("%d * %d = %d\n",a,b,c);
24
25         /*divisão a por b e guarda o resultado em c*/
26         c = a / b;
27         printf("%d / %d = %d\n",a,b,c);
28
29         return 0;
30 }
```

Não tem muito segredo aqui, não é? Esses operadores funcionam da mesma forma para os outros tipos numéricos como `float` e `double`. No entanto para os tipos inteiros, alguns cuidados são necessários no caso de operações que o resultado pode *não* ser um tipo inteiro, como uma divisão.

Veja o exemplo abaixo:

Exemplo 12 - Cap3_ex12

```
1  /*
2   * Cap3_ex12.c
3   * gcc Cap3_ex12.c -o Cap3_ex12 -Wall -g
```

```
4   *
5   */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int a = 2;
10          int b = 3;
11          int c;
12
13          c = a / b;
14          /*note que a divisão deu como resultado 0*/
15          printf("%d / %d = %d\n",a,b,c);
16
17          c = b / a;
18          /* note que o valor não é arrendondado. ao invés disso
19           * o valor é truncado
20           */
21          printf("%d / %d = %d\n",b,a,c);
22          return 0;
23  }
```

Note que o leitor *não* deve esperar valores *arrendondados* na divisão e sim valores *truncados*, ou seja, apenas descarta-se a parte decimal. Existe algum modo de fazer com o que resultado correto seja obtido? Sim! E nós veremos na próxima seção quando discutirmos *casts*.

Por enquanto vamos discutir alguns operadores que só podem ser utilizados com tipos inteiros (basicamente `int` e `char`). Veja o próximo exemplo.

Exemplo 13 - Cap3_ex13

```
1   /*
2    * Cap3_ex13.c
3    * gcc Cap3_ex13.c -o Cap3_ex13 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int a = 6;
10          int b = 3;
11          int c = 5;
12          int d;
13
14          /* c mod b. Essa expressão retorna o resto da divisão de c por b*/
15          d = c % b;
16          printf("%d mod %d = %d\n",c,b,d);
17
18          d = b % a;
19          printf("%d mod %d = %d\n",b,a,d);
20
21          d = a % b;
22          printf("%d mod %d = %d\n\n",a,b,d);
23
24          /*o valor de d agora passa a ser 1*/
25          d = 1;
26          printf("d = %d\n",d);
27          d++;
28          /*o operador ++ incrementa 1 à variavel d*/
29          printf("d = %d\n",d);
30
31          d--;
```

```
32        /*o operador – decrementa 1 à variável d*/
33        printf("d = %d\n",d);
34
35        return 0;
36  }
```

Então, para calcular o resto de uma divisão entre dois números *inteiros*, pode-se utilizar o operador %. Além diss, é muito comum em algum trecho de programa o programador precisar incrementar(ou decrementar) 1 à uma determinada variável. Ao invés de fazer `d = d + 1` pode-se fazer `d++` ou `d--` se o objetivo for decrementar.

Note que esses operadores funcionam apenas para tipos de dados que representam números *inteiros*, então eles não podem ser utilizados com variáveis do tipo `float` ou `double`.

3.3.1 Precedência e Associatividade

Quando o compilador se depara com uma expressão como $2 + 3 * 5/2 + 4$, ele precisa decidir qual é ordem que cada par de expressões será avaliada. Sendo assim, as linguagens de programação geralmente possuem uma lista de precedência indicam qual operador deve ser "resolvido" primeiro.

Além disso, para operadores de mesma precedência ele precisa de uma lista que indique o tipo de associatividade, isto é, se as operações de mesma precedência serão resolvidas da esquera para direita ou da direita para a esquerda.

De todo modo, o programador sempre pode forçar a avaliação de uma determinada expressão colocando "()" entre a expressão. E de modo geral, em operações complexas mesmo que a expressão esteja organizada corretamente de acordo com as precedências dos operadores, é uma boa prática colocar os "()" para facilitar a leitura do código-fonte.

No C, a adição e subtração tem menor precedência do que multiplicação, divisão e a operação de resto(%). E todos essas possuem associativade da esquerda para a direita. Já o operadores de incremento(`++`) e de decremento (`--`) possuem uma precedência maior do que a dos operadores já citados.

O exemplo abaixo ilustra alguns aspectos. Note que como os exemplos incluem divisão, utiliza-se variáveis do tipo `float` para evitar problemas com truncamento.

Exemplo 14 - Cap3_ex14

```
1   /*
2    * Cap3_ex14.c
3    * gcc Cap3_ex14.c -o Cap3_ex14 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9         float a = 1.0, b = 2.0, c = 3.0;
10        float d;
11        /* tanto a divisão quanto a multiplicação possuem a mesma
12         * precedência, sendo assim, qual o resultado dessa expressão?
13         */
14        d = a / b * c;
15        printf("%g / %g * %g = %g\n",a,b,c,d);
16
```

```
17          /* Essa expressão foi interpretada da seguinte forma */
18          d = (a / b) * c;
19          printf("(%g / %g) * %g = %g\n",a,b,c,d);
20
21
22           /*Agora um exemplo um pouco mais complicado*/
23          d = (a + b) / c * c + a - 2.0 / 3.0;
24          printf("(%g + %g) / %g * %g + %g - 2.0 / 3.0 = %g\n",a,b,c,c,a,d);
25
26          /*Essa expressão pode ser escrita da seguinte forma*/
27          d = ( ( (a + b) / c) * c) + a - (2.0 / 3.0);
28          /*Observe que o resultado é o mesmo*/
29          printf("(( (%g + %g) / %g) * %g) + %g - (2.0 / 3.0) = %g\n",a,b,c,c,a,d);
30
31          return 0;
32  }
```

Note que o uso de "()" produziu o mesmo resultado e torna a expressão um pouco mais legível, dessa forma é sempre bom utilizar os "()". Outro ponto interessante é que conforme foi dito anteriormente apesar do .0 não ser estritamente necessário nas constantes 2.0 e 3.0, é uma boa prática pois deixa explícito que estamos trabalhando com números em ponto-flutuante e evita a promoção de tipos automática que a linguagem efetua.

Nesse ponto talvez ainda não esteja muito claro para o leitor o que significa associatividade da esquerda para direita. Vamos supor que nós queremos descobrir qual é a ordem em que devemos avaliar a seguinte expressão:

$x = 5 * 10/5 * 3/8 * 20$ Vamos supor que a precedência dos operadores seja a mesma do C. Nesse caso, tanto o $*$ quando o / possuem a mesma precedência. Porém, sabemos que associatividade é da esquerda para a direita. Então, a operação que contém o operador *mais* a esquerda é efetuada primeiro: $x = (5 * 10)/5 * 3/8 * 20$.

Então, depois que a operação for efetuada a expressão restante será: $x = 50/5 * 3/8 * 20$. Novamente, resolvemos a operação que contém o operando mais a esquerda: $x = (50/5) * 3/8 * 20$ e ficamos com $x = 10 * 3/8 * 20$. E continuamos esse processo até a resolução final da expressão.

Vamos supor agora que o leitor queira parentetizar toda a expressão, isto é colocar "()" de modo que respeite as precedências e as associatividades. Nesse caso ficaria da seguinte forma: $x = (((((5 * 10)/5) * 3)/8) * 20)$. Como o leitor deve saber, nesse caso, o "()" mais interno é resolvido primeiro. Dessa forma, podemos obter o mesmo resultado com uma expressão um pouco mais legível.

3.4 *Casts*

Casts(coerção) é uma forma de forçar uma expressão a ser do tipo que o programador quer. Note que em C, as expressões sempre são feitas com os mesmos tipos de dados. Se um dos operandos for de um tipo diferente há uma *promoção* de tipos.

Idealmente, um programador nunca deveria atribuir à uma variável um valor que não corresponda ao seu tipo. Mas ocasionalmente há essa necessidade e é por isso que os casts existem. Note que ainda sim, o autor recomenda que se possível deve-se fazer atribuições que não envolvam *casts*. Se houver necessidade

de um *cast* o programador preferencialmente deve fazê-lo de forma explícita e não utilizar o *cast* implícito que o C possui.

A razão disso? Deixar claro no código-fonte a intenção do programador. *Casts* implícitos muitas vezes são fontes de bugs. O problema é que nem sempre o programador percebe que em determinada expressão ocorreu um cast. Ao tornar explícito o cast, *pelo menos* naquelas expressões o programador está assegurando que o cast de fato precisa acontecer.

Imagine que o leitor precisasse ler um código em que ocorresse um cast implícito. Será que dependendo da situação não poderia haver dúvida se aquele cast *realmente* era para acontecer ou foi descuido do programador?

Exemplo 15 - Cap3_ex15

```
1  /*
2   * Cap3_ex15.c
3   * gcc Cap3_ex15.c -o Cap3_ex15 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a = 1;
10         float b = 2.0;
11         double c = 15.0;
12
13         /* a é uma variável do tipo int
14          * no entanto, b é uma variável do tipo float,
15          * então o C converte implicitamente o valor de a
16          * para float e atribui esse valor ao b.
17          */
18         b = a;
19         printf("b = %g\n",b);
20
21         /* b é uma variável do tipo float e c é uma variável
22          * do tipo double. Nesse caso também há uma promoção de tipos,
23          * já que b possui menos precisão. O valor de b é convertido
24          * para double.
25          */
26         c = b;
27         printf("c = %g\n",c);
28
29         return 0;
30 }
```

Nesse exemplo, mostra-se alguns exemplos simples de expressões onde ocorrem *casts implícitos*. Note que numa expressão como $c = b$;, onde `b` é um `float` e `c` um `double`, a variável b não passa a ser um `double`. O que acontece é que o valor que está "guardado" na variável passa a ser um `double` para que a atribuição possa ser efetuada. *Casts* não mudam os tipos das variáveis, *casts* mudam os tipos das expressões.

No exemplo acima, o leitor deve notar que houve apenas *promoções* de tipo, isto é o valor foi convertido de um tipo "menor" para um tipo "maior" ou com mais precisão. Mas o contrário pode acontecer, é possível converter um `float` para `int` como mostra o exemplo abaixo.

Exemplo 16 - Cap3_ex16

```
1  /*
```

```
2   * Cap3_ex16.c
3   * gcc Cap3_ex16.c -o Cap3_ex16 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a;
10         float b;
11
12         b = 2.3;
13         /* ao fazer a = b, estamos promovendo um cast implícito e
14          * haverá perda de precisão pois uma variável do tipo int não
15          * possui parte decimal.
16          */
17         a = b;
18         printf("a = %d\n",a);
19
20         /*
21          * Agora estamos fazendo o contrário. Note que a precisão não
22          * é "recuperada"
23          */
24         b = a;
25         printf("b = %g\n",b);
26
27         return 0;
28 }
```

Inicialmente atribui-se 2.3 para a variável `a`, que é um `float`. Depois atribui-se à variável `b` à variável `a`, que é um `int`. Ao imprimir na tela com `printf` apareceu 2, como era de se esperar, já que variáveis do tipo `int` não consegue representar a parte decimal dos números reais. Porém ao fazer o processo inverso, atribuir `a` à `b` a precisão não foi recuperada. Esse que é o perigo dos *casts*.

Agora que o leitor já viu alguns *casts implícitos*, é hora de mostrar os *casts explícitos*.

Exemplo 17 - Cap3_ex17

```
1  /*
2   * Cap3_ex17.c
3   * gcc Cap3_ex17.c -o Cap3_ex17 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a = 3;
10         int b = 2;
11         float c;
12         /*note que o valor da expressão é truncado*/
13         c = a / b;
14         printf("c = %g\n",c);
15         /* é essencialmente o mesmo exemplo anterior só que constantes ao
16          * invés de variáveis
17          */
18         c = 3 / 2;
19         printf("c = %g\n",c);
20
21         /* nos dois casos anteriores foi impresso na tela 1. O que não é
22          * o que nós esperamos. Dessa forma, podemos forçar o tipo da
23          * expressão para que ao invés de uma divisão inteira, seja feito
```

```
24          * uma divisão em ponto-flutuante.
25          */
26         c = (float) a / b;
27         printf("c = %g\n",c);
28         /* nesse caso específico, o cast poderia ter sido evitado se
29          * utilizasse constantes em ponto-flutuante.
30          */
31         c = 3.0 / 2.0;
32         printf("c = %g\n",c);
33
34         return 0;
35 }
```

O leitor talvez esteja se perguntando porque nas duas primeiras atribuições o resultado obtido foi 1 e não 1.5. A resposta é simples: foram efetuadas divisões *inteiras* e a parte decimal foi truncada. Na terceira atribuição efetuou-se um *cast* explícito para *forçar* a expressão a ser avaliada como ponto-flutuante.

A quarta atribuição mostra uma coisa interessante: 2.0/3.0 é diferente de 2/3. Mais uma vez, o autor ressalta que ao trabalhar com números em ponto-flutuante o .0 é *essencial* para destacar o fato de serem números em ponto-flutuante e não números inteiros. E como o leitor viu, faz toda a diferença.

Para efetuar um cast explícito basta fazer `(tipo) expressão`. Onde "tipo" é um tipo de dado qualquer que se deseja converter a expressão.

Exemplo 18 - Cap3_ex18

```
1  /*
2   * Cap3_ex18.c
3   * gcc Cap3_ex18.c -o Cap3_ex18 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8   int main(void){                                                    main
9         int a = 5;
10        float b;
11        double c;
12
13        /* note o uso do cast para que seja obtido o valor correto da
14         * expressão. O leitor é convidado a testar o resultado obtido
15         * sem o cast
16         */
17        b = (float) a/2 + 3;
18        printf("b = %g\n",b);
19
20        /*
21         * Mesmo quando é uma atribuição de um tipo em ponto flutuante
22         * para outro tipo em ponto flutuante é bom utilizar o cast.
23         */
24        c = (double) b*(b + 1.0) - 3.553;
25        printf("c = %g\n",c);
26        /*
27         * Note que nesse caso estamos evitando o cast implícito e
28         * deixamos claro a intenção de converte para um int.
29         */
30        a = (int) b;
31        printf("a = %d\n",a);
32
33        return 0;
34  }
```

Bom, é basicamente isso que há para se aprender sobre casts. Novamente, o autor gostaria de deixar bem claro o programador deve fazer o máximo possível para *evitar* utilizar *casts*.

3.5 Outra forma de fazer atribuições

É bastante comum ao fazer atribuição querer somar ou subtrair uma quantidade do valor atual da variável. Existem alguns comandos especiais de atribuição que tornam essas expressões mais simples. O exemplo abaixo ilustra bem esses comandos.

Exemplo 19 - Cap3_ex19

```
1  /*
2   * Cap3_ex19.c
3   * gcc Cap3_ex19.c -o Cap3_ex19 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9
10         int a = 0;
11
12         a = a + 4*3;
13         printf("a = %d\n",a);
14
15
16         a = 0;
17         /*adiciona 4*3 ao valor atual da variável a*/
18         a += 4*3;
19         printf("a = %d\n",a);
20         /*subtrai 2 do valor da variável a*/
21         a -= 2;
22         printf("a = %d\n",a);
23         /*multiplica por 4 o valor da variável a*/
24         a *= 4;
25         printf("a = %d\n",a);
26         /*divide por 10 o valor da variável a */
27         a /= 10;
28         printf("a = %d\n",a);
29         /* atribui à variável a o valor de a mod 2, isto é o resto
30          * da divisão do valor atual de a por 2
31          */
32         a %= 2;
33         printf("a = %d\n",a);
34         return 0;
35
36 }
```

Naturalmente, essas atribuições funcionam para todos os outros tipos de dados vistos até agora. Bom, o `%=` não funcionam para `float` nem `double`, mas isso é razoável já que o operador `%` só funciona para os tipos inteiros, como o `int`

3.6 Lendo números do teclado

O objetivo dessa seção é servir de uma breve introdução à função `scanf` utilizada obter entrada do usuário. Assim como o `printf`, o scanf também é uma função *vararg*, isto é, recebe um número variável de argumentos.

O exemplo abaixo ilustra o uso do `scanf`

Exemplo 20 - Cap3_ex20

```
1  /*
2   * Cap3_ex20.c
3   * gcc Cap3_ex20.c -o Cap3_ex20 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a;
10         float b;
11         double c;
12
13         /*após digitar cada valor, o usuário deve pressionar enter*/
14
15         printf("Informe o valor de a: ");
16         /*lê do teclado um inteiro e armazena em a*/
17         scanf("%d",&a);
18         /*imprime na tela um \n para pular para a próxima linha*/
19         printf("\n");
20
21         printf("Informe o valor de b: ");
22         /*lê do teclado um float e armazena em b*/
23         scanf("%f",&b);
24         /*imprime na tela um \n para pular para a próxima linha*/
25         printf("\n");
26
27         printf("Informe o valor de c: ");
28         /*lê do teclado um double e armazena em c*/
29         scanf("%lf",&c);
30         /*imprime na tela um \n para pular para a próxima linha*/
31         printf("\n");
32
33         /*imprime na tela os três valores lidos do terminal*/
34         printf("a = %d\n",a);
35         printf("b = %g\n",b);
36         printf("c = %g\n",c);
37
38
39         printf("Informe o valor de a, b e c:\n");
40         /*lê os três valores simultaneamente*/
41         scanf("%d%f%lf",&a,&b,&c);
42
43         /*imprime na tela os três valores lidos do terminal*/
44         printf("a = %d\n",a);
45         printf("b = %g\n",b);
46         printf("c = %g\n",c);
47
48         return 0;
49 }
```

A função `scanf` funciona de forma semelhante à função `printf`, cada tipo de variável possui um código correspondente. Note que com o `printf`, poderia-se

utilizar tanto `%f` quanto `%g` para as variáveis do tipo `float` e `double`. No caso do `scanf`, `%f` não funciona com `double` e deve-se utilizar `lf` ao invés.

O leitor pode estar se perguntando o porquê de utilizar o operador "&" nas chamadas à função `scanf`. A razão é simples: No C, a passagem de parâmetros é feita por *valor*, isto é, ao passar um parâmetro para uma função cria-se uma cópia desse parâmetro. Quando uma variável é modificada *dentro* da função, essa modificação não se reflete *fora* da função, pois tudo que a função possui é uma cópia dessa variável.

Nesse caso, nós estamos querendo simular uma passagem de parâmetros por *referência*, isto é, queremos que o valor das variáveis seja modificado dentro da função `scanf` e queremos que essa modificação se reflita fora da função. Bom, o que o operador "&" faz? Ele retorna *endereço* na memória da variável.

Na verdade, a função `scanf` não recebeu o valor da variável `a`, a função recebeu o *endereço* da variável `a`. Dessa forma, a função modifica os dados que estão gravados nessa porção da memória e assim ao modificar os dados nesse endereço, essa modificação se reflete fora da função.

Talvez, nesse ponto, ainda não esteja muito claro como isso funciona de fato. Esse assunto ficará mais claro quando discutirmos mais sobre funções e ponteiros. O autor não gosta de deixar pontos sem explicação e por isso prefere expor essas questões e discutir com mais detalhes numa etapa posterior.

O autor talvez esteja curioso em relação ao fato do operador "&" retornar o endereço de uma variável. No C é possível imprimir esse endereço na tela. Sendo assim, o leitor achará interessante esse exemplo abaixo.

Exemplo 21 - Cap3_ex21

```
1  /*
2   * Cap3_ex21.c
3   * gcc Cap3_ex21.c -o Cap3_ex21 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a = 2;
10         float b = 3;
11
12         /* %p é o código utilizado para variáveis que são ponteiros.
13          * Ponteiros são variáveis que guardam endereços de outras
14          * variáveis. Então ao utilizar o operador &, estamos mandando
15          * para a função printf o endereço da variável a e da variável b
16          * (no segundo printf)
17          *
18          * Note que os endereços são impressos em hexadecimal.
19          */
20         printf("O endereço de a eh %p\n",&a);
21         printf("O endereço de b eh %p\n",&b);
22
23         return 0;
24 }
```

3.7 Algumas considerações

Como o leitor já deve ter programado em alguma outra linguagem, ele possivelmente deve ter lido algo sobre quais nomes podem ser utilizados para as variáveis e quais não podem. É claro que usando o bom-senso, dificilmente o programador esbarrará em um erro causado pelo *nome* da variável. De todo modo, abaixo segue uma lista de proibições em relação aos nomes de variáveis e funções:

- Não colocarás números no começo dos nomes. Exemplo errado: 156teste. Exemplo certo: x1.
- Sempre começarás um nome com uma letra ou um underline. Exemplo: teste, _teste.
- Não colocarás sinais de pontuação como !, ? e ... nos nomes, sob pena de o programa falhar na compilação.
- Não utilizarás nomes demasiadamente grandes a menos que sejam *estritamente* necessários[5].

Além disso, o C faz a diferenciação entre letras minúsculas e maiúsculas. Isso vale tanto para os nomes das variáveis e funções quanto para as palavras-chave da linguagem. Então ao escrever o tipo `int`, não escreva `Int` ou `INT`.

No entanto, é digno de menção as chamadas "palavras-chave". Essas são as palavras que a linguagem reserva para si e que o programador não deve utilizá-las para declarar variáveis e funções.

3.7.1 Palavras-Chave

Palavras-chave são identificador que são reservados pela linguagem e não podem ser utilizados como nomes de funções e variáveis. O C comparado à outras linguagens como o FORTRAN possui um conjunto pequeno de palavras-chave e elas estão listadas aqui[6]:

Palavras-chave do ANSI C				
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Isso significa que se eu posso utilizar *qualquer* outra palavra para ser nome de variável ou função? Não! A biblioteca padrão define mais alguns nomes e apelidos para alguns tipos de dados como `size_t` e `time_t`. De todo modo, alguns desses nomes são muito específicos e dificilmente um programador precisaria declarar uma variável ou função com esse nome. O que nos leva para a próxima questão.

[5] Nomes grandes às vezes se fazem necessário quando utiliza-se convenções de nomes, ou quando a funcionalidade da função é melhor descrita por um nome grande. Lembre-se sempre, bom senso é essencial.

[6] Essas palavras chaves correspondem ao C89 ou ANSI C

3.7.2 Escolhendo nomes

Um nome mais explicativo não vai tornar o código mais eficiente. Para o computador tanto faz se sua variável se chama "i" ou "pneumoultramicroscopicossilicovulcanoconiótico". A escolha de um nome que seja explicativo é útil primeiramente para o programador e para qualquer outra pessoa que precisa ler o código fonte.

Os nomes das variáveis devem ser refletir o uso que se faz delas. Quando estudarmos estruturas de iteração no próxima capítulo, veremos que as variáveis que controlam a execução dos loops geralmente se chamam i ou j, que é uma convenção informal que remete à matemática. Nomes que são usados sempre para o mesmo propósito facilitam o entendimento, já que ao ver o nome, o leitor talvez já possuia uma idéia do que se trata, possivelmente porque ele já viu esse mesmo nome sendo usado numa situação semelhante.

Por exemplo, ao ver algo como $f(x)$, o leitor provavelmente sabe que se trata de uma função, porém nada impede que se utilize $c(o)$ ao invés de $f(x)$. Naturalmente, $c(o)$ parecerá "estranho" dependendo do contexto e apesar de estar correto, poderá ser fonte potencial de confusão.

Use se possível nomes mnenômicos, ou seja, que facilitem o entendimento e que indiquem de alguma forma o papel da variável no programa.

3.8 Para finalizar o capítulo...

O autor esperar que o leitor tenha achado este capítulo interessante. Pouco a pouco, as características do C e da biblioteca padrão são mostradas ao leitor.

A princípio esse capítulo deveria ser apenas sobre tipos de dados. Mas discutir um pouco sobre outros aspectos do C torna a exposição do conteúdo um pouco mais interessante e atiça a curiosidade do leitor.

Obviamente, não discutiu-se absolutamente tudo que há para discutir sobre os tipos de dados primitivos e suas operações. Mas o autor acredita que discutiu-se o *suficiente* para o leitor conseguir assimilar o conteúdo. Esse tópico com certeza será revisitado acrescentando mais detalhes, mas até lá, espera-se que o leitor já tenha um pouco mais de experiência e proficiência no C.

Ao final, há uma seção de exercícios, o autor recomenda que seja feito *pelo menos* as letras a), b) e c) do exercício 2, que trata de permutar valores de variáveis e o exercício 3.

3.9 Exercícios

Nesse primeiro conjunto de exercícios, o objetivo é verificar se o leitor consegue fazer tarefas básicas com os tipos de dados.

[1]1 - Crie um programa que declare uma variável do tipo `int`, `float`, `double` e do tipo `char`. Atribua valores adequados à essas variáveis e imprima-as utilizando `printf`. Observe se os valores impressos corresponderam aos valores atribuídos.

[5]2 - Uma tarefa comum que o leitor certamente terá que fazer um dia, se já não tiver feito, é trocar valores de variáveis. O que o autor observa é que embora em geral o resultado final seja correto, muitas vezes os programadores utilizam mais código do que o necessário. Observe o seguinte trecho de código:

```
1 int a = 2;
2 int b = 3;
3 int c = 4;
```

a)Suponha que o leitor queira trocar os valores dessas variáveis de forma que `a` seja 4, `b` seja 2 e `c` seja 3, como ele deve proceder a troca seja efetuada com o mínimo de atribuições? Crie um programa que efetue essa tarefa e imprima na tela os valores da variáveis antes e depois da troca.

b)Suponha agora que `a` devesse mudar para 3, `b` para 4 e `c` para 2, o que teria que ser mudado no programa? O número de atribuições necessárias mudou?

c)E se fossem 4 variáveis? O algoritmo para permutar seus valores mudaria sensivelmente?

d)Imagine agora que sejam n variáveis e que o leitor desejasse permutar seus valores de modo que nenhuma variável fique com o seu valor inicial. Qual é o número mínimo e suficiente de atribuições para completar a tarefa? O leitor é capaz de provar matematicamente esse resultado?(*Note que é a mesma tarefa pedida nas letras anteriores, porém para um número qualquer de variáveis*)

[3]3 - É razoavelmente comum precisar transcrever uma expressão aritmética para código, isso acontece quando precisa-se programar uma função, por exemplo. Seja a seguinte função: $f(x) = ((\frac{(x-2)*(x-3)*(3x+2)}{x}) * (\frac{3}{x^2-2x+4}))^2$ Crie um programa que leia do teclado o valor de x e imprima o $f(x)$ correspondente. Assuma que o usuário não colocará um valor inválido para x.

Inicialmente o leitor pode se sentir tentado a fazer algo como:

```
1         double x;
2         double fx;
3
4         printf("Informe o valor de x: ");
5         scanf("%lf",&x);
6         fx = /*expressão muito complicada que representa a função*/
7         printf("\nf(x) eh: %lf\n",fx);
```

Se o leitor fizer assim, a expressão pode ficar demasiadamente grande o que significa mais suscetível a erros. Ao invés de fazer desse modo, utilize variáveis auxiliares que representam pedaços da expressão e ao final "junte" esses pedaços da expressão.

O seguinte exemplo ilustra melhor essa idéia: Vamos supor que $f(x) = \frac{x+2}{x-3}$

Exemplo 22 - Cap3_ex22

```
1  /*
2   * Cap3_ex22.c
3   * gcc Cap3_ex22.c -o Cap3_ex22 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9         double x;
10        double fx;
11        double aux1;
12        double aux2;
13
14        printf("Informe x: ");
```

```
15          scanf("%lf",&x);
16
17          /*é possível fazer assim*/
18          fx = (x+2.0)/(x−3.0);
19          printf("\nf(x) = %lf\n",fx);
20
21          /*para expressões complicadas seria melhor que fosse feito assim*/
22          aux1 = x+2.0;
23          aux2 = x−3.0;
24          fx = aux1/aux2;
25          printf("\nf(x) = %lf\n",fx);
26
27
28          return 0;
29  }
```

Dividindo uma expressão complicada em partes, fica mais fácil programá-la além de deixar o código mais limpo. Tendo esse exemplo em mente, tente programar a função do exercício. Note que fica a critério do leitor utilizar variáveis do tipo `float` ou do tipo `double`. Naturalmente, espera-se que o leitor utiliza os códigos corretos para cada tipo de variável ao utilizar `printf` e `scanf`.

[1]4 - Se o leitor já programava em alguma outra linguagem, responda como os tipos de dados nessa linguagem diferem dos do C. Há diferenças nos tipos de operações possíveis? Essas operações estão embutidas na linguagem ou precisam ser efetuadas através de funções ou procedimentos adicionais? Que tipos de dados essa linguagem possui que o C não tem e vice-versa?

[3]5 - Um erro muito comum é admitir que os tipos `float` e `double` possuem precisão infinita. Como foi discutido na seção 3.2.2, esses tipos de dados possuem uma determinada precisão, ou seja, é possível ter certeza do resultado até um número determinado de casas decimais.

O que muitos não percebem é que isso implica que uma série de conceitos da matemática não se aplicam à operações com números em ponto-flutuante. Um exemplo prático disso é a convergência de séries. Algumas séries que teoricamente deveriam apresentar comportamento divergente(como a série harmônica), aparentemente começam a convergir. O leitor consegue imaginar uma explicação para isso? (*Sugestão: Imagine o que acontece quando não se tem precisão infinita e soma-se um número muito grande com um número muito pequeno*)

Capítulo 4

Seleção e Iteração

Se o leitor já programou em alguma linguagem anteriormente, ele deve estar familiarizado com o conceito de estruturas de seleção e estruturas de iteração.

De todo jeito, estruturas de seleção são aquelas que através de uma determinada condição estabelecem se um trecho de código deve ser executado ou não. Holy moly, isso pareceu complicado? Bom, na verdade isso é apenas o que os "ifs-then-else" nas diversas linguagens fazem.

E o que fazem as estruturas de iteração? Bom... Como o nome indica elas *iteram* um determinado trecho de código n vezes. E o que determina esse n? Pode ser uma condição, por exemplo. Infelizmente, pode acontecer dessa condição nunca ser atingida, é o que chamamos de *loop infinito*. Exemplos de estruturas de iteração são o `for`, `while` e `do..while`.

Nas próximas seções, nós discutiremos como essas estruturas estão representadas no C. Além disso, o autor oferecerá uma pequena introdução à funções.

4.1 Estruturas de Seleção

4.1.1 "if"

Primeiro, um exemplo simples:

Exemplo 1 - Cap4_ex1.c

```
1  /*
2   * Cap4_ex1.c
3   * gcc  Cap4_ex1.c -o Cap4_ex1 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int num;
10
11         printf("Informe um número inteiro: ");
12         /*não se esqueça de colocar o & antes das variáveis ao utilizar
13          *o scanf!*/
14         scanf("%d",&num);
15
16         /*note que há uma expressão booleana entre ()*/
17         if (num>0){/*note que estamos abrindo um novo bloco de código*/
```

```
18              printf("%d eh maior do que 0",num);
19      }/*fecha o bloco de código*/
20      /*se a expressão for falsa, esse trecho será executado*/
21      else{/*abre bloco de código*/
22              printf("%d naum eh maior do que 0",num);
23      }/*fecha bloco de código*/
24
25      return 0;
26 }
```

O programa lê um número do teclado e testa se ele é maior que 0. Caso seja maior do que 0, ele entra no primeiro bloco de código depois do `if`, caso contrário, ele entra no bloco de código correspondente ao `else`.

Note que dentro do `if` poderia estar *qualquer expressão que produza um valor numérico*. Conforme o leitor verá adiante, um valor 0 é falso e diferente de 0 é verdadeiro. Se aquela expressão for verdadeira o bloco de código seguinte será executado. Caso a expressão seja falsa, duas coisas podem acontecer:

- Se houver um `else` correspodente, o bloco de código correspondente ao `else` será executado
- Caso não exista um `else` correspondente, o bloco de código correspodnente ao `if` simplesmente será "pulado".

No exemplo seguinte, temos um `if` que não possui um `else` correspondente.

Exemplo 2 - Cap4_ex2.c

```
1  /*
2   * Cap4_ex2.c
3   * gcc  Cap4_ex2.c -o Cap4_ex2 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int num;
10         int num2;
11
12         printf("Informe um número inteiro: ");
13         scanf("%d",&num);
14
15         printf("Informe outro número inteiro: ");
16         scanf("%d",&num2);
17
18         /*note que para testar igualdade utilizamos '==' e não '=' */
19         if (num == num2){
20                 printf("Os dois números são iguais!\n");
21                 printf("Nunca se esqueça que pode existir mais de ");
22                 printf("um comando dentro do bloco de código!\n");
23         }
24
25         printf("De todo jeito, essa linha de código será executada...");
26
27         return 0;
28 }
```

Um erro comum é esquecer de colocar "()" ao redor da expressão. Então, não se esqueça que após o `if` sempre deve-se colocar o "()"

Antes de prosseguir, vamos fazer um breve pausa para discutir os operadores booleanos no C.

Operadores Booleanos e Relacionais

Operadores booleanos são funções lógicas que pegam um número determinado de operandos e retorna um valor verdadeiro ou falso. No C, não existe um tipo booleano, então considera-se diferente de 0 como verdadeiro e o valor 0 como falso.

E o que seriam essas funções lógicas? Exemplos delas são as funções *E*, *OU* e o *NÃO*. No C, essas funções correspondem aos operadores `&&`, `||` e `!` respectivamente.

No C, qualquer expressão pode ser utilizada como expressão booleana, basta que ela retorne um valor numérico. Conforme dito anteriormente, se a expressão retornar um valor diferente de 0, a expressão será considerada verdadeira, caso contrário, a expressão será falsa.

Os operadores relacionais são os seguintes: *MAIOR*, *MAIOR OU IGUAL*, *IGUAL*, *MENOR*, *MENOR OU IGUAL* e *DIFERENTE*. Esses operadores correspondem aos seguintes operadores no C, `>`, `>=`, `==`, `<`, `<=` e `!=`.

Os operadores booleanos e relacionais podem ser misturados livremente para expressar diferentes tipos de sentenças.

Talvez ainda não esteja muito claro para o leitor a questão dos operadores e de expressões booleanas. O autor espera que os exemplos seguintes ajudem a esclarecer essa situação.

Por enquanto, utilizaremos apenas tipos *inteiros* dentro das expressões a serem avaliadas. Conforme veremos mais adiante, é preciso ter alguns cuidados ao utilizar expressões booleanas com números em ponto flutuante.

Exemplo 3 - Cap4_ex3.c

```
1  /*
2   * Cap4_ex3.c
3   * gcc  Cap4_ex3.c -o Cap4_ex3 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          /*
10          * 1 é diferente de 0, portanto representa o valor verdade
11          */
12         int a = 1;
13         /*
14          * 0 é falso
15          */
16         int b = 0;
17
18         /*
19          * && representa a operação booleana "E"
20          * Essa expressão retorna verdade apenas quando os dois
21          * operadores forem verdadeiros. No caso do C, isso
22          * significa quando os dois operadores representarem
23          * valores diferentes de 0.
24          */
25         if (a && b){
26                 /*
```

```
27                * Esse trecho de código nunca será executado
28                */
29               printf("Nunca imprimirei na tela!\n");
30        }
31
32        /*
33         * || representa a operação booleana "ou"
34         * Essa expressão retorna verdade sempre que um dos
35         * dois operadores for verdadeiro. Ela só é falsa,
36         * quando os dois operadores são falsos.
37         */
38        if (a || b){
39               /*
40                * Como o valor de a e b não mudam, esse trecho
41                * de código sempre será executado.
42                */
43               printf("Sempre imprimirei na tela!\n");
44        }
45        /*
46         * ! é o operador "não". Como b é falso, !b é verdadeiro.
47         * Como a e !b são ambos verdadeiros.
48         * Note que ! é um operador unário, portanto só precisa de
49         * um único operando. Os operadores unários tem maior precedência
50         * sobre os operadores binários(&&,||, por exemplo)
51         */
52        if (a && !b){
53               printf("A expressão é verdadeira, então eu imprimo!\n");;
54        }
55
56        /*
57         * !a produz um valor de verdade falso. b representa um valor
58         * de verdade falso também. Sendo assim !a || b retornará um
59         * valor falso.
60         */
61        if (!a || b){
62               printf("Nunca imprimirei na tela, tambem! \n");
63        }
64
65        /*
66         * Embora, essa estrutura seja estranha e não recomendável, ela
67         * funciona. Note que como a possui um valor de 1, então esse
68         * bloco de código sempre será executado
69         */
70        if (a){
71               printf("Isso eh estranho, naum?\n");
72        }
73
74        /*
75         * De modo análogo, a construção abaixo também funciona. Porém,
76         * o trecho de código abaixo nunca será executado, já que o valor
77         * de b é sempre 0 e portanto falso.
78         */
79        if (b){
80               printf("Isso tambem eh estranho, mas naum serah executado!\n");
81        }
82
83        return 0;
84  }
```

Note que na variável `b` utilizamos o valor 1 para expressar verdade, mas como discutimos anteriormente, poderia ser *qualquer* valor diferente de 0. De

todo jeito, o autor gostaria de frisar que não é uma má idéia utilizar o 1 para servir de valor verdade, muitos programadores fazem dessa forma, é uma espécie de "convenção informal".

Definitivamente, esse não é o uso mais comum dos operadores booleanos. Quase sempre os operadores booleanos estão associados à operadores relacionais. Antes de discutir um exemplo mais elaborado, o autor terá que insultar a inteligência do leitor com outro exemplo trivial, mas dessa vez com o uso de operadores relacionais.

Exemplo 4 - Cap4_ex4.c

```
1  /*
2   * Cap4_ex4.c
3   * gcc Cap4_ex4.c -o Cap4_ex4 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int a = 10;
10         int b = 5;
11
12         /*
13          * Se a for igual a 10 essa expressão gera um valor de verdade
14          * verdadeiro
15          */
16         if (a == 10){
17                 printf("a = %d\n",a);
18                 printf("b = %d\n",b);
19                 a++;
20                 b++;
21         }
22         /*
23          * Se b for diferente de 6 essa expressão gera um valor de verdade
24          * verdadeiro
25          */
26         if (b != 6){
27                 /*esse trecho não será executado!*/
28         }
29         else{
30                 /*note que é possível ter ifs dentro de ifs ou elses,
31                  *são os chamados "ifs aninhados"
32                  */
33                 /*se a for maior do que b esse trecho de código
34                  *será executado
35                  */
36                 if (a > b){
37                         printf("a = %d\n",a);
38                         printf("b = %d\n",b);
39                         a++;
40                         b++;
41                 }
42         }
43
44         a = 0;
45         b = 15;
46         /*testa se a é maior ou igual a b*/
47         if (a >= b){
48                 printf("nunca entrarei aqui!\n");
49         }
50         /*note que é possível ter o else condicionado a um if.
```

```
51          *Isto é para executar o bloco de código associado
52          *ao else é preciso que a condição do if seguinte
53          *também seja verdade.
54          */
55          else if(b <= 15){
56                  printf("Felizmente, eu entrarei aqui!\n");
57          }
58
59          return 0;
60  }
```

Com esses dois exemplos, é possível observar que os operadores booleanos e relacionais são razoavelmente intuitivos. Outra informação interessante é que até as operações aritméticas retornam valores de verdade ou falsidade.

Agora que o leitor já viu operadores booleanos e relacionais, é hora de misturar os dois. O leitor sabe identificar quando um ano é bissexto? Primeiro verifica-se se o ano é divisível por 400. Se for, o ano é bissexto, caso contrário verifica-se se o ano é divisível por 4 e não por 100, caso essa condição seja verdade o ano também é bissexto, caso seja falsa, o ano não é bissexto.

Como ficaria um programa que dado um determinado ano, verifica se ele é bissexto ou não?

Exemplo 5 - Cap4_ex5.c

```
1   /*
2    * Cap4_ex5.c
3    * gcc Cap4_ex5.c -o Cap4_ex5 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int ano;
10
11
12          printf("Informe um ano: ");
13          scanf("%d",&ano);
14          /* Quando um número é múltiplo do outro, o resto da divisão
15           * desse número pelo outro deve ser 0.
16           */
17          if ( ano % 400 == 0){
18                  printf("O ano é bissexto!\n");
19
20          }
21          /* Como a primeira condição para ser bissexto não foi verdadeira,
22           * testa-se a segunda condição. Para o ano ser bissexto, ele
23           * deve ser múltiplo de 4 e não deve ser múltiplo de 100.
24           * Novamente, para testar se um número é múltiplo de outro,
25           * basta dividir o número em questão pelo outro, se o resto
26           * da divisão for o 0, um número é múltiplo do outro.
27           *
28           * Note o uso de () entre cada parte da expressão, note também
29           * que funciona de maneira análoga aos () utilizados nas
30           * expressões aritméticas, forçando o conteúdo dos () a ser
31           * avaliado primeiro
32           */
33          else if( (ano % 4 == 0) && (ano % 100 != 0) ){
34                  printf("O ano é bissexto!\n");
35          }
36          /*note que esse else está "ligado" ao último if*/
```

```
37        else{
38              printf("O ano naum eh bissexto!\n");
39        }
40
41        return 0;
42  }
```

O leitor é convidado a ler com calma e executar várias vezes o programa para entender as expressões dentro do `if`. Note o uso de um `if` aninhado e que o segundo `else` só será executado a condição presente no o *último* `if` for falsa.

Se o leitor for sagaz, ele perceberá que esse programa contém um pequeno problema. Vamos supor que o usuário informe um número *menor* que 400. Para todo número menor que 400 a expressão `ano % 400 == 0` sempre será verdadeira, porque o resto sempre será igual a 0. Então é preciso fazer checagens adicionais para ver se o ano informado é menor do que 400, correto? Errado. Se o leitor for *ainda mais* sagaz, ele saberá que o calendário gregoriano foi instituído em 1582 então, tecnicamente *não* existiram anos bissextos antes dessa data, afinal anos bissextos são característicos do calendário gregoriano.

Implementando outras funções booleanas

O C fornece 3 operadores booleanos: `&& || e !`. Embora pareça pouco, esse é um conjunto completo de operadores, no sentido que a partir deles é possível derivar todos os outros.

Existem 16 tipos de operadores booleanos binários, isto é que recebem 2 operandos e retornam um valor de verdade (verdadeiro ou falso). As mais conhecidas são provavelmente o `&&` e `||`(E e OU). Alguns dos 16 operadores booleanos binários não são muito utilizado, porém às vezes eles ajudam a tornar as expressões um pouco mais limpas.

Exemplos de outros operadores binários comuns são: $NAND$(**N**ot **AND**), XOR(e**X**clusive **OR**). Na seção de exercícios, o autor propõe que o leitor implemente algumas funções boolenas. De todo jeito, segue abaixo um exemplo que contém a implementação do XOR.

Exemplo 6 - Cap4_ex6.c

```
1   /*
2    * Cap4_ex6.c
3    * gcc Cap4_ex6.c -o Cap4_ex6 -Wall -g
4    *
5    */
6
7   #include <stdio.h>
8
9   int main(void){                                                    main
10        int a;
11        int b;
12
13
14        printf("Digite 1(verdadeiro) ou 0 (falso): ");
15        scanf("%d",&a);
16
17        printf("\nDigite 1(verdadeiro) ou 0 (falso): ");
18        scanf("%d",&b);
19
20        /*
```

```
21          * a operação XOR retorna verdadeiro se apenas 1 dos
22          * operandos for verdadeiro.
23          */
24         if ( (a || b) && !(a && b) ){
25                 printf("O resultado eh verdadeiro!\n");
26         }
27         else{
28                 printf("O resultado eh falso!\n");
29         }
30
31         return 0;
32 }
```

4.1.2 switch

O `switch` é um comando de seleção múltipla, ao invés de apenas condicionar a execução de um bloco de código como o `if` ele condição a execução de vários blocos de códigos. Observe o exemplo:

Exemplo 7 - Cap4_ex7.c

```
1  /*
2   * Cap4_ex7.c
3   * gcc Cap4_ex7.c -o Cap4_ex7 -Wall -g
4   *
5   */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10         int opcao;
11
12         printf("Informe um numero no intervalo de 1 a 3: ");
13         scanf("%d",&opcao);
14
15         /*
16          * Verifica se o usuário informou um número dentro do
17          * intervalo de 1 a 3
18          */
19         if (opcao < 1 || opcao > 3){
20                 printf("\nVoce informou um numero fora do intervalo! =\\ \n");
21         }
22         else{
23                 /*
24                  * Dentro dos () coloca-se uma expressão.
25                  * O valor dessa expressão determinará qual case será
26                  * executado.
27                  */
28                 switch(opcao) {/*começo do bloco de código do switch*/
29
30                 /*
31                  * se a expressão dentro do switch assumir um valor de 1,
32                  * essa sequência de comandos será executada.
33                  */
34                 case 1:
35                         /*após cada case, pode seguir uma sequência de comandos*/
36                         printf("Este eh o primeiro case!\n");
37                         printf("Este ainda eh o primeiro case!\n");
38                         break;
39                 /*
```

```
40                * se a expressão dentro do switch assumir um valor de 2,
41                * essa sequência de comandos será executada.
42                */
43               case 2:
44                     printf("Este eh o segundo case!\n");
45                     break;
46               case 3:
47                     printf("Este eh o terceiro case!\n");
48                     break;
49               /*
50                * Caso a expressão dentro do switch não corresponda a nenhum
51                * dos valores dos cases, o programador pode optar por incluir
52                * um comando default. Os comandos correspondentes ao default
53                * serão executados caso a expressão do switch não corresponda
54                * a nenhum case.
55                *
56                */
57               default:
58                     printf("Dentro de condições normais,\n"),
59                     printf("eu nunca entraria aqui!\n");
60               }/*final do bloco de código do switch*/
61          }
62
63          return 0;
64     }
```

O autor imagina que o leitor deve ter algumas dúvidas relacionadas aos comandos novos presentes nesse exemplo.

- As expressões dentro do `switch` devem ser apenas com número inteiros ou caracteres, ou seja, nada de utiliza `float` ou `double`.

- `break`: O comando `break` funciona dentro de blocos de código. O leitor deve se lembrar que blocos de códigos estão envoltos por {}. O que `break` faz é forçar a saída do bloco de código. Ao sair do bloco de código a sentença seguinte é executada. Note que antes de cada `case` há um `break`. O leitor é convidado a testar e retirar os `break`s e verificar o que acontece. Basicamente, *sem* o `break` assim que a execução entre num determinado `case` além de executar os comandos associados àquele `case`, executa os comandos dos `case`s seguintes e possivelmente até do comando `default`(se estiver presente). Naturalmente, quando encontrar um `break`, será executada a primeira sentença fora do bloco de código.

- `default`: Conforme dito nos comentários do exemplo, não é necessário incluir o `default`, *porém* não é uma má idéia colocá-lo ainda que o programa nunca entre no `default`. Nesse caso, seria interessante deixar um comentário explicando o motivo de nunca entrar no `default`.

- O leitor deve ter notado que é possível reescrever o o programa utilizando `if-else-if`. Nesse caso, o autor acredita que utilizar o `switch` torna o código mais legível. De todo jeito o próximo exemplo é exatamente o mesmo programa porém escrito com `if-else-if`

Exemplo 8 - Cap4_ex8.c

```
1  /*
```

```
2  * Cap4_ex8.c
3  * gcc Cap4_ex8.c -o Cap4_ex8 -Wall -g
4  *
5  */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10         int opcao;
11
12         printf("Informe um numero no intervalo de 1 a 3: ");
13         scanf("%d",&opcao);
14
15         /*
16          * Verifica se o usuário informou um número dentro do
17          * intervalo de 1 a 3
18          */
19         if (opcao < 1 || opcao > 3){
20                 printf("\nVoce informou um numero fora do intervalo! =\\ \n");
21         }
22         /*note o uso de vários else if em sequência*/
23         else if(opcao == 1){
24                 printf("Este eh o primeiro else!\n");
25                 printf("Este ainda eh o primeiro else!\n");
26         }
27         else if(opcao == 2){
28                 printf("Este eh o segundo else!\n");
29         }
30         else if(opcao == 3){
31                 printf("Este eh o terceiro else!\n");
32         }
33         /*
34          * esse último else equivale semanticamente ao default do exemplo
35          * anterior.
36          */
37         else{
38                 printf("Dentro de condições normais,\n"),
39                 printf("eu nunca entraria aqui!\n");
40         }
41
42         return 0;
43 }
```

O próximo exemplo ilustra o uso de um caracter como expressão dentro do `switch`, além de ser mais uma referência infame à Monty Python. Exemplo

```
1  /*
2  * Cap4_ex9.c
3  * gcc Cap4_ex9.c -o Cap4_ex9 -Wall -g
4  *
5  */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10         char opcao;
11
12         printf("What about a Monty Python quote?(s/n) ");
13         scanf("%c",&opcao);
14
15         switch(opcao){
```

```
16              case 's':
17              printf("\nWe are the knight who say. . . Ni!");
18              break;
19              case 'n':
20              printf("\nBye!");
21              break;
22              default:
23              printf("\n'%c' is not an option! Bye!\n",opcao);
24      }
25      return 0;
26 }
```

4.1.3 O operador ?

O operador `?` é raro de ser encontrado em programas. A razão disso é que ele basicamente efetua o trabalho de um `if`. É verdade que pode tornar o código fonte mais simples, mas seu uso desnecessário e exagerado denota virtuosismo pedante por parte do programador.

Seu uso é da seguinte forma:

expressão `?` *expressão2* `:` *expressão3*.

Basicamente o que acontece é o seguinte: Se *expressão* for verdadeira, o valor de toda a expressão passa a ser *expressão2*. Caso contrário, o valor passa a ser a *expressão3*. Por enquanto, parece meio confuso, porém um exemplo ajudará a esclarecer melhor os fatos.

Exemplo 10 - Cap4_ex10.c

```
1  /*
2   * Cap4_ex10.c
3   * gcc Cap4_ex10.c -o Cap4_ex10 -Wall -g
4   *
5   */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10      int num;
11      int aux;
12
13      printf("Informe um número inteiro: ");
14      scanf("%d",&num);
15
16      /*
17       * Se num for igual a 0, atribui 15 a aux.
18       * Se num for diferente de 0, atribui 20 a aux.
19       */
20      aux = num == 0 ? 15 : 20;
21      printf("\naux = %d\n",aux);
22
23      /*
24       * É equivalente a fazer o seguinte if.
25       * Naturalmente, utilizando o operador ? o código fica
26       * mais sucinto. Porém, ao ler o código fonte nem
27       * sempre fica claro o que o ? faz.
28       */
29      if (num == 0){
30              aux = 15;
31      }
```

```
32          else{
33                  aux = 20;
34          }
35          printf("\naux = %d\n",aux);
36
37          return 0;
38  }
```

O exemplo acima mostra um uso do `?` interessante, afinal utiliza-se menos linhas de código do que ao utilizar o `if`. Porém, é fácil complicar ao utilizar o `?` como mostra o exemplo abaixo:

Exemplo 11 - Cap4_ex11.c

```
1   /*
2    * Cap4_ex11.c
3    * gcc Cap4_ex11.c -o Cap4_ex11 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int a;
10
11          printf("Informe um número inteiro: ");
12          scanf("%d",&a);
13          /*
14           * Sem rodar o programa, o leitor é capaz de dizer o que
15           * será impresso na tela?
16           */
17          printf(a ? "%d!":"a=0",a);
18
19          /*
20           * O leitor sabe dizer em quais condições o programa
21           * imprimirá na tela "Hello!"?
22           */
23          if (a>0 ? a>= 5 : a<-15){
24                  printf("Hello!\n");
25          }
26          return 0;
27  }
```

Naturalmente, o leitor deve evitar se possível utilizar esses tipos de construções que tornam o código mais complicado de ler. Embora, eles certamente denotem a proficiência do programador na linguagem em questão, eles mostram que o programador tem pouca ou nenhuma preocupação com as pobres almas que terão que revisar e ler o código fonte dele. Programação não é só conhecer a linguagem em questão, é saber escrever código simples e efeciente.

É claro que existem situações que a complexidade é inerente à tarefa. Ainda sim, deve-se tentar obter a solução mais simples possível. Trata-se de sempre se possível, evitar complexidade *desnecessária*. O acrônimo KISS(**K**eep **I**t **S**imple, **S**tupid!) reflete bem essa a idéia[1].

[1] A distribuição de Linux, Slackware, é um exemplo famoso do que pode ser considerado um projeto que segue o princípio do KISS. É uma das distribuições favoritas daqueles que gostam de customizar ao máximo o sistema e ter liberdade quase total de configuração.

4.2 Estruturas de Iteração

Ah! Iteração... Agora as coisas definitivamente vão ficar mais interessantes. As estruturas de iteração no C permitem um grau muito elevado de liberdade. Possivelmente, `for` e o `while` são as duas estruturas de iteração mais utilizadas. Há porém o `do..while` que ocasionalmente precisa ser utilizado.

Não há muito segredo no uso dessas estruturas e nas seções seguintes o autor descreverá cada uma dessas e estruturas e fornecerá exemplos.

4.2.1 while

O `while` é utilizado da seguinte forma:

while (condição_de_execução){
... *comandos*
}

Note que depois da *condição_de_execução* entre (), deve-se abrir um novo bloco de código.O termo "condição de execução" foi escolhido porque a execução do bloco de código está condicionada à essa expressão. Enquanto a condição de execução for *verdadeira*, o bloco de código será executado. Quando a condição de execução for *falsa* o bloco de código não será mais executado. [2] Observe o exemplo: Exemplo 12 - Cap4_ex12.c

```
1  /*
2   * Cap4_ex12.c
3   * gcc Cap4_ex12.c -o Cap4_ex12 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          /* note que a variável que é utilizada com condição
10          * de execução está sendo inicializada!
11          */
12         int i = 0;
13
14         /* executa o bloco de código enquanto o valor de i for
15          * menor do que 50
16          */
17         while (i < 50){
18                 printf("Dentro do loop: %d\n",i);
19                 /*incrementa i*/
20                 i++;
21         }
22         /*note que i sai do loop com o valor igual a 50*/
23         printf("Fora do loop: %d\n",i);
24
25         /*o mesmo loop com um pequena modificação na condição de execução*/
26         i = 0;
27         while (i <= 50){
28                 printf("Dentro do loop: %d\n",i);
29                 /*incrementa i*/
30                 i++;
```

[2] Na verdade se o bloco de código consiste de apenas uma único comando, é possível omitir os { }. Isso também vale para as outras estruturas de iteração e para as estruturas de seleção. De todo modo, o autor explicará numa seção posterior o porquê de mesmo se for apenas um único comando, deve-se utilizar { }.

```
31          }
32          /*note que i sai do loop com o valor igual a 51*/
33          printf("Fora do loop: %d\n",i);
34
35          i = 51;
36          /*note que antes de entrar no loop testa-se a condição*/
37          while(i <= 50){
38                  printf("não entrarei aqui!\n");
39                  i++;
40          }
41          return 0;
42  }
```

Observe que a condição de execução pode ser qual expressão booleana ou relacional ou uma combinação das duas. Conforme veremos adiante, a condição de execução ser falsa não é a *única* forma de sair de um loop ou de um bloco de código. Existem outras formas de sair de um bloco de código que não seja o critério de parada, isto é a condição de execução ser falsa, da estrutura em questão.

De todo modo, se o programador optar por utilizar uma condição de execução que envolvam variáveis, ele deve tomar os cuidados necessários para que todas as variáveis sejam modificadas corretamente para que o loop chegue de fato em um fim.

Não foi à toa que o autor escreveu o exemplo acima. É muito comum ocorreremm erros por 1. Isto é, ao escrever um loop (independente do tipo: `while`, `for` ou `do..while`), às vezes acontecem erros em que o resultado obtido diverge do esperado por 1 unidade. O programador deve ficar atento que *assim que a condição de execução é falsa, o loop não é executado novamente*. Além disso, pode acontecer da condição de execução ser falsa antes de acontecer a primeira repetição. Por isso que é comum dizer que com o `while` primeiro, testa-se a condição de execução e *depois* executa-se o loop.

Um exemplo mais interessante

O leitor talvez esteja familiarizado com séries de Taylor. Caso não esteja, aqui vai uma breve explicação:

Séries de Taylor fornecem aproximações através de polinômios para uma determinada função. Geralmente, esses polinômios são expressos através de termos de uma série e ocasionalmente essa série acaba por convergir para a função em questão.

Como as séries de Taylor são infinitas, geralmente utiliza-se somas parciais da série de Taylor. Ou seja, utiliza-se os primeiros n-termos para obter um aproximação da função. Naturalmente, a medida que n aumenta, obtemos aproximações cada vez melhores.

Quando a série de Taylor de uma função é expandida em torno de $x_0 = 0$, é comum se referir a ela como *série de Maclaurin*. Para uma função de uma variável, a série de Taylor em torno de x_0 é:

$$f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \dots$$

Sendo assim, define-se o *n-ésimo* polinômio de Taylor de f em torno de x_0 como:

$$P_n = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

Através dessa fórmula chega-se nas tão conhecidas expressões para aproximar as funções trigonométricas através de polinômios. O problema que o autor propõe é o seguinte: construir um programa que receba do usuário o valor de um ângulo em radianos e que imprima na tela o seno desse ângulo.

A o *n-ésimo* polinômio de Maclaurin (já que vamo expandir em torno de $x_0 = 0$) para $f(x) = sen(x)$ é:

$$P_n = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + \frac{(-1^n)}{(2n+1)!}x^{2n+1}$$

Bom, note que é necessário o cálculo do fatorial em cada um dos termos. Vamos inicialmente fazer um programa que calcule o fatorial de um número. Para efeito de demonstração o cálculo do fatorial será feito duas vezes, primeiro com um variável do tipo `int` e depois com uma variável do tipo `double`.

Exemplo 13 - Cap4_ex13.c

```
1  /*
2   * Cap4_ex13.c
3   * gcc Cap4_ex13.c -o Cap4_ex13 -Wall -g -lm
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int fat = 1;
10         double fat2 = 1.0;
11         int i = 1;
12         int num;
13
14         printf("Informe n inteiro: ");
15         scanf("%d",&num);
16
17
18         while (i <= num){
19                 /*lembre-se que isso é igual a fat = fat * i;*/
20                 fat *= i;
21                 /*incrementa i*/
22                 i++;
23         }
24
25         i = 1;
26         while (i <= num){
27                 /*lembre-se que i é int e o cast é necessário*/
28                 fat2 *= (double) i;
29                 /*incrementa i*/
30                 i++;
31         }
32
33         printf("\nFatorial de %d = %d\n",num,fat);
34         printf("\nFatorial de %d = %g\n",num,fat2);
35
36         return 0;
37 }
```

O leitor atento sabe que o fatorial é uma operação restrita ao números naturais, sendo assim, teoricamente poderia-se utilizar o tipo `int` para o cálculo do

fatorial, correto?[3] Bom... não exatamente. Como o fatorial cresce rapidamente, com um `int` de 32-bits só é possível obter o resultado correto até 13!. 14! possui magnitude maior do que o armazenável em um `int` de 32-bits. Variáveis do tipo `double` podem representar números com magnitudes mais elevadas do que as do `int`.

O leitor pode efetuar teste informando ao programa o valor 14 e observar que o fatorial calculado com a variável do tipo `int` e a variável do tipo `double` divergem[4]. Nesse caso, a escolha entre `int` e `double` depende do tipo de aplicação que o programador esteja desenvolvendo. Caso a aplicação nunca precise calcular o fatorial de um número que seja maior do que um `int` pode suportar, então um utilizar um `int` é uma boa idéia.

Bom, com isso, podemos apresentar o programa que calcula o seno:

Exemplo 14 - Cap4_ex14.c

```
1  /*
2   * Cap4_ex14.c
3   * gcc Cap4_ex14.c -lm -o Cap4_ex14 -Wall -g
4   *
5   */
6  #include <stdio.h>
7  /*
8   * note que estamos incluindo um arquivo cabeçalho a mais. Este
9   * arquivo corresponde à biblioteca math que está presente no
10  * padrão ANSI. Note que o comando de compilação mudou, foi
11  * acrescentado um -lm.
12  */
13 #include <math.h>
14
15 int main(void){                                                    main
16         double rad;
17         double seno = 0.0;
18         /*variável auxiliar*/
19         double tmp = 0.0;
20         double fat = 0;
21         /*inicializa a variável i e a variável j*/
22         int i = 0, j = 0;
23
24
25         printf("Infome o ângulo desejado em radianos: ");
26         scanf("%lf",&rad);
27
28         /*
29          * Observe a condição de execução: Enquanto i for menor do que
30          * 10, o bloco de código será executado.
31          */
32         while (i < 10){
33                 /*
34                  * pow é uma função que recebe dois argumentos, o primeiro
35                  * é a base e o segundo é o expoente. Os dois argumentos
36                  * são do tipo double , então é interessante efetuar um cast.
37                  * A função pow retorna o resultado de base^expoente.
38                  */
39                 tmp = pow(-1.0,(double) i) * pow(rad,(double) 2*i + 1);
40
41                 /* Esse trecho serve para calcular o fatorial de
```

[3] Existe uma generalização do fatorial para números reais e complexos que é chamada a função Gamma, e é representada por $\Gamma(x)$

[4] Considerando um inteiro de 32-bits

```
42                 * 2*i + 1
43                 */
44                /*reseta o valor de j e de fat*/
45                j = 1;
46                fat = 1.0;
47                /*este é o loop que calcula de fato o fatorial*/
48                while (j <= (2*i+1)){
49                        /* lembre-se que j é int portanto o cast se faz
50                         * necessário. Lembre-se, explícito é melhor que
51                         * implícito!
52                         */
53                        fat     *= (double) j;
54                        /*incrementa j*/
55                        j++;
56                }
57
58                seno += tmp/fat;
59
60                /*incrementa i*/
61                i++;
62        }
63        printf("seno(%g) = %g\n",rad,seno);
64
65        return 0;
66 }
```

Esse exemplo possui algumas peculiaridades:

- Note que o programa utiliza a biblioteca math. A biblioteca math deve ser lincada à parte, pois geralmente existe a opção de efetuar a lincagem estaticamente(em oposição à lincagem dinâmica), o que teoricamente melhoraria a performance ao custo de aumentar o tamanho do executável. Essa é a única biblioteca do padrão ANSI que precisa de um comando especial de compilação no gcc(-lm).

  Como é possível deduzir, essa biblioteca possui funções matemáticas, como seno, cosseno, logaritmo e outras. Aos poucos serão introduzidas outras funções da biblioteca math. Além disso, num capítulo posterior haverá um tópico sobre lincagem dinâmica e estática.

- Como a função `pow` recebe dois parâmetros do tipo `double`, o cast foi necessário. Mas é preciso que fique bem claro que ainda sem o cast explícito o programa funcionaria corretamente. Porém, o autor é da opinião de que *explícito é melhor do que implícito* e como já foi discutido anteriormente, é sempre bom tornar os casts explícitos.

Naturalmente, essa não é a *melhor* forma de se fazer esse programa. Quando discutirmos funções, o leitor perceberá uma outra forma de fazer esse programa, utilizando uma função para calcular o fatorial, ao invés de embutir o cálculo do fatorial no loop. No entanto, uma forma mais inteligente de fazer esse programa se daria da seguinte forma:

- Note que na iteração na qual é calculada o $n!$, o valor de $(n-2)!$ terá sido calculado na iteração anterior.

- Dessa forma, ao invés de atribuir 1 à variável `fat` à cada iteração, poderíamos aproveitar os valores das iterações anteriores.

- De forma análoga, podemos poupar o cálculo de x^{2n+1} e de $(-1)^n$, evitando assim chamar a função `pow` da biblioteca padrão.

Exemplo 15 - Cap4_ex14_2.c

```
1  /*
2   * Cap4_ex14.c
3   * gcc Cap4_ex14.c -lm -o Cap4_ex14 -Wall -g
4   *
5   */
6  #include <stdio.h>
7  /*
8   * note que estamos incluindo um arquivo cabeçalho a mais. Este
9   * arquivo corresponde à biblioteca math que está presente no
10  * padrão ANSI. Note que o comando de compilação mudou, foi
11  * acrescentado um -lm.
12  */
13 #include <math.h>
14
15 int main(void){                                                    main
16         double rad;
17         double seno = 0.0;
18         /*variáveis auxiliares*/
19         double tmp;
20         double tmp2 = 1.0;
21         double fat;
22         int i = 0;
23
24
25
26         printf("Infome o ângulo desejado em radianos: ");
27         scanf("%lf",&rad);
28
29         /*
30          * Note que não mais resetaremos o valor de fat após cada iteração!
31          * fat é inicializado com 1 e tmp com rad.
32          */
33         fat = 1.0;
34         tmp = rad;
35         while (i < 10){
36                 seno += (tmp/fat)*tmp2;
37
38                 /*
39                  * Quando i = 0, tmp ficará com o valor correspondente à
40                  * x^3, pronto para a iteração seguinte. Quando i = 1,
41                  * tmp ficará com o valor correspondente à x^5, pronto para a
42                  * iteração seguinte. E assim por diante.
43                  */
44                 tmp *= rad*rad;
45                 /*A cada iteração tmp2 troca de sinal*/
46                 tmp2 *= -1.0;
47                 /*
48                  * Na primeira iteração, i = 0, então fat será multiplicado por
49                  * 2 e por 3, dessa forma teremos 3!, o número necessário para
50                  * a próxima iteração.
51                  * Quando i = 1, fat será multiplicado por 4 e por 5, teremos
52                  * 5!, que é necessário para a iteração seguinte.
53                  * E assim por diante.
54                  * Observe que está ocorrendo dois casts implícitos!
55                  */
56                 fat *= ( 2*(i+1) );
57                 fat *= ( 2*(i+1) + 1);
```

```
58
59              i++;
60          }
61          printf("seno(%g) = %g\n",rad,seno);
62
63          return 0;
64  }
65
```

O autor espera que o leitor tenha achado esse exemplo construtivo, e então, sem nos desviar mais, vamos prosseguir com mais iteração!

4.2.2 for

for (inicialização ; condição de execução ; incremento ou decremento){
... comandos
}
O for funciona da seguinte forma:

1. Executa-se os comandos da inicialização
2. Verifica-se a condição de execução. Se for verdadeira, o bloco de código é executado. Se for falsa, o bloco de código não é executado.
3. Se o bloco de código tiver sido executado, ao final de todos os comandos presentes no bloco de código, executa-se os comandos de incremento ou decremento.
4. Volta para o passo 2.

Apesar de levemente mais complicado do que o `while`, o for é uma estrutura bastante interessante. Note que o for é composto por três elementos: *inicialização; condição de execução; incremente ou decremento*, porém os três são *opcionais*, isto é, eles não precisam estar presentes no *for*! Por enquanto, utilizaremos o `for` "completo". Observe o exemplo:

Exemplo 16 - Cap4_ex15.c

```
1   /*
2    * Cap4_ex15.c
3    * gcc  Cap4_ex15.c -o  Cap4_ex15 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int i;
10          /*
11           * Primeiro indica-se a inicialização, depois a condição de
12           * execução (i < 10) e depois o incremento. Note que tudo é separado
13           * por ;
14           */
15          for (i = 0; i < 10; i++){
16                  printf("Dentro do for: %d\n",i);
17          }
18          printf("Fora do for: %d\n",i);
19
20          return 0;
```

```
21 }
```

Observe que o `while` e o `for` são estruturas equivalentes, não há nada que um possa fazer que o outro não possa. Anteriormente, fizemos um programa que calcula o fatorial utilizando o `while`. Naturalmente, é possível fazer o mesmo programa utilizando o `for`:

Exemplo 17 - Cap4_ex16.c

```
1  /*
2   * Cap4_ex16.c
3   * gcc  Cap4_ex16.c -o Cap4_ex16 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int i = 0;
10         int num;
11         int fat = 0;
12
13         printf("Informe um número inteiro: ");
14         scanf("%d",&num);
15
16         /*
17          * observe que existem dois comandos de inicialização:
18          * i = 1 e fat = 1. Em cada elemento for, pode-se espeficar
19          * mais de um comando, basta separá-los por vírgula.
20          */
21         for (i = 1, fat = 1; i <= num; i++){
22                 fat *= i;
23         }
24         printf("%d! = %d\n",num,fat);
25
26         return 0;
27 }
```

Como o exemplo mostra é possível ter mais de um comando de inicialização e conforme veresmos no próximo exemplo é possível ter mais de um comando de incremento também. Apenas a condição de parada que deve ser única, ou não deve existir. O leitor deve ter paciência, já que como não discutimos o comando `break` mais a fundo, não será mostrado um `for` sem condição de execução.

Exemplo 18 - Cap4_ex17.c

```
1  /*
2   * Cap4_ex17.c
3   * gcc  Cap4_ex17.c -o Cap4_ex17 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int i,j;
10
11         /*
12          * inicialização: i = 1 e j = 5;
13          * condição de execução: enquanto i < j
14          * incremento: i+= 2 e j++
15          */
```

```
16      for (i = 1, j = 5; i < j; i+=2, j++){
17              printf("Dentro do loop, i = %d e j = %d\n",i,j);
18      }
19      printf("Fora do loop, i= %d e j = %d\n\n",i,j);
20
21      i = 15;
22      j = 10;
23      /*
24       * A menos que haja uma razão muito boa para não colocar
25       * a inicialização no for, é preferível utilizar o
26       * for "completo".
27       *
28       * De todo jeito, se for necessário colocar a inicialização
29       * à parte, pode-se simplesmente colocar ; e escrever
30       * a condição de execução, se houver.
31       *
32       * A condição de execução deve ser verdadeira para o bloco
33       * de código ser executado, isto é, apenas se i > 10 E j < 18
34       * o bloco de código será executado.
35       */
36      for ( ; i > 10 && j < 18; i--, j++){
37              printf("Dentro do loop, i = %d e j = %d\n",i,j);
38      }
39      printf("Fora do loop, i= %d e j = %d\n\n",i,j);
40
41      i = 1;
42      j = 1;
43
44      /*
45       * esse for possui apenas a condição de execução. Nesse caso
46       * ele funciona exatamente como um while.
47       */
48      for ( ; i < 10 ; ){
49              printf("Dentro do loop, i = %d e j = %d\n",i,j);
50              i++;
51              j++;
52      }
53      printf("Fora do loop, i= %d e j = %d\n",i,j);
54
55      return 0;
56  }
```

De modo geral, é uma boa idéia utilizar o for "completo", isto é com inicialização, condição de execução e incremento. Novamente, isso é uma questão de bom-senso, pode ser que a inicialização se dê com várias variáveis e seja demasiadamente complexa para colocar dentro do `for`. O programador deve optar por utilizar a estrutura que fique mais eficiente e fácil de ler.

4.2.3 do..while

O `do..while` funciona exatamente da mesma forma do `while`. A diferença é que o teste que condiciona a execução do bloco de código é feito *ao final*. Isso significa que o bloco de código será executado *pelo menos* uma vez.

Um uso comum do `do..while` é na construção de menus para o usuário. Se o usuário informar uma opção inválida, escreve-se novamente o menu na tela com as opções para usuário escolher uma outra opção. Observe o exemplo:

Exemplo 19 - Cap4_ex18.c

```
1  /*
2   * Cap4_ex18.c
3   * gcc Cap4_ex18.c -o Cap4_ex18 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          int opcao;
10
11         do{
12                 /*imprime as opções na tela*/
13                 printf("Opcao 1 \n");
14                 printf("Opcao 2 \n");
15                 printf("Opcao 3 \n");
16                 printf("Informe uma opção pelo número: ");
17                 scanf("%d",&opcao);
18
19                 /*executa uma ação adequada para cada opcao*/
20                 switch(opcao){
21                         case 1:
22                         printf("\nVoce escolheu a opcao 1!\n");
23                         break;
24                         case 2:
25                         printf("\nVoce escolheu a opcao 2!\n");
26                         break;
27                         case 3:
28                         printf("\nVoce escolheu a opcao 3!\n");
29                         default:
30                         printf("\nVoce escolheu uma opcao invalida!\n");
31                 }
32         /*
33          * repete o bloco de código enquanto o usuário
34          * informar uma opcao inválida, isto é maior do que
35          * 3 ou menor do que 1
36          */
37         }while(opcao > 3 || opcao < 1);
38
39         return 0;
40 }
41
```

Primeiro executa-se o bloco de código, depois testa-se a condição dentro do `while`. Enquanto a condição for verdadeira, executa-se o bloco de código.

4.3 Variáveis em ponto flutuante em estruturas condicionais

Até esse ponto, nenhum dos exemplos apresentados acima no estudo das estruturas condicionais e de iteração apresentava variáveis em ponto-flutuante nas expressões booleanas e relacionais.

Uma questão que nem sempre fica clara ao utilizar números em ponto-flutuante é que os números *nem sempre* possuem uma representação exata. Lembre-se que no C, para utilizar um número inteiro, apenas guarda-se nas variáveis sua representação em binário. Mas, isso não é verdade para os números em ponto-flutuante.

Antes de discutir o mérito das variáveis em ponto-flutuante em estruturas condicionais, é preciso entender melhor como os números em ponto flutuante são representados.

4.3.1 Representação dos números em ponto-flutuante

Um número em ponto-flutuante é representado da seguinte forma[5]:

$N = s \times 2^{Exp-Bias} \times m$

Onde: s é o sinal (positivo ou negativo).

Exp é o expoente.

m é a mantissa, que é um valor real tal que $1 \leq m < 2$.

$Bias$ é um offset cujo valor depende se estamos trabalhando com precisão simples ou dupla..

Esses quatro valores s, Exp, $Bias$ m são os valores que ficam guardados nas variáveis do tipo `float` e `double`. Naturalmente, esses valores são armazenados em *binário*.

Com um `float` de 32-bits, a distribuição dos bits para cada componente é a seguinte:

- 1-bit para o sinal
- 8-bits para o expoente
- 23-bits para a mantissa.
- O $Bias$ possui valor de 127

Para variáveis do tipo `double` de 64-bits, a distribuição é a seguinte:

- 1-bit para o sinal
- 11-bits para o expoente
- 52-bits para a mantissa.
- O $Bias$ possui valor de 1023

Essa é a representação utilizada para armazenar um número em ponto-flutuante em variáveis do tipo `float` e `double`. Naturalmente, nem todo número possui uma representação *exata* ao utilizar essa forma. Sendo assim a comparação de variáveis em ponto-flutuante deve levar em conta essa *imprecisão*, ainda que ocasionalmente seja pequena, para evitar resultados inesperados.

4.3.2 Comparando números em ponto-flutuante

Conforme dito anteriormente, nem todos os números possuem representação exata. Observe o seguinte exemplo:

Exemplo 20 - Cap4_ex19.c

```
1 /*
2  * Cap4_ex19.c
3  * gcc Cap4_ex19.c -o Cap4_ex19 -Wall -g
4  *
```

[5] Essa explicação é baseada na página `http://en.wikipedia.org/wiki/IEEE_754`

```
5  */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9          double a = 11.0;
10         double b;
11
12         b = 1.1 * 10.0;
13
14         /*
15          * Provavelmente essa comparação retornará um valor
16          * falso.
17          */
18         if (a == b){
19                 printf("Provavelmente não entrarei aqui\n");
20         }
21         else{
22                 printf("Inesperadamente eu entrei aqui!\n");
23         }
24
25         return 0;
26 }
```

No computador do autor, a expressão dentro do `if` retornou ***falso*** e provavelmente no computador do leitor o resultado será o mesmo. Na verdade por conta do fato de alguns números não serem representáveis de forma exata ou pelo fato das operações aritméticas acarretarem uma pequena imprecisão, os dois números são considerados *diferentes*.

O leitor deve refletir por um momento: "O que é ser *igual*?". Para efeitos práticos, o programador quer que a expressão $1 == 0.999999$ retorne falso? Dependendo da precisão desejada para o programa, provavelmente a resposta será *não*.

Uma forma de contornar esse problema, seria admitir um determinado ϵ que represente o erro máximo para que dois números sejam considerados iguais. Por exemplo, se $\epsilon = 0.00001$, dois números a e b serão considerados diferentes apenas se $|a - b| > 0.00001$.

Como ficaria o exemplo acima da forma "correta"?

Exemplo 21 - Cap4_ex20.c

```
1  /*
2   * Cap4_ex20.c
3   * gcc  Cap4_ex20.c -o  Cap4_ex20 -Wall -g -lm
4   *
5   */
6  #include <stdio.h>
7  /* observe que estamos incluindo a math.h e precisamos acrescentar
8   * -lm como parâmetro de compilação!
9   */
10 #include <math.h>
11
12 int main(void){                                                    main
13         double a = 11.0;
14         double b;
15         double epsilon = 0.00001;
16
17         b = 1.1 * 10.0;
18
19         /*
```

```
20          * a função fabs recebe uma variável do tipo double e retorna
21          * o módulo dessa variável.
22          */
23         if (fabs(a−b) > epsilon){
24                 printf("a e b saum diferentes!\n");
25         }
26         else{
27                 printf("a e b saum iguais!\n");
28         }
29
30         /*
31          * se o leitor não quiser usar a função fabs, pode ser feito
32          * dessa forma também.
33          */
34
35         if (  ( (a > b) && (a − b > epsilon) ) ||
36                   ( (a < b) && (b − a > epsilon) ) ){
37
38                 printf("a e b saum diferentes!\n");
39         }
40         else{
41                 printf("a e b saum iguais!\n");
42         }
43
44         return 0;
45 }
```

Esse conceito pode ser extendido para as outras estruturas de seleção e de iteração, com exceção do `switch`, que aceita apenas valores inteiros.

O operador `!=` sofre do mesmo problema do `==` e a solução é análoga. Os outros operadores relacionais como `>`, `<`, `>=`, `<=` embora sejam relativamente mais "seguros", o programador pode optar por utilizar esse mesmo conceito para definir até que ponto um número em ponto-flutuante pode ser considerado maior ou menor do que outro.

O importante é o programador estar ciente do problema. A solução adotada deve ser a que melhor reflete o tipo de programa que se deseja obter. O exemplo abaixo ilustra uma possibilidade para os operadores relacionais `>` e `<`. Note que dessa vez não se utiliza o módulo.

Exemplo 22 - Cap4_ex21.c

```
1  /*
2   * Cap4_ex21.c
3   * gcc  Cap4_ex21.c -o Cap4_ex21 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8
9  int main(void){                                                    main
10         double a;
11         double b;
12         double eps;
13
14         printf("Informe a: ");
15         scanf("%lf",&a);
16         printf("\nInforme b: ");
17         scanf("%lf",&b);
18         printf("\nInforme epsilon: ");
19         scanf("%lf",&eps);
```

```
20
21          /*
22           * Vamos supor que nós consideramos que a > b,
23           * se a diferença entre eles for maior do que epsilon
24           */
25          if ( a − b > eps){
26                  printf("a eh maior que b!\n");
27          }
28          /*
29           * Agora, vamos considerar que a < b, se a diferença
30           * entre eles for menor que -epsilong
31           */
32           if (a − b < −eps){
33                  printf("a eh menor que b!\n");
34           }
35
36          return 0;
37  }
```

Leitura Adicional

Naturalmente, existem outras abordagens, por exemplo: Ao invés de considerar um ϵ, que representa a diferença máxima, pode-se obtar por utilizar um erro relativo máximo.

Essa e outras abordagens estão exemplificadas em `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`. Além disso, o artigo encontrado em `http://hal.archives-ouvertes.fr/hal-00128124` oferece uma valiosa fonte de comportamentos não intuitivos encontrados no uso de números em ponto-flutuante.

4.4 Pequena introdução às funções

Um capítulo posterior será dedicado ao estudo de funções. No entanto, como elas efetuam um papel central na linguagem C, é interessante que o leitor já se habitue a utilizá-las e a criá-las.

4.4.1 Declaração de funções

O formato das declaração e definições[6] das funções é o seguinte:

tipo_de_retorno **nome_da_função**(*tipo* arg1, *tipo*, arg2, . . . , *tipo* argn){
. . .
. . .
comandos
}

- Se a função não receber argumentos, ao invés do nome e dos tipos do argumentos, coloca-se a palavra-chave *void*. A mesma coisa vale para se a função não possuir retorno. Ao invés do tipo de retorno, deve-se utilizar *void*.

[6] Na verdade, existe uma diferença entre *declarar* uma função ou uma variável e *definir* uma variável ou função. Em linhas gerais, o primeiro apenas indica que o elemento *existe* e outro *cria* o elemento propriamente dito. Esse tópico será discutido numa etapa posterior. Por enquanto, todas as declarações também serão definições.

- Dentro de uma função pode-se declarar variáveis de maneira análoga à função *main*.

Um exemplo simples, primeiro:

Exemplo 23 - Cap4_ex22.c

```
1  /*
2   * Cap4_ex22.c
3   * gcc  Cap4_ex22.c -o Cap4_ex22 -Wall -g
4   *
5   */
6  #include <stdio.h>
7  /*
8   * Essa função retorna um int. E note que o comando return, retorna
9   * a + b, que é uma expressão do tipo int.
10  */
11 int soma(int a, int b){                                          soma
12
13        return a + b;
14 }
15
16 int main(void){                                                  main
17        int num1,num2;
18        int resultado;
19
20        printf("Informe o primeiro numero: ");
21        scanf("%d",&num1);
22        printf("Informe o segundo numero: ");
23        scanf("%d",&num2);
24
25        resultado = soma(num1,num2);
26
27        printf("%d + %d = %d",num1,num2,resultado);
28
29        return 0;
30 }
```

Antes da função *main* foi criada uma pequena função que recebe dois números e retorna a soma deles. Para indicar o retorno da função basta utilizar o comando *return*. Isso é diferente de indicar o *tipo* de retorno. Identifica-se o tipo de retorno antes do nome da função.

Além disso, a função pode ter mais de um comando *return*. Sempre que um *return* é executado, a execução do programa retorna ao ponto seguinte à chamada da função. Exemplo:

Exemplo 24 - Cap4_ex23.c

```
1
2  /*
3   * Cap4_ex23.c
4   * gcc  Cap4_ex23.c -o Cap4_ex23 -Wall -g
5   *
6   */
7  #include <stdio.h>
8
9  /*
10  * Se a for maior do que b, retorna a. Se b > a
11  * retorna b. O terceiro return só será executado se
12  * a não for nem maior nem menor do que b. Ou seja, só será
13  * executado se os dois forem iguais.
```

```
14  */
15 int maior(int a, int b){                                              maior
16         if (a > b){
17                 return a;
18         }
19         else if (b > a){
20                 return b;
21         }
22         return 0;
23 }
24
25
26
27 int main(void){                                                        main
28         int num1,num2;
29
30
31         printf("Informe o primeiro numero: ");
32         scanf("%d",&num1);
33         printf("Informe o segundo numero: ");
34         scanf("%d",&num2);
35
36         printf("O maior numero eh: %d",maior(num1,num2));
37
38         return 0;
39 }
```

Assim que um dos *return*s é encontrado, nenhum comando adicional da função é executado. Agora um exemplo de uma função que não recebe argumentos:

Exemplo 25 - Cap4_ex24.c

```
1  /*
2   * Cap4_ex24.c
3   * gcc Cap4_ex24.c -o Cap4_ex24 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  void imprimir_oi(void){                                              imprimir_oi
9          printf("Oi!\n");
10 }
11
12 int main(void){                                                       main
13         imprimir_oi();
14
15         return 0;
16 }
```

Apesar de trivial, o exemplo acima esconde um sutileza que é uma fonte de erros. Às vezes, ao chamar uma função que não recebe argumentos, esquece-se de colocar () após o nome da função. Note que isso não é um erro, conforme veremos quando estudarmos ponteiros para a função, mas isso não equivale a chamar a função. Para chamar a função é ***obrigatório*** colocar os ().

O leitor deve ter notado quen os exemplos acima, as funções foram escritas antes da função main. Elas poderiam ter sido escritas *depois*? A resposta é *sim*. Mas para fazer isso, deríamos que *declarar* a função primeiro antes de efetuar a *definição*. Por alto, declarar uma função ou variável significa indicar que a

variável *existe* e definir uma função ou variável implica em criar a variável ou função.

Para usar uma função em um programa é preciso que esta função esteja declarada previamente. Novamente, note que isso não significa que ela precisa estar *definida*. No momento, o leitor só viu declarações que *também* são definições. Em um capítulo posterior discutiremos os *protótipos de função* que servem ao propósito de *declarar* sem *definir*.

Observe o exemplo:

Exemplo 26 - Cap4_ex25.c

```
1  /*
2   * Cap4_ex25.c
3   * gcc  Cap4_ex25.c -o Cap4_ex25 -Wall -g
4   *
5   */
6  #include <stdio.h>
7  /*
8   * a função fun1 não pode chamar nem fun2 nem fun3, pois elas não
9   * estão declaradas antes.
10  */
11 void fun1(void){                                                    fun1
12         printf("Eu sou a funcao 1!\n");
13 }
14 /*
15  * a função fun2 pode chamar a função fun1, pois ela está declarada
16  * antes.
17  */
18 void fun2(void){                                                    fun2
19         printf("Eu sou a funcao 2!\n");
20 }
21 /*
22  * a função fun3 pode chamar a função fun1 e fun2.
23  */
24 void fun3(void){                                                    fun3
25         printf("Eu sou a funcao 3!\n");
26 }
27 /*
28  *  a função main pode chamar fun1,fun2,fun3, mas não pode chamar
29  *  fun4, já que ela está declarada depois.
30  */
31 int main(void){                                                     main
32         fun1();
33         fun2();
34         fun3();
35
36         return 0;
37 }
38
39 void fun4(void){                                                    fun4
40         printf("A main naum pode me chamar =’(\n");
41 }
```

O leitor é convidado a mexer nas funções e tentar acrescentar comandos para chamar outras funções para verificar a questão da declaração de funções. Uma forma de pensar é que uma função só tem acesso a funções e variáveis declaradas "acima" dela.

4.4.2 Escopo

Escopo se refere à quando no programa pode-se utilizar determinada função ou variável. Já discutimos a questão do escopo relativo à "quem pode chamar quem". Mas existem alguns outros pequenos detalhes que o autor deve chamar atenção.

Variáveis definidas dentro de uma função só podem ser acessadas dentro da função em questão. Vamos supor que uma função `fun1` declare uma variável: `int a;`. Outras funções como a *main*, não possuem acesso à essa variável. O que significa não possuir acesso? Nâo é possível utilizar a variável em expressões, atribuir valores, ou seja, é como se as outras funções *não enxergassem a variávei.*

Se o programar declarar 3 funções distintas e em cada uma, declarar uma variável chamada 'a', as três variáveis serão distintas e não possuirão nenhum tipo de conexão. Isso é interessante porque permite que nomes comuns como "i", "j" sejam utilizados por diferentes funções sem problema algum. Observe:

Exemplo 27 - Cap4_ex26.c

```
1  /*
2   * Cap4_ex26.c
3   * gcc Cap4_ex26.c -o Cap4_ex26 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  /*Cada função possui uma variável chamada a!*/
9
10 void fun1(void){                                              fun1
11         double a;
12
13         a = 3.0;
14         printf("O a dentro da fun1 eh %g\n",a);
15 }
16
17 void fun2(void){                                              fun2
18         double a;
19
20         a = 2.0;
21         printf("O a dentro da fun2 eh %g\n",a);
22 }
23
24 int main(void){                                               main
25         double a;
26
27         a = 1.0;
28         printf("O a dentro da main eh: %g\n",a);
29         fun1();
30         fun2();
31
32         return 0;
33 }
```

Se o programador quiser compartilhar variáveis entre diferentes funções ele deve utilizar variáveis globais. Não discutiremos variáveis globais até chegar no capítulo de funções, portanto tenha paciência!

4.4.3 Um pouco mais sobre passagem de parâmetros

Pode parecer curioso deixar para discutir a passagem de parâmetros depois de mostrar como se define as funções. A questão é que a maioria dos leitores já possuem alguma idéia do que acontece na passagem de parâmetros. Toda linguagem decente possui funções, procedimentos e para linguagens orientadas a objetos, métodos.

No C, a passagem de parâmetros é sempre por *valor*. Isto é, os parâmetros da função recebem **cópias dos valores** passados para a função. E o que isso significa? Variáveis passadas como argumento de uma função e que são modificadas *dentro* da função não alteram o valor da variável fora da função. Observe o exemplo:

Exemplo 28 - Cap4_ex27.c

```
1  /*
2   * Cap4_ex27.c
3   * gcc  Cap4_ex27.c -o  Cap4_ex27 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  void fun1(int a){                                                    fun1
9          printf("O a dentro da funcao fun1 eh %d\n",a);
10         a = 15;
11         printf("O a dentro da funcao fun1 eh %d\n",a);
12 }
13
14 int main(void){                                                     main
15         int a = 2;
16
17         printf("Antes de chamar a funcao, a = %d\n",a);
18
19         fun1(a);
20
21         printf("Apos chamar a funcao, a = %d\n", a);
22
23         return 0;
24 }
```

A variável `a` na função fun1 e a variável `a` na função main são completamente distintas e ocupam espaços de memória diferentes. Quando a variável `a` dentro da main é passada com argumento para a função fun1, copia-se o valor de `a` para a outra variável chamada `a` dentro de fun1. Como as duas variáveis, ocupam espaços de memória distintos, a alteração em uma não é refletida na outra.

O leitor talvez esteja familiarizado com a passagem de parâmetros por referência. Nessa modalidade, o parâmetro da função e o argumento passado a função possuem o mesmo endereço de memória. Ao efetuar a passagem de parâmetros cria-se uma *referência* ao argumento passado para função. Essa referência geralmente é o endereço de memória do argumento passado para a função. Dessa forma, se o parâmetro for modificado dentro da função, essa mudança se refletirá na variável que foi passada como argumento à função[7].

[7]No Pascal, utiliza-se a palavra *var* para indicar que determinado parâmetro deve ser passado como referência. No C++, há a opção de se utilizar referências também, basta colocar o & antes do nome do parâmetro da função.

O C não possui um mecanismo de passagem de parâmetros por referência, porém é possível simular esse tipo de passagem de parâmetros através do uso de *ponteiros*. Como esse assunto é mais delicado, trataremos dele numa etapa posterior.

Por enquanto, é importante não se esquecer que no C as passagens de parâmetros são feitas por *valor* e não por *referência*.

Capítulo 5

Vetores e Strings

Capítulo 6

Outros tipos de dados Compostos

Capítulo 7

Funções

Capítulo 8

Arquivos

Parte II

Intermediário

8.1 Comentários do autor

Bem-vindo, caro leitor, à parte intermediária desse texto. Nesse ponto, o leitor provavelmente já possui ferramentas para realizar um grande número de tarefas na linguagem C.

Nessa parte, começaremos a tratar de temas um pouco mais espinhosos, como ponteiros, alocação dinâmica de memória e discutiremos algumas idéias para implementar estruturas de dados. Além disso, o autor dedicará um capítulo para discutir um pouco sobre modularização e divisão do programa em arquivos-fontes distintos.

Como é inevitável, abordaremos alguns temas que são controversos no C: como o uso processador e a ausência de garbage collecton para a gerência de memória.

Os temas vão ficar mais densos, sendo assim é necessário que o leitor tenha compreendido bem os temas abordados nos capítulos passados, na parte Básica. Antes de prosseguir, o autor convida o leitor a reler e praticar os pontos em que ainda haja alguma insegurança.

Sem mais ressalvas, daremos início à parte Intermediária.

Capítulo 9

Ponteiros - Primeira Parte

Ponteiros são variáveis que guardam posições de memórias de outras variáveis ou funções. É simplesmente isso. Mas, essa aparente simplicidade é a causa de muitas frustrações de diversos programadores.

Não é possível programar de forma séria em C sem lidar com ponteiros, no entanto a manipulação de ponteiros nem sempre é uma tarefa trivial. Ponteiros adicionam uma flexibilidade muito grande, mas para o programador iniciante, pode ser uma fonte de erros difíceis de serem detectados.

É por isso que é importante treinar o uso de ponteiros. Assim que se pega "o jeito da coisa", a manipulação passa a ser razoavelmente intuitiva e o número de erros cometidos diminui.

9.1 Declaração e atribuição de ponteiros

Na verdade, não existe uma única variável do tipo ponteiro, existem vários tipos diferentes de ponteiros. O formato geral da declaração de um ponteiro é:

*tipo_de_dado * nome_do_ponteiro;*

Com isso, estamos declarando um variável do tipo *ponteiro* para *tipo_de_dado*. Naturalmente, *tipo_de_dado* pode ser *qualquer* tipo de dado primitivo ou criado pelo programador através de *structs*, por exemplo.

Observe o exemplo abaixo:

Exemplo 1 - Cap9_ex1.c

```
1  /*
2   * Cap9_ex1.c
3   * gcc Cap9_ex1.c -o Cap9_ex1 -Wall -g
4   *
5   */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10         /*Declaração de um ponteiro para uma variável do tipo int*/
11         int * int_ptr;
12         /*Declaração de um ponteiro para uma variável do tipo float*/
13         float * float_ptr;
14
15         /*
```

```
16          * Observe que o endereço apontado por ponteiros não inicializadas
17          * é razoavelmente aleatório. Novamente, variáveis em C não são
18          * inicializadas por padrão!
19          */
20         printf("int_ptr aponta para o seguinte endereco: %p\n",int_ptr);
21         printf("float_ptr aponta para o seguinte endereco: %p\n",float_ptr);
22
23
24         /* Atribui-se ao ponteiro o valor de NULL, para indicar que ele não
25          * aponta para nenhum endereço de memória.
26          */
27         int_ptr = NULL;
28         float_ptr = NULL;
29
30         printf("int_ptr aponta para o seguinte endereco: %p\n",int_ptr);
31         printf("float_ptr aponta para o seguinte endereco: %p\n",float_ptr);
32
33         return 0;
34 }
```

Foram declarados uma variável do tipo ponteiro para `int` e outra variável do tipo ponteiro para `float`. Essas variáveis podem guardar endereços de outras variáveis do tipo `int` e `float`, respectivamente.

Uma questão importantíssima que esse exemplo mostra é a seguinte: ***Ponteiros devem ser inicializados antes do uso, isto é, preferencialmente devem ser atribuídos o valor NULL antes de utilizá-lo!*** A macro *NULL* é definida no arquivo cabeçalho *stddef.h* e no arquivo *stdio.h*. Incluir um ou outro define a macro NULL para o mesmo valor. A razão para se inicializar ponteiros é simples: Como as variáveis não são inicializadas por padrão, se a inicialização não for feita, há um risco do programador utilizar esse ponteiro de forma indevida e acabar acessando um endereço de memória que não pertence ao programa, ocasionando um tipo de erro razoavelmente difícil de detectar.

Ponteiros que guardam endereços inválidos ou de variáveis que não sejam do tipo correto, são chamados de *wild pointers* e devem ser evitados de toda forma! Como o leitor deve ter visto ao compilar o programa, o primeiro par de `printf` imprime endereços que provavelmente são inválidos ou não devem ser acessados pelo programa.

Bom, não falemos mais de tragédias! Abaixo, temos um exemplo de como atribuir um endereço à um ponteiro.

Exemplo 2 - Cap9_ex2.c

```
1  /*
2   * Cap9_ex2.c
3   * gcc Cap9_ex2.c -o Cap9_ex2 -Wall -g
4   *
5   */
6
7  #include <stdio.h>
8
9  int main(void){                                                    main
10         /*Declaração de um ponteiro para inteiro*/
11         int * int_ptr = NULL;
12         /*Declaração de um ponteiro para float*/
13         float * float_ptr = NULL;
14
15         int a;
16         float b;
```

```
17
18          /*Lê do teclado um int*/
19          printf("Informe o valor de a: ");
20          scanf("%d",&a);
21          printf("\n");
22          /*Lê do teclado um float*/
23          printf("Informe o valor de b: ");
24          scanf("%f",&b);
25          printf("\n");
26
27          printf("Endereco de int_ptr antes: %p\n",int_ptr);
28          printf("Endereco de float_ptr antes: %p\n\n",float_ptr);
29
30          /*
31           * Observe o uso do operador &, quando aplicado a uma única variável,
32           * ele retorna o endereço na memória dessa variável.
33           *
34           * Dessa forma, como queremos atribuir a um ponteiro um endereço,
35           * é necessário que utilizemos o operador &
36           */
37          int_ptr = &a;
38          float_ptr = &b;
39
40          /*
41           * Naturalmente o endereço da variável e do ponteiro após a atribuição
42           * são iguais!
43           */
44          printf("Endereco de int_ptr depois: %p\n",int_ptr);
45          printf("Endereco de a: %p\n\n",&a);
46
47          printf("Endereco de float_ptr depois: %p\n",float_ptr);
48          printf("Endereco de b: %p\n",&b);
49
50          return 0;
51  }
```

Não importa o tipo de ponteiro, sempre é possível utilizar o operador `&` para atribuir o endereço à um ponteiro. Note, que mantivemos a coerência: À um ponteiro para `int`, atribuímos o endereço de uma variável do tipo `int` e à um ponteiro para `float`, atribuímos o endereço de uma variável do tipo `float`.

Assim como uma variável `int` guarda um valor inteiro, um ponteiro guarda um endereço de memória. Naturalmente, como um ponteiro também é uma variável, o ponteiro também possui um endereço de memória! Essa constatação também é uma grande fonte de confusão, mas o autor espera que fique mais claro com o exemplo abaixo:

Exemplo 3 - Cap9_ex3.c

```
1   /*
2    * Cap9_ex3.c
3    * gcc Cap9_ex3.c -o Cap9_ex3 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           int * int_ptr = NULL;
10          int a = 2;
11
12
```

```
13        /*Atribuímos o endereço de a ao ponteiro int_ptr...*/
14        int_ptr = &a;
15
16
17        /*Os dois printfs abaixo vão mostra o mesmo endereço*/
18        printf("int_ptr aponta para %p\n",int_ptr);
19        printf("Endereço de a: %p\n\n",&a);
20        /*Já o terceiro será diferente...
21         *O que é mostrado na tela é o endereço do ponteiro.
22         *Pode-se pensar que neste endereço é onde "reside" o conteúdo
23         *da variável que no caso do ponteiro também é um endereço!
24         */
25        printf("Endereço de int_ptr: %p\n",&int_ptr);
26        return 0;
27  }
```

Os dois primeiros printfs imprimem exatamente o mesmo endereço de memória na tela. Porém, o terceiro revela um conteúdo diferente. Observe que colocamos para imprimir a expressão `&int_ptr`, então o operador `&` se encarrega de tomar o endereço da variável e é isso que é passado para o printf.

9.2 Derreferenciação

Aos poucos, a coisa vai ficando mais interessante. Conforme o leitor viu na seção anterior, pode-se atribuir um endereço à um ponteiro. Mas e agora? O que fazer com isso?

Se o ponteiro guarda o endereço de *uma outra variável*, então não seria possível acessar o *valor* dessa variável através do seu endereço? A resposta, meu caríssimo leitor, é *sim*. É possível efetuar essa operação através de um processo chamado *derreferenciação*.

Qual o motivo do nome complexo? Bom... Se abstrairmos um pouco e pensarmos que uma variável é na verdade uma *referência* à um endereço de memória e que nesse endereço está localizado o valor da variável, o processo *derreferência* seria a obtenção do *valor* da variável em questão, a partir do seu *endereço*.

Como um ponteiro armazena um *endereço*, podemos obter o conteúdo daquilo que está armazenado nesse endereço através desse processo. Em C, o operador de derreferenciação é o `*`. E seu uso está ilustrado no próximo exemplo:

Exemplo 4 - Cap9_ex4.c

```
1  /*
2   * Cap9_ex4.c
3   * gcc Cap9_ex4.c -o Cap9_ex4 -Wall -g
4   *
5   */
6  #include <stdio.h>
7
8  int main(void){                                                    main
9         int a = 1, b = 2;
10        int * int_ptr = NULL;
11
12        /*Atribui-se o endereço de a ao ponteiro*/
13        int_ptr = &a;
14
15        /*Observe que será impresso o endereço da variável a*/
```

```
16          printf("O endereco armazenado eh: %p\n",int_ptr);
17          /*
18           * Através do operador *, obtemos o valor correspodente a esse endereço.
19           * Observe também, que *int_ptr = 2! Isto é, exatamente o mesmo
20           * valor de a
21           */
22          printf("O valor armazenado nesse endereco eh: %d\n",*int_ptr);
23
24
25          /*Repetimos agora esse mesmo processo para a variável b*/
26          int_ptr = &b;
27          printf("O endereco armazenado eh: %p\n",int_ptr);
28          /*Agora, o valor impresso correspodente ao valor da variável b*/
29          printf("O valor armazenado nesse endereco eh: %d\n",*int_ptr);
30
31          return 0;
32  }
```

Isso nos leva a uma conclusão interessante e a nossa primeira aplicação de ponteiros: não seria possível, então, modificar o conteúdo de uma variável através de um ponteiro que aponta para ela? Certamente. E é dessa forma que podemos simular a passagem de parâmetros por referência no C.[1] Observe o exemplo abaixo:

Exemplo 5 - Cap9_ex5.c

```
1   /*
2    * Cap9_ex5.c
3    * gcc Cap9_ex5.c -o Cap9_ex5 -Wall -g
4    *
5    */
6   #include <stdio.h>
7
8   int main(void){                                                    main
9           char c = 'a';
10          char * ptr = NULL;
11
12          printf("%c\n",c);
13          /*Atribuímos o endereço de c ao ponteiro ptr*/
14          ptr = &c;
15          /*
16           * Modificamos o conteúdo do endereço apontado por ptr. Ou seja,
17           * modificamos o conteúdo de c através de ptr.
18           */
19          *ptr = 'b';
20          /*Milagrosamente, o valor de c de fato é modificado*/
21          printf("%c\n",c);
22
23          return 0;
24  }
```

[1] Alguns autores optam por fazer a distinção entre a passagem de parâmetros por referência e a passagem de parâmetros por ponteiro, que é o caso do C.

9.3 Ponteiros e Funções

Como já discutimos no capítulo sobre funções, a passagem de parâmetros é feita por *valor*, isto é, a função recebe uma cópia dos parâmetros e mudanças nessas cópias *não* alteram o conteúdo das variáveis passadas às funções.

Suponha, por exemplo, que nós quiséssemos criar uma pequena função que troque o valor dos parâmetros passados à função e que essa mudança se refletisse de fato nos parâmetros passados. Uma forma de fazer tal função seria assim:

Exemplo 6 - Cap9_ex6.c

```
1  /*
2   * Cap9_ex6.c
3   * gcc Cap9_ex6.c -o Cap9_ex6 -Wall -g
4   *
5   */
6  #include <stdio.h>
7  /*
8   * Observe que a função recebe ponteiros. Nesse caso, podemos tanto
9   * passar ponteiros do tipo correspondente como endereços de variáveis
10  * do tipo correspodente.
11  */
12 void swap(double * a, double * b){                                    swap
13         double aux;
14         aux = *a;
15         *a = *b;
16         *b = aux;
17 }
18
19 int main(void){                                                       main
20         double i,j;
21
22         i = 2.5;
23         j = 4.5;
24         printf("i = %f\nj = %f\n",i,j);
25         /*Note que estamos passando o endereço das variáveis*/
26         swap(&i,&j);
27         printf("Após swap. . .\n");
28         printf("i = %f\nj = %f\n",i,j);
29
30         return 0;
31 }
```

Observe a assinatura da função `swap`. Note que há a presença de dois ponteiros para `double`. Portanto podemos chamar a função passando tanto ponteiros para `double` quanto um endereço de um `double`. Isso soa estranho? Pense da seguinte forma: suponha que uma função tenha a seguinte assinatura `void foo(int a)`, de quais formas possíveis podemos chamar essa função? Podemos chamá-la passando uma variável inteira(`foo(i)`) por exemplo), com um valor numérico ou expressão(`foo(3-1)`, por exemplo). A idéia, nos dois casos, é exatamente a mesma.

O mesmo exemplo poderia ter sido escrito com um pouco mais de verbosidade da seguinte forma:

Exemplo 7 - Cap9_ex7.c

```
1  /*
2   * Cap9_ex7.c
3   * gcc Cap9_ex7.c -o Cap9_ex7 -Wall -g
```

```
4   *
5   */
6  #include <stdio.h>
7  /*
8   * A mesma função do exemplo anterior...
9   */
10 void swap(double * a, double * b){                                    swap
11         double aux;
12         aux = *a;
13         *a = *b;
14         *b = aux;
15 }
16
17 int main(void){                                                       main
18         double i,j;
19         double * ptr1 = NULL;
20         double * ptr2 = NULL;
21
22         i = 2.5;
23         j = 4.5;
24         printf("i = %f\nj = %f\n",i,j);
25
26         ptr1 = &i;
27         ptr2 = &j;
28         /*
29          * Note que agora não usamos o operador & para passar os parâmetros.
30          * Mas isso é natural, não é mesmo? Já que a função realmente
31          * recebe ponteiros para double!
32          */
33         swap(ptr1,ptr2);
34         printf("Após swap...\n");
35         printf("i = %f\nj = %f\n",i,j);
36
37         return 0;
38 }
39
```

Aqui cabe uma pequena discussão. Note então, que somos capazes de simular passagem de parâmetros por referência através do uso de ponteiros. Mas note que isso continua sendo apenas uma simulação! Existem linguagens que realmente possuem referências, como o Pascal, o C++ e o Java(no caso de objetos). Mas o leitor atento perceberá ou já percebeu que na verdade continuamos utilizando a passagem de parâmetros por *valor*, mas como esse "valor" é um endereço, temos acesso de à variáveis fora do escopo da função e não às cópias delas.

No segundo exemplo, os ponteiros `ptr1` e `ptr2` são passados à função, mas o que a função recebe de fato são cópias desses ponteiros. Mas tanto o ponteiro original quanto a cópia apontam para o mesmo endereço, portanto conseguimos modificar a variável "apontada" na função.

E é essa a razão pela qual alguns preferem dizer que o C possui além da passagem por valor, passagem de parâmetros por *ponteiro*, que nada mais é do que a passagem por valor camuflada. Se você, meu caríssimo leitor, ainda não pegou a idéia da coisa, experimente e modifique os exemplos. Verifique, no exemplo anterior, o que acontece quando imprime-se na tela os endereços dos ponteiros dentro da função *main* e dentro da função *swap*.

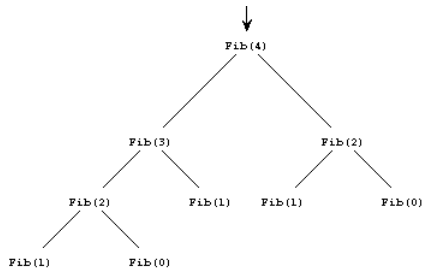

Figura 9.1: Árvore de recursão de Fib(4)

9.4 A promíscua relação entre ponteiros e vetores

Vamos agora, para um exemplo mais interessante. Quando discutimos recursão no capítulo sobre funções, vimos uma versão de um algoritmo recursivo para calcular a série de Fibonacci. Infelizmente, ele é extremamente ineficiente se comparado à versão iterativa. Mas isso acontece porque o mesmo valor é calculado várias vezes! Se o leitor fizer a análise da árvore de recursão observará esse fato. Como então escrever uma versão recursiva de Fibonacci que aproxime em desempenho da versão iterativa? Observe o exemplo:

Exemplo 8 - Cap9_fib.c

```
1  /*
2   * Cap9_fib.c
3   * gcc Cap9_fib.c -o Cap9_fib -Wall -g
4   *
5   */
6
7  #include <stdio.h>
8
9  void fib_aux(int n, int aux[]){                                    fib_aux
10         if (n < 2){
11                 aux[n] = n;
12                 return;
13         }
14
15         if (aux[n-1] == 0){
16                 fib_aux(n-1,aux);
17         }
18         if (aux[n-2] == 0){
19                 fib_aux(n-2,aux);
20         }
21         aux[n] = aux[n-1] + aux[n-2];
22 }
23
24 void fib(int n, int aux[]){                                        fib
25
26         fib_aux(n,aux);
27 }
28
29 int main(void){                                                    main
30         int i;
31         int aux[100] = {0};
```

```
32
33          fib(45,aux);
34
35          for (i = 0; i < 46; i++){
36                  printf("%d = %d\n",i,aux[i]);
37          }
38
39          return 0;
40  }
41
```